реальные проекты на Python

Data Science в действии

Леонард Апельцин

MANNING

Data Science Bookcamp

FIVE PYTHON PROJECTS

LEONARD APELTSIN

MANNING
SHELTER ISLAND

Data Science в действии

Пять реальных проектов Python

Леонард Апельцин

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

ББК 32.973.233.02
УДК 004.65
А76

Апельцин Леонард

А76 Data Science в действии. — СПб.: Питер, 2023. — 736 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1982-0

В проектах обработки и анализа данных много движущихся частей, и требуются практика и знания, чтобы создать гармоничную комбинацию кода, алгоритмов, наборов данных, форматов и визуальных представлений. Эта уникальная книга содержит описание пяти практических проектов, включая отслеживание вспышек заболеваний по заголовкам новостей, анализ социальных сетей и поиск закономерностей в данных о переходах по рекламным объявлениям.

Автор не ограничивается поверхностным обсуждением теории и искусственными примерами. Исследуя представленные проекты, вы узнаете, как устранять распространенные проблемы, такие как отсутствующие и искаженные данные и алгоритмы, не соответствующие создаваемой модели. По достоинству оцените подробные инструкции по настройке и детальные обсуждения решений, в которых описываются типичные точки отказа, и обретите уверенность в своих навыках.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.233.02
УДК 004.65

ISBN 9781617296253 англ.
ISBN 978-5-4461-1982-0

Краткое содержание

Практическое задание 3. Отслеживание вспышек заболеваний по новостным заголовкам

Практическое задание 4. Улучшение своего резюме аналитика данных на основе онлайн-вакансий

Практическое задание 5. Прогнозирование будущих знакомств на основе данных социальной сети

Оглавление

Практическое задание 2. Анализ значимости переходов по онлайн-объявлениям

Практическое задание 3. Отслеживание вспышек заболеваний по новостным заголовкам

Практическое задание 4. Улучшение своего резюме аналитика данных на основе онлайн-вакансий

Практическое задание 5. Прогнозирование будущих знакомств на основе данных социальной сети

Посвящается моему наставнику Александру Вишневскому, научившему меня рассуждать.

Предисловие

Еще один перспективный кандидат провалил свое интервью по data science, и мне стало интересно почему. Это был 2018 год, и передо мной стояла задача расширить команду по обработке данных своего стартапа. Я беседовал с десятками, казалось бы, квалифицированных соискателей, но в итоге им всем приходилось отказывать. Последним таким кандидатом был доктор экономических наук из престижного учебного заведения. Он лишь недавно перешел в сферу data science, пройдя десятинедельное обучение. Я попросил его подумать над актуальной для нашей компании задачей по аналитике, на что он тут же представил модный алгоритм, который для обозначенной ситуации явно не годился. Когда же я начал оспаривать пригодность его решения, кандидат растерялся. Он даже не знал, как именно этот алгоритм работает или в каких конкретно обстоятельствах должен использоваться. На пройденном учебном курсе им об этом не рассказали.

Когда соискатель ушел, я задумался о том, как сам когда-то получал образование в области data science. Насколько же тогда все было иначе! В 2006 году эта сфера еще не являлась столь вожделенным карьерным выбором, а учебных курсов по этому направлению просто не существовало. В те дни я был бедным студентом выпускного курса, с трудом оплачивавшим аренду жилья в дорогом Сан-Франциско. В качестве исследования для дипломной работы мне нужно было провести анализ миллионов генетических связей с заболеваниями. Тогда я осознал, что мои навыки вполне переносимы и на другие сферы анализа, — с этого и началась моя консалтинговая деятельность в области обработки данных.

Не ставя в известность своего научного руководителя, я начал предлагать услуги по аналитике разным компаниям в районе залива Сан-Франциско. Эта подработка

позволяла платить по счетам, так что особо выбирать среди поручаемых заданий не приходилось. В результате я подписался работать с различными проектами по анализу данных — от простого статистического анализа до сложного предиктивного моделирования. Иногда я сталкивался, казалось бы, с неразрешимыми задачами, но в итоге мое упорство брало верх. Благодаря своему усердию я освоил всяческие нюансы разнообразных техник аналитики и научился наилучшим образом их комбинировать для получения элегантных решений. Но более важно то, что я узнал, каким образом привычные приемы могут давать ошибки и как их преодолевать для достижения продуктивных результатов. В итоге я стал ведущим специалистом в этой области.

Достиг бы я того же уровня успеха, если бы просто штудировал материалы на десятинедельном учебном курсе? Вряд ли. Многие такие курсы делают упор на изучении отдельных алгоритмов, слабо развивая комплексные навыки решения задач. Более того, шумиха вокруг мощи и потенциала алгоритмов лишь затмевает их недостатки. В результате студенты иногда оказываются плохо подготовленными к решению реальных задач по обработке данных. Такое положение вещей и вдохновило меня на написание этой книги.

Я решил воспроизвести собственный путь обучения data science, представив вам, дорогой читатель, серию задач по аналитике с возрастающей сложностью. На этом пути я также снаряжу вас инструментами и техниками, необходимыми для их эффективного решения. Моя цель — всесторонне помочь вам выработать собственные навыки решения задач по аналитике. Так что по завершении этого пути, когда вы будете проходить собеседование на позицию инженера по данным, ваши шансы на ее получение окажутся существенно выше, чем до его начала.

Благодарности

Писать эту книгу было трудно, и в одиночку я бы точно не справился. К счастью, моя семья и друзья поддержали меня во время этого трудоемкого путешествия. В первую очередь хочу поблагодарить свою мать, Ирину Апельцин (Irina Apeltsin). Она подогревала мою мотивацию в тяжелые дни, когда стоявшие передо мной задачи казались неразрешимыми. Также благодарю мою бабушку, Веру Фишер (Vera Fisher), чьи дельные советы не давали мне сбиться с пути, пока я перелопачивал материал для своей книги.

Кроме того, выражаю признательность своему другу детства Вадиму Стольнику (Vadim Stolnik). Вадим — великолепный дизайнер графики, который помог мне оформить книгу бесчисленными иллюстрациями. Также благодарю своего друга и коллегу Эммануэля Йеру (Emmanuel Yera), который оказывал поддержку в самом начале проекта. Не могу не упомянуть и мою дорогую партнершу по танцам Александрию Лоу (Alexandria Law), которая не давала мне пасть духом в трудную минуту и помогла выбрать обложку для книги.

Выражаю свою благодарность и редактору издательства Manning Элеше Хайд (Elesha Hyde). На протяжении последних трех лет ты неустанно трудилась, стремясь сделать так, чтобы мой труд оказался полезным для читателей. Я буду вечно признателен за твои терпение, оптимизм и непреклонное стремление к качественному результату. Ты стимулировала меня к оттачиванию своих писательских навыков, и эти усилия в конечном итоге пойдут на пользу читателям. Хочу также поблагодарить технического редактора Артура Зубарова (Artur Zubarov) и технического корректора Рафаэллу Вентальо (Rafaella Ventaglio). Ваше содействие помогло мне сделать книгу более грамотной и чистой. Выражаю благодарность главному

редактору Дейрдре Хиам (Deirdre Hiam), выпускающему редактору Тиффани Тейлор (Tiffany Taylor), корректору Кэйти Теннант (Katie Tennant) и всем остальным сотрудникам Manning, приложившим руку к созданию этой книги.

Отдельная благодарность всем рецензентам — Адаму Шеллеру (Adam Scheller), Адриану Бейерцу (Adriaan Beiertz), Алану Богусевичу (Alan Bogusiewicz), Амарешу Раджасекхарану (Amaresh Rajasekharan), Айону Рою (Ayon Roy), Биллу Митчеллу (Bill Mitchell), Бобу Квинтусу (Bob Quintus), Дэвиду Джейкобсу (David Jacobs), Диего Каселле (Diego Casella), Дункану Макрею (Duncan McRae), Элиасу Рангелю (Elias Rangel), Франку Л. Кинтане (Frank L. Quintana), Гжегожу Бернасу (Grzegorz Bernas), Джейсону Хейлсу (Jason Hales), Жан-Франсуа Морену (Jean-François Morin), Джеффу Смиту (Jeff Smith), Джиму Амрхайну (Jim Amrhein), Джо Юстесену (Joe Justesen), Джону Касевичу (John Kasiewicz), Максиму Купферу (Maxim Kupfer), Майклу Джонсону (Michael Johnson), Михалу Амброзевичу (Michał Ambroziewicz), Раффаэлле Вентальо (Raffaella Ventaglio), Рави Саджнани (Ravi Sajnani), Роберту Диане (Robert Diana), Симоне Сгуацца (Simone Sguazza), Шрираму Мачарле (Sriram Macharla) и Стюарту Вудворду (Stuart Woodward) — ваши рекомендации помогли мне сделать эту книгу лучше.

О книге

Способность решать многовариантные задачи — ключевой фактор для карьеры аналитика данных. К сожалению, эту способность нельзя выработать в результате простого чтения. Для того чтобы стать эффективным в этом деле, необходимо постоянно практиковаться, решая трудные задачи. Учитывая это, я выстроил книгу вокруг практических заданий — многовариантных задач, смоделированных на базе реальных ситуаций из жизни. Эти исследования простираются от анализа онлайн-рекламы до отслеживания вспышек заболеваний на основе новых данных. По завершении этих проектов вы будете отлично подготовлены для того, чтобы начать карьеру в области обработки данных.

ДЛЯ КОГО ЭТА КНИГА

Эта книга задумана для новичков в сфере data science. Когда я представляю типичного читателя, то вижу студента выпускного курса кафедры экономики, желающего освоить более широкий спектр возможностей аналитики, или уже окончившего вуз химика, который хочет пойти по карьерному пути, больше ориентированному на обработку данных. Среди читателей вполне могут оказаться также успешные фронтенд-разработчики с очень ограниченными знаниями по математике, которые подумывают переключиться на работу с данными. Предполагается, что никто из моих потенциальных читателей не посещал ранее занятия по этой науке и не имеет опыта применения различных техник анализа данных. Цель книги как раз в том, чтобы компенсировать этот недостаток навыков.

При этом читателям необходимо знать азы программирования на Python. Умений, полученных в ходе самостоятельного изучения, должно быть достаточно для проработки примеров из книги. Математических знаний выше школьного курса тригонометрии не потребуется.

СТРУКТУРА ИЗДАНИЯ

Книга содержит пять практических заданий возрастающей сложности. Каждое из них начинается с подробного описания условий задачи, которую нужно будет решить. За условиями задачи следует от двух до пяти глав, рассказывающих о необходимых для их решения навыках и инструментах. Эти главы охватывают основные библиотеки, а также математические и алгоритмические техники. В заключительной главе исследования приводится решение поставленной задачи.

Первое практическое задание связано с базовой теорией вероятности.

- В главе 1 рассказывается, как вычислять вероятности с помощью простого кода Python.
- В главе 2 вводится понятие распределения вероятностей. Здесь же вы познакомитесь с библиотекой визуализации Matpotlib, которую можно использовать для визуального представления этих распределений.
- В главе 3 разбирается метод оценки вероятностей с помощью рандомизированных симуляций. Здесь для эффективного выполнения симуляции вводится библиотека численных вычислений NumPy.
- В главе 4 содержится решение этого практического задания.

Второе практическое задание расширяется из области вероятностей на область статистики.

- В главе 5 вы познакомитесь с простыми статистическими измерениями центральности и дисперсии. Здесь же вводится библиотека научных вычислений SciPy, которая содержит полезный модуль статистики.
- В главе 6 речь пойдет о центральной предельной теореме, которую можно использовать для составления статистических прогнозов.
- В главе 7 разбираются различные техники статистического вывода, с помощью которых можно отличить интересные паттерны данных от случайного шума. Кроме того, эта глава знакомит вас с опасностями неверного применения статистического вывода и способами избежать их.
- В главе 8 описывается библиотека Pandas, которую можно задействовать для предварительной обработки табличных данных перед статистическим анализом.
- В главе 9 приводится решение данного практического задания.

Третье практическое задание посвящено неуправляемой кластеризации географических данных.

- В главе 10 показано, как измерение центральности можно использовать для кластеризации данных по группам. Здесь также вводится библиотека scikit-learn, которая позволит выполнять кластеризацию более эффективно.
- В главе 11 разбирается тема извлечения и визуализации географических данных. Извлечение из текста выполняется с помощью библиотеки GeoNamesCache, а для визуализации применяется библиотека отрисовки карт Cartopy.
- В главе 12 приводится решение.

Четвертое практическое задание посвящается обработке естественного языка при помощи масштабных численных вычислений.

- В главе 13 показано, как эффективно вычислять сходство между текстами при помощи матричного умножения. Для этого активно используются встроенные в NumPy матричные оптимизации.
- В главе 14 демонстрируется применение уменьшения размерности для повышения эффективности матричного анализа. Здесь параллельно с объяснением методов уменьшения размерности, содержащихся в библиотеке scikit-learn, рассматривается математическая теория.
- В главе 15 техники обработки естественного языка применяются к очень большому текстовому набору данных. Здесь же речь пойдет о лучших способах изучения и кластеризации текстовых данных.
- В главе 16 демонстрируется, как извлекать текст из онлайн-данных с помощью библиотеки парсинга HTML Beautiful Soup.
- В главе 17 приводится решение.

Пятое практическое задание завершает книгу разбором теории сетей и машинного обучения (МО) с учителем.

- В главе 18 разъясняется базовая теория сетей, а также вводится библиотека NetworkX, используемая для анализа данных, представленных в виде графа.
- В главе 19 показано, как задействовать сетевой поток для обнаружения кластеров в сетевых данных. Для достижения эффективной кластеризации здесь применяются вероятностные симуляции и матричное умножение.
- В главе 20 вводится простой алгоритм машинного обучения с учителем, основанный на теории сетей. Попутно с этим демонстрируются распространенные приемы оценки эффективности моделей МО при помощи scikit-learn.
- В главе 21 разбираются дополнительные техники машинного обучения, опирающиеся на линейные классификаторы с эффективным использованием памяти.

- В главе 22 речь пойдет о слабых местах ранее представленных методов обучения с учителем. Эти недостатки устраняются с помощью нелинейных классификаторов, а именно деревьев решений.
- В главе 23 приводится решение этого практического задания.

Все главы строятся на основе алгоритмов и библиотек, представленных ранее. В связи с этим рекомендуется прочесть книгу от начала до конца, чтобы исключить возможное непонимание. Если же вы уже знакомы с некоторыми темами, то можете смело их пропускать. Ну и в завершение я настоятельно рекомендую самостоятельно решить каждую задачу, прежде чем смотреть в решение. Это позволит вам получить максимум пользы от книги.

О КОДЕ

Эта книга содержит множество примеров исходного кода как в пронумерованных листингах, так и в тексте. В обоих случаях код для наглядности выделяется моноширинным шрифтом. В листингах он структурирован по модульным фрагментам и содержит письменные пояснения, сопровождающие каждую модульную часть. Такой стиль представления кода отлично подходит для блокнотов Jupyter, поскольку они связывают функциональные примеры с текстовым описанием. В связи с этим исходный код каждого практического задания доступен для скачивания в блокнотах Jupyter по адресу www.manning.com/books/data-science-bookcamp. Эти блокноты объединяют листинги с обобщенным описанием из книги. Как обычно, в блокнотах между отдельными ячейками существуют взаимозависимости, поэтому рекомендуется выполнять примеры кода именно в том порядке, в котором они приводятся в блокноте, иначе вы рискуете допустить ошибки.

Об авторе

Леонард Апельцин (Leonard Apeltsin) является главой отдела обработки данных в Anomaly. Его команда использует продвинутые методы аналитики для выявления случаев мошенничества, растрат и злоупотреблений в сфере здравоохранения. До этого Леонард руководил проектами машинного обучения в Primer AI — стартапе, специализирующемся на обработке естественного языка. Будучи одним из основателей, он помог расширить команду Primer AI с четырех до почти ста сотрудников. Прежде чем начать развивать стартапы, Леонард работал в сфере науки, выявляя скрытые паттерны в заболеваниях, связанных с генетикой. Его открытия публиковались в приложениях к журналам *Science* и *Nature*. Леонард получил степени бакалавра по биологии и computer science в Университете Карнеги — Меллона, а также степень доктора наук в Калифорнийском университете в Сан-Франциско.

Иллюстрация на обложке

Рисунок на обложке называется *Habitante du Tyrol* — «Житель Тироля». Эта иллюстрация взята из коллекции рисунков, изображающих гардероб жителей различных стран, под названием *Costumes de Différents Pays*, собранной Жаком Грассе де Сен-Совером (Jacques Grasset de Saint-Sauveur) (1757–1810) и опубликованной им во Франции в 1797 году. Каждая такая иллюстрация была нарисована от руки и раскрашена. Богатое разнообразие коллекции Грассе де Сен-Совера отчетливо напоминает нам о том, насколько культурно разнообразными были города и отдельные области мира всего лишь 200 лет назад. Живя изолированно друг от друга, люди говорили на разных диалектах и языках. На улицах города или в сельской местности по одежде человека можно было легко определить, где он живет и какое положение в обществе занимает.

С тех пор наша манера одеваться сильно изменилась, и региональное разнообразие, бывшее прежде столь богатым, ныне утрачено. Теперь уже сложно различить жителей разных континентов, не говоря уже о разных городах, областях или странах. Быть может, мы променяли культурное разнообразие на более разнообразную личную жизнь — и уж точно на более разнообразную и изменчивую технологическую.

Во времена, когда сложно отличить одну книгу по компьютерным наукам от другой, издательство Manning демонстрирует изобретательность и инициативность компьютерного бизнеса, используя обложки, основанные на богатом разнообразии жизни различных регионов два века назад, возвращаемом к жизни благодаря иллюстрациям Грассе де Сен-Совера.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Практическое задание 1

Поиск выигрышной стратегии в карточной игре

УСЛОВИЕ ЗАДАЧИ

Хотите выиграть немного денег? Предлагаю сыграть в простенькую карточную игру с небольшими ставками. Перед вами лежит перетасованная колода из 52 карт. Все они повернуты рубашкой вверх. Одна половина этих карт красная, вторая — черная. Я буду переворачивать их одну за другой. Если последняя перевернутая мной карта окажется красной, вы выиграете доллар. В противном случае выиграю я.

В игре будет правило: вы сможете попросить меня остановить ее в любой момент. Как только вы говорите: «Стоп», я переворачиваю очередную карту и игра заканчивается. То есть эта карта будет выступать в роли последней, и если она окажется красной, доллар ваш, как показано на рис. ПЗ.1.

Мы можем сыграть в эту игру сколько угодно раз, и каждый раз колода будет перетасовываться. Расчет тоже после каждого раунда. Какой подход вы посчитаете наилучшим для победы в этой игре?

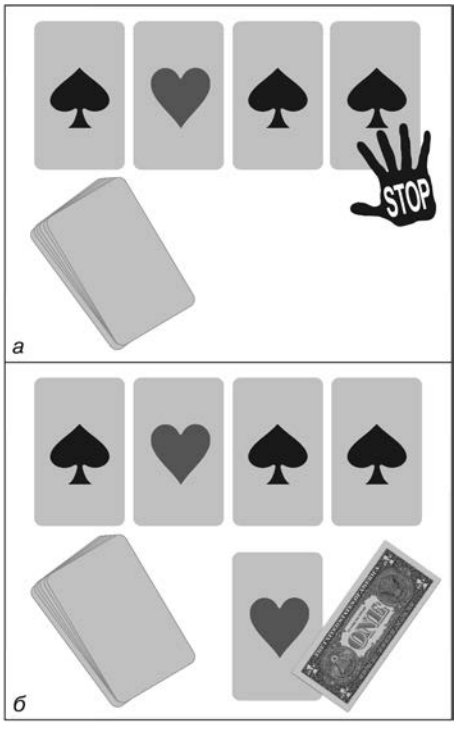


Рис. П31.1. Игра с переворачиванием карт. Начинаем с перетасованной колоды. Далее я поочередно переворачиваю верхние карты. На рисунке *а* я только что перевернул четвертую карту. Вы говорите: «Стоп». На рисунке *б* я перевернул пятую, последнюю, карту. Она оказалась красной. Доллар ваш

ОПИСАНИЕ

Для решения задачи нужно знать следующее.

1. Как вычислить вероятность происхождения наблюдаемых событий при помощи анализа пространства элементарных исходов, или вероятностного пространства.
2. Как отобразить вероятность событий для ряда значений интервалов.
3. Как симулировать случайные процессы, такие как подбрасывание монеты и перетасовка карт, с помощью Python.
4. Как оценивать свою уверенность в решениях, полученных в результате симуляций, используя анализ доверительных интервалов.

1 Вычисление вероятностей с помощью Python

В этой главе

- ✓ Основы теории вероятности.
- ✓ Вычисление вероятностей одного наблюдения.
- ✓ Вычисление вероятностей среди множества наблюдений.

Лишь немногие явления в жизни определены, большинство же происходят по воле случая. Всякий раз, болея за любимую спортивную команду, покупая лотерейный билет или вкладывая деньги в фондовый рынок, мы надеемся на некий результат, но его нельзя гарантировать. Наша повседневная жизнь пронизана случайностями. К счастью, эту случайность все же можно в какой-то степени контролировать. Мы знаем, что некоторые непредсказуемые события случаются не так часто, как другие, и что отдельные решения несут в себе меньше неопределенности, чем другой, более рисковый выбор. Поездка на работу на машине безопаснее, чем на мотоцикле. Инвестировать часть сбережений в пенсионный вклад безопаснее, чем поставить их все на одну руку в блек-джеке. Мы можем уверенно предугадывать эти компромиссы, потому что даже самые непредсказуемые системы все равно демонстрируют некоторое прогнозируемое поведение. И такое поведение тщательно изучалось на основе *теории вероятности*. Эта теория — сложная область математики, тем не менее ее элементы можно понять, не зная всей математической подоплеки. В действительности сложные задачи на вероятность можно решать с помощью Python, не зная ни одного математического уравнения. Подобный избавленный от уравнений подход требует базового понимания пространства *элементарных исходов*.

1.1. АНАЛИЗ ПРОСТРАНСТВА ЭЛЕМЕНТАРНЫХ ИСХОДОВ: ЛИШЕННЫЙ УРАВНЕНИЙ ПОДХОД ДЛЯ ИЗМЕРЕНИЯ НЕОПРЕДЕЛЕННОСТИ РЕЗУЛЬТАТОВ

Определенные действия имеют *измеримые исходы*. Пространство элементарных исходов — это набор всех возможных исходов действия. Возьмем, к примеру, простое подбрасывание монеты. Она может упасть вверх орлом или решкой. Таким образом, это действие может привести к одному из двух измеримых исходов: *орел* (`Heads`) или *решка* (`Tails`). Сохранив эти варианты в наборе Python, мы можем создать пространство элементарных исходов для подбрасывания монеты (листинг 1.1).

Листинг 1.1. Создание пространства исходов для подбрасывания монеты

```
sample_space = {'Heads', 'Tails'}
```

Сохраняя элементы в фигурных скобках, мы создаем множество Python. Множество Python — это коллекция уникальных неупорядоченных элементов

Представим, что случайно выбираем элемент из `sample_space`. В какой доле случаев этот выбранный элемент будет равняться `Heads`? Что ж, наше пространство исходов содержит два возможных элемента, каждый из которых занимает равную долю пространства. Следовательно, мы ожидаем, что `Heads` будет иметь частоту 1/2. Формально она определяется как *вероятность* исхода. Все исходы в `sample_space` имеют одинаковую вероятность, которая равняется `1 / len(sample_space)`.

Листинг 1.2. Вычисление вероятности выпадения орла

```
probability_heads = 1 / len(sample_space)
print(f'Probability of choosing heads is {probability_heads}')

Probability of choosing heads is 0.5
```

Вероятность выбора `Heads` равна 0,5 (листинг 1.2). Это напрямую связано с действием подбрасывания монеты. Предположим, что монета у нас без подвоха, а значит, с равной вероятностью может упасть на любую из сторон. Таким образом, ее подбрасывание в принципиальном смысле равнозначно выбору случайного элемента из `sample_space`. В связи с этим вероятность приземления монеты орлом вверх составляет 0,5, вероятность приземления монеты решкой вверх также равна 0,5.

Мы присвоили вероятности измеримым исходам. Однако здесь есть и еще один вопрос. Какова вероятность, что монета вообще упадет вверх орлом или решкой? Выражаясь более парадоксально, насколько вероятно, что она зависнет, вращаясь в воздухе, и так никогда и не упадет? Для получения точного ответа нужно определить понятие события. Событие — это подмножество элементов в `sample_space`, которые удовлетворяют некому *условию события* (рис. 1.1). Условие события — это простая логическая функция, на входе получающая один элемент из `sample_space`. Эта функция возвращает `True`, только если этот элемент удовлетворяет требованиям условий.

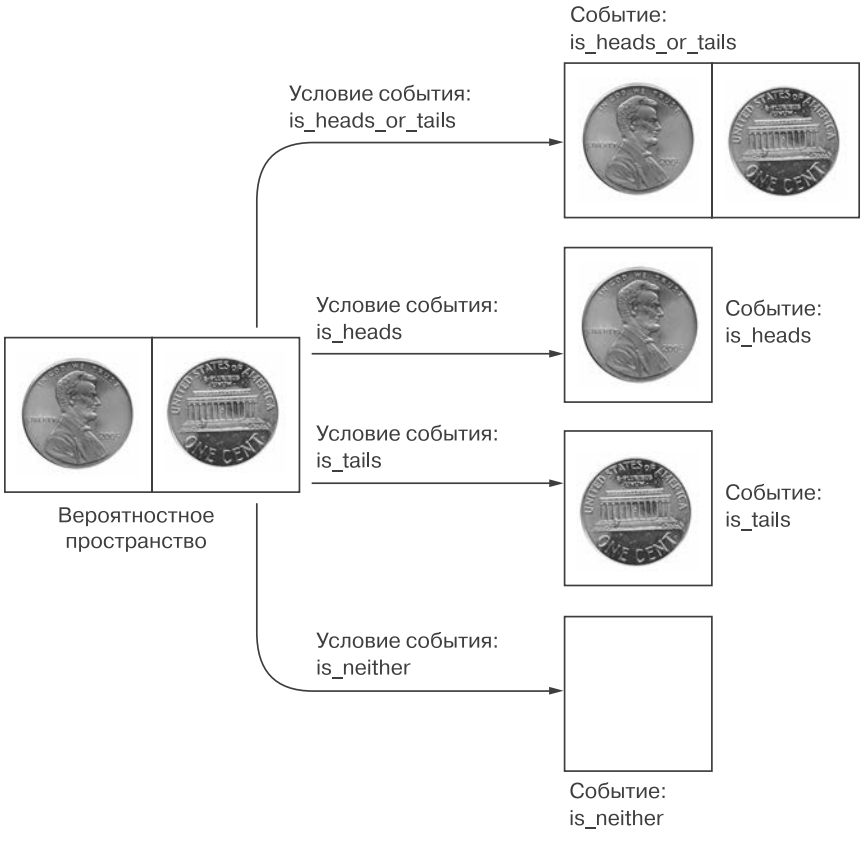


Рис. 1.1. Четыре условия события, примененные к пространству элементарных исходов. Это пространство содержит два элемента: heads и tails. Стрелки представляют условия событий. Каждое условие события является функцией «да/нет». Каждая функция отбрасывает исходы, которые не удовлетворяют ее условиям. Оставшиеся исходы формируют событие. Каждое событие содержит подмножество исходов, содержащихся в пространстве элементарных исходов. Здесь возможны четыре события: heads, tails, heads или tails и ни heads, ни tails

Определим два условия: одно, при котором монета выпадает орлом или решкой, и второе, при котором она не выпадает ни тем ни другим (листинг 1.3).

Листинг 1.3. Определение условий событий

```
def is_heads_or_tails(outcome): return outcome in {'Heads', 'Tails'}
def is_neither(outcome): return not is_heads_or_tails(outcome)
```

Также ради полноценности определим условия событий для двух базовых событий, в которых монета удовлетворяет ровно одному из четырех возможных исходов (листинг 1.4).

Листинг 1.4. Определение дополнительных условий событий

```
def is_heads(outcome): return outcome == 'Heads'
def is_tails(outcome): return outcome == 'Tails'
```

Эти условия событий можно передать в общую функцию `get_matching_event`. Она определяется в листинге 1.5. На входе она получает условие события и обобщенное пространство исходов. Перебрав полученное пространство, эта функция возвращает множество исходов, в котором `event_condition(outcome)` является `True`.

Листинг 1.5. Определение функции выявления события

```
def get_matching_event(event_condition, sample_space):
    return set([outcome for outcome in sample_space
                if event_condition(outcome)])
```

Давайте выполним `get_matching_event` для наших четырех условий, получив на выходе четыре итоговых события (листинг 1.6).

Листинг 1.6. Выявление событий на основе их условий

```
event_conditions = [is_heads_or_tails, is_heads, is_tails, is_neither]

for event_condition in event_conditions:
    print(f"Event Condition: {event_condition. name }")
    event = get_matching_event(event_condition, sample_space)
    print(f'Event: {event}\n')
```

Выводит имя функции event_condition

```
Event Condition: is_heads_or_tails
Event: {'Tails', 'Heads'}

Event Condition: is_heads
Event: {'Heads'}

Event Condition: is_tails
Event: {'Tails'}

Event Condition: is_neither
Event: set()
```

Мы успешно извлекли из `sample_space` четыре события. Какова же вероятность возникновения каждого из них? Ранее мы показали, что вероятность возникновения исхода, представленного одним элементом, для честной монеты равна `1 / len(sample_space)`. Это свойство можно обобщить на многоэлементные события. Вероятность события равна `len(event)/len(sample_space)`, но только если известно, что все исходы имеют одинаковый шанс случиться. Иными словами, вероятность многоэлементного события для честной монеты будет равна размеру этого события, разделенному на размер пространства элементарных исходов. Теперь мы используем размер события для вычисления вероятностей четырех событий (листинг 1.7).

Листинг 1.7. Вычисление вероятностей событий

```
def compute_probability(event_condition, generic_sample_space):
    event = get_matching_event(event_condition, generic_sample_space)
    return len(event) / len(generic_sample_space)

for event_condition in event_conditions:
    prob = compute_probability(event_condition, sample_space)
    name = event_condition. name
    print(f"Probability of event arising from '{name}' is {prob}")

Probability of event arising from 'is_heads_or_tails' is 1.0
Probability of event arising from 'is_heads' is 0.5
Probability of event arising from 'is_tails' is 0.5
Probability of event arising from 'is_neither' is 0.0
```

Функция compute_probability извлекает событие, соответствующее введенному условию события, для вычисления его вероятности

Вероятность равна размеру события, разделенному на размер пространства элементарных исходов

В результате выполнения кода мы получим диапазон вероятностей событий, наименьшая из которых будет равна 0,0, а наибольшая — 1,0. Эти значения представляют нижнюю и верхнюю границы вероятности. Никогда вероятность не может быть ниже 0,0 или выше 1,0.

1.1.1. Анализ несимметричной монеты

Мы вычислили вероятности для честной монеты. Но что произойдет, если монета будет иметь некую несбалансированность? Допустим, вероятность ее выпадения орлом в четыре раза выше, чем выпадения решкой. Как вычислить вероятность получения того или иного исхода, если их веса не равны? В этой ситуации можно выстроить взвешенное пространство элементарных исходов, представленное словарем Python. Каждый исход будет рассматриваться как ключ, чье значение сопоставляется с определенным весом. В нашем примере `Heads` будет иметь в четыре раза больший вес, нежели `Tails`, значит, `Tails` сопоставим с `1`, а `Heads` — с `4` (листинг 1.8).

Листинг 1.8. Представление пространства взвешенных исходов

```
weighted_sample_space = {'Heads': 4, 'Tails': 1}
```

Новое пространство хранится в словаре. Это позволяет переопределить его как сумму всех весов словаря. Внутри `weighted_sample_space` эта сумма будет равна `5` (листинг 1.9).

Листинг 1.9. Проверка размера пространства взвешенных исходов

```
sample_space_size = sum(weighted_sample_space.values())
assert sample_space_size == 5
```

Аналогичным образом можно переопределить размер событий. Каждое событие имеет набор исходов, и они сопоставляются с весами. Суммирование всех весов даст нам размер события. Таким образом, размер события, удовлетворяющего условию события `is_heads_or_tails`, также будет равен `5` (листинг 1.10).

Листинг 1.10. Проверка размера взвешенного события

```
event = get_matching_event(is_heads_or_tails, weighted_sample_space)
event_size = sum(weighted_sample_space[outcome] for outcome in event)
assert event_size == 5
```

Напомню, что эта функция перебирает каждый исход в полученном пространстве. Это значит, что она будет работать для входных словарей ожидаемым образом, поскольку Python перебирает ключи словаря, а не пары «ключ — значение», как многие другие популярные языки программирования

Обобщенные определения размера пространства исходов позволяют создать функцию `compute_event_probability`. На входе она будет получать переменную `generic_sample_space`, которая может быть либо взвешенным словарем, либо невзвешенным множеством (листинг 1.11).

Листинг 1.11. Определение обобщенной функции вычисления вероятности события

```
def compute_event_probability(event_condition, generic_sample_space):
    event = get_matching_event(event_condition, generic_sample_space)
    if type(generic_sample_space) == type(set()):
        return len(event) / len(generic_sample_space)

    event_size = sum(generic_sample_space[outcome]
                     for outcome in event)
    return event_size / sum(generic_sample_space.values())
```

Проверяет, установлено ли generic_event_space

Теперь можно вывести все вероятности событий для монеты со смещением, не переопределяя четыре функции условий событий (листинг 1.12).

Листинг 1.12. Вычисление вероятностей взвешенных событий

```
for event_condition in event_conditions:
    prob = compute_event_probability(event_condition, weighted_sample_space)
    name = event_condition. name
    print(f"Probability of event arising from '{name}' is {prob}")

Probability of event arising from 'is_heads' is 0.8
Probability of event arising from 'is_tails' is 0.2
Probability of event arising from 'is_heads_or_tails' is 1.0
Probability of event arising from 'is_neither' is 0.0
```

С помощью всего нескольких строк кода мы построили инструмент для решения множества задач по определению вероятности. Далее применим этот инструмент к более сложным случаям, нежели простое подбрасывание монеты.

1.2. ВЫЧИСЛЕНИЕ СЛОЖНЫХ ВЕРОЯТНОСТЕЙ

Далее решим несколько задач, используя `compute_event_probability`.

Задача 1. Анализ семьи с четырьмя детьми

Предположим, в семье четверо детей. Какова вероятность того, что двое из них мальчики? Предположим, что каждый ребенок в равной степени может быть либо мальчиком, либо девочкой. Таким образом мы сможем выстроить невзвешенное пространство элементарных исходов, в котором каждый исход будет представлять одну из возможных комбинаций четырех детей (листинг 1.13; рис. 1.2).

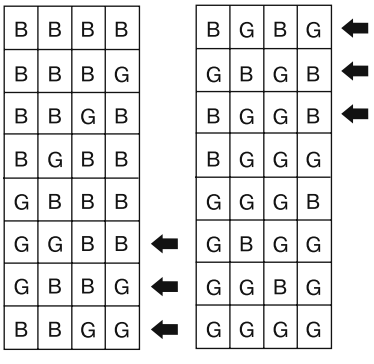


Рис. 1.2. Пространство элементарных исходов для четырех детей в семье. Каждый ряд содержит один из 16 возможных исходов. Каждый результат представляет уникальную комбинацию четырех детей. Пол ребенка обозначен буквой: B — мальчик, G — девочка. Исходы с двумя мальчиками отмечены стрелками. Здесь шесть стрелок, значит, вероятность наличия в семье двух мальчиков равна 6/16

Листинг 1.13. Вычисление пространства исходов комбинаций детей

```
possible_children = ['Boy', 'Girl']
sample_space = set()
for child1 in possible_children:
    for child2 in possible_children:
        for child3 in possible_children:
            for child4 in possible_children:
                outcome = (child1, child2, child3, child4)
                sample_space.add(outcome)
```

Каждая возможная комбинация четырех детей представлена кортежем из четырех элементов

Мы выполнили четыре вложенных цикла `for` для определения возможных последовательностей четырех рождений. Но это неэффективное использование кода. Есть более простой способ сгенерировать пространство исходов с помощью встроенной в Python функции `itertools.product`, которая возвращает все попарные

комбинации всех элементов входных списков. Мы передаем четыре экземпляра списка `possible_children` в `itertools.product`. После этого функция-произведение перебирает все четыре экземпляра списка, вычисляя все комбинации их элементов. Итоговый вывод будет равняться нашему пространству элементарных исходов (листинг 1.14).

Листинг 1.14. Вычисление пространства исходов с помощью product

```
from itertools import product
all_combinations = product(*(4 * [possible_children]))
assert set(all_combinations) == sample_space
```

Оператор * распаковывает множество аргументов, хранящихся в списке, после чего они передаются конкретной функции. Таким образом, вызов product(*(4 * [possible_children])) равнозначен вызову product(possible_children, possible_children, possible_children, possible_children)

Заметьте, что после выполнения этой строки все комбинации окажутся пусты. Дело в том, что product возвращает итерируемый объект Python, который можно перебрать только раз. Но для нас это не проблема — мы собираемся вычислить пространство исходов даже более эффективно, и all_combinations в последующем коде использоваться не будет

Этот код можно сделать еще более эффективным, выполнив `set(product(possible_children, repeat=4))`. Как правило, выполнение `product(possible_children, repeat=n)` возвращает итерируемый объект, включающий все возможные комбинации *n* детей (листинг 1.15).

Листинг 1.15. Передача repeat в product

```
sample_space_efficient = set(product(possible_children, repeat=4))
assert sample_space == sample_space_efficient
```

Далее вычислим долю `sample_space`, состоящую из семей с двумя мальчиками. Мы определяем условие события `has_two_boys` и передаем его в `compute_event_probability` (листинг 1.16).

Листинг 1.16. Вычисление вероятности наличия двух мальчиков

```
def has_two_boys(outcome): return len([child for child in outcome
                                       if child == 'Boy']) == 2
prob = compute_event_probability(has_two_boys, sample_space)
print(f"Probability of 2 boys is {prob}")

Probability of 2 boys is 0.375
```

Вероятность того, что в семье с четырьмя детьми могли родиться два мальчика, составляет 0,375. Следовательно, мы ожидаем, что 37,5 % семей, где есть всего четыре ребенка, включают в себя равное число мальчиков и девочек. Конечно же, фактический наблюдаемый процент таких семей будет варьироваться из-за элемента случайности.

Задача 2. Анализ множества бросков игрального кубика

Предположим, нам показали честный шестигранный кубик, чьи грани пронумерованы от 1 до 6, и бросили его шесть раз. Какова вероятность того, что результаты этих шести бросков в сумме дадут 21?

Мы начинаем с определения возможных значений одного броска, и у нас есть всего шесть целых чисел от 1 до 6 (листинг 1.17).

Листинг 1.17. Определение всех возможных бросков шестигранного кубика

```
possible_rolls = list(range(1, 7))
print(possible_rolls)
```

```
[1, 2, 3, 4, 5, 6]
```

Далее для шести последовательных бросков создаем пространство элементарных исходов с помощью функции `product` (листинг 1.18).

Листинг 1.18. Пространство исходов для шести последовательных бросков кубика

```
sample_space = set(product(possible_rolls, repeat=6))
```

В завершение определяем условие события `has_sum_of_21`, которое затем передаем в `compute_event_probability` (листинг 1.19).

Листинг 1.19. Вычисление вероятности получения определенной суммы в результате шести бросков

```
def has_sum_of_21(outcome): return sum(outcome) == 21

prob = compute_event_probability(has_sum_of_21, sample_space)
print(f"6 rolls sum to 21 with a probability of {prob}")
```

В принципиальном смысле шесть поочередных бросков одного кубика равнозначны броску шести кубиков одновременно

```
6 rolls sum to 21 with a probability of 0.09284979423868313
```

Шесть бросков кубика дадут сумму 21 более чем в 9 % случаев. Обратите внимание на то, что наш анализ можно записать в коде более сжато, используя лямбда-выражение (листинг 1.20). *Лямбда-выражения* — это однострочные анонимные функции, которые не требуют имени. В книге мы применяем их для передачи кратких функций другим функциям.

Листинг 1.20. Вычисление вероятности с использованием лямбда-выражения

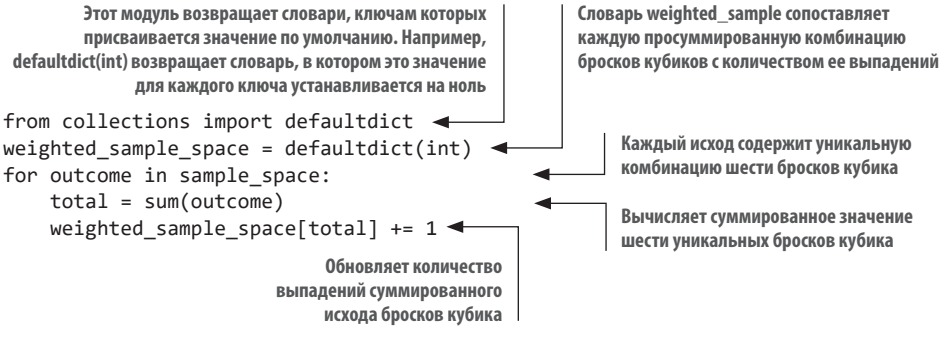
```
prob = compute_event_probability(lambda x: sum(x) == 21, sample_space)
assert prob == compute_event_probability(has_sum_of_21, sample_space)
```

Лямбда-выражения позволяют определять короткие функции в одной строке кода. Написание lambda x: функционально равнозначно написанию func(x):. Таким образом, lambda x: sum(x) == 21 функционально равнозначно has_sum_of_21

Задача 3. Вычисление вероятностей исходов броска кубика с помощью пространств взвешенных исходов

Мы только что вычислили вероятность получения суммы 21 в результате шести бросков кубика. Теперь еще раз вычислим эту вероятность с помощью пространства взвешенных исходов. То есть нужно преобразовать невзвешенное множество пространства исходов во взвешенный словарь данного пространства. Для этого потребуется подсчитать, сколько раз каждая сумма получается при всех возможных комбинациях бросков. Эти комбинации уже хранятся в вычисленном множестве `sample_space`. Сопоставив суммы бросков кубика с количеством их выпадений, получим `weighted_sample_space` (листинг 1.21).

Листинг 1.21. Сопоставление сумм бросков кубика с количеством их выпадений



Прежде чем повторно вычислять вероятность, вкратце рассмотрим свойства `weighted_sample_space` (листинг 1.22). Не все веса этого пространства исходов равны — некоторые из них намного меньше других. К примеру, есть всего один вариант, при котором общая сумма бросков составит 6: для этого необходимо выбросить шесть раз 1. Следовательно, мы ожидаем, что `weighted_sample_space[6]` равно `1`. При этом ожидаем, что `weighted_sample_space[36]` также будет равно `1`, поскольку для получения суммы 36 нужно шесть раз выбросить 6.

Листинг 1.22. Проверка очень редких комбинаций бросков кубика

```
assert weighted_sample_space[6] == 1
assert weighted_sample_space[36] == 1
```

При этом значение `weighted_sample_space[21]` будет заметно выше (листинг 1.23).

Листинг 1.23. Проверка более частых комбинаций бросков кубика

```
num_combinations = weighted_sample_space[21]
print(f"There are {num_combinations } ways for 6 die rolls to sum to 21")

There are 4332 ways for 6 die rolls to sum to 21
```

Как видно из вывода, существуют 4332 варианта получить с помощью шести бросков кубика сумму 21. Например, можно выбросить четыре 4, затем одну 3 и одну 2. Либо выбросить три 4, потом 5, 3 и 1. Здесь возможны тысячи комбинаций. Именно поэтому сумма 21 оказывается намного более вероятной, чем 6 (листинг 1.24).

Листинг 1.24. Исследование различных вариантов получения суммы 21

```
assert sum([4, 4, 4, 4, 3, 2]) == 21
assert sum([4, 4, 4, 5, 3, 1]) == 21
```

Обратите внимание на то, что полученное количество 4332 равно длине невзвешенного события, чьи броски кубика дают сумму 21. Кроме того, сумма значений в `weighted_sample` равна длине `sample_space`. Следовательно, между вычислением вероятности невзвешенного и взвешенного событий существует связь (листинг 1.25).

Листинг 1.25. Сравнение взвешенных и стандартных событий

```
event = get_matching_event(lambda x: sum(x) == 21, sample_space)
assert weighted_sample_space[21] == len(event)
assert sum(weighted_sample_space.values()) == len(sample_space)
```

А теперь еще раз вычислим вероятность, используя словарь `weighted_sample_space`. Итоговая вероятность выбрасывания суммы 21 должна остаться неизменной (листинг 1.26).

Листинг 1.26. Вычисление вероятности взвешенного события бросков кубика

```
prob = compute_event_probability(lambda x: x == 21,
                                 weighted_sample_space)
assert prob == compute_event_probability(has_sum_of_21, sample_space)
print(f"6 rolls sum to 21 with a probability of {prob}")

6 rolls sum to 21 with a probability of 0.09284979423868313
```

В чем польза применения пространства взвешенных исходов перед невзвешенным? Меньшее потребление памяти! Как мы далее увидим, невзвешенное множество `sample_space` содержит в 150 раз больше элементов, чем словарь пространства взвешенных исходов (листинг 1.27).

Листинг 1.27. Сравнение размера пространств взвешенных и невзвешенных исходов

```
print('Number of Elements in Unweighted Sample Space:')
print(len(sample_space))
print('Number of Elements in Weighted Sample Space:')
print(len(weighted_sample_space))
Number of Elements in Unweighted Sample Space:
46656
Number of Elements in Weighted Sample Space:
31
```

1.3. ВЫЧИСЛЕНИЕ ВЕРОЯТНОСТЕЙ ПО ДИАПАЗОНАМ ИНТЕРВАЛОВ

До сих пор мы анализировали только условия событий, удовлетворяющих какому-то одному значению. Теперь же займемся анализом условий событий, которые охватывают интервалы значений. *Интервал* — это множество всех чисел между двух граничных точек, включая и их самих. Давайте определим функцию `is_in_interval`, которая проверяет, попадает ли число в указанный интервал (листинг 1.28). Границы интервала будем контролировать передачей параметров `minimum` и `maximum`.

Листинг 1.28. Определение функции интервала

```
def is_in_interval(number, minimum, maximum):
    return minimum <= number <= maximum
```

Определяет закрытый интервал, в который включаются границы min/max. При необходимости можно определять и открытые интервалы. В них исключается как минимум одна граница

Имея функцию `is_in_interval`, мы можем вычислить вероятность того, что связанное с событием значение попадет в некий конкретный интервал. В качестве примера определим, насколько вероятно то, что последовательные шесть бросков кубика дадут сумму от 10 до 21 включительно (листинг 1.29).

Листинг 1.29. Вычисление вероятности попадания в интервал

```
prob = compute_event_probability(lambda x: is_in_interval(x, 10, 21),
                                 weighted_sample_space)
print(f"Probability of interval is {prob}")

Probability of interval is 0.5446244855967078
```

Лямбда-функция, которая получает некий ввод x и возвращает True, если x попадает в интервал от 10 до 21. Эта однострочная функция служит условием события

Шесть бросков кубика попадают в заданный интервал более чем в 54 % случаев. Получается, что сумме бросков от 13 до 20 удивляться не стоит.

1.3.1. Оценка крайних значений с помощью интервального анализа

Интервальный анализ является важнейшим инструментом для решения целого класса задач по вероятности и статистике, одна из которых связана с оценкой экстремальных значений, когда все сводится к определению того, не являются ли наблюдаемые данные слишком невероятными, чтобы оказаться правдой.

Данные выглядят невероятными, когда оказываются слишком необычными, чтобы получиться случайно. Предположим, мы наблюдаем десять подбрасываний якобы честной монеты, которая восемь из десяти раз выпадает орлом. Будет ли это логичным исходом для честной монеты? А может, ее тайком подделали так, чтобы она

чаще падала орлом вверх? Чтобы это выяснить, нужно ответить на вопрос: «Какова вероятность того, что десять подбрасываний монеты приведут к невероятному количеству выпадения орлов?» Это невероятное количество мы определим как восемь и более. Тогда можно будет описать задачу так: какова вероятность того, что десять честных подбрасываний монеты дадут от восьми до десяти исходов с орлом?

Ответ получим, вычислив вероятность интервала. Но сначала нам потребуется пространство элементарных исходов для каждой возможной последовательности исходов десяти подброшенных монет. Давайте сгенерируем пространство взвешенных исходов. Как уже говорилось, это будет более эффективно, чем использование невзвешенного представления.

Код в листинге 1.30 создает словарь `weighted_sample_space`. Его ключи равняются общему числу получаемых в результате подбрасываний орлов в диапазоне от 0 до 10. Эти количества сопоставляются со значениями, каждое из которых отражает число комбинаций подбрасываний монеты, дающих указанное ключом количество орлов. Таким образом, мы ожидаем, что `weighted_sample_space[10]` будет равняться `1`, поскольку возможен всего один вариант подбросить монету десять раз и получить десять орлов. При этом мы ожидаем, что `weighted_sample_space[9]` будет равно `10`, так как одна решка среди девяти орлов может выпасть в десяти разных позициях.

Листинг 1.30. Вычисление пространства исходов для десяти подбрасываний монеты

```
def generate_coin_sample_space(num_flips=10):
    weighted_sample_space = defaultdict(int)
    for coin_flips in product(['Heads', 'Tails'], repeat=num_flips):
        heads_count = len([outcome for outcome in coin_flips
                           if outcome == 'Heads'])
        weighted_sample_space[heads_count] += 1
    return weighted_sample_space

weighted_sample_space = generate_coin_sample_space()
assert weighted_sample_space[10] == 1
assert weighted_sample_space[9] == 10
```

С целью повторного использования мы определяем общую функцию, возвращающую пространство взвешенных исходов для num_flips подбрасываний монеты. Параметр num_flips устанавливается на 10 подбрасываний

Количество орлов в уникальной последовательности num_flips подбрасываний монеты

Пространство взвешенных исходов готово. Теперь вычисляем вероятность попадания в интервал от восьми до десяти выпадений орлов (листинг 1.31).

Листинг 1.31. Вычисление шанса выпадения маловероятного количества орлов

```
prob = compute_event_probability(lambda x: is_in_interval(x, 8, 10),
                                 weighted_sample_space)
print(f"Probability of observing more than 7 heads is {prob}")

Probability of observing more than 7 heads is 0.0546875
```

Десять подбрасываний честной монеты дают более семи орлов приблизительно в 5 % случаев. Получение такого количества орлов нельзя назвать частым. Значит ли это, что монета подделана? Не обязательно. Мы еще не рассмотрели невероятные количества выпадения решки. Если бы выпало девять решек, а не орлов, это тоже вызвало бы подозрение относительно монеты. Вычисленный нами интервал не берет в расчет этот маловероятный исход. Получается, что мы рассматривали восемь и более выпадений решки как еще одну нормальную вероятность. Для оценки честности монеты необходимо включить в расчеты вероятность получения восьми и более решек, что будет равнозначно выпадению двух или менее орлов.

Сформулируем задачу так: «Какова вероятность того, что десять подбрасываний честной монеты дадут либо от нуля до двух орлов, либо от восьми до десяти решек?» Проще говоря, какова вероятность того, что подбрасывания монеты не дадут от трех до семи решек? Вычисляется эта вероятность следующим кодом (листинг 1.32).

Листинг 1.32. Вычисление вероятности получения экстремального интервала

```
prob = compute_event_probability(lambda x: not is_in_interval(x, 3, 7),
                                 weighted_sample_space)
print(f"Probability of observing more than 7 heads or 7 tails is {prob}")

Probability of observing more than 7 heads or 7 tails is 0.109375
```

Десять подбрасываний честной монеты дадут не менее восьми одинаковых исходов примерно в 7 % случаев. Это невысокая вероятность, но она все равно вполне возможна. Без дополнительных доказательств сложно решить, действительно ли монета поддельная. Так что предлагаю эти доказательства отыскать. Предположим, мы подбрасываем монету еще десять раз и опять получаем восемь орлов. Теперь имеем уже 16 орлов после 20 подбрасываний монеты. Наша уверенность в честности монеты уменьшается, но насколько? Это можно выяснить, измерив изменение вероятности. Для этого найдем вероятность того, что 20 подбрасываний честной монеты не дадут от 5 до 15 орлов (листинг 1.33).

Листинг 1.33. Анализ экстремального количества выпадения орлов при 20 подбрасываниях монеты

```
weighted_sample_space_20_flips = generate_coin_sample_space(num_flips=20)
prob = compute_event_probability(lambda x: not is_in_interval(x, 5, 15),
                                 weighted_sample_space_20_flips)
print(f"Probability of observing more than 15 heads or 15 tails is {prob}")

Probability of observing more than 15 heads or 15 tails is 0.011817932128906250
```

Новая вероятность упала с примерно 0,1 до примерно 0,01. Полученное дополнительное свидетельство вызвало десятикратное снижение нашей уверенности в честности монеты. Но, несмотря на уменьшение вероятности, соотношение орлов и решек осталось прежним, то есть 4:1. И изначальный, и последующий

эксперименты давали 80 % орлов и 20 % решек. Это подводит нас к интересному вопросу: «Почему при увеличении количества подбрасываний монеты вероятность получения экстремального исхода снижается?» Выяснить это можно, выполнив подробный математический анализ. Намного более интуитивным решением будет просто визуализировать распределение количества орлов по двум словарям пространства элементарных исходов. Эта визуализация, по сути, будет графиком ключей (количества орлов), сопоставленных со значениями (количествами комбинаций), присутствующими в каждом словаре. Реализовать это можно с помощью Matplotlib — наиболее популярной библиотеки визуализации Python. В следующей главе рассмотрим ее использование и применение к теории вероятности.

РЕЗЮМЕ

- *Пространство элементарных исходов* — это множество всех возможных исходов, которые может произвести действие.
- *Событие* — это подмножество пространства исходов, содержащее только те исходы, которые удовлетворяют определенному *условию события*. Условие события — это логическая функция, получающая на входе исход и возвращающая `True` либо `False`.
- *Вероятность* события равна доле исходов событий из всех возможных исходов в пространстве элементарных исходов.
- Вероятности можно вычислять для *численных интервалов*. Интервал определяется как множество всех чисел, содержащихся между двух граничных значений.
- Вычисление вероятности интервала полезно для определения того, является ли наблюдение невероятным.

2 Графическое отображение вероятностей с помощью Matplotlib

В этой главе

- ✓ Создание простых графиков с помощью Matplotlib.
- ✓ Разметка отрисованных данных.
- ✓ Что такое распределение вероятностей.
- ✓ Графическое отображение и сравнение множественных распределений вероятностей.

Графики данных — один из наиболее ценных инструментов в арсенале аналитика. Без хороших визуализаций наша возможность получения полезной информации на основе имеющихся данных оказывается ограниченной. К счастью, в нашем распоряжении есть внешняя библиотека Matplotlib, которая полноценно оптимизирована для вывода высокоточных графиков и визуализаций данных. Здесь мы используем ее, чтобы лучше разобраться с вероятностями исходов подбрасываний монет, вычисленных в главе 1.

2.1. ОСНОВНЫЕ ГРАФИКИ MATPLOTLIB

Начнем с установки библиотеки.

ПРИМЕЧАНИЕ

Для установки Matplotlib выполните в терминале pip install matplotlib.

По завершении установки импортируйте `matplotlib.pyplot` — основной модуль генерации графиков. По соглашению этот модуль обычно импортируется с помощью сокращенного псевдонима `plt` (листинг 2.1).

Листинг 2.1. Импорт Matplotlib

```
import matplotlib.pyplot as plt
```

Теперь отобразим кое-какие данные с помощью `plt.plot`. Этот метод получает на входе два итерируемых объекта: `x` и `y`. Вызов `plt.plot(x, y)` подготавливает 2D-график `x` относительно `y`. Для фактического отображения графика необходимо далее вызвать `plt.show()`. Давайте присвоим `x` равномерно распределенные числа от 0 до 10, а `y` — двойные значения `x`. Следующий код (листинг 2.2) визуализирует эту линейную связь (рис. 2.1).

Листинг 2.2. Графическая отрисовка линейного соотношения

```
x = range(0, 10)
y = [2 * value for value in x]
plt.plot(x, y)
plt.show()
```

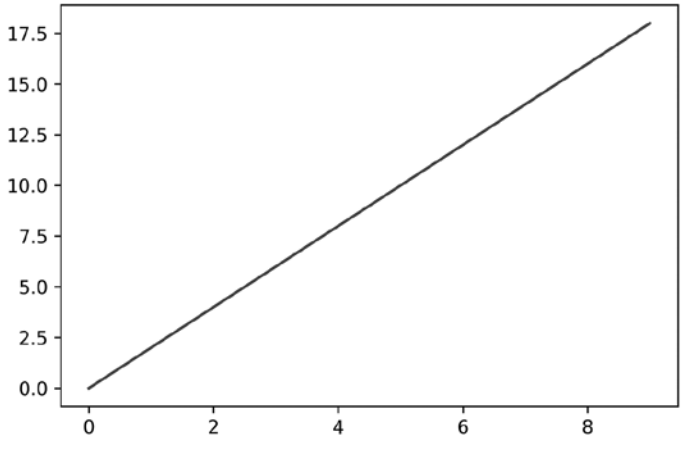


Рис. 2.1. График Matplotlib, отображающий соотношение x и 2x. Переменная x представляет целые числа от 0 до 10

ВНИМАНИЕ

Оси этого линейного графика разделены неравномерно, поэтому наклон линии получился менее крутой, чем есть на самом деле. Выровнять оси можно с помощью вызова plt.axis('equal'), однако это приведет к странной визуализации, где будет слишком много пустого пространства. На протяжении книги мы будем опираться на автоматическую подстройку осей в Matplotlib, попутно обращая внимание на подстраиваемые ею длины.

Визуализация готова. На ней десять точек по оси *Y* были объединены с помощью плавных отрезков. Если мы решим отрисовать десять точек по отдельности, то сможем сделать это при помощи метода `plt.scatter` (листинг 2.3; рис. 2.2).

Листинг 2.3. Отрисовка отдельных точек данных

```
plt.scatter(x, y)
plt.show()
```

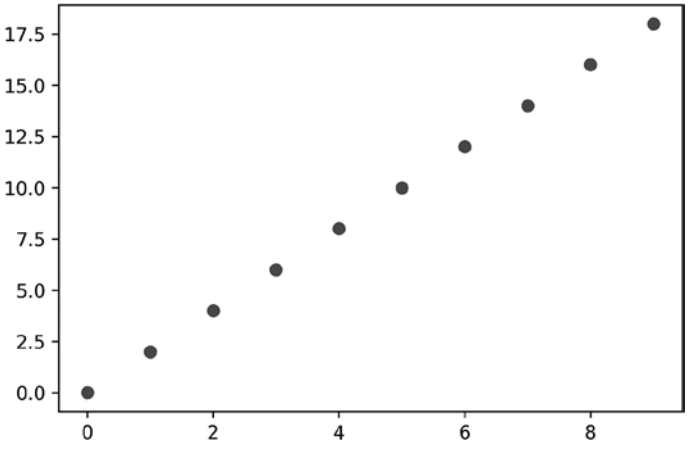


Рис. 2.2. Точечный график Matplotlib, отображающий соотношение x и 2x. Переменная x представляет целые числа от 0 до 10. Отдельные числа видимы как точки, распределенные по графику

Предположим, что мы хотим выделить интервал, в котором `x` начинается с `2` и завершается `6`. Для этого затеняем область указанного интервала под отрисованной кривой с помощью метода `plt.fill_between`. Этот метод получает на входе параметры `x`, `y` и `where`. Последний определяет охват интервала. Входным значением `where` является список логических значений, в котором элемент является `True`, если значение `x` в соответствующем индексе попадает в заданный интервал. В коде листинга 2.4 устанавливаем параметр `where` как равный `[is_in_interval(value, 2, 6) for value in x]`. Также выполняем `plt.plot(x,y)`, чтобы присовокупить затененный интервал к общему отрезку (рис. 2.3).

Листинг 2.4. Затенение интервала под отрисованным графиком

```
plt.plot(x, y)
where = [is_in_interval(value, 2, 6) for value in x]
plt.fill_between(x, y, where=where)
plt.show()
```

На этот момент мы рассмотрели три метода визуализации: `plt.plot`, `plt.scatter` и `plt.fill_between`. Теперь объединим выполнение всех этих методов на одном

графике (листинг 2.5; рис. 2.4). Таким образом мы выделим интервал под непрерывной линией, а также выразим отдельные ее координаты.

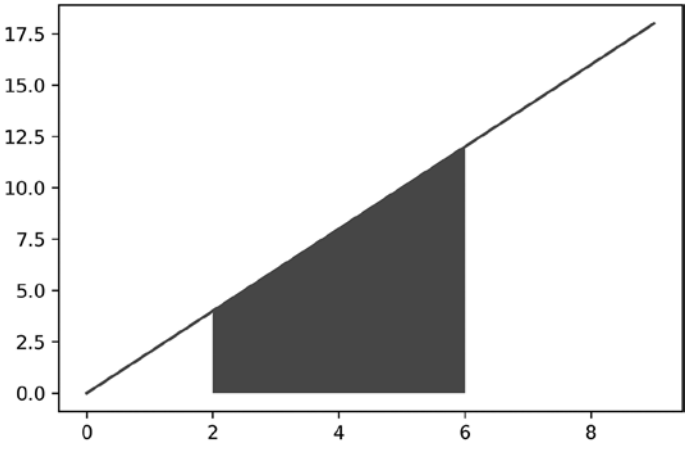


Рис. 2.3. Соединенная линия графика с затененным интервалом, охватывающим значения от 2 до 6

Листинг 2.5. Выражение отдельных координат на непрерывном графике

```
plt.scatter(x, y)
plt.plot(x, y)
plt.fill_between(x, y, where=where)
plt.show()
```

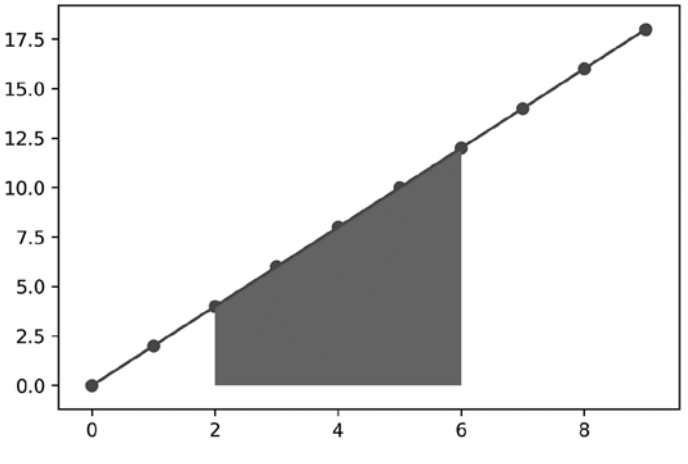


Рис. 2.4. Соединенный и точечный графики объединены с затененным интервалом. Отдельные числа отражают точки, относительно которых строится плавная неделимая линия

Ни один график данных не будет полноценным без подписей к осям *X* и *Y*. Эти подписи можно установить с помощью методов `plt.xlabel` и `plt.ylabel` (листинг 2.6; рис. 2.5).

Листинг 2.6. Добавление подписей осей

```
plt.plot(x, y)
plt.xlabel('Values between zero and ten')
plt.ylabel('Twice the values of x')
plt.show()
```

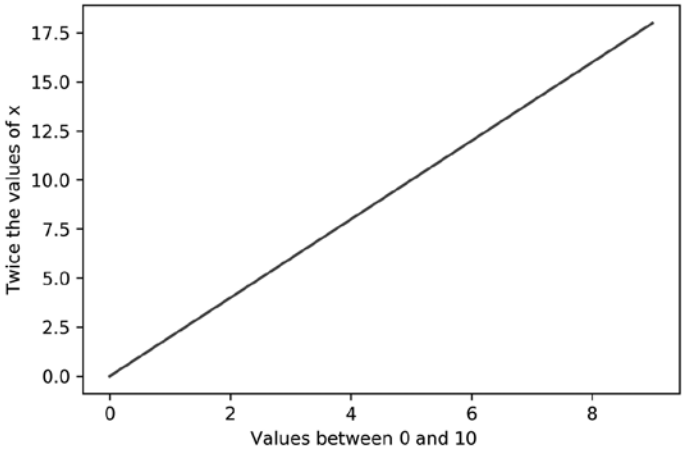


Рис. 2.5. График Matplotlib с подписями осей X и Y

РАСПРОСТРАНЕННЫЕ МЕТОДЫ MATPLOTLIB

- `plt.plot(x, y)` — рисует график расположения элементов x относительно элементов y. Отрисованные точки соединяются плавными линиями.
- `plt.scatter(x, y)` — рисует элементы x относительно элементов y. Отрисованные точки визуализируются по отдельности без соединения линиями.
- `plt.fill_between(x, y, where=booleans)` — выделяет часть области под отрисованной кривой. Сама эта кривая получается отрисовкой x относительно y. Параметр `where` определяет все выделенные интервалы, для чего получает список логических значений, соответствующих элементам x. Каждое логическое значение оказывается `True`, если соответствующее значение x расположено в указанном интервале.
- `plt.xlabel(label)` — устанавливает подпись оси *X* отрисованной кривой как равную `label`.
- `plt.ylabel(label)` — устанавливает подпись оси *Y* отрисованной кривой как равную `label`.

2.2. ПОСТРОЕНИЕ ГРАФИКА ВЕРОЯТНОСТЕЙ ИСХОДОВ ПОДБРАСЫВАНИЯ МОНЕТЫ

Теперь у нас есть инструменты для визуализации связей между количеством подбрасываний монеты и вероятностью выпадения орла. В главе 1 мы выяснили, насколько вероятно получить этот результат в 80 % и более случаев подбрасывания монеты. Когда же количество подбрасываний возросло, эта вероятность уменьшилась, и нам стало интересно почему. Вскоре мы выясним это, начертив график связи количества полученных орлов с количеством комбинаций подбрасываний монеты. В главе 1 мы эти значения уже вычислили. Ключи словаря `weighted_sample_space` содержат все возможные количества орлов, которые можно получить за десять подбрасываний монеты. Эти количества сопоставляются с числом комбинаций. В то же время словарь `weighted_sample_space_20_flips` содержит аналогичные сопоставления, но уже для 20 подбрасываний монеты.

Наша цель — сравнить графики данных, полученные на основе обоих словарей (листинг 2.7). Начнем с отрисовки элементов `weighted_sample_space`: его ключи мы отрисовываем на оси *X* относительно соответствующих значений, отражаемых на оси *Y*. Ось *X* относится к `'Head-count'`, а ось *Y* — к `'Number of coin-flip combinations with x heads'`. Мы используем точечный график для непосредственной визуализации связей между ключами и значениями без соединения отдельных точек (рис. 2.6).

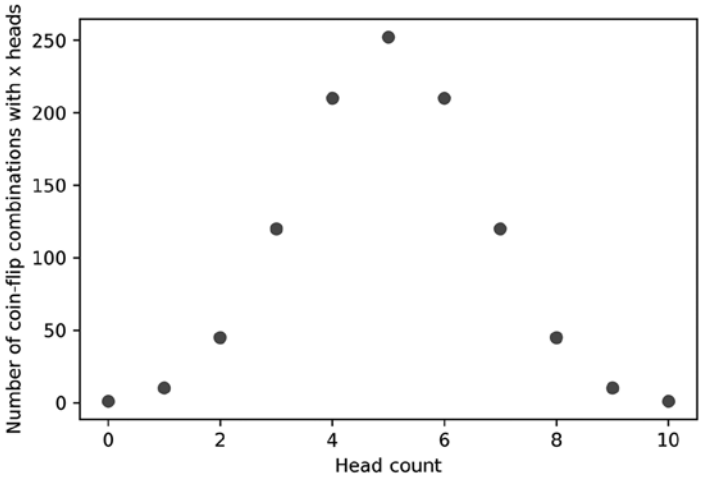


Рис. 2.6. Точечный график пространства исходов для десяти подброшенных монет. Симметричный график центрирован относительно пика в точке, отражающей получение пяти орлов

Листинг 2.7. Построение пространства взвешенных исходов подбрасываний монеты

```
x_10_flips = list(weighted_sample_space.keys())
y_10_flips = [weighted_sample_space[key] for key in x_10_flips]
plt.scatter(x_10_flips, y_10_flips)
plt.xlabel('Head-count')
plt.ylabel('Number of coin-flip combinations with x heads')
plt.show()
```

Визуализированное пространство элементарных исходов имеет симметричную форму. Эта симметрия строится вокруг пиковой точки, соответствующей получению пяти орлов. Следовательно, комбинации, дающие количество орлов, близкое к пяти, случаются чаще, чем те, которые от этого значения удалены. Как мы узнали в предыдущей главе, эти частоты соответствуют вероятностям. Таким образом, чем ближе количество получаемых орлов к пяти, тем такой результат вероятнее. Давайте подчеркнем это, построив соответствующие вероятности непосредственно на оси *Y* (рис. 2.7). График вероятности позволит нам заменить длинную подпись оси *Y* более короткой формулировкой `'Probability'`. Вычислить вероятности для этой оси можно, взяв имеющиеся количества комбинаций и разделив их на общий размер пространства исходов (листинг 2.8).

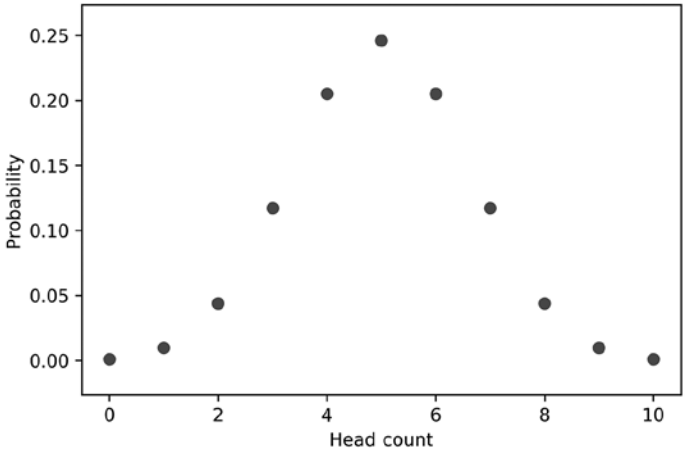


Рис. 2.7. Точечный график, отображающий сопоставление количества выпадений орлов с вероятностью выпадения этого количества. Вероятности можно вывести, взглянув непосредственно на график

Листинг 2.8. Отрисовка вероятностей исходов подбрасываний монеты

```
sample_space_size = sum(weighted_sample_space.values())
prob_x_10_flips = [value / sample_space_size for value in y_10_flips]
plt.scatter(x_10_flips, prob_x_10_flips)
```

```
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

Полученный график позволяет визуально оценить вероятность получения любого количества орлов. Таким образом, просто глядя на него, мы можем определить, что вероятность получения пяти результатов с орлами составляет примерно 0,25. Это сопоставление между значениями `x` и вероятностями называется *распределением вероятностей*. Распределения вероятностей демонстрируют определенные математически согласующиеся свойства, которые делают их полезными при анализе вероятностей. Рассмотрим в качестве примера значения `x` любого распределения вероятностей: они соответствуют всем возможным значениям случайной переменной `r`. Вероятность того, что `r` попадет в определенный интервал, равна области под кривой вероятностей в пределах этого интервала. Следовательно, общая область под распределением вероятностей всегда равна 1. Это остается верным для любого распределения, включая наш график количеств исходов с орлом. И листинг 2.9 подтверждает это, выполняя `sum(prob_x_10_flips)`.

ПРИМЕЧАНИЕ

Мы можем вычислить площадь под каждой вероятностью получения определенного количества орлов p, используя вертикальный прямоугольник. Высота этого прямоугольника равна p, а ширина — 1,0, поскольку все последовательные количества орлов на оси X отделены друг от друга на один шаг. Следовательно, площадь прямоугольника составляет p × 1,0, что равно p. Значит, общая площадь под распределением равна sum([p for p in prob_x_10_flips]). В главе 3 мы подробно разберем применение прямоугольников для определения площади.

Листинг 2.9. Подтверждение того, что все вероятности суммируются в 1,0

```
assert sum(prob_x_10_flips) == 1.0
```

Область под интервалом, охватывающим диапазон от восьми до десяти исходов с орлами, равна вероятности получения восьми или более орлов. Визуализируем эту область с помощью метода `plt.fill_between` (листинг 2.10). Мы также задействуем `plt.plot` и `plt.scatter` для отображения отдельных количеств орлов, включающих относящихся к затененному интервалу (рис. 2.8).

Листинг 2.10. Затенение интервала под кривой вероятностей

```
plt.plot(x_10_flips, prob_x_10_flips)
plt.scatter(x_10_flips, prob_x_10_flips)
where = [is_in_interval(value, 8, 10) for value in x_10_flips]
plt.fill_between(x_10_flips, prob_x_10_flips, where=where)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

ПРИМЕЧАНИЕ

Мы целенаправленно сгладили затененный интервал, чтобы сделать график более визуально привлекательным. Однако истинная область интервала не столь гладкая — она состоит из отдельных прямоугольных блоков, напоминающих ступеньки. Отдельные они потому, что количества орлов представлены неделимыми целыми числами. Если мы хотим визуализировать фактическую область в форме ступенек, то нужно передать параметр ds="steps-mid" в plt.plot и параметр step="mid" в plt.fill_between.

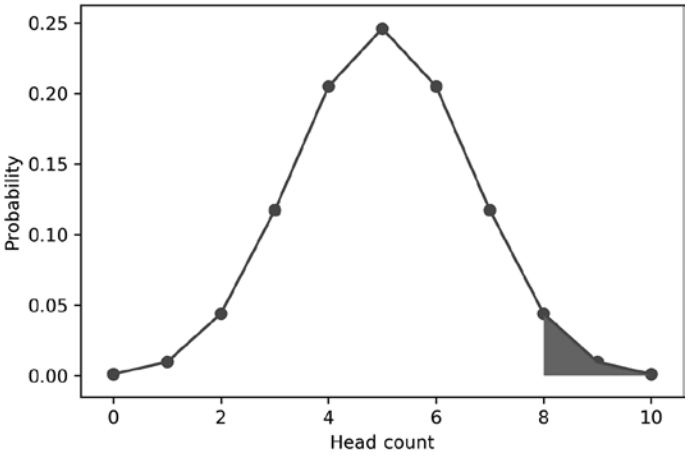


Рис. 2.8. Наложение представлений соединенного и точечного графиков распределения вероятностей исходов подбрасывания монет. Затененный интервал охватывает количества от восьми до десяти. Эта затененная область равна вероятности получения восьми или более исходов с орлом

Теперь затеним интервал, включающий вероятность получения восьми или более решек. Приведенный в листинге 2.11 код выделяет эти экстремальные значения вдоль обеих границ результатов с решками в распределении вероятностей (рис. 2.9).

Листинг 2.11. Затенение интервала под экстремальными значениями кривой вероятностей

```
plt.plot(x_10_flips, prob_x_10_flips)
plt.scatter(x_10_flips, prob_x_10_flips)
where = [not is_in_interval(value, 3, 7) for value in x_10_flips]
plt.fill_between(x_10_flips, prob_x_10_flips, where=where)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

Эти два симметричных затененных интервала охватывают правый и левый края кривой результатов. Исходя из предыдущего анализа, мы знаем, что вероятность

получения более семи орлов или решек равна примерно 10 %. Следовательно, каждый симметрично затененный сегмент для решек охватывает примерно 5 % общей площади под кривой.

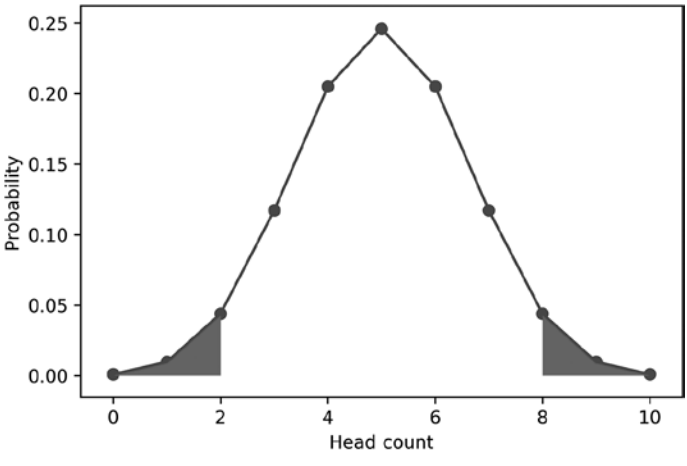


Рис. 2.9. Наложение представлений соединенного и точечного графиков распределения вероятностей исходов подбрасываний монеты. Затененные интервалы охватывают крайние значения для исходов с орлами и решками. Эти интервалы симметричны и визуально указывают на то, что их вероятности равны

2.2.1. Сравнение нескольких распределений вероятностей исходов подбрасывания монеты

Построение графика распределения исходов десяти подбрасываний монеты упрощает визуальное восприятие связанных с ними вероятностей интервалов. Далее мы расширим график распределения, чтобы он отражал исходы уже 20 подбрасываний монеты. Оба этих распределения мы отобразим на одном рисунке, но для начала нужно вычислить также число исходов с орлами (ось X) и вероятности (ось Y) для распределения исходов 20 подбрасываний (листинг 2.12).

Листинг 2.12. Вычисление вероятностей для распределения исходов 20 подбрасываний монеты

```
x_20_flips = list(weighted_sample_space_20_flips.keys())
y_20_flips = [weighted_sample_space_20_flips[key] for key in x_20_flips]
sample_space_size = sum(weighted_sample_space_20_flips.values())
prob_x_20_flips = [value / sample_space_size for value in y_20_flips]
```

Теперь можно визуализировать два распределения одновременно (рис. 2.10), выполнив `plt.plot` и `plt.scatter` для обоих. Один из параметров, `color`, служит для

выделения второго распределения. Для него устанавливаем цвет `black`, передавая `color='black'`. В качестве альтернативы можно избежать ввода всего имени цвета, передав `'k'` — односимвольный код Matplotlib, означающий черный.

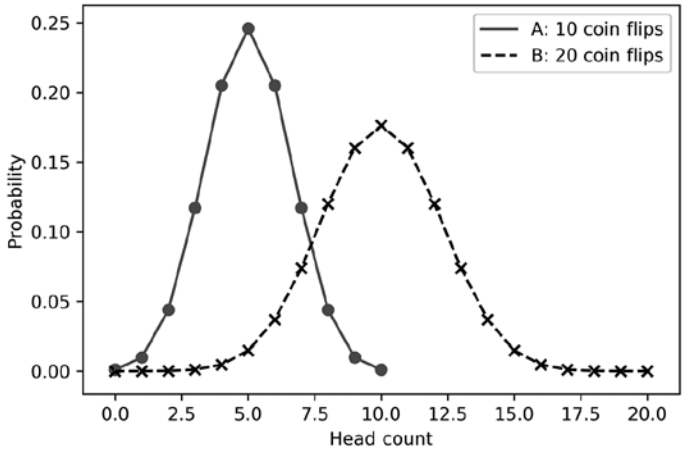


Рис. 2.10. Распределение вероятностей для 10 (A) и 20 (B) подбрасываний монеты. Распределение для 20 подбрасываний выделено пунктирной линией и x-образными точками

Выделить второе распределение можно и другими способами — передав `linestyle=='--'` в `plt.plot`, чтобы точки распределения были связаны пунктирной линией, а не обычной (листинг 2.13). Также можно выделить сами точки, используя для них *x*-образные маркеры вместо закрашенных кружочков. Для этого нужно передать `marker='x'` в `plt.scatter`. Наконец, мы добавляем на рисунок легенду, передавая параметр `label` в каждый из двух вызовов `plt.plot` и выполняя метод `plt.legend()` для отображения легенды. Внутри нее распределения исходов 10 и 20 подбрасываний монеты обозначены A и B соответственно.

Листинг 2.13. Отрисовка двух распределений одновременно

```
plt.plot(x_10_flips, prob_x_10_flips, label='A: 10 coin-flips')
plt.scatter(x_10_flips, prob_x_10_flips)
plt.plot(x_20_flips, prob_x_20_flips, color='black', linestyle='--',
         label='B: 20 coin-flips')
plt.scatter(x_20_flips, prob_x_20_flips, color='k', marker='x')
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.legend()
plt.show()
```

ТИПИЧНЫЕ ПАРАМЕТРЫ СТИЛИЗАЦИИ В MATPLOTLIB

- `color` — определяет цвет отрисовки. В качестве этой установки может выступать имя цвета или односимвольный код. И `color='black'`, и `color='k'` генерируют черный график, а `color='red'` и `color='r'` — красный.
- `linestyle` — определяет стиль отрисовываемой линии, соединяющей точки данных. По умолчанию этот параметр равен `'-'`. Установка `linestyle='-'` генерирует цельную линию, `linestyle='--'` — пунктирную, `linestyle=':'` — линию из точек, а `linestyle='.'` рисует линию, перемежая точки и тире.
- `marker` — определяет стиль маркеров, присваиваемых отдельным точкам данных на графике. По умолчанию значение этого параметра равно `'o'`. Установка `marker='o'` приводит к генерации круглых маркеров, `marker='x'` рисует их в форме x, `marker='s'` создает квадратные маркеры, а `marker='p'` — пятиугольные.
- `label` — сопоставляет метку с указанным цветом и стилем. Это сопоставление отражается в легенде графика. Для визуального отображения легенды необходимо следом выполнить `plt.legend()`.

Мы визуализировали два распределения. Далее выделим интересующий нас интервал (80 % орлов или решек) на каждой из двух кривых (листинг 2.14; рис. 2.11). Заметьте, что область под краями распределения B очень мала. Мы удалим точки распределения на графиках, чтобы отчетливее выделить интервалы на их крайних участках. Также заменим пунктирный стиль отрисовки распределения B более отчетливым точечным, установив `linestyle=':'`.

Листинг 2.14. Выделение интервалов под двумя графиками распределений

```
plt.plot(x_10_flips, prob_x_10_flips, label='A: 10 coin-flips')
plt.plot(x_20_flips, prob_x_20_flips, color='k', linestyle=':',
         label='B: 20 coin-flips')

where_10 = [not is_in_interval(value, 3, 7) for value in x_10_flips]
plt.fill_between(x_10_flips, prob_x_10_flips, where=where_10)
where_20 = [not is_in_interval(value, 5, 15) for value in x_20_flips]
plt.fill_between(x_20_flips, prob_x_20_flips, where=where_20)

plt.xlabel('Head-Count')
plt.ylabel('Probability')
plt.legend()
plt.show()
```

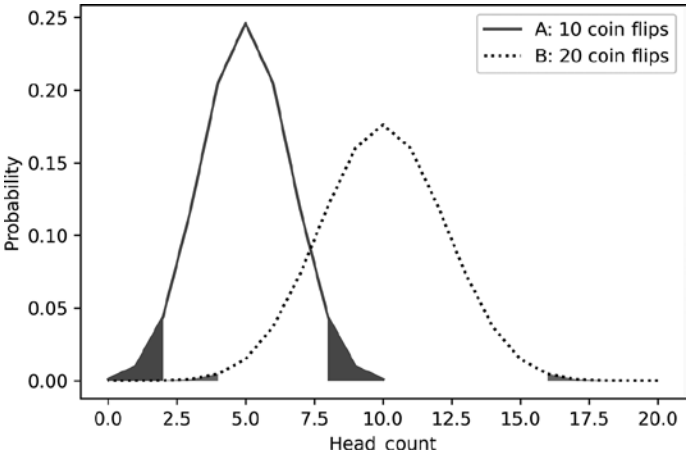


Рис. 2.11. Распределение вероятностей для 10 (A) и 20 (B) подбрасываний монеты. Затененные интервалы снизу представляют экстремальное количество орлов и решек. Площадь темного интервала под B в десять раз меньше площади аналогичного интервала под A

Затененная область под крайними участками распределения B намного меньше, чем аналогичный интервал под распределением A. Объясняется это тем, что распределение A имеет более широкие и приподнятые края, которые охватывают большую площадь. Толщина крайних участков обусловливается разницей в вероятностях интервалов.

Такая визуализация информативна, но только если выделить интервалы под обеими кривыми. Без вызовов `plt.fill_between` мы не сможем ответить на поставленный ранее вопрос: «Почему вероятность получения 80 % и более исходов с орлами уменьшается при увеличении числа подбрасываний честной монеты?» Ответ сложно экстраполировать, поскольку два распределения имеют слабое наложение, что усложняет прямое визуальное сравнение. Возможно, нам удастся улучшить этот график, объединив распределения в их пиковых точках. Распределение A центрируется в точке пяти исходов орлов (из десяти подбрасываний монеты), а распределение B — в точке десяти исходов орлов (из 20). Если преобразовать число исходов орлов в их частоту (разделив на общее количество подбрасываний монеты), тогда пики обоих распределений выровняются по частоте 0,5. Это преобразование приведет также к выравниванию интервалов, охватывающих от 8 до 10 и от 16 до 20 исходов с орлами, в результате чего они оба будут лежать в общем интервале от 0,8 до 1,0. Код листинга 2.15 выполняет это преобразование и заново генерирует график (рис. 2.12).

Листинг 2.15. Преобразование числа исходов с орлами в их частоту

```
x_10_frequencies = [head_count /10 for head_count in x_10_flips]
x_20_frequencies = [head_count /20 for head_count in x_20_flips]
```

```
plt.plot(x_10_frequencies, prob_x_10_flips, label='A: 10 coin-flips')
plt.plot(x_20_frequencies, prob_x_20_flips, color='k', linestyle=':',
    label='B: 20 coin-flips')
plt.legend()

plt.xlabel('Head-Frequency')
plt.ylabel('Probability')
plt.show()
```

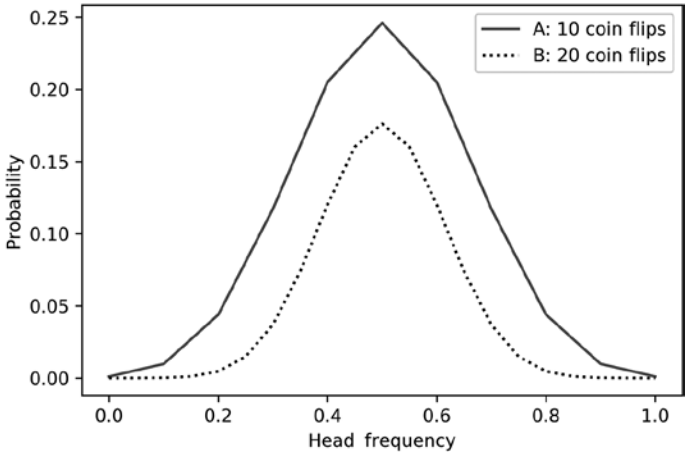


Рис. 2.12. Частота выпадения разных количеств орлов в результате 10 (A) и 20 (B) подбрасываний монеты, отображенная относительно ее вероятности. Оба пика по оси *Y* выравниваются по частоте 0,5. Площадь A полностью охватывает площадь B, поскольку общая площадь каждого графика больше не суммируется в 1,0

Как и ожидалось, теперь оба пика выровнялись по частоте 0,5. Тем не менее деление на количество орлов привело к сокращению областей под этими двумя кривыми в 10 и 20 раз соответственно. Теперь общая площадь под каждой из них уже не равна 1,0, и это проблема — как говорилось ранее, если нужно вывести вероятность интервалов, то общая площадь под кривой должна суммироваться в 1,0. Однако это можно исправить, если умножить значения оси *Y* кривых A и B на 10 и 20 соответственно (листинг 2.16). Скорректированные значения y больше не будут означать вероятности, значит, их надо назвать иначе. Подходящим термином здесь будет относительная вероятность, которая в математическом смысле отражает значение оси *Y* внутри кривой, чья общая площадь равна 1,0. Следовательно, мы назовем новые переменные оси *Y* `relative_likelihood_10` и `relative_likelihood_20`.

Листинг 2.16. Вычисление относительных вероятностей частот

```
relative_likelihood_10 = [10 * prob for prob in prob_x_10_flips]
relative_likelihood_20 = [20 * prob for prob in prob_x_20_flips]
```

Преобразование выполнено. Пора построить две новые кривые, попутно выделив интервалы, связанные с логическими массивами `where_10` и `where_20` (листинг 2.17; рис. 2.13).

Листинг 2.17. Построение выровненных кривых относительной вероятности

```
plt.plot(x_10_frequencies, relative_likelihood_10, label='A: 10 coin-flips')
plt.plot(x_20_frequencies, relative_likelihood_20, color='k',
         linestyle=':', label='B: 20 coin-flips')

plt.fill_between(x_10_frequencies, relative_likelihood_10, where=where_10)
plt.fill_between(x_20_frequencies, relative_likelihood_20, where=where_20)

plt.legend()
plt.xlabel('Head-Frequency')
plt.ylabel('Relative Likelihood')
plt.show()
```

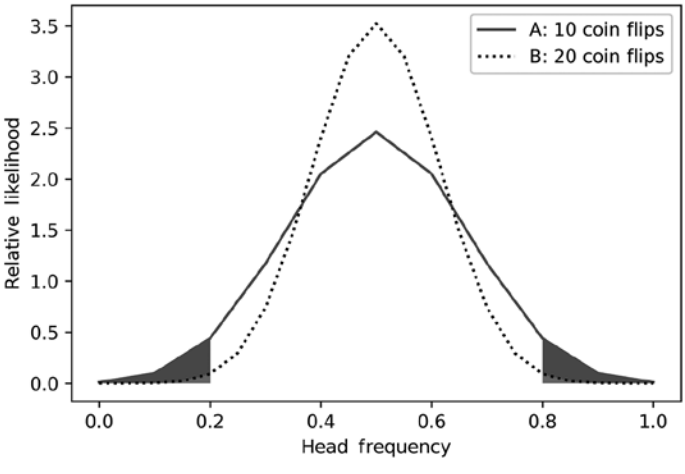


Рис. 2.13. Частота исходов с орлом для 10 (A) и 20 (B) подбрасываний монеты в сопоставлении с ее относительной вероятностью. Затененные интервалы под обоими графиками представляют экстремальные количества исходов с орлами и решками. Площади этих интервалов соответствуют вероятностям, поскольку общая площадь каждого графика суммируется в 1,0

На этом графике кривая A напоминает низкого, но широкоплечего атлета, а кривую B можно сравнить с более высоким и худым человеком. Поскольку A шире, ее область, охватывающая экстремальные интервалы частот исходов с орлами, больше. В связи с этим вероятность получить исходы с этими частотами выше при десяти подбрасываниях монеты, чем при 20. При этом более узкая и вертикальная кривая B охватывает больший участок вокруг центральной частоты 0,5.

А если подбросить монету более 20 раз, как это скажется на распределении частот исходов? Согласно теории вероятности каждое дополнительное подбрасывание монеты будет делать кривую еще выше и тоньше (рис. 2.14). Кривая будет изменяться подобно резинке, вытягиваемой вертикально вверх, утрачивая ширину в обмен на высоту. По мере того как общее число подбрасываний монеты будет достигать миллионов и миллиардов, кривая полностью лишится поперечного размера, превратившись в один сплошной вертикальный пик, чей центр привязан к частоте исходов с орлами 0,5. Вне этой частоты несуществующая область под вертикальной линией будет стремиться к нулю. Из этого следует, что область под пиком будет, наоборот, стремиться к 1,0, поскольку общая площадь всегда должна равняться 1,0. Площадь 1,0 соответствует вероятности 1,0. Таким образом, по мере приближения количества подбрасываний монеты к бесконечности частота исходов с орлами станет равна фактической вероятности получения орлов с абсолютной достоверностью.

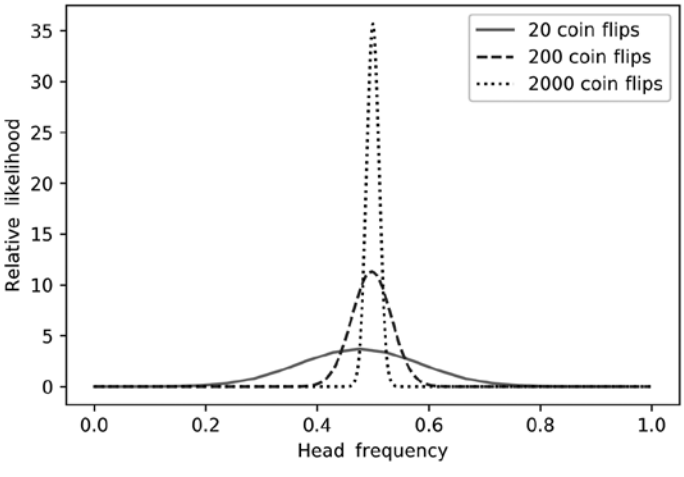


Рис. 2.14. Гипотетические частоты исходов с орлом при возрастании количества подбрасываний монеты. Все пики оси Y выравниваются в точке частоты 0,5, становясь все выше и у́же по мере увеличения числа подбрасываний. При 2000 подбрасываний суженная область пика практически полностью центрируется в точке 0,5. При бесконечном числе подбрасываний получающийся пик вытянется в одну вертикальную линию, идеально центрированную по частоте 0,5

Связь между бесконечным количеством подбрасываний монеты и абсолютной достоверностью утверждается фундаментальной теоремой, относящейся к теории вероятности, — *законом больших чисел*. Согласно этому закону, когда число наблюдений становится очень большим, частота наблюдения становится практически неотличимой от вероятности этого наблюдения. Следовательно, при достаточном числе подбрасываний монеты частота исходов орлов будет равняться фактической

вероятности получения такого исхода, которая равна 0,5. Помимо подбрасывания монеты, этот закон можно применить и к более сложным явлениям, например карточным играм. Если выполнить достаточно симуляций игры в карты, то частота побед будет равна фактической вероятности победы.

В следующей главе вы увидите, как закон больших чисел можно совместить со случайными симуляциями для аппроксимации вероятностей. В итоге мы выполним симуляции для определения вероятностей случайно вытаскиваемых карт. Как показывает закон больших чисел, эти симуляции должны выполняться в очень больших, вычислительно затратных масштабах. Поэтому для реализации эффективной симуляции нам потребуется познакомиться с библиотекой численных вычислений NumPy, которая рассматривается в главе 3.

РЕЗЮМЕ

- Графически отображая каждый возможный численный исход относительно его вероятности, мы генерируем распределение вероятностей. Общая площадь под распределением вероятности суммируется в 1,0. Область под определенным интервалом этого распределения равняется вероятности получения некоего значения в рамках этого интервала.
- Значения по оси *Y* распределения вероятностей не обязательно должны равняться вероятностям, если площадь графика будет равна 1,0.
- Распределение вероятностей комбинаций исходов подбрасываний честной монеты напоминает симметричную кривую. Число исходов с орлами можно преобразовать в их частоту. Во время этого преобразования можно сохранить равенство общей площади 1,0, преобразовав вероятности по оси *Y* в относительные вероятности. Пик преобразованной кривой центрируется в частоте 0,5. Если число подбрасываний монеты возрастает, пик также поднимается, а кривая по сторонам все больше сужается.
- Согласно *закону больших чисел*, по мере существенного увеличения общего числа наблюдений частота любого наблюдения будет приближаться к его вероятности. Таким образом, при возрастании количества подбрасываний честной монеты распределение их исходов будет все больше приближаться к центральной частоте 0,5.

3 Выполнение случайных симуляций в NumPy

В этой главе

- ✓ Базовое использование библиотеки NumPy.
- ✓ Симуляция случайных наблюдений с помощью NumPy.
- ✓ Визуализация симулированных данных.
- ✓ Приблизительная оценка неизвестных вероятностей в симулированных наблюдениях.

NumPy, или Numerical Python, — это механизм, на котором реализуется работа с данными в Python. Несмотря на множество достоинств, Python просто не подходит для обширного численного анализа. По этой причине аналитикам данных для эффективной работы с данными и их хранения приходится задействовать внешнюю библиотеку. NumPy — невероятно мощный инструмент для обработки больших коллекций сырых чисел, поэтому работу с ней поддерживают многие другие внешние библиотеки Python для работы с данными. Одной из них является Matplotlib, с которой мы познакомились в предыдущей главе. Прочие связанные с NumPy библиотеки будут рассмотрены в этой книге позже. Текущая глава посвящена созданию рандомизированных численных симуляций. Здесь мы с помощью NumPy будем анализировать миллиарды случайных точек данных, чтобы определить скрытые вероятности.

3.1. СИМУЛИРОВАНИЕ СЛУЧАЙНЫХ ПОДБРАСЫВАНИЙ МОНЕТЫ И БРОСКОВ КУБИКА С ПОМОЩЬЮ NUMPY

NumPy уже должна быть установлена в вашу рабочую среду в качестве одного из необходимых Matplotlib компонентов. По общему соглашению использования NumPy она импортируется как `np` (листинг 3.1).

ПРИМЕЧАНИЕ

Эту библиотеку можно установить и независимо от Matplotlib, выполнив pip install numpy из терминала.

Листинг 3.1. Импорт NumPy

```
import numpy as np
```

После импортирования можно начать выполнять случайные симуляции, используя модуль `np.random`. Этот модуль полезен для генерации случайных значений и симуляции случайных процессов. К примеру, вызов `np.random.randint(1, 7)` сгенерирует случайное целое число между 1 и 6 (листинг 3.2). Этот метод с равной вероятностью выбирает одно из шести возможных целых чисел, симулируя таким образом один бросок стандартного кубика.

Листинг 3.2. Симуляция случайного броска кубика

```
die_roll = np.random.randint(1, 7)
assert 1 <= die_roll <= 6
```

Сгенерированное значение `die_roll` является случайным, поэтому присваиваемая ему величина у разных читателей будет различаться. Эта несогласованность усложняет идеальное воссоздание определенного числа симуляций в этой главе. Нам же необходимо сделать так, чтобы все наши случайные результаты вы могли воссоздать у себя. Для этой цели можно использовать удобный метод `np.random.seed(0)`. Его вызов позволяет воссоздать последовательности случайно генерируемых значений. После этого вызова можно быть уверенными, что три броска кубика дадут 5, 6 и 1 (листинг 3.3).

Листинг 3.3. Определение начального числа для получения воспроизводимых результатов бросков кубика

```
np.random.seed(0)
die_rolls = [np.random.randint(1, 7) for _ in range(3)]
assert die_rolls == [5, 6, 1]
```

Изменение входного `x` в `np.random.randint(0, x)` позволяет симулировать любое число отдельных результатов. К примеру, установка `x` равным `52` приведет к симуляции случайно взятой из колоды карты. Давайте сгенерируем подбрасывание

монеты вызовом `np.random.randint(0, 2)` (листинг 3.4). Этот вызов метода возвращает случайное значение, равное `0` или `1`. Предположим, что `0` означает решку, а `1` — орла.

Листинг 3.4. Симуляция одного подбрасывания честной монеты

```
np.random.seed(0)
coin_flip = np.random.randint(0, 2)
print(f"Coin landed on {'heads' if coin_flip == 1 else 'tails'}")

Coin landed on tails
```

Далее мы симулируем последовательность из десяти подбрасываний монеты и вычислим частоту исходов с орлом (листинг 3.5).

Листинг 3.5. Симуляция десяти подбрасываний честной монеты

```
np.random.seed(0)
def frequency_heads(coin_flip_sequence):
    total_heads = len([head for head in coin_flip_sequence if head == 1])
    return total_heads / len(coin_flip_sequence)

coin_flips = [np.random.randint(0, 2) for _ in range(10)]
freq_heads = frequency_heads(coin_flips)
print(f"Frequency of Heads is {freq_heads}")

Frequency of Heads is 0.8
```

Заметьте, что можно более эффективно вычислить число орлов, выполнив sum(coin_flip_sequence)

Мы наблюдаем частоту 0,8, которая не особо пропорциональна фактической вероятности выпадения орла. Однако, как мы уже знаем, десять подбрасываний монеты будут давать подобную экстремальную частоту примерно в 10 % случаев. Для оценки же фактической вероятности необходимо сделать больше подбрасываний.

Далее посмотрим, что происходит при подбрасывании монеты 1000 раз. После каждого подбрасывания мы регистрируем общую частоту исходов с орлом, получаемых в этой последовательности. После завершения визуализируем результат, отобразив график подбрасываний монеты относительно частоты исходов с орлами (листинг 3.6; рис. 3.1). В этот график будет включена также горизонтальная линия вдоль фактической вероятности, равной 0,5. Ее мы генерируем вызовом `plt.axhline(0.5, color='k')`.

Листинг 3.6. Отрисовка частот исходов с орлом при симулированных подбрасываниях монеты

```
np.random.seed(0)
coin_flips = []
frequencies = []
for _ in range(1000):
    coin_flips.append(np.random.randint(0, 2))
    frequencies.append(frequency_heads(coin_flips))
```

```
plt.plot(list(range(1000)), frequencies)
plt.axhline(0.5, color='k')
plt.xlabel('Number of Coin Flips')
plt.ylabel('Head-Frequency')
plt.show()
```

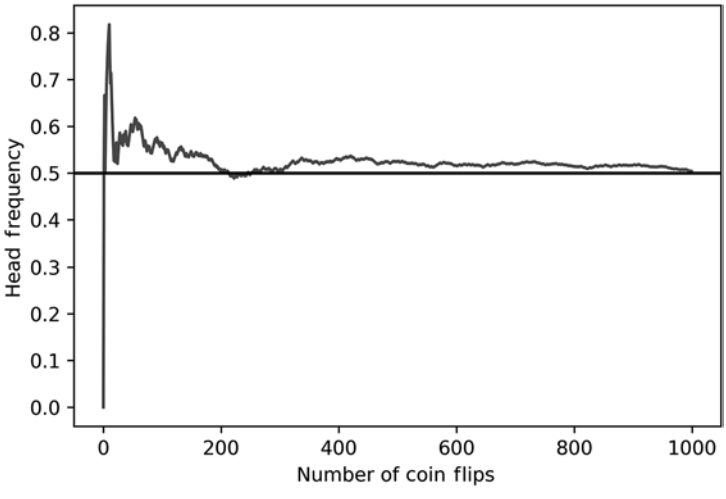


Рис. 3.1. Количество подбрасываний честной монеты в сопоставлении с частотой выпадения орла. Эта частота сильно колеблется, после чего устанавливается в районе 0,5

Вероятность получения орла медленно сходится к точке 0,5, что подтверждает закон больших чисел.

3.1.1. Анализ подбрасываний монеты со смещенным центром тяжести

Мы симулировали последовательность подбрасываний сбалансированной монеты, но что, если нам нужно симулировать такую монету, которая падает орлом вверх в 70 % случаев. Подобный результат можно сгенерировать вызовом `np.random.binomial(1, 0.7)` (листинг 3.7). Имя метода `binomial` означает общее распределение подбрасываний монеты, которое математики называют *биномиальным распределением*. Этот метод получает на входе два параметра: количество подбрасываний монеты и вероятность получения желаемого исхода. Далее он выполняет заданное число подбрасываний и подсчитывает случаи, в которых был получен желаемый исход. Когда количество подбрасываний установлено на `1`, метод возвращает двоичное значение `0` или `1`. В нашем случае значение `1` представляет желаемый результат получения орла.

Листинг 3.7. Симуляция подбрасывания монеты со смещением

```
np.random.seed(0)
print("Let's flip the biased coin once.")
coin_flip = np.random.binomial(1, 0.7)
print(f"Biased coin landed on {'heads' if coin_flip == 1 else 'tails'}.")

print("\nLet's flip the biased coin 10 times.")
number_coin_flips = 10
head_count = np.random.binomial(number_coin_flips, .7)
print((f"{head_count} heads were observed out of "
       f"{number_coin_flips} biased coin flips"))

Let's flip the biased coin once.
Biased coin landed on heads.

Let's flip the biased coin 10 times.
6 heads were observed out of 10 biased coin flips
```

Далее сгенерируем последовательность из 1000 подбрасываний монеты со смещенным центром тяжести, после чего проверим, будет ли частота исходов с орлом сходиться к 0,7.

Листинг 3.8. Вычисление схождения частоты исходов с орлом

```
np.random.seed(0)
head_count = np.random.binomial(1000, 0.7)
frequency = head_count / 1000
print(f"Frequency of Heads is {frequency}")

Frequency of Heads is 0.697
```

Частота получения орла аппроксимируется к 0,7, но фактически 0,7 не равна. В действительности ее значение на 0,003 единицы меньше, чем истинная вероятность получения орла. Предположим, что повторно вычисляем частоту исходов с орлом для 1000 подбрасываний еще пять раз. Будут ли все итоговые значения частоты меньше 0,7? Будут ли определенные частоты в точности соответствовать 0,7? Это мы выясним, выполнив `np.random.binomial(1000, 0.7)` для пяти циклических итераций (листинг 3.9).

Листинг 3.9. Повторное вычисление схождения частоты исходов с орлом

Напомню, что мы определили начальное значение в генераторе случайных чисел, чтобы поддерживать согласованный вывод. Поэтому псевдослучайная генерация исходов вернет ранее полученную частоту 0,697. Мы пропустим этот результат, сгенерировав пять свежих частот

```
np.random.seed(0)
assert np.random.binomial(1000, 0.7) / 1000 == 0.697
for i in range(1, 6):
    head_count = np.random.binomial(1000, 0.7)
    frequency = head_count / 1000
```

```
    print(f"Frequency at iteration {i} is {frequency}")
    if frequency == 0.7:
        print("Frequency equals the probability!\n")

Frequency at iteration 1 is 0.69
Frequency at iteration 2 is 0.7
Frequency equals the probability!

Frequency at iteration 3 is 0.707
Frequency at iteration 4 is 0.702
Frequency at iteration 5 is 0.699
```

Всего одна из пяти итераций дала значение, равное реальной вероятности. Дважды полученная частота оказалась чуть меньше и дважды — чуть больше. Итоговая частота исходов с орлом колеблется для каждой последовательности из 1000 подбрасываний монеты. Похоже, несмотря на то что закон больших чисел позволяет аппроксимировать фактическую вероятность, некоторая неоднозначность все же сохранятся. Наука о данных немного запутанна, и мы не можем всегда быть уверены в заключениях, которые строим относительно данных. Однако неуверенность можно измерить и ограничить с помощью математического принципа доверительного интервала.

3.2. ВЫЧИСЛЕНИЕ ДОВЕРИТЕЛЬНЫХ ИНТЕРВАЛОВ С ПОМОЩЬЮ ГИСТОГРАММ И МАССИВОВ NUMPY

Предположим, что нам дали монету с неизвестным смещением центра тяжести. Мы подбрасываем ее 1000 раз и получаем орла с частотой 0,709. Мы знаем, что эта частота аппроксимируется к фактической вероятности, но насколько? Говоря точнее, каковы шансы на то, что фактическая вероятность попадет в интервал, близкий к 0,709 (например, диапазон между 0,700 и 0,710)? Чтобы это выяснить, необходим дополнительный анализ.

До этого мы проанализировали результаты в ходе пяти итераций, состоявших из 1000 подбрасываний каждая. В итоге получили некоторые колебания частоты. Теперь же давайте подробнее разберем эти колебания, увеличив количество итераций с 5 до 500 (листинг 3.10). Для проведения дополнительного анализа выполняем `[np.random.binomial(1000, 0.7) for_in range(500)]`.

Листинг 3.10. Вычисление частот исходов с орлами в 500 итерациях

```
np.random.seed(0)
head_count_list = [np.random.binomial(1000, 0.7) for _ in range(500)]
```

Можно и более эффективно проанализировать 500 итераций, выполнив `np.random.binomial(coin_flip_count, 0.7, size=500)`. Необязательный параметр `size` позволяет выполнить `np.random.binomial(coin_flip_count, 0.7)` 500 раз, используя внутренние оптимизации NumPy (листинг 3.11).

Листинг 3.11. Оптимизация вычисления частоты при подбрасывании монеты

```
np.random.seed(0)
head_count_array = np.random.binomial(1000, 0.7, 500)
```

Результатом будет не список Python, а массив NumPy. Как уже говорилось, массивы NumPy могут более эффективно хранить численные данные. Фактические численные количества, хранящиеся в `head_count_array` и `head_count_list`, остаются прежними. Подтвердить это можно преобразованием данного массива в список с помощью метода `head_count_array.tolist()` (листинг 3.12).

Листинг 3.12. Преобразование массива NumPy в список Python

```
assert head_count_array.tolist() == head_count_list
```

И наоборот, можно преобразовать список Python в эквивалентный по значениям массив NumPy, вызвав `np.array(head_count_list)` (листинг 3.13). Равнозначность между преобразованным массивом и `head_count_array` можно подтвердить с помощью метода `np.array_equal`.

Листинг 3.13. Преобразование списка Python в массив NumPy

```
new_array = np.array(head_count_list)
assert np.array_equal(new_array, head_count_array) == True
```

Почему лучше использовать массив NumPy, а не стандартный список Python? Что ж, помимо уже упомянутой оптимизации потребления памяти и ускорения анализа, NumPy упрощает написание более чистого кода. К примеру, она предлагает более простые операции умножения и деления. Деление массива NumPy непосредственно на `x` ведет к созданию нового массива, все элементы которого разделены на `x`. Таким образом, выполнение `head_count_array/1000` автоматически преобразует количество исходов с орлом в их частоту (листинг 3.14). В противоположность этому вычисление частоты для `head_count_list` требует либо перебора всех элементов списка, либо использования функции Python `map`.

Листинг 3.14. Вычисление частот с помощью NumPy

```
frequency_array = head_count_array / 1000
assert frequency_array.tolist() == [head_count / 1000
                                    for head_count in head_count_list]
assert frequency_array.tolist() == list(map(lambda x: x / 1000,
                                        head_count_list))
```

ПОЛЕЗНЫЕ МЕТОДЫ NUMPY ДЛЯ ВЫПОЛНЕНИЯ СЛУЧАЙНЫХ СИМУЛЯЦИЙ

- `np.random.randint(x, y)` — возвращает случайное целое между `x` и `y-1` включительно.
- `np.random.binomial(1, p)` — возвращает одно случайное значение, равное 0 или 1. Вероятность того, что оно будет равно 1, составляет `p`.
- `np.random.binomial(x, p)` — выполняет `x` экземпляров `np.random.binomial(1, p)` и возвращает суммированный результат. Возвращенное значение представляет количество ненулевых исходов среди `x` повторений.
- `np.random.binomial(x, p, size=y)` — возвращает массив из `y` элементов. Каждый элемент равняется случайному результату `np.random.binomial(x, p)`.
- `np.random.binomial(x, p, size=y)/x` — возвращает массив из `y` элементов. Каждый элемент представляет частоту ненулевых исходов среди `x` повторений.

Мы преобразовали массив количеств исходов с орлом в массив их частот с помощью простой операции деления. Далее изучим содержимое `frequency_array` более подробно и начнем с вывода первых 20 вычисленных частот, используя тот же делитель индексов `:`, что и в списках Python (листинг 3.15). Заметьте, что, в отличие от списков, массив NumPy в своем выводе запятых не содержит.

Листинг 3.15. Вывод массива частот NumPy

```
print(frequency_array[:20])
```

```
[ 0.697  0.69   0.7    0.707  0.702  0.699  0.723  0.67   0.702  0.713
  0.721  0.689  0.711  0.697  0.717  0.691  0.731  0.697  0.722  0.728]
```

Значения полученных частот колеблются от 0,69 до приблизительно 0,731. Естественно, дополнительные 480 частот остаются во `frequency_array`. Давайте извлечем максимальное и минимальное значения массива, вызвав методы `frequency_array.min()` и `frequency_array.max()` (листинг 3.16).

Листинг 3.16. Поиск максимального и минимального значений частот

```
min_freq = frequency_array.min()
max_freq = frequency_array.max()
print(f"Minimum frequency observed: {min_freq}")
print(f"Maximum frequency observed: {max_freq}")
print(f"Difference across frequency range: {max_freq - min_freq}")

Minimum frequency observed: 0.656
Maximum frequency observed: 0.733
Difference across frequency range: 0.07699999999999996
```

Где-то в диапазоне частот от 0,656 до 0,733 находится истинная вероятность получения орла. Этот интервал довольно большой — он представляет разницу более 7 % между максимальным и минимальным значениями. Но мы можем сузить этот диапазон частот, отобразив все их уникальные значения относительно числа их вхождений (листинг 3.17; рис. 3.2).

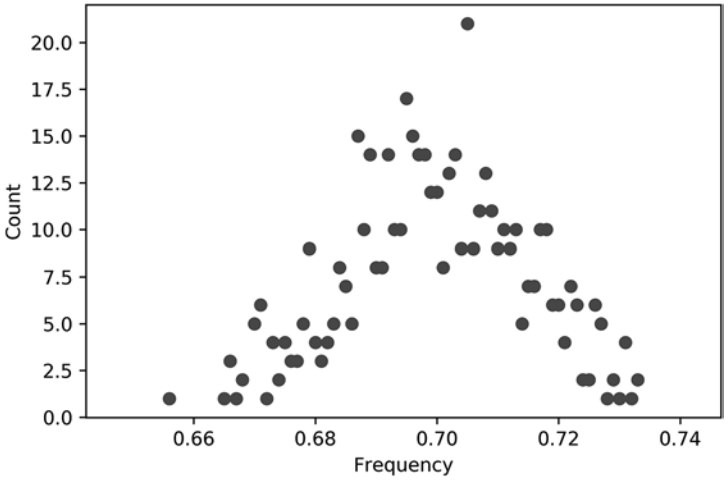


Рис. 3.2. Точечный график соотношения 500 частот исходов с орлом и количества этих частот. Все частоты группируются вокруг 0,7. Некоторые близкие частоты на графике накладываются друг на друга

Листинг 3.17. Отрисовка измеренных частот

```
frequency_counts = defaultdict(int)
for frequency in frequency_array:
    frequency_counts[frequency] += 1

frequencies = list(frequency_counts.keys())
counts = [frequency_counts[freq] for freq in frequencies]
plt.scatter(frequencies, counts)
plt.xlabel('Frequency')
plt.ylabel('Count')
plt.show()
```

Эта визуализация довольно информативна: частоты, близкие к 0,7, встречаются чаще других, более удаленных значений. Тем не менее этот график имеет изъян, поскольку почти идентичные частоты отражены на нем как накладывающиеся точки. Эти близкие частоты нам нужно сгруппировать, а не рассматривать как отдельные.

3.2.1. Сортировка схожих точек в столбчатых диаграммах

Далее выполним более детальную визуализацию, сгруппировав частоты, находящиеся очень близко друг к другу. Мы дополнительно разделим частотный диапазон на N равноудаленных друг от друга интервалов (бинов, bin), после чего распределим по ним все значения частот. По определению значения в любом интервале находятся друг от друга на расстоянии не более $1/N$. Далее подсчитаем общее количество значений в каждом интервале и визуализируем их на графике.

Основанный на интервалах график, который мы только что описали, называется *гистограммой*[1]. В Matplotlib гистограммы генерируются с помощью вызова `plt.hist`. Этот метод получает на входе последовательность значений для распределения по интервалам и дополнительный параметр `bins`, который устанавливает общее количество интервалов. Таким образом, вызов `plt.hist(frequency_array, bins=77)` приведет к разделению наших данных по 77 интервалам, каждый из которых будет охватывать ширину 0,01 единицы. В качестве альтернативы можно указать `bins=auto`, и Matplotlib сама выберет подходящую ширину интервалов, используя распространенную технику оптимизации (ее подробное рассмотрение выходит за рамки темы этой книги). На рис. 3.3 показана итоговая гистограмма с оптимизированной шириной интервалов, полученная вызовом `plt.hist(frequency_array, bins='auto')`.

ПРИМЕЧАНИЕ

В листинге 3.18 мы также включили параметр edgecolor='black'. Это помогает визуально различать интервалы за счет выделения их границ черным цветом.

Листинг 3.18. Построение гистограммы частот с помощью plt.hist

```
plt.hist(frequency_array, bins='auto', edgecolor='black')
plt.xlabel('Binned Frequency')
plt.ylabel('Count')
plt.show()
```

В полученной гистограмме интервал с максимальным числом частот оказался между 0,69 и 0,70. Он заметно возвышается примерно над десятком других. Более точное количество интервалов можно получить с помощью `counts`, являющегося массивом NumPy, возвращаемым `plt.hist`. Этот массив содержит отражаемые на оси Y количества частот для каждой группы интервалов. Давайте вызовем `plt.hist`

[1] Гистограмма строится следующим образом. Сначала множество значений, которое может принимать элемент выборки, разбивается на несколько интервалов. Эти интервалы откладываются на горизонтальной оси, затем над каждым рисуется прямоугольник. Если все интервалы были одинаковыми, то высота каждого прямоугольника пропорциональна числу элементов выборки, попадающих в соответствующий интервал. Если интервалы разные, то высота прямоугольника выбирается таким образом, чтобы его площадь была пропорциональна числу элементов выборки, которые попали в этот интервал. — *Примеч. ред.*

для получения `counts` и последующего обращения к `counts.size` для выяснения общего числа групп интервалов (листинг 3.19).

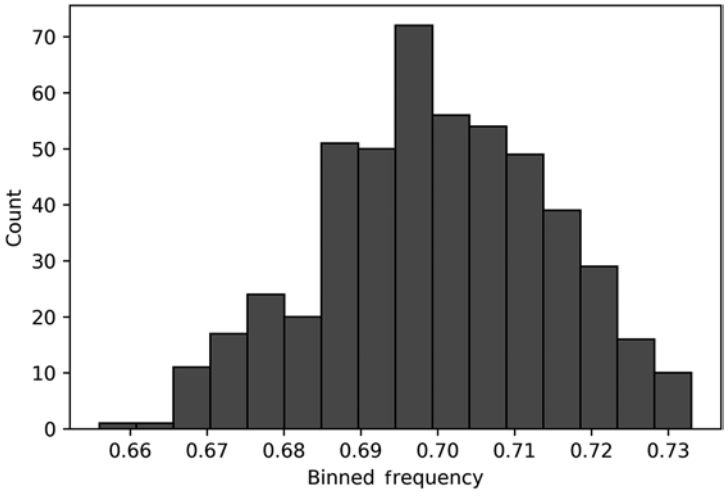


Рис. 3.3. Гистограмма соотношения 500 частот, распределенных по интервалам, и количества элементов в каждом интервале. Интервал с наибольшим числом элементов центрирован около частоты 0,7

Листинг 3.19. Подсчет интервалов в гистограмме

```
counts, _, _ = plt.hist(frequency_array, bins='auto',
                        edgecolor='black')

print(f"Number of Bins: {counts.size}")
```

counts — это одна из трех переменных, возвращаемых plt.hist. Остальные переменные будут рассмотрены в этой главе далее

```
Number of Bins: 16
```

Всего в гистограмме получилось 16 интервалов. Насколько каждый из них широк? Это можно выяснить, разделив общий диапазон частот на 16 (листинг 3.20). В качестве альтернативы можно использовать массив `bin_edges`, являющийся второй переменной, возвращаемой `ply.hist`. Он содержит позиции вертикальных границ интервалов по оси *X*, значит, разница между любыми двумя последовательными позициями границ равна ширине интервала.

Листинг 3.20. Определение ширины интервалов на гистограмме

```
counts, bin_edges, _ = plt.hist(frequency_array, bins='auto',
                                edgecolor='black')

bin_width = bin_edges[1] - bin_edges[0]
assert bin_width == (max_freq - min_freq) / counts.size
print(f"Bin width: {bin_width}")
```

```
Bin width: 0.004812499999999997
```

ПРИМЕЧАНИЕ

Размер bin_edges всегда на один больше размера counts. Почему? Представьте, что у нас всего один прямоугольник, соответствующий интервалу: он будет ограничен двумя вертикальными линиями. Добавление дополнительного интервала увеличит общее число таких границ на 1. Если экстраполировать эту логику на N интервалов, то можно ожидать получения N + 1 линий границ.

Массив `bin_edges` можно использовать в сочетании с `counts` для вывода числа элементов и диапазона охвата любого указанного интервала. Определим функцию `output_bin_coverage`, выводящую количество и охват любого интервала в позиции `i` (листинг 3.21).

Листинг 3.21. Получение диапазона частот в интервале и его размера

```
def output_bin_coverage(i):
    count = int(counts[i])
    range_start, range_end = bin_edges[i], bin_edges[i+1]
    range_string = f"{range_start} – {range_end}"
    print((f"The bin for frequency range {range_string} contains "
           f"{count} element{'' if count == 1 else 's'}"))

output_bin_coverage(0)
output_bin_coverage(5)
```

Интервал в позиции i содержит counts[i] частот

Интервал в позиции i охватывает диапазон частот от bin_edges[i] до bin_edges[i+1]

```
The bin for frequency range 0.656 – 0.6608125 contains 1 element
The bin for frequency range 0.6800625 – 0.684875 contains 20 elements
```

Теперь вычислим количество частот и их диапазон в наивысшем пике нашей гистограммы. Для этого потребуется индекс `counts.max()`. Удобно то, что в массивах NumPy есть встроенный метод `argmax`, который возвращает индекс максимального значения в массиве (листинг 3.22).

Листинг 3.22. Поиск индекса максимального значения в массиве

```
assert counts[counts.argmax()] == counts.max()
```

В результате вызов `output_bin_coverage(counts.argmax())` даст нам запрошенный вывод (листинг 3.23).

Листинг 3.23. Использование argmax для возвращения характеристик пика гистограммы

```
output_bin_coverage(counts.argmax())
```

```
The bin for frequency range 0.6945 – 0.6993125 contains 72 elements
```

3.2.2. Получение вероятностей из гистограмм

Наиболее заполненный интервал гистограммы содержит 72 элемента и охватывает диапазон частот приблизительно от 0,694 до 0,699. Как можно определить, попадает ли фактическая вероятность получения орлов в этот диапазон (не зная

ответа наперед)? Один из вариантов — вычислить вероятность того, что случайно измеренная частота попадает в диапазон от 0,694 до 0,699. Если эта вероятность окажется 1, значит, в данный диапазон будет попадать 100 % измеренных частот. Эти частоты будут иногда включать фактическую вероятность получения орла, и мы будем абсолютно уверены в том, что истинная вероятность лежит где-то между 0,694 и 0,699. И даже если эта вероятность окажется меньше, например 95 %, мы все равно будем практически уверены, что этот диапазон включает значение истинной вероятности.

Как можно вычислить эту вероятность? Ранее мы выяснили, что вероятность попадания в интервал равна его площади под кривой, но только когда общая отрисованная область суммируется в 1. Площадь под нашей гистограммой больше 1, а значит, требует изменения путем передачи `density=True` в `plt.hist`. Этот передаваемый параметр сохраняет форму гистограммы, но при этом делает так, чтобы ее площадь суммировалась в 1,0 (листинг 3.24).

Листинг 3.24. Отрисовка относительных вероятностей гистограммы

```
likelihoods, bin_edges, _ = plt.hist(frequency_array, bins='auto',
                                     edgecolor='black', density=True)
plt.xlabel('Binned Frequency')
plt.ylabel('Relative Likelihood')
plt.show()
```

Теперь количества интервалов оказались заменены относительными вероятностями, которые хранятся в массиве `likelihoods` (рис. 3.4). Как уже говорилось, относительная вероятность — это термин, применяемый к значениям оси *Y* графика, площадь которого суммируется в 1. Естественно, площадь под нашей гистограммой теперь составляет 1. Подтвердить это можно, суммировав площади прямоугольников всех интервалов, каждая из которых равняется значению вертикальной вероятности интервала, умноженному на `bin_width`. Из этого следует, что площадь под гистограммой равна сумме вероятностей, умноженной на `bin_width`. Таким образом, вызов `likelihoods.sum() * bin_width` должен возвращать `1`.

ПРИМЕЧАНИЕ

Общая площадь равняется сумме площадей прямоугольников гистограммы. На рис. 3.4 размер самых длинных прямоугольников довольно велик, поэтому мы визуально оцениваем общую площадь как превышающую 1,0 (листинг 3.25).

Листинг 3.25. Вычисление общей площади под гистограммой

```
assert likelihoods.sum() * bin_width == 1.0
```

Общая площадь гистограммы суммируется в 1. Таким образом, площадь под ее пиком теперь равняется вероятности того, что случайно измеренная частота попадет в диапазон от 0,694 до 0,699. Давайте вычислим это значение, определив площадь интервала, расположенного в `likelihoods.argmax()` (листинг 3.26).

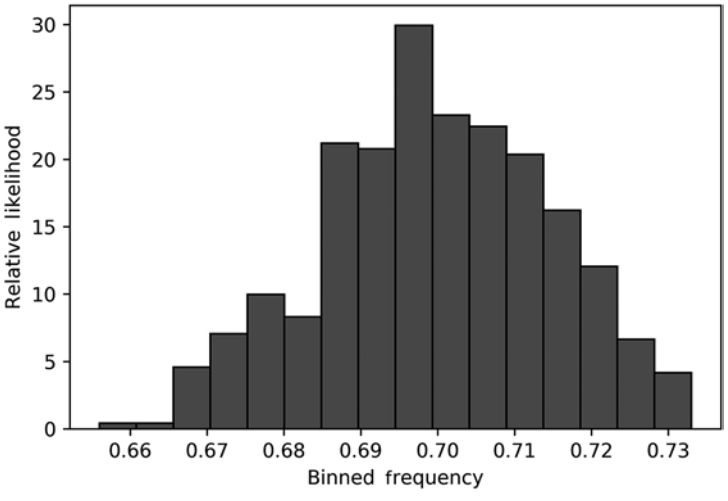


Рис. 3.4. Гистограмма соотношения 500 частот, распределенных по интервалам, и относительных вероятностей. Площадь гистограммы суммируется в 1,0. Вычислить ее можно, сложив площади прямоугольников всех интервалов

Листинг 3.26. Вычисление вероятности получения частот пика

```
index = likelihoods.argmax()
area = likelihoods[index] * bin_width
range_start, range_end = bin_edges[index], bin_edges[index+1]
range_string = f"{range_start} – {range_end}"
print(f"Sampled frequency falls within interval {range_string} with
        probability {area}")

Sampled frequency falls within interval 0.6945 – 0.6993125 with probability 0.144
```

Вероятность составляет примерно 14 %. Это низкое значение, но его можно увеличить, расширив диапазон нашего интервала за границы одного интервала (листинг 3.27). Раздвинем диапазон так, чтобы он охватывал соседние интервалы в индексах `likelihoods.argmax() – 1` и `likelihoods.argmax() + 1`.

ПРИМЕЧАНИЕ

Напомню, что нотация индексов в Python включает стартовый индекс и исключает конечный. В связи с этим мы устанавливаем конечный индекс как равный likelihoods.argmax() + 2, чтобы включить likelihoods.argmax() + 1.

Листинг 3.27. Увеличение вероятности попадания в диапазон частот

```
peak_index = likelihoods.argmax()
start_index, end_index = (peak_index – 1, peak_index + 2)
area = likelihoods[start_index: end_index + 1].sum() * bin_width
```

```
range_start, range_end = bin_edges[start_index], bin_edges[end_index]
range_string = f"{range_start} – {range_end}"
print(f"Sampled frequency falls within interval {range_string} with
        probability {area}")

Sampled frequency falls within interval 0.6896875 – 0.704125 with probability 0.464
```

Эти три интервала охватывают диапазон частот примерно от 0,689 до 0,704. Связанная с ними вероятность составляет 0,464. Таким образом, они представляют то, что статистики называют *доверительным интервалом*, который в данном случае равен 46,4 %. Это значение подразумевает 46,4 % уверенности, что наша истинная вероятность попадет в диапазон этих трех интервалов. Это довольно низкий процент уверенности. Статистики предпочитают доверительные интервалы 95 % и более. И мы достигнем такой его величины, итеративно расширяя диапазон интервалов влево и вправо, пока площадь доверительного интервала не выйдет за 0,95 (листинг 3.28).

Листинг 3.28. Вычисление высокого доверительного интервала

```
def compute_high_confidence_interval(likelihoods, bin_width):
    peak_index = likelihoods.argmax()
    area = likelihoods[peak_index] * bin_width
    start_index, end_index = peak_index, peak_index + 1
    while area < 0.95:
        if start_index > 0:
            start_index -= 1
        if end_index < likelihoods.size – 1:
            end_index += 1

        area = likelihoods[start_index: end_index + 1].sum() * bin_width

    range_start, range_end = bin_edges[start_index], bin_edges[end_index]
    range_string = f"{range_start:.6f} – {range_end:.6f}"
    print((f"The frequency range {range_string} represents a "
           f"{100 * area:.2f} % confidence interval"))
    return start_index, end_index

compute_high_confidence_interval(likelihoods, bin_width)

The frequency range 0.670438 – 0.723375 represents a 95.40 % confidence interval
```

Диапазон частот от 0,670 до 0,723 представляет доверительный интервал 95,4 %. Таким образом, проанализированная последовательность из 1000 подбрасываний монеты со смещенным центром тяжести должна попадать в этот диапазон в 95,4 % случаев. Мы искренне уверены в том, что истинная вероятность лежит где-то между 0,670 и 0,723. Тем не менее все еще не можем наверняка сказать, будет ли эта вероятность ближе к 0,67 или к 0,72. Необходимо как-то сузить этот диапазон, чтобы получить более информативную оценку вероятности.

3.2.3. Сужение диапазона высокого доверительного интервала

Как же можно сузить этот диапазон, сохранив величину доверительного интервала 95 %? Думаю, можно попробовать увеличить количество частот с 500 до заметно большего значения. Ранее мы анализировали 500 частот, каждая из которых отражала 1000 подбрасываний монеты со смещенным центром тяжести. Теперь же вычислим 100 000 значений частот, также сделав по 1000 подбрасываний монеты для вычисления каждой (листинг 3.29).

Листинг 3.29. Вычисление 100 000 частот

```
np.random.seed(0)
head_count_array = np.random.binomial(1000, 0.7, 100000)
frequency_array = head_count_array / 1000
assert frequency_array.size == 100000
```

Мы повторно вычисляем гистограмму для обновленного массива `frequency_array`, который теперь содержит в 200 раз больше значений частоты. Далее визуализируем эту гистограмму и находим высокий доверительный интервал. После этого встраиваем данный интервал в визуализацию, закрашивая столбцы, попадающие в его диапазон (рис. 3.5).

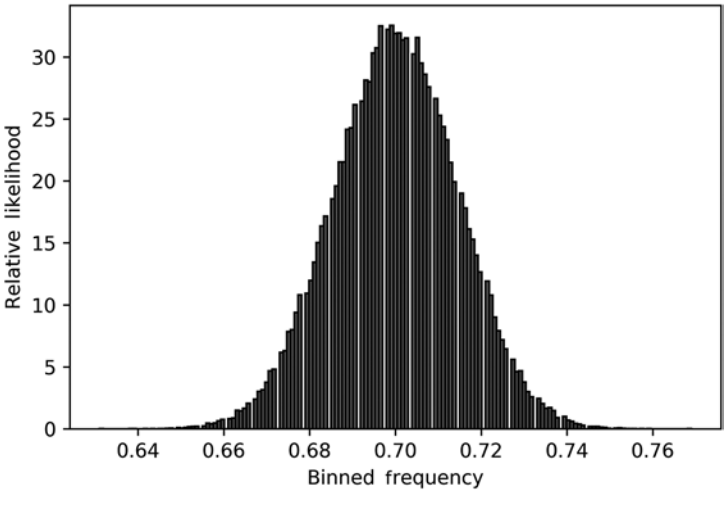


Рис. 3.5. Гистограмма из 100 000 частот, распределенных по интервалам, сопоставленных с их относительными вероятностями. Выделенные столбцы обозначают доверительный интервал 95 % от всей площади гистограммы. Он охватывает диапазон частот примерно от 0,670 до 0,727

Визуальное представление столбцов можно менять с помощью `patches` — третьей переменной, возвращаемой `plt.hist`. Графические детали каждого столбца в интервале по индексу `i` можно получить через `patches[i]`. Если нужно закрасить столбец в интервале по индексу `i` желтым, следует просто вызвать `patches[i].set_facecolor('yellow')` (листинг 3.30). Таким образом можно выделять все заданные столбцы гистограммы, попадающие в диапазон обновленного интервала.

Листинг 3.30. Закрашивание столбцов гистограммы в интервале

```
likelihoods, bin_edges, patches = plt.hist(frequency_array, bins='auto',
                                           edgecolor='black', density=True)
bin_width = bin_edges[1] – bin_edges[0]
start_index, end_index = compute_high_confidence_interval(likelihoods,
                                                          bin_width)

for i in range(start_index, end_index):
    patches[i].set_facecolor('yellow')
plt.xlabel('Binned Frequency')
plt.ylabel('Relative Likelihood')

plt.show()

The frequency range 0.670429 – 0.727857 represents a 95.42 % confidence interval
```

Повторная гистограмма имеет симметричную форму, похожую на колокол. Многие ее столбцы мы выделили с помощью метода `set_facecolor`. Они представляют доверительный интервал 95 %, охватывающий диапазон частот примерно от 0,670 до 0,727. Этот новый диапазон практически идентичен прежнему, увеличение количества анализируемых частот его не уменьшило. Возможно, стоит также увеличить число подбрасываний монеты с 1000 до 50 000 (рис. 3.6). При этом мы сохраним количество частот — 100 000, получив теперь 5 млрд подбрасываний монеты (листинг 3.31).

Листинг 3.31. Анализ 5 млрд подбрасываний монеты

```
np.random.seed(0)
head_count_array = np.random.binomial(50000, 0.7, 100000)
frequency_array = head_count_array / 50000

likelihoods, bin_edges, patches = plt.hist(frequency_array, bins='auto',
                                           edgecolor='black', density=True)
bin_width = bin_edges[1] – bin_edges[0]
start_index, end_index = compute_high_confidence_interval(likelihoods,
                                                          bin_width)

for i in range(start_index, end_index):
    patches[i].set_facecolor('yellow')
```

```
plt.xlabel('Binned Frequency')
plt.ylabel('Relative Likelihood')

plt.show()

The frequency range 0.695769 — 0.703708 represents a 95.06 % confidence interval
```

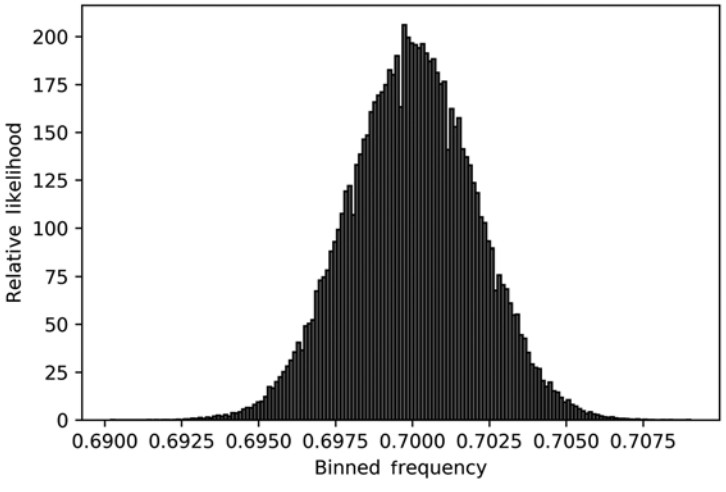


Рис. 3.6. Гистограмма из 100 000 частот, распределенных по интервалам, сопоставленных с их относительными вероятностями. Выделенные столбцы отражают доверительный интервал 95 %, охватывающий 95 % площади гистограммы. Этот интервал включает диапазон частот примерно от 0,695 до 0,703

Новый доверительный интервал охватывает диапазон приблизительно от 0,695 до 0,703. Если округлить его до двух десятичных знаков, то получится от 0,7 до 0,7. Таким образом, мы получили исключительную уверенность в том, что истинная вероятность равна приблизительно 0,7. Увеличивая число подбрасываний монеты для каждого вычисления частоты, мы успешно сузили диапазон доверительного интервала до 95 %.

Помимо этого, кривая обновленной гистограммы снова напоминает колокол. Называют эту кривую *гауссовым распределением*, или *нормальным распределением*. Нормальное распределение играет очень важную роль в теории вероятности и статистике в контексте центральной *предельной теоремы*. Согласно этой теореме, при достаточно большом количестве измерений распределения полученных частот принимают форму нормального распределения. Более того, эта теорема прогнозирует сужение вероятных частот по мере роста набора данных, анализируемого для получения каждой из них. Это полностью согласуется с нашими наблюдениями, которые можно обобщить так.

1. Изначальный анализ 1000 подбрасываний монеты 500 раз.
2. Преобразование каждой последовательности из 1000 подбрасываний монеты в частоту.
3. Составление гистограммы из 500 частот, представляющих 50 000 подбрасываний монеты в целом.
4. Форма гистограммы получилась несимметричной. Ее пик находился примерно в точке 0,7.
5. Увеличение количества вычисляемых частот с 500 до 100 000.
6. Отображение на графике 100 000 частот, представляющих 1 млн подбрасываний монеты в целом.
7. Новая гистограмма приобрела форму нормальной кривой, пик которой по-прежнему находился в точке 0,7.
8. Сложение прямоугольных областей в интервале вокруг пика до тех пор, пока их сумма не стала равна 95 % от общей площади под гистограммой.
9. Эти интервалы охватили диапазон частот приблизительно от 0,670 до 0,723.
10. Увеличение числа подбрасываний монеты для каждого измерения с 1000 до 50 000.
11. Отображение гистограммы из 100 000 частот, представляющих 5 млрд подбрасываний монеты в целом.
12. Форма обновленной гистограммы по-прежнему напоминает нормальную кривую.
13. Повторное вычисление диапазона, охватывающего 95 % площади гистограммы.
14. Ширина этого диапазона уменьшилась и теперь составляет примерно от 0,695 до 0,703.
15. Таким образом, за счет увеличения количества подбрасываний для вычисления каждой частоты удалось добиться сужения диапазона вероятных частот примерно до 0,7.

3.2.4. Вычисление гистограмм в NumPy

Вызов метода `plt.hist` автоматически генерирует гистограмму. Но можно ли получить ее значения вероятностей и границ интервалов без построения самого графика? Да, поскольку `plt.hist` использует не визуальную функцию NumPy `np.histogram`. Эта функция получает на входе все параметры, не связанные с визуализацией гистограммы, такие как `frequency_arrays`, `bins='auto'` и `density=True`, возвращая в ответ две переменные, не относящиеся к управлению графиком: `likelihoods` и `bin_edges`. Следовательно, можно выполнить `compute_high_confidence_interval` без применения визуализации, просто вызвав `np.histogram` (листинг 3.32).

Листинг 3.32. Вычисление гистограммы с помощью np.histogram

```
np.random.seed(0)
```

Мы больше не храним переменные начального и конечного индексов, возвращаемые этой функцией, поскольку нет нужды выделять на графике диапазон интервала

```
likelihoods, bin_edges = np.histogram(frequency_array, bins='auto',
                                      density=True)
bin_width = bin_edges[1] – bin_edges[0]
compute_high_confidence_interval(likelihoods, bin_width)
```

```
The frequency range 0.695769 – 0.703708 represents a 95.06 % confidence interval
```

ПОЛЕЗНЫЕ ФУНКЦИИ ГИСТОГРАММЫ

- `plt.hist(data, bins=10)` — строит гистограмму, на которой элементы `data` распределены по десяти интервалам одинаковой ширины.
- `plt.hist(data, bins='auto')` — строит гистограмму, количество интервалов которой определяется автоматически на основе распределения данных. `auto` — это предустановленная настройка для `_bins`.
- `plt.hist(data, edges='black')` — в создаваемых гистограммах границы каждого интервала выделяются черными вертикальными линиями.
- `counts, _, _ = plt.hist(data)` — массив `counts` — это первая из трех переменных, возвращаемых `plt.hist`. Он содержит количество элементов, находящихся в каждом интервале. Эти количества отражаются на оси *y* графика.
- `_, bin_edges, _ = plt.hist(data)` — массив `bin_edges` — это вторая из трех переменных, возвращаемых `plt.hist`. Он содержит позиции вертикальных границ интервалов по оси *X* графика. Вычитание `bin_edges[i]` из `bin_edges[i + 1]` приводит к возвращению ширины каждого интервала. Умножение этой ширины на `counts[i]` приводит к возвращению площади прямоугольника в интервале в позиции `i`.
- `likelihoods, _, _ = plt.hist(data, density=True)` — эти количества преобразуются в вероятности, чтобы площадь под гистограммой суммировалась в 1,0. Таким образом, гистограмма преобразуется в распределение вероятностей. При умножении ширины интервала на `likelihoods[i]` возвращается вероятность того, что случайный исход попадет в диапазон от `bin_edges[i]` до `bin_edges[i+1]`.
- `_, _, patches = plt.hist(data)` — список `patches` — это третья из трех переменных, возвращаемых `plt.hist`. Графические установки каждого интервала по индексу `i` сохраняются в `patches[i]`. Вызов `patches[i].set_facecolor('yellow')` изменяет цвет столбца гистограммы в позиции `i`.
- `likelihoods, bin_edges = np.histogram(data, density=True)` — возвращает значения вероятностей гистограммы и границы интервалов без фактической отрисовки результатов.

3.3. ИСПОЛЬЗОВАНИЕ ДОВЕРИТЕЛЬНЫХ ИНТЕРВАЛОВ ДЛЯ АНАЛИЗА СМЕЩЕННОЙ КОЛОДЫ КАРТ

Предположим, что вам показали смещенную колоду из 52 карт. Каждая из них красного либо черного цвета, но количество красных и черных карт не равно. Сколько красных карт в колоде? Выяснить это можно путем их подсчета одна за другой, но так будет слишком просто. Внесем ограничение, чтобы сделать эту задачку поинтереснее. Вам позволено видеть лишь первую карту в колоде. Если хотите увидеть очередную, то сначала должны колоду перетасовать. Делать это можно любое количество раз, после каждого открывая верхнюю карту.

С учетом этих ограничений нужно решить задачу, используя случайный набор данных. Начнем с моделирования колоды из 52 карт с неизвестным количеством красных. Это будет число от 0 до 52, которое можно сгенерировать при помощи `np.random.randint` (листинг 3.33). Мы будем держать случайное значение `red_card_count` в тайне, пока не найдем решение с помощью анализа.

Листинг 3.33. Генерация случайного количества красных карт

```
np.random.seed(0)
total_cards = 52
red_card_count = np.random.randint(0, total_cards + 1)
```

А теперь вычислим `black_card_count` на основе условия, что `red_card_count` и `black_card_count` в сумме должны давать 52 карты (листинг 3.34). При этом мы сохраняем смещение, обеспечивая неравенство количеств черных и красных карт.

Листинг 3.34. Генерация числа черных карт

```
black_card_count = total_cards – red_card_count
assert black_card_count != red_card_count
```

Во время фазы моделирования мы будем перетасовывать колоду и переворачивать верхнюю карту. Какова вероятность того, что она окажется красной? Что ж, красная карта представляет один из двух возможных исходов: черный или красный. Эти исходы можно охарактеризовать пространством событий `{'red_card', 'black_card'}`, но только когда вероятности этих исходов равны. В нашей смещенной колоде исходы имеют вес, определяемый `red_card_count` и `black_card_count`. Значит, нужно взвешенное пространство событий `dict`, в котором значения словаря равны переменным, отражающим количество карт. Мы размечаем связанные ключи `'red_card'` и `'black_card'`. Передача `weighted_sample_space` в `compute_event_probability` позволит вычислить вероятность вытягивания красной карты (листинг 3.35).

Листинг 3.35. Использование пространства событий для вычисления вероятности извлечения красной карты

```
weighted_sample_space = {'red_card': red_card_count, 'black_card': black_card_count}
prob_red = compute_event_probability(lambda x: x == 'red_card',
                                     weighted_sample_space)
```

Напомню, что функция `compute_event_probability` делит переменную `red_card_count` на сумму `red_card_count` и `black_card_count`, вычисляя вероятность. При этом сумма `red_card_count` и `black_card_count` равна `total_cards`. Следовательно, вероятность вытаскивания красной карты равна `red_card_count`, разделенной на `total_cards`. Надо это проверить (листинг 3.36).

Листинг 3.36. Использование деления для вычисления вероятности вытащить красную карту

```
assert prob_red == red_card_count / total_cards
```

Как нужно применять `prob_red` для моделирования переворачивания первой карты? Переворачивание карты даст один из двух исходов: красный или черный. Эти два варианта можно смоделировать как подбрасывание монет, на которых орлы и решки заменены цветами. Следовательно, перевернутую карту мы моделируем при помощи биномиального распределения (листинг 3.37). Вызов `np.random.binomial (1, prob_red)` возвращает `1`, если первая карта красная, и `0` в противном случае.

Листинг 3.37. Симуляция вытаскивания случайной карты

```
np.random.seed(0)
color = 'red' if np.random.binomial(1, prob_red) else 'black'
print(f"The first card in the shuffled deck is {color}")

The first card in the shuffled deck is red
```

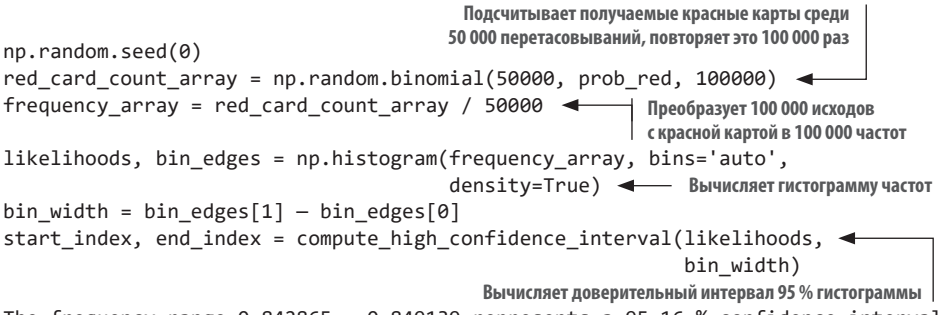
Мы перетасовываем колоду десять раз, после каждого открывая первую карту (листинг 3.38).

Листинг 3.38. Симуляция вытаскивания десяти случайных карт

```
np.random.seed(0)
red_count = np.random.binomial(10, prob_red)
print(f"In {red_count} of out 10 shuffles, a red card came up first.")

In 8 of out 10 shuffles, a red card came up first.
```

Красная карта попалась в восьми из десяти случаев. Означает ли это, что 80 % карт в колоде красные? Конечно же, нет. Ранее уже демонстрировалось, что подобные результаты при небольшом размере пространства событий вполне реальны. Теперь же мы перетасуем колоду не 10, а 50 000 раз, после чего вычислим частоту и повторим процедуру перетасовывания еще 100 000 раз. Эти шаги мы выполним путем вызова `np.random.binomial(50000, prob_red, 100000)` и деления на 50 000. Полученный массив частот можно преобразовать в гистограмму, которая позволит вычислить доверительный интервал 95 % для открытия красной карты. Этот интервал мы получим, расширяя диапазон интервалов, окружающих пик гистограммы, до тех пор пока он не охватит 95 % ее площади (листинг 3.39).

Листинг 3.39. Вычисление доверительных интервалов для вероятности вскрытия красной карты




```
np.random.seed(0)
red_card_count_array = np.random.binomial(50000, prob_red, 100000)
frequency_array = red_card_count_array / 50000

likelihoods, bin_edges = np.histogram(frequency_array, bins='auto',
                                      density=True)
bin_width = bin_edges[1] - bin_edges[0]
start_index, end_index = compute_high_confidence_interval(likelihoods,
                                                          bin_width)
The frequency range 0.842865 - 0.849139 represents a 95.16 % confidence interval
```

Здесь высока уверенность в том, что `prob_red` лежит где-то между 0,842865 и 0,849139. Нам также известно, что `prob_red` равна `red_card_count/total_cards`, а значит, `red_card_count` равно `prob_red * total_cards`. Таким образом, мы имеем очень высокую уверенность в том, что `red_card_count` находится между 0,842865 × `total_cards` и 0,849139 × `total_cards`. Теперь выясним вероятный диапазон `red_card_count` (листинг 3.40). Мы округляем конечные точки этого диапазона до ближайших целых чисел, потому что `red_card_count` соответствует целочисленному значению.

Листинг 3.40. Приблизительная оценка количества красных карт

```
range_start = round(0.842771 * total_cards)
range_end = round(0.849139 * total_cards)
print(f"The number of red cards in the deck is between {range_start} and
     {range_end}")

The number of red cards in the deck is between 44 and 44
```

Теперь мы почти уверены, что в колоде 44 красные карты. Проверим, верно ли это решение (листинг 3.41).

Листинг 3.41. Проверка количества красных карт

```
if red_card_count == 44:
    print('We are correct! There are 44 red cards in the deck')
else:
    print('Oops! Our sampling estimation was wrong.')

We are correct! There are 44 red cards in the deck
```

В колоде действительно 44 красные карты. Определить это удалось без ручного подсчета. Для получения решения оказалось достаточно симулировать случайное перетасовывание и извлечение карты, после чего вычислить доверительный интервал.

3.4. ПЕРЕТАСОВКА КАРТ С ПОМОЩЬЮ ПЕРМУТАЦИЙ

Перетасовка карт требует случайного переупорядочения элементов колоды. Выполнить его можно с помощью метода `np.random.shuffle`, который получает на входе упорядоченный массив либо список и перемешивает его элементы. Код листинга 3.42 случайным образом перетасует колоду, содержащую две красные карты (представленные единицами) и две черные (представленные нулями).

Листинг 3.42. Перетасовка колоды из четырех карт

```
np.random.seed(0)
card_deck = [1, 1, 0, 0]
np.random.shuffle(card_deck)
print(card_deck)

[0, 0, 1, 1]
```

Этот метод переупорядочил элементы в `card_deck`. Если же нужно перетасовать колоду, сохранив копию ее оригинального представления, то это можно сделать при помощи `np.random.permutation` (листинг 3.43). Этот метод возвращает массив NumPy, содержащий случайный порядок карт. При этом порядок элементов исходной колоды остается неизменным.

Листинг 3.43. Возвращение копии перетасованной колоды

```
np.random.seed(0)
unshuffled_deck = [1, 1, 0, 0]
shuffled_deck = np.random.permutation(unshuffled_deck)
assert unshuffled_deck == [1, 1, 0, 0]
print(shuffled_deck)

[0 0 1 1]
```

Такое случайное упорядочение элементов, возвращаемых методом `np.random.permutation`, в математике называется *пермутацией*. Операция пермутации в большинстве случаев дает порядок, отличный от исходного, хотя в редких случаях может с ним совпадать. Какова вероятность того, что перемешанный результат будет в точности соответствовать `unshuffled_deck`?

Естественно, это можно выяснить с помощью развернутого анализа. Однако колода из четырех элементов довольно мала для того, чтобы сделать это с помощью пространств элементарных исходов событий. Для составления такого пространства потребуется перебрать все возможные варианты пермутации колоды. Это несложно реализовать с помощью функции `itertools.permutations`. Вызов `itertools.permutations(unshuffled_deck)` вернет результат перебора всех возможных пермутаций колоды. Давайте используем эту функцию для вывода первых трех пермутаций

(листинг 3.44). Заметьте, что они выводятся как кортежи Python, а не как массивы или списки. Кортежи отличаются тем, что не допускают изменения элементов и представляются в круглых скобках.

Листинг 3.44. Перебор пермутаций карт

```
import itertools
for permutation in list(itertools.permutations(unshuffled_deck))[:3]:
    print(permutation)

(1, 1, 0, 0)
(1, 1, 0, 0)
(1, 0, 1, 0)
```

Первые две пермутации оказались идентичными. Почему? Первая из них — это просто исходная `unshuffled_deck`, в которой порядок элементов не менялся. Вторая была сгенерирована перестановкой третьего и четвертого элементов первой. Но, поскольку оба они оказались нулями, перестановка по факту ничего не изменила. Убедиться в том, что она действительно произошла, можно, рассмотрев первые три пермутации `[0, 1, 2, 3]` (листинг 3.45).

Листинг 3.45. Отслеживание перестановок при пермутациях

```
for permutation in list(itertools.permutations([0, 1, 2, 3]))[:3]:
    print(permutation)

(0, 1, 2, 3)
(0, 1, 3, 2)
(0, 2, 1, 3)
```

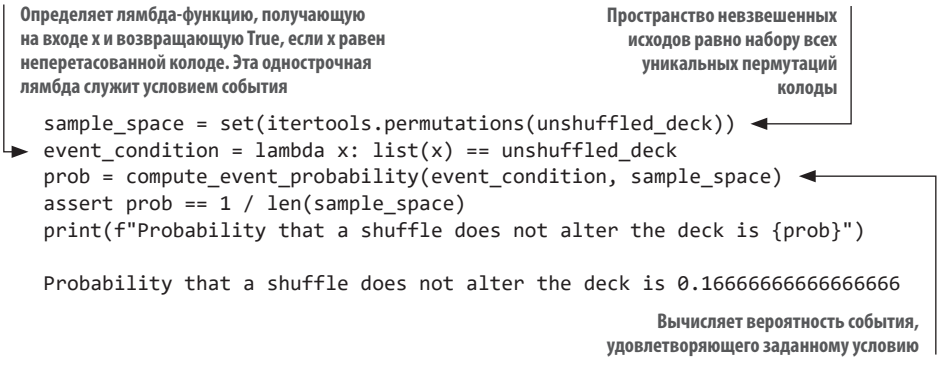
Некоторые пермутации колоды из четырех карт происходят более одного раза. Значит, можно предположить, что определенные пермутации происходят чаще других. Давайте проверим эту гипотезу, сохранив количество пермутаций в словаре `weighted_sample_space` (листинг 3.46).

Листинг 3.46. Вычисление числа пермутаций

```
weighted_sample_space = defaultdict(int)
for permutation in itertools.permutations(unshuffled_deck):
    weighted_sample_space[permutation] += 1

for permutation, count in weighted_sample_space.items():
    print(f"Permutation {permutation} occurs {count} times")

Permutation (1, 1, 0, 0) occurs 4 times
Permutation (1, 0, 1, 0) occurs 4 times
Permutation (1, 0, 0, 1) occurs 4 times
Permutation (0, 1, 1, 0) occurs 4 times
Permutation (0, 1, 0, 1) occurs 4 times
Permutation (0, 0, 1, 1) occurs 4 times
```

Все пермутации случаются с равной частотой (листинг 3.47). Это значит, что все варианты порядка карт равновероятны и взвешенное пространство элементарных исходов не требуется. Так что для решения задачи вполне хватит невзвешенного, которое равно `set(itertools.permutations(unshuffled_deck))`.

Листинг 3.47. Вычисление вероятностей пермутаций



Предположим, что дана обобщенная `unshuffled_deck` размером N, в которой содержится $N/2$ красных карт. Математически можно показать, что все пермутации цветов в колоде произойдут с одинаковой вероятностью. Таким образом, можно вычислить вероятности непосредственно с помощью невзвешенного пространства исходов. К сожалению, создание этого пространства для колоды из 52 карт нецелесообразно, поскольку число возможных пермутаций окажется невероятно большим — $8{,}06 \times 10^{67}$ вариантов, что превосходит число атомов на Земле. Если мы попытаемся вычислить пространство элементарных исходов для 52 карт, программе потребуется много дней, прежде чем она окончательно не исчерпает память. Однако подобное пространство можно без труда вычислить для колоды из десяти карт (листинг 3.48).

Листинг 3.48. Вычисление пространства исходов для десяти карт

```
red_cards = 5 * [1]
black_cards = 5 * [0]
unshuffled_deck = red_cards + black_cards
sample_space = set(itertools.permutations(unshuffled_deck))
print(f"Sample space for a 10-card deck contains {len(sample_space)}
        elements")

Sample space for a 10-card deck contains 252 elements
```

Перед нами стояла задача найти наилучшую стратегию для вытаскивания именно красной карты. В ее решении нам пригодилась `sample_space`, состоящая из десяти карт: этот набор позволяет напрямую вычислить вероятности при помощи разных стратегий. Исходя из этого, можно применить самые удачные из них к колоде из 52 карт.

РЕЗЮМЕ

- Метод `np.random.binomial` может симулировать случайные подбрасывания монеты. Его имя происходит от *биномиального распределения*, которое является обобщенным распределением, описывающим вероятности исхода подбрасывания монеты.
- При повторяющемся подбрасывании монеты частота исходов с орлом постепенно сходится к фактической вероятности его выпадения. Однако итоговая частота может несколько от нее отличаться.
- Вариативность частот исходов можно отобразить визуально с помощью *гистограммы*. Она показывает количество интервалов для получаемых численных значений. Эти количества можно преобразовать в относительные вероятности, чтобы площадь под гистограммой суммировалась в 1,0. По сути, такая преобразованная гистограмма становится распределением вероятностей. Область вокруг пика этого распределения представляет *доверительный интервал* — вероятность того, что неизвестная вероятность попадет в определенный диапазон частот. В статистике желательно, чтобы этот интервал составлял 95 % и больше.
- Когда количество вычисленных частот становится довольно значительным, кривая их гистограммы по своей форме начинает напоминать колокол. Называют эту кривую *гауссовым*, или *нормальным, распределением*. Согласно *центральной предельной теореме*, доверительный интервал 95 %, связанный с этой кривой, становится тем у́же, чем больше количество данных, на основе которых вычисляется каждая частота.
- Симуляцию перетасовки карт можно выполнить с помощью метода `np.random.permutation`, который возвращает случайную пермутацию входной колоды карт. *Пермутация* представляет собой случайное упорядочение элементов. Все возможные варианты пермутаций можно перебрать с помощью `itertools.permutations`. Перебор всех пермутаций для колоды из 52 карт вычислить невозможно. Однако можно без проблем охватить все их варианты для колоды из десяти карт и использовать для вычисления пространства элементарных исходов этой колоды.

1 Решение практического задания 1

В этой главе

- ✓ Симуляции карточной игры.
- ✓ Оптимизация стратегии вычисления вероятности.
- ✓ Доверительные интервалы.

Мы планируем сыграть в игру, в которой карты поочередно открываются до тех пор, пока мы не говорим дилеру: «Стоп». После этого открывается еще одна карта: если она оказывается красной, мы выигрываем доллар, в противном случае теряем его. Задача — определить стратегию, которая наилучшим образом прогнозирует открытие именно красной карты. Для этого сделаем следующее.

1. Разработаем несколько стратегий для прогнозирования выпадения красных карт в случайно перетасованной колоде.
2. Применим каждую стратегию для нескольких симуляций, вычислив вероятность успеха с высоким доверительным интервалом. Если эти вычисления окажутся неподъемными, мы переключимся на те стратегии, которые лучше всего справятся с пространством элементарных исходов для десяти карт.
3. Вернем простейшую стратегию, связанную с наивысшей вероятностью успеха.

ВНИМАНИЕ!

Спойлер! Вскоре вы узнаете решение для рассматриваемого исследования. Я настоятельно рекомендую вам попытаться решить эту задачу самостоятельно, прежде чем изучать готовое решение. Описание задачи находится в начале этого исследования.

4.1. ПРОГНОЗИРОВАНИЕ КРАСНЫХ КАРТ В ПЕРЕТАСОВАННОЙ КОЛОДЕ

Начнем с создания колоды, содержащей 26 красных и 26 черных карт (листинг 4.1). Черные карты представлены нулями, а красные — единицами.

Листинг 4.1. Моделирование колоды из 52 карт

```
red_cards = 26 * [1]
black_cards = 26 * [0]
unshuffled_deck = red_cards + black_cards
```

Далее колода перетасовывается (листинг 4.2).

Листинг 4.2. Перетасовывание колоды из 52 карт

```
np.random.seed(1)
shuffled_deck = np.random.permutation(unshuffled_deck)
```

Теперь мы поочередно открываем карты, останавливаясь, когда очередная с наибольшей вероятностью должна оказаться красной. После этого открываем очередную карту и выигрываем, если она действительно оказывается красной.

Теперь осталось лишь решить, когда же останавливаться. Одной из предполагаемых простых стратегий будет закончить игру, когда количество оставшихся в колоде красных карт будет больше числа оставшихся черных. Применим эту стратегию к перетасованной колоде (листинг 4.3).

Листинг 4.3. Перенос в код стратегии для карточной игры

```
remaining_red_cards = 26
for i, card in enumerate(shuffled_deck[:-1]):
    remaining_red_cards -= card
    remaining_total_cards = 52 - i - 1
    if remaining_red_cards / remaining_total_cards > 0.5:
        break

print(f"Stopping the game at index {i}.")
final_card = shuffled_deck[i + 1]
color = 'red' if final_card else 0
print(f"The next card in the deck is {'red' if final_card else 'black'}.")
print(f"We have {'won' if final_card else 'lost'}!")
```

Вычитает общее число извлеченных карт из 52. Это число равно i+1, поскольку отсчет i начался с нуля. В качестве альтернативы можно выполнить enumerate(shuffled_deck[:-1], 1), чтобы отсчет i начинался с единицы

```
Stopping the game at index 1.
The next card in the deck is red.
We have won!
```

Такая стратегия привела к победе на первой же попытке. По ее правилам мы останавливаемся, когда доля красных карт оказывается больше половины общего числа оставшихся карт. Эту долю можно обобщить как равную параметру `min_red_fraction`, чтобы останавливаться, когда соотношение красных карт окажется

больше указанного значения параметра. В общем виде эта стратегия реализована в листинге 4.4 при `min_red_fraction`, равном `0.5`.

Листинг 4.4. Обобщение победной стратегии для карточной игры

```
np.random.seed(0)
total_cards = 52
total_red_cards = 26
def execute_strategy(min_fraction_red=0.5, shuffled_deck=None,
                     return_index=False):
    if shuffled_deck is None:
        shuffled_deck = np.random.permutation(unshuffled_deck)

    remaining_red_cards = total_red_cards

    for i, card in enumerate(shuffled_deck[:-1]):
        remaining_red_cards -= card
        fraction_red_cards = remaining_red_cards / (total_cards - i - 1)
        if fraction_red_cards > min_fraction_red:
            break
    return (i+1, shuffled_deck[i+1]) if return_index else shuffled_deck[i+1]
```

Если входной колоды нет, перемешивает неперетасованную колоду

При необходимости возвращает вместе с последней картой ее индекс

4.1.1. Оценка вероятности успеха стратегии

Применим нашу базовую стратегию к серии из 1000 случайных перемешиваний карт, проще говоря, игр (листинг 4.5).

Листинг 4.5. Выполнение стратегии в 1000 игр

```
observations = np.array([execute_strategy() for _ in range(1000)])
```

Общая доля единиц в результатах соответствует полученной доле красных карт, а значит, доле побед. Вычислить ее можно, сложив единицы в `observations` и разделив результат на размер массива. В качестве альтернативы этот расчет можно выполнить с помощью `observations.mean()` (листинг 4.6).

Листинг 4.6. Вычисление частоты побед

```
frequency_wins = observations.sum() / 1000
assert frequency_wins == observations.mean()
print(f"The frequency of wins is {frequency_wins}")
```

```
The frequency of wins is 0.511
```

Мы победили в 51,1 % игр! Похоже, что наша стратегия работает: 511 побед и 489 поражений принесли нам общую прибыль 22 доллара (листинг 4.7).

Выбранная стратегия отлично сработала при 1000 игр. Теперь нарисуем график сходимости частоты побед для серии разных количеств игр — от 1 до 10 000 (листинг 4.8; рис. 4.1).

Листинг 4.7. Вычисление общей прибыли

```
dollars_won = frequency_wins * 1000
dollars_lost = (1 – frequency_wins) * 1000
total_profit = dollars_won – dollars_lost
print(f"Total profit is ${total_profit:.2f}")

Total profit is $22.00
```

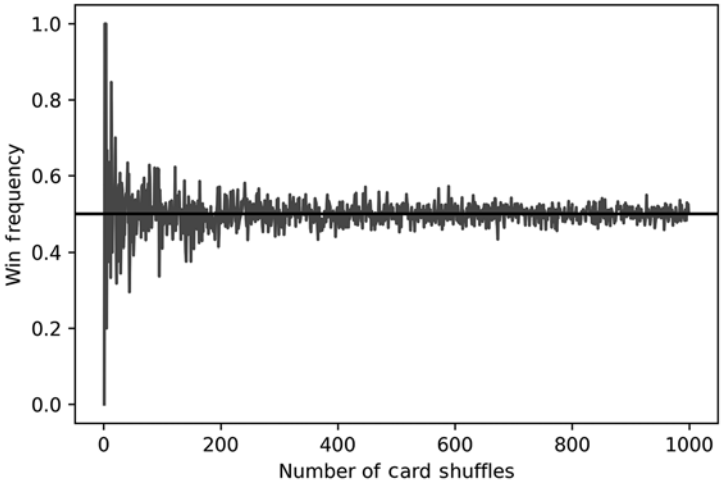


Рис. 4.1. Соотношение количества сыгранных игр и частоты побед. Частоты колеблются вокруг значения 0,5. Здесь нельзя сказать, находится вероятность победы выше или ниже 0,5

Листинг 4.8. Отрисовка симулированных частот побед

```
np.random.seed(0)
def repeat_game(number_repeats):
    observations = np.array([execute_strategy()
                             for _ in range(number_repeats)])
    return observations.mean()

frequencies = []
for i in range(1, 1000):
    frequencies.append(repeat_game(i))

plt.plot(list(range(1, 1000)), frequencies)
plt.axhline(0.5, color='k')
plt.xlabel('Number of Card Shuffles')
plt.ylabel('Win-Frequency')
plt.show()
print(f"The win-frequency for 10,000 shuffles is {frequencies[-1]}")

The win-frequency for 10,000 shuffles is 0.5035035035035035
```

Возвращает частоту побед в заданном числе игр

При 10 000 игр эта стратегия дает частоту побед более 50 %. Тем не менее в процессе анализа она иногда проседает ниже 50 %. Какова же наша уверенность в том, что вероятность победы фактически окажется больше 0,5? Выяснить это можно с помощью анализа доверительного интервала (рис. 4.2). Этот интервал мы вычисляем путем 300-кратной оценки 10 000 перетасовываний, что в сумме дает 3 млн игр. Перемешивание массива вычислительно затратно, поэтому код листинга 4.9 выполняется примерно 40 с.

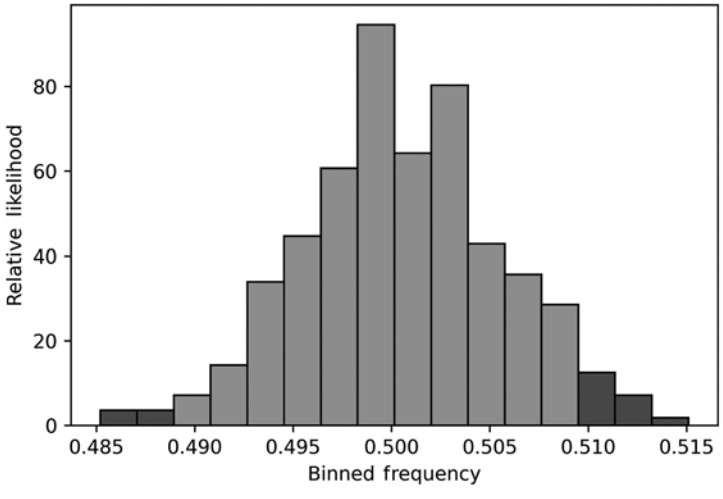


Рис. 4.2. Гистограмма 300 частот, распределенных по интервалам в соотношении с их относительными вероятностями. Выделенные столбцы обозначают доверительный интервал 95 %, охватывающий диапазон частот примерно от 0,488 до 0,508

Листинг 4.9. Вычисление доверительного интервала для 3 млн игр

```
np.random.seed(0)
frequency_array = np.array([repeat_game(10000) for _ in range(300)])
likelihoods, bin_edges, patches = plt.hist(frequency_array, bins='auto',
                                           edgecolor='black', density=True)
bin_width = bin_edges[1] – bin_edges[0]
start_index, end_index = compute_high_confidence_interval(likelihoods,
    bin_width)
for i in range(start_index, end_index):
    patches[i].set_facecolor('yellow')
plt.xlabel('Binned Frequency')
plt.ylabel('Relative Likelihood')

plt.show()

The frequency range 0.488938 – 0.509494 represents a 97.00 % confidence interval
```

Напомню, что функцию compute_high_confidence_interval мы определили в главе 3

Создается довольно высокая уверенность в том, что фактическая вероятность находится где-то между 0,488 и 0,509. Тем не менее до сих пор неизвестно, выше

она 0,5 или ниже. И это проблема, так как даже малейшая неточность в определении истинной вероятности может привести к потере денег.

Представьте, что истинная вероятность составляет 0,5001. Если мы применим нашу стратегию к 1 млрд игр, то можем ожидать выигрыш 200 000 долларов. А теперь предположим, что мы ошибались и фактическая вероятность равна 0,4999. В таком случае мы эти 200 000 долларов потеряем. Едва заметная ошибка в четвертом десятичном разряде обойдется не в одну сотню тысяч долларов.

Поэтому нам необходима абсолютная уверенность в том, что истинная вероятность находится выше 0,5. Значит, нужно сузить доверительный интервал путем увеличения количества игр в одной серии, за что мы поплатимся более длительным временем выполнения. Код листинга 4.10 анализирует 50 000 игр 3000 раз.

ПРЕДУПРЕЖДЕНИЕ

Следующий код выполняется примерно час.

Листинг 4.10. Вычисление доверительного интервала для 150 млн игр

```
np.random.seed(0)

frequency_array = np.array([repeat_game(50000) for _ in range(3000)])
likelihoods, bin_edges = np.histogram(frequency_array, bins='auto', density=True)
bin_width = bin_edges[1] – bin_edges[0]
compute_high_confidence_interval(likelihoods, bin_width)

The frequency range 0.495601 – 0.504345 represents a 96.03 % confidence interval
```

Мы выполнили анализ. К сожалению, новый доверительный интервал по-прежнему не проясняет, находится ли истинная вероятность выше 0,5. Что же делать? Дальнейшее увеличение числа перетасовываний будет уже вычислительно нецелесообразным (если только вы не горите желанием пару дней подождать завершения симуляции). Возможно, улучшения удастся добиться, увеличив `min_red_fraction` с 0,50 до 0,75. Обновим нашу стратегию и пойдем погуляем, пока симуляция выполняется в течение очередного часа.

ПРЕДУПРЕЖДЕНИЕ

Код листинга 4.11 выполняется около часа.

Листинг 4.11. Вычисление доверительного интервала для обновленной стратегии

```
np.random.seed(0)
def repeat_game(number_repeats, min_red_fraction):
    observations = np.array([execute_strategy(min_red_fraction)
                             for _ in range(number_repeats)])
    return observations.mean()

frequency_array = np.array([repeat_game(50000, 0.75) for _ in range(3000)])
likelihoods, bin_edges = np.histogram(frequency_array, bins='auto', density=True)
```

```
bin_width = bin_edges[1] – bin_edges[0]
compute_high_confidence_interval(likelihoods, bin_width)

The frequency range 0.495535 – 0.504344 represents a 96.43 % confidence interval
```

Ни в какую! Охват доверительного интервала по-прежнему не дает конкретики, включая в себя как выгодные, так и невыгодные вероятности.

Возможно, удастся прояснить этот вопрос, применив наши стратегии к колоде из десяти карт. Ее пространство элементарных исходов можно изучить полноценно, что позволит вычислить точные шансы на победу.

4.2. ОПТИМИЗАЦИЯ СТРАТЕГИЙ С ПОМОЩЬЮ ПРОСТРАНСТВА ИСХОДОВ ДЛЯ КОЛОДЫ ИЗ ДЕСЯТИ КАРТ

Код листинга 4.12 просчитывает пространство элементарных исходов для колоды из десяти карт, после чего применяет к нему базовую стратегию. Итоговым выводом является вероятность получения выигрыша с помощью этой стратегии.

Листинг 4.12. Применение базовой стратегии к колоде из десяти карт

```
total_cards = 10
total_red_cards = int(total_cards / 2)
total_black_cards = total_red_cards
unshuffled_deck = [1] * total_red_cards + [0] * total_black_cards
sample_space = set(itertools.permutations(unshuffled_deck))
win_condition = lambda x: execute_strategy(shuffled_deck=np.array(x))
prob_win = compute_event_probability(win_condition, sample_space)
print(f"Probability of a win is {prob_win}")

Probability of a win is 0.5
```

Напомню, что itertools мы импортировали в главе 3

Условие события, при котором наша базовая стратегия ведет к выигрышу

Функцию compute_event_probability мы определили в главе 1

Удивительно, но наша основная стратегия дает победу лишь в 50 % случаев. Это не лучше, чем выбирать первую карту наугад! Возможно, параметр `min_red_fraction` был недостаточно мал. Выяснить это можно, проанализировав все значения с двумя десятичными разрядами между 0,5 и 1,0. Код листинга 4.13 вычисляет вероятности победы для диапазона значений `min_red_fraction`, возвращая минимальные и максимальные вероятности.

Листинг 4.13. Применение нескольких стратегий к колоде из десяти карт

```
def scan_strategies():
    fractions = [value / 100 for value in range(50, 100)]
    probabilities = []
    for frac in fractions:
        win_condition = lambda x: execute_strategy(frac, shuffled_deck=np.array(x))
```

```
        probabilities.append(compute_event_probability(win_condition, sample_space))
    return probabilities

probabilities = scan_strategies()
print(f"Lowest probability of win is {min(probabilities)}")
print(f"Highest probability of win is {max(probabilities)}")

Lowest probability of win is 0.5
Highest probability of win is 0.5
```

И минимальная, и максимальная вероятности равны 0,5. Ни одна из наших стратегий не превзошла случайный выбор карты. Возможно, корректировка размера колоды позволит добиться улучшения. Проанализируем пространства элементарных исходов колод, состоящих из двух, четырех, шести и восьми карт. Применим обе стратегии к каждому пространству и вернем их вероятности победы, среди которых найдем ту, которая не равна 0,5 (листинг 4.14).

Листинг 4.14. Применение нескольких стратегий к нескольким колодам

```
for total_cards in [2, 4, 6, 8]:
    total_red_cards = int(total_cards / 2)
    total_black_cards = total_red_cards
    unshuffled_deck = [1] * total_red_cards + [0] * total_black_cards

    sample_space = set(itertools.permutations(unshuffled_deck))
    probabilities = scan_strategies()
    if all(prob == 0.5 for prob in probabilities):
        print(f"No winning strategy found for deck of size {total_cards}")
    else:
        print(f"Winning strategy found for deck of size {total_cards}")

No winning strategy found for deck of size 2
No winning strategy found for deck of size 4
No winning strategy found for deck of size 6
No winning strategy found for deck of size 8
```

Все стратегии, будучи примененными к небольшим колодам, дают вероятность победы 0,5. При каждом увеличении размера колоды мы добавляем в нее две карты, но к улучшению показателей это не ведет. Стратегия, которая не справляется с колодой из двух карт, не справляется и с колодой из четырех, а стратегия, которая проваливается для восьми карт, проваливается также для десяти. Эту логику можно экстраполировать и дальше. Стратегия, которая не справляется с колодой из десяти карт, наверняка не справится и с 12, 14 и 16 картами.

В конечном итоге она не подойдет и для колоды из 52 карт. С качественной точки зрения это индуктивное рассуждение имеет смысл, и его можно подтвердить математически. Но сейчас мы не хотим нагружать себя математикой. Нам важно опровергнуть сигналы своей интуиции. Наши стратегии не работают для колоды из десяти карт, и у нас нет убедительных оснований рассчитывать, что они сработают для колоды из 52 карт. Почему же они не справляются?

Казалось, что начальная стратегия была разумна: если в колоде остается больше красных карт, чем черных, то вероятность извлечь красную выше. Однако мы допустили оплошность, не приняв во внимание сценарии, в которых красные карты никогда не превосходят по количеству черные. Предположим, что первые 26 карт красные, а остальные черные. В таких условиях наши стратегии не дадут сигнал остановки игры и мы проиграем. Или возьмем, к примеру, перетасованную колоду, в которой первые 25 карт красные, следующие 26 — черные и последняя — красная. В этом случае стратегии также не найдут удачного места для остановки, но мы все равно победим. Получается, что каждая стратегия может привести к одному из четырех сценариев.

- Стратегия останавливает игру, и очередная карта оказывается красной. Победа.
- Стратегия останавливает игру, и очередная карта оказывается черной. Проигрыш.
- Стратегия не останавливает игру, и последняя карта оказывается красной. Победа.
- Стратегия не останавливает игру, и последняя карта оказывается черной. Проигрыш.

Давайте проанализируем, как часто эти четыре сценария возникают среди 50 000 игр. Мы запишем эти частоты по всем значениям, входящим в диапазон `min_red_fraction`. После этого на графике отобразим каждое значение `min_red_fraction` относительно частоты наблюдения каждого из четырех сценариев (листинг 4.15; рис. 4.3).

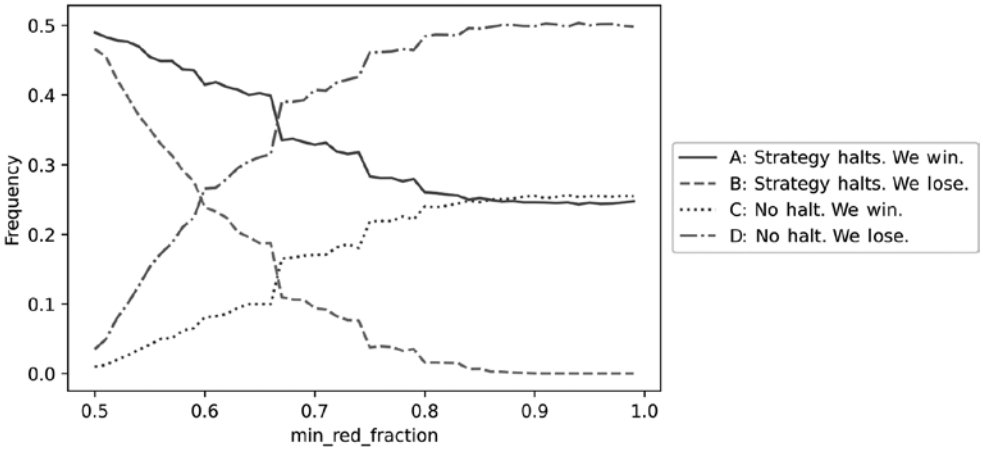


Рис. 4.3. Параметр min_red_fraction, отображенный относительно частот возникновения всех четырех сценариев. Частота сценария A изначально равна примерно 0,49, но в итоге падает до 0,25. Частота сценария C начинается примерно с 0,01 и в результате доходит до 0,25. Сумма частот A и C сохраняется около 0,5, тем самым отражая 50%-ный шанс на победу

Листинг 4.15. Построение графика результатов стратегий для колоды из 52 карт

```
np.random.seed(0)
total_cards = 52
total_red_cards = 26
unshuffled_deck = red_cards + black_cards

def repeat_game_detailed(number_repeats, min_red_fraction):
    observations = [execute_strategy(min_red_fraction, return_index=True)
                    for _ in range(num_repeats)]
    successes = [index for index, card, in observations if card == 1]
    halt_success = len([index for index in successes if index != 51])
    no_halt_success = len(successes) - halt_success
    failures = [index for index, card, in observations if card == 0]
    halt_failure = len([index for index in failures if index != 51])
    no_halt_failure = len(failures) - halt_failure
    result = [halt_success, halt_failure, no_halt_success, no_halt_failure]
    return [r / number_repeats for r in result]

fractions = [value / 100 for value in range(50, 100)]
num_repeats = 50000
result_types = [[], [], [], []]

for fraction in fractions:
    result = repeat_game_detailed(num_repeats, fraction)
    for i in range(4):
        result_types[i].append(result[i])

plt.plot(fractions, result_types[0],
         label='A) Strategy Halts. We Win.')
plt.plot(fractions, result_types[1], linestyle='--',
         label='B) Strategy Halts. We Lose.')
plt.plot(fractions, result_types[2], linestyle=':',
         label='C) No Halt. We Win.')
plt.plot(fractions, result_types[3], linestyle='-.',
         label='D) No Halt. We Lose.')
plt.xlabel('min_red_fraction')
plt.ylabel('Frequency')
plt.legend(bbox_to_anchor=(1.0, 0.5))
plt.show()
```

Список всех случаев поражений

Сценарий, в котором остановка происходит и мы побеждаем

Список всех случаев побед

Мы выполняем стратегию для num_repeats симуляций

Сценарий, в котором остановки не происходит и мы побеждаем

Возвращение полученных частот всех четырех сценариев

Просмотр частот сценариев в нескольких стратегиях

Сценарий, в котором остановки не происходит и мы проигрываем

Сценарий, в котором остановка происходит и мы проигрываем

Параметр bbox_to_anchor используется для размещения легенды над графиком, чтобы исключить наложение на четыре кривые

Рассмотрим график при значении `min_red_fraction`, равном `0.5`. Здесь сценарий A (стратегия приводит к остановке игры и победе) является самым частым исходом с частотой примерно 0,49. В то же время остановка ведет к поражению примерно в 46 % случаев (сценарий B). Почему же мы сохраняем 50%-ный шанс на победу? Что ж, в 1 % случаев наша стратегия к остановке игры не приводит, но мы все

равно побеждаем (сценарий C). Слабость стратегии уравновешивается элементом случайности.

На графике по мере увеличения `min_red_fraction` частота сценария A также увеличивается. Чем мы сдержаннее играем, тем с меньшей вероятностью останавливаем игру преждевременно, благодаря чему побеждаем. В то же время растет показатель успеха стратегии C. Чем сдержаннее мы действуем, тем выше вероятность дойти до последней карты и выиграть по воле случая.

По ходу увеличения `min_red_fraction` сценарии A и C сходятся к частоте 0,25. Таким образом, вероятность победы остается 50 %. Иногда наша стратегия ведет к остановке и мы побеждаем. В других случаях она также ведет к остановке, но мы проигрываем. Любое преимущество, обеспечиваемое любой из стратегий, автоматически этими поражениями аннулируется. Тем не менее время от времени нам везет — стратегия не приводит к остановке, но в игре мы побеждаем. Эти счастливые победы компенсируют поражения, и вероятность выигрыша остается прежней. Что бы мы ни делали, вероятность выиграть будет 50/50 (листинг 4.16). Следовательно, оптимальной стратегией будет открытие первой карты перетасованной колоды.

Листинг 4.16. Оптимальная стратегия для победы

```
def optimal_strategy(shuffled_deck):
return shuffled_deck[0]
```

РЕЗЮМЕ

- Вероятности не всегда совпадают с нашей интуицией. По наитию мы предположили, что спланированная нами игра в карты пройдет удачнее, чем случайная. На деле же оказалось не так. Нужно быть внимательными, работая со случайными процессами. Стоит тщательно протестировать все интуитивные предположения, прежде чем делать ставку на будущий исход.
- Иногда даже крупномасштабные симуляции не способны выявить вероятность в рамках требуемого уровня точности. Однако за счет упрощения задачи можно получить более точное представление с помощью пространств элементарных исходов. Эти пространства позволяют проверить наше чутье. Если интуитивное решение не годится для упрощенной версии задачи, то наверняка оно не справится и с реальной ее версией.

Практическое задание 2

Анализ значимости переходов по онлайн-объявлениям

УСЛОВИЕ ЗАДАЧИ

У вас есть верный друг Фред, которому нужна помощь. Фред только что открыл закусочную в центре Брисбена. Она ориентирована на бизнес-аудиторию, но люди идут вяло. Ваш друг хочет привлечь новых клиентов, чтобы они пришли отведать его вкуснейших бургеров. Для этого он запустит кампанию онлайн-рекламы, направленную на жителей Брисбена. Каждый день между 11:00 и 13:00 он будет оплачивать 3000 объявлений, привлекающих проголодавшихся людей. Каждое такое объявление будет просматриваться одним жителем. Текст же объявлений будет следующий: «Проголодался? Отведай лучший бургер в Брисбене. Приходи к Фреду». Клик по этому тексту будет перенаправлять потенциального клиента на сайт Фреда. Каждый показ объявления обходится вашему другу в 1 цент, но Фред верит, что результат оправдает вложения.

Сейчас он готовится к запуску этой кампании. Однако перед ним возникла проблема. Фред просматривает макет объявления, текст которого набран синим. Он считает, что это скучный цвет, и верит, что другие цвета обеспечат больше переходов по его рекламе. К счастью, используемое им рекламное ПО позволяет выбирать аж из 30 цветов. Существует ли такой цвет, который обеспечит больше переходов, чем синий? Фред хочет это выяснить и запускает эксперимент.

Каждый день в течение месяца он оплачивает 3000 онлайн-показов своего объявления, текст каждого из которых окрашен в один из 30 цветов. Эти объявления распределяются по цветам поровну. В результате 100 объявлений одного цвета каждый день просматривают 100 человек. Например, 100 человек видят синий текст, а другие 100 — зеленый. Эти количества суммируются в 3000 просмотров объявлений, равно разделенных на 30 разных цветов. Используемое Фредом ПО автоматически отслеживает ежедневные просмотры, а также записывает переходы, связанные с каждым цветом. Собранные данные сохраняются в таблице, которая содержит информацию о ежедневных щелчках кнопкой мыши и просмотрах для каждого цвета. Каждая строка этой таблицы сопоставляет цвет с просмотрами и щелчками по всем проанализированным дням.

Фред провел свой эксперимент и собрал данные для 20 дней. Все собранные данные организованы по цветам. Теперь он хочет знать, есть ли среди этих цветов такой, который обеспечивает существенно большее число переходов, чем синий. К сожалению, Фред не умеет правильно интерпретировать результаты и не не уверен в том, какие переходы были целенаправленными, а какие произошли случайно. Фред превосходно готовит бургеры, но совсем не разбирается в анализе данных. Именно поэтому он и обратился к вам с просьбой проанализировать его таблицу и сравнить количества ежедневных переходов. Его интересует цвет, который привлекает намного больше внимания, чем синий. Хотите помочь своему другу? Если да, то он обещает кормить вас бургерами в течение целого года!

ОПИСАНИЕ НАБОРА ДАННЫХ

Данные Фреда хранятся в файле `colored_ad_click_table.csv`. Файл `.csv` — это таблица, сохраненная в виде текста. Столбцы этой таблицы разделены запятыми. Первая строка файла содержит разделенные запятыми метки столбцов. Первые 99 символов этой строки — `Color,Click Count: Day 1,View Count: Day 1,Click Count: Day 2,View Count: Day 2,Click Count: Day 3,`.

Вкратце поясню, что обозначают метки столбцов.

- Столбец 1: `Color` (Цвет). Каждая строка в нем соответствует одному из 30 возможных цветов текста.
- Столбец 2: `Click Count: Day 1` (Число переходов: день 1). Здесь указано, сколько раз по каждому цветному объявлению произошел переход в первый день эксперимента.
- Столбец 3: `View Count: Day 1`. Содержит число просмотров каждого объявления в первый день. Если верить Фреду, то все ежедневные просмотры должны суммироваться в 100.
- Остальные 38 столбцов содержат количество переходов и просмотров для каждого из следующих 19 дней эксперимента.

ОБЗОР

Для решения поставленной задачи нужно уметь:

- измерять центральность и рассеяние собранных данных;
- интерпретировать значимость двух расходящихся средних с помощью вычисления p-значения;
- минимизировать ошибки, связанные с неверными измерениями p-значения;
- загружать данные таблицы и управлять ими с помощью Python.

5

Базовая вероятность и статистический анализ с помощью SciPy

В этой главе

- ✓ Анализ биномиальных распределений с помощью библиотеки SciPy.
- ✓ Определение центральности набора данных.
- ✓ Определение рассеяния набора данных.
- ✓ Вычисление центральности и дисперсии распределений вероятностей.

Статистика — это ответвление математики, работающее со сбором и интерпретированием численных данных. Она лежит в основе всей современной науки о данных. Сам термин *«статистика»* в исходном смысле означал «наука о государстве», поскольку статистические методы изначально были разработаны для анализа данных в государственной деятельности. Начиная с древних времен правительственные структуры собирали сведения, относящиеся к населению. Они использовались для сбора налогов и организации военных кампаний. В связи с этим важнейшие государственные решения зависели от качества собранной информации. Неточности при ее сборе могли привести к катастрофическим результатам. Именно поэтому государственные чиновники были очень обеспокоены любыми случайными отклонениями в записях. В конечном итоге теория вероятности преодолела эти отклонения, позволив интерпретировать случайность. С тех пор статистика и теория вероятности очень тесно переплелись.

Но, несмотря на тесную связь, в некоторых смыслах эти два столпа науки о данных сильно различаются. Теория вероятности изучает случайные процессы для потенциально бесконечного числа измерений. Она не ограничена рамками реального мира, что позволяет нам моделировать поведение монеты, представляя миллионы ее подбрасываний. Но в реальной жизни подбрасывание монеты миллионы раз окажется бессмысленным и затратным по времени занятием. Конечно же, можно пожертвовать некоторым объемом данных вместо того, чтобы подбрасывать монету день и ночь напролет. Статистики признают эти ограничения, налагаемые процессом сбора данных. Сбор реальных данных — дорогостоящее и длительное занятие. Каждая точка таких данных имеет свою цену. Мы не можем опросить все население страны без найма специальных людей, как и протестировать наши онлайн-объявления, не заплатив за каждое, по которому человек перешел. Таким образом, размер итогового набора данных обычно зависит от размера нашего бюджета. Если он ограничен, то и собранные данные будут ограничены. Этот компромисс между данными и ресурсами для их получения является центральным аспектом в современной статистике. Статистика помогает понять, какого конкретно количества данных будет достаточно, чтобы получить осмысленное представление и принять значимые решения. Цель статистики — найти в данных смысл, даже когда их количество ограниченно.

Статистика основывается на математике и обычно преподается с использованием математических уравнений. Однако прямое знакомство с такими уравнениями не является необходимой предпосылкой для понимания этой науки. В действительности многие специалисты по данным при выполнении статистического анализа не пишут формулы. Вместо этого они задействуют библиотеки Python вроде SciPy, которые реализуют все сложные математические вычисления за них. Тем не менее для правильного использования библиотеки все же требуется понимать статистические процессы. И в текущей главе мы выработаем это понимание, применив теорию вероятности к реальным задачам.

5.1. ИЗУЧЕНИЕ СВЯЗИ МЕЖДУ ДАННЫМИ И ВЕРОЯТНОСТЬЮ С ПОМОЩЬЮ SCIPY

SciPy, или *Scientific Python*, предоставляет множество полезных методов для научного анализа. Эта библиотека включает целый модуль `scipy.stats` для решения задач по нахождению вероятности и построению статистики. Начнем с ее установки и импорта модуля `stats` (листинг 5.1).

ПРИМЕЧАНИЕ

Для установки библиотеки SciPy выполните из терминала команду pip install scipy.

Листинг 5.1. Импорт модуля stats из SciPy

```
from scipy import stats
```

Модуль `stats` очень полезен для анализа случайности данных. К примеру, в главе 1 мы вычисляли вероятность того, что при 20 подбрасываниях честной монеты выпадет не менее 16 орлов. Наши вычисления требовали оценки всех возможных комбинаций при 20 подбрасываниях монеты. Тогда, чтобы измерить случайность получаемых результатов, мы вычислили вероятность получения 16 или более орлов либо решек. SciPy позволяет измерять эту вероятность непосредственно с помощью метода `stats.binom_test`. Его название отражает биномиальное распределение, которое управляет вариантами падения подброшенной монеты.

Этот метод требует трех параметров: количества орлов, общего числа подбрасываний монеты и вероятности того, что выпадет орел. Далее мы применим это биномиальное тестирование к 16 орлам, полученным в результате 20 подбрасываний монеты. Итоговый выход должен будет соответствовать ранее вычисленному значению, равному приблизительно 0,011.

ПРИМЕЧАНИЕ

SciPy и стандартный Python обрабатывают малые десятичные значения по-разному. В главе 1 при вычислении вероятности итоговое значение было округлено до 17 старших разрядов. SciPy же возвращает значение, содержащее 18 старших разрядов. Поэтому из соображений согласованности мы округлим вывод SciPy до 17 десятичных разрядов (листинг 5.2).

Листинг 5.2. Анализ маловероятного количества орлов с помощью SciPy

```
num_heads = 16
num_flips = 20
prob_head = 0.5
prob = stats.binom_test(num_heads, num_flips, prob_head)
print(f"Probability of observing more than 15 heads or 15 tails is {prob:.17f}")

Probability of observing more than 15 heads or 15 tails is 0.01181793212890625
```

Стоит подчеркнуть, что `stats.binom_test` не вычислял вероятность получения 16 орлов. Вместо этого он вернул вероятность того, что мы увидим последовательность подбрасываний монеты, в которой 16 или более исходов покажут одну ее сторону. Если нас интересует вероятность получения именно 16 орлов, то нужно использовать метод `stats.binom.pmf`. Он представляет *функцию вероятности биномиального распределения*. Эта функция сопоставляет входные целочисленные значения с вероятностью их получения. Таким образом, вызов `stats.binom.pmf(num_heads, num_flips, prob_heads)` вернет вероятность того, что монета даст `num_heads` исходов с орлом (листинг 5.3). При текущих установках это равно вероятности того, что честная монета упадет орлом вверх в 16 из 20 случаев.

Листинг 5.3. Вычисление точной вероятности с помощью stats.binom.pmf

```
prob_16_heads = stats.binom.pmf(num_heads, num_flips, prob_head)
print(f"The probability of seeing {num_heads} of {num_flips} heads is
     {prob_16_heads}")

The probability of seeing 16 of 20 heads is 0.004620552062988271
```

Мы использовали `stats.binom.pmf` для выяснения вероятности получения именно 16 орлов. Однако этот метод может вычислять и несколько вероятностей одновременно. Вероятности получения нескольких разных количеств орлов можно вычислить, если передать ему список значений, эти количества описывающий. К примеру, при передаче [4, 16] вернется двухэлементный массив NumPy, содержащий вероятности получения 4 и 16 орлов соответственно. Следовательно, вероятность получения 4 орлов и 16 решек равна вероятности получения 4 решек и 16 орлов. Таким образом, в результате выполнения `stats.binom.pmf([4, 16], num_flips, prob_head)` должен вернуться двухэлементный массив, чьи элементы равны. Проверим (листинг 5.4).

Листинг 5.4. Вычисление массива вероятностей с помощью stats.binom.pmf

```
probabilities = stats.binom.pmf([4, 16], num_flips, prob_head)
assert probabilities.tolist() == [prob_16_heads] * 2
```

Передача списка позволяет вычислить вероятности для интервалов. К примеру, если передать `range(21)` в `stats.binom.pmf`, то вычисленный массив будет содержать все вероятности в интервале, включающем все возможные количества орлов. Как говорилось в главе 1, сумма этих вероятностей должна равняться 1,0.

ПРИМЕЧАНИЕ

Вычисление суммы малых десятичных значений имеет свои тонкости. В процессе их суммирования накапливаются мелкие ошибки, в результате которых итоговая сумма вероятности будет чуть отличаться от 1,0, если не округлить ее до 14 старших разрядов. И это округление реализуется в коде листинга 5.5.

Листинг 5.5. Вычисление интервальной вероятности с помощью stats.binom.pmf

```
interval_all_counts = range(21)
probabilities = stats.binom.pmf(interval_all_counts, num_flips, prob_head)
total_prob = probabilities.sum()
print(f"Total sum of probabilities equals {total_prob:.14f}")

Total sum of probabilities equals 1.00000000000000
```

Кроме того, как говорилось в главе 2, отрисовка `interval_all_counts` относительно `probabilities` раскрывает форму распределения исходов 20 подбрасываний монеты. Это позволяет сгенерировать график распределения, не перебирая возможные комбинации исходов подбрасываний (листинг 5.6; рис. 5.1).

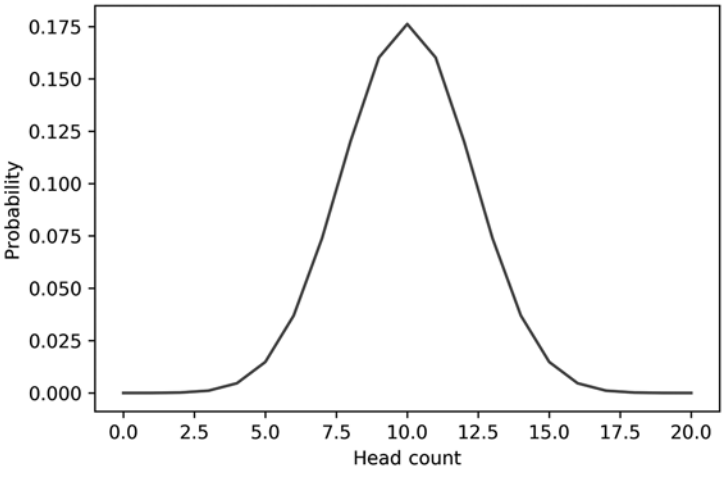


Рис. 5.1. Распределение вероятностей 20 подбрасываний монеты, сгенерированное с помощью SciPy

Листинг 5.6. Построение графика биномиального распределения исходов 20 подбрасываний монеты

```
import matplotlib.pyplot as plt
plt.plot(interval_all_counts, probabilities)
plt.xlabel('Head-count') plt.ylabel('Probability')
plt.show()
```

В главе 2 возможность визуализировать биномиальное распределение была ограничена общим количеством комбинаций исходов подбрасываний монеты, которые нужно было вычислить. Сейчас же ситуация иная. Метод `stats.binom.pmf` позволяет отобразить любое распределение, связанное с произвольным количеством подбрасываний. Воспользуемся этой свободой, чтобы одновременно отрисовать распределения для 20, 80, 140 и 200 подбрасываний монеты (листинг 5.7; рис. 5.2).

Листинг 5.7. Построение графиков разных биномиальных распределений

```
flip_counts = [20, 80, 140, 200]
linestyles = ['-', '--', '-.', ':']
colors = ['b', 'g', 'r', 'k']

for num_flips, linestyle, color in zip(flip_counts, linestyles, colors):
    x_values = range(num_flips + 1)
    y_values = stats.binom.pmf(x_values, num_flips, 0.5)
    plt.plot(x_values, y_values, linestyle=linestyle, color=color,
             label=f'{num_flips} coin-flips')
plt.legend()
```

```
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

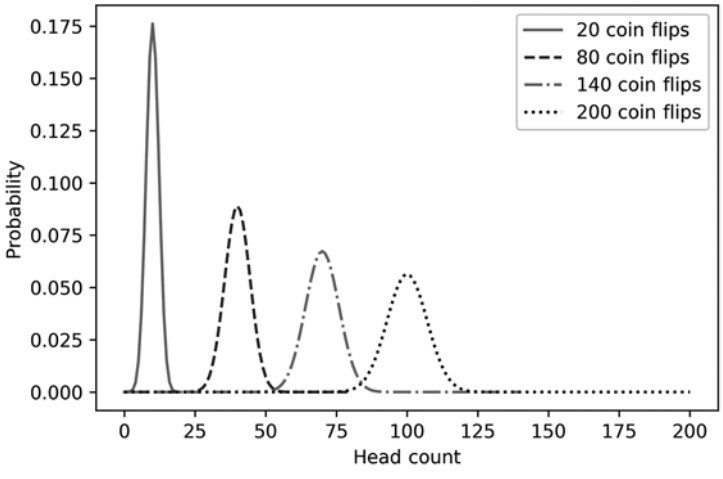


Рис. 5.2. Биномиальные распределения для 20, 80, 140 и 200 подбрасываний монеты. По мере увеличения числа подбрасываний центры этих распределений смещаются вправо. При этом чем больше подбрасываний, тем более рассеянным становится распределение вокруг своего центра

На графике центральные пики распределений смещаются вправо по мере увеличения числа подбрасываний монеты. При этом распределение для 20 подбрасываний оказывается заметно уже, чем его аналог для 200. Иными словами, при смещении центральных позиций вправо соответствующие им распределения становятся все более рассеянными вокруг этих центров.

Подобные смещения центральности и рассеяние встречаются в анализе данных повсеместно. Ранее мы уже видели смещение рассеяния в главе 3, когда применяли случайно сгенерированные данные для визуализации нескольких гистограмм распределений. Толщина полученной гистограммы зависела от количества использованных данных. В тот раз эти вычисления были исключительно качественными, поскольку нам не хватало показателя для сравнения толщины двух графиков. Тем не менее недостаточно просто отметить, что один график толще другого, равно как мало сказать, что один график смещен вправо больше, чем другой. Необходимо выяснить количественную разницу между распределениями. Центральности и рассеянию нужно присвоить определенные числа, чтобы отследить, как эти числа изменяются от графика к графику. Для этого потребуется познакомиться с понятиями *дисперсии случайной величины* и *среднего арифметического*.

5.2. СРЕДНЕЕ ЗНАЧЕНИЕ КАК МЕРА ЦЕНТРАЛЬНОСТИ

Предположим, что мы хотим изучить местную температуру воздуха в течение первой недели лета. Когда оно наступает, мы начинаем следить за термометром за окном. В обед первого дня температура равна 80 градусам[1]. Повторяем эти наблюдения в течение шести следующих дней и в целом получаем следующий итог: 80, 77, 73, 61, 74, 79 и 81 градус. Сохраним эти измерения в массиве NumPy (листинг 5.8).

Листинг 5.8. Сохранение показаний температуры в массиве NumPy

```
import numpy as np
measurements = np.array([80, 77, 73, 61, 74, 79, 81])
```

Теперь попробуем сложить эти показания с помощью одного центрального значения. Сначала нужно упорядочить их, вызвав функцию `measurements.sort()`. Затем построим график упорядоченных температур, чтобы оценить их центральность (листинг 5.9; рис. 5.3).

Листинг 5.9. Построение графика зарегистрированных температур

```
measurements.sort()
number_of_days = measurements.size
plt.plot(range(number_of_days), measurements)
plt.scatter(range(number_of_days), measurements)
plt.ylabel('Temperature')
plt.show()
```

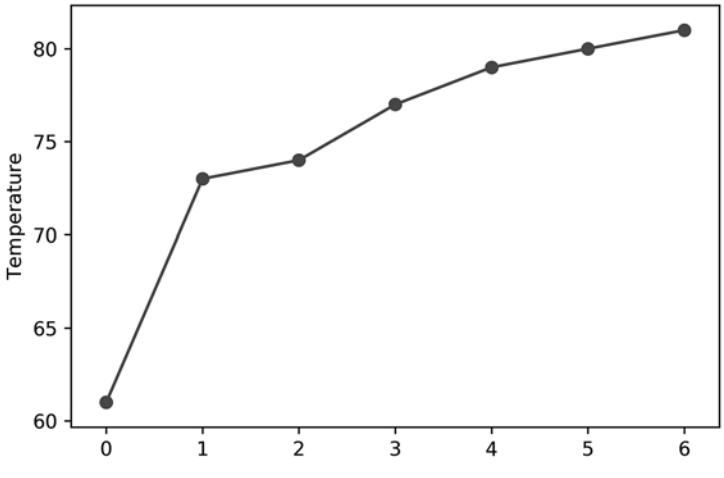


Рис. 5.3. График, содержащий семь упорядоченных значений температуры. Центральная температура находится где-то между 60 и 80 градусами

[1] Здесь и далее приведены градусы по шкале Фаренгейта. — *Примеч. ред.*

Исходя из графика, центральная температура находится где-то между 60 и 80 градусами. Следовательно, можно без лишних размышлений предположить, что центр расположен около 70 градусов. Эту оценку мы количественно определим как среднюю точку между наименьшим и наибольшим значениями графика. Вычисляется она прибавлением к минимальной температуре половины разницы между минимальным и максимальным значениями температуры.

Иначе это значение можно получить прямым сложением минимума и максимума и последующим делением их суммы на 2 (листинг 5.10).

Листинг 5.10. Поиск средней температуры

```
difference = measurements.max() – measurements.min()
midpoint = measurements.min() + difference / 2
assert midpoint == (measurements.max() + measurements.min()) / 2
print(f"The midpoint temperature is {midpoint} degrees")

The midpoint temperature is 71.0 degrees
```

Средняя температура равна 71 градусу. Отметим эту точку на графике горизонтальной линией, которую нарисуем с помощью `plt.axhline(midpoint)` (листинг 5.11; рис. 5.4).

Листинг 5.11. Отображение на графике средней температуры

```
plt.plot(range(number_of_days), measurements)
plt.scatter(range(number_of_days), measurements)
plt.axhline(midpoint, color='k', linestyle='--')
plt.ylabel('Temperature')
plt.show()
```

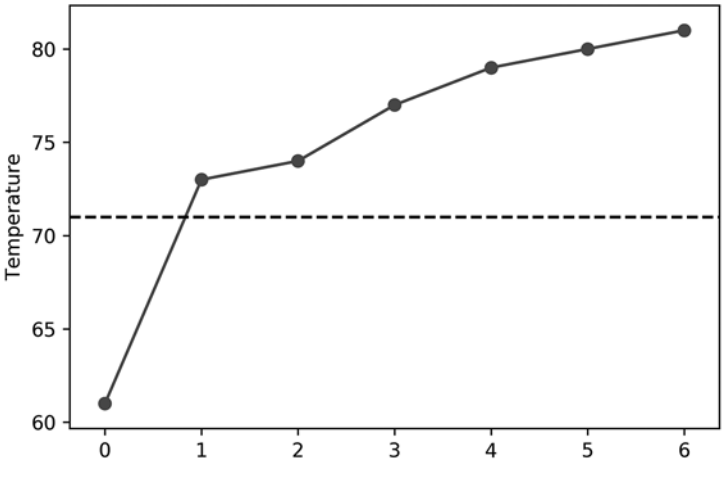


Рис. 5.4. График семи упорядоченных значений температуры. Температура 71 градус отмечает среднюю точку между максимальным и минимальным значениями. Она расположена чересчур низко, так как шесть из семи температур находятся над ней

Похоже, что наша средняя точка располагается низковато: шесть из семи значений температуры оказались выше нее. Чисто по наитию: центральное значение должно разделять измерения более ровно — количество показаний температуры выше и ниже него должно быть примерно одинаковым. Равенства можно добиться, выбрав средний элемент в нашем упорядоченном массиве. Этот элемент, который статистики называют медианой, разделит измерения на две равные части. Три показания окажутся под медианой, а три — над ней. При этом 3 будет индексом *медианы* в массиве `measurements`. Добавим медиану на график (листинг 5.12; рис. 5.5).

Листинг 5.12. Отображение на графике медианы температуры

```
median = measurements[3]
print(f"The median temperature is {median} degrees")
plt.plot(range(number_of_days), measurements)
plt.scatter(range(number_of_days), measurements)
plt.axhline(midpoint, color='k', linestyle='--', label='midpoint')
plt.axhline(median, color='g', linestyle='-.', label='median')
plt.legend()
plt.ylabel('Temperature')
plt.show()
```

```
The median temperature is 77 degrees
```

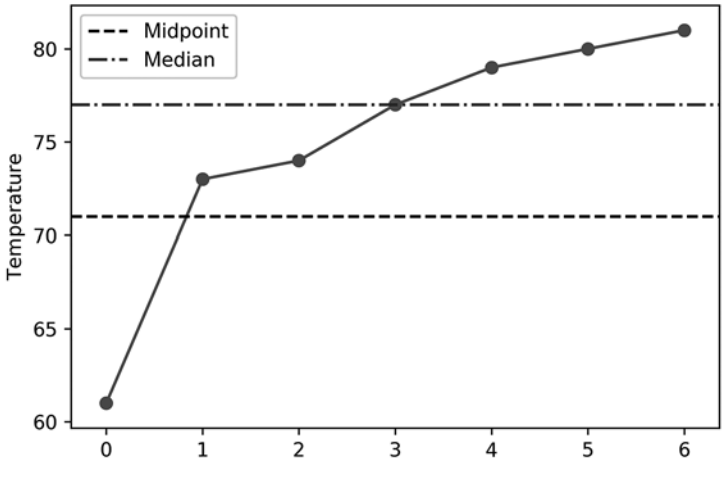


Рис. 5.5. График семи упорядоченных значений температуры. Медиана, соответствующая 77 градусам, делит значения температур пополам. При этом ее баланс слегка нарушен — она находится ближе к трем верхним значениям, чем к трем нижним

Наша медиана, равная 77 градусам, делит температурные значения пополам. Однако разделение получается неравномерным, поскольку медиана оказывается ближе к верхней тройке значений на графике. В частности, она заметно удалена от минимального значения — 61 градуса. Возможно, нам удастся сбалансировать деление, применив к медиане компенсацию. Реализуем мы ее с помощью *квадрата*

расстояния, который будет просто отражать квадрат разницы между двумя значениями (листинг 5.13). Квадрат расстояния растет квадратично по мере удаления двух значений друг от друга. Таким образом, если мы компенсируем центральную точку на основе ее расстояния до 61 градуса, то компенсация будет существенно возрастать по мере ее удаления от этого значения (рис. 5.6).

Листинг 5.13. Сдвиг центров с помощью квадрата их расстояния от минимума

```
def squared_distance(value1, value2):
    return (value1 – value2) ** 2

possible_centers = range(measurements.min(), measurements.max() + 1)
penalties = [squared_distance(center, 61) for center in possible_centers]
plt.plot(possible_centers, penalties)
plt.scatter(possible_centers, penalties)
plt.xlabel('Possible Centers')
plt.ylabel('Penalty')
plt.show()
```

Использует диапазон значений между минимальной и максимальной температурами в качестве набора возможных центров

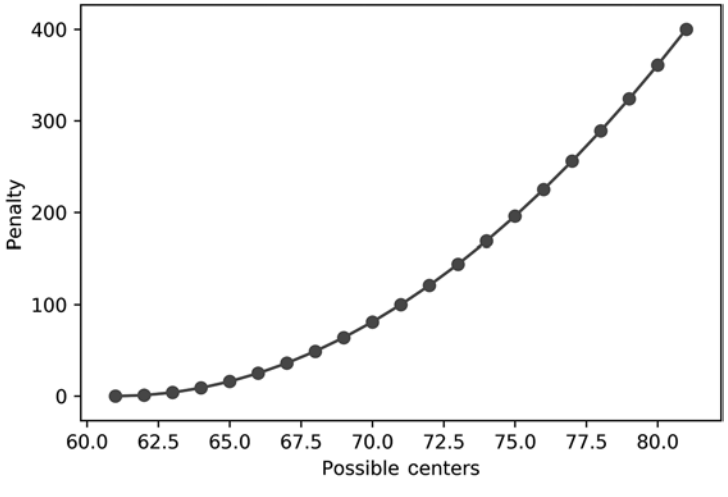


Рис. 5.6. График возможных центров, компенсированных на основе квадрата их расстояния относительно минимальной температуры 61 градус. Неудивительно, что минимальная компенсация происходит при 61 градусе. К сожалению, она не учитывает расстояния до оставшихся шести значений температур

Наш график выстраивает компенсацию для диапазона возможных центров на основе их удаления от минимума. По мере смещения центров к 61 градусу компенсация уменьшается, но их расстояние до оставшихся шести точек данных растет. Получается, нужно компенсировать каждый потенциальный центр на основе квадрата его расстояния до всех семи значений температуры. Для этого мы определим функцию суммирования квадратов расстояний, которая будет складывать квадраты расстояний между некоторым значением и массивом наших измерений

(листинг 5.14). Эта функция будет выступать в качестве новой компенсации. Построение графика возможных центров относительно их компенсаций позволит найти центр с минимальной компенсацией (рис. 5.7).

Листинг 5.14. Компенсация центров с помощью общей суммы квадратов расстояний

```
def sum_of_squared_distances(value, measurements):
    return sum(squared_distance(value, m) for m in measurements)
penalties = [sum_of_squared_distances(center, measurements)
                                        for center in possible_centers]
plt.plot(possible_centers, penalties)
plt.scatter(possible_centers, penalties)
plt.xlabel('Possible Centers')
plt.ylabel('Penalty')
plt.show()
```

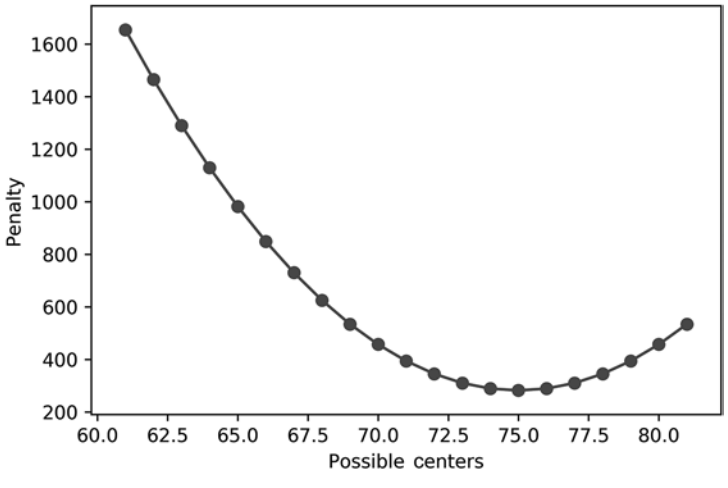


Рис. 5.7. График возможных центров, компенсированных на основе суммы квадратов их расстояний относительно всех зарегистрированных температур. Минимальная компенсация происходит при 75 градусах

Исходя из полученного графика, температура 75 градусов подразумевает наименьшую компенсацию. Мы неформально будем называть эту температуру *наименее компенсированным центром*. Обозначим его на графике горизонтальной линией (листинг 5.15; рис. 5.8).

Листинг 5.15. Отображение на графике наименее компенсированной температуры

```
least_penalized = 75
assert least_penalized == possible_centers[np.argmin(penalties)]

plt.plot(range(number_of_days), measurements)
plt.scatter(range(number_of_days), measurements)
plt.axhline(midpoint, color='k', linestyle='--', label='midpoint')
```

```
plt.axhline(median, color='g', linestyle='-.', label='median')
plt.axhline(least_penalized, color='r', linestyle='-',
            label='least penalized center')
plt.legend()
plt.ylabel('Temperature')
plt.show()
```

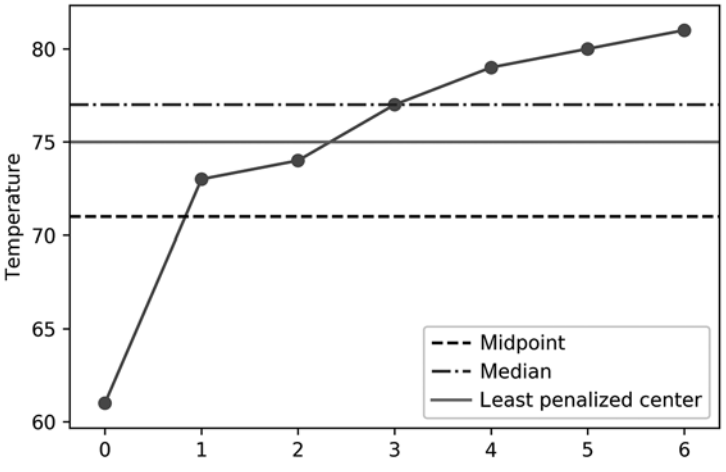


Рис. 5.8. График семи упорядоченных температур. Наименее компенсированный центр, 75 градусов, разделяет их равномерно

Центр с минимальной компенсацией делит значения температур поровну: четыре измерения оказываются выше него, а три — ниже. Таким образом, этот центр сохраняет равномерное разделение данных, одновременно обеспечивая более близкое расстояние до самой низкой из зафиксированных температур относительно медианы.

Наименее компенсированный центр является отличной мерой центральности. Он минимизирует все компенсации, примененные из-за излишнего удаления от любой заданной точки, что обеспечивает сбалансированные расстояния между ним и любой точкой данных. К сожалению, вычисление этого центра оказалось малоэффективным. Просмотр всех возможных компенсаций не является масштабируемым решением. Есть ли более эффективный способ вычисления центра? Да! Математики доказали, что погрешность суммы квадратов расстояний всегда минимизируется *средним* значением набора данных. Значит, можно вычислить наименее компенсированный центр напрямую. Нужно лишь сложить все элементы в `measurements` и разделить эту сумму на размер массива (листинг 5.16).

Листинг 5.16. Вычисление наименее компенсированного центра с помощью среднего значения

```
assert measurements.sum() / measurements.size == least_penalized
```

Сумма массива значений, разделенная на его размер, формально называется *арифметическим средним*. Неформально же это значение называют просто *средним* или *средним массива*. Среднее можно вычислить, вызвав метод `mean` для массива NumPy (листинг 5.17). Второй способ его получения — это вызов методов `np.mean` и `np.average`.

Листинг 5.17. Вычисление среднего с помощью NumPy

```
mean = measurements.mean()
assert mean == least_penalized
assert mean == np.mean(measurements)
assert mean == np.average(measurements)
```

Метод `np.average` отличается от метода `np.mean` тем, что получает на входе необязательный параметр `weights` — список численных весов, которые отражают значимость измерений относительно друг друга (листинг 5.18). Когда все веса равны, выход `np.average` ничем не отличается от выхода `np.mean`. Если же в весах появляются различия, то и результаты получаются разные.

Листинг 5.18. Передача весов в np.average

```
equal_weights = [1] * 7
assert mean == np.average(measurements, weights=equal_weights)

unequal_weights = [100] + [1] * 6
assert mean != np.average(measurements, weights=unequal_weights)
```

Параметр `weights` пригождается для вычисления среднего в наборах, содержащих повторяющиеся измерения. Предположим, мы анализируем десять показаний температуры, среди которых 75 градусов встречается восемь раз, а 77 градусов — всего один. Полный список этих измерений будет представлен как $9 \times [75] + [77]$. Можно вычислить среднее, вызвав для этого списка `np.mean`. Также его можно вычислить, вызвав `np.average([75, 77], weights=[9, 1])`, оба вычисления будут равны (листинг 5.19).

Листинг 5.19. Вычисление взвешенного среднего повторяющихся значений

```
weighted_mean = np.average([75, 77], weights=[9, 1])
print(f"The mean is {weighted_mean}")
assert weighted_mean == np.mean(9 * [75] + [77]) == weighted_mean

The mean is 75.2
```

Когда в наборе данных имеются повторы, вычисление взвешенного среднего служит своеобразным сокращением для вычисления стандартного среднего. В этом случае взаимное соотношение количеств уникальных измерений представляется соотношением их весов. Таким образом, даже если преобразовать абсолютные количества 9 и 1 в относительные веса 900 и 100, итоговое значение `weighted_mean` должно остаться неизменным. Это верно и в том случае, если веса преобразовать в относительные вероятности 0,9 и 0,1 (листинг 5.20).

Листинг 5.20. Вычисление взвешенного среднего относительных весов

```
assert weighted_mean == np.average([75, 77], weights=[900, 100])
assert weighted_mean == np.average([75, 77], weights=[0.9, 0.1])
```

Можно рассматривать вероятности как веса. Это позволит вычислять среднее любого распределения вероятностей.

5.2.1. Поиск среднего распределения вероятностей

К этому моменту мы уже довольно хорошо знакомы с биномиальным распределением 20 подбрасываний монеты. Пик этого распределения симметрично центрирован в точке десяти исходов с орлами. А как этот пик соотносится со средним своего распределения? Чтобы это выяснить, вычислим среднее, передав в параметр `weights` метода `np.average` массив `probabilities` (листинг 5.21). Затем отобразим на графике это среднее в виде вертикальной линии, разрезающей распределение пополам (рис. 5.9).

Листинг 5.21. Вычисление среднего биномиального распределения

```
num_flips = 20
interval_all_counts = range(num_flips + 1)
probabilities = stats.binom.pmf(interval_all_counts, 20, prob_head)
mean_binomial = np.average(interval_all_counts, weights=probabilities)
print(f"The mean of the binomial is {mean_binomial:.2f} heads")
plt.plot(interval_all_counts, probabilities)
plt.axvline(mean_binomial, color='k', linestyle='--')
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

Метод axvline рисует вертикальную линию в заданной координате x

```
The mean of the binomial is 10.00 heads
```

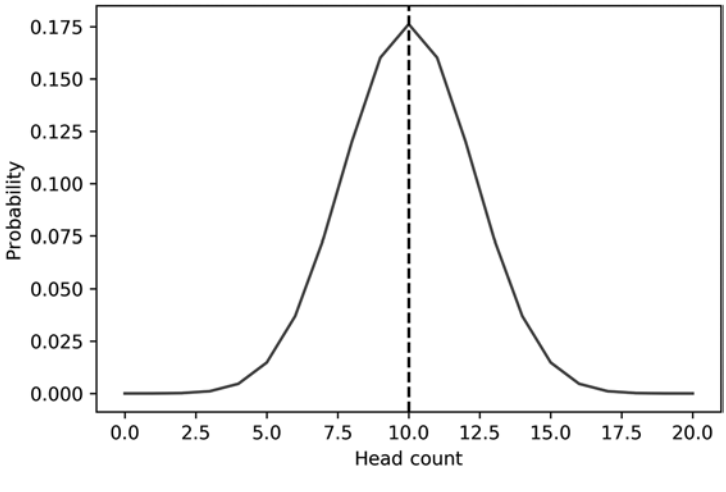


Рис. 5.9. Биномиальное распределение 20 подбрасываний монеты, разделенное пополам его средним значением. Это среднее расположено ровно в центре распределения

Средним данного распределения будут десять исходов с орлом. Изображающая его линия проходит вдоль центрального пика распределения, идеально отражая его центральность. По этой причине SciPy позволяет получить среднее любого биномиального распределения простым вызовом `stats.binom.mean` (листинг 5.22). Метод `stats.binom.mean` получает на входе два параметра: количество подбрасываний монеты и вероятность выпадения орла.

Листинг 5.22. Вычисление среднего биномиального распределения с помощью SciPy

```
assert stats.binom.mean(num_flips, 0.5) == 10
```

С помощью `stats.binom.mean` можно подробно проанализировать связь между центральностью распределения и количеством подбрасываний монеты. Далее мы отобразим на графике это среднее для диапазона количества подбрасываний от 0 до 500 (листинг 5.23; рис. 5.10).

Листинг 5.23. Отображение на графике нескольких средних распределения

```
means = [stats.binom.mean(num_flips, 0.5) for num_flips in range(500)]
plt.plot(range(500), means)
plt.xlabel('Coin Flips')
plt.ylabel('Mean')
plt.show()
```

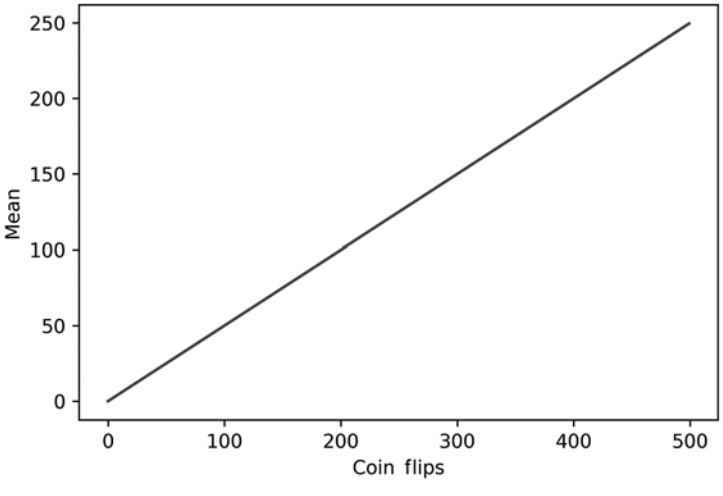


Рис. 5.10. График количества подбрасываний монеты относительно среднего биномиального распределения. Эта связь линейна. Среднее каждого биномиального распределения равно половине количества подбрасываний в нем

Количество подбрасываний монеты и среднее имеют линейную связь, в которой среднее равно половине количества подбрасываний. С учетом этого мы рассмотрим

среднее биномиального распределения одного подбрасывания монеты (обычно его называют распределением Бернулли). Количество подбрасываний монеты в распределении Бернулли равно 1, значит, его среднее будет равно 0,5 (листинг 5.24). Неудивительно, что вероятность приземления честной монеты орлом вверх равна среднему этого распределения.

Листинг 5.24. Прогнозирование среднего в распределении Бернулли

```
num_flips = 1
assert stats.binom.mean(num_flips, 0.5) == 0.5
```

Используя полученную линейную связь, можно прогнозировать среднее для распределения 1000 подбрасываний монеты (листинг 5.25). Ожидается, что в этом случае оно будет равно 500 и расположено в центре распределения. Проверим, так ли это (рис. 5.11).

Листинг 5.25. Прогнозирование среднего для распределения 1000 подбрасываний монеты

```
num_flips = 1000
assert stats.binom.mean(num_flips, 0.5) == 500

interval_all_counts = range(num_flips)
probabilities = stats.binom.pmf(interval_all_counts, num_flips, 0.5)
plt.axvline(500, color='k', linestyle='--')
plt.plot(interval_all_counts, probabilities)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

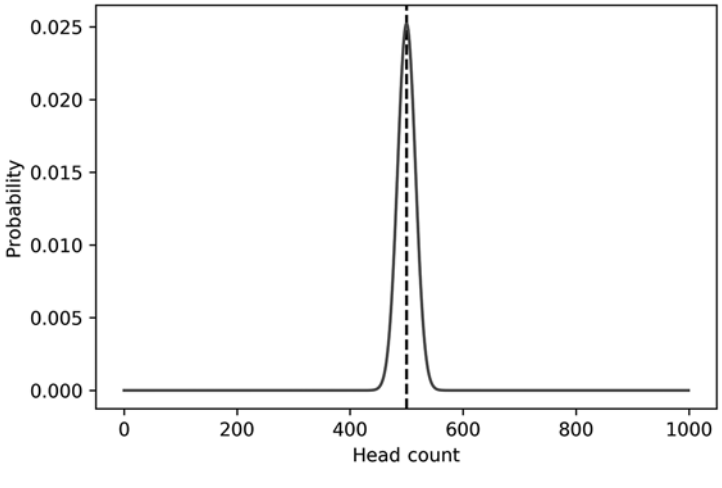


Рис. 5.11. Биномиальное распределение 1000 подбрасываний монеты, разделенное его средним, которое расположено в самом центре распределения

Среднее распределения служит прекрасной мерой центральности. Далее мы переходим к изучению использования дисперсии в качестве среднего для рассеяния.

5.3. ДИСПЕРСИЯ КАК МЕРА РАССЕЯНИЯ

Рассеяние — это разброс точек данных вокруг некоторого центрального значения. Чем меньше рассеяние, тем более прогнозируемы данные. Чем оно больше, тем колебания в данных выше. Рассмотрим сценарий, в котором измеряем летнюю температуру в Калифорнии и Кентукки. Мы соберем в произвольных местах по три показателя для каждого из этих штатов. Калифорния — очень большой штат, для которого характерен разнообразный климат, поэтому мы ожидаем увидеть в собранных данных колебания. Так и вышло, по результатам измерений у нас получилось 52, 77 и 96 градусов.

В Кентукки мы получили температуры 71, 75 и 79 градусов. Мы сохраняем все эти значения и вычисляем их средние (листинг 5.26).

Листинг 5.26. Измерение средних значений нескольких массивов температур

```
california = np.array([52, 77, 96])
kentucky = np.array([71, 75, 79])

print(f"Mean California temperature is {california.mean()}")
print(f"Mean Kentucky temperature is {california.mean()}")

Mean California temperature is 75.0
Mean Kentucky temperature is 75.0
```

Средние значения обоих массивов измерений равны 75. Выходит, что Калифорния и Кентукки имеют одинаковое центральное значение температуры. Но при этом два полученных массива далеко не равны. Температуры в Калифорнии намного более разнообразны и непредсказуемы — от 52 до 96 градусов. В то же время стабильный температурный диапазон Кентукки охватывает значения от 71 до 79 градусов. В этом случае они более тесно сосредоточены вокруг среднего. Разницу в этих двух рассеяниях мы визуализируем путем построения графиков для их массивов (листинг 5.27; рис. 5.12). Также дополнительно обозначаем на этих графиках среднее, проводя горизонтальную линию.

Листинг 5.27. Визуализация различий в рассеянии

```
plt.plot(range(3), california, color='b', label='California')
plt.scatter(range(3), california, color='b')
plt.plot(range(3), kentucky, color='r', linestyle='-.', label='Kentucky')
plt.scatter(range(3), kentucky, color='r')
plt.axhline(75, color='k', linestyle='--', label='Mean')
plt.legend()
plt.show()
```

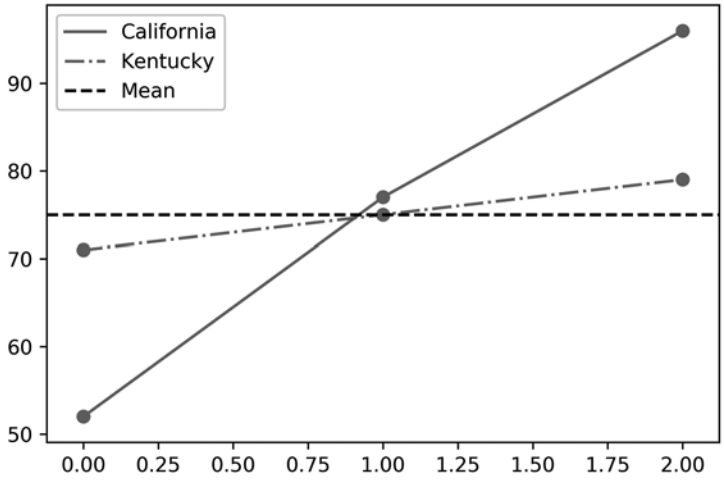


Рис. 5.12. График упорядоченных температур в Калифорнии и Кентукки. В обоих штатах среднее температур равно 75 градусам. При этом температуры в Калифорнии больше разбросаны вокруг своего среднего

На графике три температуры, зарегистрированные в Кентукки, очень близки к их среднему. При этом в Калифорнии две из трех температур удалены от своего среднего намного дальше. Эти наблюдения можно оценить количественно, если компенсировать измерения в Калифорнии. До этого мы вычисляли компенсации с помощью функции сумм квадратов расстояний. В этом случае вычислим сумму квадратов расстояний между показателями Калифорнии и их средним. Статистики называют сумму квадратов расстояний просто *суммой квадратов*. Определим функцию `sum_of_squares` и применим ее к температурам Калифорнии (листинг 5.28).

Листинг 5.28. Вычисление суммы квадратов для температур Калифорнии

```
def sum_of_squares(data):
    mean = np.mean(data)
    return sum(squared_distance(value, mean) for value in data)

california_sum_squares = sum_of_squares(california)
print(f"California's sum of squares is {california_sum_squares}")

California's sum of squares is 974.0
```

Сумма квадратов для значений из Калифорнии получилась 974. Ожидается, что в Кентукки аналогичное значение окажется намного меньше. Проверим (листинг 5.29).

Листинг 5.29. Вычисление суммы квадратов для температурных значений в Кентукки

```
kentucky_sum_squares = sum_of_squares(kentucky)
print(f"Kentucky's sum of squares is {kentucky_sum_squares}")

Kentucky's sum of squares is 32.0
```

В Кентукки сумма квадратов получилась 32. Таким образом, мы видим 30-кратную разницу между результатами этого штата и Калифорнии. Но это неудивительно, поскольку точки данных Кентукки рассеяны намного меньше. Сумма квадратов позволяет измерить это рассеяние, однако наши измерения неидеальны. Предположим, что удваиваем количество температур в массиве `california`, записав каждое показание дважды. Уровень рассеяния останется тем же, хотя сумма квадратов удвоится (листинг 5.30).

Листинг 5.30. Вычисление суммы квадратов после дублирования массива

```
california_duplicated = np.array(california.tolist() * 2)
duplicated_sum_squares = sum_of_squares(california_duplicated)
print(f"Duplicated California sum of squares is {duplicated_sum_squares}")
assert duplicated_sum_squares == 2 * california_sum_squares
```

```
Duplicated California sum of squares is 1948.0
```

Сумма квадратов — это не очень удачная мера для рассеяния, потому что на нее влияет размер массива. К счастью, это влияние легко устранить, разделив сумму квадратов на размер массива (листинг 5.31). Деление `california_sum_squares` на `california.size` дает значение, равное `duplicated_sum_squares/california_duplicated.size`.

Листинг 5.31. Деление суммы квадратов на размер массива

```
value1 = california_sum_squares / california.size
value2 = duplicated_sum_squares / california_duplicated.size
assert value1 == value2
```

Деление суммы квадратов на количество измерений дает величину, которую статистики называют *дисперсией*. В принципиальном смысле дисперсия равна среднему квадрату расстояния от среднего значения (листинг 5.32).

Листинг 5.32. Вычисление дисперсии из среднего квадрата расстояния

```
def variance(data):
    mean = np.mean(data)
    return np.mean([squared_distance(value, mean) for value in data])

assert variance(california) == california_sum_squares / california.size
```

Дисперсия для массивов `california` и `california_duplicated` равна, поскольку их уровни рассеяния идентичны (листинг 5.33).

Листинг 5.33. Вычисление дисперсии после дублирования значений массива

```
assert variance(california) == variance(california_duplicated)
```

При этом для дисперсий массивов `california` и `kentucky` сохраняется 30-кратная разница, которая обусловлена различием в рассеянии (листинг 5.34).

Листинг 5.34. Сравнение дисперсий температур Калифорнии и Кентукки

```
california_variance = variance(california)
kentucky_variance = variance(kentucky)
print(f"California Variance is {california_variance}")
print(f"Kentucky Variance is {kentucky_variance}")

California Variance is 324.6666666666667
Kentucky Variance is 10.666666666666666
```

Дисперсия является хорошей мерой рассеяния. Ее можно вычислить, вызвав `np.var` для списка Python или массива NumPy (листинг 5.35). Для массива NumPy дисперсию можно вычислить также с помощью встроенного в массив метода `var`.

Листинг 5.35. Вычисление дисперсии с помощью NumPy

```
assert california_variance == california.var()
assert california_variance == np.var(california)
```

Дисперсия зависит от среднего значения. Если мы вычисляем взвешенное среднее, то нужно также вычислить взвешенную дисперсию. Делается это просто: как уже говорилось, дисперсия — это просто среднее всех квадратов расстояний от среднего значения, значит, взвешенная дисперсия — это взвешенное среднее всех квадратов расстояний от взвешенного среднего значения. Далее мы определим функцию `weighted_variance`, получающую на входе два параметра: список данных и веса. На основе этих параметров она вычислит взвешенное среднее и использует метод `np.average` для получения взвешенного среднего квадратов расстояний от этого среднего (листинг 5.36).

Листинг 5.36. Вычисление взвешенной дисперсии с помощью np.average

```
def weighted_variance(data, weights):
    mean = np.average(data, weights=weights)
    squared_distances = [squared_distance(value, mean) for value in data]
    return np.average(squared_distances, weights=weights)

assert weighted_variance([75, 77], [9, 1]) == np.var(9 * [75] + [77])
```

weighted_variance позволяет рассматривать дублированные элементы как веса

Функция `weighted_variance` может получать на входе массив вероятностей. Это позволяет вычислить дисперсию любого распределения вероятностей.

5.3.1. Определение дисперсии распределения вероятностей

Далее мы вычислим дисперсию биномиального распределения, связанного с 20 подбрасываниями честной монеты. Это действие выполняется присваиванием массива `probabilities` параметру `weights` в `weighted_variance` (листинг 5.37).

Листинг 5.37. Вычисление дисперсии биномиального распределения

```
interval_all_counts = range(21)
probabilities = stats.binom.pmf(interval_all_counts, 20, prob_head)
variance_binomial = weighted_variance(interval_all_counts, probabilities)
print(f"The variance of the binomial is {variance_binomial:.2f} heads")

The variance of the binomial is 5.00 heads
```

Дисперсия этого биномиального распределения — 5, что равно половине его среднего. Эту дисперсию можно вычислить более прямым путем с помощью метода SciPy `stats.binom.var` (листинг 5.38).

Листинг 5.38. Вычисление дисперсии биномиального распределения с помощью SciPy

```
assert stats.binom.var(20, prob_head) == 5.0
assert stats.binom.var(20, prob_head) == stats.binom.mean(20, prob_head) / 2
```

Применив метод `stats.binom.var`, можно подробно проанализировать связь между биномиальным распределением и количеством подбрасываний монеты. Давайте построим график дисперсии распределения для диапазона подбрасываний монеты от 0 до 500 (листинг 5.39; рис. 5.13).

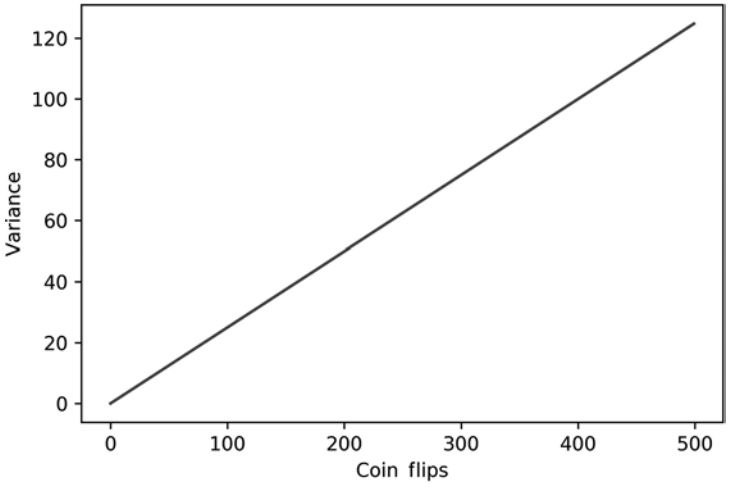


Рис. 5.13. Количество подбрасываний монеты относительно дисперсии распределения. Их связь линейна. Дисперсия каждого биномиального распределения равна 1/4 количества подбрасываний монеты

Листинг 5.39. Построение графика нескольких дисперсий биномиального распределения

```
variances = [stats.binom.var(num_flips, prob_head)
             for num_flips in range(500)]
plt.plot(range(500), variances)
```

```
plt.xlabel('Coin Flips')
plt.ylabel('Variance')
plt.show()
```

Дисперсия биномиального распределения, как и его среднее, связана с количеством подбрасываний монеты линейно. При этом она равняется 1/4 от этого количества. Таким образом, дисперсия распределения Бернулли — 0,25, так как количество подбрасываний в этом случае 1. Следуя этой логике, можно ожидать, что дисперсия распределения 1000 подбрасываний монеты будет равна 250 (листинг 5.40).

Листинг 5.40. Прогнозирование дисперсий биномиальных распределений

```
assert stats.binom.var(1, 0.5) == 0.25
assert stats.binom.var(1000, 0.5) == 250
```

ТИПИЧНЫЕ МЕТОДЫ SCIPY ДЛЯ АНАЛИЗА БИНОМИАЛЬНЫХ РАСПРЕДЕЛЕНИЙ

- `stats.binom.mean(num_flips, prob_heads)` — возвращает среднее распределения при количестве подбрасываний `num_flips` и вероятности выпадения орлов `prob_heads`.
- `stats.binom.var(num_flips, prob_heads)` — возвращает дисперсию распределения при количестве подбрасываний `num_flips` и вероятности выпадения орла `prob_heads`.
- `stats.binom.pmf(head_count_int, num_flips, prob_heads)` — возвращает вероятность получения `head_count_int` исходов с орлом из `num_flips` подбрасываний монеты. Вероятность выпадения орла при одном подбрасывании монеты устанавливается равной `prob_heads`.
- `stats.binom.pmf(head_count_array, num_flips, prob_heads)` — возвращает массив вероятностей для биномиального распределения. Получаются эти вероятности выполнением `stats.binom.pmf(e, num_flips, prob_head)` для каждого элемента `e` из `head_count_array`.
- `stats.binom_test(head_count_int, num_flips, prob_heads)` — возвращает вероятность того, что `num_flips` подбрасываний монеты дадут не менее `head_count_int` орлов либо `tail_count_int` решек. Вероятность выпадения орла при одном подбрасывании монеты устанавливается равной `prob_heads`.

Дисперсия — это мощный инструмент измерения рассеяния данных. Тем не менее статистики зачастую используют альтернативное средство, которое называют *стандартным отклонением*. Стандартное отклонение равно квадратному корню из дисперсии и вычисляется вызовом `np.std` (листинг 5.41). Получается, что возведение в квадрат результата `np.std` даст значение дисперсии.

Листинг 5.41. Вычисление стандартного отклонения

```
data = [1, 2, 3]
standard_deviation = np.std(data)
assert standard_deviation ** 2 == np.var(data)
```

Иногда вместо дисперсии используется стандартное отклонение, что упрощает отслеживание единиц измерения. У всех измерений есть единицы. К примеру, температуры ранее приводились в градусах Фаренгейта. Когда мы возводили в квадрат расстояния между температурами и их средним, то возводили в квадрат и единицы их измерения. Следовательно, получаемая дисперсия выражалась в градусах Фаренгейта в квадрате. Подобные возведенные в квадрат единицы измерения не так просто концептуализировать. Извлечение же квадратного корня преобразует единицы измерения обратно в градусы Фаренгейта: стандартное отклонение в градусах Фаренгейта проще интерпретировать, чем дисперсию.

Среднее значение и стандартное отклонение — невероятно важные измерения, которые позволяют делать следующее.

- *Сравнивать числовые наборы данных*. Предположим, что даны два массива температур, зарегистрированных в течение двух последовательных летних сезонов. Можно количественно выразить разницу между этими записями, используя стандартное отклонение и среднее значение.
- *Сравнивать распределения вероятностей*. Предположим, что две лаборатории по изучению климата опубликовали распределения вероятностей. Каждое из этих распределений охватывает все вероятности температур в течение типичного летнего дня. Можно обобщить различия между этими двумя распределениями, сравнив их средние значения и стандартные отклонения.
- *Сравнить числовой набор данных с распределением вероятности*. Предположим, хорошо известное распределение вероятности охватывает набор вероятностей температур, собранных за десятилетие. Однако последние зарегистрированные температуры противоречат этим данным. Является ли это признаком изменения климата или же просто случайной аномалией? Выяснить это можно, сопоставив центральность и рассеяние известного распределения и набора данных температур.

Третий вариант использования этого приема лежит в основе многих статистических исследований. В последующих главах мы научимся сравнивать наборы данных с распределением вероятностей. Многие из наших сравнений ориентированы на нормальное распределение, которое зачастую встречается в анализе данных. Удобно, что характерная для этого распределения колоколообразная кривая является прямой функцией среднего значения и стандартного отклонения. Вскоре мы используем SciPy вместе с этими двумя параметрами, чтобы лучше понять значительность нормальной кривой.

РЕЗЮМЕ

- *Функция вероятности* сопоставляет входные целочисленные значения с вероятностью их возникновения.
- Функцию вероятности для биномиального распределения можно сгенерировать вызовом `stats.binom.pmf`.
- *Арифметическое среднее* является хорошей мерой центральности набора данных. Оно минимизирует *сумму квадратов* относительно этого набора данных. Невзвешенное среднее можно вычислить, сложив значения набора данных и разделив результат на его размер. Взвешенное же вычисляется передачей `weights` в `np.average`. Взвешенное среднее биномиального распределения растет линейно по мере увеличения числа подбрасываний монеты.
- *Дисперсия* — это хорошая мера рассеяния набора данных. Она равняется среднему квадрату расстояния от точек данных до их среднего. Взвешенная дисперсия биномиального распределения при увеличении количества подбрасываний монеты возрастает линейно.
- *Стандартное отклонение* — это альтернативная мера рассеяния, которая вычисляется как корень из дисперсии. Стандартное отклонение сохраняет единицы измерения набора данных.

6 Составление прогнозов с помощью центральной предельной теоремы и SciPy

В этой главе

- ✓ Анализ нормальной кривой с помощью библиотеки SciPy.
- ✓ Прогнозирование среднего и дисперсии с помощью центральной предельной теоремы.
- ✓ Прогнозирование характеристик совокупности с помощью центральной предельной теоремы.

Нормальное распределение — это колоколообразная кривая, с которой мы познакомились в главе 3. Эта кривая, согласно центральной предельной теореме, возникает естественным образом при анализе случайных данных. Ранее уже отмечалось, что, согласно этой теореме, повторно вычисляемые частоты формируют график нормальной кривой. Более того, теорема прогнозирует сужение этой кривой по мере увеличения размера каждого набора вычисляемых частот. Иными словами, при увеличении размера серии испытаний стандартное отклонение распределения их исходов будет уменьшаться.

В основе всей классической статистики лежит центральная предельная теорема. В текущей главе мы изучим ее более подробно, задействуя вычислительную мощь SciPy. В итоге научимся использовать теорему для составления прогнозов на основе ограниченных данных.

6.1. УПРАВЛЕНИЕ НОРМАЛЬНЫМ РАСПРЕДЕЛЕНИЕМ С ПОМОЩЬЮ SCIPY

В главе 3 было показано, что при анализе случайных подбрасываний монеты получается нормальная кривая. Давайте сгенерируем нормальное распределение, построив гистограмму серии подбрасываний монеты. Входные данные для гистограммы будут содержать 100 000 частот исходов с орлом. Вычисление этих частот потребует проанализировать совокупность подбрасываний монеты 100 000 раз. Каждая такая совокупность будет отражена массивом из 0 и 1, представляющим 10 000 подбрасываний монеты. Длину этого массива мы будем называть размером совокупности. Если разделить сумму значений этой совокупности на ее размер, то мы вычислим наблюдаемую частоту исходов с орлом. В принципиальном смысле эта частота равна простому взятию среднего совокупности.

Код листинга 6.1 вычисляет частоту исходов с орлом одной случайной совокупности и подтверждает ее связь со средним. Заметьте, что каждая точка данных в этой совокупности получена из распределения Бернулли.

Листинг 6.1. Вычисление частот исходов с орлом из среднего

```
np.random.seed(0)
sample_size = 10000
sample = np.array([np.random.binomial(1, 0.5) for _ in range(sample_size)])
head_count = sample.sum()
head_count_frequency = head_count / sample_size
assert head_count_frequency == sample.mean()
```

Частота исходов с орлом идентична среднему совокупности

Конечно же, можно вычислить все 100 000 частот исходов с орлом в одной строке кода (листинг 6.2), о чем говорилось в главе 3.

Листинг 6.2. Вычисление 100 000 частот исходов с орлом

```
np.random.seed(0)
frequencies = np.random.binomial(sample_size, 0.5, 100000) / sample_size
```

Каждая вычисленная частота равна среднему из 10 000 случайно подброшенных монет. Поэтому переименовываем переменную частоты в `sample_means`, после чего визуализируем данные `sample_means` в виде гистограммы (листинг 6.3; рис. 6.1).

Листинг 6.3. Визуализация средних значений совокупностей на гистограмме

```
sample_means = frequencies
likelihoods, bin_edges, _ = plt.hist(sample_means, bins='auto',
                                     edgecolor='black', density=True)
plt.xlabel('Binned Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

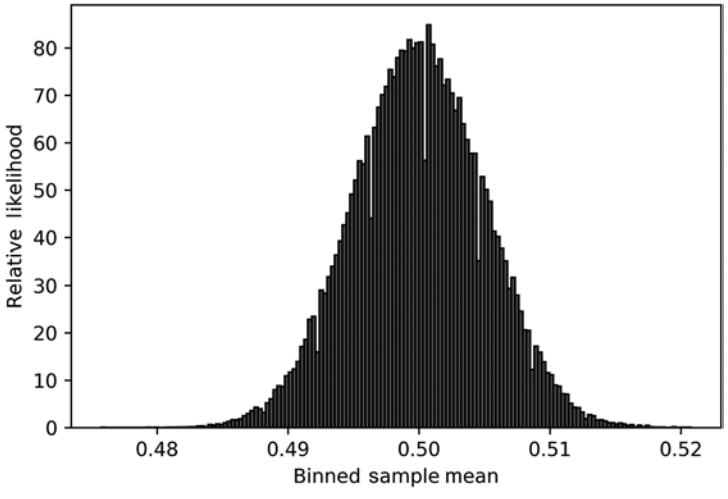


Рис. 6.1. Гистограмма 100 000 средних в сопоставлении с их относительной вероятностью. Форма гистограммы напоминает колоколообразное нормальное распределение

Гистограмма имеет форму нормального распределения. Теперь вычислим для этого распределения среднее арифметическое и стандартное отклонение (листинг 6.4).

Листинг 6.4. Вычисление среднего арифметического и стандартного отклонения гистограммы

```
mean_normal = np.average(bin_edges[:-1], weights=likelihoods)
var_normal = weighted_variance(bin_edges[:-1], likelihoods)
std_normal = var_normal ** 0.5
print(f"Mean is approximately {mean_normal:.2f}")
print(f"Standard deviation is approximately {std_normal:.3f}")

Mean is approximately 0.50
Standard deviation is approximately 0.005
```

Среднее распределения равно приблизительно 0,5, а его стандартное отклонение — приблизительно 0,005. В нормальном распределении эти значения можно вычислить напрямую из пика, для чего потребуются только его координаты x и y. Значение x равно среднему распределения, а стандартное отклонение — обратному значению y, умноженному на $(2\pi)^{1/2}$. Эти свойства выводятся в результате математического анализа нормальной кривой. Код листинга 6.5 вычисляет среднее арифметическое и стандартное отклонение, используя только координаты пика.

Листинг 6.5. Вычисление среднего арифметического и стандартного отклонения на основе координат пика

```
import math
peak_x_value = bin_edges[likelihoods.argmax()]
print(f"Mean is approximately {peak_x_value:.2f}")
peak_y_value = likelihoods.max()
std_from_peak = (peak_y_value * (2* math.pi) ** 0.5) ** -1
print(f"Standard deviation is approximately {std_from_peak:.3f}")

Mean is approximately 0.50
Standard deviation is approximately 0.005
```

Эти величины можно вычислить также, просто вызвав `stats.norm.fit(sample_means)` (листинг 6.6). Этот метод SciPy возвращает два параметра, необходимых для воссоздания нормального распределения, сформированного нашими данными.

Листинг 6.6. Вычисление среднего арифметического и стандартного отклонения с помощью stats.norm.fit

```
fitted_mean, fitted_std = stats.norm.fit(sample_means)
print(f"Mean is approximately {fitted_mean:.2f}")
print(f"Standard deviation is approximately {fitted_std:.3f}")

Mean is approximately 0.50
Standard deviation is approximately 0.005
```

Полученные величины мы используем для воссоздания нормальной кривой, сгенерировав ее вызовом `stats.norm.pdf(bin_edges, fitted_ mean, fitted_std)`. Метод SciPy `stats.norm.pdf` представляет *плотность распределения вероятностей* нормального распределения. Плотность распределения вероятностей подобна функции вероятностей, но имеет одно ключевое отличие — возвращает не вероятности, а относительные вероятности. Как говорилось в главе 2, относительные вероятности — это значения по оси *Y* кривой, общая площадь которой равняется 1. В отличие от вероятностей относительные вероятности могут быть равны значениям больше 1. Несмотря на это, общая площадь под интервалом относительных вероятностей продолжает равняться вероятности получения в данном интервале случайного значения.

Далее мы вычислим относительные вероятности с помощью `stats.norm.pdf` (листинг 6.7), после чего отрисуем их вместе с гистограммой исходов подбрасываний монеты (рис. 6.2).

Листинг 6.7. Вычисление нормальной кривой относительных вероятностей с помощью stats.norm.pdf

```
normal_likelihoods = stats.norm.pdf(bin_edges, fitted_mean, fitted_std)
plt.plot(bin_edges, normal_likelihoods, color='k', linestyle='--',
         label='Normal Curve')
```

```
plt.hist(sample_means, bins='auto', alpha=0.2, color='r', density=True)
plt.legend()
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

Параметр alpha делает гистограмму более прозрачной для повышения контраста между гистограммой и кривой относительных вероятностей

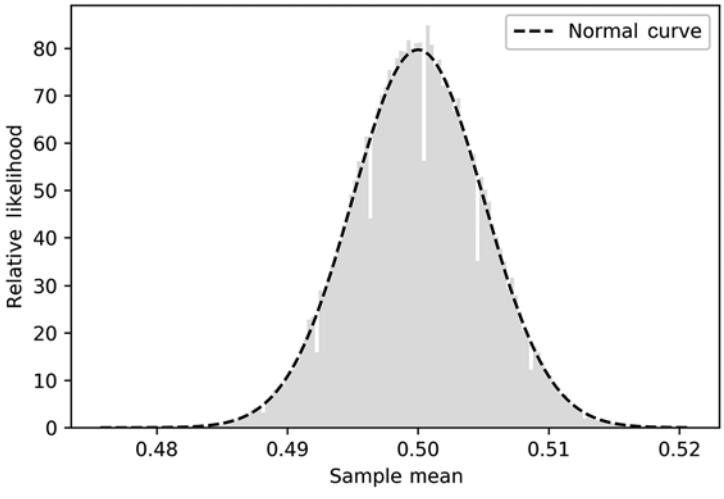


Рис. 6.2. Гистограмма, на которую наложена нормальная кривая. Параметры, определяющие нормальную кривую, были вычислены с помощью SciPy. При этом нормальная кривая четко ложится на гистограмму

Вычисленная кривая отлично наложилась на гистограмму. Пик этой кривой расположился на позиции 0,5 по оси *X* и возвышается примерно до позиции 80 по оси *Y*. Напомню, что координаты *x* и *y* пика являются прямой функцией `fitted_mean` и `fitted_std`. Чтобы подчеркнуть эту важную связь, проделаем простое упражнение — сдвинем пик на 0,01 единицы вправо, одновременно удвоив его высоту (листинг 6.8; рис. 6.3). Как произвести этот сдвиг? Центральная ось пика равна среднему, поэтому мы скорректируем входное среднее на `fitted_mean +0.01`. Кроме того, высота пика обратно пропорциональна стандартному отклонению, а значит, ввод `fitted_std / 2` должен его высоту удваивать.

Листинг 6.8. Манипулирование координатами пика нормальной кривой

```
adjusted_likelihoods = stats.norm.pdf(bin_edges, fitted_mean + 0.01,
                                      fitted_std / 2)
plt.plot(bin_edges, adjusted_likelihoods, color='k', linestyle='--')
plt.hist(sample_means, bins='auto', alpha=0.2, color='r', density=True)
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

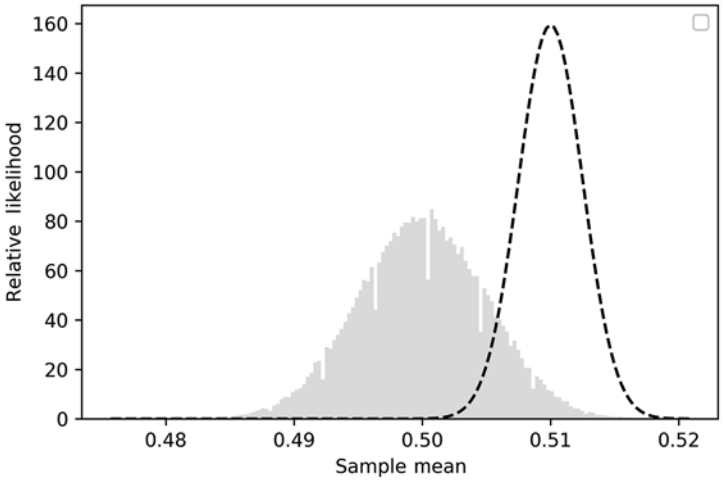


Рис. 6.3. Измененная нормальная кривая, чей центр смещен вправо от гистограммы на 0,01 единицы. Пик кривой стал вдвое выше пика гистограммы. Для реализации этих изменений понадобилось скорректировать среднее арифметическое и стандартное отклонение гистограммы

6.1.1. Сравнение двух нормальных кривых

SciPy позволяет анализировать и корректировать форму нормального распределения на основе входных параметров. Кроме того, значения этих входных параметров зависят от того, как мы обрабатываем случайные данные. Увеличим размер совокупности подбрасываний монеты вчетверо — до 40 000 — и построим график итогового изменения распределения. Код листинга 6.9 сравнивает формы старого и нового нормальных распределений, которые мы обозначим A и B соответственно (рис. 6.4).

Листинг 6.9. Построение графика двух кривых для разных размеров совокупности

```
np.random.seed(0)
new_sample_size = 40000
new_head_counts = np.random.binomial(new_sample_size, 0.5, 100000)
new_mean, new_std = stats.norm.fit(new_head_counts / new_sample_size)
new_likelihoods = stats.norm.pdf(bin_edges, new_mean, new_std)
plt.plot(bin_edges, normal_likelihoods, color='k', linestyle='--',
         label='A: Sample Size 10K')
plt.plot(bin_edges, new_likelihoods, color='b', label='B: Sample Size 40K')
plt.legend()
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

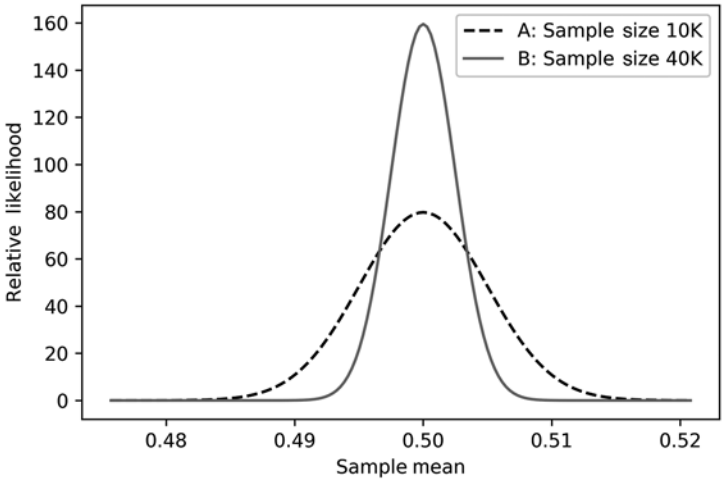


Рис. 6.4. Два нормальных распределения, сгенерированных на основе данных о подбрасываниях монеты. Распределение A получилось при 10 000 подбрасываний в каждой совокупности, а распределение B — при 40 000. Оба распределения центрированы вокруг среднего 0,5. Однако распределение B намного менее рассеяно вокруг своего центра, а его пик вдвое выше пика распределения A. Учитывая связь между высотой пика и дисперсией, можно сделать вывод, что дисперсия распределения B составляет 1/4 от дисперсии распределения A

Оба нормальных распределения центрированы вокруг среднего арифметического совокупности, равного 0,5. Однако распределение с бóльшим размером этой совокупности оказалось сильнее сконцентрировано вокруг своего пика. Это согласуется с тем, что мы видели в главе 3. Тогда мы наблюдали, что по мере увеличения размера совокупности расположение пика остается постоянным, а площадь вокруг него сужается. Это сужение пика ведет к уменьшению диапазона доверительного интервала. Доверительный интервал представляет диапазон возможных значений, охватывающий истинную вероятность исхода с орлом. Ранее мы использовали эти интервалы для оценки вероятности выпадения орла на основе частот исходов с орлом, отраженных на оси *X*. Теперь же ось *X* представляет средние совокупности, каждое из которых идентично частоте исходов с орлом. Таким образом, можно применять эти средние для нахождения вероятности выпадения орла. Также напомню, что все генеральные совокупности подбрасываний монеты были получены из распределения Бернулли. Недавно было показано, что среднее распределения Бернулли равно вероятности выпадения орла, значит, среднее каждой совокупности служит приблизительной оценкой истинного среднего распределения Бернулли. Доверительный интервал можно интерпретировать как диапазон вероятных значений, охватывающий истинное среднее распределения Бернулли.

Теперь вычислим доверительный интервал 95 % для истинного среднего распределения Бернулли с помощью нормального распределения B. До этого мы вручную вычисляли доверительный интервал 95 %, анализируя площадь кривой вокруг ее пика. Однако SciPy позволяет автоматически извлекать этот диапазон с помощью вызова `stats.norm.interval(0.95, mean, std)` (листинг 6.10). Этот метод возвращает интервал, охватывающий 95 % площади под нормальным распределением, определяемым `mean` и `std`.

Листинг 6.10. Вычисление доверительного интервала с помощью SciPy

```
mean, std = new_mean, new_std
start, end = stats.norm.interval(0.95, mean, std)
print(f"The true mean of the sampled binomial distribution is between
     {start:.3f} and {end:.3f}")
```

```
The true mean of the sampled binomial distribution is between 0.495 and 0.505
```

Здесь мы на 95 % уверены, что истинное среднее распределения Бернулли находится между 0,495 и 0,505. На деле же это среднее в точности равно 0,5, в чем можно убедиться с помощью SciPy (листинг 6.11).

Листинг 6.11. Подтверждение среднего распределения Бернулли

```
assert stats.binom.mean(1, 0.5) == 0.5
```

Далее мы попробуем оценить дисперсию распределения Бернулли на основе нормальных кривых. На первый взгляд это кажется непростой задачей. Несмотря на то что средние двух графиков распределений остаются равными 0,5, их дисперсии заметно смещаются. Относительный сдвиг дисперсии можно оценить, сравнив пики. Пик распределения B вдвое выше пика распределения A. Эта высота обратно пропорциональна стандартному отклонению, значит, стандартное отклонение распределения B вдвое меньше, чем у A. Поскольку стандартное отклонение — это квадратный корень из дисперсии, можно сделать вывод, что дисперсия распределения B равна 1/4 дисперсии распределения A. Таким образом, увеличение размера совокупности в четыре раза, с 10 000 до 40 000, ведет к четырехкратному увеличению дисперсии (листинг 6.12).

Листинг 6.12. Анализ сдвига дисперсии после увеличения числа испытаний

```
variance_ratio = (new_std ** 2) / (fitted_std ** 2)
print(f"The ratio of variances is approximately {variance_ratio:.2f}")
```

```
The ratio of variances is approximately 0.25
```

ТИПИЧНЫЕ МЕТОДЫ SCIPY ДЛЯ АНАЛИЗА НОРМАЛЬНОЙ КРИВОЙ

- `stats.norm.fit(data)` — возвращает среднее арифметическое и стандартное отклонение, необходимое для сопоставления нормальной кривой с `data`.
- `stats.norm.pdf(observation, mean, std)` — возвращает относительную вероятность, сопоставленную с одним значением нормальной кривой, определяемой средним `mean` и стандартным отклонением `std`.
- `stats.norm.pdf(observation_array, mean, std)` — возвращает массив относительных вероятностей нормальной кривой, которые вычисляются выполнением `stats.norm.pdf(e, mean, std)` для каждого элемента e массива `observation_array`.
- `stats.norm.interval(x_percent, mean, std)` — возвращает доверительный интервал `x_percent`, определяемый средним `mean` и стандартным отклонением `std`.

Получается, что дисперсия обратно пропорциональна размеру совокупности. Если это так, то уменьшение размера совокупности в четыре раза, с 10 000 до 2500 подбрасываний, должно привести к четырехкратному увеличению дисперсии. Код листинга 6.13 генерирует ряд серий исходов с орлом, используя размер совокупности 2500, и подтверждает этот факт.

Листинг 6.13. Анализ сдвига дисперсии после уменьшения совокупности

```
np.random.seed(0)
reduced_sample_size = 2500
head_counts = np.random.binomial(reduced_sample_size, 0.5, 100000)
_, std = stats.norm.fit(head_counts / reduced_sample_size)
variance_ratio = (std ** 2) / (fitted_std ** 2)
print(f"The ratio of variances is approximately {variance_ratio:.1f}")

The ratio of variances is approximately 4.0
```

Доказано: четырехкратное уменьшение размера совокупности ведет к четырехкратному увеличению дисперсии. Значит, если уменьшить размер совокупности с 10 000 до 1, то можно ожидать увеличения дисперсии в 10 000 раз. Такая дисперсия при размере совокупности 1 должна равняться `(fitted_std ** 2) * 10000` (листинг 6.14).

Листинг 6.14. Прогнозирование дисперсии при размере совокупности 1

```
estimated_variance = (fitted_std ** 2) * 10000
print(f"Estimated variance for a sample size of 1 is
     {estimated_variance:.2f}")

Estimated variance for a sample size of 1 is 0.25
```

Предполагаемая дисперсия при размере совокупности 1 равна 0,25. Однако если бы размер был 1, тогда массив `sample_means` являлся бы просто последовательностью случайно зарегистрированных 1 и 0. По определению такой массив представлял бы вывод распределения Бернулли, и выполнение `sample_means.var` аппроксимировалось бы к дисперсии распределения Бернулли. Таким образом, оценочная дисперсия для совокупности размером 1 будет равна дисперсии распределения Бернулли, которая составляет 0,25 (листинг 6.15).

Листинг 6.15. Подтверждение прогнозируемой дисперсии при размере совокупности 1

```
assert stats.binom.var(1, 0.5) == 0.25
```

Мы только что использовали нормальное распределение для вычисления дисперсии и среднего распределения Бернулли, на основе которого делали анализ. Давайте еще раз пройдемся по цепочке проделанных шагов.

1. Вычисление 1 и 0 для распределения Бернулли.
2. Группировка каждой последовательности из 1 и 0 размером `sample_size` в одну совокупность.
3. Вычисление среднего для каждой совокупности.
4. Все вместе средние совокупностей дали нормальную кривую, для которой было найдено среднее арифметическое и стандартное отклонение.
5. Среднее этой нормальной кривой равнялось среднему распределения Бернулли.
6. Дисперсия этой нормальной кривой, умноженная на размер совокупности, равнялась дисперсии распределения Бернулли.

А что, если выполнить вычисления на основе другого распределения, не Бернулли? Сможем ли мы по-прежнему оценить среднее и дисперсию с помощью вычисления случайных величин? Да! Согласно центральной предельной теореме, вычисление средних значений почти для любого распределения будет давать нормальную кривую. К ним относятся следующие распределения.

- *Распределение Пуассона* (`stats.poisson.pmf`). Обычно используется для моделирования:
  - количества посетителей магазина в час;
  - количества переходов по онлайн-объявлениям в секунду.
- *Гамма-распределение* (`scipy.stats.gamma.pdf`). Обычно используется для моделирования:
  - месячного количества осадков в регионе;
  - дефолтов по банковским кредитам в зависимости от их размера.

- *Лог-нормальное* распределение (`scipy.stats.lognorm.pdf`). Обычно используется для моделирования:
  - колебаний цен на бирже;
  - инкубационного периода болезней.
- Множественные распределения, встречающиеся в природе, которым еще не было дано имя.

ВНИМАНИЕ

В некоторых граничных случаях анализ не ведет к построению нормальной кривой. Это иногда случается с распределением Парето, с помощью которого моделируют неравенство доходов.

Построив нормальную кривую, можно анализировать с ее помощью лежащее в основе распределение. Среднее нормальной кривой аппроксимируется к среднему внутреннего распределения. Помимо этого, дисперсия нормальной кривой, умноженная на размер генеральной совокупности, аппроксимируется к дисперсии внутреннего распределения.

ПРИМЕЧАНИЕ

Иными словами, если мы выполним вычисление на основе распределения с дисперсией var, то получим нормальную кривую с дисперсией sample_size / var. По мере приближения размера совокупности к бесконечности дисперсия нормальной кривой начинает стремиться к нулю. При нулевой дисперсии нормальная кривая сворачивается в одну вертикальную линию, расположенную в позиции среднего. Это свойство можно применять для выведения закона больших чисел, с которым мы познакомились в главе 2.

На связи между нормальным распределением и свойствами лежащего в его основе распределения базируется вся статистика. Используя эту взаимосвязь, можно с помощью нормальной кривой оценивать среднее и дисперсию практически любого распределения путем случайного моделирования.

6.2. ОПРЕДЕЛЕНИЕ СРЕДНЕГО И ДИСПЕРСИИ СОВОКУПНОСТИ С ПОМОЩЬЮ СЛУЧАЙНОГО МОДЕЛИРОВАНИЯ

Предположим, мы получили задание вычислить средний возраст населения города, в котором проживают 50 000 человек. Код листинга 6.16 симулирует возраст каждого жителя с помощью метода `np.random.randint`.

Листинг 6.16. Генерация случайной популяции

```
np.random.seed(0)
population_ages = np.random.randint(1, 85, size=50000)
```

Как теперь вычислить средний возраст всех этих жителей? Один довольно хлопотный подход предполагает проведение переписи граждан. Можно записать все 50 000 возрастов и вычислить их среднее. Оно будет охватывать все население города и поэтому называется *средним по генеральной совокупности*. При этом дисперсия всего населения в таком случае называется *дисперсией генеральной совокупности*. Давайте быстро вычислим оба эти показателя для нашего симулированного города (листинг 6.17).

Листинг 6.17. Вычисление среднего значения и дисперсии генеральной совокупности

```
population_mean = population_ages.mean()
population_variance = population_ages.var()
```

При наличии симулированных данных вычислить среднее по совокупности легко. Однако в реальной жизни получение этих данных окажется очень затратным по времени занятием. При отсутствии дополнительных ресурсов опросить всех жителей города будет крайне сложно.

Более простым решением будет опросить десять случайных людей. Мы запишем возраст представителей этой случайной выборки, после чего вычислим их среднее значение. Давайте симулируем процесс сэмплирования, взяв десять случайных возрастов с помощью метода `np.random.choice` (листинг 6.18). Выполнение `np.random.choice(age, size=sample_size)` приведет к возврату массива из десяти случайно отобранных возрастов. Па завершении этого процесса мы вычислим среднее для полученного массива.

Листинг 6.18. Симуляция десяти опрошенных людей

```
np.random.seed(0)
sample_size = 10
sample = np.random.choice(population_ages, size=sample_size)
sample_mean = sample.mean()
```

Конечно же, среднее нашей выборки будет шумным и неточным. Измерить этот шум можно, найдя различие в процентах между `sample_mean` и `population_mean` (листинг 6.19).

Листинг 6.19. Сравнение среднего по выборке и среднего по совокупности

```
percent_diff = lambda v1, v2: 100 * abs(v1 – v2) / v2
percent_diff_means = percent_diff(sample_mean, population_mean)
print(f"There is a {percent_diff_means:.2f} percent difference between means.")

There is a 27.59 percent difference between means
```

Разница между средним по выборке и средним по совокупности составляет примерно 27 %. Очевидно, что такой выборки недостаточно для адекватной оценки среднего значения — нужно увеличить количество образцов. Попробуем увеличить их число до 1000. Это выглядит разумной альтернативой опросу всех 50 000 человек, но и 1000 жителей опросить не так уж просто. Даже если предположить идеальный сценарий, в котором на каждого из них уходит по две минуты, то в целом потребуется

восемь часов. Гипотетически можно оптимизировать затраты времени, наладив процесс параллельного опроса. Мы разместим в местной газете объявление о поиске 100 добровольцев, каждый из которых опросит десять случайных людей и отправит нам вычисленное среднее их возрастов. Таким образом, получим 100 средних значений по выборкам, в общем представляющим 1000 опросов.

ПРИМЕЧАНИЕ

Каждый доброволец отправит нам среднее по своей выборке. В теории эти средние значения будут предпочтительнее, и вот почему. Во-первых, они не требуют столько же памяти, сколько полноценные данные. Во-вторых, эти средние можно отобразить в виде гистограммы, чтобы проверить равенство размеров их выборок. Если полученная гистограмма не аппроксимируется к нормальной кривой, значит, потребуется сделать дополнительные выборки.

Код листинга 6.20 симулирует запланированный нами опрос.

Листинг 6.20. Вычисление средних по выборкам для 1000 человек

```
np.random.seed(0)
sample_means = [np.random.choice(population_ages, size=sample_size).mean()
                for _ in range(100)]
```

Согласно центральной предельной теореме, гистограмма средних выборок должна напоминать нормальное распределение. Кроме того, среднее нормального распределения должно аппроксимироваться к среднему совокупности. Подтвердить это можно, подогнав средние выборок под нормальное распределение (листинг 6.21; рис. 6.5).

Листинг 6.21. Сопоставление средних выборок с нормальной кривой

```
likelihoods, bin_edges, _ = plt.hist(sample_means, bins='auto', alpha=0.2,
                                     color='r', density=True)
mean, std = stats.norm.fit(sample_means)
normal_likelihoods = stats.norm.pdf(bin_edges, mean, std)
plt.plot(bin_edges, normal_likelihoods, color='k', linestyle='--')
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

Гистограмма получилась не очень гладкая, потому что мы обработали всего 100 точек данных. Однако ее форма все равно аппроксимируется к нормальному распределению. Выводим среднее этого распределения и сравниваем его со средним совокупности (листинг 6.22).

Листинг 6.22. Сравнение среднего нормального распределения и среднего совокупности

```
print(f"Actual population mean is approximately {population_mean:.2f}")
percent_diff_means = percent_diff(mean, population_mean)
print(f"There is a {percent_diff_means:.2f} % difference between means.")

Actual population mean is approximately 42.53
There is a 2.17 % difference between means.
```

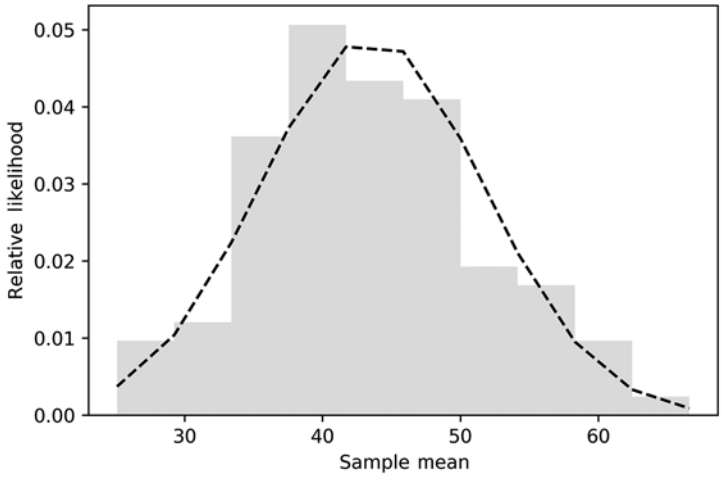


Рис. 6.5. Гистограмма на основе 100 возрастных выборок. На эту гистограмму наложено связанное с ней нормальное распределение. Среднее нормального распределения и его стандартное отклонение были получены из данных гистограммы

Полученное среднее возраста населения составило примерно 43. Фактическое же среднее по совокупности равно примерно 42,5. Между оценочным средним и фактическим получилась разница около 2 %, значит, наш результат хотя и неидеален, но довольно близок к фактическому среднему возрасту жителей города.

Теперь ненадолго переключим внимание на стандартное отклонение, вычисленное из нормального распределения. Возведение в квадрат стандартного отклонения дает дисперсию распределения. Согласно центральной предельной теореме, ее можно использовать для оценки дисперсии возраста горожан. Для этого нужно лишь умножить вычисленную дисперсию на размер выборки (листинг 6.23).

Листинг 6.23. Оценка дисперсии совокупности

```
normal_variance = std ** 2
estimated_variance = normal_variance * sample_size
```

Сравним оценочную дисперсию с дисперсией совокупности (листинг 6.24).

Листинг 6.24. Сравнение оценочной дисперсии с дисперсией совокупности

```
print(f"Estimated variance is approximately {estimated_variance:.2f}")
print(f"Actual population variance is approximately
     {population_variance:.2f}")
percent_diff_var = percent_diff(estimated_variance, population_variance)
print(f"There is a {percent_diff_var:.2f} percent difference between
        variances.")

Estimated variance is approximately 576.73
Actual population variance is approximately 584.33
There is a 1.30 percent difference between variances.
```

Разница между оценочной дисперсией и дисперсией совокупности составляет примерно 1,3 %. Таким образом, мы довольно точно оценили дисперсию возраста в городе, проанализировав всего 2 % его жителей. Эта оценка может не быть идеальной, но количество сэкономленного времени с лихвой оправдывает незначительную неточность.

До сих пор мы использовали центральную предельную теорему только для оценки среднего значения и стандартного отклонения генеральной совокупности. Однако возможности этой теоремы выходят далеко за пределы простой оценки параметров распределения. С ее помощью можно делать прогнозы и относительно людей.

6.3. СОСТАВЛЕНИЕ ПРОГНОЗОВ НА ОСНОВЕ СРЕДНЕГО И ДИСПЕРСИИ

Далее мы рассмотрим новый сценарий, в котором проанализируем класс школьников. Миссис Манн — прекрасный преподаватель пятого класса. Она в течение 25 лет с любовью обучала детей. В ее классе 20 учащихся. Получается, что за все эти годы в целом она обучила 500 человек.

ПРИМЕЧАНИЕ

Предполагается, что каждый год миссис Манн обучает ровно 20 учащихся. Но в реальной жизни размер класса, естественно, может из года в год меняться.

Ее ученики зачастую превосходят по успеваемости пятиклассников из других школ штата. Оценивается успеваемость в ходе ежегодных экзаменов, результаты которых — это числа от 0 до 100 баллов. Все полученные оценки можно узнать, запросив информацию в базе данных штата. Однако из-за ее неудачной структуры запрашиваемые результаты экзаменов не отражают год, в который они были получены.

Представим, что нам дали задание ответить на следующий вопрос: «Обучала ли за все время миссис Манн класс, который коллективно сдал экзамен на отлично?» Говоря конкретнее, обучала ли она когда-либо класс из 20 учащихся, средний показатель которых на экзамене оказался выше 89 %?

Для ответа на этот вопрос предположим, что мы запросили базу данных штата, из которой получили оценки всех прежних учеников миссис Манн. Конечно же, отсутствие временно́й информации не позволяет сгруппировать оценки по годам. В связи с этим мы не можем просто просканировать записи на предмет наличия среднего балла ежегодного экзамена выше 89 %. Однако можем вычислить среднее и дисперсию для всех 500 оценок. Предположим, что это среднее равно 84 при дисперсии, равной 25 (листинг 6.25). Эти значения мы назовем средним генеральной

совокупности и дисперсией генеральной совокупности, поскольку они охватывают всю совокупность учеников миссис Манн.

Листинг 6.25. Среднее и дисперсия совокупности экзаменационных баллов

```
population_mean = 84
population_variance = 25
```

Далее смоделируем ежегодные показатели экзаменов класса миссис Манн в виде коллекции из 20 оценок, случайно взятых из распределения со средним `population_mean` и дисперсией `population_variance`. Это упрощенная модель, в которой мы делаем несколько допущений.

- Успеваемость каждого ученика не зависит ни от какого другого ученика. В реальной жизни это допущение верно не всегда. Например, беспокойные ученики негативно влияют на показатели успеваемости других.
- Каждый год сложность экзаменов одинаковая. В реальности Министерство образования может корректировать стандарты ежегодных экзаменов.
- Местные экономические факторы не учитываются. В реальности экономические колебания влияют на школьный бюджет, а также на бытовые условия учеников. В результате эти внешние факторы могут сказаться и на итоговой успеваемости.

Принятые нами упрощения могут повлиять на точность прогноза. Тем не менее, учитывая ограниченность анализируемых данных, особого выбора у нас нет. Статистики зачастую вынуждены идти на подобные компромиссы, чтобы решить иначе неразрешимые задачи. И в большинстве случаев их упрощенные прогнозы вполне адекватно отражают реальное положение вещей.

Используя упрощенную модель, мы можем сделать случайную выборку из 20 оценок. Какова вероятность того, что среднее значение этих оценок окажется не менее 90? Эту вероятность можно легко вычислить с помощью центральной предельной теоремы. Согласно ей распределение вероятности средних оценок будет походить на нормальную кривую. Среднее этой нормальной кривой будет равняться `population_mean`, а дисперсия — `population_variance`, разделенной на размер выборки из 20 учеников. Взятие квадратного корня из этой дисперсии дает стандартное отклонение кривой, которое статистики называют *стандартной ошибкой среднего* (SEM). По определению SEM равняется стандартному отклонению совокупности, разделенному на квадратный корень из размера выборки. Далее мы вычислим параметры этой кривой и построим график нормальной кривой (листинг 6.26; рис. 6.6).

Площадь под полученной кривой при значениях выше 89 % стремится к нулю. Эта площадь также равна вероятности данного события. Следовательно, вероятность получения средней оценки не менее 90 % крайне мала. И все же для полной уверенности нужно вычислить фактическую вероятность. Для этого нам нужно каким-то образом точно измерить площадь под этим нормальным распределением.

Листинг 6.26. Построение кривой нормального распределения с помощью среднего и SEM

```
mean = population_mean
population_std = population_variance ** 0.5
sem = population_std / (20 ** 0.5)
grade_range = range(101)
normal_likelihoods = stats.norm.pdf(grade_range, mean, sem)
plt.plot(grade_range, normal_likelihoods)
plt.xlabel('Mean Grade of 20 Students ( %)')
plt.ylabel('Relative Likelihood')
plt.show()
```

Стандартное отклонение совокупности равно квадратному корню из дисперсии совокупности

SEM равна population_std, разделенному на размер выборки. В качестве альтернативы SEM можно вычислить (population_variance / 20) ** 0.5

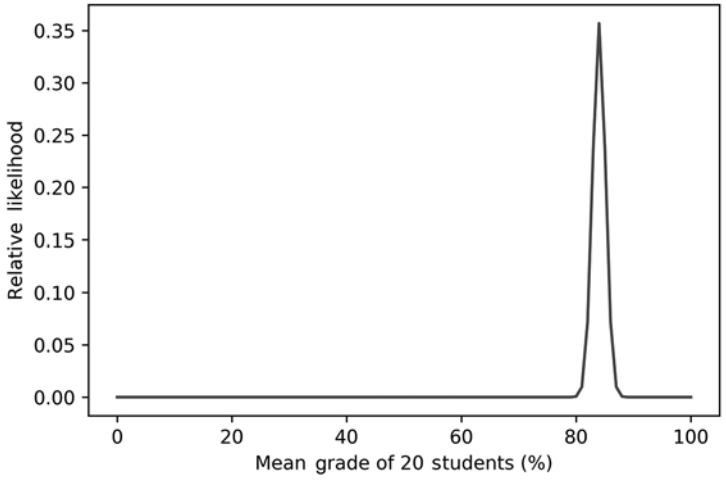


Рис. 6.6. Нормальное распределение, полученное на основе среднего значения совокупности и стандартной ошибки среднего. SEM равна стандартному отклонению, разделенному на квадратный корень из размера выборки. Площадь под полученной кривой можно использовать для вычисления вероятностей

6.3.1. Вычисление площади под нормальной кривой

В главе 3 мы вычисляли площади под гистограммами. Определить их оказалось несложно. Все гистограммы по определению состоят из небольших прямоугольных блоков. Мы можем просуммировать площади этих прямоугольников, составляющих заданный интервал, получив его площадь. К сожалению, наша гладкая кривая не разбивается на прямоугольники. Как же вычислить ее площадь? Простым решением будет разбить эту кривую на маленькие трапециевидные элементы. Эта древняя техника называется *методом трапеций*. Трапеция — это четырехугольник с двумя параллельными сторонами. Площадь трапеции равна сумме параллельных

сторон, умноженной на половину расстояния между ними. Сложение площадей нескольких последовательных трапеций аппроксимируется к площади включающего их интервала, как показано на рис. 6.7.

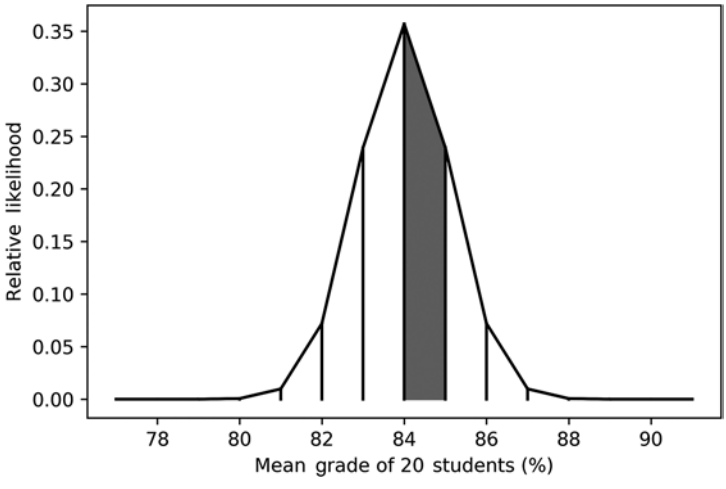


Рис. 6.7. Нормальное распределение, разбитое на трапеции. Нижний левый угол каждой трапеции находится в координате x, равной i. Параллельные стороны каждой трапеции определяются stats.norm.pdf(i) и stats.norm.pdf(i + 1). Находятся они друг от друга на расстоянии 1. Площадь трапеции в позиции 84 выделена темным. Она равна (stats.norm.pdf(84) + stats.norm.pdf(85)) / 2. Сложение площадей трапеций вдоль интервала аппроксимируется к общей площади этого интервала

Метод трапеций очень просто реализуется в нескольких строках кода. В качестве альтернативы можно задействовать метод NumPy `np.trapz`, который вычислит площадь переданного ему массива. Применим метод трапеций к нормальному распределению (листинг 6.27). Нам нужно проверить, как он в итоге аппроксимирует общую площадь, охватываемую `normal_likelihoods`. В идеале эта площадь должна быть около 1,0.

Листинг 6.27. Аппроксимация площади с помощью метода трапеций

```
# Площадь каждой трапеции равна сумме двух последовательных
# вероятностей, разделенной на 2. Расстояние координат x сторон трапеции
# друг от друга равно 1, поэтому оно в расчетах не учитывается
total_area = np.sum([normal_likelihoods[i: i + 2].sum() / 2
                     for i in range(normal_likelihoods.size - 1)])

assert total_area == np.trapz(normal_likelihoods)
print(f"Estimated area under the curve is {total_area}")

Estimated area under the curve is 1.0000000000384808
```

Заметьте, что NumPy выполняет метод трапеций математически более эффективно

Оценочная площадь получилась очень близкой к 1, но не равна этому значению в точности. В действительности она немного больше него. Если мы готовы смириться с этой мелкой неточностью, то метод трапеций можно счесть вполне приемлемым. В противном же случае нам потребуется способ вычисления точной площади нормального распределения. И обеспечивается эта точность библиотекой SciPy. Добиться математически точного решения можно с помощью метода `stats.norm.sf` (листинг 6.28). Он представляет *функцию выживаемости* нормальной кривой. Эта функция равна площади распределения в определенном интервале, который больше некоторого x. Иными словами, функция выживаемости — это способ нахождения точной площади, аппроксимируемой `np.trapz(normal_likelihoods[x:])`. В итоге можно ожидать, что `stats.norm.sf(0, mean, sem)` будет равняться `1.0`.

Листинг 6.28. Вычисление общей площади с помощью SciPy

```
assert stats.norm.sf(0, mean, sem) == 1.0
```

Теоретически нижнее значение x нормальной кривой уходит в отрицательную бесконечность. Следовательно, эта фактическая площадь микроскопически меньше 1,0. Однако разница эта настолько незначительна, что SciPy не может ее обнаружить. Так что для наших целей можно рассматривать эту точную площадь как равную 1,0

Аналогично ожидается, что `stats.norm.sf(mean, mean, sem)` равна `0.5`, поскольку среднее идеально разделяет нормальную кривую на две равные половины (рис. 6.8). Таким образом, интервал значений, выходящих за среднее, охватывает половины площади нормальной кривой. При этом мы ожидаем, что `np.trapz(normal_likelihoods[mean:])` будет приближаться к значению `0.5`, но не равняться ему. Проверим (листинг 6.29).

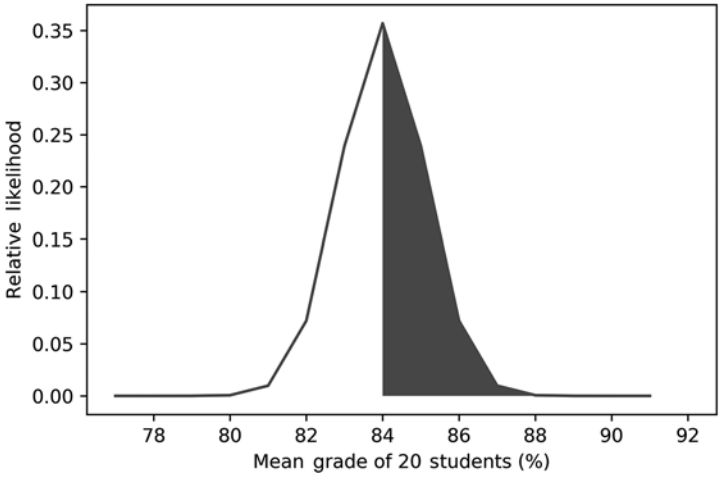


Рис. 6.8. Мы выделили площадь, обозначенную stats.norm.sf(mean, mean, sem). Она охватывает интервал значений, которые больше среднего либо равны ему, и при этом равна половине общей площади кривой. Ее точное значение — 0,5

Листинг 6.29. Подстановка среднего в функцию выживаемости

```
assert stats.norm.sf(mean, mean, sem) == 0.5
estimated_area = np.trapz(normal_likelihoods[mean:])
print(f"Estimated area beyond the mean is {estimated_area}")

Estimated area beyond the mean is 0.5000000000192404
```

ТИПИЧНЫЕ МЕТОДЫ ИЗМЕРЕНИЯ ПЛОЩАДИ КРИВОЙ

- `numpy.trapz(array)` — выполняет метод трапеций для оценки площади `array`. Различие между координатами *x* элементов массива установлено как 1.
- `numpy.trapz(array, dx=dx)` — выполняет метод трапеций для оценки площади `array`. Различие между координатами *x* элементов массива установлено как `dx`.
- `stats.norm.sf(x_value, mean, std)` — возвращает площадь под нормальной кривой, охватывая интервал, который больше `x_value` либо равен ему. Среднее значение и стандартное отклонение нормальной кривой установлены как `mean` и `std` соответственно.
- `stats.norm.sf(x_array, mean, std)` — возвращает массив площадей, которые получаются путем выполнения `stats.norm.sf(e, mean, std)` для каждого элемента `e` в `x_array`.

Теперь выполним `stats.norm.sf(90, mean, sem)`. Она вернет площадь для интервала значений, выходящих за 90 %. Эта площадь представляет вероятность того, что все 20 учеников сдадут экзамен на отлично (листинг 6.30).

Листинг 6.30. Вычисление вероятности коллективной сдачи экзамена на отлично

```
area = stats.norm.sf(90, mean, sem)
print(f"Probability of 20 students acing the exam is {area}")

Probability of 20 students acing the exam is 4.012555633463782e-08
```

Как и ожидалось, вероятность такого исхода очень мала.

6.3.2. Интерпретация вычисленной вероятности

Вероятность того, что все ученики получат на экзамене отличную оценку, равна примерно 1/25 000 000. Экзамен проводится всего раз в год, так что потребуется около 25 млн лет, пока случайный коллектив учеников не достигнет такого уровня успеваемости. При этом миссис Манн преподает всего 25 лет, то есть шанс выпустить за

эти годы класс, который сдал бы экзамен на суммарный балл выше 90 %, практически равен нулю. Так что можно смело сделать вывод, что такого класса у нее не было.

ПРИМЕЧАНИЕ

Фактические шансы можно вычислить, выполнив 1 – stats.binom.pmf(0, 25, stats.norm.sf(90, mean, sem)). Сможете догадаться почему?

Конечно же, мы могли ошибиться. Возможно, в одном классе внезапно собралась группа очень талантливых ребятишек. Это очень маловероятно, но все же возможно. Кроме того, наши простые вычисления не принимали в расчет изменения в сложности экзамена. Что, если он с каждым годом становится все легче? Это сделает неактуальным подход к анализу оценок с помощью случайной выборки.

Похоже, что наше итоговое заключение несовершенно. Мы сделали все, что могли, с учетом известных данных, но некоторая неуверенность все же сохраняется. Чтобы от нее избавиться, нам потребовались бы результаты проведения всех экзаменов. К сожалению, эти данные предоставлены не были. Довольно часто статистикам приходится делать логические заключения на основе ограниченной информации. Рассмотрим следующие два сценария.

- Кофейная ферма поставляет 500 т кофейных бобов в год, упаковывая их в мешки по 5 фунтов. В среднем 1 % бобов оказываются плесневелыми со стандартным отклонением 0,2 %. Управление по контролю качества пищевых продуктов и медикаментов (FDA) допускает наличие не более 3 % плесневелых бобов на мешок. Существует ли мешок, нарушающий эти требования?

  Здесь можно применить центральную предельную теорему, если предположить, что рост плесени не зависит от времени. Однако плесень может расти быстрее во время влажных летних месяцев. К сожалению, никаких данных об этом у нас нет.

- В приморском городке строят волнолом для защиты от цунами. Согласно историческим данным, средняя высота цунами составляет 23 фута со стандартным отклонением 4 фута. Запланировано, что высота возводимого волнолома будет 33 фута. Окажется ли ее достаточно для защиты города?

  Есть соблазн предположить, что средняя высота цунами из года в год меняться не будет. Однако ряд исследований показывает, что изменение климата вызывает подъем уровня моря. Климатические перемены могут привести к появлению более мощных волн в будущем. К сожалению, научные данные недостаточно проясняют ситуацию для того, чтобы сделать уверенные выводы относительно этого.

В обоих случаях необходимо принимать важные решения, опираясь на статистические приемы. Они зависят от определенных допущений, которые могут оказаться неверными. В связи с этим необходимо быть особо внимательными, делая выводы на основе ограниченной информации. В следующей главе мы продолжим изучать как риски, так и преимущества принятия решений на основе неполной информации.

РЕЗЮМЕ

- Среднее арифметическое и стандартное отклонение нормального распределения определяются по положению его пика. Среднее равняется координате x этого пика, а стандартное отклонение — обратной координате y, умноженной на $(2\pi)^{1/2}$.
- Функция плотности распределения вероятностей сопоставляет входные значения с плавающей запятой с весами их вероятности. Вычисление площади под этой кривой дает вероятность.
- Повторное вычисление среднего из практически любого распределения ведет к формированию нормальной кривой. Среднее этой нормальной кривой аппроксимируется к среднему лежащего в ее основе распределения. Кроме того, дисперсия нормальной кривой, умноженная на размер выборки, аналогичным образом аппроксимируется к дисперсии внутреннего распределения.
- *Стандартная ошибка среднего* равна стандартному отклонению генеральной совокупности, разделенному на квадратный корень из размера выборки. Следовательно, деление дисперсии совокупности на размер выборки с последующим взятием квадратного корня также генерирует SEM. В тандеме со средним совокупности SEM позволяет нам вычислять вероятность получения определенных комбинаций выборки.
- *Метод трапеций* дает возможность оценить площадь под кривой, разбив ее на трапециевидные элементы и суммировав площади полученных трапеций.
- *Функция выживаемости* измеряет площадь распределения для интервала, который больше некоторого x.
- Необходимо внимательно анализировать принимаемые допущения при составлении прогнозов на основе ограниченных данных.

7 Проверка статистических гипотез

В этой главе

- ✓ Сравнение средних выборки со средними генеральной совокупности.
- ✓ Сравнение средних двух разных выборок.
- ✓ Что такое статистическая значимость?
- ✓ Типичные статистические ошибки и способы их избежать.

Многие обычные люди ежедневно вынуждены принимать сложные решения. В особенности это касается присяжных в судебной системе США, ведь от их решения зависит судьба подсудимого. Они рассматривают доказательства, после чего склоняются к одной из двух противоположных гипотез:

- подзащитный невиновен;
- подзащитный виновен.

Эти две гипотезы имеют разный вес, так как подзащитный считается невиновным, пока не доказано обратное. Таким образом, присяжные исходят из того, что верна гипотеза о невиновности. В этом случае они могут лишь отвергнуть ее, если доказательства обвинителя окажутся достаточно убедительными. Но все же доказательства редко бывают на 100 % полноценными, некоторое сомнение в виновности подзащитного остается, и требуется принимать его во внимание. Присяжные должны придерживаться гипотезы о невиновности, если относительно вины

подзащитного существует разумное сомнение. Отвергнуть эту гипотезу они могут, только если подзащитный окажется виновен вне всяческих разумных сомнений.

Разумное сомнение — это абстрактное понятие, которому трудно дать точное определение. Тем не менее в ряде реальных сценариев можно провести различие между разумным и неразумным сомнением. Рассмотрим следующие два судебных примера.

- Подзащитного с преступлением непосредственно связывают результаты анализа ДНК. Существует один шанс из миллиарда, что исследованная ДНК принадлежит не подзащитному.
- Подзащитного с преступлением непосредственно связывают результаты анализа группы крови. Существует один шанс из 15, что исследованная кровь принадлежит не подзащитному.

В первом сценарии присяжные не могут быть на 100 % уверены в виновности подзащитного. Существует один шанс из миллиарда, что перед ними невиновный человек. Но подобный вариант очень маловероятен. Будет неразумным предположить, что дело обстоит именно так, поэтому присяжные могут отвергнуть гипотезу о невиновности.

А вот во втором сценарии сомнение будет уже куда более ощутимым, так как группа крови подзащитного совпадает с группой крови каждого 15-го человека. Вполне разумно предположить, что на месте преступления мог присутствовать и кто-то еще. И хотя присяжные могут сомневаться в невиновности подзащитного, у них также будут довольно веские основания усомниться в его вине. Таким образом, присяжные не могут отвергнуть гипотезу о невиновности, пока не получат дополнительные доказательства вины.

В этих двух случаях присяжные проверяют *статистическую гипотезу*. Подобные проверки позволяют статистикам выбирать между двумя противоположными гипотезами, которые возникают из неоднозначных данных. Одна из гипотез принимается либо отвергается на основе измеримого уровня сомнения. В данной главе мы изучим несколько известных техник проверки статистических гипотез, начав с простого теста, измеряющего, насколько значительно отклоняется среднее выборки от среднего генеральной совокупности.

7.1. АНАЛИЗ РАСХОЖДЕНИЯ МЕЖДУ СРЕДНИМ ВЫБОРКИ И СРЕДНИМ СОВОКУПНОСТИ

В главе 6 мы с помощью статистики проанализировали группу пятиклассников. Теперь давайте представим сценарий, в котором нужно проанализировать все аналогичные классы в Северной Дакоте. Одним весенним днем все пятиклассники штата пошли на один и тот же экзамен. Полученные на нем оценки были внесены в базу данных штата, после чего для них всех были вычислены среднее арифметическое

и дисперсия. Согласно записям, среднее получилось 80, а дисперсия — 100. Сохраним эти значения для последующего использования (листинг 7.1).

Листинг 7.1. Среднее и дисперсия генеральной совокупности оценок в Северной Дакоте

```
population_mean = 80
population_variance = 100
```

Теперь предположим, что мы отправились в Южную Дакоту и встретили класс, который получил на экзамене средний балл 84 %. Этот класс из 18 учеников опередил своим результатом общую совокупность учеников из Северной Дакоты на 4 %. Неужели в Южной Дакоте пятиклассники более образованны, чем их ровесники из Северной Дакоты? Если это так, то Северная Дакота должна внедрить методы преподавания своих коллег из Южной Дакоты. Корректировка учебного курса грозит серьезными затратами, но отдача в виде более образованных учеников того стоит. Естественно, также есть вероятность, что наблюдаемая разница в результатах экзаменов объясняется просто статистическими колебаниями. Как же это выяснить? Здесь нам поможет тестирование гипотез.

Перед нами две противоположные вероятности. Во-первых, возможно, что общая совокупность учеников в двух штатах одинакова. Иными словами, типичный класс в Южной Дакоте не отличается от типичного класса в Северной Дакоте. В таком случае среднее значение и дисперсия для совокупности из Южной Дакоты будут неотличимы от их аналогов для Северной. Статистики называют эту гипотетическую равнозначность параметров *нулевой гипотезой*. Если нулевая гипотеза оказывается верна, тогда более успешный класс из Южной Дакоты является просто исключением и не представляет фактическое среднее всей совокупности.

При этом также может оказаться, что высокие показатели класса все-таки отражают параметры общей совокупности учащихся Южной Дакоты. Тогда среднее и дисперсия для этого штата будут отличаться от параметров общей совокупности из Северной Дакоты. Статистики называют это *альтернативной гипотезой*. Если альтернативная гипотеза окажется верна, мы обновим в Северной Дакоте учебную программу для пятого класса. Однако эта гипотеза оказывается верна только в случае ложности нулевой гипотезы и наоборот. Следовательно, чтобы оправдать изменение учебной программы, необходимо сначала показать, что нулевая гипотеза вряд ли истинна. Измерить правдоподобность этого можно с помощью центральной предельной теоремы.

Давайте временно предположим, что нулевая гипотеза верна и обе части Дакоты имеют одинаковые показатели среднего и дисперсии. Тогда можно будет смоделировать класс из 18 учащихся как случайную выборку, взятую из нормального распределения. Среднее этого распределения будет равняться `population_mean`, а его стандартное отклонение — стандартной ошибке среднего, определяемой как `(population_variance / 18) ** 0.5` (листинг 7.2).

Листинг 7.2. Параметры нормальной кривой, если нулевая гипотеза верна

```
mean = population_mean
sem = (population_variance / 18) ** 0.5
```

Если нулевая гипотеза верна, вероятность наблюдения среднего экзаменационного балла 84 % будет равна `stats.norm.sf(84 mean, sem)`. Проверим эту вероятность (листинг 7.3).

Листинг 7.3. Нахождение вероятности получения высокой средней экзаменационной оценки

```
prob_high_grade = stats.norm.sf(84, mean, sem)
print(f"Probability of an average grade >= 84 is {prob_high_grade}")

Probability of an average grade >= 84 is 0.044843010885182284
```

В условиях нулевой гипотезы средний экзаменационный балл случайного класса из Южной Дакоты будет равен 84 % с вероятностью 0,044. Это низкая вероятность, а значит, отличие 4 % от среднего совокупности является экстремальным. Но действительно ли оно экстремально? В главе 1 мы ставили аналогичный вопрос, когда оценивали вероятность получения восьми орлов при десяти подбрасываниях монеты. Тогда в ходе анализа мы суммировали вероятность существенного преобладания исходов с орлом с вероятностью их существенного дефицита. Иными словами, складывали вероятность получения восьми или более орлов с вероятностью получения двух или менее. Здесь перед нами аналогичная дилемма. Анализа повышенных показателей на экзамене недостаточно для оценки их экстремальности. Необходимо также учесть вероятность в той же степени экстремально низкого показателя. Следовательно, нам нужно вычислить вероятность получения среднего выборки, которое на 4 % или более ниже среднего по генеральной совокупности, которое равно 80 %.

То есть теперь мы вычислим вероятность получения на экзамене среднего балла, который меньше или равен 76 %. Это можно сделать с помощью метода SciPy `stats.norm.cdf`, который вычисляет *кумулятивную функцию распределения* (или просто функцию распределения) нормальной кривой. Функция распределения является прямой противоположностью функции выживаемости, что отражено на рис. 7.1. Применение `stats.notm.cdf` к x возвращает площадь под нормальной кривой, которая простирается от отрицательной бесконечности до x.

Теперь используем `stats.norm.cdf` для нахождения вероятности крайне редкого среднего экзаменационного балла (листинг 7.4).

Листинг 7.4. Нахождение вероятности получения низкого экзаменационного балла

```
prob_low_grade = stats.norm.cdf(76, mean, sem)
print(f"Probability of an average grade <= 76 is {prob_low_grade}")

Probability of an average grade <= 76 is 0.044843010885182284
```

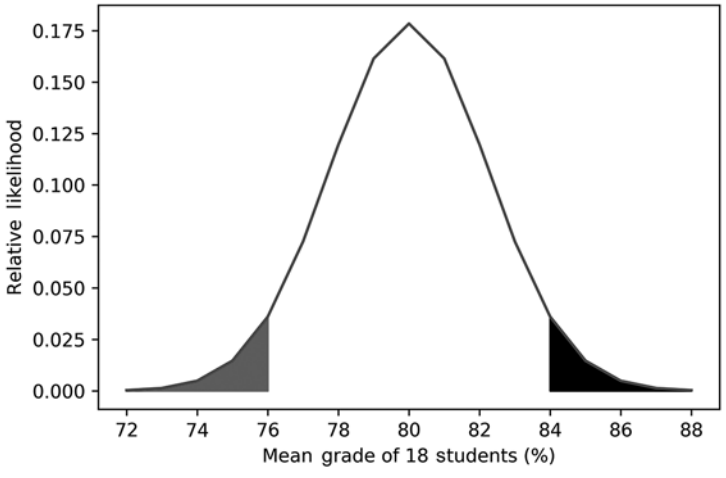


Рис. 7.1. Под нормальной кривой выделены две области. Левая из них охватывает все значения x, которые меньше или равны 76 %. Ее площадь можно вычислить с помощью функции распределения. Для выполнения этой функции нужно вызвать stats.norm.cdf(76, mean, sem). При этом правая область охватывает все значения x, которые больше либо равны 84 %. Ее площадь можно вычислить с помощью функции выживаемости, для выполнения которой нужно вызвать stats.norm.sf(84, mean, sem)

Оказывается, `prob_low_grade` равна `prob_high_grade`. Это равенство обусловливается симметричной формой нормальной кривой. Функция распределения и функция выживаемости являются зеркальными представлениями друг друга, которые отражаются в средних значениях (листинг 7.5). Таким образом, `stats.norm.sf(mean + x, mean, sem)` для любого входного x всегда равняется `stats.norm.cdf(mean – x, mean, sem)`. Далее мы визуализируем обе функции, чтобы подтвердить их зеркальность относительно вертикально построенного среднего (рис. 7.2).

Листинг 7.5. Сравнение функции выживаемости и функции распределения

```
for x in range(-100, 100):
    sf_value = stats.norm.sf(mean + x, mean, sem)
    assert sf_value == stats.norm.cdf(mean – x, mean, sem)

plt.axvline(mean, color='k', label='Mean', linestyle=':')
x_values = range(60, 101)
plt.plot(x_values, stats.norm.cdf(x_values, mean, sem),
         label='Cumulative Distribution')
plt.plot(x_values, stats.norm.sf(x_values, mean, sem),
         label='Survival Function', linestyle='--', color='r')
plt.xlabel('Sample Mean')
plt.ylabel('Probability')
plt.legend()
plt.show()
```

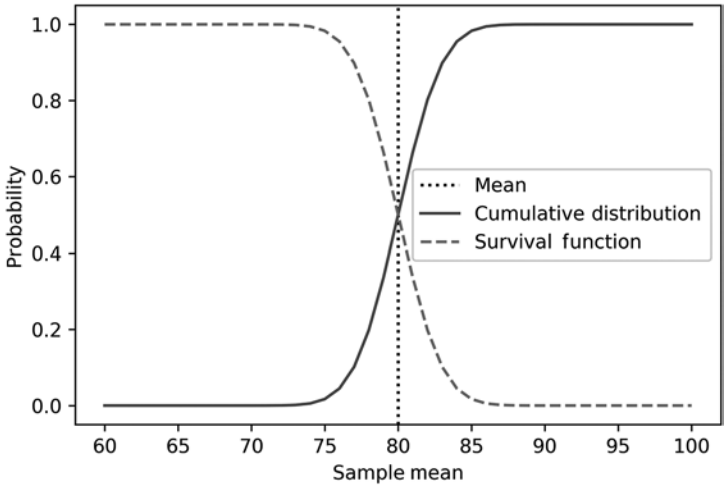


Рис. 7.2. Функция распределения нормальной кривой, построенная наряду с функцией выживаемости, с которой они представляют зеркальное отражение друг друга. Они отражаются вдоль среднего нормальной кривой, представленного в виде вертикальной линии

Вот теперь можно складывать `prob_high_grade` и `prob_low_grade`. Ввиду симметрии их сумма окажется равной `2 * prob_high_grade`. В принципиальном смысле она представляет вероятность наблюдения экстремального отклонения от среднего совокупности, когда нулевая гипотеза верна. Статистики называют эту вероятность, управляемую нулевой гипотезой, *p*-значением. Выведем *p*-значение на основе наших данных (листинг 7.6).

Листинг 7.6. Вычисление p-значения, управляемого нулевой гипотезой

```
p_value = prob_low_grade + prob_high_grade
assert p_value == 2 * prob_high_grade
print(f"The p-value is {p_value}")

The p-value is 0.08968602177036457
```

В случае верности нулевой гипотезы существует примерно 9%-ная вероятность случайным образом встретить на экзамене экстремальный средний балл. В связи с этим вполне вероятно, что нулевая гипотеза верна и тестируемое нами экстремальное среднее оказывается просто случайным колебанием. Мы еще не подтвердили это однозначно, но наши вычисления вызывают серьезные сомнения относительно необходимости изменения в Северной Дакоте учебной программы пятого класса. А что, если среднее класса из Южной Дакоты было бы равно 85 %, а не 84 %? Проверим, насколько столь незначительное изменение повлияло бы на *p*-значение (листинг 7.7).

Листинг 7.7. Вычисление p-значения для скорректированного среднего выборки

```
def compute_p_value(observed_mean, population_mean, sem):
    mean_diff = abs(population_mean – observed_mean)
    prob_high = stats.norm.sf(population_mean + mean_diff, population_mean, sem)
    return 2 * prob_high

new_p_value = compute_p_value(85, mean, sem)
print(f"The updated p-value is {new_p_value}")

The updated p-value is 0.03389485352468927
```

Едва заметное увеличение среднего балла вызвало трехкратное уменьшение *p*-значения. Теперь есть всего лишь 3,3 % вероятности в условиях нулевой гипотезы получить на экзамене экстремальный средний балл, равный 85 % или выше. Это очень низкая вероятность, поэтому у нас может возникнуть желание отвергнуть нулевую гипотезу. Нужно ли тогда принять альтернативную гипотезу и вложить усилия и средства в перестройку учебной программы школ Северной Дакоты?

Это непростой вопрос. Обычно статистики склонны отвергать нулевую гипотезу, если *p*-значение оказывается меньше либо равно 0,05. Этот порог называется *уровнем значимости*, и *p*-значения ниже него считаются *статистически значимыми*. Однако 0,05 — это всего лишь произвольный предел, предназначенный для эвристического раскрытия интересных данных, а не для принятия критических решений. Впервые этот порог был введен в 1935 году статистиком Рональдом Фишером (Ronald Fisher). Позднее он сказал, что уровень значимости должен не оставаться статичным, а корректироваться вручную на основе природы анализируемого феномена. Жаль, что это заявление было сделано поздновато и предел 0,05 уже был признан в качестве стандартной меры значимости. Сегодня большинство статистиков соглашаются с тем, что *p*-значение ниже 0,05 подразумевает наличие в данных интересного сигнала, значит, *p*-значения 0,033 будет достаточно, чтобы временно отвергнуть нулевую гипотезу и добиться публикации полученных кем-то данных в журнале. К сожалению, порог 0,05 на самом деле не строится на законах математики и статистки — это ситуативное значение, выбранное академическим сообществом в качестве требования для исследовательских публикаций. Как следствие, многие научные журналы переполнены *ошибками первого рода*, которые подразумевают, что нулевая гипотеза отвергнута ошибочно. Они возникают, когда флуктуации случайных данных интерпретируются как истинные отклонения от среднего генеральной совокупности. Научные статьи, содержащие ошибки первого рода, утверждают, что существуют различия между средними значениями там, где этих различий нет.

Как же сократить количество подобных ошибок? Некоторые ученые считают порог 0,05 неоправданно высоким и предлагают отвергать нулевую гипотезу лишь тогда, когда *p*-значение намного ниже. Но пока нет единого мнения о том, действительно ли использование более низкого *p*-значения — подходящее решение, поскольку это приведет к увеличению количества *ошибок второго рода*, подразумевающих,

что ошибочно отвергнута альтернативная гипотеза. Когда ученые допускают такую ошибку, они упускают из виду реальное открытие.

Выбрать оптимальный уровень значимости сложно. Тем не менее мы временно установим его на 0,001. Каков будет минимальный средний балл, попадающий под этот порог? В дальнейшем мы это выясним. Для этого нужно будет перебрать все средние баллы выше 80 %, по ходу вычисляя их *p*-значения (листинг 7.8). Остановимся мы, когда встретим *p*-значение, которое окажется меньше либо равным 0,001.

Листинг 7.8. Поиск результата, дающего строгое p-значение

```
for grade in range(80, 100):
    p_value = compute_p_value(grade, mean, sem)
    if p_value < 0.001:
        break

print(f"An average grade of {grade} leads to a p-value of {p_value}")

An average grade of 88 leads to a p-value of 0.0006885138966450773
```

С учетом нового порога для отклонения нулевой гипотезы потребуется средний балл не менее 88 %. Таким образом, средний показатель 87 % не будет считаться статистически значимым, несмотря на то что он заметно выше среднего генеральной совокупности. Из-за снижения порога мы неизбежно рисковали бы допустить ошибку второго рода. По этой причине в данной книге мы сохраняем общепринятую границу для *p*-значения, то есть 0,05. Но будем продвигаться с осторожностью, чтобы избежать ошибочного отклонения нулевой гипотезы. В частности, постараемся минимизировать количество наиболее распространенных случаев ошибок первого рода, и темой следующего раздела станет «выуживание данных».

7.2. ВЫУЖИВАНИЕ ДАННЫХ: ПРИХОД К СЛОЖНЫМ ВЫВОДАМ ИЗ-ЗА РЕСЭМПЛИНГА

Иногда изучающие статистику студенты используют *p*-значение неверно. Представим простой сценарий. Два соседа по комнате высыпают мешок конфет пяти разных цветов. Больше всего среди них оказывается синих. Первый сосед предполагает, что синих конфет в любом аналогичном мешке всегда больше. Второй с ним не соглашается: он вычисляет *p*-значение на основе нулевой гипотезы, предполагающей, что все цвета встречаются с равной вероятностью. Это *p*-значение больше 0,05. Однако первый не сдается и открывает еще один мешок конфет, для содержимого которого также вычисляется *p*-значение. На этот раз *p*-значение оказывается равным 0,05. Первый сосед заявляет о победе, утверждая, что ввиду низкого *p*-значения нулевая гипотеза наверняка является ложной. Но при этом он все же ошибается.

Первый сосед фундаментально исказил смысл *p*-значения. Он ошибочно предположил, что оно представляет вероятность истинности нулевой гипотезы. В действительности же *p*-значение представляет вероятность получения отклонений, если нулевая гипотеза окажется истинной. Разница между этими определениями довольно тонка, но очень важна. В первом подразумевается, что нулевую гипотезу можно считать несостоятельной, если *p*-значение мало. Второе же определение гарантирует, что при многократном повторении подсчета конфет мы в конечном итоге получим низкое *p*-значение, даже когда нулевая гипотеза будет верна. Более того, частота получения низкого *p*-значения будет равняться самому *p*-значению. Следовательно, если открыть 100 мешков конфет, то можно ожидать получения *p*-значения, равного 0,05, примерно в пяти случаях. При многократном повторении случайных измерений мы в конечном итоге получим статистически значимый результат даже при отсутствии реальной статистической значимости.

Многократное выполнение одного и того же эксперимента повышает риск допустить ошибку первого рода. Давайте рассмотрим это утверждение в контексте нашего анализа экзамена в пятом классе. Предположим, что результаты единого экзамена в Северной Дакоте не отличаются от его результатов в других 49 штатах. Говоря точнее, мы предположим, что национальное среднее и дисперсия равны `population_mean` и `population_variance` результатов экзаменов в Северной Дакоте. Таким образом, нулевая гипотеза оказывается истинной для всех штатов в США.

Кроме того, допустим, что нам еще неизвестно, что нулевая гипотеза всегда истинна. Знаем мы лишь среднее и дисперсию совокупности результатов Северной Дакоты. В связи с этим организуем путешествие в поиске штата, где распределение баллов отличается от распределения Северной Дакоты. К сожалению, поиск обречен на провал, потому что таких штатов нет.

Первой на нашем пути оказывается Монтана. В этом штате выбираем случайный класс из 18 учащихся и вычисляем их средний экзаменационный балл. Поскольку нулевая гипотеза скрыто является истинной, можно симулировать значение этого среднего балла, сделав выборку на основании нормального распределения, определяемого `mean` и `sem`. Далее мы симулируем показатели класса вызовом `np.random.normal(mean, sem)` (листинг 7.9). Вызов этого метода выполняет выборку из нормального распределения, определяемого входными переменными.

Листинг 7.9. Случайная выборка результатов экзаменов в Монтане

```
np.random.seed(0)
random_average_grade = np.random.normal(mean, sem)
print(f"Average grade equals {random_average_grade:.2f}")

Average grade equals 84.16
```

Средний экзаменационный балл класса равен примерно 84,16. Можно определить, представляет ли это среднее статистическую значимость, проверив, не является ли его *p*-значение равным или меньше 0,05 (листинг 7.10).

Листинг 7.10. Проверка значимости показателей экзаменов в Монтане

```
if compute_p_value(random_average_grade, mean, sem) <= 0.05:
    print("The observed result is statistically significant")
else:
    print("The observed result is not statistically significant")

The observed result is not statistically significant
```

Этот средний балл не является статистически значимым. Мы продолжаем путешествие по остальным 48 штатам и в каждом посещаем по одному классу из 18 учащихся, вычисляя для них средний балл и *p*-значение. Как только найдем статистически значимое *p*-значение, путешествие завершится.

Код листинга 7.11 — симуляция этой поездки. Он перебирает 48 оставшихся штатов, случайно выбирая средний балл для каждого. Как только будет обнаружен статистически значимый показатель, итерация прервется.

Листинг 7.11. Случайный поиск значимых результатов штата

```
np.random.seed(0)
for i in range(1, 49):
    print(f"We visited state {i + 1}")
    random_average_grade = np.random.normal(mean, sem)
    p_value = compute_p_value(random_average_grade, mean, sem)
    if p_value <= 0.05:
        print("We found a statistically significant result.")
        print(f"The average grade was {random_average_grade:.2f}")
        print(f"The p-value was {p_value}")
        break

if i == 48:
    print("We visited every state and found no significant results.")

We visited state 2
We visited state 3
We visited state 4
We visited state 5
We found a statistically significant result.
The average grade was 85.28
The p-value was 0.025032993883401307
```

Пятый штат дает статистически значимый результат! Средний балл класса из этого штата составляет 85,28, а связанное с ним *p*-значение 0,025 попадает под порог 0,05. Оказывается, можно отвергнуть нулевую гипотезу. Однако это вычисление ошибочно, поскольку нулевая гипотеза истинна. Что же пошло не так? Как уже говорилось, частота получения низкого *p*-значения будет равна самому *p*-значению. Следовательно, мы ожидаем встретить его величину 0,025 приблизительно в 2,5 % случаев, даже если нулевая гипотеза верна. Поскольку мы путешествуем по 49 штатам, а 2,5 % из 49 равно 1,225, то стоит ожидать, что нам попадется примерно один штат, где случайное *p*-значение окажется равно приблизительно 0,025.

Наш квест по нахождению статистически значимого результата был обречен с самого начала из-за неверного применения статистики. Мы совершили величайший статистический грех, называемый выуживанием данных, или *p*-хакингом. При выуживании данных эксперименты многократно повторяются, пока не встречается статистически значимый результат. Данный результат демонстрируется другим, а остальные провальные эксперименты отбрасываются. В научных публикациях выуживание данных оказывается наиболее частой причиной ошибок первого рода. Печально, но иногда исследователи формулируют гипотезу и повторяют эксперимент до тех пор, пока конкретно ложная гипотеза не будет оценена как истинная. К примеру, исследователь может высказать предположение, что определенные конфеты вызывают рак у мышей. Он начнет кормить группу зверьков конфетами определенного производителя, но никакой связи с раком не обнаружит. Тогда он сменит производителя конфет и повторит эксперимент и т. д. Спустя годы ученый все же отыщет конфеты, якобы связанные с раком. Естественно, фактический результат такого эксперимента можно считать практически мошенническим. Никакой реальной статистической связи между раком и конфетами не существует — исследователь просто выполнил эксперимент очень много раз, пока случайным образом не было получено низкое *p*-значение.

Избежать выуживания данных несложно — нужно просто заранее определить конечное число предполагаемых экспериментов. После этого установить уровень значимости равным значению 0,05, разделенному на запланированное число. Этот простой прием называется *поправкой по методу Бонферрони*. Далее мы повторим анализ показателей экзаменов в США, используя эту поправку. Анализ требует посещения 49 штатов для оценки 49 классов, следовательно, уровень значимости нужно установить как 0,05/49 (листинг 7.12).

Листинг 7.12. Применение поправки Бонферрони для корректировки значимости

```
num_planned_experiments = 49
significance_level = .05 / num_planned_experiments
```

Мы повторно выполняем анализ, который завершится в случае получения *p*-значения, меньшего либо равного `significance_level` (листинг 7.13).

Листинг 7.13. Повторное выполнение анализа с использованием скорректированного уровня значимости

```
np.random.seed(0)
for i in range(49):
    random_average_grade = np.random.normal(mean, sem)
    p_value = compute_p_value(random_average_grade, mean, sem)
    if p_value <= significance_level:
        print("We found a statistically significant result.")
        print(f"The average grade was {random_average_grade:.2f}")
        print(f"The p-value was {p_value}")
        break
```

```
if i == 48:
    print("We visited every state and found no significant results.")

We visited every state and found no significant results.
```

Мы посетили 49 штатов и не нашли никаких статистически значимых отклонений от среднего и дисперсии показателей Северной Дакоты. В данном случае поправка Бонферрони помогла нам избежать ошибки первого рода.

В качестве заключительного предостережения добавлю, что эта поправка работает, только если разделить 0,05 на количество запланированных экспериментов. То есть она окажется неэффективной, если делить на количество проведенных экспериментов. К примеру, если мы планируем провести 1000 испытаний, но *p*-значение первого из них равно 0,025, то нам не нужно изменять уровень значимости на 0,05/1. Аналогично, если *p*-значение второго завершенного испытания окажется 0,025, нужно сохранить уровень значимости равным 0,05/1000, а не исправлять его на 0,05/2. В противном случае мы рискуем получить заключение, ошибочно смещенное в сторону первых результатов испытаний. Все испытания должны рассматриваться как равные, чтобы получить корректное заключение.

Поправка Бонферрони — полезный прием для более точного тестирования гипотез. Ее можно применять ко всевозможным видам тестов статистических гипотез, выходя за рамки простых проверок, в которых задействуется как среднее, так и дисперсия совокупности. И это хорошо, потому что статистические тесты имеют различную степень сложности. В следующем разделе мы изучим более сложный тест, который не зависит от того, известна ли нам дисперсия совокупности.

7.3. БУТСТРЭППИНГ С ВОСПОЛНЕНИЕМ: ТЕСТИРОВАНИЕ ГИПОТЕЗ ПРИ НЕИЗВЕСТНОЙ ДИСПЕРСИИ СОВОКУПНОСТИ

Можно без проблем вычислить *p*-значение, когда известны среднее и дисперсия совокупности. К сожалению, во многих реальных ситуациях дисперсия совокупности оказывается неизвестной. Рассмотрим следующий случай с очень большим аквариумом. В нем плавают 20 тропических рыб длиной от 2 до 120 см. Средняя длина рыбы составляет 27 см. Мы представим их длины с помощью массива `fish_lengths` (листинг 7.14).

Листинг 7.14. Определение длин рыб в аквариуме

```
fish_lengths = np.array([46.7, 17.1, 2.0, 19.2, 7.9, 15.0, 43.4,
                         8.8, 47.8, 19.5, 2.9, 53.0, 23.5, 118.5,
                         3.8, 2.9, 53.9, 23.9, 2.0, 28.2])
assert fish_lengths.mean() == 27
```

Точно ли наш аквариум отражает распределение длин реальных тропических рыб? Нужно это выяснить. Заслуживающий доверия источник сообщает, что средняя длина диких тропических рыб по их генеральной совокупности равна 37 см. Получается, что между средним нашей выборки и средним совокупности есть разница 10 см. Чисто по ощущениям это различие кажется значительным, но в тщательном статистическом анализе нет места ощущениям. Для того чтобы сделать обоснованный вывод, нам необходимо определить, является ли такая разница в среднем размере статистически значимой.

До сих пор мы измеряли статистическую значимость с помощью функции `compute_p_value`. Но эту функцию нельзя применить к данным рыб, так как дисперсия совокупности нам неизвестна. Без этой информации не получится вычислить SEM, которая является необходимой переменной для `compute_p_value`. Как же выяснить стандартную ошибку среднего, когда дисперсия совокупности неизвестна?

На первый взгляд кажется, что вычислить SEM не получится. Можно наивно использовать в качестве приблизительного значения дисперсии совокупности дисперсию нашей выборки, выполнив `fish_lengths.var()`. К сожалению, небольшие выборки склонны к случайным колебаниям дисперсии, поэтому любое подобное приблизительное значение окажется весьма ненадежным. Получается, мы застряли. Перед нами неразрешимая задача, и мы вынуждены опираться на, как может показаться, невероятное решение — *бутстрэппинг с восполнением*. В основе термина «бутстрэппинг» лежит смысл «вытащить самого себя за волосы из болота». Естественно, подобное действие невозможно. В технике бутстрэппинга с восполнением мы пытаемся сделать нечто столь же невозможное, вычисляя *p*-значение напрямую из имеющихся ограниченных данных. И несмотря на кажущуюся бредовость, наша затея завершится успехом.

Этот процесс мы начнем с извлечения из аквариума случайной рыбы, длину которой измерим, чтобы использовать результат в дальнейшем (листинг 7.15).

Листинг 7.15. Извлечение из аквариума случайной рыбы

```
np.random.seed(0)
random_fish_length = np.random.choice(fish_lengths, size=1)[0]
sampled_fish_lengths = [random_fish_length]
```

Теперь вернем извлеченную рыбу обратно в аквариум. Именно этот шаг и обусловливает название «бутстрэппинг с восполнением». После возвращения первой рыбы мы так же случайно достаем еще одну. Существует 1 шанс из 20, что при этом нам попадется та же рыба, что и прежде, — это абсолютно приемлемо. Далее мы записываем длину второй извлеченной рыбы и возвращаем ее в аквариум. Эта процедура повторяется еще 18 раз, пока у нас не получится 20 длин случайно извлеченных рыб (листинг 7.16).

Листинг 7.16. Сэмплинг 20 случайных рыб с повторением

```
np.random.seed(0)
for _ in range(20):
    random_fish_length = np.random.choice(fish_lengths, size=1)[0]
    sampled_fish_lengths.append(random_fish_length)
```

Список `sampled_fish_lengths` содержит 20 измерений, все полученные из 20-элементного массива `fish_lengths`. Тем не менее элементы `fish_lengths` и `sampled_fish_lengths` не идентичны. Ввиду случайного сэмплинга средние значения массива и списка наверняка будут различаться (листинг 7.17).

Листинг 7.17. Сравнение среднего выборки со средним по аквариуму

```
sample_mean = np.mean(sampled_fish_lengths)
print(f"Mean of sampled fish lengths is {sample_mean:.2f} cm")

Mean of sampled fish lengths is 26.03 cm
```

Среднее значение длин рыб из выборки — 26,03 см. Оно отклоняется от изначального среднего на 0,97 см. Значит, сэмплинг с восполнением внес в наблюдения некоторую вариативность. Если произвести еще 20 аналогичных измерений по аквариуму, то можно ожидать, что полученное среднее также будет отклоняться от 27 см. Убедимся в этом, повторив процедуру сэмплинга с помощью всего одной строки кода — `np.random.choice(fish_lengths, size=20, replace=True)`. Установка параметра `replace` на `True` обеспечивает выполнение сэмплинга с восполнением из массива `fish_lengths` (листинг 7.18).

Листинг 7.18. Сэмплинг с восполнением при помощи NumPy

```
np.random.seed(0)
new_sampled_fish_lengths = np.random.choice(fish_lengths, size=20,
                                            replace=True)
new_sample_mean = new_sampled_fish_lengths.mean()
print(f"Mean of the new sampled fish lengths is {new_sample_mean:.2f} cm")

Mean of the new sampled fish lengths is 26.16 cm
```

Параметр replace по умолчанию установлен внутри функции на True

Новое среднее по выборке равно 26,16 см. При выполнении выборки с восполнением полученные средние значения будут колебаться: колебания подразумевают случайность, в связи с чем наши средние значения распределяются случайным образом. Давайте изучим форму этого распределения, повторив процесс выборки 150 000 раз. В ходе повторений мы вычисляем среднее 20 случайных рыб, после чего строим гистограмму для 150 000 полученных средних значений (листинг 7.19; рис. 7.3).

Листинг 7.19. Построение графика распределения 150 000 средних

```
np.random.seed(0)
sample_means = [np.random.choice(fish_lengths,
                                 size=20,
                                 replace=True).mean()
                for _ in range(150000)]
likelihoods, bin_edges, _ = plt.hist(sample_means, bins='auto',
                                     edgecolor='black', density=True)
plt.xlabel('Binned Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

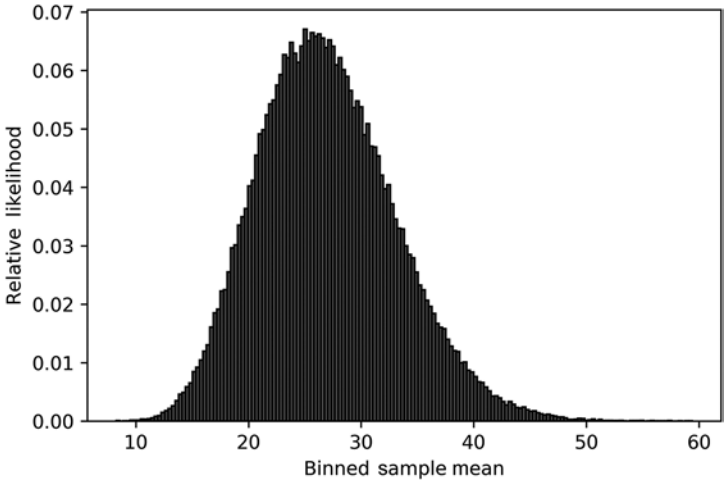


Рис. 7.3. Гистограмма средних выборки, вычисленных при помощи сэмплинга с восполнением. В этом случае гистограмма имеет не колоколообразную, а асимметричную форму

Сгенерированная гистограмма не является нормальной кривой. Перед нами несимметричная форма — подъем ее левой стороны оказался более крутым, чем правой. В математике такая асимметрия называется скосом и выражается коэффициентом асимметрии. Убедиться в присутствии скоса можно с помощью вызова `stats.skew(sample_means)`. Метод `stats.skew` возвращает ненулевое значение, когда входные данные оказываются асимметричными (листинг 7.20).

Листинг 7.20. Вычисление коэффициента асимметрии распределения

```
assert abs(stats.skew(sample_means)) > 0.4
```

Никакие данные не бывают абсолютно симметричными, и скос редко оказывается равен 0,0, даже если выборка делается из нормальной кривой. Однако для нормальных данных характерна величина скоса, близкая к 0,0. Любые данные, коэффициент асимметрии которых превышает 0,04, вряд ли получены из нормального распределения

Нашу асимметричную гистограмму невозможно смоделировать с помощью нормального распределения. Тем не менее она представляет непрерывное распределение

вероятностей. Как и все непрерывные распределения, эту гистограмму можно сопоставить с функцией плотности вероятностей, функцией распределения и функцией выживаемости. Здесь будет полезно знать, каков вывод этих функций. К примеру, функция выживаемости сообщит нам вероятность получения среднего выборки, которое больше среднего совокупности. Выводы функций можно получить, вручную написав код, вычисляющий площадь под кривой на основе массивов `bin_edges` и `likelihoods`.

В качестве альтернативы можно использовать SciPy, которая предоставляет метод `stats.rv_histogram`, позволяющий получить из гистограммы вывод всех трех функций. Этот метод принимает на входе кортеж, определяемый массивами `bin_edges` и `likelihoods`. Вызов `stats.rv_histogram((likelihoods, bin_edges))` ведет к возврату объекта SciPy `random_variable`, содержащего методы `pdf`, `cdf` и `sf`, также как `stats.norm`. Метод `random_variable.pdf` выводит плотность вероятностей гистограммы. Аналогично методы `random_variable.cdf` и `random_variable.sf` выводят функцию распределения и функцию выживаемости соответственно.

Код листинга 7.21 вычисляет объект `random_variable`, формируемый из гистограммы. Затем с помощью вызова `random_variable.pdf(bin_edges)` мы отражаем плотность распределения вероятностей (рис. 7.4).

Листинг 7.21. Сопоставление данных с общим распределением при помощи SciPy

```
random_variable = stats.rv_histogram((likelihoods, bin_edges))
plt.plot(bin_edges, random_variable.pdf(bin_edges))
plt.hist(sample_means, bins='auto', alpha=0.1, color='r', density=True)
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

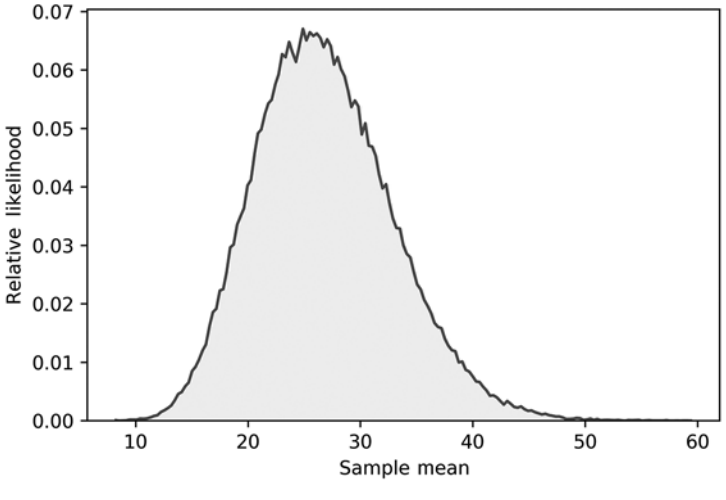


Рис. 7.4. Асимметричная гистограмма с наложением плотности распределения вероятностей. Плотность вероятностей была получена из этой гистограммы при помощи SciPy

Как и ожидалось, кривая плотности распределения вероятностей идеально повторяет форму гистограммы. Теперь мы построим функцию распределения и функцию выживаемости, связанные с `random_variable`. Следует ожидать, что эти две функции не будут симметричными относительно среднего. Чтобы наглядно показать асимметрию, отобразим на графике среднее распределения вертикальной линией (листинг 7.22). Само же среднее получим из вызова `random_variable.mean()` (рис. 7.5).

Листинг 7.22. Построение среднего и интервалов для общего распределения

```
rv_mean = random_variable.mean()
print(f"Mean of the distribution is approximately {rv_mean:.2f} cm")

plt.axvline(random_variable.mean(), color='k', label='Mean', linestyle=':')
plt.plot(bin_edges, random_variable.cdf(bin_edges),
         label='Cumulative Distribution')
plt.plot(bin_edges, random_variable.sf(bin_edges),
         label='Survival', linestyle='--', color='r')
plt.xlabel('Sample Mean')
plt.ylabel('Probability')
plt.legend()
plt.show()
```

```
Mean of the distribution is approximately 27.00 cm
```

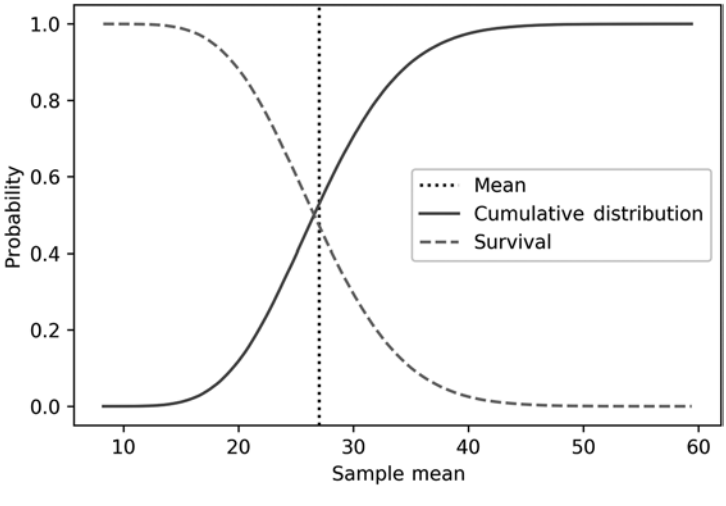


Рис. 7.5. Функция распределения вместе с функцией выживаемости. Эти две функции больше не отражаются симметрично относительно среднего, как было при анализе нормальной кривой. Следовательно, уже нельзя вычислить *p*-значение простым удвоением выходного значения функции выживаемости

Среднее распределения равно примерно 27 см, что совпадает со средней длиной рыбы в аквариуме. Случайная выборка рыб с высокой вероятностью даст значение, близкое к среднему по аквариуму. Однако сэмплинг с восполнением иногда дает значение более 37 и менее 17 см. Вероятности получения этих крайних величин можно вычислить на основе двух отображенных нами на графике функций. Разберем их более подробно.

Если исходить из графика, то функция распределения и функция выживаемости не являются зеркальными. Они также не пересекаются непосредственно в позиции среднего, как было при анализе нормальной кривой. Данное распределение не действует подобно симметричной нормальной кривой, что порождает определенные следствия. С помощью симметричной кривой можно было вычислять *p*-значения путем удваивания функции выживаемости. В асимметричном же распределении одной только функции выживаемости оказывается недостаточно для вычисления вероятностей в хвостовых частях. К счастью, используя эту функцию совместно с функцией распределения, можно раскрыть вероятности получения экстремальных значений и уже на их основе определить статистическую значимость.

Для измерения значимости нужно будет ответить на вопрос: «Какова вероятность того, что 20 отобранных (с восполнением) рыб дадут среднее, которое окажется таким же экстремальным, как и среднее совокупности?» Напомню: по совокупности среднее равно 37 см, что на 10 см больше среднего распределения. Следовательно, *экстремальность* определяется как вывод сэмплинга, удаленный от `rv_mean` не менее чем на 10 см. Исходя из прежних рассуждений, данную задачу можно разбить на вычисление двух отдельных значений. Во-первых, нужно вычислить вероятность получения среднего выборки не менее 37 см, а во-вторых — вероятность получения среднего выборки меньше либо равного 17 см. Первая вероятность равна `random_variable.sf(37)`, а вторая — `random_variable.cdf(17)`. Суммирование этих значений даст искомый ответ (листинг 7.23).

Листинг 7.23. Вычисление вероятности получения экстремального среднего выборки

```
prob_extreme= random_variable.sf(37) + random_variable.cdf(17)
print("Probability of observing an extreme sample mean is approximately "
      f"{prob_extreme:.2f}")

Probability of observing an extreme sample mean is approximately 0.10
```

Вероятность получения в результате сэмплинга экстремального значения — примерно 0,10. Иначе говоря, одна десятая часть случайных выборок из аквариума будет давать среднее, которое окажется не менее экстремальным, чем среднее совокупности. Среднее совокупности не настолько удалено от среднего аквариума, как нам казалось. На деле расхождение среднего 10 см и более будет встречаться

в 10 % выборок. Значит, разница между средним выборки, равным 27 см, и средним совокупности, равным 37 см, не является статистически значимой.

Теперь все это должно показаться знакомым. Величина `prob_extreme` — это просто скрытое *p*-значение. Когда нулевая гипотеза оказывается верна, разница между средним выборки и средним совокупности будет составлять не менее 10 единиц в 10 % случаев. *P*-значение 0,1 больше установленного порога 0,05, значит, отвергнуть нулевую гипотезу нельзя. Получается, что между средним выборки и средним совокупности нет какой-либо статистической значимости.

Мы вычислили *p*-значение обходным путем. Некоторым читателям наши методы могут показаться подозрительными — все же выборка из ограниченной коллекции в 20 рыб кажется странным способом получения статистических выводов. Тем не менее описанная техника вполне пригодна для использования. Бутстрэппинг с восполнением — надежный способ извлечения *p*-значения, особенно в работе с ограниченными данными.

ПОЛЕЗНЫЕ МЕТОДЫ ДЛЯ БУТСТРЭППИНГА С ВОСПОЛНЕНИЕМ

- `rv = stats.rv_histogram((likelihoods, bin_edges))` — создает объект случайной величины `rv` на основе полученных из гистограммы `likelihoods`, `bin_edges = np.hist(data)`.
- `p_value = rv.sf(head_extreme) + random_variable.cdf(tail_extreme)` — вычисляет *p*-значение из случайной величины, полученной на основе вывода функции выживаемости и функции распределения для передних экстремальных значений и хвостовых экстремальных значений соответственно.
- `z = np.random.choice(x, size=y, replace=True)` — выбирает `y` элементов из массива `x` с восполнением. Полученные образцы сохраняются в массиве `z`. В бутстрэппинге с восполнением `y == x.size`.

Техника бутстрэппинга тщательно изучалась на протяжении более чем 40 лет. Статистики нашли множество ее вариаций для вычисления точного *p*-значения. Мы здесь разобрали лишь одну подобную вариацию и в дальнейшем познакомимся еще с одной. Ранее было показано, что сэмплинг с восполнением аппроксимируется к SEM набора данных. Как правило, стандартное отклонение полученной выборки равно SEM, когда нулевая гипотеза истинна. Таким образом, если нулевая гипотеза истинна, недостающая SEM будет равна `random_variable.std`. Это дает нам еще один способ нахождения *p*-значения. Нужно лишь выполнить `compute_p_value(27, 37, random_variable.std)`, и полученное в итоге *p*-значение должно равняться приблизительно 0,1. Проверим (листинг 7.24).

Листинг 7.24. Использование бутстрэппинга для оценки SEM

```
estimated_sem = random_variable.std()
p_value = compute_p_value(27, 37, estimated_sem)
print(f"P-value computed from estimated SEM is approximately {p_value:.2f}")

P-value computed from estimated SEM is approximately 0.10
```

Как и ожидалось, полученное *p*-значение равно примерно 0,1. Мы показали, что бутстрэппинг с восполнением предоставляет два различных подхода для вычисления *p*-значения. Первый требует выполнить следующие действия.

1. Сформировать из данных выборку, используя восполнение. Повторить процесс десятки тысяч раз, чтобы получить список средних выборок.
2. Сгенерировать на основе средних выборок гистограмму.
3. Преобразовать гистограмму в распределение с помощью метода `stats.rv_histogram`.
4. Получить площадь под левым и правым краями кривой распределения с помощью функции выживаемости и функции распределения.

Второй подход оказывается несколько проще.

1. Сформировать из данных выборку с помощью восполнения. Повторить процесс десятки тысяч раз, чтобы получить список средних выборок.
2. Вычислить стандартное отклонение средних, чтобы приблизительно оценить SEM.
3. Использовать полученную SEM для тестирования базовой гипотезы с помощью функции `compute_p_value`.

Далее кратко разберем третий подход, который реализуется еще проще. Он не потребует построения гистограммы, равно как не будет опираться на `compute_value_function`. Вместо этого здесь задействуется закон больших чисел, представленный в главе 2. Согласно этому закону, при достаточно большом размере выборки частота наблюдаемых событий аппроксимируется к вероятности возникновения события. Значит, можно оценить *p*-значение, просто вычислив частоту наблюдения экстремальных величин. Давайте применим эту технику к `sample_mean`, подсчитав средние, которые не попадают в диапазон 17–37 см. Затем для получения *p*-значения разделим это количество на `len(sample_means)` (листинг 7.25).

Листинг 7.25. Вычисление p-значения из прямого подсчета

```
number_extreme_values = 0
for sample_mean in sample_means:
    if not 17 < sample_mean < 37:
        number_extreme_values += 1
```

```
p_value = number_extreme_values / len(sample_means)
print(f"P-value is approximately {p_value:.2f}")

P-value is approximately 0.10
```

Бутстрэппинг с восполнением — это простой, но мощный способ получения информации на основе ограниченных данных. Однако он все равно требует знания среднего генеральной совокупности. К сожалению, в реальных ситуациях среднее по совокупности известно редко. К примеру, в данном практическом задании нам нужно проанализировать таблицу переходов по онлайн-объявлениям, которая не включает среднего совокупности. Но недостаток информации нас не остановит, и в следующем разделе вы узнаете, как сравнивать собранные выборки, когда неизвестны ни среднее совокупности, ни ее дисперсия.

7.4. ПЕРМУТАЦИОННЫЙ ТЕСТ: СРАВНЕНИЕ СРЕДНИХ ВЫБОРОК ПРИ НЕИЗВЕСТНЫХ ПАРАМЕТРАХ СОВОКУПНОСТИ

Иногда в статистике нам требуется сравнить средние двух разных выборок при неизвестных параметрах генеральной совокупности. Далее мы разберем подобную ситуацию.

Предположим, что у нашей соседки тоже есть аквариум. В нем плавают десять рыб средней длиной 46 см. Длины этих новых рыб мы представим массивом `new_fish_lengths` (листинг 7.26).

Листинг 7.26. Определение длин рыб в новом аквариуме

```
new_fish_lengths = np.array([51, 46.5, 51.6, 47, 54.4, 40.5, 43, 43.1,
                             35.9, 47.0])
assert new_fish_lengths.mean() == 46
```

Нам нужно сравнить обитателей аквариума соседки с обитателями нашего. Для начала найдем разницу между `new_fish_lengths.mean()` и `fish_lengths.mean()` (листинг 7.27).

Листинг 7.27. Вычисление разницы между средними двух выборок

```
mean_diff = abs(new_fish_lengths.mean() - fish_lengths.mean())
print(f"There is a {mean_diff:.2f} cm difference between the two means")

There is a 19.00 cm difference between the two means
```

Различие между средними значениями длин рыб из двух аквариумов составляет 19 см. Это значительная разница, но является ли она статистически значимой?

Нужно разобраться. Все наши прежние анализы опирались на среднее совокупности. Сейчас же у нас есть средние двух выборок, но нет среднего по совокупности. Это усложняет задачу оценки нулевой гипотезы, которая предполагает, что средняя длина рыб в обоих аквариумах одинакова. Это предполагаемое общее значение теперь неизвестно. Что же делать?

Нам нужно переформулировать нулевую гипотезу, чтобы она не зависела непосредственно от среднего совокупности. Если нулевая гипотеза верна, тогда все 20 рыб из первого аквариума и 10 рыб из второго взяты из одной совокупности. При истинности данной гипотезы не имеет значения, какие 20 рыб окажутся в аквариуме A, а какие 10 — в аквариуме B. Перемещение рыб между этими двумя аквариумами не будет иметь особого значения. Такая случайная перестановка приведет к колебаниям переменной `mean_diff`, но эта разница между средними должна колебаться предсказуемым образом.

Получается, что для оценки нулевой гипотезы не обязательно знать среднее выборки. Вместо этого можно сосредоточиться на случайных пермутациях рыбы между двумя аквариумами. Это позволит нам выполнить *пермутационный тест*, в котором для вычисления статистической значимости будет использоваться `mean_diff`. Подобно бутстрэппингу с восполнением этот пермутационный тест опирается на случайный сэмплинг данных.

Начинается тест с помещения всех 30 рыб в один аквариум. Эту унификацию всех рыб можно смоделировать с помощью метода `np.hstack`. Он получает на входе список массивов NumPy, которые сливает в один массив (листинг 7.28).

Листинг 7.28. Слияние двух массивов с помощью np.hstack

```
total_fish_lengths = np.hstack([fish_lengths, new_fish_lengths])
assert total_fish_lengths.size == 30
```

После группировки рыб мы позволим им поплавать в произвольных направлениях, что полностью рандомизирует их положение в аквариуме. В коде подобное перемешивание позиций рыб реализуем с помощью метода `np.random.shuffle` (листинг 7.29).

Листинг 7.29. Перемешивание позиций объединенных рыб

```
np.random.seed(0)
np.random.shuffle(total_fish_lengths)
```

Далее мы отбираем 20 из перемешанных рыб и помещаем их в отдельный аквариум. Остальные 10 остаются в первом. Теперь у нас снова есть 20 рыб в аквариуме A и 10 — в аквариуме B. Однако средние длины рыб в каждом из аквариумов наверняка будут отличаться от `fish_lengths.mean()` и `new_fish_lengths.mean()`, значит, разница между средними длинами рыб также изменится. Убедимся в этом (листинг 7.30).

Листинг 7.30. Вычисление разницы между средними двух случайных выборок

```
random_20_fish_lengths = total_fish_lengths[:20]
random_10_fish_lengths = total_fish_lengths[20:]
mean_diff = random_20_fish_lengths.mean() – random_10_fish_lengths.mean()
print(f"The new difference between mean fish lengths is {mean_diff:.2f}")

The new difference between mean fish lengths is 14.33
```

Полученная разница теперь равна уже не 19, а 14,33 см. Как и ожидалось, `mean_diff` является колеблющейся случайной величиной, значит, можно выяснить ее распределение путем случайного сэмплинга. Далее мы повторяем процесс перемешивания рыб 30 000 раз и получаем гистограмму значений `mean_diff` (листинг 7.31; рис. 7.6).

Листинг 7.31. Построение графика колеблющейся разницы между средними

```
np.random.seed(0)
mean_diffs = []
for _ in range(30000):
    np.random.shuffle(total_fish_lengths)
    mean_diff = total_fish_lengths[:20].mean() -
        total_fish_lengths[20:].mean()
    mean_diffs.append(mean_diff)

likelihoods, bin_edges, _ = plt.hist(mean_diffs, bins='auto',
                                     edgecolor='black', density=True)
plt.xlabel('Binned Mean Difference')
plt.ylabel('Relative Likelihood')
plt.show()
```

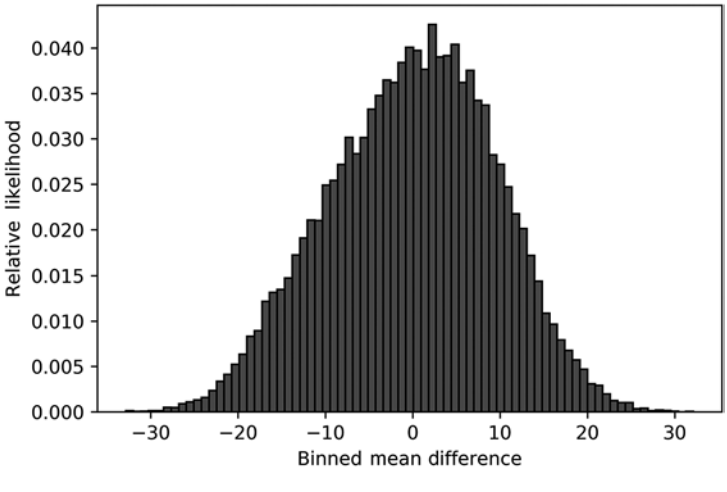


Рис. 7.6. Гистограмма различий в средних выборок, вычисленных с помощью случайного переупорядочения образцов в двух отдельных группах

Теперь сопоставляем полученную гистограмму со случайной переменной с помощью метода `stats.rv_histogram` (листинг 7.32).

Листинг 7.32. Сопоставление гистограммы со случайной переменной

```
random_variable = stats.rv_histogram((likelihoods, bin_edges))
```

Наконец, с помощью объекта `random_variable` тестируем гипотезу. Нам нужно узнать вероятность получения экстремального значения при истинности нулевой гипотезы. Экстремальность мы определяем как разницу между средними, имеющую абсолютную величину не менее 19 см. Таким образом, *p*-значение будет равняться `random_variable.cdf(-19)` + `random_variable.sf(19)` (листинг 7.33).

Листинг 7.33. Вычисление p-значения для пермутации

```
p_value = random_variable.sf(19) + random_variable.cdf(-19)
print(f"P-value is approximately {p_value:.2f}")

P-value is approximately 0.04
```

P-значение равно приблизительно 0,04, что немного ниже установленного порога значимости 0,05. Выходит, средняя разница между длинами рыб статистически значима. Получается, что рыбы в двух аквариумах происходят не из общего распределения.

В качестве отступления можно упростить пермутационный тест, используя закон больших чисел. Нужно просто вычислить частоту экстремальных зафиксированных образцов, как мы делали при бутстрэппинге с восполнением. Код листинга 7.34 применяет этот альтернативный метод для повторного вычисления *p*-значения, равного примерно 0,04.

Листинг 7.34. Вычисление p-значения пермутации из прямого подсчета

```
number_extreme_values = 0.0
for min_diff in mean_diffs:
    if not -19 < min_diff < 19:
        number_extreme_values += 1

p_value = number_extreme_values / len(mean_diffs)
print(f"P-value is approximately {p_value:.2f}")

P-value is approximately 0.04
```

Пермутационный тест позволяет статистически сравнить разницу между двумя списками собранных образцов. Природа этих образцов роли не играет. Ими могут быть длины рыб либо количество переходов по объявлению. Подобный тест поможет сравнить зарегистрированные количества переходов по объявлениям, чтобы определить для них наиболее эффективные цвета.

РЕЗЮМЕ

- Тестирование статистических гипотез требует выбирать между двух противоположных их версий. Согласно *нулевой гипотезе*, две совокупности идентичны. Согласно же *альтернативной гипотезе*, две совокупности не идентичны.
- Для оценки нулевой гипотезы необходимо вычислить *p-значение*. Оно равняется вероятности получения наших данных в случаях, когда нулевая гипотеза верна. Эта гипотеза отвергается, если *p*-значение меньше установленного порога *уровня значимости*. Как правило, этот порог определяется равным 0,05.
- Если верная нулевая гипотеза отвергнута, возникает *ошибка первого рода*. Если же нулевую гипотезу не отвергнуть, а истинной окажется альтернативная гипотеза, то возникнет *ошибка второго рода*.
- *Выуживание данных* повышает риск допустить ошибку первого рода. В случае выуживания данных испытание повторяется до тех пор, пока *p*-значение не попадет под заданный уровень значимости. Минимизировать выуживание данных можно с помощью *поправки Бонферрони*, которая подразумевает деление уровня значимости на общее число испытаний.
- Сравнить среднее выборки со средним и дисперсией совокупности можно с помощью центральной предельной теоремы. Дисперсия совокупности необходима для вычисления SEM. Если же дисперсия совокупности неизвестна, можно примерно оценить SEM, используя *бутстрэппинг с восполнением*.
- Сравнить средние двух отдельных выборок можно с помощью *пермутационного теста*.

8 Анализ таблиц с помощью Pandas

В этой главе

- ✓ Сохранение 2D-таблиц с помощью библиотеки Pandas.
- ✓ Суммирование содержимого 2D-таблицы.
- ✓ Управление содержимым строк и столбцов.
- ✓ Визуализация таблиц при помощи библиотеки Seaborn.

Данные о переходах по объявлениям из второго практического задания сохранены в двухмерной таблице. Информация зачастую хранится именно в таблицах данных. Эти таблицы могут быть представлены в различных форматах — некоторые сохраняются в виде электронных таблиц Excel, а другие представляют собой текстовые CSV-файлы, в которых столбцы разделяются запятыми. Форматирование таблицы значения не имеет. Важную роль играет ее структура. Все таблицы имеют общие структурные особенности — горизонтальные строки и вертикальные столбцы, а в заголовках столбцов зачастую отражаются их имена.

8.1. СОХРАНЕНИЕ ТАБЛИЦ С ПОМОЩЬЮ ЧИСТОГО PYTHON

Давайте определим образец таблицы в Python (листинг 8.1). Она будет хранить размеры различных видов рыб в сантиметрах. В таблице будут три столбца: `Fish`, `Length` и `Width`. Столбец `Fish` хранит названия видов рыб, а в столбцах `Length`

и `Width` указаны их длина и ширина. Саму таблицу мы представим как словарь. Имена столбцов будут выступать в роли его ключей, сопоставленных со списками значений из столбцов.

Листинг 8.1. Сохранение таблицы с помощью структур данных Python

```
fish_measures = {'Fish': ['Angelfish', 'Zebrafish', 'Killifish', 'Swordtail'],
                 'Length':[15.2, 6.5, 9, 6],
                 'Width': [7.7, 2.1, 4.5, 2]}
```

Предположим, мы хотим узнать длину данио-рерио (Zebrafish). Для этого необходимо сначала обратиться к индексу элемента `'Zebrafish'` в `fish_measures['Fish']`. Затем проверить этот индекс в `fish_measures['Length']`. Как видно в коде листинга 8.2, процесс получается немного путаным.

Листинг 8.2. Обращение к столбцам таблицы при помощи словаря

```
zebrafish_index = fish_measures['Fish'].index('Zebrafish')
zebrafish_length = fish_measures['Length'][zebrafish_index]
print(f"The length of a zebrafish is {zebrafish_length:.2f} cm")

The length of a zebrafish is 6.50 cm
```

Такое представление в виде словаря вполне действенно, но сложно в использовании. Более удачное решение предлагает библиотека Pandas, созданная специально для работы с таблицами.

8.2. ИЗУЧЕНИЕ ТАБЛИЦ С ПОМОЩЬЮ PANDAS

Для начала установим Pandas, после чего импортируем ее как `pd` с помощью стандартного соглашения использования Pandas (листинг 8.3).

ПРИМЕЧАНИЕ

Для установки Pandas выполните в командной строке pip install pandas.

Листинг 8.3. Импорт библиотеки Pandas

```
import pandas as pd
```

Теперь загружаем таблицу `fish_measures` в Pandas вызовом `pd.DataFrame(fish_measures)` (листинг 8.4). В ответ этот метод вернет объект Pandas `DataFrame`. Термин *«датафрейм»* (фрейм данных) на статистическом жаргоне — распространенный синоним *таблицы*. По сути, объект `DataFrame` преобразует наш словарь в двухмерную таблицу. По соглашению объекты `DataFrame` в Pandas присваиваются переменной `df`. Здесь мы выполняем `df = pd.DataFrame(fish_measures)`, после чего выводим содержимое `df`.

Листинг 8.4. Загрузка таблицы в Pandas

```
df = pd.DataFrame(fish_measures)
print(df)
```

```
        Fish  Length  Width
1  Angelfish    15.2    7.7
2  Zebrafish     6.5    2.1
3  Killifish     9.0    4.5
4  Swordtail     6.0    2.0
```

В полученном выводе очевидно выравнивание между строками и столбцами таблицы. В нашем случае таблица мала и вывести ее несложно. Однако для более крупных таблиц может оказаться предпочтительнее выводить только первые несколько строк. Вызов `print(df.head(x))` выводит лишь начальные *x* строк таблицы. Давайте выведем первые две строки с помощью `print(df.head(2))` (листинг 8.5).

Листинг 8.5. Обращение к первым двум строкам таблицы

```
print(df.head(2))
```

```
        Fish  Length  Width
0  Angelfish    15.2    7.7
1  Zebrafish     6.5    2.1
```

Более удачным способом обобщения крупных таблиц Pandas является выполнение `pd.describe()`. По умолчанию этот метод генерирует статистику для всех численных столбцов таблицы. Этот статистический вывод включает минимальное и максимальное значения столбцов, а также стандартное отклонение и среднее значение. В нашем случае при вводе `pd.describe()` следует ожидать получения информации из численных столбцов `Length` и `Width`, но не из строкового столбца `Fish` (листинг 8.6).

ВЫВОДИМЫЕ МЕТОДОМ DESCRIBE ПОКАЗАТЕЛИ

- `count` — количество элементов в каждом столбце.
- `mean` — среднее элементов в каждом столбце.
- `std` — стандартное отклонение элементов в каждом столбце.
- `min` — минимальное значение в каждом столбце.
- `25%` — 25 % элементов столбца оказываются ниже этого значения.
- `50%` — 50 % элементов столбца оказываются ниже этого значения. Само значение при этом является идентичным медиане.
- `75%` — 75 % элементов столбца оказываются ниже этого значения.
- `max` — максимальное значение в каждом столбце.

Листинг 8.6. Обобщение численных столбцов

```
print(df.describe())
```

```
          Length     Width
count   4.000000  4.000000
mean    9.175000  4.075000
std     4.225616  2.678775
min     6.000000  2.000000
25%     6.375000  2.075000
50%     7.750000  3.300000
75%    10.550000  5.300000
max    15.200000  7.700000
```

Согласно полученному обобщению, среднее для `Length` равно 9,175 см, а среднее для `Width` — 4,075 см. В вывод включается также дополнительная статистическая информация. Иногда она оказывается ненужной — если нас интересует лишь среднее, то можно опустить прочий вывод, вызвав `df.mean()` (листинг 8.7).

Листинг 8.7. Вычисление среднего по столбцу

```
print(df.mean())
```

```
Length    9.175
Width     4.075
dtype: float64
```

Метод `df.describe()` выполняется для численных столбцов. Однако можно заставить его обработать и строки, вызвав `df.describe(include=[np.object])`. Установка параметра `include` равным `[np.object]` инструктирует Pandas искать столбцы таблицы, построенные на основе строковых массивов NumPy. Поскольку нельзя выполнить статистический анализ для строк, итоговый вывод не будет содержать статистическую информацию. Вместо этого в нем будут подсчитаны общее число уникальных строк и частота, с которой встречается самая распространенная из них. Сама эта строка также сюда включается. Столбец `Fish` содержит четыре уникальные строки, каждая из которых упомянута лишь раз (листинг 8.8). Следовательно, мы ожидаем, что наиболее частая строка будет выбрана случайно с частотой 1.

Листинг 8.8. Обобщение строковых столбцов

```
print(df.describe(include=[np.object]))
```

```
count           4
unique          4
top     Zebrafish
freq            1
```

Количество строк в каждом столбце

Количество уникальных строк в каждом столбце

Самая частая строка в каждом столбце

Частота, с которой встречается самая частая строка

Как уже говорилось, столбец `Fish` строится на основе строкового массива NumPy. В действительности весь датафрейм строится на основе двухмерного массива NumPy. С целью более оперативного управления Pandas сохраняет все данные в NumPy. Внутренний же массив NumPy можно извлечь с помощью `df.values` (листинг 8.9).

МЕТОДЫ ОБОБЩЕНИЯ В PANDAS

- `df.head()` — возвращает первые пять строк датафрейма `df`.
- `df.head(x)` — возвращает первые *x* строк датафрейма `df`.
- `df.describe()` — возвращает статистику по численным столбцам в `df`.
- `df.describe(include=[np.object])` — возвращает статистику по строковым столбцам в `df`.
- `df.mean()` — возвращает средние для всех численных столбцов в `df`.

Листинг 8.9. Извлечение таблицы в виде 2D-массива NumPy

```
print(df.values)
assert type(df.values) == np.ndarray
```

```
[['Angelfish' 15.2 7.7]
 ['Zebrafish' 6.5 2.1]
 ['Killifish' 9.0 4.5]
 ['Swordtail' 6.0 2.0]]
```

8.3. ИЗВЛЕЧЕНИЕ СТОЛБЦОВ ТАБЛИЦЫ

Теперь переключим внимание на извлечение отдельных столбцов, к которым можно обращаться по их именам. Вывод всех имен столбцов реализуется вызовом `print(df.columns)` (листинг 8.10).

Листинг 8.10. Вывод имен всех столбцов

```
print(df.columns)
```

```
Index(['Fish', 'Length', 'Width'], dtype='object')
```

Далее мы выведем все данные, хранящиеся в столбце `Fish`, выполнив `df.Fish` (листинг 8.11).

Листинг 8.11. Обращение к отдельному столбцу

```
print(df.Fish)
```

```
0    Angelfish
1    Zebrafish
2    Killifish
3    Swordtail
Name: Fish, dtype: object
```

Имейте в виду, что выводится здесь не массив NumPy. Напротив, `df.Fish` — это объект Pandas, который представляет одномерный массив. Для вывода массива NumPy нужно выполнить `print (df.Fish.values)` (листинг 8.12).

Листинг 8.12. Извлечение столбца в виде массива NumPy

```
print(df.Fish.values)
assert type(df.Fish.values) == np.ndarray
```

```
['Angelfish' 'Zebrafish' 'Killifish' 'Swordtail']
```

Мы обратились к столбцу `Fish` при помощи `df.Fish`. Иначе к `Fish` можно обратиться, используя подобную словарю скобочную форму `df['Fish']` (листинг 8.13).

Листинг 8.13. Обращение к столбцу с помощью скобок

```
print(df['Fish'])
```

```
0    Angelfish
1    Zebrafish
2    Killifish
3    Swordtail
Name: Fish, dtype: object
```

Скобочное представление позволяет извлекать множество столбцов с помощью выполнения `df[name_list]`, где `name_list` — это список имен столбцов. Предположим, мы хотим извлечь столбцы `Fish` и `Length`. Выполнение `df[['Fish', 'Length']]` приведет к возврату обрезанной таблицы, содержащей только эти два столбца (листинг 8.14).

Листинг 8.14. Обращение к нескольким столбцам с помощью скобок

```
print(df[['Fish', 'Length']])
```

```
        Fish  Length
0  Angelfish    15.2
1  Zebrafish     6.5
2  Killifish     9.0
3  Swordtail     6.0
```

Анализировать хранящиеся в `df` данные можно разными способами. К примеру, можно упорядочить строки на основании значения одного столбца. Вызов `df.sort_values('Length')` вернет новую таблицу, строки которой будут упорядочены по длине (листинг 8.15).

Листинг 8.15. Упорядочение строк по значению столбца

```
print(df.sort_values('Length'))
```

```
        Fish  Length  Width
3  Swordtail     6.0    2.0
1  Zebrafish     6.5    2.1
2  Killifish     9.0    4.5
0  Angelfish    15.2    7.7
```

Более того, с помощью значений из столбцов можно отфильтровывать ненужные строки. К примеру, вызов `df[df.Width >= 3]` вернет таблицу, строки которой будут содержать ширину не менее 3 см (листинг 8.16).

Листинг 8.16. Фильтрация строк по значению столбца

```
print(df[df.Width >= 3])
```

```
        Fish  Length  Width
0  Angelfish    15.2    7.7
2  Killifish     9.0    4.5
```

МЕТОДЫ ИЗВЛЕЧЕНИЯ СТОЛБЦОВ В PANDAS

- `df.columns` — возвращает имена столбцов датафрейма `df`.
- `df.x` — возвращает столбец `x`.
- `df[x]` — возвращает столбец `x`.
- `df[[x,y]]` — возвращает столбцы `x` и `y`.
- `df.x.values` — возвращает столбец `x` как массив NumPy.
- `df.sort_values(x)` — возвращает датафрейм, упорядоченный по значениям в столбце `x`.
- `df[df.x > y]` — возвращает датафрейм, отфильтрованный по значениям в столбце `x`, которые больше `y`.

8.4. ИЗВЛЕЧЕНИЕ СТРОК ТАБЛИЦЫ

А теперь рассмотрим извлечение строк в `df`. В отличие от столбцов строки не имеют присвоенных имен. Для восполнения этого недостатка Pandas присваивает каждой строке специальный индекс. Они отображаются с левого края таблицы. Судя по полученному выводу, индекс `Angelfish` — 0, а индекс `Swordtail` — 3. К этим строкам можно обратиться вызовом `df.loc[[0, 3]]`. Как правило, выполнение `df.loc[[index_list]]` находит все строки, чьи индексы входят в `index_list`. Найдем строки, которые совпадают с индексами `Swordtail` и `Angelfish` (листинг 8.17).

Листинг 8.17. Обращение к строкам по индексу

```
print(df.loc[[0, 3]])
```

```
        Fish  Length  Width
0  Angelfish    15.2    7.7
3  Swordtail     6.0    2.0
```

Предположим, мы хотим извлечь строки, используя названия видов рыб, а не численные индексы. Говоря точнее, нам нужно получить строки, столбец `Fish` которых содержит либо `'Angelfish'`, либо `'Swordtail'`. В Pandas такой процесс имеет свои особенности — необходимо выполнить `df[booleans]`, где `booleans` является списком логических значений, которые `True`, если соответствуют интересующей нас строке. По сути, индексы `True`-значений должны совпадать со строками, которые соответствуют `'Angelfish'` либо `'Whitefish'`. Как же получить список `booleans`? Первым простым подходом здесь будет перебрать `df.Fish`, возвращая `True`, если значение столбца находится в `['Angelfish', 'Swordtail']`. Код листинга 8.18 именно это и делает.

Листинг 8.18. Обращение к строкам по значению столбца

```
booleans = [name in ['Angelfish', 'Swordtail']
            for name in df.Fish]
print(df[booleans])

        Fish  Length  Width
0  Angelfish    15.2    7.7
3  Swordtail     6.0    2.0
```

Определить интересующие нас строки можно и более сжато с помощью метода `isin` (листинг 8.19). Вызов `df.Fish.isin(['Angelfish', Swordtail'])` вернет аналог ранее вычисленного списка `booleans`. Таким образом можно извлечь все строки с помощью одной строки кода, выполнив `df[df.Fish.isin(['Angelfish', 'Swordtail'])]`.

Листинг 8.19. Обращение к строкам по значению столбца с помощью isin

```
print(df[df.Fish.isin(['Angelfish', 'Swordtail'])])

        Fish  Length  Width
0  Angelfish    15.2    7.7
3  Swordtail     6.0    2.0
```

Таблица `df` хранит два размера четырех видов рыб. Можно с легкостью обратиться к размерам в столбцах, но вот обращение к строкам по видам уже сложнее, поскольку индексы строк не совпадают с названиями видов. Эту ситуацию можно исправить, заменив индексы строк названиями видов с помощью метода `df.set_index` (листинг 8.20). Вызов `df.set_index('Fish', inplace=True)` установит индексы равными названиям видов из столбца `Fish`. Параметр `inplace=True` модифицирует индексы изнутри, а не возвращает измененную копию `df`.

Листинг 8.20. Замена индексов строк значениями столбцов

```
df.set_index('Fish', inplace=True)
print(df)

           Fish  Length  Width
Angelfish    15.2    7.7
Zebrafish     6.5    2.1
Killifish     9.0    4.5
Swordtail     6.0    2.0
```

Левый крайний столбец больше не является численным — числа в нем были заменены названиями видов. Теперь можно обратиться к столбцам `Angelfish` и `Swordtail`, выполнив `df.loc[['Angelfish', 'Swordtail']]` (листинг 8.21).

Листинг 8.21. Обращение к строкам по строковому индексу

```
print(df.loc[['Angelfish', 'Swordtail']])
```

```
     Fish  Length  Width
Angelfish    15.2    7.7
Swordtail     6.0    2.0
```

МЕТОДЫ ИЗВЛЕЧЕНИЯ СТРОК В PANDAS

- `df.loc[[x, y]]` — возвращает строки, расположенные в индексах `x` и `y`.
- `df[booleans]` — возвращает строки, в которых `booleans[i]` является `True` для столбца `i`.
- `df[name in array for name in df.x]` — возвращает строки, в которых название столбца `x` присутствует в `array`.
- `df[df.x.isin(array)]` — возвращает строки, в которых название столбца `x` присутствует в `array`.
- `df.set_index('x', inplace=True)` — заменяет индексы строк значениями из столбца `x`.

8.5. ИЗМЕНЕНИЕ СТРОК И СТОЛБЦОВ ТАБЛИЦЫ

Сейчас каждая строка таблицы содержит длину и ширину указанной рыбы. А что произойдет, если поменять местами строки и столбцы? Выяснить это можно, выполнив `df.T`. `T` означает транспонирование — в ходе этой операции элементы таблицы переставляются по диагонали, в результате чего строки и столбцы меняются местами. Давайте транспонируем нашу таблицу и выведем результат (листинг 8.22).

Листинг 8.22. Перестановка местами строк и столбцов

```
df_transposed = df.T
print(df_transposed)
```

```
Fish    Angelfish  Zebrafish  Killifish  Swordtail
Length       15.2        6.5        9.0        6.0
Width         7.7        2.1        4.5        2.0
```

Таблица была модифицирована — каждый столбец теперь относится к отдельному виду рыбы, а каждая строка — к конкретному типу измерения. В первой строке содержится длина, а во второй — ширина. Таким образом, вызов `print(df_transposed.Swordtail)` выведет длину и ширину меченосца (`Swordtail`) (листинг 8.23).

Листинг 8.23. Вывод транспонированного столбца

```
print(df_transposed.Swordtail)

Length   6.0
Width    2.0
Name: Swordtail, dtype: float64
```

Далее мы изменим таблицу, добавив в `df_transposed` размеры рыбы-клоуна (`Clownfish`). Ее длина и ширина — 10,6 и 3,7 см соответственно. Эти размеры мы добавляем с помощью `df_transposed['Clownfish'] = [10.6, 3.7]` (листинг 8.24).

Листинг 8.24. Добавление нового столбца

```
df_transposed['Clownfish'] = [10.6, 3.7]
print(df_transposed)

Fish    Angelfish  Zebrafish  Killifish  Swordtail  Clownfish
Length       15.2        6.5        9.0        6.0       10.6
Width         7.7        2.1        4.5        2.0        3.7
```

В качестве альтернативы мы присваиваем новые столбцы, используя метод `df_transposed.assign`. С его помощью можно добавлять несколько столбцов, передавая ему более одного названия столбца. К примеру, вызов `df_transposed.assign(Clownfish2=[10.6, 3.7], Clown-fish3=[10.6, 3.7])` вернет таблицу с двумя новыми столбцами, `Clownfish2` и `Clownfish3` (листинг 8.25). Обратите внимание на то, что метод `assign` никогда не добавляет столбцы непосредственно в таблицу — вместо этого он возвращает ее копию, содержащую новые данные.

Листинг 8.25. Добавление нескольких новых столбцов

```
df_new = df_transposed.assign(Clownfish2=[10.6, 3.7], Clownfish3=[10.6, 3.7])
assert 'Clownfish2' not in df_transposed.columns
assert 'Clownfish2' in df_new.columns
print(df_new)

Fish    Angelfish  Zebrafish  Killifish  Swordtail  Clownfish  Clownfish2 \
Length       15.2        6.5        9.0        6.0       10.6        10.6
Width         7.7        2.1        4.5        2.0        3.7         3.7

Fish    Clownfish3
Length        10.6
Width          3.7
```

Добавленные нами столбцы оказываются лишними. Удаляем мы их вызовом `df_new.drop(columns=['Clownfish2', 'Clownfish3'], inplace=True)`. Метод `df_new.drop` исключает из таблицы все указанные столбцы (листинг 8.26).

Листинг 8.26. Удаление нескольких столбцов

```
df_new.drop(columns=['Clownfish2', 'Clownfish3'], inplace=True)
print(df_new)
```

```
Fish    Angelfish  Zebrafish  Killifish  Swordtail  Clownfish
Length       15.2        6.5        9.0        6.0       10.6
Width         7.7        2.1        4.5        2.0        3.7
```

Теперь мы используем сохраненные размеры для вычисления площади поверхности каждой рыбы. Для этого рыбу можно рассматривать как эллипс, площадь которого равна `math.pi * length * width / 4`. Чтобы вычислить все площади, необходимо перебрать значения в каждом столбце. Перебор столбцов датафрейма подобен перебору элементов словаря — достаточно выполнить `df_new.items()`. В результате вернется итерируемый объект кортежей, содержащий имена столбцов и их значения. В качестве примера код листинга 8.27 перебирает столбцы в `df_now`, вычисляя площадь каждой рыбы.

Листинг 8.27. Перебор значений столбца

```
areas = []
for fish_species, (length, width) in df_new.items():
    area = math.pi * length * width / 4
    print(f"Area of {fish_species} is {area}")
    areas.append(area)

Area of Angelfish is 91.92300104403735
Area of Zebrafish is 10.720684930375171
Area of Killifish is 31.808625617596654
Area of Swordtail is 9.42477796076938
Area of Clownfish is 30.80331596844792
```

Теперь добавим вычисленные площади в таблицу. Можно внести новую строку `Area`, выполнив `df_new.loc['Area'] = areas` (листинг 8.28). Далее нужно выполнить `df_new.reindex()`, чтобы дополнить индексы строк названием `Area`.

Листинг 8.28. Добавление новой строки

```
df_new.loc['Area'] = areas
df_new.reindex()
print(df_new)

Fish    Angelfish  Zebrafish  Killifish  Swordtail  Clownfish
Length  15.200000   6.500000   9.000000   6.000000  10.600000
Width    7.700000   2.100000   4.500000   2.000000   3.700000
Area    91.923001  10.720685  31.808626   9.424778  30.803316
```

Обновленная таблица теперь содержит три строки и пять столбцов. Убедиться в этом можно с помощью `df_new.shape` (листинг 8.29).

Листинг 8.29. Проверка формы таблицы

```
row_count, column_count = df_new.shape
print(f"Our table contains {row_count} rows and {column_count} columns")

Our table contains 3 rows and 5 columns
```

ИЗМЕНЕНИЕ ДАТАФРЕЙМОВ В PANDAS

- `df.T` — возвращает транспонированный датафрейм, в котором строки и столбцы поменялись местами.
- `df[x] = array` — создает новый столбец `x`. `df.x` сопоставляется со значениями в `array`.
- `df.assign(x=array)` — возвращает датафрейм, содержащий все элементы `df` и новый столбец `x`. `df.x` сопоставляется со значениями в `array`.
- `df.assign(x=array, y=array2)` — возвращает датафрейм, содержащий два новых столбца, `x` и `y`.
- `df.drop(columns=[x, y])` — возвращает датафрейм, в котором столбцы `x` и `y` были удалены.
- `df.drop(columns=[x, y], inplace=True)` — удаляет столбцы `x` и `y` по месту, тем самым изменяя `df`.
- `df.loc[x] = array` — добавляет строку по индексу `x`. Для доступа к этой строке необходимо выполнить `df.reindex()`.

8.6. СОХРАНЕНИЕ И ЗАГРУЗКА ТАБЛИЧНЫХ ДАННЫХ

Мы закончили внесение изменений в таблицу. Теперь нужно сохранить ее для последующего использования. Вызов `df_new.to_csv('Fish_measurements.csv')` приведет к сохранению таблицы в файл CSV, в котором столбцы разделены запятыми (листинг 8.30).

Листинг 8.30. Сохранение таблицы в CSV-файл

```
df_new.to_csv('Fish_measurements.csv')
with open('Fish_measurements.csv') as f:
    print(f.read())
```

```
,Angelfish,Zebrafish,Killifish,Swordtail,Clownfish
Length,15.2,6.5,9.0,6.0,10.6
Width,7.7,2.1,4.5,2.0,3.7
Area,91.92300104403735,10.720684930375171,31.808625617596654,9.42477796076938
,30.80331596844792
```

Файл CSV можно загрузить в Pandas с помощью метода `pd.read_csv` (листинг 8.31). Вызов `pd.read_csv('Fish_measurements.csv', index_col=0)` возвращает датафрейм, содержащий всю табличную информацию. Необязательный параметр `index_col` указывает, какой столбец содержит названия индексов строк. Если такой столбец не задан, строкам автоматически присваиваются численные индексы.

Листинг 8.31. Загрузка таблицы из CSV-файла

```
df = pd.read_csv('Fish_measurements.csv', index_col=0)
print(df)
print("\nRow index names when column is assigned:")
print(df.index.values)

df_no_assign = pd.read_csv('Fish_measurements.csv')
print("\nRow index names when no column is assigned:")
print(df_no_assign.index.values)
```

```
        Angelfish  Zebrafish  Killifish  Swordtail  Clownfish
Length  15.200000   6.500000   9.000000   6.000000  10.600000
Width    7.700000   2.100000   4.500000   2.000000   3.700000
Area    91.923001  10.720685  31.808626   9.424778  30.803316

Row index names when column is assigned:
['Length' 'Width' 'Area']

Row index names when no column is assigned:
[0 1 2]
```

С помощью `pd.csv` можно загрузить в Pandas нашу таблицу с данными о переходах по объявлениям, что позволит эффективно ее проанализировать.

СОХРАНЕНИЕ И ЗАГРУЗКА ДАТАФРЕЙМОВ В PANDAS

- `pd.DataFrame(dictionary)` — преобразует данные `dictionary` в датафрейм.
- `pd.read_csv(filename)` — преобразует CSV-файл в датафрейм.
- `pd.read_csv(filename, index_col=i)` — преобразует CSV-файл в датафрейм. Столбец `i` содержит имена индексов строк.
- `df.to_csv(filename)` — сохраняет содержимое `df` в CSV-файл.

8.7. ВИЗУАЛИЗАЦИЯ ТАБЛИЦ С ПОМОЩЬЮ SEABORN

Содержимое таблицы Pandas можно просмотреть с помощью простой команды `print`. Однако некоторые численные таблицы слишком велики для просмотра таким образом. Их проще отображать с помощью тепловых карт. *Тепловая карта* — это графическое представление таблицы, в котором численные ячейки раскрашиваются в зависимости от значения. Оттенки цвета при этом смещаются постепенно, отражая значения. Итоговым результатом становится обобщенное представление различий значений таблицы.

Проще всего тепловую карту создать с помощью библиотеки визуализации Seaborn. Она построена на основе Matplotlib и тесно интегрирована с датафреймами Pandas. Начнем с ее установки и импорта как `sns` (листинг 8.32).

ПРИМЕЧАНИЕ

Для установки библиотеки Seaborn выполните в терминале pip install seaborn.

Листинг 8.32. Импорт библиотеки Seaborn

```
import seaborn as sns
```

Теперь визуализируем датафрейм как тепловую карту с помощью `sns.heatmap(df)` (листинг 8.33; рис. 8.1).

Листинг 8.33. Визуализация тепловой карты с помощью Seaborn

```
sns.heatmap(df)
plt.show()
```

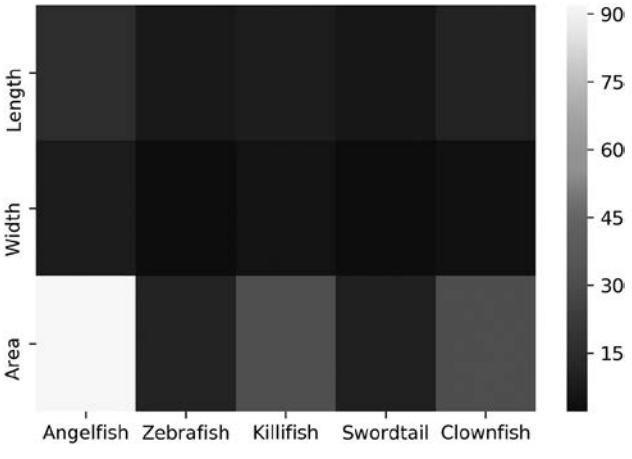


Рис. 8.1. Тепловая карта размеров рыб

Мы построили тепловую карту размеров рыб. Полученные цвета соответствуют конкретным значениям этих размеров. Сопоставление оттенков цветов и значений отражено в легенде справа от графика. Чем светлее цвет, тем больше соответствующее значение, чем темнее — тем меньше. Таким образом, можно с ходу понять, что площадь рыбы-ангела наибольшая.

Цветовую палитру тепловой карты можно изменять с помощью параметра `cmap`. Код листинга 8.34 выполняет `sns.heatmap(df, cmap='YlGnBu')`, создавая карту, в которой оттенки цветов переходят от желтого к зеленому, а затем к синему (рис. 8.2).

Листинг 8.34. Корректировка цветов тепловой карты

```
sns.heatmap(df, cmap='YlGnBu')
plt.show()
```

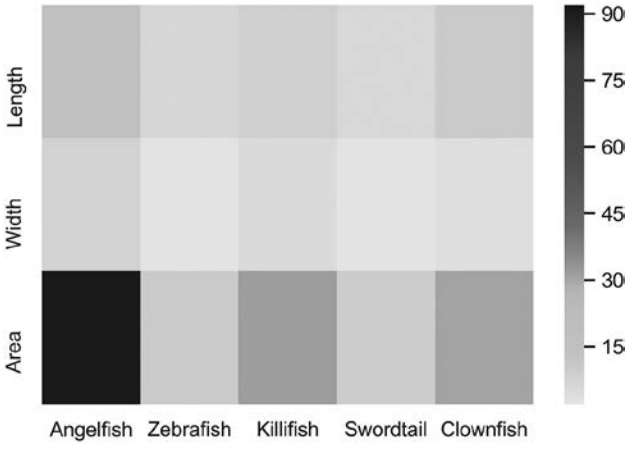


Рис. 8.2. Тепловая карта размеров рыб. Чем темнее цвет, тем больше значение, чем светлее — тем меньше

На обновленной тепловой карте оттенки цветов поменялись местами — теперь темные соответствуют бóльшим размерам. Подтвердить это можно, нанеся на график фактические размеры. Для разметки нужно передать `anot=True` в метод `sns.heatmap` (листинг 8.35; рис. 8.3).

Листинг 8.35. Аннотация тепловой карты

```
sns.heatmap(df, cmap='YlGnBu', annot=True)
plt.show()
```

Рис. 8.3. Аннотированная тепловая карта размеров рыб

Как уже говорилось, библиотека Seaborn построена на основе Matplotlib. Это значит, что для изменения содержимого тепловой карты можно использовать команды Matplotlib. К примеру, вызов `plt.yticks(rotation=0)` развернет метки измерений по оси *Y*, сделав их более удобными для чтения (листинг 8.36; рис. 8.4).

Листинг 8.36. Вращение меток тепловой карты с помощью Matplotlib

```
sns.heatmap(df, cmap='YlGnBu', annot=True)
plt.yticks(rotation=0)
plt.show()
```

Рис. 8.4. Тепловая карта размеров рыб. Метки на оси Y были повернуты для лучшего восприятия

Наконец, стоит отметить, что и метод `sns.heatmap` может обрабатывать 2D-списки и массивы. В результате выполнение `sns.heatmap(df.values)` тоже приведет к созданию тепловой карты, но без меток на осях *Y* и *X*. Для указания меток нужно передать в этот метод параметры `xticklabels` и `yticklabels`. Код листинга 8.37 задействует представление нашей таблицы в виде массива для воссоздания содержимого (рис. 8.4).

Листинг 8.37. Визуализация тепловой карты из массива NumPy

```
sns.heatmap(df.values,
            cmap='YlGnBu', annot=True,
            xticklabels=df.columns,
            yticklabels=df.index)
plt.yticks(rotation=0)
plt.show()
```

Напомню, что df.values возвращает 2D-массив NumPy, лежащий в основе датафрейма

Метки рыб на оси X вручную устанавливаются как равные именам столбцов

Метки размеров на оси Y вручную устанавливаются как равные индексам строк

КОМАНДЫ ВИЗУАЛИЗАЦИИ ТЕПЛОВОЙ КАРТЫ В SEABORN

- `sns.heatmap(array)` — генерирует тепловую карту из содержимого 2D `array`.
- `sns.heatmap(array, xticklabels=x, yticklabels=y)` — генерирует тепловую карту из содержимого 2D `array`. Метки на осях *X* и *Y* устанавливаются равными `x` и `y` соответственно.
- `sns.heatmap(df)` — генерирует тепловую карту из содержимого датафрейма `df`. Метки на осях *X* и *Y* автоматически устанавливаются равными `df.columns` и `df.index` соответственно.
- `sns.heatmap(df, cmap=m)` — генерирует тепловую карту, цветовая схема которой задается `m`.
- `sns.heatmap(df, annot=True)` — генерирует тепловую карту, включающую подписи значений.

РЕЗЮМЕ

- Двухмерные табличные структуры удобно обрабатывать с помощью Pandas. Загружать данные в Pandas можно при помощи словарей либо внешних файлов.
- Pandas сохраняет каждую таблицу в датафрейме, построенном на основе массива NumPy.
- Столбцы датафрейма имеют имена, по которым к ним можно обращаться. При этом строки датафрейма по умолчанию получают численные индексы, которые также можно использовать, чтобы обращаться к ним. При желании численные индексы можно заменить строковыми именами.
- Содержимое датафрейма обобщается с помощью метода `describe`. Он возвращает статистику, такую как среднее значение и стандартное отклонение.
- Содержимое датафрейма можно визуализировать с помощью цветной *тепловой карты*.

9 Решение практического задания 2

В этой главе

- ✓ Измерение статистической значимости.
- ✓ Пермутационное тестирование.
- ✓ Управление таблицами с помощью Pandas.

Перед нами стоит задача проанализировать данные о переходах по онлайн-объявлениям, собранные нашим приятелем Фредом. В таблице этих данных отслеживаются переходы по 30 разным цветам. От нас требуется определить среди них такой, который обусловливает намного большее число переходов, чем синий. Для этого мы проделаем несколько шагов.

1. Загрузим и очистим рекламные данные с помощью Pandas.
2. Проведем пермутационное тестирование между синим и остальными присутствующими в таблице цветами.
3. Проверим полученные *p*-значения на статистическую значимость, используя правильно выбранный уровень значимости.

ВНИМАНИЕ

Спойлер! Далее раскрывается решение второго практического задания. Настоятельно призываю попытаться справиться с этой задачей самостоятельно, прежде чем смотреть готовое решение. Исходные условия приведены в начале практического задания.

9.1. ОБРАБОТКА ТАБЛИЦЫ ПЕРЕХОДОВ ПО ОБЪЯВЛЕНИЮ В PANDAS

Начнем с загрузки таблицы в Pandas, после чего проверим количество строк и столбцов в ней (листинг 9.1).

Листинг 9.1. Загрузка таблицы данных о переходах в Pandas

```
df = pd.read_csv('colored_ad_click_table.csv')
num_rows, num_cols = df.shape
print(f"Table contains {num_rows} rows and {num_cols} columns")
```

```
Table contains 30 rows and 41 columns
```

Полученная таблица содержит 30 строк и 41 столбец. Строки должны соответствовать ежедневному числу просмотров и переходов, связанному с отдельными цветами. Убедимся в этом, проверив названия столбцов (листинг 9.2).

Листинг 9.2. Проверка имен столбцов

```
print(df.columns)
```

```
Index(['Color', 'Click Count: Day 1', 'View Count: Day 1',
       'Click Count: Day 2', 'View Count: Day 2', 'Click Count: Day 3',
       'View Count: Day 3', 'Click Count: Day 4', 'View Count: Day 4',
       'Click Count: Day 5', 'View Count: Day 5', 'Click Count: Day 6',
       'View Count: Day 6', 'Click Count: Day 7', 'View Count: Day 7',
       'Click Count: Day 8', 'View Count: Day 8', 'Click Count: Day 9',
       'View Count: Day 9', 'Click Count: Day 10', 'View Count: Day 10',
       'Click Count: Day 11', 'View Count: Day 11', 'Click Count: Day 12',
       'View Count: Day 12', 'Click Count: Day 13', 'View Count: Day 13',
       'Click Count: Day 14', 'View Count: Day 14', 'Click Count: Day 15',
       'View Count: Day 15', 'Click Count: Day 16', 'View Count: Day 16',
       'Click Count: Day 17', 'View Count: Day 17', 'Click Count: Day 18',
       'View Count: Day 18', 'Click Count: Day 19', 'View Count: Day 19',
       'Click Count: Day 20', 'View Count: Day 20'],
    dtype='object')
```

Столбцы вполне согласуются с нашими ожиданиями: первый содержит все проанализированные цвета, а остальные 40 — количество переходов и просмотров для каждого дня эксперимента. В качестве проверки правильности мы оценим качество хранящихся в таблице данных. Начнем с вывода проанализированных имен цветов (листинг 9.3).

Листинг 9.3. Проверка названий цветов

```
print(df.Color.values)
```

```
['Pink' 'Gray' 'Sapphire' 'Purple' 'Coral' 'Olive' 'Navy' 'Maroon' 'Teal'
'Cyan' 'Orange' 'Black' 'Tan' 'Red' 'Blue' 'Brown' 'Turquoise' 'Indigo'
'Gold' 'Jade' 'Ultramarine' 'Yellow' 'Viridian' 'Violet' 'Green'
'Aquamarine' 'Magenta' 'Silver' 'Bronze' 'Lime']
```

В столбце `Color` перечислены 30 распространенных цветов. Первая буква названия каждого из них — прописная. Таким образом, присутствие синего можно подтвердить выполнением `assert 'Blue' in df.Color` (листинг 9.4).

Листинг 9.4. Проверка присутствия синего

```
assert 'Blue' in df.Color.values
```

Строковый столбец `Color` выглядит так, как и должен. Теперь разберемся с оставшимися 40 численными столбцами. Вывод их всех приведет к чрезмерно большому количеству данных. Вместо этого мы проверим столбцы для первого дня эксперимента — `Click Count: Day 1` и `View Count: Day 1`. Выбираем эти столбцы и обобщаем их содержимое с помощью `describe()`(листинг 9.5).

Листинг 9.5. Обобщение данных первого дня эксперимента

```
selected_columns = ['Color', 'Click Count: Day 1', 'View Count: Day 1']
print(df[selected_columns].describe())
```

```
       Click Count: Day 1  View Count: Day 1
Count           30.000000               30.0
Mean            23.533333              100.0
std              7.454382                0.0
min             12.000000              100.0
25%             19.250000              100.0
50%             24.000000              100.0
75%             26.750000              100.0
max             49.000000              100.0
```

Значения в столбце `Click Count: Day 1` отражают от 12 до 49 переходов. При этом и минимальное, и максимальное значения просмотров в столбце `View Count: Day 1` равны 100. Следовательно, все значения этого столбца равны 100. Это ожидаемая ситуация, ведь изначально было сказано, что объявление каждого цвета просматривается ежедневно 100 раз. Убедимся, что все ежедневные просмотры равны 100 (листинг 9.6).

Листинг 9.6. Подтверждение количества ежедневных просмотров

```
view_columns = [column for column in df.columns if 'View' in column]
assert np.all(df[view_columns].values == 100)
```

Эффективный код NumPy, проверяющий, равна ли сумма значений в массиве NumPy 100

В каждом случае количество просмотров равно 100, значит, все 20 столбцов `ViewCount` являются излишними и их можно из таблицы удалить (листинг 9.7).

Листинг 9.7. Удаление из таблицы данных о количестве просмотров

```
df.drop(columns=view_columns, inplace=True)
print(df.columns)
```

```
Index(['Color', 'Click Count: Day 1', 'Click Count: Day 2',
       'Click Count: Day 3', 'Click Count: Day 4', 'Click Count: Day 5',
       'Click Count: Day 6', 'Click Count: Day 7', 'Click Count: Day 8',
       'Click Count: Day 9', 'Click Count: Day 10', 'Click Count: Day 11',
       'Click Count: Day 12', 'Click Count: Day 13', 'Click Count: Day 14',
       'Click Count: Day 15', 'Click Count: Day 16', 'Click Count: Day 17',
       'Click Count: Day 18', 'Click Count: Day 19', 'Click Count: Day 20'],
    dtype='object')
```

Лишние столбцы удалены. Теперь в таблице остались только данные о цветах и количестве переходов. Наши 20 столбцов `Click Count` соответствуют числу переходов на каждые 100 ежедневных просмотров, значит, можно рассматривать их в процентном представлении. По сути, цвет в каждой строке отображает процент ежедневных переходов по объявлению. Далее мы обобщим процент ежедневных переходов для синих объявлений (листинг 9.8). Чтобы сгенерировать такую сводку, нужно проиндексировать каждую строку по цвету, после чего вызвать `df.T.Blue.describe()`.

Листинг 9.8. Обобщение ежедневной статистики по синему объявлению

```
df.set_index('Color', inplace=True)
print(df.T.Blue.describe())

count   20.000000
mean    28.350000
std      5.499043
min     18.000000
25%     25.750000
50%     27.500000
75%     30.250000
Max     42.000000
Name: Blue, dtype: float64
```

Ежедневный процент переходов по синему объявлению входит в диапазон от 18 до 42 %. Средний процент — 28,35 %, то есть в среднем при просмотре синих объявлений по ним переходят в 28,35 % случаев. Это довольно высокий средний показатель переходов. Но как он соотносится с остальными 29 цветами? Далее мы это выясним.

9.2. ВЫЧИСЛЕНИЕ P-ЗНАЧЕНИЙ ИЗ РАЗНИЦ МЕЖДУ СРЕДНИМИ ЗНАЧЕНИЯМИ

Начнем с отфильтровывания данных. Удалим синий, оставив остальные 29 цветов, а затем транспонируем таблицу для обращения к цветам по имени столбца (листинг 9.9).

Листинг 9.9. Создание таблицы без синего

```
df_not_blue = df.T.drop(columns='Blue')
print(df_not_blue.head(2))
```

```
Color               Pink  Gray  Sapphire  Purple  Coral  Olive  Navy  Maroon \
Click Count: Day 1    21    27        30      26     26     26    38      21
Click Count: Day 2    20    27        32      21     24     19    29      29

Color               Teal  Cyan    Ultramarine  Yellow  Viridian  Violet \
Click Count: Day 1    25    24             49      14        27      15
Click Count: Day 2    25    22             41      24        23      22

Color              Green  Aquamarine  Magenta  Silver  Bronze  Lime
Click Count: Day 1    14          24       18      26      19    20
Click Count: Day 2    25          28        21     24      19    19

[2 rows x 29 columns]
```

Таблица `df_not_blue` содержит процентное выражение переходов для 29 цветов. Нам нужно сравнить эти процентные показатели с процентным показателем синего объявления. Говоря точнее, нужно узнать, существует ли цвет, средний показатель переходов по которому статистически отличается от среднего показателя для синего. Как сравнить эти средние? Получить среднее по выборке для каждого цвета легко, но у нас нет среднего по совокупности. Значит, лучшим вариантом будет пермутационный тест. Для его выполнения нужно определить повторно используемую функцию, которая будет получать два массива NumPy и возвращать *p*-значение (листинг 9.10).

Листинг 9.10. Определение функции для пермутационного тестирования

```
def permutation_test(data_array_a, data_array_b):
    data_mean_a = data_array_a.mean()
    data_mean_b = data_array_b.mean()
    extreme_mean_diff = abs(data_mean_a - data_mean_b)
    total_data = np.hstack([data_array_a, data_array_b])
    number_extreme_values = 0.0
    for _ in range(30000):
        np.random.shuffle(total_data)
        sample_a = total_data[:data_array_a.size]
        sample_b = total_data[data_array_a.size:]
        if abs(sample_a.mean() - sample_b.mean()) >= extreme_mean_diff:
            number_extreme_values += 1

    p_value = number_extreme_values / 30000
    return p_value
```

Наблюдаемая разница между средними выборок

Разница между средними выборок очень большая

Далее мы выполним пермутационный тест между синим и 29 другими цветами, после чего упорядочим эти цвета на основе полученных *p*-значений (листинг 9.11). Вывод будет визуализирован в виде тепловой карты (рис. 9.1), чтобы лучше подчеркнуть разницу между *p*-значениями.

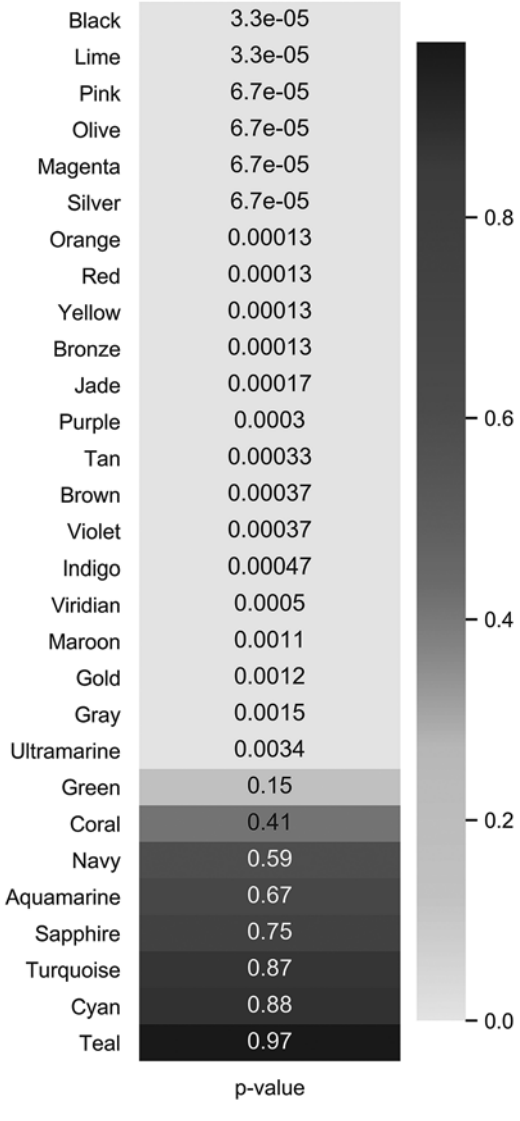


Рис. 9.1. Тепловая карта пар «p-значение/цвет», полученная в результате пермутационного теста: 21 цвет отображается в p-значение ниже 0,05

Листинг 9.11. Выполнение пермутационного теста между цветами

```
np.random.seed(0)
blue_clicks = df.T.Blue.values
color_to_p_value = {}
for color, color_clicks in df_not_blue.items():
    p_value = permutation_test(blue_clicks, color_clicks)
    color_to_p_value[color] = p_value
```

```
sorted_colors, sorted_p_values = zip(*sorted(color_to_p_value.items(),
                                             key=lambda x: x[1]))
plt.figure(figsize=(3, 10))
sns.heatmap([[p_value] for p_value in sorted_p_values],
            cmap='YlGnBu', annot=True, xticklabels=['p-value'],
            yticklabels=sorted_colors)
plt.show()
```

Корректирует ширину и высоту тепловой карты на 3 и 10 дюймов соответственно. Это повышает качество визуализации

Метод sns.heatmap получает на входе 2D-таблицу. Таким образом, мы преобразуем одномерный список p-значений в 2D-таблицу, содержащую 29 строк и 1 столбец

Эффективный код Python для упорядочивания словаря и возвращения двух списков: упорядоченных значений и связанных с ними ключей. Каждое упорядоченное p-значение в позиции i выравнивается с цветом в sorted_colors[i]

Большинство цветов генерируют *p*-значение, которое заметно меньше 0,05. Минимальная его величина соответствует черному цвету — процент перехода по этим объявлениям существенно отклоняется от процента переходов по синим. Но с точки зрения дизайна черный не является особо кликабельным цветом. Текстовые ссылки редко бывают черными, поскольку тогда их будет трудно отличить от обычного текста. Получается подозрительная ситуация: «Какова конкретно разница между зафиксированным числом переходов по черным и синим объявлениям?» Это можно проверить, выполнив `df_not_blue.Black.mean()` (листинг 9.12).

Листинг 9.12. Определение среднего количества переходов по черным объявлениям

```
mean_black = df_not_blue.Black.mean()
print(f"Mean click-rate of black is {mean_black}")

Mean click-rate of black is 21.6
```

Средний показатель переходов по черному объявлению равен 21,6. Это значение существенно ниже среднего для синего объявления, которое равно 28,35. Значит, статистическая разница между этими цветами вызвана тем, что по черному переходят меньше людей. Возможно, прочие *p*-значения также вызваны более низкими показателями переходов. Отфильтруем цвета, чье среднее будет меньше среднего для синих объявлений, после чего выведем оставшиеся (листинг 9.13).

Листинг 9.13. Отфильтровывание цветов с меньшим процентом переходов

Эффективный однострочный код для отфильтровывания цветов. Сначала создается логический массив, указывающий, какие цвета содержат среднее значение, которое больше среднего для синего. Этот массив передается в df для фильтрации. Индексы отфильтрованных результатов указывают на названия оставшихся цветов

```
remaining_colors = df[df.T.mean().values > blue_clicks.mean()].index
size = remaining_colors.size
print(f"{size} colors have on average more clicks than Blue.")
print("These colors are:")
print(remaining_colors.values)
```

```
5 colors have on average more clicks than Blue.
These colors are:
['Sapphire' 'Navy' 'Teal' 'Ultramarine' 'Aquamarine']
```

Остается всего пять цветов. Каждый из них — это некий оттенок синего. Выведем упорядоченные *p*-значения этих цветов, а также среднее число переходов для упрощения анализа (листинг 9.14).

Листинг 9.14. Вывод пяти оставшихся цветов

```
for color, p_value in sorted(color_to_p_value.items(), key=lambda x: x[1]):
    if color in remaining_colors:
        mean = df_not_blue[color].mean()
        print(f"{color} has a p-value of {p_value} and a mean of {mean}")
```

```
Ultramarine has a p-value of 0.0034 and a mean of 34.2
Navy has a p-value of 0.5911666666666666 and a mean of 29.3
Aquamarine has a p-value of 0.6654666666666667 and a mean of 29.2
Sapphire has a p-value of 0.7457666666666667 and a mean of 28.9
Teal has a p-value of 0.9745 and a mean of 28.45
```

9.3. ОПРЕДЕЛЕНИЕ СТАТИСТИЧЕСКОЙ ЗНАЧИМОСТИ

Для четырех цветов *p*-значения получились большими, малым оно оказалось всего у одного — ультрамарина, являющегося особым оттенком синего. Его среднее равно 34,2, то есть превосходит среднее синего, равное 28,35. *P*-значение ультрамарина равно 0,0034. Имеет ли такое значение статистическую значимость? Что ж, оно более чем в десять раз меньше стандартного уровня значимости 0,05. Однако этот уровень не учитывает нашего сравнения синего с 29 другими цветами. Каждое такое сравнение является экспериментом, определяющим, насколько эффективность того или иного цвета отличается от эффективности синего. Если выполнить достаточное число таких экспериментов, то рано или поздно мы гарантированно получим низкое *p*-значение. Скорректировать это лучше всего поправкой Бонферрони — в противном случае мы попадем в ситуацию выуживания данных. Для применения поправки снижаем уровень значимости до 0,05/29 (листинг 9.15).

Листинг 9.15. Применение поправки Бонферрони

```
significance_level = 0.05 / 29
print(f"Adjusted significance level is {significance_level}")
if color_to_p_value['Ultramarine'] <= significance_level:
    print("Our p-value is statistically significant")
```

```
else:
    print("Our p-value is not statistically significant")

Adjusted significance level is 0.001724137931034483
Our p-value is not statistically significant
```

В итоге наше *p*-значение не оказалось статистически значимым — Фред провел слишком много экспериментов для формирования весомого заключения. На деле же не все эти эксперименты были необходимы. Нет никакой объективной причины ожидать, что черный, коричневый либо серый покажут бóльшую эффективность, чем синий. Возможно, если бы Фред исключил некоторые цвета, анализ оказался бы более плодотворным. Чисто гипотетически, если бы Фред просто сравнил синий с пятью другими вариантами этого цвета, мы могли бы получить статистически значимый результат. Давайте разберем такую гипотетическую ситуацию, в которой Фред проводит пять экспериментов и *p*-значение ультрамарина остается тем же, что и прежде (листинг 9.16).

Листинг 9.16. Изучение гипотетического уровня значимости

```
hypothetical_sig_level = 0.05 / 5
print(f"Hypothetical significance level is {hypothetical_sig_level}")
if color_to_p_value['Ultramarine'] <= hypothetical_sig_level:
    print("Our hypothetical p-value would have been statistically significant")
else:
    print("Our hypothetical p-value would not have been statistically significant")

Hypothetical significance level is 0.01
Our hypothetical p-value would have been statistically significant
```

При таких теоретических условиях результаты будут статистически значимыми. К сожалению, нельзя использовать гипотетические условия для снижения уровня значимости. Нет никакой гарантии, что повторное выполнение экспериментов воспроизведет *p*-значение 0,0034. *P*-значения колеблются, а излишние эксперименты повышают вероятность возникновения недостоверных колебаний. Рассматривая большое количество проведенных Фредом экспериментов, мы просто не можем сделать статистически значимое заключение.

Однако еще не все потеряно. Ультрамарин по-прежнему представляет собой обнадеживающую замену для синего. Стоит ли Фреду попробовать реализовать эту замену? Возможно. Мы рассмотрим два альтернативных сценария. В первом нулевая гипотеза будет верна. Если так и окажется, тогда и синий, и ультрамарин имеют общее среднее по совокупности. В таких условиях замена синего на ультрамарин не повлияет на показатель числа переходов. Во втором же сценарии более высокий показатель переходов для ультрамарина является статистически значимым. Если так и есть, тогда перекрашивание синих объявлений в ультрамарин приведет к повышению частоты переходов. Следовательно, Фред, ничем

не рискуя, может повысить эффективность своего объявления, перекрасив его в ультрамарин.

С логической точки зрения Фред определенно должен это сделать, но в таком случае некоторая неуверенность все равно сохранится. Фред так и не узнает, действительно ли ультрамарин обеспечивает больше переходов, чем синий. Что, если любопытство Фреда возьмет верх? Если он действительно желает знать ответ, то единственный выход — провести еще один эксперимент. В нем половина показываемых объявлений будут синими, а вторая половина — ультрамариновыми. Программное обеспечение Фреда будет демонстрировать эти два вида объявлений, регистрируя просмотры и переходы по ним. После этого можно будет еще раз вычислить *p*-значение и сравнить его с подходящим уровнем значимости, который по-прежнему будет равен 0,05. Поправка Бонферрони здесь не пригодится, так как мы проведем всего один эксперимент. После сравнения *p*-значений Фред наконец-то узнает, действительно ли ультрамарин эффективнее синего.

9.4. ПОУЧИТЕЛЬНАЯ ИСТОРИЯ ИЗ РЕАЛЬНОЙ ЖИЗНИ: 41 ОТТЕНОК СИНЕГО

Фред предположил, что анализ каждого цвета даст более основательные результаты, но ошибся. Больше данных не всегда означает более точный результат. Иногда их чрезмерное количество лишь вносит неуверенность.

Фред не статистик, и его можно простить за то, что он не предугадал последствия излишне подробного анализа. Но этого нельзя сказать о конкретных экспертах, работающих в современном бизнесе. Возьмем, к примеру, нашумевший случай, произошедший в хорошо известной корпорации. В ней нужно было выбрать цвет для веб-ссылок на сайте. Старший дизайнер подобрал визуально привлекательный оттенок синего, но один из директоров отклонил этот выбор. Почему дизайнер выбрал именно этот оттенок синего, а не другой?

Директор был сторонником количественного анализа и настаивал на том, чтобы цвет ссылок был выбран научным путем посредством обширных аналитических тестов, которые должны определить идеальный оттенок синего. В результате абсолютно случайным образом веб-ссылки компании оформили в 41 оттенок синего, после чего зафиксировали миллионы переходов по ним. В конечном итоге «оптимальный» оттенок был выбран на основе максимального количества переходов относительно просмотров.

Затем директор обнародовал выбранный им метод анализа, на что статистики по всему миру лишь усмехнулись. Навязанное им решение показало незнание директором статистических основ, что бросило тень как на него самого, так и на его компанию.

РЕЗЮМЕ

- Больше данных — не всегда лучше. Выполнение бессмысленных лишних аналитических тестов повышает вероятность получения аномального результата.
- Не стоит жалеть времени на тщательное обдумывание задачи, прежде чем переходить к анализу. Если бы Фред лучше подумал над 31 цветом, то понял бы, что тестировать их все смысла нет. Многие цвета сильно портят вид ссылок. Такие цвета, как черный, вообще вряд ли обеспечат больше переходов, чем синий. Предварительная очистка набора цветов привела бы к более информативному результату.
- Несмотря на то что эксперимент Фреда провалился, мы все равно смогли извлечь из него полезную информацию. Ультрамарин может оказаться разумной заменой синему, но здесь необходимы дополнительные тесты. Иногда аналитики получают для работы неполноценные данные, но даже из них можно извлечь полезные выводы.

Практическое задание 3

Отслеживание вспышек заболеваний по новостным заголовкам

УСЛОВИЕ ЗАДАЧИ

Поздравляю! Вы только что наняты Департаментом здравоохранения США. Он мониторит эпидемии заболеваний внутри страны и за ее пределами. Важнейшим компонентом мониторинга является анализ новых публикуемых данных. Каждый день департамент получает сотни заголовков новостей, описывающих вспышки заболеваний в различных точках планеты.

Но этих новостных заголовков слишком много для ручного анализа, поэтому перед вами ставят задачу: обрабатывать ежедневные поступления новостей, извлекая упомянутые в них места вспышек заболеваний. После этого нужно будет сгруппировать заголовки на основе географического распределения. В завершение останется просмотреть самые большие группы внутри и вне США и доложить всю интересную информацию руководству.

ОПИСАНИЕ НАБОРА ДАННЫХ

Файл `headlines.txt` содержит сотни заголовков, которые нужно проанализировать. Каждый заголовок располагается на отдельной строке.

ОБЗОР

Для решения поставленной задачи нам нужно уметь:

- группировать наборы данных при помощи нескольких техник и измерения расстояния;
- измерять расстояния между локациями на сферическом глобусе;
- визуализировать локации на карте;
- извлекать координаты локаций из текста заголовков.

10 Кластеризация данных по группам

В этой главе

- ✓ Кластеризация данных по центральности.
- ✓ Кластеризация данных по плотности.
- ✓ Компромиссы между алгоритмами кластеризации.
- ✓ Выполнение кластеризации с помощью библиотеки scikit-learn.
- ✓ Перебор кластеров с помощью Pandas.

Кластеризация — это процесс организации точек данных в осмысленные группы. Что делает группу осмысленной? Простого ответа на этот вопрос нет. Полезность любого кластеризованного вывода зависит от поставленной задачи.

Представим, что нам нужно кластеризовать коллекцию фотографий домашних животных. Поместим ли мы рыб и ящериц в одну группу, а зверей, покрытых мехом (хомячков, кошек и собак), — в другую? А может, хомяков, кошек и собак можно выделить в собственные группы? Если да, то нам, пожалуй, нужно рассматривать кластеризацию животных по породе. Тогда чихуахуа и немецкий дог попадут в разные группы. Проведение различия между породами собак вызовет сложности. Однако можно без проблем отличить чихуахуа от дога на основе размера породы. Возможно, стоит пойти на компромисс — группировать животных по наличию меха и размеру, обходя таким образом различие между керн-терьером и похожим на него норвич-терьером.

Оправдан ли такой компромисс? Все зависит от конкретной задачи аналитики. Представим, что мы работаем на производителя кормов для животных и наша цель — оценить востребованность кормов: собачьего, кошачьего и предназначенного для ящериц. При таких условиях необходимо провести различие между пушистыми собаками, пушистыми кошками и чешуйчатыми ящерицами. Однако нам не потребуется выискивать различия между породами собак. А представьте себе аналитика в ветеринарном офисе, который пытается сгруппировать животных-пациентов по породе. Такая задача потребует намного более детального подхода к группировке.

Различные ситуации требуют различных техник кластеризации. Будучи аналитиками данных, мы должны выбирать из них наиболее подходящее решение. Во время работы мы кластеризуем тысячи, если не десятки тысяч наборов данных, используя всевозможные техники кластеризации. Наиболее распространенные алгоритмы опираются на некоторое определение центральности, относительно которой и происходит деление на кластеры.

10.1. ВЫДЕЛЕНИЕ КЛАСТЕРОВ НА ОСНОВЕ ЦЕНТРАЛЬНОСТИ

В главе 5 мы узнали, как центральность данных можно выразить с помощью среднего. Позднее, в главе 7, вычислили среднюю длину одной группы рыб. В итоге мы сравнили два отдельных набора рыб, проанализировав различия между их средними. На основе чего определили, принадлежит ли рыба к одной и той же общей совокупности. Чисто интуитивно все точки данных в одной группе должны кластеризоваться вокруг одного центрального значения. При этом измерения в двух разноплановых группах должны группироваться вокруг двух разных средних. Получается, что можно использовать центральность для проведения различий между двумя разноплановыми группами.

Предположим, что мы отправились в оживленный местный паб, где видим две висящие рядом мишени для дартса. Эти мишени, а также окружающие их стены, усыпаны дротиками. Хмельные игроки стремятся попасть в яблочко — в центр одной из мишеней. Зачастую они промахиваются, в результате чего образуются отметки от дротиков, сконцентрированные вокруг центров мишеней.

Давайте просимулируем разброс отметок численно. Мы будем рассматривать каждое расположение центра как двухмерные координаты. Дротики случайным образом запускаются в эти координаты. В результате двухмерные позиции дротиков распределяются случайным образом. Наиболее подходящим для моделирования этих позиций будет нормальное распределение, и на то есть причины.

- Типичный метатель дротиков целится в центр, а не в край мишени. Значит, каждый дротик, вероятнее всего, попадет близко к ее центру. Это поведение

согласуется со случайными выборками из нормального распределения, в котором значения, более приближенные к среднему, встречаются чаще других, более удаленных от него.

- Мы ожидаем, что дротики будут попадать в мишень симметрично относительно центра, то есть станут втыкаться на три дюйма левее и на три дюйма правее центра с равной частотой. Эта симметрия отражается нормальной кривой в форме колокола.

Предположим, что первый центр расположен в координате [0, 0]. Дротик запускается по этим координатам. Мы смоделируем позиции x и y дротика, используя два нормальных распределения. Эти распределения имеют общее среднее 0, и мы дополнительно предположим, что также они имеют общую дисперсию, равную 2. Код листинга 10.1 генерирует случайные координаты дротика.

Листинг 10.1. Моделирование координат дротиков с помощью двух нормальных распределений

```
import numpy as np
np.random.seed(0)
mean = 0
variance = 2
x = np.random.normal(mean, variance ** 0.5)
y = np.random.normal(mean, variance ** 0.5)
print(f"The x coordinate of a randomly thrown dart is {x:.2f}")
print(f"The y coordinate of a randomly thrown dart is {y:.2f}")

The x coordinate of a randomly thrown dart is 2.49
The y coordinate of a randomly thrown dart is 0.57
```

ПРИМЕЧАНИЕ

Координаты дротиков можно моделировать более эффективно с помощью метода np.random.multivariate_normal. Он выбирает одну случайную точку в многомерном нормальном распределении. Многомерная нормальная кривая — это простая нормальная кривая, которая расширена на дополнительные измерения. Наше двухмерное нормальное распределение будет походить на округлый холм с вершиной, расположенной в точке [0, 0].

Далее мы симулируем 5000 случайных бросков дротиков в центр, находящийся в позиции [0, 0], а также 5000 случайных бросков во второй центр, расположенный в позиции [0, 6] (листинг 10.2). После этого сгенерируем точечный график всех случайных координат дротиков (рис. 10.1).

Листинг 10.2. Симуляция случайных бросков дротиков

```
import matplotlib.pyplot as plt
np.random.seed(1)
bulls_eye1 = [0, 0]
bulls_eye2 = [6, 0]
bulls_eyes = [bulls_eye1, bulls_eye2]
```

```
x_coordinates, y_coordinates = [], []
for bulls_eye in bulls_eyes:
    for _ in range(5000):
        x = np.random.normal(bulls_eye[0], variance ** 0.5)
        y = np.random.normal(bulls_eye[1], variance ** 0.5)
        x_coordinates.append(x)
        y_coordinates.append(y)

plt.scatter(x_coordinates, y_coordinates)
plt.show()
```

ПРИМЕЧАНИЕ

Листинг 10.2 включает вложенный пятистрочный цикл for, начинающийся с for_in range(5000). С помощью NumPy можно реализовать этот цикл всего в одной строке кода: выполнение x_coordinates, y_coordinates = np.random.multivariate_normal(bulls_eye, np.diag(2 * [variance]), 5000).T возвращает 5000 координат x и y, полученных из многомерного нормального распределения.

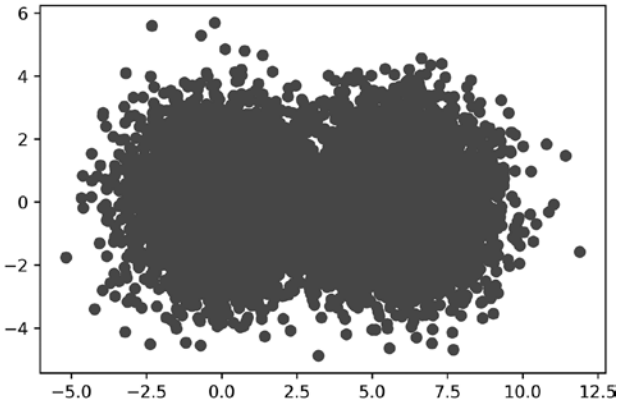


Рис. 10.1. Симуляция дротиков, произвольно разбросанных вокруг центров мишеней

На графике видны две накладывающиеся друг на друга группы отметок дротиков, представляющие 10 000 бросков. Половина дротиков были нацелены в центр слева, а вторая половина — в центр справа.

У каждого дротика была намечена цель, которую можно определить, посмотрев на график. Дротики, находящиеся ближе к [0, 0], скорее всего, были брошены в центр слева. Это предположение мы отобразим на графике. Для этого припишем каждый дротик ближайшему к нему центру, начав с определения функции `nearest_bulls_eye`, получающей список `dart` с координатами *x* и *y* дротиков. Она будет возвращать индекс центра, наиболее приближенного к `dart`. Близость дротика будем измерять с помощью *евклидова расстояния*, которое является стандартным прямым расстоянием между двумя точками.

ПРИМЕЧАНИЕ

Евклидово расстояние вычисляется с использованием теоремы Пифагора. Предположим, что оцениваем дротик в позиции [x_dart, y_dart] относительно центра в позиции [x_bull, y_bull]. Согласно теореме Пифагора distance2 = (x_dart – x_bull)2 + (y_dart – y_bull)2. Вычисляется расстояние с помощью собственной евклидовой функции. В качестве альтернативы можно взять функцию scipy.spatial.distance.euclidean, предоставляемую SciPy.

Код листинга 10.3 определяет `nearest_bulls_eye` и применяет ее к дротикам [0, 1] и [6, 1].

Листинг 10.3. Приписывание дротиков ближайшему центру

```
from scipy.spatial.distance import euclidean
def nearest_bulls_eye(dart):
    distances = [euclidean(dart, bulls_e) for bulls_e in bulls_eyes]
    return np.argmin(distances)

darts = [[0,1], [6, 1]]
for dart in darts:
    index = nearest_bulls_eye(dart)
    print(f"The dart at position {dart} is closest to bulls-eye {index}")

The dart at position [0, 1] is closest to bulls-eye 0
The dart at position [6, 1] is closest to bulls-eye 1
```

Получает евклидово расстояние между дротиком и каждым центром, используя евклидову функцию, импортированную из SciPy

Возвращает индекс, соответствующий кратчайшему расстоянию до центра в массиве

Теперь применим `nearest_bulls_eye` ко всем вычисленным координатам дротиков. Каждая точка от дротика отображается на графике одним из двух цветов, чтобы можно было различить принадлежащие двум разным центрам (листинг 10.4; рис. 10.2).

Листинг 10.4. Раскрашивание дротиков согласно ближайшему центру

```
def color_by_cluster(darts):
    nearest_bulls_eyes = [nearest_bulls_eye(dart) for dart in darts]
    for bs_index in range(len(bulls_eyes)):
        selected_darts = [darts[i] for i in range(len(darts))
                          if bs_index == nearest_bulls_eyes[i]]
        x_coordinates, y_coordinates = np.array(selected_darts).T
        plt.scatter(x_coordinates, y_coordinates,
                    color=['g', 'k'][bs_index])
    plt.show()

darts = [[x_coordinates[i], y_coordinates[i]]
         for i in range(len(x_coordinates))]
color_by_cluster(darts)
```

Вспомогательная функция, отрисовывающая цветные элементы входного списка дротиков. Каждый дротик в darts подается на вход в nearest_bulls_eye

Выбирает дротики, более близкие к bulls_eyes[bs_index]

Разделяет координаты x и y каждого дротика, транспонируя массив выбранных дротиков. Как говорилось в главе 8, транспонирование меняет местами позиции строк и столбцов 2D-структуры данных

Объединяет отдельные координаты каждого дротика в один список координат x и y

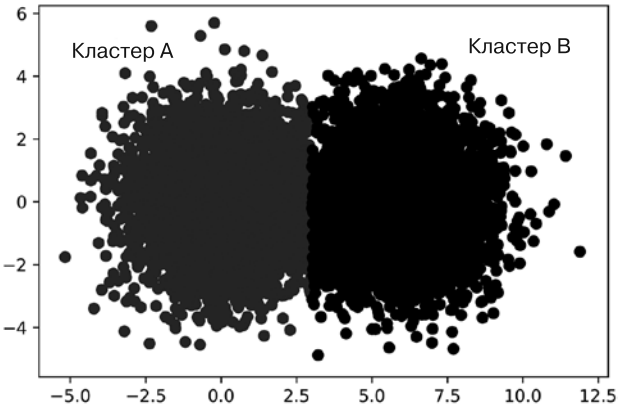


Рис. 10.2. Дротики, окрашенные согласно приближенности к одному из двух центров. Кластер A представляет все точки, находящиеся ближе к левому центру, а кластер B — все точки, расположенные ближе к правому

Окрашенные дротики логично делятся на два равных кластера. А как определять подобные кластеры при отсутствии центральных координат? Довольно примитивный вариант — просто предположить, какими могут быть позиции центров. Можно выбрать два случайных дротика и надеяться, что они окажутся довольно близко к каждому из центров, хотя вероятность этого очень мала (листинг 10.5). В большинстве случаев окрашивание дротиков на основе двух случайно выбранных центров не даст хороших результатов (рис. 10.3).

Листинг 10.5. Приписывание дротиков случайно выбранным центрам

```
bulls_eyes = np.array(darts[:2])
color_by_cluster(darts)
```

Случайно выбирает первые два дротика в качестве центров

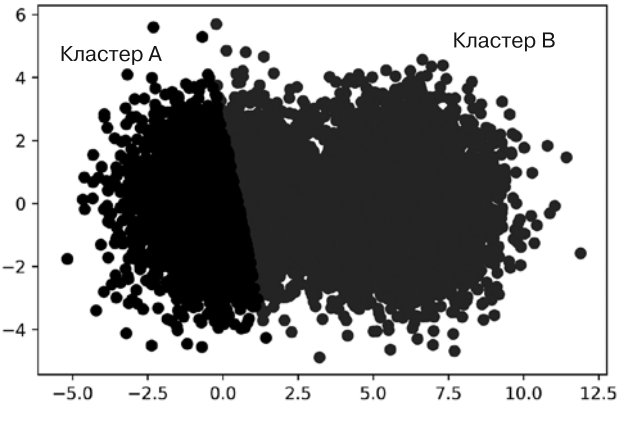


Рис. 10.3. Дротики, окрашенные на основе приближенности к случайно выбранным центрам. Кластер B слишком вытянулся влево

Выбор центров наугад привел к количественной ошибке. Говоря конкретнее, кластер B слишком вытянулся влево. Выбранные нами произвольные центры не соответствуют фактическим. Однако есть способ исправить ситуацию — можно вычислить средние координаты всех точек в растянутой правой группе, а затем использовать их для корректировки ее предположительного центра (листинг 10.6). После присваивания средних координат кластера его центру можно повторно применить нашу основанную на расстоянии технику для корректировки его границ. По факту же для наибольшей эффективности, прежде чем выполнять кластеризацию относительно центров, мы сначала аналогичным образом скорректируем центр левого кластера (рис. 10.4).

ПРИМЕЧАНИЕ

При вычислении среднего одномерного массива возвращаем одно значение. Теперь мы расширяем это определение, включая несколько измерений. При вычислении среднего двухмерного массива возвращаются средние всех координат x и y. Итоговым выводом будет 2D-массив, содержащий средние для осей X и Y.

Листинг 10.6. Присваивание дротиков центрам на основе средних значений

```
def update_bulls_eyes(darts):
    updated_bulls_eyes = []
    nearest_bulls_eyes = [nearest_bulls_eye(dart) for dart in darts]
    for bs_index in range(len(bulls_eyes)):
        selected_darts = [darts[i] for i in range(len(darts))
                          if bs_index == nearest_bulls_eyes[i]]
        x_coordinates, y_coordinates = np.array(selected_darts).T
        mean_center = [np.mean(x_coordinates), np.mean(y_coordinates)]
        updated_bulls_eyes.append(mean_center)

return updated_bulls_eyes

bulls_eyes = update_bulls_eyes(darts)
color_by_cluster(darts)
```

Получает среднее по координатам x и y для всех дротиков, отнесенных к заданному центру. Затем средние координаты используются для обновления предполагаемой позиции центра. Это вычисление можно произвести более эффективно, выполнив np.mean(selected_darts, axis=0)

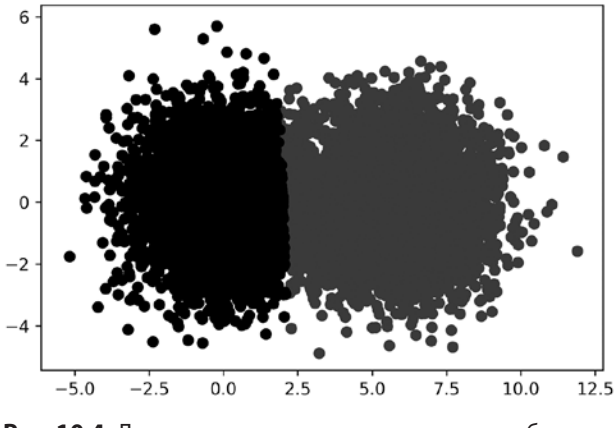


Рис. 10.4. Дротики, окрашенные на основе приближенности к повторно вычисленным центрам. Теперь два кластера выглядят более равномерно

Результат выглядит более удачным, хотя можно получить еще бóльшую эффективность. Центры кластеров по-прежнему слегка смещены. Давайте исправим полученный результат, повторив корректировку центральности на основе среднего значения еще десять раз (листинг 10.7; рис. 10.5).

Листинг 10.7. Корректировка позиций центров в ходе десяти итераций

```
for i in range(10):
    bulls_eyes = update_bulls_eyes(darts)

color_by_cluster(darts)
```

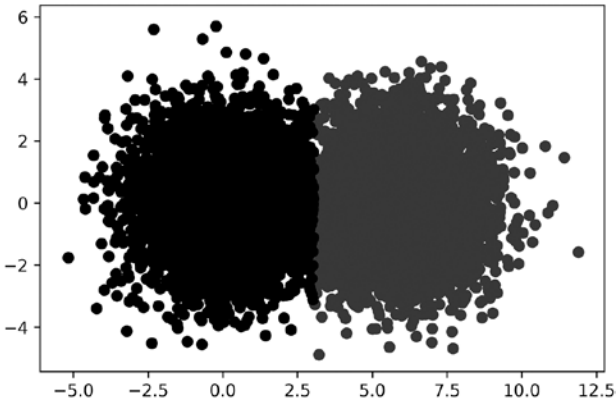


Рис. 10.5. Дротики, окрашенные на основе приближенности к итеративно вычисленным центрам

Теперь два набора дротиков прекрасно сгруппированы. По сути, мы воссоздали алгоритм *K*-средних, который организует данные при помощи центральности.

10.2. K-СРЕДНИЕ: АЛГОРИТМ КЛАСТЕРИЗАЦИИ ДЛЯ ГРУППИРОВКИ ДАННЫХ ПО K ЦЕНТРАЛЬНЫХ ГРУПП

Алгоритм *K*-средних предполагает, что входные точки данных сосредотачиваются вокруг *K* разных центров. Координаты каждого центра подобны скрытому яблочку, окруженному разбросанными точками данных. Задача данного алгоритма — выявить эти скрытые центральные координаты.

Мы инициализируем *K*-средние, начав с выбора *K*, представляющего количество искомых центральных координат. При анализе мишени *K* равнялось 2, хотя по факту может равняться любому целому числу. Алгоритм случайно выбирает *K* точек данных, которые начинают рассматриваться как истинные центры. После

этого алгоритм итеративно обновляет выбранные центральные точки, которые аналитики называют *центрами масс*. В рамках одной итерации каждая точка данных приписывается к ближайшему центру, в результате чего формируются K групп. Затем центр каждой группы обновляется. При этом он оказывается равен среднему координат его группы. Если этот процесс повторять достаточно долго, то средние группы сойдутся к K репрезентативным центрам (рис. 10.6). Такое схождение гарантируется математически, однако необходимое для этого количество итераций заранее узнать нельзя. Общепринятый прием заключается в том, чтобы прекращать повторения, когда ни один из вновь вычисленных центров не отклоняется значительно от своего предшественника.

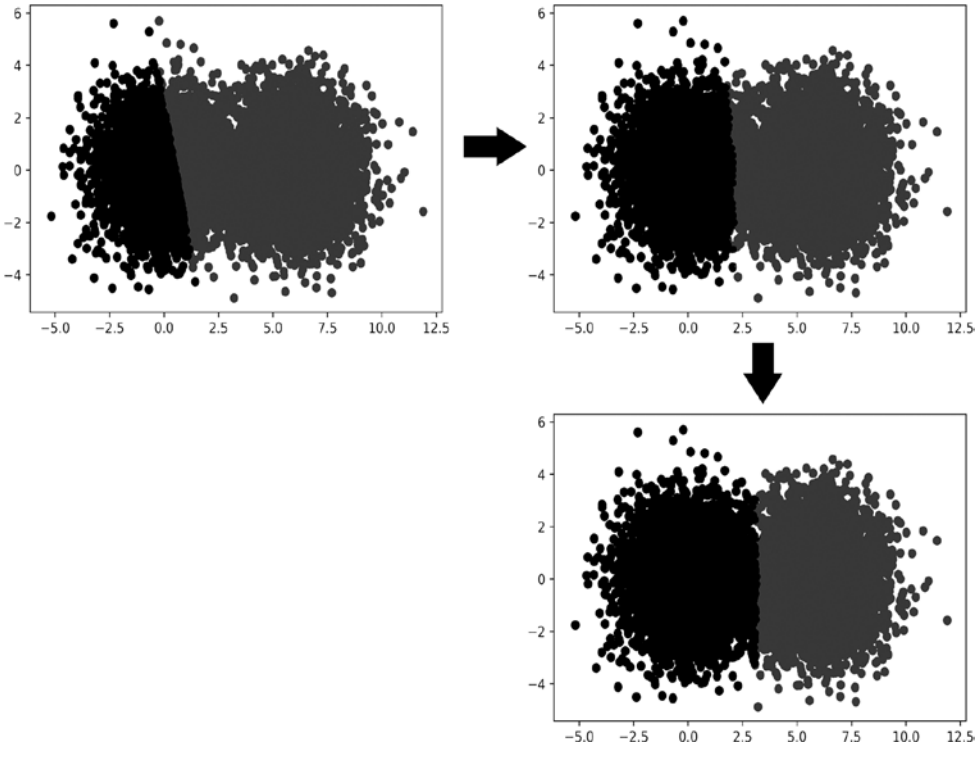


Рис. 10.6. Алгоритм K-средних итеративно сходится из двух случайно выбранных центров масс к фактическим центрам

Метод K-средних имеет и ограничения. Этот алгоритм основывается на нашем знании K, то есть количества искомых кластеров. Зачастую подобная информация оказывается недоступной. К тому же, хотя этот метод обычно и находит разумные центры, он не дает математической гарантии, что они окажутся наилучшими для рассматриваемых данных. Временами он возвращает нелогичные или неоптимальные группы из-за неудачного выбора случайных центров масс на этапе инициализации. Наконец, этот метод предполагает, что кластеры данных фактически

сосредоточены вокруг *K* центральных точек, хотя чуть позже мы узнаем, что это предположение не всегда верно.

10.2.1. Кластеризация по методу K-средних с помощью scikit-learn

При удачной реализации алгоритм *K*-средних выполняется за вполне приемлемое время. Подобную быстродействующую реализацию может обеспечить внешняя библиотека scikit-learn. Она представляет собой невероятно популярный инструмент машинного обучения, построенный на базе NumPy и SciPy. Эта библиотека несет в себе различные фундаментальные алгоритмы классификации, регрессии и кластеризации, включая, конечно же, метод *K*-средних. Для начала мы ее установим (листинг 10.8), после чего импортируем необходимый для кластеризации класс `KMeans`.

ПРИМЕЧАНИЕ

Для установки scikit-learn выполните из терминала pip install scikit-learn.

Листинг 10.8. Импорт KMeans из scikit-learn

```
from sklearn.cluster import KMeans
```

Применить KMeans к нашим `darts` будет просто (листинг 10.9). Сначала нужно выполнить `KMeans(n_clusters=2)`, что приведет к созданию объекта `cluster_model`, способного найти два центра. Затем можно будет запустить сам алгоритм вызовом `cluster_model.fit_predict(darts)`. Этот метод вернет массив `assigned_bulls_eyes`, хранящий индекс центра каждой мишени.

Листинг 10.9. Кластеризация методом K-средних с помощью scikit-learn

```
cluster_model = KMeans(n_clusters=2)
assigned_bulls_eyes = cluster_model.fit_predict(darts)

print("Bull's-eye assignments:")
print(assigned_bulls_eyes)

Bull's-eye assignments:
[0 0 0 ... 1 1 1]
```

Создает объект cluster_model, в котором количество центров установлено на 2

Оптимизирует два центра с помощью метода K-средних и возвращает для каждого дротика соответствующий кластер

Теперь для проверки результатов раскрасим дротики согласно принадлежности к тому или иному кластеру (листинг 10.10; рис. 10.7).

Листинг 10.10. Раскрашивание точек данных на основе принадлежности к кластеру

```
for bs_index in range(len(bulls_eyes)):
    selected_darts = [darts[i] for i in range(len(darts))
                      if bs_index == assigned_bulls_eyes[i]]
    x_coordinates, y_coordinates = np.array(selected_darts).T
    plt.scatter(x_coordinates, y_coordinates,
                color=['g', 'k'][bs_index])
plt.show()
```

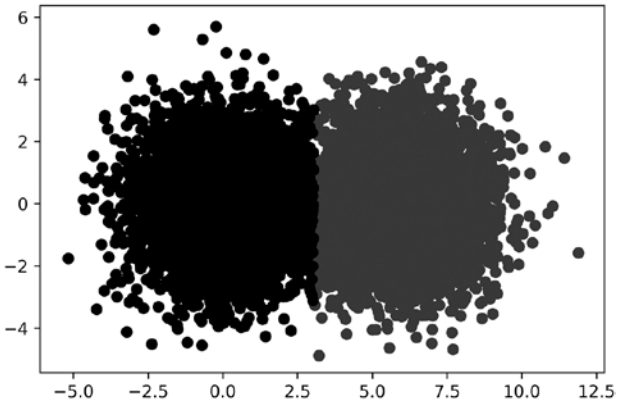


Рис. 10.7. Результаты метода K-средних, возвращенные scikit-learn, соответствуют ожиданиям

Наша модель кластеризации обнаружила в данных центры масс. Теперь эти центры масс можно использовать для анализа новых точек данных, которые модель еще не видела. Выполняя `cluster_model.predict([x, y])`, мы присваиваем центр масс точке данных, определяемой координатами x и y (листинг 10.11). Для кластеризации двух новых точек данных применяется метод `predict`.

Листинг 10.11. Кластеризация новых данных при помощи cluster_model

```
new_darts = [[500, 500], [-500, -500]]
new_bulls_eye_assignments = cluster_model.predict(new_darts)
for i, dart in enumerate(new_darts):
    bulls_eye_index = new_bulls_eye_assignments[i]
    print(f"Dart at {dart} is closest to bull's-eye {bulls_eye_index}")

Dart at [500, 500] is closest to bull's-eye 0
Dart at [-500, -500] is closest to bull's-eye 1
```

10.2.2. Выбор оптимального K методом локтя

Алгоритм K-средних опирается на входное значение K, что может стать серьезной помехой, когда количество подлинных кластеров в данных заранее неизвестно. Но мы все же можем подобрать подходящее значение K, используя технику под названием *«метод локтя»*.

Если $K = 1$, тогда инерция будет равна сумме всех возведенных в квадрат расстояний до среднего набора данных. Это значение, как говорилось в главе 5, прямо пропорционально дисперсии. В свою очередь, дисперсия является мерой рассеяния. Таким образом, если $K = 1$, инерция является оценочным показателем рассеяния. Это свойство работает, даже если $K > 1$. По сути, инерция приблизительно показывает рассеяние вокруг K вычисленных средних.

Оценивая рассеяние, можно определить, является ли наше значение *K* излишне большим или малым. Представим, что установили его равным 1. Не исключено, что многие точки данных окажутся размещены очень далеко от одного центра. Рассеяние и инерция получатся большими. По мере увеличения *K* в направлении более разумного числа возникающие дополнительные центры приведут к уменьшению инерции. В конечном итоге, если излишне увлечься и установить *K* равным общему количеству точек, каждая из них попадет в собственный кластер. При этом рассеяние полностью исчезнет, и инерция окажется равна нулю (рис. 10.8).

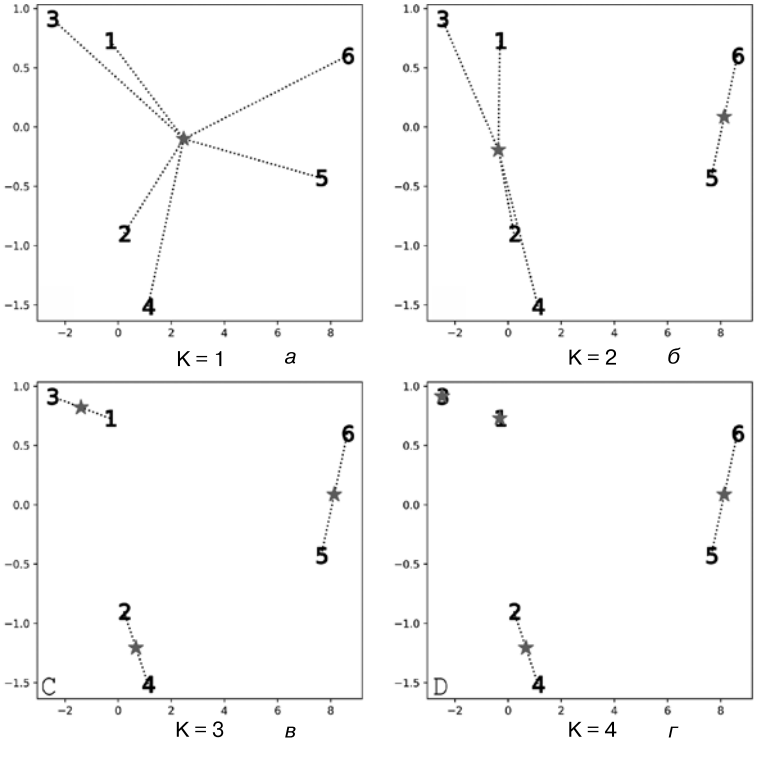


Рис. 10.8. Шесть точек с номерами от 1 до 6 отрисованы в двухмерном пространстве. Их центры, обозначенные звездочками, вычислены для разных значений K. От каждой точки к ближайшему центру проведена линия. Инерция вычисляется суммированием возведенных в квадрат длин шести линий: а — K = 1, все шесть линий исходят от одного центра, инерция в данном случае довольно велика; б — K = 2, точки 5 и 6 очень близки ко второму центру, инерция сократилась; в — K = 3, точки 1 и 3 значительно ближе к новому центру, точки 2 и 4 также существенно приблизились к новому центру, инерция сильно уменьшилась; г — K = 4, теперь точки 1 и 3 накладываются на свои центры, их вклад в инерцию изменился с очень низкого значения на нулевое, расстояние между оставшимися четырьмя точками и связанными с ними центрами остается неизменным. Таким образом, увеличение K с 3 до 4 привело к незначительному уменьшению инерции

Одни значения инерции излишне велики, другие, наоборот, слишком малы, но где-то между этими крайностями находится то самое верное значение. Как же его отыскать?

Далее мы проработаем решение этого вопроса, начав с графического отображения инерции набора данных с мишеней для большого диапазона значений *K* (листинг 10.12; рис. 10.9). Инерция автоматически вычисляется для каждого scikit-learn-объекта KMeans. Обратиться к этому сохраненному значению можно через атрибут модели `inertia_`.

Листинг 10.12. Построение графика инерции для K-средних

```
k_values = range(1, 10)
inertia_values = [KMeans(k).fit(darts).inertia_
                  for k in k_values]

plt.plot(k_values, inertia_values)
plt.xlabel('K')
plt.ylabel('Inertia')
plt.show()
```

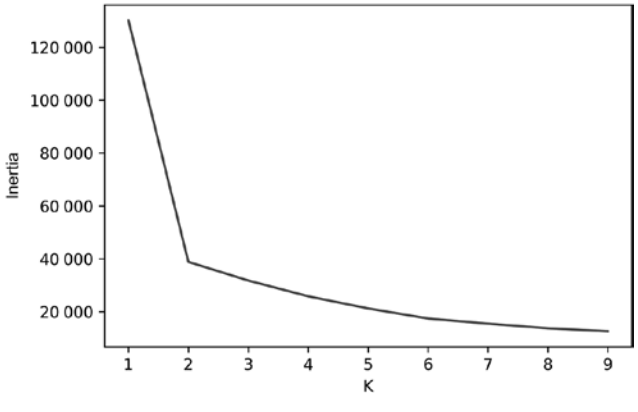


Рис. 10.9. График инерции для симуляции мишеней, содержащих два центра

Сгенерированный график напоминает руку, согнутую в локте, который указывает на значение *K*, равное 2. Как нам уже известно, это *K* точно охватывает два центра, которые мы предварительно заложили в набор данных.

Продолжит ли этот подход работать, если количество представленных центров увеличить? Это можно выяснить, добавив в симуляцию еще один. После увеличения числа кластеров до трех мы повторно сгенерируем график инерции (листинг 10.13; рис. 10.10).

Листинг 10.13. Построение графика инерции для симуляции с тремя мишенями

```
new_bulls_eye = [12, 0]
for _ in range(5000):
    x = np.random.normal(new_bulls_eye[0], variance ** 0.5)
    y = np.random.normal(new_bulls_eye[1], variance ** 0.5)
    darts.append([x, y])

inertia_values = [KMeans(k).fit(darts).inertia_
                  for k in k_values]

plt.plot(k_values, inertia_values)
plt.xlabel('K')
plt.ylabel('Inertia')
plt.show()
```

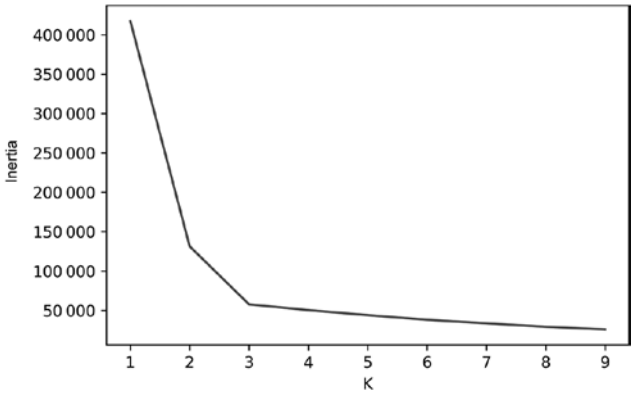


Рис. 10.10. График инерции для симуляции мишеней, содержащих три цели. Он походит на согнутую в локте руку. Нижняя позиция локтя указывает на K = 3

Добавление третьего центра дает новую форму локтя, чей нижний изгиб указывает на значение *K*, равное 3. По сути, этот график прослеживает рассеяние, включаемое каждым увеличением *K*. Быстрое увеличение инерции между последовательными значениями *K* подразумевает, что разбросанные точки данных должны быть приписаны к более плотному кластеру. По мере уплощения кривой инерции уменьшение этого показателя постепенно утрачивает свое влияние. Этот переход от вертикального уклона к более плавному углу ведет к появлению на графике формы локтя. Позицию этого локтя можно применить для выбора подобающего *K* в алгоритме *K*-средних.

Использование метода локтя — полезный эвристический прием, но он не обязательно сработает в каждом случае. В определенных условиях уровни локтя при переборе нескольких значений *K* уменьшаются медленно, что усложняет выбор единственно верного количества кластеров.

ПРИМЕЧАНИЕ

Существуют и более мощные методы выбора K, например коэффициент «силуэт», который учитывает расстояние каждой точки до соседних кластеров. Подробный разбор этого подхода выходит за рамки данной книги, но вы вполне можете изучить принцип его работы самостоятельно, используя метод sklearn.metrics.silhouette_score.

Метод локтя неидеален, но вполне пригоден, когда данные центрированы вокруг *K* раздельных средних. Естественно, это подразумевает, что наши кластеры данных различаются на основе центральности. Однако во многих случаях они различаются на основе плотности распределения точек данных в пространстве. Далее мы рассмотрим этот принцип организации регулируемых плотностью кластеров, которые не зависят от центральности.

МЕТОДЫ КЛАСТЕРИЗАЦИИ ПО K-СРЕДНИМ

- `k_means_model = KMeans(n_clusters=K)` — создает модель *K*-средних для поиска *K* различных центров масс. Эти центры масс необходимо сопоставить с входными данными.
- `clusters = k_means_model.fit_predict(data)` — выполняет алгоритм *K*-средних для входных данных, используя инициализированный объект `KMeans`. Возвращаемый массив `clusters` содержит ID кластеров в диапазоне от 0 до *K*. ID кластера `data[i]` равен `clusters[i]`.
- `clusters = KMeans(n_clusters=K).fit_predict(data)` — выполняет алгоритм *K*-средних в одной строке кода, возвращая полученные кластеры.
- `new_clusters = k_means_model.predict(new_data)` — находит ближайшие центры масс для ранее неизвестных данных, используя имеющиеся центры масс в оптимизированном данными объекте `KMeans`.
- `inertia = k_means_model.inertia_` — возвращает инерцию, связанную с оптимизированным данными объектом `KMeans`.
- `inertia = KMeans(n_clusters=K).fit(data).inertia_` — выполняет алгоритм *K*-средних в одной строке кода, возвращая полученную инерцию.

10.3. ОБНАРУЖЕНИЕ КЛАСТЕРОВ ПО ПЛОТНОСТИ

Представим, что астроном открывает новую планету в удаленном уголке Солнечной системы. Эта планета, очень похожая на Сатурн, имеет несколько колец, вращающихся на постоянных орбитах вокруг ее центра. Каждое кольцо сформировано из 1000 камней. Их мы смоделируем как отдельные точки, определяемые

координатами x и y. В коде листинга 10.14 с помощью функции `scikit-learn` `make_circles` генерируются три кольца, состоящие из множества камней (рис. 10.11).

Листинг 10.14. Симуляция колец, вращающихся вокруг планеты

```
from sklearn.datasets import make_circles

x_coordinates = []
y_coordinates = []
for factor in [.3, .6, 0.99]:
    rock_ring, _ = make_circles(n_samples=800, factor=factor,
                                noise=.03, random_state=1)
    for rock in rock_ring:
        x_coordinates.append(rock[0])
        y_coordinates.append(rock[1])

plt.scatter(x_coordinates, y_coordinates)
plt.show()
```

Функция make_circles создает три концентрических круга в 2D. Масштаб радиуса меньшего относительно большего определяется параметром factor

На графике четко видны три группы камней в форме колец. Далее мы найдем эти три кластера с помощью метода K-средних, установив K на 3 (листинг 10.15; рис. 10.12).

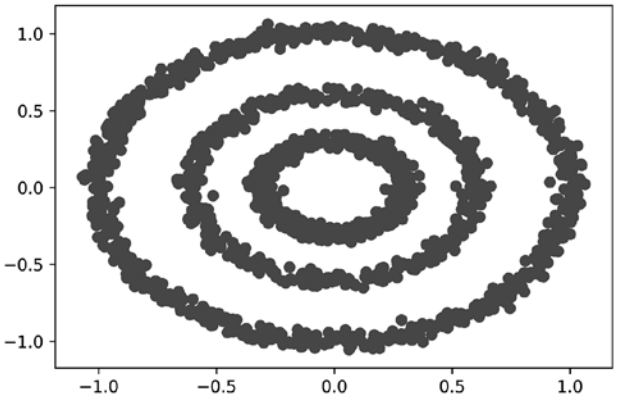


Рис. 10.11. Симуляция вокруг центра трех колец из камней

Листинг 10.15. Кластеризация колец по методу K-средних

```
rocks = [[x_coordinates[i], y_coordinates[i]]
         for i in range(len(x_coordinates))]
rock_clusters = KMeans(3).fit_predict(rocks)

colors = [['g', 'y', 'k'][cluster] for cluster in rock_clusters]
plt.scatter(x_coordinates, y_coordinates, color=colors)
plt.show()
```

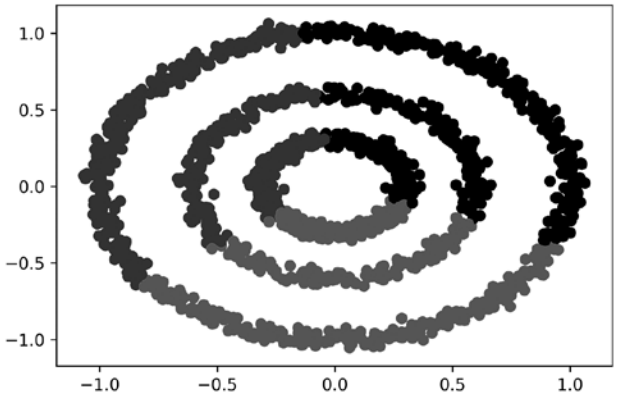


Рис. 10.12. Кластеризация по K-средним не может правильно определить три отдельных кольца

На выходе мы получаем полный провал. Метод *K*-средних рассекает данные на три симметричных сегмента, каждый из которых охватывает все три кольца. Полученное решение не соответствует тому, чего мы интуитивно ожидали, — что каждое кольцо будет относиться к собственной отдельной группе. Что же пошло не так? Просто алгоритм *K*-средних предположил, что три кластера были определены по трем уникальным центрам, хотя фактически кольца вращаются вокруг одной точки. Разница между кластерами определяется не центральностью, а плотностью. Плотные коллекции точек формируют отдельные кольца, а разреженные участки пространства эти кольца разделяют.

Нам нужно разработать алгоритм, кластеризующий данные в плотных областях пространства. Для этого потребуется определить, какая область является плотной, а какая — разреженной. Для определения *плотности* есть один простой способ: точка находится в плотной области, только если она расположена в пределах расстояния x от y других точек. x и y мы будем называть `epsilon` и `min_points` соответственно. Код листинга 10.16 устанавливает `epsilon` как `0.1`, а `min_points` — как `10`. Таким образом, наши камни находятся в плотной области, если расположены в рамках 0,1 радиуса от не менее чем десяти других камней.

Листинг 10.16. Установка параметров плотности

```
epsilon = 0.1
min_points = 10
```

Давайте проанализируем плотность в месте расположения первого камня в списке `rocks`. Начнем с поиска всех других камней в радиусе `epsilon` единиц от `rocks[0]` (листинг 10.17). Индексы соседних камней сохраним в списке `neighbor_indices`.

Листинг 10.17. Поиск соседей rocks[0]

```
neighbor_indices = [i for i, rock in enumerate(rocks[1:])
                    if euclidean(rocks[0], rock) <= epsilon]
```

Теперь сравним количество найденных соседей с `min_points`, чтобы определить, расположен ли `rocks[0]` в плотной области пространства (листинг 10.18).

Листинг 10.18. Проверка плотности пространства вокруг rocks[0]

```
num_neighbors = len(neighbor_indices)
print(f"The rock at index 0 has {num_neighbors} neighbors.")

if num_neighbors >= min_points:
    print("It lies in a dense region.")
else:
    print("It does not lie in a dense region.")

The rock at index 0 has 40 neighbors.
It lies in a dense region.
```

Камень с индексом 0 находится в плотном пространстве. А что насчет соседей `rocks[0]` — они располагаются в таком же пространстве? Непростой вопрос. В конце концов может оказаться, что число соседствующих с каждым из его соседей до `min_points` не дотягивает. В рамках нашей конкретной функции определения плотности не будем относить этих соседей к точкам из плотных областей. Однако это приведет к абсурдной ситуации, в которой плотная область состоит всего из одной точки, `rocks[0]`. Чтобы избежать такого нелепого результата, обновим определение плотности, формально описав ее так.

- Если точка расположена в рамках расстояния `epsilon` от `min_points` соседей, тогда она находится в плотной области.
- Каждый сосед точки в плотной области также включается в кластер этой области.

На основе обновленного определения можно объединить `rocks[0]` и его соседей в один плотный кластер (листинг 10.19).

Листинг 10.19. Создание плотного кластера

```
dense_region_indices = [0] + neighbor_indices
dense_region_cluster = [rocks[i] for i in dense_region_indices]
dense_cluster_size = len(dense_region_cluster)
print(f"We found a dense cluster containing {dense_cluster_size} rocks")

We found a dense cluster containing 41 rocks
```

Камень с индексом 0 и его соседи формируют один кластер плотностью 41 элемент. Принадлежат ли какие-либо соседи этих соседей к плотной области пространства? Если да, то согласно новому определению эти камни также принадлежат плотному

кластеру. Таким образом, анализируя дополнительные соседние точки, можно расширить размер `dense_region_cluster` (листинг 10.20).

Листинг 10.20. Расширение плотного кластера

```
dense_region_indices = set(dense_region_indices)
for index in neighbor_indices:
    point = rocks[index]
    neighbors_of_neighbors = [i for i, rock in enumerate(rocks)
                              if euclidean(point, rock) <= epsilon]
    if len(neighbors_of_neighbors) >= min_points:
        dense_region_indices.update(neighbors_of_neighbors)

dense_region_cluster = [rocks[i] for i in dense_region_indices]
dense_cluster_size = len(dense_region_cluster)
print(f"We expanded our cluster to include {dense_cluster_size} rocks")
```

Преобразует dense_region_indices во множество. Это позволяет обновить данное множество дополнительными индексами, не беспокоясь о повторах

```
We expanded our cluster to include 781 rocks
```

Мы перебрали всех соседей и расширили плотный кластер почти в 20 раз. Зачем же останавливаться? Можно расширить кластер еще больше, проанализировав плотность новых встречающихся соседей. Итеративный повтор анализа расширит границы нашего кластера, и в итоге в них полностью войдет одно из колец. Тогда, не имея дополнительных соседей для включения в кластер, можно повторить итеративный анализ элемента `rocks`, который еще не был рассмотрен. Это приведет к кластеризации дополнительных плотных колец.

Описанная только что процедура называется DBSCAN и представляет собой алгоритм, организующий данные на основе их пространственного распределения.

10.4. DBSCAN: АЛГОРИТМ КЛАСТЕРИЗАЦИИ ДЛЯ ГРУППИРОВКИ ДАННЫХ НА ОСНОВЕ ПРОСТРАНСТВЕННОЙ ПЛОТНОСТИ

DBSCAN — это акроним, означающий *«плотностный алгоритм кластеризации пространственных данных в присутствии шума»*. Это до смешного громоздкое название на самом деле относится к довольно простой процедуре.

1. Выбрать из списка `data` координаты `point`.
2. Получить всех соседей в радиусе `epsilon` от этой `point`.
3. Если найдено меньше `min_points` соседей, повторить шаг 1, используя другую случайную точку. В противном случае сгруппировать `point` и ее соседей в один кластер.

4. Итеративно повторить шаги 2 и 3 для всех обнаруженных соседей. Все соседние, расположенные в плотной области точки сливаются в один кластер. Когда кластер перестает расширяться, итерации прекращаются.
5. После извлечения всего кластера повторить шаги 1–4 для всех точек данных, чья плотность еще не была проанализирована.

Процедуру DBSCAN можно запрограммировать меньше чем в 20 строках кода. Однако любая базовая реализация будет выполняться для списка `rocks` очень медленно. Для создания ее ускоренного варианта потребуются некоторые очень тонкие оптимизации, которые повышают скорость обхода соседей и выходят за рамки темы этой книги. К счастью, у нас нет необходимости воссоздавать этот алгоритм с нуля — `scikit-learn` предоставляет быстродействующий класс DBSCAN, который можно импортировать из `sklearn.cluster`. Давайте сделаем это и инициализируем данный класс, присвоив `epsilon` и `min_points` с помощью параметров `eps` и `min_samples`. Затем задействуем DBSCAN для кластеризации наших трех колец (листинг 10.21; рис. 10.13).

Листинг 10.21. Кластеризация колец с помощью DBSCAN

```
from sklearn.cluster import DBSCAN
cluster_model = DBSCAN(eps=epsilon, min_samples=min_points)
rock_clusters = cluster_model.fit_predict(rocks)
colors = [['g', 'y', 'k'][cluster] for cluster in rock_clusters]
plt.scatter(x_coordinates, y_coordinates, color=colors)
plt.show()
```

Создает объект cluster_model для выполнения кластеризации по плотности. Значение epsilon 0,1 передается с помощью параметра eps, а значение min_points 10 — с помощью параметра min_samples

Группирует кольца камней на основе плотности, возвращая для каждого камня соответствующий кластер

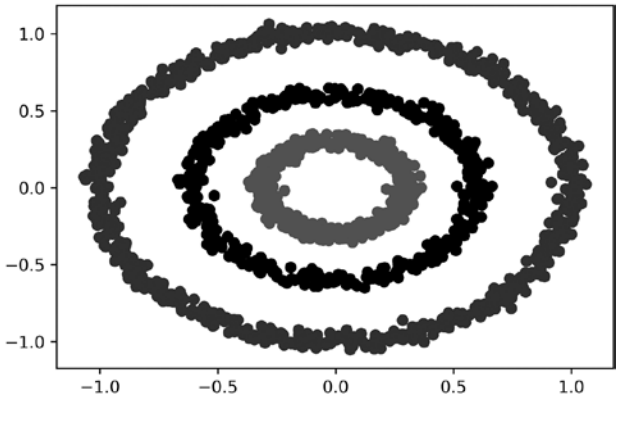


Рис. 10.13. Кластеризация с помощью DBSCAN точно определяет три отдельных кольца камней

DBSCAN благополучно определил три кольца камней, преуспев там, где не справился метод *K*-средних.

10.4.1. Сравнение DBSCAN и метода K-средних

DBSCAN — это более выигрышный алгоритм для кластеризации данных, имеющих изогнутую форму или различную плотность. Кроме того, в отличие от *K*-средних, он не требует для своего выполнения аппроксимации количества кластеров, а также может отфильтровывать случайные выбросы, расположенные в разреженных областях пространства (листинг 10.22). К примеру, если прибавить выброс, расположенный за границами колец, DBSCAN присвоит ему ID кластера –1. Отрицательное значение указывает, что выброс нельзя кластеризовать в рамках остального набора данных.

ПРИМЕЧАНИЕ

В отличие от K-средних подогнанную модель DBSCAN нельзя повторно применить к новым данным. Вместо этого необходимо сначала совместить новые данные со старыми, после чего повторять кластеризацию с нуля. Дело в том, что вычисленные центры K-средних можно легко сравнить с дополнительными точками данных. Однако дополнительные точки данных могут повлиять на плотность распределения уже проанализированных данных, что вынудит DBSCAN вычислить кластеры повторно.

Листинг 10.22. Обнаружение выбросов с помощью DBSCAN

```
noisy_data = rocks + [[1000, -1000]]
clusters = DBSCAN(eps=epsilon,
                  min_samples=min_points).fit_predict(noisy_data)
assert clusters[-1] == -1
```

Еще одно преимущество DBSCAN в том, что эта техника не зависит от среднего. А вот алгоритм *K*-средних требует вычисления координат среднего сгруппированных точек. Как говорилось в главе 5, эти координаты минимизируют сумму квадратов расстояний до центра. Это свойство минимизации сохраняется, только если квадраты расстояний выражены в евклидовой метрике. Таким образом, если координаты будут неевклидовы, то пользы от среднего получится немного и алгоритм *K*-средних применять не стоит. Тем не менее евклидово расстояние — это не единственный показатель для расчета отдаленности точек друг от друга, подобную процедуру можно реализовать с помощью множества других показателей. Некоторые из них мы изучим в следующем разделе и попутно разберем, как интегрировать эти показатели в наш вывод кластеризации DBSCAN.

10.4.2. Кластеризация с помощью неевклидовой метрики

Представим, что мы гуляем по Манхэттену и хотим узнать расстояние, пройденное от Эмпайр-стейт-билдинг до Коламбус-сёркл. Эмпайр-стейт-билдинг расположен на пересечении 34-й улицы и 5-й авеню, а Коламбус-сёркл — на пересечении 57-й улицы и 8-й авеню. Улицы и авеню в Манхэттене располагаются перпендикулярно друг к другу. Это позволяет представить остров как двухмерную систему координат, в которой улицы расположены по оси *X*, а авеню — по оси *Y*. В рамках такого представления координаты Эмпайр-стейт-билдинг (34; 5), а Коламбус-сёркл — (57; 8). Можно легко вычислить прямое евклидово расстояние между этими двумя точками. Однако в итоге пройти по этой прямой не удастся, поскольку каждый квартал застроен различными зданиями. Более правильное решение сводится к пути вдоль перпендикулярных тротуаров, которые формируют сетку городских улиц. Подобный маршрут потребует пройти три квартала между 5-й и 3-й авеню, после чего преодолеть 23 квартала между 34-й и 57-й улицами. Итого получится 26 кварталов. Средняя протяженность квартала в Манхэттене составляет 0,17 мили, значит, можно примерно оценить общее расстояние прогулки как равное 4,42 мили. Давайте вычислим это расстояние непосредственно с помощью функции `manhattan_distance` (листинг 10.23).

Листинг 10.23. Вычисление расстояния прогулки по Манхэттену

```
def manhattan_distance(point_a, point_b):
    num_blocks = np.sum(np.absolute(point_a – point_b))
    return 0.17 * num_blocks

x = np.array([34, 5])
y = np.array([57, 8])
distance = manhattan_distance(x, y)

print(f"Manhattan distance is {distance} miles")

Manhattan distance is 4.42 miles
```

Этот вывод также можно сгенерировать, импортировав cityblock из scipy.spatial.distance и выполнив 0.17 * cityblock(x, y)

Теперь предположим, что мы хотим кластеризовать более двух локаций Манхэттена. Допустим, что каждый кластер содержит точку, находящуюся в радиусе одной мили ходьбы от трех других точек кластера. Это допущение позволит применить технику DBSCAN с помощью класса DBSCAN из scikit-learn. При инициализации мы установим `eps` на `1`, а `min_samples` — на `3`. Более того, мы передадим в метод инициализации `metric=manhattan_distance`. Параметр `metric` меняет параметр евклидова расстояния на наше собственное, в результате чего расстояния в кластере будут корректно отражать городские ограничения сетки. Код листинга 10.24 кластеризует координаты в Манхэттене и отображает их на сетке, попутно обозначая принадлежность к кластеру (рис. 10.14).

Листинг 10.24. Кластеризация с помощью параметра расстояния городских кварталов

```
points = [[35, 5], [33, 6], [37, 4], [40, 7], [45, 5]]
clusters = DBSCAN(eps=1, min_samples=3,
                  metric=manhattan_distance).fit_predict(points)

for i, cluster in enumerate(clusters):
    point = points[i]
    if cluster == -1:
        print(f"Point at index {i} is an outlier")
        plt.scatter(point[0], point[1], marker='x', color='k')
    else:
        print(f"Point at index {i} is in cluster {cluster}")
        plt.scatter(point[0], point[1], color='g')

plt.grid(True, which='both', alpha=0.5)
plt.minorticks_on()

plt.show()
```

Функция manhattan_distance передается в DBSCAN через параметр metric

Выбросы обозначены x-образными маркерами

Метод grid отображает прямоугольную сетку, по которой мы вычисляем расстояние

```
Point at index 0 is in cluster 0
Point at index 1 is in cluster 0
Point at index 2 is in cluster 0
Point at index 3 is an outlier
Point at index 4 is an outlier
```

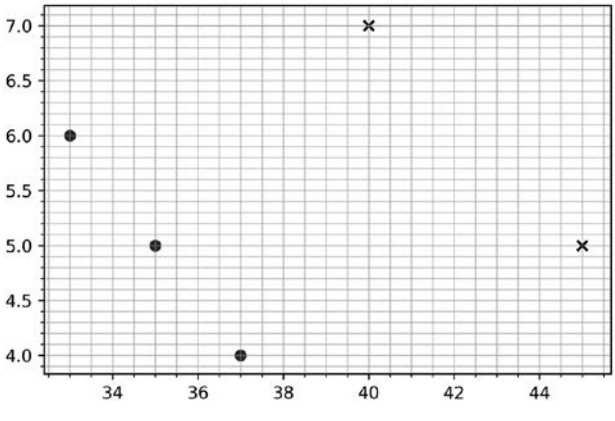


Рис. 10.14. Пять точек на прямоугольной сетке, кластеризованные с использованием параметра расстояния городских кварталов. Три точки в левом нижнем углу попадают в один кластер, а оставшиеся два выброса обозначены x

Первые три локации попадают в один кластер, а оставшиеся точки оказываются выбросами. Можно ли было обнаружить это с помощью алгоритма *K*-средних? Вероятно. В конце концов, координаты кварталов Манхэттена можно усреднить, сделав их совместимыми с реализацией метода *K*-средних. А что, если поменять манхэттенское расстояние на другой показатель, для которого средние координаты

получить не так просто? Давайте определим параметр вычисления нелинейного расстояния со следующими свойствами: две точки удалены друг от друга на ноль единиц, если все их элементы отрицательны, на две единицы, если не отрицательны, и на десять точек в остальных случаях. Можно ли, опираясь на этот нелепый способ измерения расстояния, вычислить среднее для любых двух произвольных точек? Нельзя, и алгоритм *K*-средних применить не получится. Слабость этого алгоритма в том, что он зависит от наличия среднего расстояния. В отличие же от *K*-средних техника DBSCAN не требует, чтобы функция расстояния была линейно делимой. Таким образом, можно легко выполнить кластеризацию с помощью DBSCAN, применив наш смешной показатель (листинг 10.25).

Листинг 10.25. Кластеризация с использованием несуразного показателя оценки расстояния

```
def ridiculous_measure(point_a, point_b):
    is_negative_a = np.array(point_a) < 0
    is_negative_b = np.array(point_b) < 0
    if is_negative_a.all() and is_negative_b.all():
        return 0
    elif is_negative_a.any() or is_negative_b.any():
        return 10
    else:
        return 2

points = [[-1, -1], [-10, -10], [-1000, -13435], [3,5], [5,-7]]

clusters = DBSCAN(eps=.1, min_samples=2,
                  metric=ridiculous_measure).fit_predict(points)

for i, cluster in enumerate(clusters):
    point = points[i]
    if cluster == -1:
        print(f"{point} is an outlier")
    else:
        print(f"{point} falls in cluster {cluster}")
```

Возвращает логический массив, в котором is_negative_a[i] является True, если point_a[i] < 0

Все элементы point_a и point_b отрицательны

Отрицательный элемент присутствует, но не все остальные элементы также отрицательны

Все элементы неотрицательны

```
[-1, -1] falls in cluster 0
[-10, -10] falls in cluster 0
[-1000, -13435] falls in cluster 0
[3, 5] is an outlier
[5, -7] is an outlier
```

Выполнение DBSCAN с нашим смешным показателем `ridiculous_measure` ведет к кластеризации отрицательных координат в одну группу. Все остальные координаты рассматриваются как выбросы. Такие результаты не особо пригодны для практического применения, но подобная гибкость относительно выбора показателя имеет свою ценность, делая нас в этом смысле свободными. Можно, к примеру, установить параметр для вычисления расстояния пути на основе кривизны Земли. Он окажется особенно полезным для кластеризации географических локаций.

МЕТОДЫ КЛАСТЕРИЗАЦИИ DBSCAN

- `dbscan_model = DBSCAN(eps=epsilon, min_samples=min_points)` — создает модель DBSCAN для выполнения кластеризации по плотности. Точка относится к плотной области, если имеет не менее `min_points` соседей в радиусе `epsilon`. Эти соседи считаются частью того же кластера, что и базовая точка.
- `clusters = dbscan_model.fit_predict(data)` — выполняет DBSCAN для входных данных, используя инициализированный объект `DBSCAN`. Массив `clusters` содержит ID кластеров. ID кластера `data[i]` равно `clusters[i]`. Некластеризованным точкам выбросов присваивается ID, равный –1.
- `clusters = DBSCAN(eps=epsilon, min_samples=min_points).fit_predict (data)` — выполняет DBSCAN в одной строке кода, возвращая полученные кластеры.
- `dbscan_model = DBSCAN(eps=epsilon, min_samples=min_points, metric=metric_ function)` — создает модель DBSCAN, в которой показатель определяется кастомной функцией. Показатель вычисления расстояния `metric_function` не обязательно должен быть евклидовым.

DBSCAN лишен явно выраженных недостатков. Этот алгоритм предназначен для обнаружения кластеров со схожими плотностями распределений точек. Однако в реальности плотность данных различается. К примеру, в Манхэттене пиццерии расположены более часто, чем в округе Ориндж, Калифорния. Таким образом, у нас может возникнуть сложность с выбором параметров плотности, которые позволили бы кластеризовать пиццерии в обеих локациях. Это подчеркивает еще одно ограничение DBSCAN — он требует предоставления для параметров `eps` и `min_samples` осмысленных значений. В частности, изменение `eps` будет серьезно влиять на качество кластеризации. К сожалению, не существует надежной процедуры для оценки подходящего значения `eps`. И хотя в литературе местами упоминаются определенные эвристики, пользы от них мало. В большинстве случаев для присваивания разумных значений этим двум параметрам необходимо опираться на внутреннее ощущение от задачи. К примеру, если потребуется кластеризовать набор географических локаций, то значения `eps` и `min_samples` будут зависеть от того, разбросаны эти места по всему глобусу или находятся в одном географическом регионе. В каждом из этих случаев наше понимание плотности и расстояния будет различным. Говоря в общем, если требуется кластеризовать случайные города, разбросанные по всей Земле, то можно установить параметры `min_samples` и `esp` как равные трем городам и 250 милям соответственно. Это задаст предположение, что каждый кластер содержит город, находящийся в пределах 250 миль от не менее чем трех других кластеризуемых городов. В случае же более ограниченной территории распределения локаций выбирается меньшее значение `eps`.

10.5. АНАЛИЗ КЛАСТЕРОВ С ПОМОЩЬЮ PANDAS

До этого момента мы хранили входные данные и кластеризованные выводы отдельно друг от друга. К примеру, при анализе колец из камней входные данные находятся в списке `rocks`, а кластеризованный вывод — в массиве `rock_clusters`. Одновременное отслеживание координат и кластеров требует сопоставления индексов между входным списком и выходным массивом. Значит, если мы хотим извлечь все камни из кластера 0, то должны получить все экземпляры `rocks[i]`, где `rock_clusters[i] == 0`. Такой анализ индекса получается запутанным. Можно более эффективно проанализировать кластеризованные камни, объединив их координаты и кластеры в одну таблицу Pandas.

Код листинга 10.26 создает такую таблицу с тремя столбцами: `X`, `Y` и `Cluster`. В каждом `i` ряду этой таблицы содержатся координаты x и y, а также кластер камня, расположенного в `rocks[i]`.

Листинг 10.26. Сохранение кластеризованных координат в таблице

```
import pandas as pd
x_coordinates, y_coordinates = np.array(rocks).T
df = pd.DataFrame({'X': x_coordinates, 'Y': y_coordinates,
                   'Cluster': rock_clusters})
```

Полученная таблица позволит нам легко обращаться к камням, находящимся в любом кластере. На рис. 10.15 с помощью приемов, описанных в главе 8, отображены камни, относящиеся к кластеру 0 (листинг 10.27).

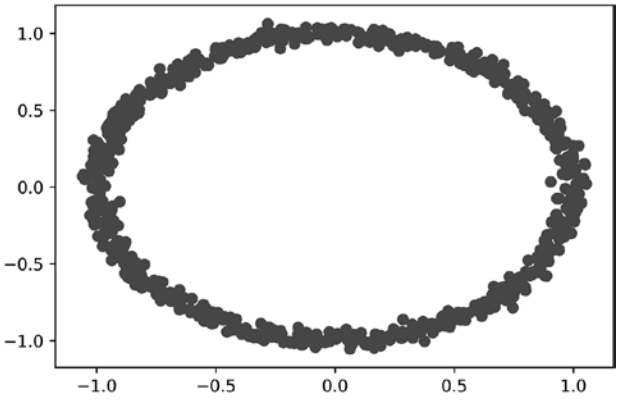


Рис. 10.15. Камни из кластера 0

Листинг 10.27. Отображение одного кластера с помощью Pandas

```
df_cluster = df[df.Cluster == 0]
plt.scatter(df_cluster.X, df_cluster.Y)
plt.show()
```

Выбирает только те строки, в которых столбец Cluster равен 0

Наносит на график столбцы X и Y выбранных строк. Заметьте, что этот точечный график также можно построить, выполнив df_cluster.plot.scatter(x='X', y='Y')

Pandas позволяет получить таблицу, содержащую элементы любого одного кластера. В качестве альтернативы нам может потребоваться получить несколько таблиц, каждая из которых будет отображаться в ID кластера. В Pandas это делается с помощью вызова `df.groupby('Cluster')`. Метод `groupby` создаст три таблицы, по одной для каждого кластера. Он вернет итерируемый объект для всех сопоставлений между ID кластера и таблицами. Далее мы с помощью метода `groupby` переберем наши три кластера, после чего графически отобразим камни только из кластеров 1 и 2 (листинг 10.28; рис. 10.16).

ПРИМЕЧАНИЕ

Вызов df.groupby('Cluster') возвращает не просто итерируемый объект, а объект DataFrameGroupBy, который предоставляет дополнительные методы для фильтрации и анализа кластера.

Листинг 10.28. Перебор кластеров с помощью Pandas

```
for cluster_id, df_cluster in df.groupby('Cluster'):
    if cluster_id == 0:
        print(f"Skipping over cluster {cluster_id}")
        continue

    print(f"Plotting cluster {cluster_id}")
    plt.scatter(df_cluster.X, df_cluster.Y)

plt.show()
```

Каждый элемент итерируемого объекта, возвращаемый df.groupby('Cluster'), является кортежем. Первый элемент кортежа — это ID кластера, полученный из df.Cluster. Второй элемент — это таблица, состоящая из всех строк, в которых df.Cluster равен ID этого кластера

```
Skipping over cluster 0
Plotting cluster 1
Plotting cluster 2
```

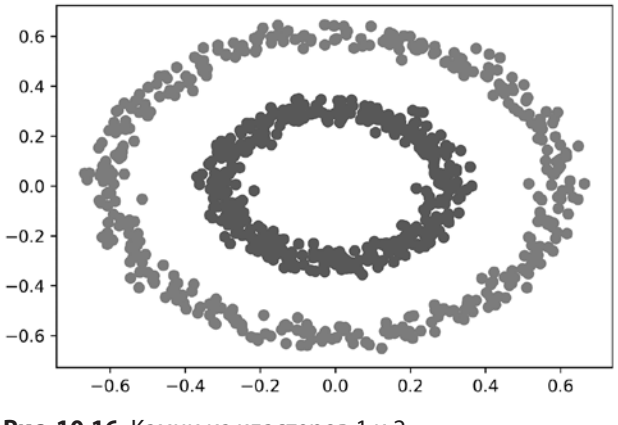


Рис. 10.16. Камни из кластеров 1 и 2

Метод Pandas `groupby` позволяет итеративно анализировать разные кластеры, что как раз и пригодится для решения нашего третьего практического задания.

РЕЗЮМЕ

- Алгоритм *K-средних* кластеризует входные данные путем нахождения *K центров масс*. Эти центры масс представляют средние координаты обнаруженных групп данных. При инициализации метода *K*-средних выбирается *K* случайных центров масс. Затем каждая точка данных кластеризуется на основе ближайшего к ней центра, а сами центры итеративно перевычисляются, пока не сойдутся в стабильных локациях.
- Алгоритм *K*-средних гарантированно сходится к решению. Однако это решение может оказаться неоптимальным.
- Алгоритму *K*-средних для проведения различия между точками требуются евклидовы расстояния. Этот алгоритм не предназначен для кластеризации неевклидовых координат.
- После кластеризации методом *K*-средних можно вычислить *инерцию* полученного результата. Она равняется сумме квадратов расстояний между каждой точкой данных и ближайшим к ней центром.
- Графическое отображение инерции для диапазона значений *K* дает *локтеобразный график*. Угол локтя в этом графике должен указывать вниз на оптимальное значение *K*. С помощью такого графика можно эвристически выбирать разумные входные *K* для алгоритма *K*-средних.
- Алгоритм *DBSCAN* кластеризует данные на основе их плотности. Она определяется с помощью параметров `epsilon` и `min_points`. Если точка расположена в пределах радиуса `epsilon` от `min_points` соседей, значит, она находится в плотной области пространства. Каждый сосед точки в плотной области также кластеризуется в этой области. DBSCAN итеративно расширяет границы плотной области пространства, пока не будет обнаружен весь кластер.
- Точки в неплотных областях алгоритмом DBSCAN не кластеризуются — они рассматриваются как выбросы.
- DBSCAN является более эффективным алгоритмом для кластеризации данных, составляющих изогнутые и плотные фигуры.
- DBSCAN может выполнять кластеризацию, используя произвольные, неевклидовы расстояния.
- Не существует надежного эвристического метода для выбора оптимальных параметров `epsilon` и `min_points`. Однако если мы хотим кластеризовать глобальные мегаполисы, то можно установить для них значения 250 миль и три города соответственно.
- Сохранение кластеризованных данных в таблице Pandas позволяет интуитивно перебирать кластеры методом `groupby`.

11 Визуализация и анализ географических локаций

В этой главе

- ✓ Вычисление расстояния между географическими локациями.
- ✓ Отображение локаций на карте с помощью библиотеки Cartopy.
- ✓ Извлечение географических координат из названий локаций.
- ✓ Поиск названий локаций в тексте с помощью регулярных выражений.

Люди опирались на информацию о местности задолго до начала ведения письменной истории. В древности пещерные жители вырезали на бивнях мамонтов карты охотничьих маршрутов. С развитием цивилизации составление подобных карт становилось все более изощренным. Древние вавилоняне подробно отражали на картах границы своей империи. Намного позднее, в 3000 году до н. э. греческие ученые усовершенствовали картографию, используя последние математические открытия. Они выяснили, что Земля круглая, и точно вычислили ее окружность. Греческие математики заложили основу для измерения расстояний вдоль изогнутой поверхности Земли. Подобные измерения потребовали создания системы географических координат — первая система, основанная на долготе и широте, появилась в 2000 году до н. э.

Совмещение картографии с широтой и долготой произвело революцию в судоходстве. Моряки смогли более свободно путешествовать по морям, уточняя свое местоположение по карте. В общем виде протокол морской навигации состоял из трех шагов.

1. *Наблюдение* — моряк регистрировал серию наблюдений, включая направление ветра, расположение звезд и — где-то после 1300 года н. э. — направление на север по компасу.
2. *Математический и алгоритмический анализ собранных данных* — штурман анализировал все собранные данные, оценивая местоположение корабля. В некоторых случаях подобный анализ требовал тригонометрических вычислений, хотя чаще штурман обращался к набору карт измерений, основанных на определенном наборе правил. Алгоритмически придерживаясь правил этих карт, он мог определить координаты корабля.
3. *Визуализация и принятие решений* — капитан оценивал вычисленное местоположение корабля на карте относительно предполагаемого места назначения, после чего отдавал приказ скорректировать направление движения, исходя из визуально представленных результатов.

Эта парадигма навигации идеально отражает стандартный процесс анализа данных. Будучи аналитиками, мы получаем просто сырые наблюдения. Эти данные мы алгоритмически анализируем, после чего визуализируем полученные результаты для принятия важных решений. Получается, что между наукой о данных и анализом местоположения есть связь, которая на протяжении многих веков становилась все крепче. Сегодня бесчисленное множество корпораций анализируют местоположение такими способами, которые древние греки не могли себе даже представить. Хедж-фонды изучают спутниковые снимки фермерских угодий, чтобы делать осмысленные ставки на глобальном рынке соевых бобов. Компании-перевозчики анализируют обширные схемы дорожного движения, чтобы эффективно прокладывать маршруты автомобилей. Эпидемиологи обрабатывают газетные данные, отслеживая глобальное распространение заболеваний.

В текущей главе мы изучим различные техники анализа и визуализации географических локаций, начав с простого задания по вычислению расстояния между двумя географическими точками.

11.1. РАССТОЯНИЕ ПО ОРТОДРОМИИ: ПОКАЗАТЕЛЬ ДЛЯ ВЫЧИСЛЕНИЯ РАССТОЯНИЯ МЕЖДУ ДВУМЯ ГЛОБАЛЬНЫМИ ТОЧКАМИ

Каков кратчайший путь между любой парой точек на Земле? Этот путь не может быть прямой линией, поскольку прямое линейное путешествие потребовало бы бурения глубокого тоннеля через кору планеты. Интересующее нас прямое расстояние между двумя точками вдоль поверхности сферы называется *расстоянием по ортодромии* (рис. 11.1).

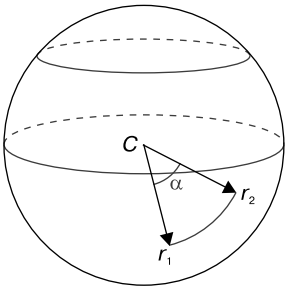


Рис. 11.1. Визуализация расстояния по ортодромии между двумя точками на поверхности сферы. Эти точки обозначены r_1 и r_2, а путь между ними показан дугой. Длина дуги равна радиусу сферы, умноженному на величину α, представляющую угол между точками относительно центра сферы C

Это расстояние можно вычислить на основе сферы и двух точек на ней. Любую точку на поверхности сферы можно представить *сферическими координатами x* и *y*, которые будут отражать углы этой точки относительно осей *X* и *Y* (рис. 11.2).

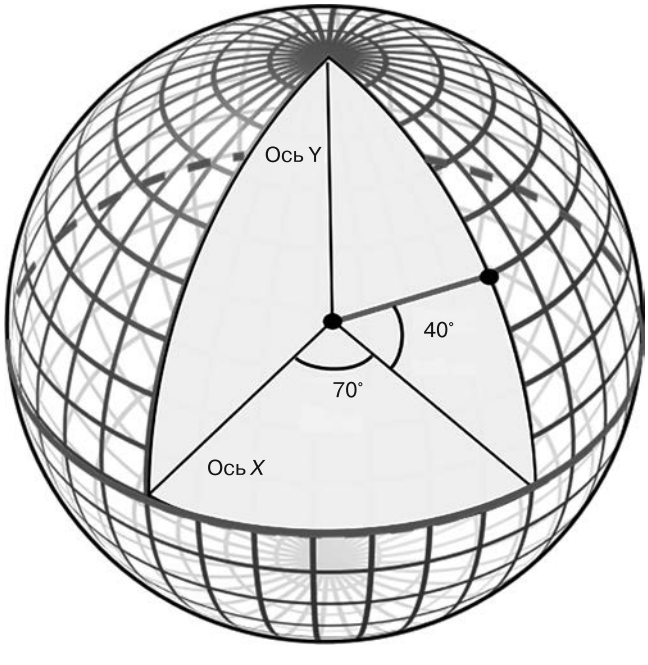


Рис. 11.2. Представление точки на поверхности сферы с помощью сферических координат. Эта точка получается при повороте на 70° от оси X и на 40° от оси Y. Таким образом, ее сферические координаты (70; 40)

Давайте определим функцию `great_circle_distance`, получающую на входе две пары сферических координат (листинг 11.1). Для простоты предположим, что эти координаты находятся на единичной сфере радиусом 1. Такое упрощение позволит прописать `great_circle_distance` всего в четырех строках кода. Эта функция зависит от серии известных тригонометрических операций, подробный разбор которых выходит за рамки нашей книги.

Листинг 11.1. Определение функции вычисления расстояния по ортодромии

```
from math import cos, sin, asin

def great_circle_distance(x1, y1, x2, y2):
    delta_x, delta_y = x2 – x1, y2 – y1
    haversin = sin(delta_x / 2) ** 2 + np.product([cos(x1), cos(x2),
                                                   sin(delta_y / 2) ** 2])
    return 2 * asin(haversin ** 0.5)
```

Импортирует из модуля Python math три типичные тригонометрические функции

Вычисляет угловую разницу между двумя парами сферических координат

Выполняет серию известных тригонометрических операций для получения расстояния по ортодромии на единичной сфере. Функция np.product перемножает три тригонометрических значения

Тригонометрические функции Python предполагают, что входной угол указан в радианах, где 0° = 0 рад, а 180° = 2π рад. Далее мы вычислим расстояние по ортодромии между двумя точками, которые удалены на 180° относительно обеих осей, *X* и *Y* (листинг 11.2).

ПРИМЕЧАНИЕ

В радианах измеряется длина дуги единичного круга относительно угла. Максимальная дуга равняется длине окружности единичного круга 2π. Для обхода всего круга требуется пройти 360°. Значит, 2π рад = 360°, а 1° = π/180 рад.

Листинг 11.2. Вычисление расстояния по ортодромии

```
from math import pi
distance = great_circle_distance(0, 0, 0, pi)
print(f"The distance equals {distance} units")

The distance equals 3.141592653589793 units
```

Точки удалены друг от друга ровно на π единиц, то есть на половину расстояния, необходимого для обхода единичного круга. Это значение является максимально возможным расстоянием, которое можно пройти между двумя точками на сфере. Можно сравнить это с путешествием между Северным и Южным полюсами любой планеты. Убедимся мы в этом, проанализировав широту и долготу Северного и Южного полюсов Земли. Земные широта и долгота являются сферическими координатами, измеряемыми в градусах. Начнем с регистрации известных координат каждого полюса (листинг 11.3).

Листинг 11.3. Определение координат полюсов Земли

```
latitude_north, longitude_north = (90.0, 0)
latitude_south, longitude_south = (-90.0, 0)
```

Технически Северный и Южный полюса не имеют официальных координат долготы. Однако математически мы уполномочены присвоить каждому полюсу в качестве долготы нулевое значение

Широта и долгота отражают сферические координаты в градусах, а не в радианах. Поэтому мы выполним преобразование из градусов в радианы, используя функцию `np.radians`. Она получает список градусов и возвращает массив радиан. Затем полученный результат можно передать в `great_circle_distance` (листинг 11.4).

Листинг 11.4. Вычисление расстояния по ортодромии между полюсами

```
to_radians = np.radians([latitude_north, longitude_north,
                         latitude_south, longitude_south])
distance = great_circle_distance(*to_radians.tolist())
print(f"The unit-circle distance between poles equals {distance} units")

The unit-circle distance between poles equals 3.141592653589793 units
```

Напомню, что выполнение func(*[arg1, arg2]) — это сокращенная форма выполнения func(arg1, arg2) в Python

Как и ожидалось, расстояние между полюсами на единичной сфере равно π. Теперь измерим расстояние между полюсами Земли. Радиус этой планеты равен не одной гипотетической единице, а 3956 милям, поэтому для получения искомых измерений нужно умножить `distance` на 3956 (листинг 11.5).

Листинг 11.5. Вычисление расстояния между полюсами Земли

```
earth_distance = 3956 * distance
print(f"The distance between poles equals {earth_distance} miles")

The distance between poles equals 12428.14053760122 miles
```

Расстояние между двумя полюсами составляет примерно 12 400 миль. Для его получения мы преобразовали широту и долготу в радианы, вычислили расстояние между полюсами на единичной сфере, после чего умножили найденное значение на радиус Земли. Теперь можно создать общую функцию `travel_distance` для вычисления пути между двумя любыми точками на земной поверхности (листинг 11.6).

Листинг 11.6. Определение функции для вычисления пути

```
def travel_distance(lat1, lon1, lat2, lon2):
    to_radians = np.radians([lat1, lon1, lat2, lon2])
    return 3956 * great_circle_distance(*to_radians.tolist())

assert travel_distance(90, 0, -90, 0) == earth_distance
```

Наша функция `travel_distance` является неевклидовым показателем для измерения расстояния между локациями. Как говорилось в предыдущем разделе, подобные

показатели можно передавать в алгоритм кластеризации DBSCAN. Значит, у нас есть возможность использовать `travel_distance` для кластеризации локаций на основе их пространственных распределений. Затем можно будет визуально оценить кластеры, отобразив локации на карте. Нанесение на карту мы выполним с помощью внешней библиотеки визуализации Cartopy.

11.2. ПОСТРОЕНИЕ КАРТ С ПОМОЩЬЮ CARTOPY

Визуализация географических данных представляет собой рядовую задачу науки о данных. Одной из используемых для этого библиотек является Cartopy — Matplotlib-совместимый инструмент для генерации карт в Python. К сожалению, установка Cartopy имеет особенность. Все остальные рассматриваемые в книге библиотеки можно установить с помощью команды `pip install`. Она вызывает систему управления пакетами pip, которая подключается к внешнему серверу, содержащему библиотеки Python. Затем pip устанавливает выбранную библиотеку вместе со всеми ее зависимостями Python.

ПРИМЕЧАНИЕ

Например, NumPy является зависимостью Matplotlib. Вызов pip install matplotlib автоматически установит NumPy на локальную машину, если ее там еще не было.

Pip прекрасно работает, когда все зависимости библиотеки написаны на Python. Однако Cartopy содержит зависимость на языке C++. В основе ее визуализаций лежит библиотека GEOS, представляющая геопространственный движок. Эту библиотеку нельзя установить с помощью pip, поэтому и Cartopy установить напрямую через pip не выйдет. Здесь есть два выхода:

- установить GEOS и Cartopy вручную;
- установить Cartopy через пакетный менеджер Conda.

Разберем плюсы и минусы каждого из этих вариантов.

ПРИМЕЧАНИЕ

Более подробно познакомиться с зависимостями Python можно в видео издательства Manning: Managing Python Dependencies по адресу www.manning.com/livevideo/talk-python-managing-python-dependencies.

11.2.1. Установка GEOS и Cartopy вручную

Установка GEOS выполняется по-разному в зависимости от операционной системы. На MacOS ее можно поставить, выполнив из командной строки `brew install proj`

geos. В Linux это делается с помощью команды apt-get, а не brew. Что касается Windows, то пользователи этой ОС могут скачать библиотеку по адресу https://trac.osgeo.org/geos. После установки можно будет добавить Cartopy со всеми ее зависимостями, поочередно выполнив следующие команды pip.

1. pip install --upgrade cython numpy pyshp six. Устанавливает все зависимости Python, кроме библиотеки отрисовки фигур Shapely.
2. pip install shapely --no-binary shapely. Для линковки Shapely с GEOS ее необходимо скомпилировать с нуля, что и делает команда no-binary.
3. pip install cartopy. Установив все зависимости, мы вызываем Cartopy с помощью pip.

Ручная установка несколько хлопотна, поэтому в качестве альтернативы можно применить пакетный менеджер Conda.

11.2.2. Использование пакетного менеджера Conda

Conda, как и pip, является менеджером пакетов, способным скачивать и устанавливать внешние библиотеки. В отличие от pip эта программа может без проблем работать с зависимостями, написанными не на Python. При этом Conda обычно не входит в предустановленный набор программ системы, поэтому ее нужно скачивать отдельно с https://docs.conda.io/en/latest/miniconda.html. После этого можно будет без проблем установить Cartopy, выполнив conda install -c conda-forge cartopy.

К сожалению, применение Conda предполагает некоторые компромиссы. Установка этим менеджером новой библиотеки Python происходит в *изолированном виртуальном окружении*. В нем имеется собственная версия Python, которая отделена от основной версии, размещенной на машине пользователя. В связи с этим библиотека Cartopy устанавливается в виртуальное окружение, а не в основное. Это может вызвать путаницу при импорте Cartopy, особенно в блокнот Jupyter, поскольку он по умолчанию связан с основным окружением. Для добавления в него окружения Conda необходимо выполнить следующие две команды:

- conda install -c anaconda ipykernel;
- python -m ipykernel install --user --name=base.

В результате блокнот Jupyter сможет взаимодействовать с окружением Conda под названием base — это предустановленное имя окружения, создаваемого Conda.

Теперь при создании нового блокнота в раскрывающемся меню Jupyter можно выбрать окружение base (рис. 11.3), что дает возможность импортировать Cartopy.

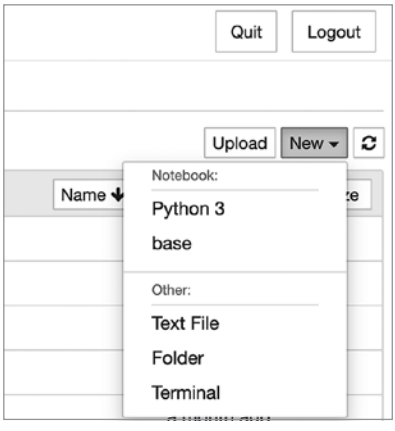


Рис. 11.3. Выбор окружения при создании блокнота. Окружение base можно выбрать из раскрывающегося меню, что позволит импортировать библиотеку Cartopy

ПРИМЕЧАНИЕ

Предустановленное виртуальное окружение Conda называется base. Однако Conda позволяет создавать и отслеживать несколько окружений. Для создания нового виртуального окружения new_env нужно выполнить из командной строки conda create -n new_env. Затем можно будет переключиться на новое окружение командой conda activate new_env. Выполнение conda activate base вернет нас обратно в base, где установлена Cartopy. При этом команда conda deactivate приведет к переключению на предустановленные в нашей машине настройки Python. Также можно проверить имя текущего окружения командой conda info.

Давайте подтвердим установку, выполнив `import cartopy` из блокнота Jupyter (листинг 11.7).

Листинг 11.7. Импорт библиотеки Cartopy

```
import cartopy
```

Установка Cartopy может оказаться несколько запутанной, но разобраться в ней стоит. Данная библиотека является лучшим и наиболее популярным инструментом визуализации карт для Python. Ну а теперь время заняться этой визуализацией.

11.2.3. Визуализация карт

Географическая карта — это двухмерное представление 3D-поверхности глобуса. Уплощение сферы глобуса — это процесс, называемый *проекцией*. Существует множество разных проекций карт — самая простая подразумевает наложение изображения глобуса на развернутый цилиндр, что дает 2D-карту, координаты (x, y) на которой в точности соответствуют широте и долготе.

ПРИМЕЧАНИЕ

В большинстве других проекций двухмерная сетка координат не соответствует сферическим координатам. В связи с этим требуется преобразовать координаты из одной системы в другую. С этой проблемой мы столкнемся в следующем разделе.

Данная техника называется *равнопромежуточной проекцией*. Для использования этой стандартной проекции в наших графиках импортируем PlateCarree из `cartopy.crs` (листинг 11.8).

ПРИМЕЧАНИЕ

Модуль cartopy.crs включает множество других видов проекции. Можно, к примеру, импортировать Orthographic, в результате чего вернется ортографическая проекция, на которой Земля представлена с перспективы смотрящего на нее из удаленной части Галактики.

Листинг 11.8. Импорт равнопромежуточной проекции

```
from cartopy.crs import PlateCarree
```

Класс `PlateCarree` можно использовать совместно с Matplotlib, чтобы визуализировать Землю. К примеру, выполнение `plt.axes(projection=PlateCarree()).coastlines()` приведет к отрисовке контуров семи континентов нашей планеты (листинг 11.9). Говоря точнее, `plt.axes(projection=PlateCarree())` инициализирует пользовательскую ось Matplotlib, позволяющую визуализировать карты. Далее метод `coastlines` очерчивает береговую линию отображенных континентов (рис. 11.4).

Листинг 11.9. Визуализация Земли с помощью Cartopy

```
plt.axes(projection=PlateCarree()).coastlines()
plt.show()
```

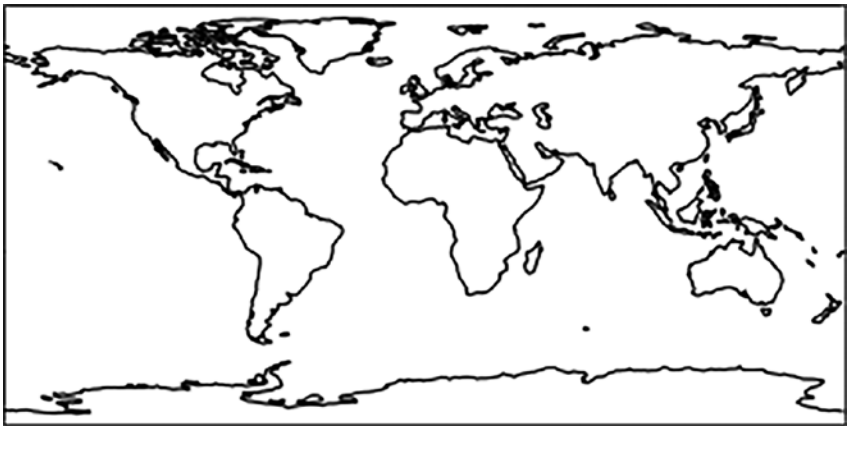


Рис. 11.4. Стандартная карта Земли с континентами и береговой линией

Карта у нас получилась небольшой. Увеличить ее можно с помощью функции Matplotlib `plt.figure`. Вызов `plt.figure(figsize=(width, height))` приведет к созданию изображения шириной `width` дюймов и высотой `height` дюймов. Код листинга 11.10 увеличивает размер рисунка до 12×8 дюймов, после чего генерирует карту мира (рис. 11.5).

ПРИМЕЧАНИЕ

Ввиду особенностей форматирования изображений фактические размеры рисунка в книге не соответствуют 12×8 дюймов.

Листинг 11.10. Визуализация увеличенной карты Земли

```
plt.figure(figsize=(12, 8))
plt.axes(projection=PlateCarree()).coastlines()
plt.show()
```

Создает более крупный рисунок шириной 12 дюймов и высотой 8 дюймов

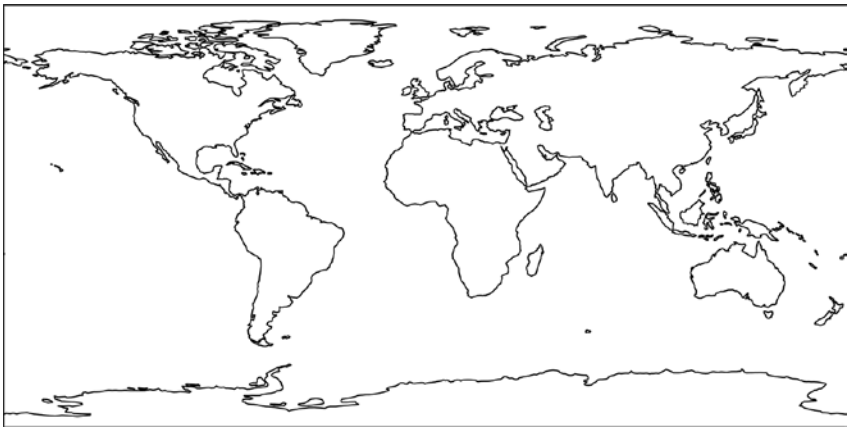


Рис. 11.5. Стандартная карта Земли с континентами и береговой линией. Размер карты был увеличен функцией Matplotlib plt.figure

Пока что наша карта выглядит пустой и невзрачной. Ее качество можно улучшить, вызвав `plt.axes(projection=PlateCarree()).stock_img()` (листинг 11.11). Этот метод раскрасит карту на основе топографической информации: океаны станут синими, а лесные массивы — зелеными (рис. 11.6).

Листинг 11.11. Раскрашивание карты Земли

```
fig = plt.figure(figsize=(12, 8))
plt.axes(projection=PlateCarree()).stock_img()
plt.show()
```

Наша цветная карта не включает границы, обозначающие береговую линию. Добавление этих границ дополнительно повысит качество карты. Однако не получится

добавить и цвет, и границы одной строкой кода. Для этого нужно будет выполнить следующие три строки (рис. 11.7).

1. `ax = plt.axes(projection=PlateCarree())`. Инициализирует пользовательскую ось Matplotlib, способную визуализировать карты. По стандартному соглашению эта ось присваивается переменной `ax`.
2. `ax.coastlines()`. Добавляет в рисунок береговые линии.
3. `ax.stock_img()`. Добавляет топографические цвета.

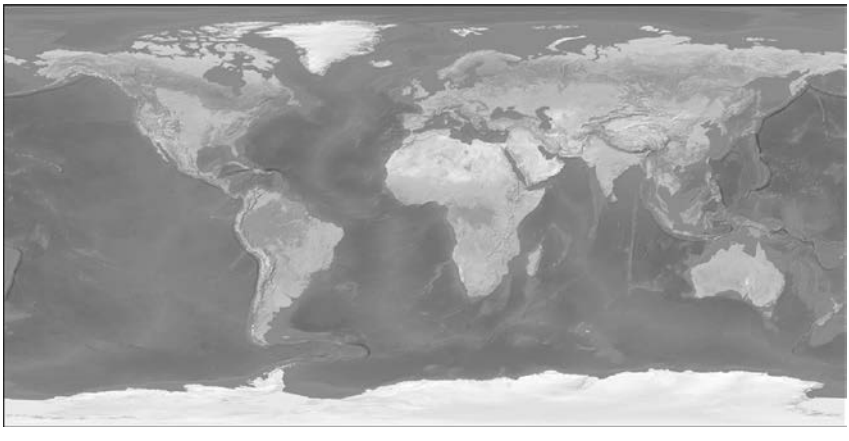


Рис. 11.6. Стандартная карта Земли, раскрашенная для выделения океанических и топографических элементов

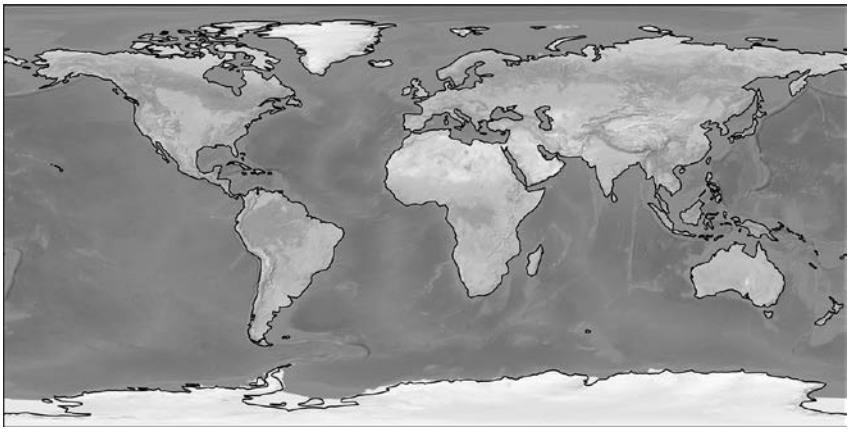


Рис. 11.7. Стандартная карта Земли, раскрашенная для выделения океанических и топографических деталей. Нанесенная дополнительно береговая линия более отчетливо показывает границы континентов

Теперь выполним все эти шаги для построения отчетливой цветной карты (листинг 11.12).

Листинг 11.12. Нанесение береговых линий и раскрашивание карты

```
plt.figure(figsize=(12, 8))
ax = plt.axes(projection=PlateCarree())
ax.coastlines()
ax.stock_img()
plt.show()
```

Имейте в виду, что `ax.stock_img()` при раскрашивании карты опирается на стоковое изображение Земли. Это изображение теряет в качестве при приближении карты (что мы вскоре сделаем). В качестве альтернативы можно раскрасить океаны и континенты с помощью метода `ax.add_feature`, который отрисовывает особые визуальные черты из модуля `cartopy.features`. К примеру, вызов `ax.add_feature(cartopy.feature.OCEAN)` раскрасит все океаны синим, а ввод `cartopy.feature.LAND` сделает все участки суши бежевыми. Далее мы раскрасим карту, используя эти расширенные возможности (листинг 11.13; рис. 11.8).

Листинг 11.13. Добавление цветов с помощью модуля feature

```
plt.figure(figsize=(12, 8))
ax = plt.axes(projection=PlateCarree())
ax.coastlines()
ax.add_feature(cartopy.feature.OCEAN)
ax.add_feature(cartopy.feature.LAND)
plt.show()
```

Продолжаем отображать береговую линию для повышения четкости изображения

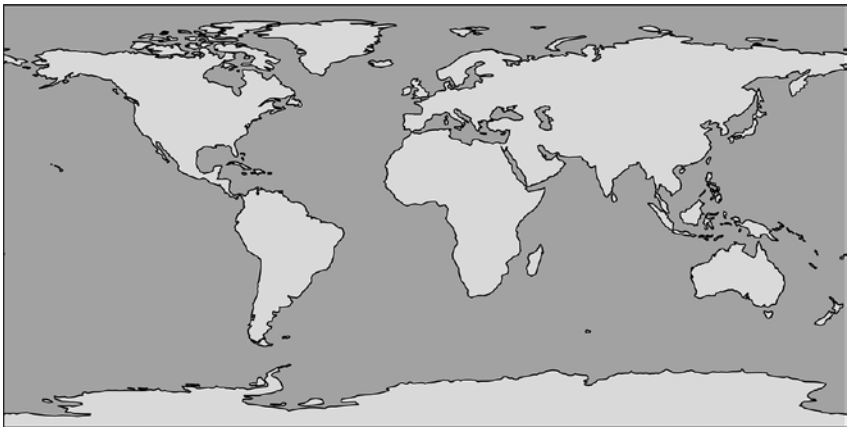


Рис. 11.8. Стандартная карта Земли, раскрашенная с помощью модуля feature

Сейчас на рисунке отсутствуют границы государств. В Cartopy они реализуются как одна из характеристик модуля `feature`. Добавить границы стран можно вызовом `ax.add_feature(cartopy.feature.BORDERS)` (листинг 11.14; рис. 11.9).

Листинг 11.14. Добавление на рисунок государственных границ

```
plt.figure(figsize=(12, 8))
ax = plt.axes(projection=PlateCarree())
ax.coastlines()
ax.add_feature(cartopy.feature.BORDERS)
ax.add_feature(cartopy.feature.OCEAN)
ax.add_feature(cartopy.feature.LAND)
plt.show()
```

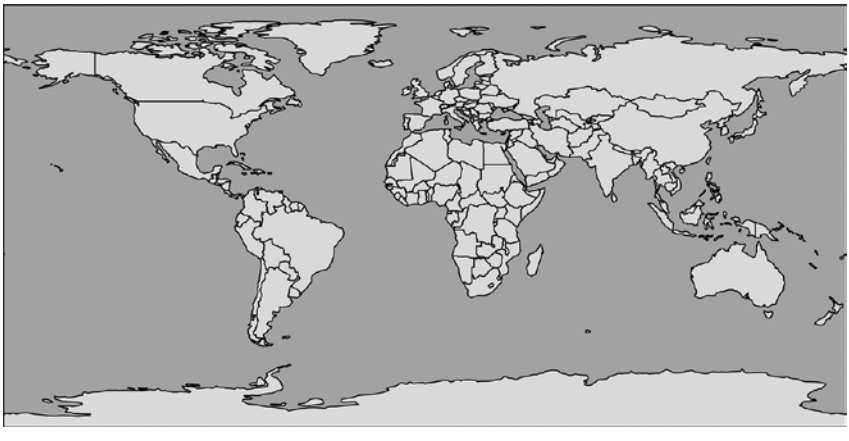


Рис. 11.9. Стандартная карта Земли с границами стран

Предположим, что нам дан список локаций, определенных парами значений широты и долготы. Эти локации можно отобразить на глобальной карте в виде стандартного точечного графика, вызвав `ax.scatter(longitudes, latitudes)`. Однако Matplotlib по умолчанию приближает наносимые на график рассеянные точки, что делает изображение неполным. Избежать этого можно вызовом `ax.set_global()`, который расширит отрисованное изображение до всех четырех краев карты мира. Код листинга 11.15 наносит на карту ряд географических точек. В целях упрощения на карте отражена лишь береговая линия (рис. 11.10).

Листинг 11.15. Отрисовка координат на карте

```
plt.figure(figsize=(12, 8))
coordinates = [(39.9526, -75.1652), (37.7749, -122.4194),
               (40.4406, -79.9959), (38.6807, -108.9769),
               (37.8716, -112.2727), (40.7831, -73.9712)]

latitudes, longitudes = np.array(coordinates).T
ax = plt.axes(projection=PlateCarree())
ax.scatter(longitudes, latitudes)
ax.set_global()
ax.coastlines()
plt.show()
```

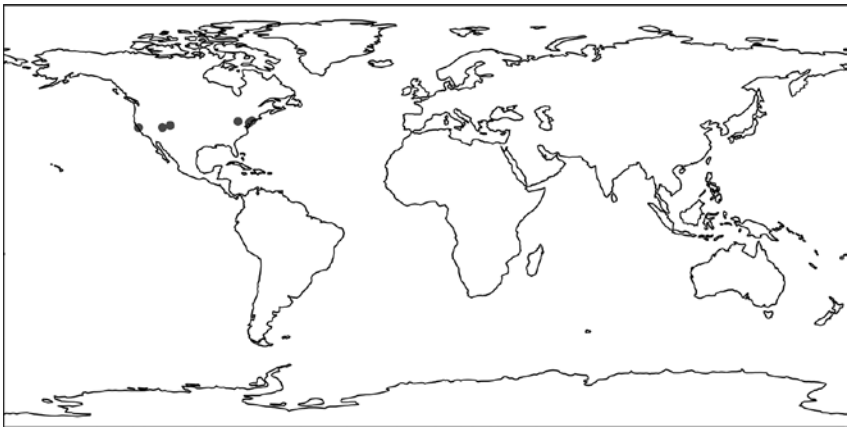


Рис. 11.10. Стандартная карта Земли с нанесенными координатами широты и долготы

ПРИМЕЧАНИЕ

Как уже говорилось, равнопромежуточная проекция дает 2D-сетку, в которой longitudes и latitudes можно отрисовать непосредственно на осях. Для других проекций так не получится — они потребуют преобразования longitudes и latitudes до генерации точечного графика. Вскоре мы разберем, как правильно выполнить это преобразование.

Все нанесенные точки находятся в рамках границ Северной Америки. Можно упростить карту, приблизив именно этот континент. Однако сначала нужно подстроить экстент карты, представляющий ее отображаемую географическую область. Экстент определяется прямоугольником, чьи углы позиционируются на дисплее в точках минимальной и максимальной широты и долготы. В Cartopy эти углы определяются с помощью четырехэлементного кортежа, имеющего вид `(min_lon, max_lon, min_lat, max_lat)`. Передача этого списка в `ax.set_extent` скорректирует границы отображаемой карты.

Теперь присвоим стандартный экстент Северной Америки переменной `north_america_extent`, после чего с помощью метода `ax.set_extent` приблизим этот континент (листинг 11.16). Мы повторно сгенерируем точечный график, на этот раз добавив цвет передачей в `ax.scatter` параметра `color='r'`. Кроме того, используем модуль `feature` для раскрашивания карты и добавления государственных границ (рис. 11.11).

Листинг 11.16. Добавление на карту координат Северной Америки

```
plt.figure(figsize=(12, 8))
ax = plt.axes(projection=PlateCarree())
north_america_extent = (-145, -50, 0, 90)
ax.set_extent(north_america_extent)
ax.scatter(longitudes, latitudes, color='r')
```

Экстент Северной Америки охватывает область между –145 и –50° долготы и между 0 и 90° широты

```
def add_map_features():
    ax.coastlines()
    ax.add_feature(cartopy.feature.BORDERS)
    ax.add_feature(cartopy.feature.OCEAN)
    ax.add_feature(cartopy.feature.LAND)

add_map_features()
plt.show()
```

Эта функция добавляет на карту стандартные черты. Она используется и в других листингах этого раздела

Рис. 11.11. Карта Северной Америки с нанесенными координатами широты и долготы

Мы успешно приблизили Северную Америку. Теперь еще увеличим масштаб, сфокусировавшись на Соединенных Штатах. К сожалению, равнопромежуточной проекции для этой цели будет недостаточно — данная техника при излишнем приближении любой страны начинает искажать карту.

Вместо этого мы используем *коническую равноугольную проекцию Ламберта*. В этой проекции поверх сферической Земли помещается конус. Его круглое основание покрывает область, которую мы хотим отобразить. Затем координаты в этой области проецируются на поверхность конуса. В завершение конус разворачивается, формируя 2D-карту. Однако двухмерные координаты этой карты не будут непосредственно равны широте и долготе.

В Cartopy есть класс `LambertConformal`, содержащийся в модуле `csr`. Выполнение `plt.axes(projection=LambertConformal())` приведет к построению осей, соответствующих равноугольной системе координат Ламберта. Далее передача экстента США в `ax.set_extent` увеличит соответствующую область карты. Код листинга 11.17 определяет `us_extent` и передает его в данный метод. Мы также отобразим на графике наши географические данные, но сначала нужно преобразовать `longitudes` и `latitudes` в координаты, совместимые с `LambertConformal`. Иными словами, необходимо преобразовать данные из PlateCarree-совместимых координат в нечто иное. Это можно сделать, передав `transform=PlateCarree()` в `ax.scatter()`. Давайте выполним это преобразование, чтобы визуализировать интересующие нас точки на карте США (рис. 11.12).

ПРИМЕЧАНИЕ

При первом выполнении этого кода Cartopy скачивает и устанавливает равноугольную проекцию Ламберта, поэтому потребуется наличие интернет-соединения.

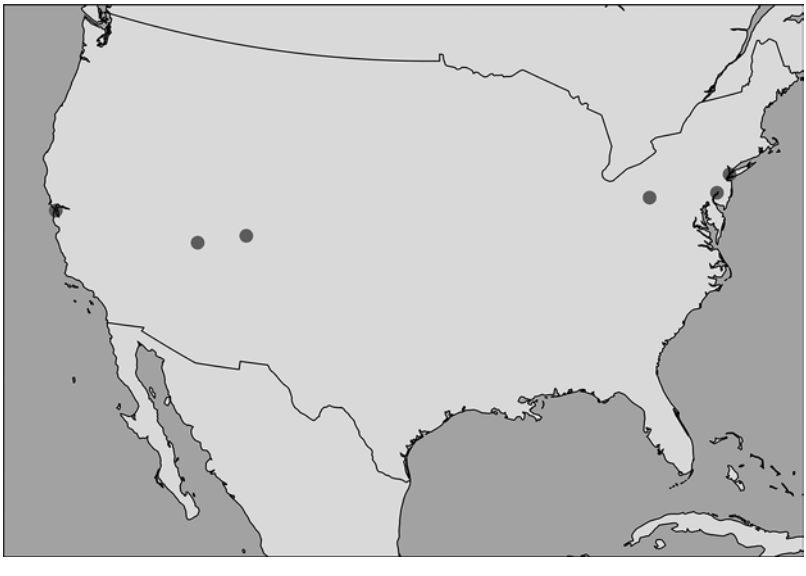


Рис. 11.12. Равноугольная проекция Ламберта территории США с нанесенными на нее координатами широты и долготы

Листинг 11.17. Построение координат США

```
from cartopy.crs import LambertConformal

plt.figure(figsize=(12, 8))
ax = plt.axes(projection=LambertConformal())
```

Импортирует равноугольную коническую проекцию Ламберта

Ось ax соответствует координатам LambertConformal

```
us_extent = (-120, -75, 20, 50)
ax.set_extent(us_extent)

ax.scatter(longitudes, latitudes, color='r',
           transform=PlateCarree(),
           s=100)

add_map_features()
plt.show()
```

Экстент США охватывает область между –120 и –75° долготы и между 20 и 50° широты

Преобразует широту и долготу из PlateCarree-совместимых координат в ax.projection-совместимые, где ax.projection равняется LambertConformal

Параметр s устанавливает размер наносимого на карту маркера. Мы увеличиваем этот размер, чтобы было лучше видно

Полученная карта США выглядит пустынной. Добавим на нее границы штатов вызовом `ax.add_feature(cartopy.feature.STATES)` (листинг 11.18; рис. 11.13).

Листинг 11.18. Построение карты США, включая границы штатов

```
fig = plt.figure(figsize=(12, 8))
ax = plt.axes(projection=LambertConformal())
ax.set_extent(us_extent)

ax.scatter(longitudes, latitudes, color='r',
           transform=PlateCarree(),
           s=100)

ax.add_feature(cartopy.feature.STATES)
add_map_features()
plt.show()
```

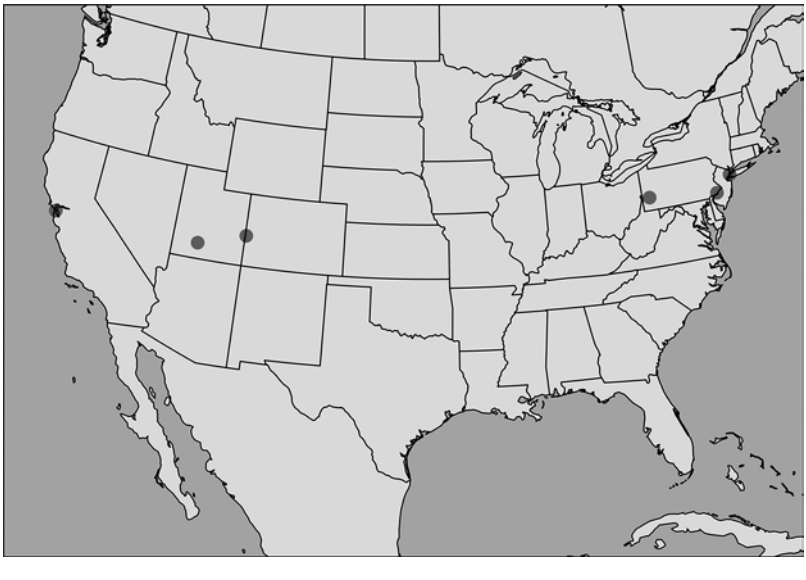


Рис. 11.13. Равноугольная проекция Ламберта территории США, включая границы штатов

СТАНДАРТНЫЕ МЕТОДЫ CARTOPY

- `ax = plt.axes(projection=PlateCarree())` — создает пользовательскую ось Matplotlib для генерации карты на основе равнопромежуточной проекции.
- `ax = plt.axes(projection=LambertConformal())` — создает пользовательскую ось Matplotlib для генерации карты на основе равноугольной конической проекции Ламберта.
- `ax.coastlines()` — наносит на карту береговую линию.
- `ax.add_feature(cartopy.feature.BORDERS)` — наносит на карту государственные границы.
- `ax.add_feature(cartopy.feature.STATES)` — наносит на карту границы штатов США.
- `ax.stock_img()` — раскрашивает карту, используя топографическую информацию.
- `ax.add_feature(cartopy.feature.OCEAN)` — закрашивает все океаны синим.
- `ax.add_feature(cartopy.feature.LAND)` — закрашивает все массивы суши бежевым.
- `ax.set_global()` — расширяет построенное изображение до всех четырех границ карты мира.
- `ax.set_extent(min_lon, max_lon, min_lat, max_lat)` — корректирует экстент отрисовываемой карты, то есть определяет, какая область должна отображаться, используя минимальные и максимальные значения широты и долготы.
- `ax.scatter(longitudes, latitudes)` — наносит на карту координаты широты и долготы.
- `ax.scatter(longitudes, latitudes, transform=PlateCarree())` — наносит на карту координаты широты и долготы, преобразуя при этом координаты из `PlateCarree`-совместимых в иные, например в равноугольные конические координаты Ламберта.

Cartopy позволяет отрисовывать на карте любые локации. Для этого нам нужны лишь их значения широты и долготы. Конечно же, знать эти географические координаты необходимо до того, как наносить их на карту, поэтому приходится сопоставлять названия локаций с их географическими характеристиками. Сопоставление реализуется с помощью библиотеки отслеживания локаций GeoNamesCache.

11.3. ОТСЛЕЖИВАНИЕ ЛОКАЦИЙ С ПОМОЩЬЮ GEONAMESCACHE

База данных GeoNames (http://geonames.org) представляет собой прекрасный источник географической информации. Она содержит более 11 млн названий мест всех стран мира. Кроме того, GeoNames хранит ценную информацию, такую как значения широты и долготы. Поэтому можно использовать эту базу данных для определения точного географического местоположения городов и стран, встречающихся в тексте.

Как же получить доступ к данным GeoNames? Как вариант, можно вручную скачать их дамп (http://download.geonames.org/export/dump), спарсить его и сохранить полученную структуру данных. Это трудно. К счастью, кто-то уже проделал всю сложную работу за нас, создав библиотеку GeoNamesCache.

GeoNamesCache предназначена для эффективного извлечения данных о континентах, странах, городах, а также округах и штатах США. Эта библиотека предоставляет шесть простых в использовании методов, позволяющих получать доступ к данным о местоположении: `get_continents`, `get_countries`, `get_cities`, `get_countries_by_name`, `get_cities_by_name` и `get_us_counties`. Установим эту библиотеку и подробнее познакомимся с ее применением. Начнем с инициализации объекта отслеживания местоположения GeoNamesCache (листинг 11.19).

ПРИМЕЧАНИЕ

Для установки библиотеки GeoNamesCache выполните в терминале команду pip install geonamescache.

Листинг 11.19. Инициализация объекта GeonamesCache

```
from geonamescache import GeonamesCache
gc = GeonamesCache()
```

Далее используем объект `gc` для анализа семи континентов. Мы выполним `gc.get_continents()` для извлечения словаря, содержащего информацию о континентах, после чего изучим структуру этого словаря, выводя его ключи (листинг 11.20).

Листинг 11.20. Получение из GeoNamesCache информации о всех семи континентах

```
continents = gc.get_continents()
print(continents.keys())
```

```
dict_keys(['AF', 'AS', 'EU', 'NA', 'OC', 'SA', 'AN'])
```

Ключи этого словаря представляют собой сокращенные коды названий континентов, в которых *Африка* обозначена `'AF'`, а *Северная Америка* — `'NA'`. Проверим значения, сопоставленные с каждым ключом, передав код для *Северной Америки*.

ПРИМЕЧАНИЕ

continents — это вложенный словарь, в котором семь ключей верхнего уровня отображаются в соответствующие содержимому структуры словаря. Код листинга 11.21 выводит эти ключи, хранящиеся в словаре continents['NA'].

Листинг 11.21. Получение из GeoNamesCache информации о Северной Америке

```
north_america = continents['NA']
print(north_america.keys())

dict_keys(['lng', 'geonameId', 'timezone', 'bbox', 'toponymName',
'asciiName', 'astergdem', 'fcl', 'population', 'wikipediaURL',
'adminName5', 'srtm3', 'adminName4', 'adminName3', 'alternateNames',
'cc2', 'adminName2', 'name', 'fclName', 'fcodeName', 'adminName1',
'lat', 'fcode', 'continentCode'])
```

Многие из элементов данных `north_america` представляют различные схемы именования Северо-Американского континента (листинг 11.22). Подобная информация особой пользы не несет.

Листинг 11.22. Вывод схем именования Северной Америки

```
for name_key in ['name', 'asciiName', 'toponymName']:
     print(north_america[name_key])

North America
North America
North America
```

Другие элементы более полезны. Например, ключи `'lat'` и `'lng'` отображаются в широту и долготу центральной локации Северной Америки. Визуализируем эту локацию на карте (листинг 11.23; рис. 11.14).

Листинг 11.23. Нанесение на карту центральных координат Северной Америки

```
latitude = float(north_america['lat'])
longitude = float(north_america['lng'])
```

Ключи lat и lng отображаются в широту и долготу центральной точки Северной Америки

```
plt.figure(figsize=(12, 8))
ax = plt.axes(projection=PlateCarree())
ax.set_extent(north_america_extent)
ax.scatter([longitude], [latitude], s=200)
add_map_features()
plt.show()
```

Рис. 11.14. Координаты центральной точки Северной Америки

11.3.1. Получение информации о странах

Возможность обращаться к данным континентов очень полезна, хотя в первую очередь нас интересует анализ городов и стран. Анализировать страны можно с помощью метода `get_countries`. Он возвращает словарь, двухсимвольные ключи которого кодируют названия 252 стран. Как и в случае континентов, коды стран отражают их сокращенные названия. Например, у *Канады* код `'CA'`, а у *США* — `'US'`. Выполнение `gc.get_countries()['US']` приведет к возврату словаря, содержащего полезные данные о США (листинг 11.24).

Листинг 11.24. Получение из GeoNamesCache данных США

```
countries = gc.get_countries()
num_countries = len(countries)
print(f"GeonamesCache holds data for {num_countries} countries.")

us_data = countries['US']
print("The following data pertains to the United States:")
print(us_data)
```

```
GeonamesCache holds data for 252 countries.
The following data pertains to the United States:
{'geonameid': 6252001,
'name': 'United States',
'iso': 'US',
'iso3': 'USA',
'isonumeric': 840,
'fips': 'US',
'continentcode': 'NA',
'capital': 'Washington',
'areakm2': 9629091,
'population': 310232863,
'tld': '.us',
'currencycode': 'USD',
'currencyname': 'Dollar',
'phone': '1',
'postalcoderegex': '^\\d{5}(-\\d{4})?$',
'languages': 'en-US,es-US,haw,fr',
'neighbours': 'CA,MX,CU'}
```

Код континента, на котором расположены США

Столица США

Площадь США, в квадратных километрах

Население США

Валюта США

Местные языки США

Соседние с США территории

Выводимые данные включают множество полезных элементов, таких как столица страны, ее валюта, площадь, местные языки и количество населения. К сожалению, GeoNamesCache не может дать центральные координаты широты и долготы, связанные с площадью страны. Однако, как мы вскоре узнаем, центральность страны можно оценить при помощи координат городов.

Кроме того, полезная информация содержится в элементе `'neighbours'` каждой страны. Ключ `'neighbours'` отображается в разделенную запятыми строку кодов стран, означающих соседние территории. Дополнительные подробности о каждом соседе можно получить, разделив данную строку и передав ее коды в словарь `'countries'` (листинг 11.25).

Листинг 11.25. Получение соседних стран

```
us_neighbors = us_data['neighbours']
for neighbor_code in us_neighbors.split(','):
    print(countries[neighbor_code]['name'])
```

```
Canada
Mexico
Cuba
```

По данным GeoNamesCache, непосредственными соседями США являются Канада, Мексика и Куба. Думаю, все согласятся с первыми двумя странами, а вот соседство Кубы вызывает сомнения. Это государство не граничит напрямую с США. К тому же если это островное государство Карибского бассейна действительно является соседом, то почему в список не вошло также Гаити? Самый же главный вопрос в том, как вообще в этот список попала Куба? Дело в том, что GeoNames — это коллективный проект, реализуемый сообществом редакторов (подобно «Википедии») и ориентированный на сбор справочной информации о различных точках земного шара. В какой-то момент редактор решил, что Куба является соседом США. Кто-то может с этим решением не согласиться, поэтому важно помнить, что GeoNames не является эталонным хранилищем информации о различных локациях планеты. Это лишь инструмент для быстрого анализа больших объемов соответствующих данных. Некоторые из этих данных могут быть неточными, поэтому используйте GeoNamesCache внимательно.

В метод `get_countries` нужно передать двухсимвольный код страны. Однако кодов большинства стран мы не знаем. К счастью, можно запрашивать все государства по названию с помощью метода `get_countries_by_names` (возвращающий словарь), элементы которого представляют названия стран, а не их коды (листинг 11.26).

Листинг 11.26. Получение информации о странах по их названию

```
result = gc.get_countries_by_names()['United States']
assert result == countries['US']
```

11.3.2. Получение информации о городах

Теперь переключимся на анализ городов. Метод `get_cities` возвращает словарь, ключи которого представляют уникальные ID, отображающиеся обратно в данные городов. Код листинга 11.27 выводит эти данные для одного города.

Листинг 11.27. Получение cities из GeoNamesCache

```
cities = gc.get_cities()
num_cities = len(cities)
print(f"GeoNamesCache holds data for {num_cities} total cities")
city_id = list(cities.keys())[0]
print(cities[city_id])
```

`cities` — это словарь, сопоставляющий уникальный `city_id` с географической информацией.

```
{'geonameid': 3041563,   <-- Уникальный ID города
'name': 'Andorra la Vella',   <-- Название города
'latitude': 42.50779,   <-- Широта
'longitude': 1.52109,   <-- Долгота
'countrycode': 'AD',   <-- Код страны, в которой находится город
'population': 20430,   <-- Население
'timezone': 'Europe/Andorra'}   <-- Часовой пояс
```

Данные каждого города содержат его название, широту и долготу, количество населения и кодовое обозначение страны, в которой он расположен. С помощью этого кода можно создать новое сопоставление между страной и всеми ее городами. Давайте изолируем и подсчитаем все города США, представленные в GeoNamesCache (листинг 11.28).

ПРИМЕЧАНИЕ

Как уже говорилось, база GeoNames неидеальна. Некоторые города США сейчас в ней могут отсутствовать. Тем не менее с течением времени их постепенно добавляют. В связи с этим по мере обновления библиотеки получаемое количество городов может изменяться.

Листинг 11.28. Получение из GeoNamesCache городов США

```
us_cities = [city for city in cities.values()
             if city['countrycode'] == 'US']
num_us_cities = len(us_cities)
print(f"GeoNamesCache holds data for {num_us_cities} US cities.")

GeoNamesCache holds data for 3248 US cities
```

GeoNamesCache содержит информацию о более чем 3000 городов США. В словаре данных каждого города хранятся его широта и долгота. Давайте найдем среднюю широту и долготу по США, которая аппроксимируется к центральным координатам страны.

Имейте в виду, что аппроксимация будет неидеальной. Вычисленное среднее не учитывает кривизну Земли и неточно взвешено по расположению городов. Непропорционально большое число городов США находится рядом с Атлантическим океаном, в связи с чем оценочная средняя точка оказывается смещена на восток. В коде листинга 11.29 мы аппроксимируем и отрисовываем эту центральную точку США, понимая, что полученная аппроксимация будет неидеальной (рис. 11.15).

Листинг 11.29. Аппроксимация центральных координат США

```
center_lat = np.mean([city['latitude']
                      for city in us_cities])
center_lon = np.mean([city['longitude']
                      for city in us_cities])

fig = plt.figure(figsize=(12, 8))
ax = plt.axes(projection=LambertConformal())
ax.set_extent(us_extent)
ax.scatter([center_lon], [center_lat], transform=PlateCarree(), s=200)
ax.add_feature(cartopy.feature.STATES)
add_map_features()
plt.show()
```

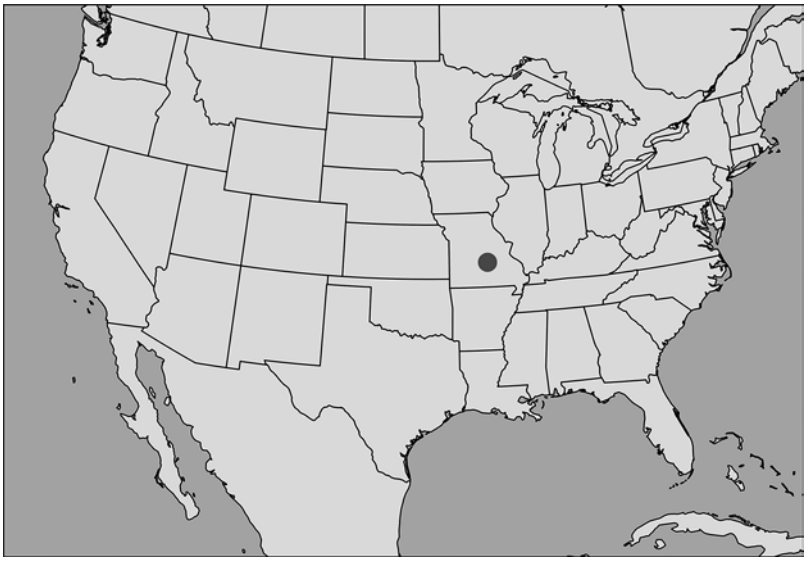


Рис. 11.15. Центральная локация США аппроксимируется путем усреднения координат каждого города США, содержащегося в GeoNamesCache. Эта аппроксимация немного смещена на восток

Метод `get_cities` подходит для перебора информации о городах, но не получения их по названиям. Для поиска по названию нужно использовать `get_cities_by_name` (листинг 11.30). Этот метод получает на входе название города и возвращает списки данных для всех городов с таким именем.

Листинг 11.30. Получение городов по названию

```
matched_cities_by_name = gc.get_cities_by_name('Philadelphia')
print(matched_cities_by_name)

[{'4560349': {'geonameid': 4560349, 'name': 'Philadelphia',
'latitude': 39.95233, 'longitude': -75.16379, 'countrycode': 'US',
'population': 1567442, 'timezone': 'America/New_York'}}]
```

Метод `get_cities_by_name` может вернуть более одного города, поскольку названия городов не всегда оказываются уникальными. Например, GeoNamesCache содержит шесть разных городов Сан-Франциско, расположенных в пяти странах. Вызов `gc.get_cities_by_name('San Francisco')` вернет данные для всех Сан-Франциско. Код листинга 11.31 перебирает эти данные и выводит страну, в которой находится каждый из них.

Листинг 11.31. Получение нескольких городов с одинаковым названием

```
matched_cities_list = gc.get_cities_by_name('San Francisco')

for i, san_francisco in enumerate(matched_cities_list):
    city_info = list(san_francisco.values())[0]
    country_code = city_info['countrycode']
    country = countries[country_code]['name']
    print(f"The San Francisco at index {i} is located in {country}")
```

```
The San Francisco at index 0 is located in Argentina
The San Francisco at index 1 is located in Costa Rica
The San Francisco at index 2 is located in Philippines
The San Francisco at index 3 is located in Philippines
The San Francisco at index 4 is located in El Salvador
The San Francisco at index 5 is located in United States
```

Нередко бывает, что одно название соответствует нескольким городам и выбор среди них может вызывать сложности. Предположим, что кто-то запрашивает поисковый движок о погоде в Афинах. В таком случае механизм поиска должен выбрать между Афинами в Огайо и Афинами в Греции. Для правильного определения интересующей локации необходима дополнительная информация. Находится ли пользователь в Огайо? Может, он планирует путешествие в Грецию? Не имея этого контекста, поисковый движок вынужден угадывать. Как правило, более вероятной догадкой оказывается город с наибольшим населением. Со статистической точки зрения более густонаселенные города чаще оказываются предметом обсуждения. Выбор города с бóльшим числом жителей не обязательно окажется верным во всех случаях, но все же этот вариант лучше, чем абсолютно случайный выбор. Посмотрим, что произойдет при нанесении на карту самого густонаселенного Сан-Франциско (листинг 11.32; рис. 11.16).

Листинг 11.32. Нанесение на карту самого густонаселенного Сан-Франциско

```
best_sf = max(gc.get_cities_by_name('San Francisco'),
              key=lambda x: list(x.values())[0]['population'])
sf_data = list(best_sf.values())[0]
sf_lat = sf_data['latitude']
sf_lon = sf_data['longitude']

plt.figure(figsize=(12, 8))
ax = plt.axes(projection=LambertConformal())
ax.set_extent(us_extent)
ax.scatter(sf_lon, sf_lat, transform=PlateCarree(), s=200)
add_map_features()
ax.text(sf_lon + 1, sf_lat, ' San Francisco', fontsize=16,
    transform=PlateCarree())
plt.show()
```

Метод ax.text позволяет написать San Francisco в точке, соответствующей указанным широте и долготе. Мы немного смещаем надпись вправо, чтобы избежать ее наложения на саму точку, обозначающую город. При этом на данной карте не нанесены границы штатов, чтобы можно было более наглядно отображать текст

Рис. 11.16. Среди шести Сан-Франциско, хранящихся в базе данных GeoNamesCache, город с наибольшим населением, как и ожидалось, находится в Калифорнии

Выбор Сан-Франциско с наибольшим населением приводит к возврату известного города в Калифорнии, а не иных, менее популярных локаций, находящихся за пределами США.

ТИПИЧНЫЕ МЕТОДЫ GEONAMESCACHE

- `gc = GeonamesCache()` — инициализирует объект `GeonamesCache`.
- `gc.get_continents()` — возвращает словарь, сопоставляющий ID континентов с данными о них.
- `gc.get_countries()` — возвращает словарь, сопоставляющий ID стран с данными о них.
- `gc.get_countries_by_names()` — возвращает словарь, сопоставляющий названия стран с данными о них.
- `gc.get_cities()` — возвращает словарь, сопоставляющий ID городов с данными о них.
- `gc.get_cities_by_name(city_name)` — возвращает список городов с названием `city_name`.

11.3.3. Ограничения библиотеки GeoNamesCache

GeoNamesCache — полезный инструмент, но у библиотеки есть и значительные недостатки. Так, в ней зарегистрированы далеко не все города. Некоторые малонаселенные пункты в сельской местности, будь то в США или Китае, в базе данных отсутствуют. Более того, метод `get_cities_by_name` сопоставляет только одну версию названия города с его географическими данными. В связи с этим возникает проблема для таких городов, как Нью-Йорк, которые имеют не один вариант названия (листинг 11.33).

Листинг 11.33. Получение New York City из GeoNamesCache

```
for ny_name in ['New York', 'New York City']:
    if not gc.get_cities_by_name(ny_name):
        print(f"'{ny_name}' is not present in the GeoNamesCache database")
    else:
        print(f"'{ny_name}' is present in the GeoNamesCache database")

'New York' is not present in the GeoNamesCache database
'New York City' is present in the GeoNamesCache database
```

Однозначное сопоставление названия с городом оказывается особенно проблематичным из-за присутствия во многих названиях диакритических знаков. *Диакритические знаки* — это знаки ударения, обозначающие правильное произношение слов, звучащих не по-английски. Они нередко встречаются в названиях городов — например, Cañon City, штат Колорадо, и Hagåtña, штат Гуам (листинг 11.34).

Листинг 11.34. Получение из GeoNamesCache городов с ударением в названии

```
print(gc.get_cities_by_name(u'Cañon City'))
print(gc.get_cities_by_name(u'Hagåtña'))

[{'5416005': {'geonameid': 5416005, 'name': 'Cañon City',
'latitude': 38.44098, 'longitude': -105.24245, 'countrycode': 'US',
'population': 16400, 'timezone': 'America/Denver'}}]
[{'4044012': {'geonameid': 4044012, 'name': 'Hagåtña',
'latitude': 13.47567, 'longitude': 144.74886, 'countrycode': 'GU',
'population': 1051, 'timezone': 'Pacific/Guam'}}]
```

Сколько названий городов, хранящихся в GeoNamesCache, содержат в себе диакритические знаки? Выяснить это можно с помощью функции `unidecode` из внешней библиотеки Unidecode (листинг 11.35). Эта функция исключает из входного текста все знаки ударения. Проверяя отличие выходного текста от входного, можно определить все названия городов, содержащие знаки ударения.

ПРИМЕЧАНИЕ

Для установки библиотеки Unidecode выполните из терминала pip install Unidecode.

Листинг 11.35. Подсчет всех названий городов из GeoNamesCache с ударением

```
from unidecode import unidecode
accented_names = [city['name'] for city in gc.get_cities().values()
                  if city['name'] != unidecode(city['name'])]
num_accented_cities = len(accented_names)

print(f"An example accented city name is '{accented_names[0]}'") print(f"{num_
accented_cities} cities have accented names")

An example accented city name is 'Khawr Fakka- n'
4896 cities have accented names
```

Примерно 5000 сохраненных в GeoNamesCache городов содержат в своих названиях диакритические знаки. Обычно эти города в публикациях указываются без ударений. Один из способов обеспечить сопоставление для всех подобных городов — создать словарь их альтернативных названий (листинг 11.36). В нем лишенный ударений вывод `unidecode` будет отображен обратно в названия с ударениями.

Листинг 11.36. Удаление знаков ударения из названий городов

```
alternative_names = {unidecode(name): name
                     for name in accented_names}
print(gc.get_cities_by_name(alternative_names['Hagatna']))

[{'4044012': {'geonameid': 4044012, 'name': 'Hagåtña',
'latitude': 13.47567, 'longitude': 144.74886, 'countrycode': 'GU',
'population': 1051, 'timezone': 'Pacific/Guam'}}]
```

Теперь можно сопоставлять лишенные ударений ключи словарей со всем входным текстом, передавая значения из словаря с ударениями в GeoNamesCache при каждом обнаружении совпадения ключа (листинг 11.37).

Листинг 11.37. Поиск в тексте названий городов без знаков ударения

```
text = 'This sentence matches Hagatna'
for key, value in alternative_names.items():
    if key in text:
        print(gc.get_cities_by_name(value))
        break

[{'4044012': {'geonameid': 4044012, 'name': 'Hagåtña',
'latitude': 13.47567, 'longitude': 144.74886, 'countrycode': 'GU',
 'population': 1051, 'timezone': 'Pacific/Guam'}}]
```

GeoNamesCache позволяет с легкостью отслеживать локации вместе с их географическими координатами. С помощью этой библиотеки можно также находить указанные названия локаций в любом входном тексте. Однако поиск названий в тексте оказывается не столь простой задачей. Если мы хотим подобающим образом

сопоставить названия локаций, то нам нужно освоить способы сопоставления текста Python, а также научиться избегать подводных камней.

ПРИМЕЧАНИЕ

Заключительный раздел предназначен для тех читателей, кто не знаком с базовым сопоставлением строк и регулярными выражениями. Если вы уже освоили эти приемы, то можете смело его пропустить.

11.4. СОПОСТАВЛЕНИЕ С НАЗВАНИЯМИ ЛОКАЦИЙ В ТЕКСТЕ

В Python можно легко определить, является ли строка частью другой строки или содержит начало строки некий предопределенный текст (листинг 11.38).

Листинг 11.38. Базовое сопоставление строк

```
assert 'Boston' in 'Boston Marathon'
assert 'Boston Marathon'.startswith('Boston')
assert 'Boston Marathon'.endswith('Boston') == False
```

К сожалению, базовый синтаксис строк в Python весьма ограничен. К примеру, в нем нет прямого строкового метода для выполнения безразличных к регистру сравнений. Кроме того, строковые методы в Python не могут непосредственно различать несколько символов в строке и части фраз в предложении. Поэтому, желая определить, имеется ли в предложении фрагмент `'in a'`, мы не можем положиться на базовое сопоставление, так как рискуем неверно сопоставить последовательности символов вроде `'sin apple'` или `'win attached'` (листинг 11.39).

Листинг 11.39. Ошибки при базовом сопоставлении фрагментов строк

```
assert 'in a' in 'sin apple'
assert 'in a' in 'win attached'
```

Для преодоления этих ограничений задействуем встроенную в Python библиотеку обработки регулярных выражений `re`. *Регулярные выражения* — это кодируемые в строковом виде паттерны, которые можно сравнивать с неким текстом. Подобные паттерны могут представлять собой как простые копии строк, так и невероятно сложные формулировки, расшифровать которые способны немногие. В текущем разделе мы сосредоточимся на составлении и сопоставлении простых регулярных выражений.

Бóльшую часть сопоставлений регулярных выражений в Python можно реализовать с помощью функции `re.search`. Она получает два входных элемента: паттерн регулярного выражения и текст, с которым он сопоставляется. При обнаружении

совпадения функция возвращает объект `Match`, в противном случае — `None`. Объект `Match` содержит методы `start` и `end`. Они возвращают соответственно начальный и конечный индексы совпавшей строки текста (листинг 11.40).

Листинг 11.40. Сопоставление строк с помощью регулярных выражений

```
import re
regex = 'Boston'
random_text = 'Clown Patty'
match = re.search(regex, random_text)
assert match is None

matchable_text = 'Boston Marathon'
match = re.search(regex, matchable_text)
assert match is not None
start, end = match.start(), match.end()
matched_string = matchable_text[start: end]
assert matched_string == 'Boston'
```

Кроме того, с помощью `re.search` можно без проблем выполнять сопоставление строк, нечувствительных к регистру. Нужно просто передать `re.IGNORECASE` в качестве добавочного параметра `flags` (листинг 11.41).

Листинг 11.41. Сопоставление без учета регистра с помощью регулярных выражений

```
for text in ['BOSTON', 'boston', 'BoSTOn']:
    assert re.search(regex, text, flags=re.IGNORECASE) is not None
```

Тот же результат можно получить, передав в re.search параметр flags=re.I

Регулярные выражения позволяют сопоставлять конкретные слова, используя обнаружение границ слов (листинг 11.42). Добавление в строку регулярного выражения паттерна `\b` обозначит начальную и конечную точки искомых слов (согласно пробелам и пунктуации). Однако, поскольку обратный слеш в стандартном лексиконе Python является особым символом, необходимо принять меры, чтобы он интерпретировался как обычный знак. Для этого нужно либо добавить к нему еще один обратный слеш (довольно хлопотный подход), либо поставить в начале строки литерал `r`. Второе решение гарантирует, что регулярное выражение в ходе анализа будет рассматриваться как типичная строка.

Листинг 11.42. Сопоставление границ слов с помощью регулярных выражений

```
for regex in ['\\bin a\\b', r'\bin a\b']:
    for text in ['sin apple', 'win attached']:
        assert re.search(regex, text) is None

    text = 'Match in a string'
    assert re.search(regex, text) is not None
```

Теперь разберем более сложный вариант сопоставления. В нем мы будем выполнять сравнение с предложением `f'I visited {city} yesterday`, в котором `{city}` представляет одну из трех возможных локаций: Бостон, Филадельфию или Сан-Франциско (листинг 11.43). Правильным синтаксисом регулярного выражения для выполнения данного сопоставления будет `r'I visited\b(Boston|Philadelphia|San Francisco)\b yesterday'`.

ПРИМЕЧАНИЕ

Вертикальная черта | обозначает условие ИЛИ. Она говорит о том, что регулярное выражение нужно сопоставлять с одним из трех городов из списка. Кроме того, область сопоставляемых городов ограничивается скобками, без которых диапазон сопоставляемого текста вышел бы за пределы 'San Francisco' вплоть до 'San Francisco yesterday'.

Листинг 11.43. Сопоставление с несколькими городами при помощи регулярных выражений

```
regex = r'I visited \b(Boston|Philadelphia|San Francisco)\b yesterday.'
assert re.search(regex, 'I visited Chicago yesterday.') is None

cities = ['Boston', 'Philadelphia', 'San Francisco']
for city in cities:
    assert re.search(regex, f'I visited {city} yesterday.') is not None
```

Наконец, нужно рассмотреть, как эффективно выполнять поиск регулярного выражения. Предположим, мы хотим сопоставить регулярное выражение со 100 строками. Для каждого сопоставления `re.search` преобразует регулярное выражение в Python `PatternObject`. Каждое такое преобразование требует немалых вычислительных затрат. Лучше будет выполнить его всего раз, используя команду `re.compile`, которая вернет скомпилированный объект `PatternObject`. Затем можно будет задействовать встроенный в этот объект метод `search`, избежав любой дополнительной компиляции (листинг 11.44).

ПРИМЕЧАНИЕ

Если мы захотим использовать скомпилированный паттерн для сопоставления без учета регистра, то нужно будет передать в re.compile параметр flags=re.IGNORECASE.

Листинг 11.44. Сопоставление строк с помощью скомпилированного регулярного выражения

```
compiled_re = re.compile(regex)
text = 'I visited Boston yesterday.'
for i in range(1000):
    assert compiled_re.search(text) is not None
```

СТАНДАРТНЫЕ ПРИЕМЫ СОПОСТАВЛЕНИЯ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

- `match = re.search(regex, text)` — если `regex` присутствует в `text`, то возвращает объект `Match`, в противном случае — `None`.
- `match = re.search(regex, text, flags=re.IGNORECASE)` — если `regex` присутствует в `text`, то возвращает объект `Match`, в противном случае — `None`. Сопоставление выполняется без учета регистра.
- `match.start()` — возвращает начальный индекс регулярного выражения, совпавшего с входным текстом.
- `match.end()` — возвращает конечный индекс регулярного выражения, совпавшего с входным текстом.
- `compiled_regex = re.compile(regex)` — преобразует строку `regex` в скомпилированный объект, соответствующий паттерну.
- `match = compiled_regex.search(text)` — использует встроенный в скомпилированный объект метод `search` для сопоставления регулярного выражения с `text`.
- `re.compile('Boston')` — компилирует регулярное выражение для сопоставления с текстом строки `'Boston'`.
- `re.compile('Boston', flags=re.IGNORECASE)` — компилирует регулярное выражение для сопоставления с текстом строки `'Boston'` без учета регистра.
- `re.compile('\\bBoston\\b')` — компилирует регулярное выражение для сопоставления с текстом слова `'Boston'`. Для выполнения точного сопоставления используются границы слова.
- `re.compile(r'\bBoston\b')` — компилирует регулярное выражение для сопоставления с текстом слова `'Boston'`. Входное регулярное выражение рассматривается как простая строка, поскольку использован литерал `r`. Это избавляет от необходимости добавлять дополнительный обратный слеш в разграничитель слов `\b`.
- `re.compile(r'\b(Boston|Chicago)\b')` — компилирует регулярное выражение для сопоставления с текстом слова `'Boston'` или `'Chicago'`.

Сопоставление регулярных выражений позволит нам отыскать в тексте названия локаций, поэтому модуль `re` окажется незаменим для решения третьего практического задания.

РЕЗЮМЕ

- Кратчайшее расстояние между наземными точками прокладывается по сферической поверхности нашей планеты. Это *расстояние по ортодромии* можно вычислить с помощью серии известных тригонометрических операций.
- Широта и долгота представляют *координаты сферы*. Ими измеряется угловая позиция точки на поверхности Земли относительно осей *X* и *Y*.
- Отобразить широту и долготу на карте можно с помощью библиотеки Cartopy. Она способна визуализировать картируемые данные с помощью различных видов проекции. Выбор подходящей проекции зависит от характера картируемых данных. Если они охватывают весь глобус, то можно использовать стандартную *равнопромежуточную проекцию*. Если же ограничены Северной Америкой, то лучше задействовать *ортографическую проекцию*. Если точки данных расположены в континентальной части США, то следует применять *равноугольную коническую проекцию Ламберта*.
- Значения широты и долготы можно получать из названий локаций при помощи библиотеки GeoNamesCache. Она также сопоставляет названия стран с относящимися к ним городами. Таким образом, зная название страны, можно аппроксимировать ее центральные координаты путем усреднения значений широты и долготы всех ее городов. Однако эта аппроксимация окажется неидеальной из-за смещения городов и кривизны поверхности Земли.
- Нередко бывает, что несколько городов имеют одинаковое название. Поэтому GeoNamesCache может сопоставить несколько разных наборов координат с одним названием города. Если в нашем распоряжении имеется лишь название города без дополнительной информации, рекомендуется возвращать координаты наиболее густонаселенного города с таким названием.
- GeoNamesCache сопоставляет координаты с названиями городов, которые могут иметь ударения. Можно удалить эти ударения с помощью функции `unidecode` из внешней библиотеки Unidecode.
- *Регулярные выражения* позволяют находить названия локаций в тексте. Совместив GeoNamesCache с Cartopy и регулярными выражениями, можно наносить на карту места, упомянутые в тексте.

12 Решение практического задания 3

В этой главе

- ✓ Извлечение и визуализация локаций.
- ✓ Очистка данных.
- ✓ Кластеризация локаций.

Наша цель — извлечь локации из новостных заголовков, сообщающих о вспышках заболеваний, чтобы обнаружить наиболее активные эпидемии внутри и вне США. Для этого мы проделаем следующее.

1. Скачаем данные.
2. Извлечем локации из текста, используя регулярные выражения и библиотеку GeoNamesCache.
3. Проверим соответствия локаций на ошибки.
4. Кластеризуем локации на основе географического расстояния между ними.
5. Визуализируем кластеры на карте и удалим все ошибки.
6. Выведем репрезентативные локации из самых крупных кластеров для формирования информативных заключений.

ВНИМАНИЕ

Спойлер! Далее раскрывается решение третьего практического задания. Настоятельно рекомендую попытаться разобраться с ним самостоятельно, прежде чем читать готовое решение. Условия задачи можно посмотреть в начале практического задания.

12.1. ИЗВЛЕЧЕНИЕ ЛОКАЦИЙ ИЗ ЗАГОЛОВКОВ

Начнем с загрузки данных новостных заголовков (листинг 12.1).

Листинг 12.1. Загрузка данных заголовков

```
headline_file = open('headlines.txt','r')
headlines = [line.strip()
             for line in headline_file.readlines()]
num_headlines = len(headlines)
print(f"{num_headlines} headlines have been loaded")

650 headlines have been loaded
```

Мы загрузили 650 заголовков. Теперь нам нужен механизм для извлечения из их текста названий городов и стран. Простейшим решением будет сопоставить локации из GeoNamesCache с каждым заголовком. Однако такой подход не позволит сопоставить названия мест, в которых регистр и диакритические знаки отличаются от сохраненных в базе GeoNamesCache. Для более эффективного сопоставления необходимо преобразовать название каждой локации в регулярное выражение, не зависящее от регистра и ударений. Сделать это можно с помощью кастомной функции `name_to_regex` (листинг 12.2). Она получает на входе название локации и возвращает скомпилированное регулярное выражение, с помощью которого можно будет определить любое выбранное местоположение.

Листинг 12.2. Преобразование названий в регулярные выражения

```
def name_to_regex(name):
    decoded_name = unidecode(name)
    if name != decoded_name:
        regex = fr'\b({name}|{decoded_name})\b'
    else:
        regex = fr'\b{name}\b'
    return re.compile(regex, flags=re.IGNORECASE)
```

Используя `name_to_regex`, можно выполнить сопоставление между регулярными выражениями и оригинальными названиями из GeoNamesCache. Мы создадим два словаря, `country_to_name` и `city_to_name`, в которых регулярные выражения будут сопоставляться с названиями стран и городов соответственно (листинг 12.3).

Листинг 12.3. Сопоставление названий с регулярными выражениями

```
countries = [country['name']
             for country in gc.get_countries().values()]
country_to_name = {name_to_regex(name): name
                   for name in countries}

cities = [city['name'] for city in gc.get_cities().values()]
city_to_name = {name_to_regex(name): name for name in cities}
```

Далее с помощью этих сопоставлений мы определим функцию, ищущую названия локаций в тексте. Она будет получать на входе заголовок и словарь локаций, после чего начнет перебирать каждый ключ регулярных выражений в этом словаре, возвращая связанное с ним значение в случае совпадения регулярного выражения с заголовком (листинг 12.4).

Листинг 12.4. Поиск локаций в тексте

```
def get_name_in_text(text, dictionary):
    for regex, name in sorted(dictionary.items(),
                              key=lambda x: x[1]):
        if regex.search(text):
            return name
    return None
```

Перебор словарей дает недетерминированную последовательность результатов. Изменение порядка этой последовательности может повлиять на то, какие локации сопоставляются с входным текстом. Это особенно актуально, если в тексте упомянуты несколько мест. Упорядочение по названию локации гарантирует, что вывод функции между ее выполнениями меняться не будет

Мы задействуем `get_name_in_text` для нахождения городов и стран, упомянутых в списке `headlines`, сохраняя результаты в таблице Pandas для упрощения дальнейшего анализа (листинг 12.5).

Листинг 12.5. Поиск локаций в заголовках

```
import pandas as pd

matched_countries = [get_name_in_text(headline, country_to_name)
                     for headline in headlines]
matched_cities = [get_name_in_text(headline, city_to_name)
                  for headline in headlines]
data = {'Headline': headlines, 'City': matched_cities,
        'Country': matched_countries}
df = pd.DataFrame(data)
```

Теперь изучим полученную таблицу локаций. Начнем с обобщения содержимого `df` при помощи метода `describe` (листинг 12.6).

Листинг 12.6. Обобщение данных о локациях

```
summary = df[['City', 'Country']].describe()
print(summary)

       City Country
count   619      15
unique  511      10
top      Of  Brazil
freq     45       3
```

ПРИМЕЧАНИЕ

Среди данных несколько стран имеют одинаковое максимальное число вхождений, равное 3. В Pandas нет детерминированного метода для выбора среди таких стран одной, превосходящей другие. В зависимости от ваших локальных настроек в качестве лидирующей страны может быть возвращена не Бразилия, но частота ее вхождений по-прежнему будет равна трем.

Таблица содержит 619 упоминаний городов, представленных 511 уникальными названиями. Она также содержит всего 15 стран, представленных десятью уникальными названиями. Чаще всего упоминается Бразилия, которая встречается в трех заголовках.

Среди городов самым часто упоминаемым, очевидно, является турецкий Of. Но это некорректно. Скорее всего, 45 вхождений Of соответствуют английскому предлогу, а не редко упоминаемому городу в Турции. Для подтверждения этой ошибки выведем некоторые вхождения Of (листинг 12.7).

Листинг 12.7. Получение названий городов Of

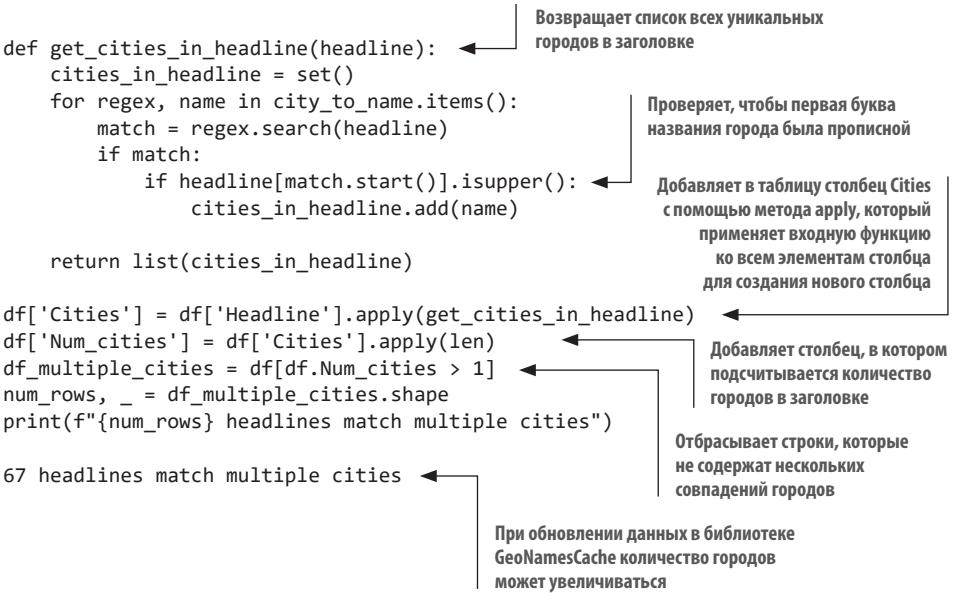
```
of_cities = df[df.City == 'Of'][['City', 'Headline']]
ten_of_cities = of_cities.head(10)
print(ten_of_cities.to_string(index=False))
```

Преобразует df в строку, где индексы строк удалены, обеспечивая более сжатый вывод

```
City                                          Headline
  Of              Case of Measles Reported in Vancouver
  Of  Authorities are Worried about the Spread of Br...
  Of  Authorities are Worried about the Spread of Ma...
  Of  Rochester authorities confirmed the spread of ...
  Of     Tokyo Encounters Severe Symptoms of Meningitis
  Of  Authorities are Worried about the Spread of In...
  Of            Spike of Pneumonia Cases in Springfield
  Of  The Spread of Measles in Spokane has been Conf...
  Of                    Outbreak of Zika in Panama City
  Of    Urbana Encounters Severe Symptoms of Meningitis
```

Как видно, обнаруженные совпадения Of действительно оказались ошибочными. Для исправления этой ошибки достаточно будет обеспечить, чтобы все совпадения были написаны с прописной буквы. Однако этот недочет является признаком более крупной проблемы: во всех ошибочно сопоставленных заголовках мы сравнивали регулярное выражение с Of, а не с фактическим названием города. Так получилось, потому что мы не учитывали возможность нескольких совпадений в заголовке. Как часто заголовки содержат более одного совпадения? Надо это выяснить. Мы отследим список всех совпадающих названий городов в заголовках при помощи дополнительного столбца `Cities` (листинг 12.8).

Листинг 12.8. Поиск заголовков с упоминанием нескольких городов




```
def get_cities_in_headline(headline):
    cities_in_headline = set()
    for regex, name in city_to_name.items():
        match = regex.search(headline)
        if match:
            if headline[match.start()].isupper():
                cities_in_headline.add(name)

    return list(cities_in_headline)

df['Cities'] = df['Headline'].apply(get_cities_in_headline)
df['Num_cities'] = df['Cities'].apply(len)
df_multiple_cities = df[df.Num_cities > 1]
num_rows, _ = df_multiple_cities.shape
print(f"{num_rows} headlines match multiple cities")

67 headlines match multiple cities
```

Мы выяснили, что 67 заголовков, представляющих приблизительно 10 % всех данных, содержат более одного города. Почему же упоминание более чем одного города наблюдается в таком количестве заголовков? Возможно, это удастся понять, изучив некоторые совпадения (листинг 12.9).

Листинг 12.9. Семплинг заголовков с несколькими названиями городов

```
ten_cities = df_multiple_cities[['Cities', 'Headline']].head(10)
print(ten_cities.to_string(index=False))

Cities                          Headline
[York, New York City]           Could Zika Reach New York City?
[Miami Beach, Miami]            First Case of Zika in Miami Beach
[San Juan, San]                 San Juan reports 1st U.S. Zika-related death
     amid outbreak
[Los Angeles, Los Ángeles]      New Los Angeles Hairstyle goes Viral
[Bay, Tampa]                    Tampa Bay Area Zika Case Count Climbs
[Ho Chi Minh City, Ho]          Zika cases in Vietnam's Ho Chi Minh City
     surge
[San, San Diego]                Key Zika Findings in San Diego Institute
[H?t, Kuala Lumpur]             Kuala Lumpur is Hit By Zika Threat
[San, San Francisco]            Zika Virus Reaches San Francisco
[Salvador, San, San Salvador]   Zika worries in San Salvador
```

Оказывается, что с заголовками сопоставляются не только длинные корректные названия городов, но и их короткие ошибочные формы. Например, город `'San'` всегда возвращается с более правдоподобными названиями вроде `'San Francisco'` и `'San Salvador'`. Как исправить эту ошибку? Одно из решений — просто возвращать при обнаружении более одного совпавшего названия самое длинное из них (листинг 12.10).

Листинг 12.10. Выбор наиболее длинных названий городов

```
def get_longest_city(cities):
    if cities:
        return max(cities, key=len)
    return None

df['City'] = df['Cities'].apply(get_longest_city)
```

В качестве проверки такого решения выведем строки, которые содержат короткое название города (четыре символа или меньше) (листинг 12.11). Таким образом мы обеспечим, чтобы заголовку не приписывалось ошибочное короткое название.

Листинг 12.11. Вывод самых коротких названий городов

```
short_cities = df[df.City.str.len() <= 4][['City', 'Headline']]
print(short_cities.to_string(index=False))
```

```
City                                          Headline
Lima                Lima tries to address Zika Concerns
Pune                     Pune woman diagnosed with Zika
Rome  Authorities are Worried about the Spread of Ma...
Molo                Molo Cholera Spread Causing Concern
Miri                               Zika arrives in Miri
Nadi  More people in Nadi are infected with HIV ever...
Baud  Rumors about Tuberculosis Spreading in Baud ha...
Kobe                     Chikungunya re-emerges in Kobe
Waco                More Zika patients reported in Waco
Erie                        Erie County sets Zika traps
Kent                       Kent is infested with Rabies
Reno  The Spread of Gonorrhea in Reno has been Confi...
Sibu                      Zika symptoms spotted in Sibu
Baku    The Spread of Herpes in Baku has been Confirmed
Bonn  Contaminated Meat Brings Trouble for Bonn Farmers
Jaen                         Zika Troubles come to Jaen
Yuma                      Zika seminars in Yuma County
Lyon                  Mad Cow Disease Detected in Lyon
Yiwu  Authorities are Worried about the Spread of He...
Suva  Suva authorities confirmed the spread of Rotav...
```

Результаты выглядят правдивыми. Теперь перейдем от городов к странам. Только 15 заголовков содержат фактическую информацию о странах. Такое количество довольно невелико, и проверить все эти заголовки можно вручную (листинг 12.12).

Листинг 12.12. Получение заголовков с упоминанием стран

```
df_countries = df[df.Country.notnull()][['City',
                                         'Country',
                                         'Headline']]
print(df_countries.to_string(index=False))
```

Метод df.Country.notnull() возвращает список логических значений, каждое из которых является True, только если страна присутствует в соответствующей строке

```
          City     Country                                  Headline
        Recife      Brazil            Mystery Virus Spreads in Recife, Brazil
Ho Chi Minh City    Vietnam     Zika cases in Vietnam's Ho Chi Minh City surge
       Bangkok     Thailand                      Thailand-Zika Virus in Bangkok
    Piracicaba      Brazil                Zika outbreak in Piracicaba, Brazil
         Klang    Malaysia                   Zika surfaces in Klang, Malaysia
 Guatemala City  Guatemala  Rumors about Meningitis spreading in Guatemala...
    Belize City      Belize                  Belize City under threat from Zika
       Campinas      Brazil                   Student sick in Campinas, Brazil
    Mexico City      Mexico               Zika outbreak spreads to Mexico City
  Kota Kinabalu    Malaysia           New Zika Case in Kota Kinabalu, Malaysia
     Johor Bahru    Malaysia                  Zika reaches Johor Bahru, Malaysia
       Hong Kong  Hong Kong                     Norovirus Exposure in Hong Kong
    Panama City      Panama                    Outbreak of Zika in Panama City
      Singapore  Singapore                  Zika cases in Singapore reach 393
    Panama City      Panama            Panama City's first Zika related death
```

Все включающие названия стран заголовки содержат также информацию о городах. Значит, можно присвоить широту и долготу, не опираясь на центральные координаты страны. Благодаря этому можно вообще исключить из анализа названия стран (листинг 12.13).

Листинг 12.13. Исключение стран из таблицы

```
df.drop('Country', axis=1, inplace=True)
```

Мы почти готовы добавить в таблицу значения широты и долготы. Однако сначала нужно рассмотреть строки, в которых локации обнаружены не были. Подсчитаем количество заголовков без совпадений, после чего выведем полученное подмножество на экран (листинг 12.14).

Листинг 12.14. Анализ заголовков без совпадений

```
df_unmatched = df[df.City.isnull()]
num_unmatched = len(df_unmatched)
print(f"{num_unmatched} headlines contain no city matches.")
print(df_unmatched.head(10)[['Headline']].values)
```

```
39 headlines contain no city matches.
[['Louisiana Zika cases up to 26']
 ['Zika infects pregnant woman in Cebu']
 ['Spanish Flu Sighted in Antigua']
 ['Zika case reported in Oton']
 ['Hillsborough uses innovative trap against Zika 20 minutes ago']
```

```
['Maka City Experiences Influenza Outbreak']
['West Nile Virus Outbreak in Saint Johns']
['Malaria Exposure in Sussex']
['Greenwich Establishes Zika Task Force']
['Will West Nile Virus vaccine help Parsons?']]
```

Примерно 6 % заголовков не содержат совпадений с какими-либо городами. В некоторых из них упоминаются существующие города, которые GeoNamesCache не смогла распознать. Как следует рассматривать такие города? Учитывая их низкую частоту, пожалуй, стоит эти недостающие упоминания удалить (листинг 12.15). Ценой их удаления станет некоторое ухудшение качества данных, но это не сильно повлияет на результаты, поскольку охват совпавших городов достаточно велик.

Листинг 12.15. Исключение заголовков без совпадений

```
df = df[~df.City.isnull()][['City', 'Headline']]
```

Символ ~ возвращает логические значения обратно в список, полученный методом df.City.isnull(). В результате каждое такое реверсированное значение является True, только если город присутствует в соответствующей строке

11.2. ВИЗУАЛИЗАЦИЯ И КЛАСТЕРИЗАЦИЯ ИЗВЛЕЧЕННЫХ ДАННЫХ О ЛОКАЦИЯХ

Все строки нашей таблицы содержат название города. Теперь можно присвоить каждому из них широту и долготу. Мы задействуем `get_cities_by_name` для возвращения координат наиболее густонаселенного города с полученным названием (листинг 12.16).

Листинг 12.16. Присваивание городам географических координат

```
latitudes, longitudes = [], []
for city_name in df.City.values:
    city = max(gc.get_cities_by_name(city_name),
               key=lambda x: list(x.values())[0]['population'])
    city = list(city.values())[0]
    latitudes.append(city['latitude'])
    longitudes.append(city['longitude'])

df = df.assign(Latitude=latitudes, Longitude=longitudes)
```

Выбирает совпавший город с наибольшим населением

Извлекает широту и долготу города

Добавляет в таблицу столбцы Latitude и Longitude

Присвоив значения широты и долготы, можно перейти к кластеризации данных. Мы применим алгоритм *K*-средних для нашего набора 2D-координат, подобрав значение *K* методом локтя (листинг 12.17; рис. 12.1).

Листинг 12.17. Построение географического изгиба локтя

```
coordinates = df[['Latitude', 'Longitude']].values
k_values = range(1, 10)
inertia_values = []
for k in k_values:
    inertia_values.append(KMeans(k).fit(coordinates).inertia_)
plt.plot(range(1, 10), inertia_values)
plt.xlabel('K')
plt.ylabel('Inertia')
plt.show()
```

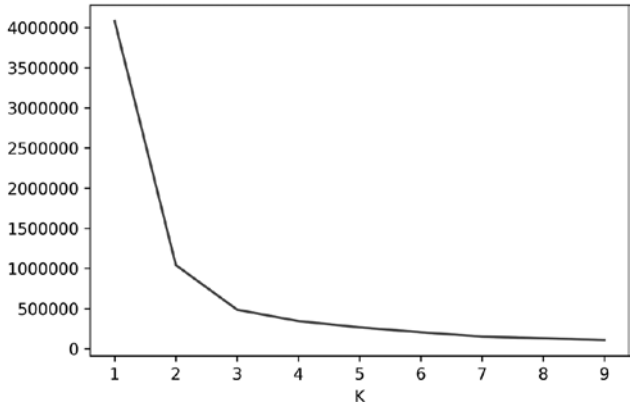


Рис. 12.1. Географическая кривая в виде локтя указывает на K, равное 3

Локоть графика указывает на *K*, равное 3. Это очень низкое значение *K*, ограничивающее область максимум до трех географических территорий. И все же нам нужно сохранять правдивость в аналитической методологии. Кластеризуем локации на три группы и отобразим их на карте (листинг 12.18; рис. 12.2).

Листинг 12.18. Кластеризация городов по трем группам с помощью алгоритма *K*-средних

На протяжении оставшейся части анализа эта функция будет повторно использоваться для построения кластеров

```
def plot_clusters(clusters, longitudes, latitudes):
    plt.figure(figsize=(12, 10))
    ax = plt.axes(projection=PlateCarree())
    ax.coastlines()
    ax.scatter(longitudes, latitudes, c=clusters)
    ax.set_global()
    plt.show()

df['Cluster'] = KMeans(3).fit_predict(coordinates)
plot_clusters(df.Cluster, df.Longitude, df.Latitude)
```

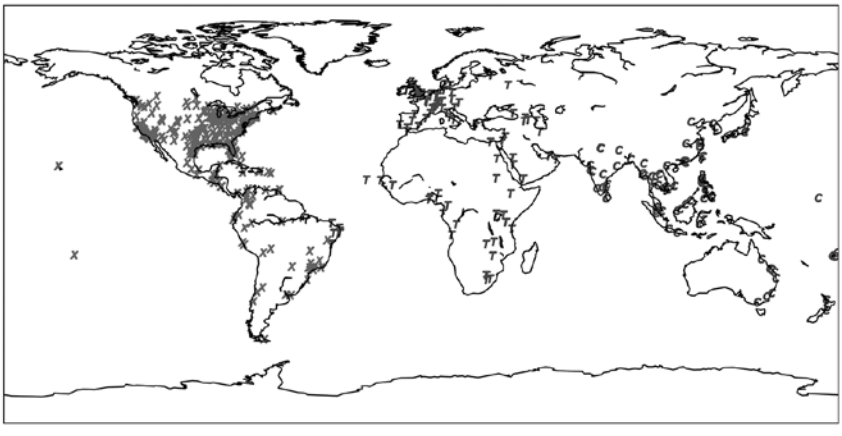


Рис. 12.2. Кластеры городов, нанесенные на карту по методу K-средних. K установлено на 3. Три полученных кластера распределены по шести континентам

ПРИМЕЧАНИЕ

Форма маркеров на рис. 12.1–12.5 была вручную скорректирована, чтобы лучше различать кластеры в черно-белой версии книги.

Результаты получились неадекватные. Три наших кластера охватывают:

- Северную и Южную Америку;
- Африку и Европу;
- Азию и Австралию.

Эти континентальные категории слишком обширны, чтобы быть полезными. Более того, все города, расположенные на восточном побережье Южной Америки, почему-то кластеризуются с африканскими и европейскими локациями, несмотря на то что их разделяет океан. Эти кластеры не помогут понять данные. Возможно, выбранное значение *K* было слишком мало. Попробуем отклониться от рекомендации метода локтя и удвоить величину *K* до 6 (листинг 12.19; рис. 12.3).

Листинг 12.19. Кластеризация городов по шести группам с помощью алгоритма K-средних

```
df['Cluster'] = KMeans(6).fit_predict(coordinates)
plot_clusters(df.Cluster, df.Longitude, df.Latitude)
```

Увеличение *K* повышает качество кластеризации в Северной и Южной Америке. Теперь Южная Америка попадает в собственный кластер, а Северная Америка разделяется между двумя группами кластеров — западной и восточной. Однако

на противоположной стороне Атлантики кластеризация по-прежнему неудачная. Геолокации Африки ошибочно разделились между Европой и Азией. Опираясь на свое понимание центральности, алгоритм *K*-средних не может провести адекватное различие между Африкой, Европой и Азией. Возможно, правильно осознать связи между точками, распределенными по изогнутой поверхности нашей планеты, ему не позволяет использование евклидова показателя.

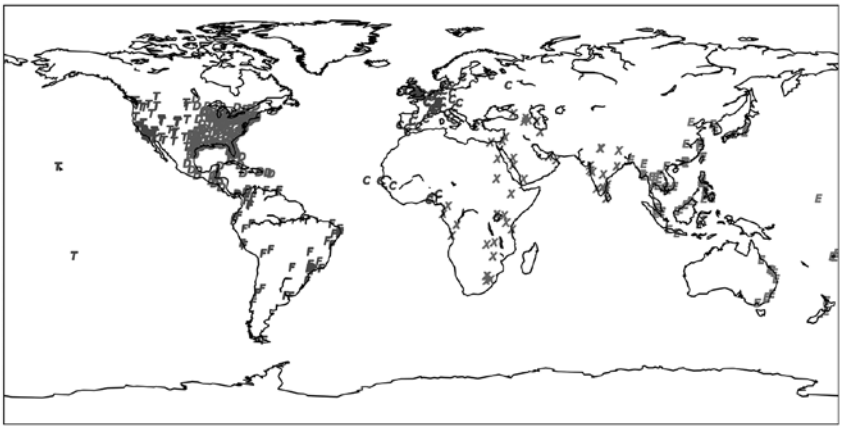


Рис. 12.3. Кластеры городов, нанесенные на карту. K установлено равным 6. Кластеризованные точки Африки ошибочно разделяются между Европой и Азией

В качестве альтернативного подхода можно попробовать кластеризацию с помощью DBSCAN, который способен работать с любым нужным нам параметром расстояния. Это позволит кластеризовать точки на основе расстояний по ортодромии. Начнем с написания функции определения этого расстояния, которая будет получать два массива NumPy (листинг 12.20).

Листинг 12.20. Определение параметра вычисления расстояния по ортодромии на основе NumPy

```
def great_circle_distance(coord1, coord2, radius=3956):
    if np.array_equal(coord1, coord2):
        return 0.0

    coord1, coord2 = np.radians(coord1), np.radians(coord2)
    delta_x, delta_y = coord2 – coord1
    haversin = sin(delta_x / 2) ** 2 + np.product([cos(coord1[0]),
                                                   cos(coord2[0]),
                                                   sin(delta_y / 2) ** 2])

    return 2 * radius * asin(haversin ** 0.5)
```

Для radius установлен размер радиуса Земли в милях

Мы определили параметр расстояния и практически готовы выполнить алгоритм DBSCAN. Однако сначала нужно выбрать для параметров `eps` и `min_samples` разумные значения. Предположим следующее: кластер глобальных городов содержит не менее трех городов, которые в среднем удалены друг от друга максимум на 250 миль. Исходя из этого предположения, мы передадим в параметры `eps` и `min_samples` значения `250` и `3` соответственно (листинг 12.21).

Листинг 12.21. Кластеризация городов с помощью DBSCAN

```
metric = great_circle_distance
dbscan = DBSCAN(eps=250, min_samples=3, metric=metric)
df['Cluster'] = dbscan.fit_predict(coordinates)
```

Выпадающим точкам данных, которые не кластеризуются, DBSCAN присваивает значение `-1`. Удалим эти выбросы из таблицы и отобразим оставшиеся результаты (листинг 12.22; рис. 12.4).

Листинг 12.22. Построение кластеров без выбросов с помощью DBSCAN

```
df_no_outliers = df[df.Cluster > -1]
plot_clusters(df_no_outliers.Cluster, df_no_outliers.Longitude,
              df_no_outliers.Latitude)
```

DBSCAN отлично справился с генерацией раздельных кластеров в Южной Америке, Азии и Южной Африке. Хотя восточная часть США попала в один очень плотный кластер. Почему? Виноват в этом несправедливый охват информационных материалов в западных медиа, которые с большей охотой освещают именно события в США, что ведет к более плотному распределению упоминаемых локаций. Как вариант, для исправления этого географического смещения можно повторно кластеризовать города США, используя более продуманный параметр `eps`. В контексте условия задачи такая стратегия выглядит разумной, поскольку нам нужно выделить ведущие кластеры отдельно среди новостных заголовков, относящихся к Америке, и среди заголовков, освещающих события в глобальном мире. Поэтому мы кластеризуем локации США независимо от остального мира. Для этого сначала присвоим каждому из городов код его страны (листинг 12.23; см. рис. 12.4).

Листинг 12.23. Присваивание городам кодов их стран

```
def get_country_code(city_name):
    city = max(gc.get_cities_by_name(city_name),
               key=lambda x: list(x.values())[0]['population'])
    return list(city.values())[0]['countrycode']

df['Country_code'] = df.City.apply(get_country_code)
```

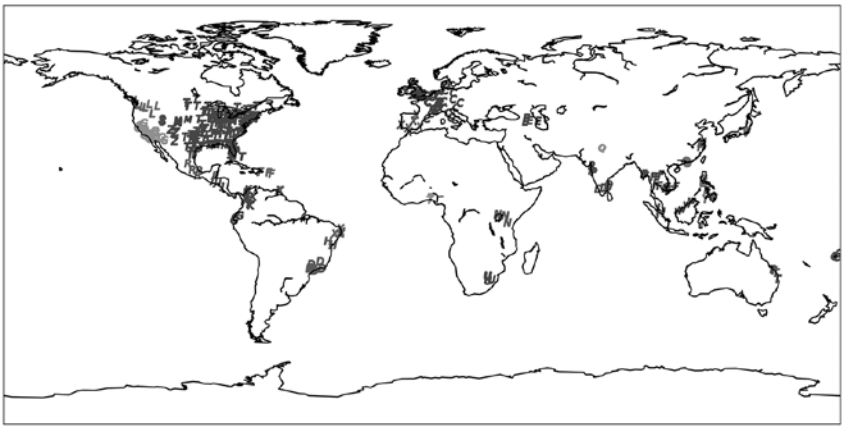


Рис. 12.4. Кластеры городов, нанесенные на карту с помощью DBSCAN и параметра вычисления расстояния по ортодромии

Присваивание кодов стран позволит нам разделить данные на два отдельных объекта `DataFrame` (листинг 12.24). Первый объект, `df_us`, содержит локации США, второй, `df_not_us`, — все остальные кластеризуемые города планеты.

Листинг 12.24. Разделение городов США и всех прочих

```
df_us = df[df.Country_code == 'US']
df_not_us = df[df.Country_code != 'US']
```

Мы разделили города на две группы: находящиеся в США и все остальные. Теперь нужно повторно кластеризовать координаты в двух отдельных таблицах (листинг 12.25). Повторная кластеризация `df_not_us` неизбежна ввиду изменения плотности, вызванного удалением всех локаций США. Однако при ее кластеризации мы сохраним значение `eps` равным 250. А вот для `df_us` уменьшим значение `eps` вдвое — до 125, чтобы учесть повышенную плотность локаций США. Наконец, после кластеризации удалим все выбросы.

Листинг 12.25. Повторная кластеризация извлеченных городов

```
def re_cluster(input_df, eps):
    input_coord = input_df[['Latitude', 'Longitude']].values
    dbscan = DBSCAN(eps=eps, min_samples=3,
                    metric=great_circle_distance)
    clusters = dbscan.fit_predict(input_coord)
    input_df = input_df.assign(Cluster=clusters)
    return input_df[input_df.Cluster > -1]

df_not_us = re_cluster(df_not_us, 250)
df_us = re_cluster(df_us, 125)
```

12.3. ФОРМИРОВАНИЕ ВЫВОДОВ НА ОСНОВЕ КЛАСТЕРОВ ЛОКАЦИЙ

Займемся анализом кластеризованных данных из таблицы `df_not_us`. Для начала сгруппируем их с помощью метода Pandas `groupby` (листинг 12.26).

Листинг 12.26. Группировка городов по их кластерам

```
groups = df_not_us.groupby('Cluster')
num_groups = len(groups)
print(f"{num_groups} Non-US clusters have been detected")

31 Non-US clusters have been detected
```

Всего обнаружен 31 глобальный кластер. Упорядочим эти группы по размеру (листинг 12.27) и подсчитаем заголовки в самом большом.

Листинг 12.27. Поиск наибольшего кластера

```
sorted_groups = sorted(groups, key=lambda x: len(x[1]),
                       reverse=True)
group_id, largest_group = sorted_groups[0]
group_size = len(largest_group)
print(f"Largest cluster contains {group_size} headlines")

Largest cluster contains 51 headlines
```

Самый крупный кластер содержит 51 заголовок. Поочередное прочтение каждого займет немало времени, но можно его сэкономить, выведя лишь те заголовки, которые представляют наиболее близкие к центру локации. Центральность мы определим путем вычисления средней широты и долготы группы (листинг 12.28). Затем можно будет вычислить расстояния между всеми локациями и средние координаты. Меньшие расстояния будут означать бóльшую центральность.

ПРИМЕЧАНИЕ

Как говорилось в главе 11, средние широта и долгота просто аппроксимируют центр, поскольку не учитывают сферичность Земли.

Далее мы определим функцию `compute_centrality`, присваивающую входной группе столбец `Distance_to_center`.

Листинг 12.28. Вычисление центральности кластера

```
def compute_centrality(group):
    group_coords = group[['Latitude', 'Longitude']].values
    center = group_coords.mean(axis=0)
    distance_to_center = [great_circle_distance(center, coord)
                          for coord in group_coords]
    group['Distance_to_center'] = distance_to_center
```

Теперь можно упорядочить все заголовки по центральности. Мы выведем пять наиболее приближенных к центру (листинг 12.29).

Листинг 12.29. Поиск центральных заголовков в самом большом кластере

```
def sort_by_centrality(group):
    compute_centrality(group)
    return group.sort_values(by=['Distance_to_center'], ascending=True)

largest_group = sort_by_centrality(largest_group)
for headline in largest_group.Headline.values[:5]:
    print(headline)

Mad Cow Disease Disastrous to Brussels
Scientists in Paris to look for answers
More Livestock in Fontainebleau are infected with Mad Cow Disease
Mad Cow Disease Hits Rotterdam
Contaminated Meat Brings Trouble for Bonn Farmers
```

Центральные заголовки в `largest_group` сосредоточены вокруг вспышки коровьего бешенства в различных городах Европы. В том, что регион кластера центрируется в Европе, можно убедиться, выведя ведущие страны, связанные с городами из кластера (листинг 12.30).

Листинг 12.30. Поиск трех ведущих стран в самом большом кластере

```
from collections import Counter
def top_countries(group):
    countries = [gc.get_countries()[country_code]['name']
                 for country_code in group.Country_code.values]
    return Counter(countries).most_common(3)

print(top_countries(largest_group))
```

Класс Counter отслеживает наиболее часто повторяющиеся элементы списка и их количество

```
[('United Kingdom', 19), ('France', 7), ('Germany', 6)]
```

Чаще всего в `largest_group` упоминаются города, расположенные в Соединенном Королевстве, Франции и Германии. Большинство локаций из `largest_group` определенно относятся к Европе.

Теперь мы повторим этот анализ для следующих четырех крупнейших глобальных кластеров. Код листинга 12.31 поможет определить, угрожает ли сейчас нашей планете какая-либо другая эпидемия.

Листинг 12.31. Обобщение содержимого крупнейших кластеров

```
for _, group in sorted_groups[1:5]:
    sorted_group = sort_by_centrality(group)
    print(top_countries(sorted_group))
    for headline in sorted_group.Headline.values[:5]:
        print(headline)
    print('\n')
```

```
[('Philippines', 16)]
Zika afflicts patient in Calamba
Hepatitis E re-emerges in Santa Rosa
More Zika patients reported in Indang
Batangas Tourism Takes a Hit as Virus Spreads
Spreading Zika reaches Bacoor

[('El Salvador', 3), ('Honduras', 2), ('Nicaragua', 2)]
Zika arrives in Tegucigalpa
Santa Barbara tests new cure for Hepatitis C
Zika Reported in Ilopango
More Zika cases in Soyapango
Zika worries in San Salvador

[('Thailand', 5), ('Cambodia', 3), ('Vietnam', 2)]
More Zika patients reported in Chanthaburi
Thailand-Zika Virus in Bangkok
Zika case reported in Phetchabun
Zika arrives in Udon Thani
More Zika patients reported in Kampong Speu

[('Canada', 10)]
Rumors about Pneumonia spreading in Ottawa have been refuted
More people in Toronto are infected with Hepatitis E every year
St. Catharines Patient in Critical Condition after Contracting Dengue
Varicella has Arrived in Milton
Rabies Exposure in Hamilton
```

Только не это! На Филиппинах распространяется вирус Зика. Его вспышки обнаружены также в Южной Азии и Центральной Америке. А вот канадский кластер содержит смесь различных освещающих заболевания заголовков, указывая на то, что сейчас в этой северной стране ни одно из них не доминирует.

Теперь переключимся на кластеры США и начнем с их визуализации на карте страны (листинг 12.32; рис. 12.5).

Листинг 12.32. Построение обнаруженных DBSCAN кластеров на территории США

```
plt.figure(figsize=(12, 10))
ax = plt.axes(projection=LambertConformal())
ax.set_extent(us_extent)
ax.scatter(df_us.Longitude, df_us.Latitude, c=df_us.Cluster,
              transform=PlateCarree())
ax.coastlines()
ax.add_feature(cartopy.feature.STATES)
plt.show()
```

Полученная карта показывает вразумительный результат. Восточные штаты больше не попадают в один плотный кластер. Мы проанализируем пять ведущих кластеров США, выведя их упорядоченные по центральности заголовки (листинг 12.33).

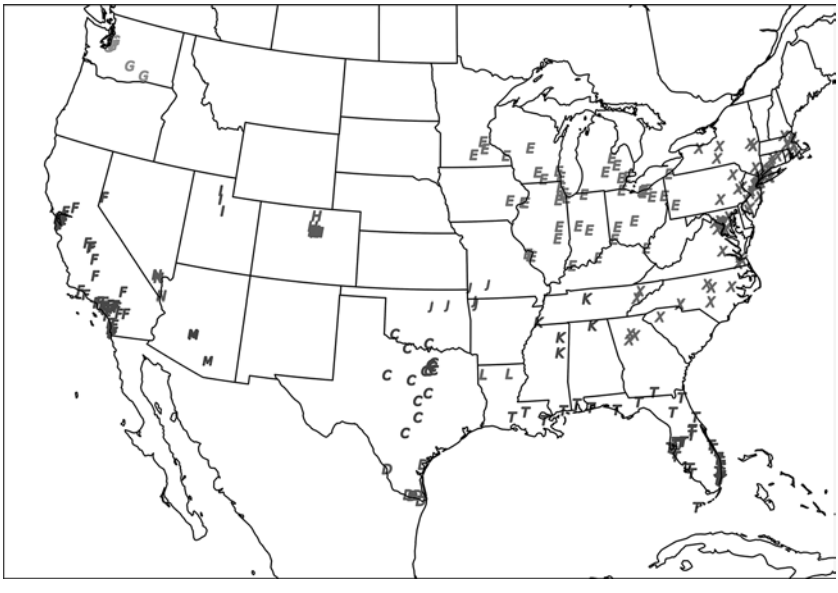


Рис. 12.5. Нанесенные на карту кластеры локаций в границах США, полученные с помощью DBSCAN

Листинг 12.33. Обобщение содержимого крупнейших кластеров США

```
us_groups = df_us.groupby('Cluster')
us_sorted_groups = sorted(us_groups, key=lambda x: len(x[1]),
                          reverse=True)
for _, group in us_sorted_groups[:5]:
    sorted_group = sort_by_centrality(group)
    for headline in sorted_group.Headline.values[:5]:
        print(headline)
    print('\n')
```

```
Schools in Bridgeton Closed Due to Mumps Outbreak
Philadelphia experts track pandemic
Vineland authorities confirmed the spread of Chlamydia
Baltimore plans for Zika virus
Will Swine Flu vaccine help Annapolis?

Bradenton Experiences Zika Troubles
Tampa Bay Area Zika Case Count Climbs
Zika Strikes St. Petersburg
New Zika Case Confirmed in Sarasota County
Zika spreads to Plant City

Rhinovirus Hits Bakersfield
Schools in Tulare Closed Due to Mumps Outbreak
New medicine wipes out West Nile Virus in Ventura
```

```
Hollywood Outbreak Film Premieres
Zika symptoms spotted in Hollywood

How to Avoid Hepatitis E in South Bend
Hepatitis E Hits Hammond
Chicago's First Zika Case Confirmed
Rumors about Hepatitis C spreading in Darien have been refuted
Rumors about Rotavirus Spreading in Joliet have been Refuted

More Zika patients reported in Fort Worth
Outbreak of Zika in Stephenville
Zika symptoms spotted in Arlington
Dallas man comes down with case of Zika
Zika spreads to Lewisville
```

Эпидемия вируса Зика ударила по Флориде и Техасу. Это очень волнующая новость. Однако в остальных ведущих кластерах никаких выраженных паттернов заболеваемости не наблюдается. На данный момент распространение эпидемии Зика ограничено южными штатами. Мы незамедлительно сообщим об этом нашему руководству, чтобы они как можно раньше могли предпринять необходимые меры. И в рамках подготовки отчета построим итоговое изображение, которым оформим его первую полосу (рис. 12.6). Это изображение будет обобщать угрожающий масштаб распространения эпидемии лихорадки Зика — на нем будут показаны все кластеры на территории США и в остальном мире, где вспышка вируса Зика упоминается более чем в 50 % новостных заголовков (листинг 12.34).

Листинг 12.34. Построение кластеров, отражающих эпидемию лихорадки Зика

```
def count_zika_mentions(headlines):
    zika_regex = re.compile(r'\bzika\b',
                            flags=re.IGNORECASE)
    zika_count = 0
    for headline in headlines:
        if zika_regex.search(headline):
            zika_count += 1

    return zika_count

fig = plt.figure(figsize=(15, 15))
ax = plt.axes(projection=PlateCarree())
for _, group in sorted_groups + us_sorted_groups:
    headlines = group.Headline.values
    zika_count = count_zika_mentions(headlines)
    if float(zika_count) / len(headlines) > 0.5:
        ax.scatter(group.Longitude, group.Latitude)

ax.coastlines()
ax.set_global()
plt.show()
```

Подсчитывает, сколько раз вирус Зика упоминается в списке заголовков

Регулярное выражение, сопоставляющее экземпляр слова Zika с заголовками. Регистр при этом не учитывается

Перебирает кластеры США и остального мира

Наносит на карту кластеры, где вирус Зика упоминается более чем в 50 % заголовков

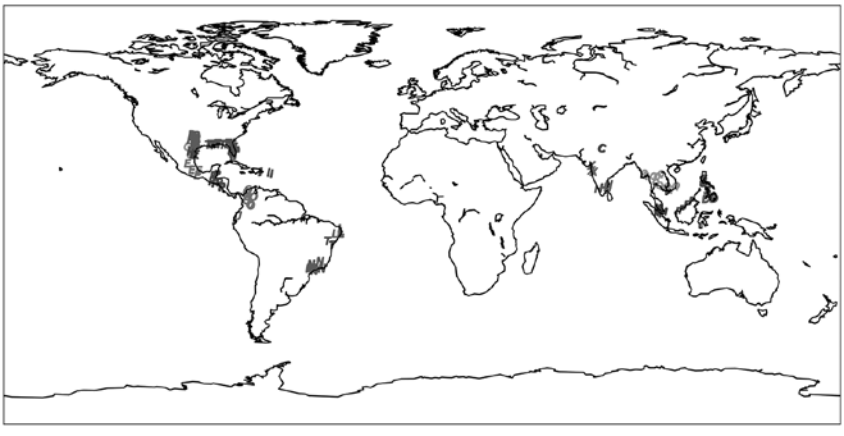


Рис. 12.6. Нанесенные на карту кластеры локаций, в которых Зика упоминается более чем в 50 % заголовков новостей

Мы успешно кластеризовали заголовки по локациям и нанесли на карту кластеры, в которых доминирует слово Zika. Эта связь между кластерами и их текстовым содержимым позволяет задать интересный вопрос: «Возможно ли кластеризовать заголовки на основе сходства текста, а не географического расстояния?» Иными словами, можно ли сгруппировать заголовки на основе совпадения их текста, чтобы все упоминания вируса Зика автоматически оказались в одном кластере? Да, это возможно. В следующем практическом задании вы узнаете, как измерять сходство между текстами, чтобы группировать документы по определенной теме.

РЕЗЮМЕ

- Инструменты аналитики данных порой оказываются непригодными по неожиданным причинам. Когда мы применили GeoNamesCache для новостных заголовков, эта библиотека ошибочно сопоставила с входным текстом короткие названия городов, такие как Of и San. Изучив данные, мы смогли вычислить эти ошибки. Если бы вместо этого мы слепо кластеризовали локации, то в качестве итогового вывода получили бы бессмыслицу. Прежде чем приступать к серьезному анализу, данные необходимо тщательно просматривать.
- Иногда в хорошем наборе данных встречаются проблемные места. В нашем случае примерно в 6 % заголовков города не были обнаружены из-за неполноценности информации, содержащейся в библиотеке. Исправить эту ошибку было бы трудно, поэтому мы предпочли просто удалить эти заголовки из набора данных. Иногда проблемные образцы данных можно безболезненно выбросить, если это не окажет ощутимого влияния на весь набор данных. Однако, прежде чем принимать такое решение, необходимо взвесить все его плюсы и минусы.

- Метод локтя эвристически выбирает значение K для алгоритма кластеризации по K-средним. Эвристические инструменты не обязательно корректно сработают в любой ситуации. В нашем анализе график локтя указал на значение $K = 3$, но оно оказалось слишком мало. В связи с этим мы решили вмешаться и выбрали собственное K. А если бы не стали вникать и просто доверились результату метода локтя, то вся кластеризация оказалась бы бессмысленной.
- При анализе кластеризованного результата необходимо руководствоваться здравым смыслом. Сначала мы проверили вывод алгоритма K-средних, в котором $K = 6$. По его результату наблюдали общую кластеризацию городов Центральной Африки и Европы. Этот результат оказался явно ошибочным — Европа и Центральная Африка являются совершенно разными локациями. Поэтому мы применили другой метод кластеризации. Когда здравый смысл говорит, что кластеризация выполнена ошибочно, необходимо пробовать другой подход.
- Иногда допустимо разбить набор данных на части и проанализировать каждую из них отдельно. В изначальном анализе с помощью DBSCAN алгоритм не смог верно кластеризовать города США. Большинство городов восточной части страны попали в один кластер. Можно было отказаться от подхода DBSCAN, но мы вместо этого кластеризовали города США отдельно от всех остальных, используя более подходящие параметры. Такой анализ набора данных в виде двух отдельных частей позволил получить уже куда более точную кластеризацию.

Практическое задание 4

Улучшение своего резюме аналитика данных на основе онлайн-вакансий

УСЛОВИЕ ЗАДАЧИ

Пришло время подумать о расширении карьеры аналитика данных. Через шесть месяцев мы будем пробовать устроиться на новую работу, так что пора начинать готовить резюме. В своем первичном виде оно будет приблизительным и неполным, в том числе без описания карьерных целей и образования. Тем не менее в базовой форме оно охватит первые четыре практических задания этой книги, включая текущее, которое мы закончим перед тем, как приступать к поиску нового места работы.

Наше черновое резюме далеко от идеала. Вероятно, в нем не хватает некоторых необходимых навыков. И если да, то каких именно? Выясним это с помощью анализа, ведь мы же аналитики данных. Мы заполняем пробелы в знаниях, используя тщательный анализ, так почему бы не применить этот анализ к самим себе?

Для начала нам нужно найти данные. Для этого посетим популярный сайт поиска работы, где размещены миллионы вакансий. Встроенный механизм поиска позволяет отфильтровать предложения о работе по ключевому слову, например *analyst* или *data scientist*. Кроме того, он может сопоставлять вакансии с загружаемыми

документами. Эта функция предназначена для поиска объявлений на основе содержания резюме. К сожалению, наше резюме еще в разработке, поэтому произведем поиск на основе содержания этой книги. Для этого скопируем первые 15 глав в текстовый файл.

Далее этот файл нужно будет загрузить на сайт поиска работы. Механизм сайта сопоставит первые четыре практических задания с миллионами размещенных вакансий, вернув тысячи совпадающих результатов. Среди них будут более и менее соответствующие критериям запроса. Мы не можем ручаться за общее качество работы системы, но все эти данные окажутся для нас полезными. Нужно будет скачать HTML-файл каждого объявления.

Наша цель — извлечь из скачанных данных общие навыки, характерные для аналитиков. После мы сравним эти навыки со своим резюме, определив, каких в нем недостает. Для достижения поставленной цели будем следовать такому алгоритму.

1. Спарсим весь текст из скачанных HTML-файлов.
2. Изучим полученный вывод, чтобы определить типичные навыки аналитиков данных, указываемые в вакансиях. Возможно, какие-то HTML-теги используются в описании навыков более часто.
3. Отфильтруем из полученного набора данных все нерелевантные вакансии. Механизм поиска неидеален, и мы вполне могли по ошибке скачать не самые подходящие объявления. Релевантность можно оценить сравнением вакансий с нашим резюме и содержанием книги.
4. Кластеризуем навыки в релевантных объявлениях и визуализируем их кластеры.
5. Сравним кластеризованные навыки с содержанием резюме, после чего спланируем его обновление с учетом недостающих навыков аналитика.

ОПИСАНИЕ НАБОРА ДАННЫХ

Черновик нашего резюме хранится в файле `resume.txt` и выглядит так:

```
Experience

1. Developed probability simulations using NumPy
2. Assessed online ad clicks for statistical significance using permutation testing
3. Analyzed disease outbreaks using common clustering algorithms

Additional Skills

1. Data visualization using Matplotlib
2. Statistical analysis using SciPy
3. Processing structured tables using Pandas
4. Executing K-means clustering and DBSCAN clustering using scikit-learn
```

```
5. Extracting locations from text using GeoNamesCache
6. Location analysis and visualization using GeoNamesCache and Cartopy
7. Dimensionality reduction with PCA and SVD using scikit-learn
8. NLP analysis and text topic detection using scikit-learn[1]
```

Навыки 7 и 8 мы освоим в последующих главах, в которых рассматривается данное практическое задание

Этот исходный черновик краткий и незавершенный. Чтобы восполнить недостаток материала, мы используем также часть содержания этой книги, которое хранится в файле `table_of_contents.txt`. Здесь перечислены первые 15 глав книги, а также все заголовки разделов верхнего уровня. Файл содержания использовался для поиска тысяч релевантных вакансий, которые мы скачали и сохранили в каталоге `job_postings`. В нем каждый файл является HTML-файлом, связанным с отдельной вакансией. Их все можно просмотреть локально в браузере.

ОБЗОР

Для решения поставленной задачи нам нужно уметь:

- оценивать сходство текстов;
- эффективно кластеризовать крупные текстовые наборы данных;
- визуально представлять множественные текстовые кластеры;
- парсить HTML-файлы для получения текстового содержимого.

---

[1] Опыт

1. Разработка симуляция вероятности с помощью NumPy.
2. Анализ статистической значимости онлайн-переходов по объявлениям с помощью пермутационного тестирования.
3. Анализ вспышек заболеваний с помощью распространенных алгоритмов кластеризации.

Дополнительные навыки

1. Визуализация данных с помощью Matplotlib.
2. Статистический анализ с помощью SciPy.
3. Обработка структурированных таблиц с помощью Pandas.
4. Выполнение кластеризации с помощью метода K-средних и DBSCAN, используя scikit-learn.
5. Извлечение локаций из текста, используя GeoNamesCache.
6. Анализ и визуализация локаций с помощью GeoNamesCache и Cartopy.
7. Уменьшение размерности с помощью PCA и SVD, используя scikit-learn.
8. NLP-анализ и определение темы текста с помощью scikit-learn.

13 Измерение сходства текстов

В этой главе

- ✓ Что такое обработка естественного языка.
- ✓ Сравнение текстов на основе совпадений слов.
- ✓ Сравнение текстов с использованием одномерных массивов, называемых векторами.
- ✓ Сравнение текстов с применением двухмерных массивов, называемых матрицами.
- ✓ Эффективные матричные вычисления с помощью NumPy.

Быстрый анализ текста способен спасать жизни. Представьте реальный случай: солдаты США взяли штурмом лагерь террористов и обнаружили там компьютер, хранящий терабайты заархивированных данных. Среди них находятся документы, текстовые сообщения и электронные письма, связанные с террористической деятельностью. Документов слишком много, чтобы их мог перечитать один человек. К счастью, у бойцов имелось специальное ПО для выполнения очень быстрого анализа текста. Оно позволило обработать все текстовые данные, не покидая лагерь. Проведенный на месте анализ раскрыл информацию о запланированном неподалеку теракте, что позволило военным незамедлительно отреагировать и предотвратить угрозу.

Такая быстрая реакция оказалась бы невозможной без технологии *обработки естественного языка* (NLP). NLP — это одна из ветвей науки о данных, которая

ориентирована на ускоренный анализ текста. Работает эта техника обычно с огромными наборами текстовых данных. В общей сложности NLP охватывает очень широкий спектр сфер применения, включая:

- корпоративный мониторинг постов в социальных сетях для оценки отношения к бренду компании;
- анализ расшифрованных телефонных разговоров для отслеживания наиболее распространенных жалоб клиентов;
- сопоставление людей на сайтах знакомств, исходя из общих интересов, описанных в их профилях;
- обработку врачебных записей для подтверждения правильности постановки диагноза.

Все эти случаи требуют быстрого анализа. Задержка получения важных сведений в подобных сценариях может обойтись дорого. К сожалению, прямая обработка текста — медленный процесс. Большинство вычислительных техник оптимизированы для работы с цифрами, а не с текстом. В связи с этим методы NLP опираются на преобразование чистого текста в численное представление. После замены всех слов и предложений числами данные можно проанализировать очень быстро.

В данной главе мы сосредоточимся на базовой задаче в области NLP — измерении сходства двух текстов. После этого изучим несколько численных техник для быстрого вычисления подобного сходства. Для максимальной эффективности всех этих вычислений нам потребуется преобразовывать входные тексты в двухмерные численные таблицы.

13.1. ПРОСТОЕ СРАВНЕНИЕ ТЕКСТОВ

Многие задачи NLP зависят от анализа сходства текстов и различий между ними. Предположим, мы хотим сравнить три простых текстовых выражения.

- `text1` — *She sells seashells by the seashore.*
- `text2` — *"Seashells! The seashells are on sale! By the seashore."*
- `text3` — *She sells 3 seashells to John, who lives by the lake.*

Наша цель — определить, с каким из выражений, `text2` или `text3`, больше совпадает `text1`. Начнем с присваивания текстов трем переменным (листинг 13.1).

Листинг 13.1. Присваивание выражений переменным

```
text1 = 'She sells seashells by the seashore.'
text2 = '"Seashells! The seashells are on sale! By the seashore."'
text3 = 'She sells 3 seashells to John, who lives by the lake.'
```

Теперь нужно численно оценить различия между текстами. Как вариант, можно просто подсчитать слова, содержащиеся в каждой паре текстов. Для этого потребуется разбить каждый текст на список слов (листинг 13.2). В Python за эту операцию отвечает встроенный метод `split`.

ПРИМЕЧАНИЕ

Разбивка текста на отдельные слова называется токенизацией.

Листинг 13.2. Разбивка текста на слова

```
words_lists = [text.split() for text in [text1, text2, text3]]
words1, words2, words3 = words_lists

for i, words in enumerate(words_lists, 1):
    print(f"Words in text {i}")
    print(f"{words}\n")

Words in text 1
['She', 'sells', 'seashells', 'by', 'the', 'seashore.']

Words in text 2
['"Seashells!', 'The', 'seashells', 'are', 'on', 'sale!', 'By', 'the',
'seashore."']

Words in text 3
['She', 'sells', '3', 'seashells', 'to', 'John,', 'who', 'lives', 'by', 'the',
'lake.']
```

Но даже после разбивки текста точное сопоставление слов выполнить сразу не получится. На то есть две причины:

- *несогласованность прописных букв* — слова *she* и *seashells* в одних текстах пишутся с прописной буквы, а в других — со строчной, что усложняет сравнение;
- *несогласованность пунктуации* — например, в `text2` слово *seashells* сопровождается восклицательным знаком и кавычками, обозначающими начало цитаты, чего в других текстах не наблюдается.

Несогласованность прописных букв можно устранить вызовом встроенного строкового метода `lower`, который переводит все строки в нижний регистр. Более того, можно убрать всю пунктуацию из строки `word`, вызвав `word.replace(punctuation, '')`, где `punctuation` устанавливается на `'!'` или `'"'`. Теперь можно применить описанные методы, чтобы избавиться от всех этих несогласованностей (листинг 13.3). Мы определим функцию `simplify_text`, которая преобразует текст в нижний регистр и удалит из него всю пунктуацию.

Листинг 13.3. Устранение разницы в регистрах и пунктуации

```
def simplify_text(text):                                    ← Удаляет из строки всю пунктуацию
    for punctuation in ['.', ',', '!', '?', '"']:             и переводит ее в нижний регистр
        text = text.replace(punctuation, '')

    return text.lower()

for i, words in enumerate(words_lists, 1):
    for j, word in enumerate(words):
        words[j] = simplify_text(word)

    print(f"Words in text {i}")
    print(f"{words}\n")
```

Напомню, что наша основная цель такова.

1. Подсчитать в `text1` все уникальные слова, которые имеются и в `text2`.
2. Подсчитать в `text1` все уникальные слова, которые есть и в `text3`.
3. Используя результаты подсчетов, определить, какой текст — `text2` или `text3` — больше совпадает с `text1`.

Сейчас нас интересует лишь сравнение уникальных слов. Значит, повторяющиеся слова вроде *seashore*, которое в `text2` встречается дважды, будут посчитаны по одному разу. Для исключения повторов можно преобразовать каждый список слов в их множество (листинг 13.4).

Листинг 13.4. Преобразование списков слов в их множества

```
words_sets = [set(words) for words in words_lists]
for i, unique_words in enumerate(words_sets, 1):
    print(f"Unique Words in text {i}")
    print(f"{unique_words}\n")

Unique Words in text 1
{'sells', 'seashells', 'by', 'seashore', 'the', 'she'}

Unique Words in text 2
{'by', 'on', 'are', 'sale', 'seashore', 'the', 'seashells'}

Unique Words in text 3
{'to', 'sells', 'seashells', 'lake', 'by', 'lives', 'the', 'john', '3', 'who', 'she'}
```

Имея два множества Python, `set_a` и `set_b`, можно извлечь все совпадающие элементы, выполнив `set_a & set_b` (листинг 13.5). Оператор `&` используется для подсчета совпадающих слов между парами текстов (`text1`, `text2`) и (`text1`, `text3`).

ПРИМЕЧАНИЕ

Строго говоря, множество совпадающих элементов называется пересечением двух множеств.

Листинг 13.5. Извлечение совпадающих слов между двумя текстами

```
words_set1 = words_sets[0]
for i, words_set in enumerate(words_sets[1:], 2):
    shared_words = words_set1 & words_set
    print(f"Texts 1 and {i} share these {len(shared_words)} words:")
    print(f"{shared_words}\n")

Texts 1 and 2 share these 4 words:
{'seashore', 'by', 'the', 'seashells'}

Texts 1 and 3 share these 5 words:
{'sells', 'by', 'she', 'the', 'seashells'}
```

Обнаружены четыре общих слова между текстами 1 и 2 и пять — между текстами 1 и 3. Означает ли это, что `text1` больше похож на `text3`, чем на `text2`? Не обязательно. Несмотря на то что тексты 1 и 3 имеют пять совпадающих слов, они содержат и слова, определяющие их расхождение, которые есть в одном тексте, но отсутствуют в другом. Давайте подсчитаем все такие слова между парами текстов (`text1, text2`) и (`text1, text3`) (листинг 13.6). Для извлечения этих слов задействуем оператор `^`.

Листинг 13.6. Извлечение слов, определяющих расхождение двух текстов

```
for i, words_set in enumerate(words_sets[1:], 2):
    diverging_words = words_set1 ^ words_set
    print(f"Texts 1 and {i} don't share these {len(diverging_words)} words:")
    print(f"{diverging_words}\n")

Texts 1 and 2 don't share these 5 words:
{'are', 'sells', 'sale', 'on', 'she'}

Texts 1 and 3 don't share these 7 words:
{'to', 'lake', 'lives', 'seashore', 'john', '3', 'who'}
```

Тексты 1 и 3 содержат на два слова, определяющих расхождение, больше, чем тексты 1 и 2. Значит, тексты 1 и 3 демонстрируют как значительное совпадение, так и значительное расхождение. Чтобы интегрировать эти показатели в единый показатель сходства, нужно сначала совместить все совпадающие и все различные слова этих текстов. Такая агрегация, называемая *объединением*, будет содержать все уникальные слова двух текстов. Имея два множества Python, `set_a` и `set_b`, можно вычислить их объединение, выполнив `set_a | set_b`.

Расхождение в словах, их пересечение и объединение показаны на рис. 13.1. Здесь уникальные слова из текстов 1 и 2 отображены в трех прямоугольниках. Левый

и правый представляют слова, определяющие расхождение между текстами 1 и 2 соответственно. Средний же содержит все общие слова — пересечение тех же текстов 1 и 2. Все вместе эти три прямоугольника представляют объединение всех слов двух текстов.

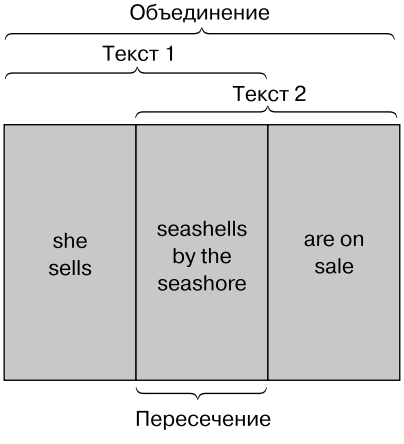


Рис. 13.1. Представление объединения и пересечения двух текстов, а также расхождения между ними

ТИПИЧНЫЕ ОПЕРАЦИИ НАД МНОЖЕСТВАМИ В PYTHON

- `set_a & set_b` — возвращает все совпадающие элементы `set_a` и `set_b`.
- `set_a set_b` — возвращает все элементы, определяющие расхождение между `set_a` и `set_b`.
- `set_a | set_b` — возвращает объединение всех элементов `set_a` и `set_b`.
- `set_a - set_b` — возвращает все элементы `set_a`, которых нет в `set_b`.

Задействуем оператор | для подсчета общего числа уникальных слов в парах текстов (`text1, text2`) и (`text1, text3`) (листинг 13.7).

Листинг 13.7. Извлечение объединения слов двух текстов

```
for i, words_set in enumerate(words_sets[1:], 2):
    total_words = words_set1 | words_set
    print(f"Together, texts 1 and {i} contain {len(total_words)} "
          f"unique words. These words are:\n {total_words}\n")
```

```
Together, texts 1 and 2 contain 9 unique words. These words are:
{'sells', 'seashells', 'by', 'on', 'are', 'sale', 'seashore', 'the', 'she'}

Together, texts 1 and 3 contain 12 unique words. These words are:
{'sells', 'lake', 'by', 'john', 'the', 'she', 'to', 'lives', 'seashore',
    '3', 'who', 'seashells'}
```

В целом в `text1` и `text3` содержатся 12 уникальных слов. Пять из них повторяются, а семь — различаются. Таким образом, повторение и расхождение представляют дополняющие друг друга проценты от общего количества уникальных слов в текстах. Давайте выведем эти процентные соотношения для пар текстов (`text1, text2`) и (`text1, text3`) (листинг 13.8).

Листинг 13.8. Определение соотношения общих и различающихся слов двух текстов в процентах

```
for i, words_set in enumerate(words_sets[1:], 2):
    shared_words = words_set1 & words_set
    diverging_words = words_set1 ^ words_set
    total_words = words_set1 | words_set
    assert len(total_words) == len(shared_words) + len(diverging_words)
    percent_shared = 100 * len(shared_words) / len(total_words)
    percent_diverging = 100 * len(diverging_words) / len(total_words)

    print(f"Together, texts 1 and {i} contain {len(total_words)} "
          f"unique words. \n{percent_shared:.2f}% of these words are "
          f"shared. \n{percent_diverging:.2f}% of these words diverge.\n")
```

Процент слов, совпадающих с text1, от общего числа

Процент слов, расходящихся с text1, от общего числа

```
Together, texts 1 and 2 contain 9 unique words.
44.44% of these words are shared.
55.56% of these words diverge.

Together, texts 1 and 3 contain 12 unique words.
41.67% of these words are shared.
58.33% of these words diverge.
```

В текстах 1 и 3 совпадает 41,67 % слов. Оставшиеся 58,33 % слов — различные. При этом между текстами 1 и 2 общих слов зафиксировано 44,44 % от всех имеющихся. Этот показатель выше, из чего следует, что `text1` имеет больше общего с `text2`, чем с `text3`.

По сути, мы выработали простую процедуру анализа сходства текстов. Работает она так.

1. Имея два текста, мы извлекаем список слов каждого из них.
2. Подсчитываем уникальные слова, содержащиеся в обоих текстах.
3. Делим количество общих слов на число уникальных слов обоих текстов, получая дробный показатель, отражающий долю общих уникальных слов, имеющихся в обоих текстах.

Этот показатель сходства называется *мерой Жаккара*, или *коэффициентом Жаккара*.

Коэффициент Жаккара между текстами 1 и 2 показан на рис. 13.2, где они представлены в виде двух кругов. Левый соответствует тексту 1, а правый — тексту 2. Каждый круг содержит слова из соответствующего текста. Эти два круга пересекаются, и в области пересечения содержатся все слова, имеющиеся в обоих текстах. Коэффициент Жаккара равен доле, которую слова из пересечения занимают в общем числе уникальных слов. В данном случае пересечение содержит четыре из девяти слов, поэтому коэффициент Жаккара равен 4/9.

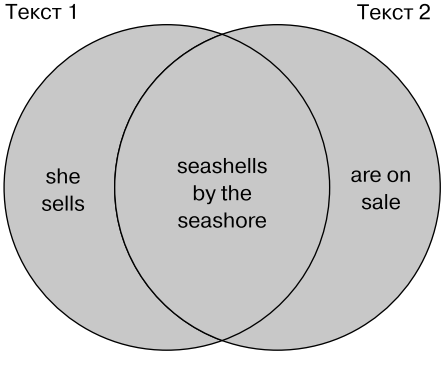


Рис. 13.2. Представление коэффициента Жаккара между двумя текстами

13.1.1. Изучение коэффициента Жаккара

Коэффициент Жаккара представляет собой разумную меру сходства текстов по следующим причинам.

- В нем учитываются и совпадения, и расхождения текстов.
- Дробный показатель сходства всегда находится между 0 и 1. Эту дробь легко интерпретировать: 0 означает, что общих слов нет, 0,5 — что совпадает половина слов, а 1 указывает на совпадение всех слов.
- Вычислять коэффициент Жаккара легко.

В листинге 13.9 приводится определение соответствующей функции.

Листинг 13.9. Вычисление коэффициента Жаккара

```
def jaccard_similarity(text_a, text_b):
    word_set_a, word_set_b = [set(simplify_text(text).split())
                              for text in [text_a, text_b]]
    num_shared = len(word_set_a & word_set_b)
    num_total = len(word_set_a | word_set_b)
    return num_shared / num_total
```

```
for text in [text2, text3]:
    similarity = jaccard_similarity(text1, text)
    print(f"The Jaccard similarity between '{text1}' and '{text}' "
          f"equals {similarity:.4f}." "\n")
```

```
The Jaccard similarity between 'She sells seashells by the seashore.' and
'"Seashells! The seashells are on sale! By the seashore."' equals 0.4444.

The Jaccard similarity between 'She sells seashells by the seashore.' and 'She
sells 3 seashells to John, who lives by the lake.' equals 0.4167.
```

Наша реализация получилась функциональной, но не особо эффективной. Данная функция выполняет две операции сравнения множеств: `word_set_a & word_set_b` и `word_set_a | word_set_b`. Они сравнивают все слова двух множеств и находят, в чем их сходство и расхождение. В Python подобные действия вычислительно затратнее, чем упорядоченный численный анализ.

Как же сделать функцию более эффективной? Для начала можно убрать вычисление объединения `word_set_a | word_set_b`. Мы выполняем его для подсчета уникальных слов двух множеств, но это можно сделать и более просто. Рассмотрите следующий вариант.

1. Сложение `len(word_set_a)` и `len(word_set_b)` дает количество слов, в котором общие слова подсчитаны дважды.
2. Вычитание `len(word_set_a & word_set_b)` из этой суммы устранит двойной счет, и итоговый результат будет равен `len(word_set_a | word_set_b)`.

Вычисление объединения можно заменить на `len(word_set_a) + len(word_set_b) – num_shared`, повысив тем самым эффективность функции. Теперь изменим ее, проследив, чтобы полученный коэффициент Жаккара остался прежним (листинг 13.10).

Листинг 13.10. Эффективное вычисление коэффициента Жаккара

```
def jaccard_similarity_efficient(text_a, text_b):
    word_set_a, word_set_b = [set(simplify_text(text).split())
                              for text in [text_a, text_b]]
    num_shared = len(word_set_a & word_set_b)
    num_total = len(word_set_a) + len(word_set_b) - num_shared
    return num_shared / num_total

for text in [text2, text3]:
    similarity = jaccard_similarity_efficient(text1, text)
    assert similarity == jaccard_similarity(text1, text)
```

В отличие от прежней функции jaccard_similarity, здесь вычисляем num_total без выполнения каких-либо операций сравнения множеств

Мы улучшили функцию вычисления коэффициента Жаккара. К сожалению, она по-прежнему не будет масштабироваться — может выполняться эффективно для сотен предложений, но не тысяч больших документов. Это вызвано сохранившимся сравнением множеств, `word_set_a & word_set_b`. Возможно, нам удастся ускорить

вычисление, проделав его с помощью NumPy. Вот только NumPy работает с числами, а не со словами, так что использовать эту библиотеку мы сможем, только если заменим все слова численными значениями.

13.1.2. Замена слов численными значениями

Можно ли поменять наши слова на числа? Да! Нужно просто перебрать все слова во всех текстах и присвоить каждому уникальному слову значение `i`. Карту сопоставлений между словами и их численными значениями можно сохранить в словаре Python. Мы создадим *словарь*, охватывающий все слова трех наших текстов. А также сформируем дополнительный словарь `value_to_word`, отображающий численные значения обратно в слова.

ПРИМЕЧАНИЕ

По сути, мы нумеруем все слова в объединении текстов. Для этого итеративно перебираем слова и присваиваем каждому из них число, начиная с нуля. При этом порядок, в котором происходит выбор слов, значения не имеет, то есть можно вслепую доставать слова из набора в случайной очередности. Называется эта техника bag-of-words[1].

Листинг 13.11. Присваивание чисел словам в словаре

```
words_set1, words_set2, words_set3 = words_sets
total_words = words_set1 | words_set2 | words_set3
vocabulary = {word : i for i, word in enumerate(total_words)}
value_to_word = {value: word for word, value in vocabulary.items()}
print(f"Our vocabulary contains {len(vocabulary)} words. "
      f"This vocabulary is:\n{vocabulary}")

Our vocabulary contains 15 words. This vocabulary is:
{'sells': 0, 'seashells': 1, 'to': 2, 'lake': 3, 'who': 4, 'by': 5,
 'on': 6, 'lives': 7, 'are': 8, 'sale': 9, 'seashore': 10, 'john': 11,
 '3': 12, 'the': 13, 'she': 14}
```

ПРИМЕЧАНИЕ

Порядок слов в переменной total_words в листинге 13.11 может различаться в зависимости от установленной версии Python. Это варьирование порядка будет немного изменять рисунки, используемые для отображения текстов далее в этой главе. Установка total_words равной ['sells', 'seashells', 'to', 'lake', 'who', 'by', 'on', 'lives', 'are', 'sale', 'seashore', 'john', '3', 'the', 'she'] гарантирует согласованность выходных результатов.

Используя полученный словарь, можно преобразовать любой текст в одномерный массив чисел. В математике одномерный массив называется *вектором*, поэтому преобразование текста в вектор называется *векторизацией текста*.

---

[1] Дословно «мешок слов» или «набор слов». — *Примеч. пер.*

ПРИМЕЧАНИЕ

Размерность массива отличается от размерности данных. Точка данных имеет d измерений, если для ее пространственного представления требуется d координат. При этом массив имеет d измерений, если для описания его формы требуется d значений. Представьте, что мы записали пять точек данных, каждую с помощью трех координат. Наши данные будут трехмерными, поскольку их можно графически отобразить в трехмерном пространстве. Кроме того, мы можем сохранять эти данные в таблице, содержащей пять строк и три столбца. По форме она является двухэлементной (5, 3), в связи с чем считается двухмерной. Выходит, что мы сохраняем трехмерные данные в двухмерном массиве.

Проще всего векторизовать текст созданием вектора из двоичных элементов. Каждый индекс вектора будет соответствовать слову в словаре. Получается, что размер вектора равен размеру словаря, даже если некоторые слова последнего в связанном тексте отсутствуют. Если слова по индексу `i` в тексте нет, элемент *i* вектора устанавливается на 0. В противном случае он устанавливается на 1. В итоге каждый индекс словаря в векторе сопоставляется с 0 или 1.

Например, в `vocabulary` слово *john* сопоставляется со значением `11`. Кроме того, слово *john* отсутствует в `text1`. Значит, по индексу 11 векторизованного представления `text1` будет храниться значение `0`. При этом в `text3` слово *john* есть, в связи с чем по индексу 11 векторного представления `text3` будет сохранено значение `1` (рис. 13.3). Таким образом можно преобразовать любой текст в двоичный вектор из 0 и 1.

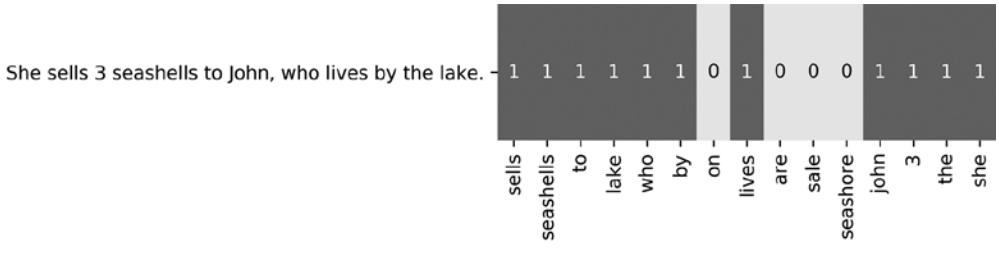


Рис. 13.3. Text3 преобразован в двоичный вектор. Каждый индекс последнего соответствует слову из словаря. Например, индекс 0 соответствует sells. Это слово в тексте имеется, значит, первый элемент вектора устанавливается на 1. При этом слов on, are, sale и seashore в тексте нет, поэтому соответствующие им элементы в векторе устанавливаются на 0

Теперь с помощью двоичной векторизации преобразуем все тексты в массивы NumPy (листинг 13.12). Мы сохраним вычисленные векторы в двухмерном списке `vectors`, который можно рассматривать как таблицу. Ее строки будут сопоставляться с текстами, а столбцы — со словарем. На рис. 13.4 данная таблица представлена в виде тепловой карты при помощи техник, рассмотренных в главе 8.

ПРИМЕЧАНИЕ

Как говорилось в главе 8, тепловые карты лучше всего визуализировать с помощью библиотеки Seaborn.

Листинг 13.12. Преобразование слов в двоичные векторы

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

vectors = []
for i, words_set in enumerate(words_sets, 1):
    vector = np.array([0] * len(vocabulary))
    for word in words_set:
        vector[vocabulary[word]] = 1
    vectors.append(vector)

sns.heatmap(vectors, annot=True, cmap='YlGnBu',
            xticklabels=vocabulary.keys(),
yticklabels=['Text 1', 'Text 2', 'Text 3'])
plt.yticks(rotation=0)
plt.show()
```

Генерирует массив 0. Его можно сгенерировать также выполнением np.zeros(len(vocabulary))

В Python 3.6 метод keys возвращает ключи словаря в соответствии с порядком их включения в него. В словаре порядок включения соответствует индексу слова

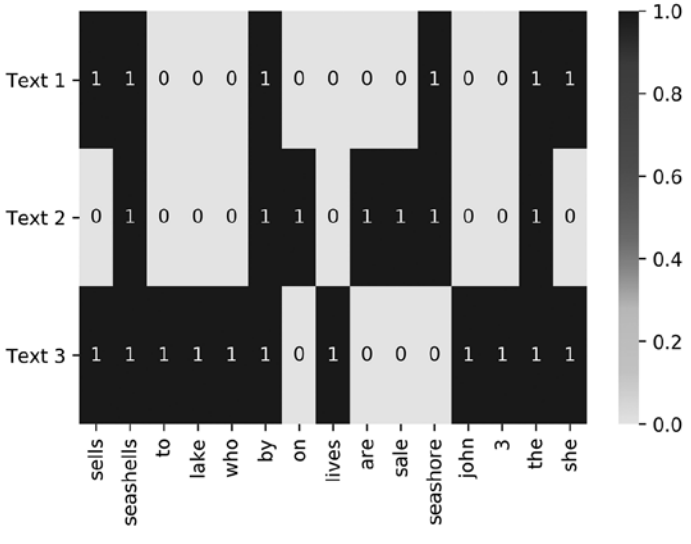


Рис. 13.4. Таблица векторизованных текстов. Строки соответствуют размеченным текстам, столбцы — размеченным словам. Двоичные элементы таблицы представлены 0 либо 1. Ненулевое значение говорит о том, что данное слово в тексте присутствует. Просматривая таблицу, можно с ходу понять, какие слова каких текстов совпадают

Используя полученную таблицу, можно без проблем определить, какие слова являются общими для каких текстов. Возьмем, к примеру, слово *sells*, относящееся к первому столбцу таблицы. В первой и третьей строках таблицы ему присвоена 1. Эти строки соответствуют `text1` и `text3`, значит, *sells* является для этих текстов общим словом. Говоря строже, это слово общее для данных текстов, потому что `vectors[0][0] == 1` и `vectors[2][0] == 1`. Более того, поскольку оба элемента равны 1, их произведение также должно равняться 1. Следовательно, слово является общим для слова в столбце `i`, если произведение `vectors[0][i]` и `vectors[2][i]` равно 1.

Такое двоичное векторное представление позволяет извлекать общие слова по числам. Предположим, нам нужно узнать, присутствует ли слово из столбца `i` в `text1` и `text2`. При условии, что соответствующие векторы обозначены `vector1` и `vector2`, это слово будет присутствовать в обоих текстах, если `vector1[i] * vector2[i] == 1`. Здесь для нахождения всех общих слов в `text1` и `text2` мы используем попарное векторное умножение (листинг 13.13).

Листинг 13.13. Поиск общих слов с помощью векторной арифметики

```
vector1, vector2 = vectors[:2]
for i in range(len(vocabulary)):
    if vector1[i] * vector2[i]:
        shared_word = value_to_word[i]
        print(f"'{shared_word}' is present in both texts 1 and 2")
```

```
'seashells' is present in both texts 1 and 2
'by' is present in both texts 1 and 2
'seashore' is present in both texts 1 and 2
'the' is present in both texts 1 and 2
```

Мы вывели все четыре слова, присутствующие в `text1` и `text2`. Количество общих слов равно сумме всех ненулевых экземпляров `vector1[i] * vector2[i]`. При этом сумма всех нулевых экземпляров равна 0. Следовательно, количество общих слов можно вычислить, просто сложив произведение `vector1[i]` и `vector2[i]` для всех возможных `i`. Иными словами, `sum(vector1[i] * vector2[i] for i in range(len(vocabulary)))` равно `len(words_set1 & words_set2)` (листинг 13.14).

Листинг 13.14. Подсчет общих слов с помощью векторной арифметики

```
shared_word_count = sum(vector1[i] * vector2[i]
                        for i in range(len(vocabulary)))
assert shared_word_count == len(words_set1 & words_set2)
```

Сумма попарных произведений для всех индексов вектора называется скалярным произведением. Имея два массива NumPy, `vector_a` и `vector_b`, можно вычислить их скалярное произведение, выполнив `vector_a.dot(vector_b)`. Его можно получить также с помощью оператора `@`, выполнив `vector_a @ vector_b`. В нашем примере

скалярное произведение равно количеству слов, общих для текстов 1 и 2, которое, естественно, соответствует размеру их пересечения. Значит, выполнение `vector_a @ vector_b` даст значение, равное `len(words_set1 & words_set2)` (листинг 13.15).

Листинг 13.15. Вычисление скалярного произведения вектора с помощью NumPy

```
assert vector1.dot(vector2) == shared_word_count
assert vector1 @ vector2 == shared_word_count
```

Скалярное произведение `vector1` и `vector2` равно числу слов, общих для `text1` и `text2`. Предположим, что вместо этого мы получаем скалярное произведение `vector1` с самим собой. Его результат должен равняться числу слов, общих для `text1` и `text1`. Говоря точнее, результат операции `vector1 @ vector1` должен быть равен количеству уникальных слов в `text1`, которое также равно `len(words_set1)`. Проверим (листинг 13.16).

Листинг 13.16. Подсчет общего числа слов с помощью векторной арифметики

```
assert vector1 @ vector1 == len(words_set1)
assert vector2 @ vector2 == len(words_set2)
```

Мы можем вычислить как количество общих слов, так и общее число уникальных слов, используя скалярные произведения векторов. По сути, можем найти коэффициент Жаккара посредством только векторных операций. Такая векторизованная реализация вычисления меры Жаккара называется *коэффициентом Танимото*.

ПОЛЕЗНЫЕ ВЕКТОРНЫЕ ОПЕРАЦИИ В NUMPY

- `vector_a.dot(vector_b)` — возвращает скалярное произведение `vector_a` и `vector_b`. Равноценно выполнению `sum(vector_a[i] * vector_b[i] for i in range(vector_a.size))`.
- `vector_b @ vector_b` — возвращает скалярное произведение `vector_a` и `vector_b`, используя оператор `@`.
- `binary_text_vector_a @ binary_text_vector_b` — возвращает количество слов, общих для `text_a` и `text_b`.
- `binary_text_vector_a @ binary_text_vector_a` — возвращает количество уникальных слов в `text_a`.

Теперь определим функцию `tanimoto_similarity`. Она получает два вектора, `vector_a` и `vector_b`, а ее вывод равен `jaccard_similarity (text_a, text_b)` (листинг 13.17).

Листинг 13.17. Вычисление сходства текстов с помощью векторной арифметики

```
def tanimoto_similarity(vector_a, vector_b):
    num_shared = vector_a @ vector_b
    num_total = vector_a @ vector_a + vector_b @ vector_b - num_shared
    return num_shared / num_total

for i, text in enumerate([text2, text3], 1):
    similarity = tanimoto_similarity(vector1, vectors[i]) assert
    similarity == jaccard_similarity(text1, text)
```

Наша функция `tanimoto_similarity` предназначалась для сравнения двоичных векторов. А что произойдет, если передать ей два массива со значениями, отличными от 0 и 1? Чисто технически она должна вернуть показатель сходства, но будет ли он иметь смысл? К примеру, векторы `[5, 3]` и `[5, 2]` практически одинаковы, поэтому ожидается, что их сходство будет почти равным 1. Проверим это, передав данные вектора в `tanimoto_similarity` (листинг 13.18).

Листинг 13.18. Вычисление сходства недвоичных векторов

```
non_binary_vector1 = np.array([5, 3])
non_binary_vector2 = np.array([5, 2])
similarity = tanimoto_similarity(non_binary_vector1, non_binary_vector2)
print(f"The similarity of 2 non-binary vectors is {similarity}")

The similarity of 2 non-binary vectors is 0.96875
```

Полученный результат очень близок к 1. Значит, `tanimoto_similarity` успешно измерила сходство между двумя почти одинаковыми векторами. Эта функция может анализировать недвоичные входные данные, что позволяет нам использовать недвоичные техники для векторизации текстов перед сравнением их содержимого.

Недвоичная векторизация текстов имеет свои преимущества, которые мы разберем далее.

13.2. ВЕКТОРИЗАЦИЯ ТЕКСТОВ С ПОМОЩЬЮ ПОДСЧЕТА СЛОВ

Двоичная векторизация отражает присутствие и отсутствие слов в тексте, но, к сожалению, не включает количество слов. Наличие информации об их количестве позволило бы обеспечить дифференцирующий сигнал между текстами. Предположим, что мы противопоставляем два текста, A и B. В тексте A слово *Duck* встречается 61 раз, а слово *Goose* — дважды. При этом в тексте B *Goose* упоминается 71 раз, а слово *Duck* — всего один. Исходя из этих расчетов, можно сделать вывод, что тексты сильно различаются в плане упоминания в них уток (duck) и гусей (goose). Это различие не отражается двоичной векторизацией, которая индексам

слов *Duck* и *Goose* из обоих текстов присваивает 1. А что, если заменить все двоичные значения реальным количеством слов? Например, индексам *Duck* и *Goose* в векторе A можно присвоить значения 61 и 2, а их индексам в векторе B — 1 и 71.

В результате этого получатся векторы, отражающие количества вхождений слова. Такой вектор принято называть *вектором частоты встречаемости термина*, или, кратко, *вектором TF*. Вычислим векторы TF для текстов A и B, используя двухэлементный словарь `{'duck': 0, 'goose': 1}` (листинг 13.19). Напомню, что каждое слово в словаре сопоставляется с индексом вектора. Имея словарь, можно преобразовать тексты в векторы TF `[61, 2]` и `[1, 71]`. Затем мы выведем коэффициент Танимото для этих двух векторов.

Листинг 13.19. Вычисление сходства векторов TF

```
similarity = tanimoto_similarity(np.array([61, 2]), np.array([1, 71]))
print(f"The similarity between texts is approximately {similarity:.3f}")

The similarity between texts is approximately 0.024
```

Сходство векторов TF между рассматриваемыми текстами очень мало. Сравним его со сходством двоичных векторов этих же текстов (листинг 13.20). Каждый текст имеет двоично-векторное представление `[1, 1]`, то есть их двоичное сходство должно равняться 1.

Листинг 13.20. Анализ сходства идентичных векторов

```
assert tanimoto_similarity(np.array([1, 1]), np.array([1, 1])) == 1
```

Замена двоичных значений количествами слов может существенно повлиять на итоговый показатель сходства. Что произойдет, если векторизовать `text1`, `text2` и `text3` на основе количеств слов? Давайте это выясним. Начнем с вычисления векторов TF для каждого текста, используя списки слов, сохраненные в `words_lists` (листинг 13.21). Эти векторы показаны на рис. 13.5 с помощью тепловой карты.

Листинг 13.21. Вычисление векторов TF на основе списков слов

```
tf_vectors = []
for i, words_list in enumerate(words_lists, 1):
    tf_vector = np.array([0] * len(vocabulary))
    for word in words_list:
        word_index = vocabulary[word]
        tf_vector[word_index] += 1
    tf_vectors.append(tf_vector)

sns.heatmap(tf_vectors, cmap='YlGnBu', annot=True,
            xticklabels=vocabulary.keys(),
yticklabels=['Text 1', 'Text 2', 'Text 3'])
plt.yticks(rotation=0)
plt.show()
```

Обновляет количества слов, используя индекс слова в словаре

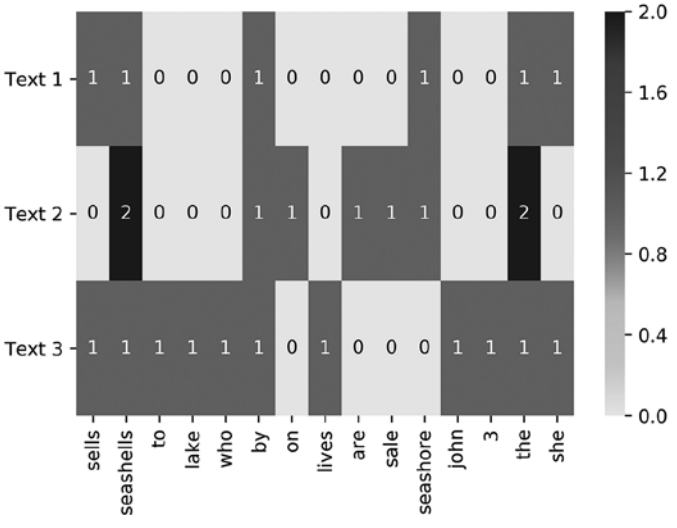


Рис. 13.5. Таблица векторов TF. Строки соответствуют размеченным текстам, а столбцы — размеченным словам. Каждое значение указывает количество вхождений определенного слова в конкретный текст. Два слова в таблице упоминаются дважды, все остальные встречаются не более одного раза

Векторы TF текстов 1 и 3 совпадают с полученными ранее двоичными векторами. А вот вектор TF текста 2 больше не является двоичным, поскольку два слова упоминаются в нем более одного раза. Как это повлияет на сходство между `text1` и `text2`? Сейчас разберемся. Код листинга 13.22 вычисляет сходство векторов TF между `text1` и двумя другими текстами. Кроме того, для сравнения он вычисляет сходство исходных двоичных векторов. Исходя из наших наблюдений, сходство между `text1` и `text2` должно измениться, а между `text1` и `text3` — остаться одинаковым.

Листинг 13.22. Сравнение показателей для определения сходства векторов

```
tf_vector1 = tf_vectors[0]
binary_vector1 = vectors[0]

for i, tf_vector in enumerate(tf_vectors[1:], 2):
    similarity = tanimoto_similarity(tf_vector1, tf_vector)
    old_similarity = tanimoto_similarity(binary_vector1, vectors[i - 1])
    print(f"The recomputed Tanimoto similarity between texts 1 and {i} is"
          f" {similarity:.4f}.")
    print(f"Previously, that similarity equaled {old_similarity:.4f} " "\n")
```

```
The recomputed Tanimoto similarity between texts 1 and 2 is 0.4615.
Previously, that similarity equaled 0.4444

The recomputed Tanimoto similarity between texts 1 and 3 is 0.4167.
Previously, that similarity equaled 0.4167
```

Как и ожидалось, сходство между `text1` и `text3` осталось одинаковым, а между `text1` и `text2` — увеличилось. Таким образом, векторизация TF сделала схожесть этих двух текстов более выраженной.

Векторы TF дают более точное сравнение, поскольку они чувствительны к количественным различиям текстов. Это полезное качество, но иногда оно может усложнить сравнение текстов разной длины. В следующем разделе мы рассмотрим недостаток, связанный со сравнением векторов TF, а после для его устранения применим технику, называемую *нормализацией*.

13.2.1. Повышение качества векторов частотности терминов с помощью нормализации

Представьте, что вы тестируете очень простой поисковый механизм. Он получает запрос и сравнивает его с заголовками документов из базы данных. Вектор TF этого запроса сопоставляется с каждым векторизованным заголовком. Заголовки с ненулевым коэффициентом Танимото возвращаются и ранжируются на основе степени сходства.

Предположим, вы выполняете запрос *Pepperoni Pizza* и получаете в ответ два заголовка:

- *заголовок A — Pepperoni Pizza! Pepperoni Pizza! Pepperoni Pizza!*;
- *заголовок B — Pepperoni.*

ПРИМЕЧАНИЕ

Эти заголовки умышленно упрощены для простоты визуализации. В большинстве реальных случаев заголовки документов сложнее.

Какой из этих заголовков больше соответствует запросу? Большинство ученых по данным согласились бы, что более соответствующим является заголовок A, поскольку и в нем, и в запросе упоминается *pepperoni pizza*, в то время как в B присутствует только *pepperoni*. Большинство аналитиков данных согласятся, что связанный с ними документ описывает пиццу в любом контексте.

Давайте проверим, оказывается ли заголовок A более релевантным нашему запросу, чем заголовок B. Начнем с построения векторов TF на основе двухэлементного словаря `{pepperoni: 0, pizza: 1}` (листинг 13.23).

Листинг 13.23. Векторизация простого механизма поиска

```
query_vector = np.array([1, 1])
title_a_vector = np.array([3, 3])
title_b_vector = np.array([1, 0])
```

Теперь сравним запрос с заголовками и упорядочим их на основе коэффициента Танимото (листинг 13.24).

Листинг 13.24. Ранжирование заголовков по степени сходства с запросом

```
titles = ["A: Pepperoni Pizza! Pepperoni Pizza! Pepperoni Pizza!", "B: Pepperoni"]
title_vectors = [title_a_vector, title_b_vector]
similarities = [tanimoto_similarity(query_vector, title_vector)
                for title_vector in title_vectors]

for index in sorted(range(len(titles)), key=lambda i: similarities[i],
                    reverse=True):
    title = titles[index]
    similarity = similarities[index]
    print(f"'{title}' has a query similarity of {similarity:.4f}")
```

```
'B: Pepperoni' has a query similarity of 0.5000
'A: Pepperoni Pizza! Pepperoni Pizza! Pepperoni Pizza!' has a query similarity
     of 0.4286
```

К сожалению, заголовок A оказывается более релевантным, чем B. Это расхождение в ранжировании вызвано размером текста. Заголовок A содержит втрое больше слов, чем запрос, притом что заголовок B отличается от него всего на одно слово. На первый взгляд это различие можно использовать для дифференцирования текстов по размеру. Однако в нашем поисковом движке сигнал размера ведет к ложному ранжированию. Необходимо сгладить влияние размера текста на результаты. Простейший вариант — разделить `title_a_vector` на 3. Данная операция даст результат, равный `query_vector`. Следовательно, выполнение `tanimoto_similarity(query_vector, title_a_vector/ 3)` должно привести к возвращению показателя сходства, равного `1` (листинг 13.25).

Листинг 13.25. Устранение разницы размеров посредством деления

```
assert np.array_equal(query_vector, title_a_vector / 3)
assert tanimoto_similarity(query_vector,
                           title_a_vector / 3) == 1
```

Используя простое деление, можно уравнять `title_a vector` с `query_vector`. Для `title_b vector` подобную манипуляцию проделать не получится. Почему? Чтобы продемонстрировать ответ, нужно построить все три вектора в двухмерном пространстве (листинг 13.26).

Листинг 13.26. Построение векторов TF в двухмерном пространстве

```
plt.plot([0, query_vector[0]], [0, query_vector[1]], c='k',
         linewidth=3, label='Query Vector')
plt.plot([0, title_a_vector[0]], [0, title_a_vector[1]], c='b',
         linestyle='--', label='Title A Vector')
plt.plot([0, title_b_vector[0]], [0, title_b_vector[1]], c='g',
         linewidth=2, linestyle='-.', label='Title B Vector')
plt.xlabel('Pepperoni')
```

```
plt.ylabel('Pizza')
plt.legend()
plt.show()
```

Как можно визуализировать векторы? С математической точки зрения все они являются геометрическими объектами. Математики рассматривают каждый вектор v как линию, протянувшуюся от исходной точки до численных координат v. По сути, наши три вектора являются просто двухмерными отрезками, исходящими из точки отсчета. Можно визуализировать эти отрезки на 2D-графике, где ось *X* представляет число вхождений *pepperoni*, а ось *Y* — число вхождений *pizza* (рис. 13.6).

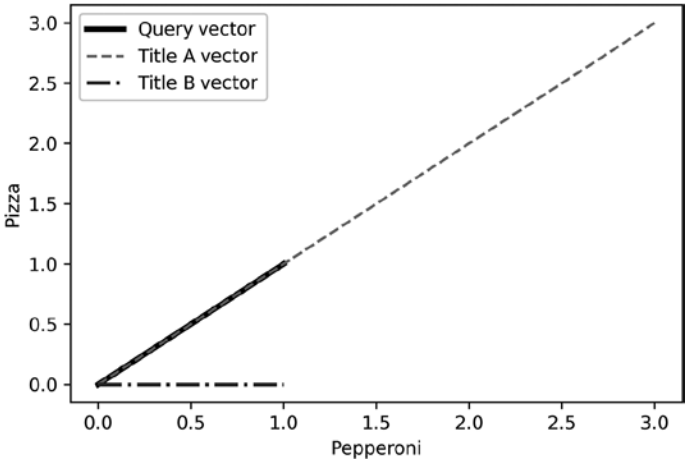


Рис. 13.6. Три вектора TF, построенные в виде линий в 2D-пространстве. Каждый вектор протягивается из точки отсчета до точки его двухмерных координат. У вектора запроса и вектора заголовка A одинаковое направление. Угол между ними равен нулю. Однако одна из этих линий в три раза длиннее другой. Корректировка длин отрезков приведет к выравниванию этих векторов

На нашем чертеже `title_a vector` и `query vector` указывают в одном направлении. Единственное различие между этими линиями в том, что `title_a vector` втрое длиннее. Уменьшение `title_a vector` приведет к уравниванию этих двух линий. При этом направления `title_b vector` и `query vector` разные, так что добиться наложения этих векторов не удастся. Уменьшение или увеличение `title_b vector` не приведет к его выравниванию с двумя другими отрезками.

Представление векторов в виде отрезков позволило нам кое-что понять. Отрезки имеют геометрическую длину. Это значит, что каждый вектор имеет геометрическую длину, которая называется *абсолютной величиной*. Также она называется *евклидовой нормой*, или *нормой L2*. Все векторы имеют абсолютную величину, даже те, которые нельзя построить в двух измерениях. К примеру, на рис. 13.7 мы иллюстрируем абсолютную величину 3D-вектора, ассоциированного с *Pepperoni Pizza Pie*.

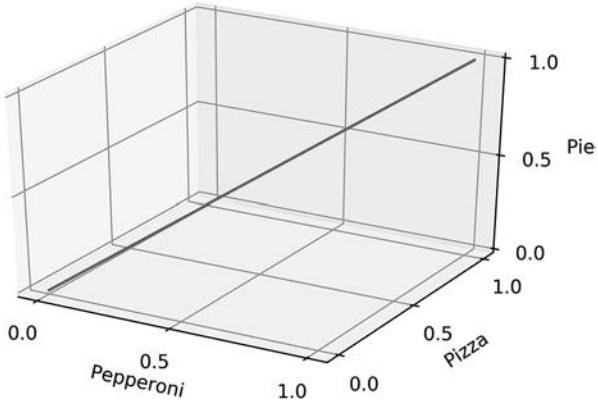
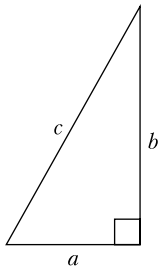


Рис. 13.7. Графическое представление вектора TF для заголовка из трех слов Pepperoni Pizza Pie. Этот 3D-вектор протягивается от исходной точки до своих координат (1, 1, 1). Согласно теореме Пифагора длина 3D-отрезка равна (1 + 1 + 1) ** 0,5. Она называется абсолютной величиной вектора

Измерение абсолютной величины позволяет учесть различия в длинах. Для вычисления абсолютной величины в Python есть несколько способов. Имея вектор `v`, можно узнать его абсолютную величину, просто измерив евклидово расстояние между `v` и началом координат. Найти абсолютную величину можно также с помощью NumPy, выполнив `np.linalg.norm(v)`. Наконец, ее можно вычислить, используя теорему Пифагора (рис. 13.8).



$a^2 + b^2 = c^2$

Рис. 13.8. Вычисление абсолютной величины вектора с помощью теоремы Пифагора. Как правило, двухмерный вектор [*a*, *b*] можно представить посредством прямоугольного треугольника. Перпендикулярные отрезки этого треугольника имеют длину *a* и *b*. При этом длина гипотенузы треугольника равна *c*. Согласно теореме Пифагора $c^2 = a^2 + b^2$. В связи с этим абсолютная величина равна sum([value * value for value in vector]) ** 0.5. Эта формула работает и за пределами двухмерного пространства вплоть до произвольного количества измерений

Согласно теореме Пифагора квадрат расстояния от координат `v` до исходной точки отсчета равен `sum([value * value for value in v])`. Это отлично согласуется с приведенным ранее определением скалярного произведения. Напомню, что скалярное произведение векторов `v1` и `v2` равно `sum([value1 * value2 for value1, value2 in zip(v1, v2)])`. Следовательно, скалярное произведение `v` с самим собой равно `sum([value * value for value in v])`. Отсюда следует, что абсолютная величина `v` равна `(v @ v) ** 0.5`.

Теперь выведем абсолютные величины векторов нашего механизма поиска. Исходя из наблюдений, абсолютная величина `title_a vector` должна быть втрое больше абсолютной величины `query vector` (листинг 13.27).

Листинг 13.27. Вычисление абсолютной величины вектора

```
from scipy.spatial.distance import euclidean
from numpy.linalg import norm

vector_names = ['Query Vector', 'Title A Vector', 'Title B Vector']
tf_search_vectors = [query_vector, title_a_vector, title_b_vector]
origin = np.array([0, 0])
for name, tf_vector in zip(vector_names, tf_search_vectors):
    magnitude = euclidean(tf_vector, origin)
    assert magnitude == norm(tf_vector)
    assert magnitude == (tf_vector @ tf_vector) ** 0.5
    print(f"{name}'s magnitude is approximately {magnitude:.4f}")

magnitude_ratio = norm(title_a_vector) / norm(query_vector)
print(f"\nVector A is {magnitude_ratio:.0f}x as long as Query Vector")
```

Абсолютная величина равна евклидову расстоянию между вектором и началом координат

Норма в NumPy возвращает абсолютную величину

Абсолютную величину можно вычислить также с помощью скалярного произведения

```
Query Vector's magnitude is approximately 1.4142
Title A Vector's magnitude is approximately 4.2426
Title B Vector's magnitude is approximately 1.0000

Vector A is 3x as long as Query Vector
```

Как и ожидалось, между абсолютными величинами `query vector` и `title_a vector` имеется трехкратное различие. Кроме того, абсолютные величины обоих этих векторов больше 1. При этом абсолютная величина `title_vector_b` составляет ровно 1. Вектор с абсолютной величиной 1 называется *единичным*. Такие векторы обладают множеством полезных свойств, которые мы вскоре рассмотрим. Одно из них состоит в простоте сравнения: поскольку единичные векторы имеют одинаковую абсолютную величину, она на их сходство не влияет. По сути, разница между единичными векторами определяется исключительно по их направлению.

Представьте, что `title_a vector` и `query vector` имеют абсолютную величину 1. В таком случае они будут равной длины и станут указывать в одном направлении. По существу, эти два вектора можно считать идентичными. Разница в количестве слов между запросом и заголовком A больше не будет иметь значения.

Чтобы это проиллюстрировать, мы преобразуем векторы TF в единичные векторы. Деление любого вектора на его абсолютную величину превращает эту величину в 1. Такая операция называется *нормализацией*, поскольку абсолютная величина называется также нормой L2. Выполнение `v / norm(v)` вернет *нормализованный вектор* с абсолютной величиной, равной 1.

Теперь нормализуем наши векторы и сгенерируем график единичных векторов (листинг 13.28; рис. 13.9). На этом графике два вектора должны быть идентичными.

Листинг 13.28. Построение графика нормализованных векторов

```
unit_query_vector = query_vector / norm(query_vector)
unit_title_a_vector = title_a_vector / norm(title_a_vector)
assert np.allclose(unit_query_vector, unit_title_a_vector)
unit_title_b_vector = title_b_vector

plt.plot([0, unit_query_vector[0]], [0, unit_query_vector[1]], c='k',
         linewidth=3, label='Normalized Query Vector')
plt.plot([0, unit_title_a_vector[0]], [0, unit_title_a_vector[1]], c='b',
         linestyle='--', label='Normalized Title A Vector')
plt.plot([0, unit_title_b_vector[0]], [0, unit_title_b_vector[1]], c='g',
         linewidth=2, linestyle='-.', label='Title B Vector')

plt.axis('equal')
plt.legend()
plt.show()
```

Это единичный вектор, нормализовывать его не нужно

Теперь эти два нормализованных вектора идентичны. Для подтверждения мы используем np.allclose, а не np.array_equal, чтобы компенсировать минимальные погрешности из-за плавающей точки, которые могут возникнуть при нормализации

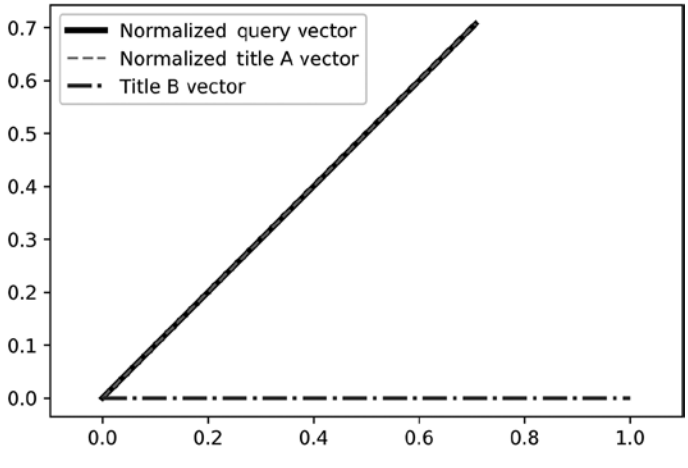


Рис. 13.9. Векторы после нормализации. Теперь все они имеют абсолютную величину 1. Нормализованный вектор запроса и нормализованный вектор заголовка A являются идентичными

Нормализованный вектор запроса и нормализованный вектор заголовка A теперь стали неотличимы друг от друга. Все различия, обусловленные разницей в размере

текста, были устранены. При этом расположение вектора заголовка B расходится с вектором запроса, поскольку эти два отрезка разнонаправленны. Если мы ранжируем единичные векторы на основе их сходства с `unit_query_vector`, то `unit_title_a_vector` превзойдет по сходству `unit_title_b_vector` (листинг 13.29). В результате заголовок A превосходит заголовок B относительно запроса.

Листинг 13.29. Ранжирование заголовков по сходству единичных векторов

```
unit_title_vectors = [unit_title_a_vector, unit_title_b_vector]
similarities = [tanimoto_similarity(unit_query_vector, unit_title_vector)
                for unit_title_vector in unit_title_vectors]

for index in sorted(range(len(titles)), key=lambda i: similarities[i],
                    reverse=True):
    title = titles[index]
    similarity = similarities[index]
    print(f"'{title}' has a normalized query similarity of {similarity:.4f}")

'A: Pepperoni Pizza! Pepperoni Pizza! Pepperoni Pizza!' has a normalized
 query similarity of 1.0000
'B: Pepperoni' has a normalized query similarity of 0.5469
```

ТИПИЧНЫЕ ОПЕРАЦИИ С АБСОЛЮТНОЙ ВЕЛИЧИНОЙ ВЕКТОРОВ

- `euclidean(vector, vector.size * [0])` — возвращает абсолютную величину вектора, равную евклидову расстоянию между `vector` и началом координат.
- `norm(vector)` — возвращает абсолютную величину вектора с помощью функции NumPy `norm`.
- `(vector @ vector) ** 0.5` — вычисляет абсолютную величину вектора, используя теорему Пифагора
- `vector / norm(vector)` — нормализует вектор, чтобы его абсолютная величина равнялась 1.0.

Нормализация векторов исправила недочет нашего механизма поиска, который теперь не является излишне чувствительным к длине заголовков. В ходе этого мы непроизвольно повысили эффективность вычисления коэффициента Танимото. Разберем почему.

Предположим, мы измеряем коэффициент Танимото для двух единичных векторов, `u1` и `u2`. Рассуждая логически, можно вывести следующее.

- Коэффициент Танимото равен `u1 @ u2 / (u1 @ u1 + u2 @ u2 – u1 @ u2)`.
- `u1 @ u1` равняется `norm(u1) ** 2`. Исходя из предыдущего обсуждения, мы знаем, что `u1 @ u1` равняется квадрату абсолютной величины `u`.

- `u1` — единичный вектор, значит, `norm(u1)` равняется `1`. Следовательно, `norm(u1) ** 2` равняется `1`. Таким образом, `u1 @ u1` равняется `1`.
- По той же логике `u2 @ u2` также равняется `1`.
- В результате вычисление коэффициента Танимото сокращается до `u1 @ u2 / (2 – u1 @ u2)`.

Получать скалярное произведение каждого вектора с самим собой больше не требуется. Единственное необходимое вычисление векторов — это `u1 @ u2`.

Теперь определим функцию `normalized_tanimoto`. Она будет получать два нормализованных вектора, `u1` и `u2`, и вычислять их коэффициент Танимото непосредственно из `u1 @ u2`. Результат этой операции будет равен `tanimoto_similarity(u1, u2)` (листинг 13.30).

Листинг 13.30. Вычисление коэффициента Танимото для единичного вектора

```
def normalized_tanimoto(u1, u2):
    dot_product = u1 @ u2
    return dot_product / (2 - dot_product)

for unit_title_vector in unit_title_vectors[1:]:
    similarity = normalized_tanimoto(unit_query_vector, unit_title_vector)
    assert similarity == tanimoto_similarity(unit_query_vector,
                                             unit_title_vector)
```

Скалярное произведение двух единичных векторов является особым значением. Его можно легко преобразовать в угол между векторами, а также в пространственное расстояние между ними. Почему это важно? Дело в том, что типичные геометрические параметры вроде угла между векторами и расстояния встречаются во всех библиотеках векторного анализа. При этом вне NLP коэффициент Танимото используется не так часто, поскольку обычно его нужно реализовывать с нуля, что может иметь серьезные последствия в реальных проектах. Представьте следующий сценарий. Компания, разрабатывающая поисковый движок, наняла вас, чтобы вы улучшили все запросы, связанные с пиццей. Вы намереваетесь применять в качестве показателя оценки релевантности запроса коэффициент Танимото. Однако ваш начальник возражает, ссылаясь на политику компании, согласно которой сотрудники должны использовать только те показатели, которые уже включены в scikit-learn.

ПРИМЕЧАНИЕ

Как ни печально, но это вполне реальный сценарий. Большинство организаций склонны проверять свои ключевые показатели на предмет скорости и качества. В крупных компаниях проверка может растягиваться на месяцы. Поэтому обычно проще опираться на уже проверенную библиотеку, чем тратить время на оценку нового показателя.

Руководитель указывает вам на документацию scikit-learn, в которой перечислены приемлемые функции показателей (http://mng.bz/9aM1). Вы видите названия показателей scikit-learn и функции, представленные в виде таблицы (рис. 13.10).

Несколько версий названия могут сопоставляться с одной функцией. Четыре показателя из восьми относятся к евклидову расстоянию, а три — к манхэттенскому расстоянию и формуле гаверсинуса (она же расстояние по ортодромии), с которыми мы познакомились в главе 11. Кроме того, в таблице упоминается еще не рассмотренный нами параметр `'cosine'`. В ней нет коэффициента Танимото, поэтому использовать его для оценки релевантности нельзя. Что же делать?

The valid distance metrics, and the function they map to, are:

metric	Function
'cityblock'	metrics.pairwise.manhattan_distances
'cosine'	metrics.pairwise.cosine_distances
'euclidean'	metrics.pairwise.euclidean_distances
'haversine'	metrics.pairwise.haversine_distances
'l1'	metrics.pairwise.manhattan_distances
'l2'	metrics.pairwise.euclidean_distances
'manhattan'	metrics.pairwise.manhattan_distances
'nan_euclidean'	metrics.pairwise.nan_euclidean_distances

Рис. 13.10. Скриншот документации scikit-learn с допустимыми реализациями параметров расстояния

К счастью, математика дает вам выход из этой ситуации. Если ваши векторы нормализованы, их коэффициент Танимото можно заменить евклидовым расстоянием и косинусным коэффициентом. Причина в том, что все эти три меры очень тесно связаны с нормализованным скалярным произведением. Давайте разберемся, почему так происходит.

13.2.2. Использование скалярного произведения единичных векторов для преобразования между параметрами релевантности

Скалярное произведение единичных векторов объединяет несколько типов параметров сравнения. Мы недавно определили, что `tanimoto_similarity(u1, u2)` является прямой функцией от `u1 @ u2`. Как выясняется, евклидово расстояние между единичными векторами также является функцией от `u1 @ u2`. Несложно доказать, что `euclidean(u1, u2)` равняется `(2 - 2* u1 @ u2) ** 0.5`. Кроме того, угол между линейными единичными векторами также зависит от `u1 @ u2`. Эти связи показаны на рис. 13.11.

Геометрически скалярное произведение двух единичных векторов равно косинусу угла между ними. Ввиду этого скалярное произведение двух единичных векторов принято называть *косинусным коэффициентом*, или *коэффициентом Отиаи*. Имея косинусный коэффициент `cs`, можно преобразовать его либо в евклидово расстояние, либо в коэффициент Танимото, выполнив `(2 - 2 * cs) ** 0.5` или `cs / (2 - cs)` соответственно.

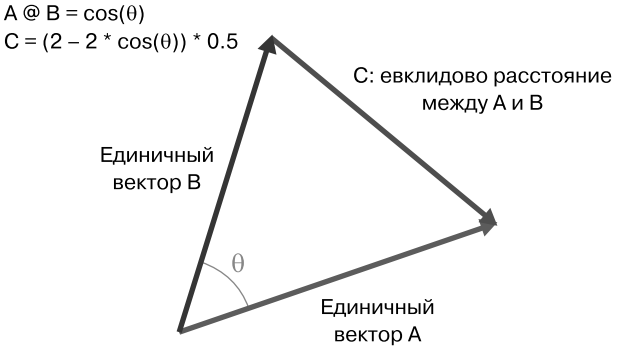


Рис. 13.11. Два единичных вектора, A и B. Угол между ними равен 0°. Их скалярное произведение равно cosine(θ). C представляет евклидово расстояние между векторами, равное $(2 - 2 \times \text{cosine}(\theta))^{0,5}$

ПРИМЕЧАНИЕ

Косинус — очень важная функция в тригонометрии. Он сопоставляет угол между линиями со значением в диапазоне от –1 до 1. Если две линии однонаправленны, то угол между ними равен 0°, а его косинус равен 1. Если же их направления различны, угол будет равен 180°, а его косинус составит –1. Имея пару векторов, v1 и v2, можно вычислить их косинусный коэффициент, выполнив (v1 / norm(v1)) @ (v2 / norm(v2)). Полученный результат мы передадим в функцию, обратную косинусу, np.arccos, чтобы измерить угол между векторами.

Листинг 13.31 показывает, как легко выполняется преобразование между коэффициентом Танимото, косинусным коэффициентом и евклидовым расстоянием. Мы вычисляем коэффициент Танимото между вектором запроса и каждым из единичных векторов заголовков. Затем полученный результат преобразуется в косинусный коэффициент, который, в свою очередь, преобразуется в евклидово расстояние.

ПРИМЕЧАНИЕ

Кроме того, мы используем косинусный коэффициент для вычисления угла между векторами. Таким образом мы подчеркиваем, что косинусный показатель отражает угол между отрезками.

Листинг 13.31. Преобразование между параметрами единичных векторов

```
unit_vector_names = ['Normalized Title A vector', 'Title B Vector']
u1 = unit_query_vector

for unit_vector_name, u2 in zip(unit_vector_names, unit_title_vectors):
    similarity = normalized_tanimoto(u1, u2)
```

```
    cosine_similarity = 2 * similarity / (1 + similarity)
    assert cosine_similarity == u1 @ u2
    angle =  np.arccos(cosine_similarity)
    euclidean_distance = (2 - 2 * cosine_similarity) ** 0.5
    assert round(euclidean_distance, 10) == round(euclidean(u1, u2), 10)
    measurements = {'Tanimoto similarity': similarity,
                    'cosine similarity': cosine_similarity,
                    'Euclidean distance': euclidean_distance,
                    'angle': np.degrees(angle)}

print("We are comparing Normalized Query Vector and "
      f"{unit_vector_name}")
for measurement_type, value in measurements.items():
    output = f"The {measurement_type} between vectors is {value:.4f}"
    if measurement_type == 'angle':
        output += ' degrees\n'
    print(output)
```

normalized_tanimoto — это функция cosine_similarity. Используя базовую алгебру, можно инвертировать эту функцию для решения cosine_similarity

```
We are comparing Normalized Query Vector and Normalized Title A vector
The Tanimoto similarity between vectors is 1.0000
The cosine similarity between vectors is 1.0000
The Euclidean distance between vectors is 0.0000
The angle between vectors is 0.0000 degrees

We are comparing Normalized Query Vector and Title B Vector
The Tanimoto similarity between vectors is 0.5469
The cosine similarity between vectors is 0.7071
The Euclidean distance between vectors is 0.7654
The angle between vectors is 45.0000 degrees
```

Коэффициент Танимото между нормализованными векторами можно трансформировать в другие параметры сходства или расстояния. Это дает следующие преимущества.

- Преобразование коэффициента Танимото в евклидово расстояние позволяет выполнить кластеризацию текстовых данных по методу *K*-средних. Кластеризация текстов по методу *K*-средних будет рассматриваться в главе 15.
- Преобразование коэффициента Танимото в косинусный коэффициент упрощает требования к вычислениям, которые в итоге сокращаются до базовых операций со скалярным произведением.

ПРИМЕЧАНИЕ

Специалисты, работающие с NLP, обычно вместо коэффициента Танимото используют косинусный коэффициент. Исследования показывают, что в долгосрочной перспективе коэффициент Танимото оказывается более точным, чем косинусный. Однако на практике во многих случаях они вполне взаимозаменяемы.

ТИПИЧНЫЕ ПАРАМЕТРЫ СРАВНЕНИЯ ЕДИНИЧНЫХ ВЕКТОРОВ

- `u1 @ u2` — косинус угла между единичными векторами `u1` и `u2`.
- `(u1 @ u2) / (2 - u1 @ u2)` — коэффициент Танимото между единичными векторами `u1` и `u2`.
- `(2 - 2 * u1 @ u2) ** 0.5` — евклидово расстояние между единичными расстояниями `u1` и `u2`.

Нормализация векторов позволяет заменять одни параметры сравнения другими. Среди прочих преимуществ нормализации можно отметить следующие.

- *Длина текста больше не выступает как дифференцирующий сигнал* — можно сравнивать длинные и короткие тексты с похожим содержанием.
- *Вычисление коэффициента Танимото более эффективно* — требуется всего одна операция со скалярным произведением.
- *Более эффективно вычисление сходства между каждой парой векторов*, то есть сходства всех со всеми.

Последнее преимущество мы еще не рассматривали. Однако вскоре узнаем, что таблицу сходства текстов можно элегантно вычислить с помощью *матричного умножения*. В математике матричное умножение расширяет скалярное произведение из одномерных векторов до двухмерных массивов, что позволяет более эффективно вычислять сходство между двумя текстами.

13.3. МАТРИЧНОЕ УМНОЖЕНИЕ ДЛЯ ЭФФЕКТИВНОГО ВЫЧИСЛЕНИЯ СХОДСТВА

При анализе текстов про *seashell* мы сравнивали каждую их пару по отдельности. А что, если вместо этого показать все сходства пар в таблице? Строки и столбцы будут соответствовать отдельным текстам, а их элементы — коэффициентам Танимото. Эта таблица предоставит общую картину всех связей между текстами, и мы наконец-то узнаем, на что `text2` похож больше — на `text1` или `text3`.

Давайте сгенерируем таблицу нормализованных коэффициентов Танимото, используя процесс, показанный на рис. 13.12 (листинг 13.32). Начнем с нормализации векторов TF в ранее вычисленном списке `tf_vectors`. Затем переберем каждую пару векторов и вычислим их коэффициент Танимото. Полученные показатели сходства сохраним в двухмерном массиве `similarities`, где `similarities[i][j]`

равняется сходствам между *i*-м и *j*-м текстами. В завершение визуализируем массив `similarities`, используя тепловую карту (рис. 13.13).

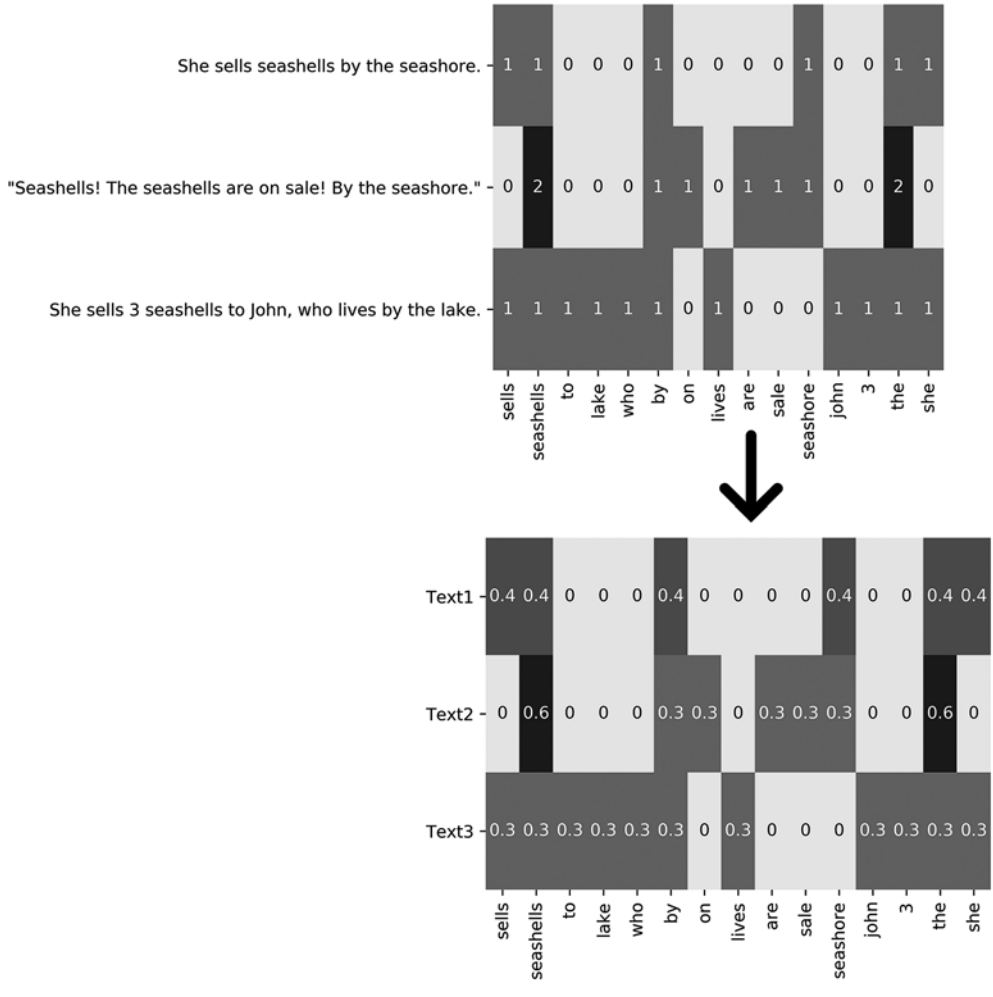


Рис. 13.12. Преобразование трех текстов в нормализованную матрицу. Исходные тексты показаны вверху слева. Их общий словарь состоит из 15 слов. Его мы используем для преобразования текстов в матрицу количества слов, показанную вверху справа. Три ее строки соответствуют трем текстам, а 15 столбцов отражают количество вхождений каждого слова в каждом тексте. Мы нормализуем эти количества делением каждой строки на ее абсолютную величину. Данная операция дает матрицу, приведенную внизу справа. Скалярное произведение между любыми двумя строками в этой нормализованной матрице равно косинусному коэффициенту между соответствующими текстами. Последующее выполнение cos/(2 – cos) преобразует косинусный коэффициент в коэффициент Танимото

Листинг 13.32. Вычисление таблицы нормализованных коэффициентов Танимото

```
num_texts = len(tf_vectors)
similarities = np.array([[0.0] * num_texts for _ in range(num_texts)])
unit_vectors = np.array([vector / norm(vector) for vector in tf_vectors])
for i, vector_a in enumerate(unit_vectors):
    for j, vector_b in enumerate(unit_vectors):
        similarities[i][j] = normalized_tanimoto(vector_a, vector_b)

labels = ['Text 1', 'Text 2', 'Text 3']
sns.heatmap(similarities, cmap='YlGnBu', annot=True,
            xticklabels=labels, yticklabels=labels)
plt.yticks(rotation=0)
plt.show()
```

Создает двухмерный массив с одними нулями. Этот массив можно создать более эффективно, выполнив np.zeros((num_texts, num_texts)), и заполнить его показателями сходства текстов

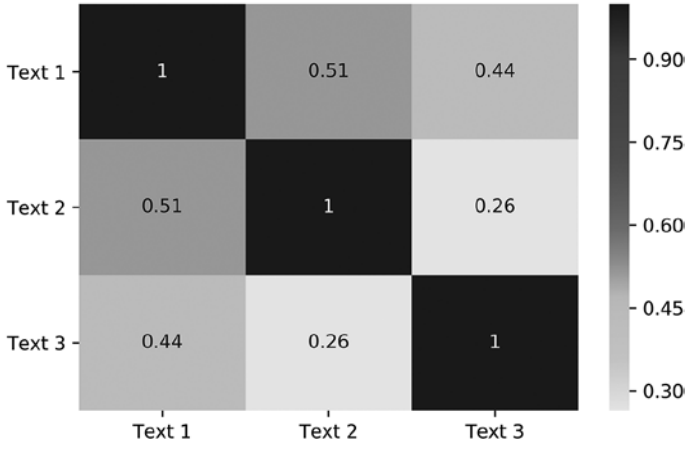


Рис. 13.13. Таблица нормализованных коэффициентов Танимото среди пар текстов. Ее диагональ представляет сходство каждого текста с самим собой. Поэтому неудивительно, что оно равно 1. Не обращая внимания на эту диагональ, мы видим, что тексты 1 и 2 обладают наивысшей степенью сходства. При этом сходство между текстами 2 и 3 оказывается минимальным

Таблица очень информативна. Глядя на нее, можно с ходу понять, какая пара текстов имеет наибольшее сходство. Однако вычисление этой таблицы опиралось на малоэффективный код, из которого вполне можно исключить следующие ненужные вычисления:

- создание пустого массива 3×3;
- перебор вложенного цикла `for` по всем парным комбинациям векторов;
- отдельное вычисление сходства каждой пары векторов.

Избавить код от этих операций можно с помощью матричного умножения. Однако сначала нужно познакомиться с матричными операциями.

13.3.1. Базовые матричные операции

Матричные операции лежат в основе многих областей науки о данных, включая NLP, сетевой анализ и машинное обучение. Поэтому знание основ работы с матрицами в карьере аналитика данных имеет особую важность. *Матрица* — это расширение одномерного вектора до двухмерного. Иными словами, матрица — это просто таблица чисел. Согласно этому определению `similarities` — это матрица, равно как и `unit_vectors`. Большинство численных таблиц, рассматриваемых в этой книге, также по сути являются матрицами.

ПРИМЕЧАНИЕ

Любая матрица является численной таблицей, но не всякая численная таблица — это матрица. То же касается всех матричных столбцов. То есть если таблица одновременно содержит пяти- и семиэлементный столбцы, то это не матрица.

Поскольку матрицы — это таблицы, их можно анализировать с помощью Pandas. И наоборот, численные таблицы можно обрабатывать с помощью двухмерных массивов NumPy. Оба эти варианта представления матрицы допустимы. В действительности датафреймы Pandas и массивы NumPy иногда можно использовать взаимозаменяемо, поскольку для них характерны определенные общие атрибуты. К примеру, `matrix.shape` возвращает количество строк и столбцов независимо от того, является `matrix` датафреймом или массивом. Аналогично `matrix.T` транспонирует строки и столбцы независимо от типа `matrix`. Убедимся в этом (листинг 13.33).

Листинг 13.33. Сравнение матричных атрибутов Pandas и NumPy

```
import pandas as pd

matrices = [unit_vectors, pd.DataFrame(unit_vectors)]
matrix_types = ['2D NumPy array', 'Pandas DataFrame']

for matrix_type, matrix in zip(matrix_types, matrices):
    row_count, column_count = matrix.shape
    print(f"Our {matrix_type} contains "
          f"{row_count} rows and {column_count} columns")
assert (column_count, row_count) == matrix.T.shape

Our 2D NumPy array contains 3 rows and 15 columns
Our Pandas DataFrame contains 3 rows and 15 columns
```

Транспонирование матрицы приводит к перестановке строк и столбцов местами

Табличные структуры Pandas и NumPy похожи. Тем не менее у хранения матриц в виде двухмерных массивов NumPy есть свои преимущества. Одно из них заключается в интеграции встроенных арифметических операторов Python, что позволяет выполнять базовые арифметические операции непосредственно над массивами NumPy.

Арифметические операции над матрицами в NumPy

Иногда в NLP требуется изменить матрицу, используя базовую арифметику. Предположим, мы хотим сравнить коллекцию документов на основе их содержания и заголовков. Допустим, что сходство заголовков вдвое важнее сходства содержания, поскольку документы с похожими заголовками, скорее всего, будут тематически связаны. В связи с этим мы решаем удвоить матрицу сходства заголовков, чтобы лучше взвесить ее относительно содержания.

ПРИМЕЧАНИЕ

Важность заголовков относительно тела особенно актуальна для новых статей. Две статьи с похожими заголовками наверняка будут относиться к одной новости, даже если их содержание предлагает разные взгляды на нее. Хорошей эвристикой для измерения сходства новостных заголовков будет вычисление 2 * title_similarity + body_similarity.

Удваивать значение матрицы очень легко в NumPy (листинг 13.34). К примеру, можно удвоить матрицу `similarities`, выполнив `2 * similarities`. Можно также прибавить `similarities` непосредственно к самой себе, выполнив `similarities + similarities`. Естественно, два полученных результата будут равны. При этом выполнение `similarities – similarities` вернет матрицу нулей. Более того, реализация `similarities – similarities – 1` приведет к вычитанию 1 из каждого нуля этой матрицы.

ПРИМЕЧАНИЕ

Мы вычитаем similarities + 1 из similarities просто для того, чтобы показать арифметическую гибкость NumPy. Обычно нет никакой объективной причины выполнять эту операцию, если только мы действительно не хотим получить матрицу отрицательных единиц.

Листинг 13.34. Сложение и вычитание массивов NumPy

```
double_similarities = 2 * similarities
np.array_equal(double_similarities, similarities + similarities)
zero_matrix = similarities - similarities
negative_1_matrix = similarities - similarities - 1

for i in range(similarities.shape[0]):
    for j in range(similarities.shape[1]):
        assert double_similarities[i][j] == 2 * similarities[i][j]
        assert zero_matrix[i][j] == 0
        assert negative_1_matrix[i][j] == -1
```

Аналогичным образом массивы NumPy можно умножать и делить. Выполнение `similarities / similarities` приведет к делению каждой матрицы сходств на саму себя и возвращению матрицы единиц. При этом выполнение `similarities * similarities` даст матрицу значений сходства, возведенных в квадрат (листинг 13.35).

Листинг 13.35. Умножение и деление массива NumPy

```
squared_similarities = similarities * similarities
assert np.array_equal(squared_similarities, similarities ** 2)
ones_matrix = similarities / similarities

for i in range(similarities.shape[0]):
    for j in range(similarities.shape[1]):
        assert squared_similarities[i][j] == similarities[i][j] ** 2
        assert ones_matrix[i][j] == 1
```

Матричная арифметика позволяет удобно переключаться между типами матриц сходства. К примеру, можно преобразовать нашу матрицу Танимото в матрицу косинусных коэффициентов, выполнив `2 * similarities / (1 + similarities)` (листинг 13.36). Таким образом, если нужно сравнить коэффициент Танимото с более популярным косинусным коэффициентом, можно вычислить вторую косинусную матрицу всего в одной строке кода.

Листинг 13.36. Преобразование между типами матриц сходства

```
cosine_similarities = 2 * similarities / (1 + similarities)
for i in range(similarities.shape[0]):
    for j in range(similarities.shape[1]):
        cosine_sim = unit_vectors[i] @ unit_vectors[j]
        assert round(cosine_similarities[i][j],
                     15) == round(cosine_sim, 15)
```

Подтверждает, что косинусный коэффициент равен фактическому скалярному произведению векторов

Округляет результаты для компенсации погрешностей из-за плавающей точки

Двухмерные массивы NumPy дают дополнительные преимущества в сравнении с Pandas. Обращаться к строкам и столбцам по индексу намного проще именно в NumPy.

Операции со строками и столбцами матриц в NumPy

Имея любой двухмерный массив `matrix`, можно обратиться к его строке по индексу `i`, выполнив `matrix[i]`. Аналогично можно обратиться к столбцу по индексу `j`, выполнив `matrix[:,j]`. Давайте используем индексацию для вывода первой строки и столбца `unit_vectors` и `similarities` (листинг 13.37).

Листинг 13.37. Обращение к строкам и столбцам матриц в NumPy

```
for name, matrix in [('Similarities', similarities),
                     ('Unit Vectors', unit_vectors)]:
    print(f"Accessing rows and columns in the {name} Matrix.")
    row, column = matrix[0], matrix[:,0]
    print(f"Row at index 0 is:\n{row}")
    print(f"\nColumn at index 0 is:\n{column}\n")

Accessing rows and columns in the Similarities Matrix.
```

```
Row at index 0 is:
[1.         0.51442439 0.44452044]

Column at index 0 is:
[1.         0.51442439 0.44452044]

Accessing rows and columns in the Unit Vectors Matrix.
Row at index 0 is:
[0.40824829 0.40824829 0.          0.40824829 0.         0.
0.          0.40824829 0.          0.         0.40824829 0.
0.          0.         0.40824829]

Column at index 0 is:
[0.40824829 0.         0.30151134]
```

Все выведенные строки и столбцы являются одномерными массивами NumPy. Имея два массива, можно вычислить их скалярное произведение, но только если их длина одинакова. В нашем выводе и `similarities[0].size`, и `unit_vectors[:,0].size` равны 3. Значит, можно получить скалярное произведение между первой строкой `similarities` и первым столбцом `unit_vectors` (листинг 13.38). Конкретно это скалярное произведение строки и столбца в нашем анализе текстов пользы не принесет, но оно демонстрирует возможность с легкостью управлять скалярным произведением строк и столбцов матрицы. Немного позже мы задействуем эту возможность для высокоэффективного вычисления сходства векторов текстов.

Листинг 13.38. Вычисление скалярного произведения строки и столбца

```
row = similarities[0]
column = unit_vectors[:,0]
dot_product = row @ column
print(f"The dot product between the row and column is: {dot_product:.4f}")
```

```
The dot product between the row and column is: 0.5423
```

По тому же принципу можно получить скалярное произведение любой строки в `similarities` и любого столбца в `unit_vectors`. Код листинга 13.39 выводит все возможные результаты скалярных произведений.

Листинг 13.39. Вычисление скалярных произведений всех строк и столбцов

```
num_rows = similarities.shape[0]
num_columns = unit_vectors.shape[1]
for i in range(num_rows):
    for j in range(num_columns):
        row = similarities[i]
        column = unit_vectors[:,j]
        dot_product = row @ column
        print(f"The dot product between row {i} column {j} is: "
              f"{dot_product:.4f}")
```

```
The dot product between row 0 column 0 is: 0.5423
The dot product between row 0 column 1 is: 0.8276
The dot product between row 0 column 2 is: 0.1340
The dot product between row 0 column 3 is: 0.6850
The dot product between row 0 column 4 is: 0.1427
The dot product between row 0 column 5 is: 0.1340
The dot product between row 0 column 6 is: 0.1427
The dot product between row 0 column 7 is: 0.5423
The dot product between row 0 column 8 is: 0.1340
The dot product between row 0 column 9 is: 0.1340
The dot product between row 0 column 10 is: 0.8276
The dot product between row 0 column 11 is: 0.1340
The dot product between row 0 column 12 is: 0.1340
The dot product between row 0 column 13 is: 0.1427
The dot product between row 0 column 14 is: 0.5509
The dot product between row 1 column 0 is: 0.2897
The dot product between row 1 column 1 is: 0.8444
The dot product between row 1 column 2 is: 0.0797
The dot product between row 1 column 3 is: 0.5671
The dot product between row 1 column 4 is: 0.2774
The dot product between row 1 column 5 is: 0.0797
The dot product between row 1 column 6 is: 0.2774
The dot product between row 1 column 7 is: 0.2897
The dot product between row 1 column 8 is: 0.0797
The dot product between row 1 column 9 is: 0.0797
The dot product between row 1 column 10 is: 0.8444
The dot product between row 1 column 11 is: 0.0797
The dot product between row 1 column 12 is: 0.0797
The dot product between row 1 column 13 is: 0.2774
The dot product between row 1 column 14 is: 0.4874
The dot product between row 2 column 0 is: 0.4830
The dot product between row 2 column 1 is: 0.6296
The dot product between row 2 column 2 is: 0.3015
The dot product between row 2 column 3 is: 0.5563
The dot product between row 2 column 4 is: 0.0733
The dot product between row 2 column 5 is: 0.3015
The dot product between row 2 column 6 is: 0.0733
The dot product between row 2 column 7 is: 0.4830
The dot product between row 2 column 8 is: 0.3015
The dot product between row 2 column 9 is: 0.3015
The dot product between row 2 column 10 is: 0.6296
The dot product between row 2 column 11 is: 0.3015
The dot product between row 2 column 12 is: 0.3015
The dot product between row 2 column 13 is: 0.0733
The dot product between row 2 column 14 is: 0.2548
```

Мы сгенерировали 45 скалярных произведений, по одному для каждой комбинации строки и столбца. Эти результаты можно в сжатом виде сохранить в таблице `dot_products`, где `dot_products[i][j]` равняется `similarities[i] @ unit_vectors[:,j]` (листинг 13.40). Естественно, по определению эта таблица является матрицей.

Листинг 13.40. Сохранение скалярных произведений всех со всеми в матрице

```
dot_products = np.zeros((num_rows, num_columns))
for i in range(num_rows):
    for j in range(num_columns):
        dot_products[i][j] = similarities[i] @ unit_vectors[:,j]

print(dot_products)
```

Возвращает пустой массив нулей

```
[[0.54227624 0.82762755 0.13402795 0.6849519  0.14267565 0.13402795
  0.14267565 0.54227624 0.13402795 0.13402795 0.82762755 0.13402795
  0.13402795 0.14267565 0.55092394]
 [0.28970812 0.84440831 0.07969524 0.56705821 0.2773501  0.07969524
  0.2773501  0.28970812 0.07969524 0.07969524 0.84440831 0.07969524
  0.07969524 0.2773501  0.48736297]
 [0.48298605 0.62960397 0.30151134 0.55629501 0.07330896 0.30151134
  0.07330896 0.48298605 0.30151134 0.30151134 0.62960397 0.30151134
  0.30151134 0.07330896 0.25478367]]
```

Только что выполненная операция называется *матричным умножением* и представляет собой расширение скалярного произведения до двух измерений. Имея две матрицы, `matrix_a` и `matrix_b`, можно найти их произведение `matrix_c`, где `matrix_c[i][j]` равняется `matrix_a[i] @ matrix_b[:,j]` (рис. 13.14). Матричное умножение — важнейшая операция во многих современных технологических решениях. Эта операция лежит в основе алгоритмов ранжирования таких мощных механизмов поиска, как Google, применяется при обучении беспилотных автомобилей, а также зачастую фигурирует в современной обработке естественного языка. Польза матричного умножения вскоре станет очевидной, но сначала нужно рассмотреть связанные с ним операции более подробно.

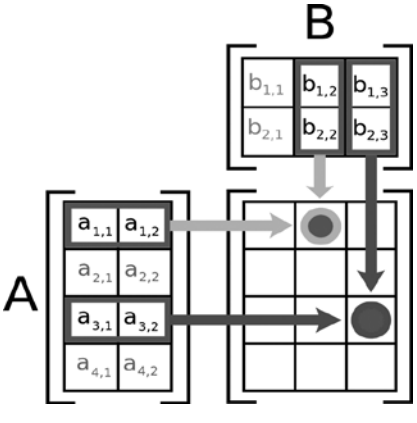


Рис. 13.14. Вычисление произведения матриц A и B. Эта операция на выходе даст новую матрицу. Каждый элемент в ряду *i* и столбце *j* будет равен скалярному произведению между строкой *i* в A и столбцом *j* в B. К примеру, элемент в первой строке второго столбца результата будет равен $a_{1,1} \times b_{1,2} + a_{1,2} \times b_{2,2}$, а элемент в третьей строке третьего столбца результата — $a_{3,1} \times b_{1,3} + a_{3,2} \times b_{2,3}$

Матричное произведение в NumPy

Простейший способ вычислить `matrix_c` — выполнить вложенные циклы `for` по `matrix_a` и `matrix_b`. Но такой подход неэффективен. Для удобства имеющийся в NumPy оператор произведения `@` можно применять как к двухмерным матрицам, так и к одномерным массивам. Если `matrix_a` и `matrix_b` являются массивами NumPy, тогда `matrix_c` будет равна `matrix-a @ matrix_b`. Таким образом, матричное произведение `similarities` и `unit_vectors` будет равно `similarities @ unit_vectors`. Проверим (листинг 13.41).

Листинг 13.41. Вычисление матричного произведения с помощью NumPy

```
matrix_product = similarities @ unit_vectors
assert np.allclose(matrix_product, dot_products)
```

Утверждает, что все элементы matrix_product практически идентичны всем элементам dot_products. В результатах присутствуют незначительные различия, вызванные погрешностями из-за плавающей точки

Что произойдет, если поменять эти матрицы местами и выполнить `unit_vectors @ similarities`? NumPy выдаст ошибку. Эта операция — скалярное перемножение векторов строк в `unit_vectors` и столбцов в `similarities`, но эти строки и столбцы имеют разную длину, поэтому данное вычисление невозможно (листинг 13.42; рис. 13.15).

Листинг 13.42. Вычисление ошибочного матричного произведения

```
try:
    matrix_product = unit_vectors @ similarities
except:
    print("We can't compute the matrix product")

We can't compute the matrix product
```

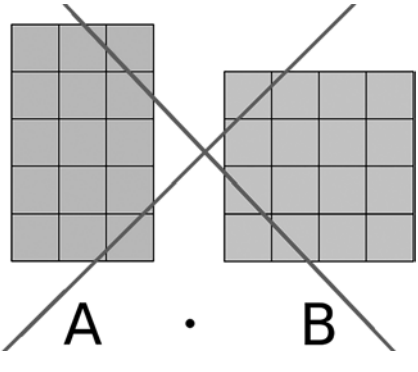


Рис. 13.15. Ошибочная попытка перемножить матрицы A и B. В матрице A по три столбца в каждой строке, а в матрице B по четыре строки в каждом столбце. Мы не можем получить скалярное произведение трехэлементной строки и четырехэлементного столбца, поэтому A @ B вызывает ошибку

Матричное произведение зависит от порядка перемножаемых матриц. Результат `matrix_a @ matrix_b` не обязательно совпадет с результатом `matrix_b @ matrix_a`. На словах провести различие между `matrix_a @ matrix_b` и `matrix_b @ matrix_a` можно так:

- `matrix_a @ matrix_b` является произведением `matrix_a` и `matrix_b`;
- `matrix_b @ matrix_a` является произведением `matrix_b` и `matrix_a`.

В математике слова «произведение» и «умножение» зачастую являются взаимозаменяемыми. Поэтому вычисление матричного произведения обычно называется матричным умножением. Это выражение настолько широко используется, что NumPy содержит специальную функцию `np.matmul`. Результат `np.matmul (matrix_a, matrix_b)` будет идентичен результату `matrix_a @ matrix_b` (листинг 13.43).

Листинг 13.43. Выполнение матричного умножения с помощью matmul

```
matrix_product = np.matmul(similarities, unit_vectors)
assert np.array_equal(matrix_product,
                      similarities @ unit_vectors)
```

ТИПИЧНЫЕ МАТРИЧНЫЕ ОПЕРАЦИИ NUMPY

- `matrix.shape` — возвращает кортеж, отражающий количество строк и столбцов в матрице.
- `matrix.T` — возвращает транспонированную матрицу, в которой строки и столбцы поменялись местами.
- `matrix[i]` — возвращает *i*-ю строку матрицы.
- `matrix[:,j]` — возвращает *j*-й столбец матрицы.
- `k * matrix` — умножает каждый элемент матрицы на константу k.
- `matrix + k` — прибавляет константу k к каждому элементу матрицы.
- `matrix_a + matrix_b` — прибавляет каждый элемент `matrix_a` к `matrix_b`. Равнозначно выполнению `matrix_c[i][j] = matrix_a[i][j] + matrix_b[i][j]` для каждого возможного `i` и `j`.
- `matrix_a * matrix_b` — умножает каждый элемент `matrix_a` на элемент `matrix_b`. Равноценно выполнению `matrix_c[i][j] = matrix_a[i][j] * matrix_b[i][j]` для каждого возможного `i` и `j`.
- `matrix_a @ matrix_b` — возвращает матричное произведение `matrix_a` и `matrix_b`. Равнозначно выполнению `matrix_c[i][j] = matrix_a[i] @ matrix_b[;,j]` для каждого `i` и `j`.
- `np.matmult(matrix_a, matrix_b)` — возвращает матричное произведение `matrix_a` и `matrix_b`, не используя оператор `@`.

NumPy позволяет выполнять матричное умножение, не задействуя вложенные циклы `for`. И это не только внешнее улучшение. Стандартные циклы `for` в Python предназначены для выполнения обобщенных списков данных и не оптимизированы для обработки чисел. А вот в NumPy итерирование массивов продумано грамотно, в связи с чем выполнение матричного умножения заметно эффективнее работает именно в этой библиотеке.

Сравним скорость выполнения матричного умножения в NumPy и обычном Python (рис. 13.16). Код листинга 13.44 строит график скорости вычисления произведения матриц переменного размера с использованием NumPy и Python циклов `for`. Время выполнения оценивается встроенным в Python модулем `time`.

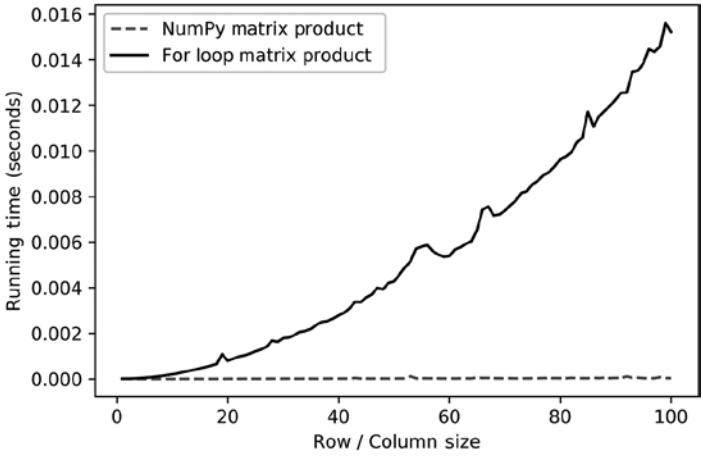


Рис. 13.16. График соотношения размера матрицы и времени выполнения умножения в NumPy и обычном Python. NumPy оказывается значительно быстрее

ПРИМЕЧАНИЕ

Время выполнения будет изменяться в зависимости от состояния выполняющей код машины.

Листинг 13.44. Сравнение времени выполнения матричного умножения

```
import time

numpy_run_times = []
for_loop_run_times = []

matrix_sizes = range(1, 101)
for size in matrix_sizes:
    matrix = np.ones((size, size))

    start_time = time.time()
    matrix @ matrix
```

Создает матрицу единиц, размер которой от 1 до 100

Возвращает текущее время выполнения в секундах

```
    numpy_run_times.append(time.time() - start_time)

    start_time = time.time()
    for i in range(size):
        for j in range(size):
            matrix[i] @ matrix[:,j]

    for_loop_run_times.append(time.time() - start_time)

plt.plot(matrix_sizes, numpy_run_times,
         label='NumPy Matrix Product', linestyle='--')
plt.plot(matrix_sizes, for_loop_run_times,
         label='For-Loop Matrix Product', color='k')
plt.xlabel('Row / Column Size')
plt.ylabel('Running Time (Seconds)')
plt.legend()
plt.show()
```

Сохраняет показатель скорости выполнения матричного умножения в NumPy

Сохраняет показатель скорости выполнения матричного умножения Python циклами for

Когда дело доходит до матричного умножения, NumPy значительно превосходит стандартный Python. Код получения матричного произведения в NumPy более эффективен в работе и проще в написании. Теперь мы используем NumPy для вычисления сходства между всеми текстами с максимальной эффективностью.

13.3.2. Вычисление сходства матриц

Мы уже вычисляли сходство наших текстов, перебирая матрицу `unit_vectors`. В ней содержатся нормализованные векторы TF для трех выражений про морские раковины (seashells). А что произойдет, если умножить `unit_vectors` на `unit_vectors.T`? Поскольку `unit_vectors.T` — это транспонированная версия `unit_vectors`, каждый столбец `i` в нем соответствует строке `i` в `unit_vectors`. Получение скалярного произведения `unit_vectors[i]` и `unit_vectors.T[:,i]` даст нам косинусный коэффициент между единичным вектором и им самим, что показано на рис. 13.17. Естественно, этот коэффициент будет равен `1`. По той же логике `unit_vectors[i] @ unit_vectors[j].T` будет равно косинусному коэффициенту между *i*-м и *j*-м векторами. Следовательно, операция `unit_vectors @ unit_vectors.T` вернет матрицу косинусных коэффициентов между всеми векторами. Она должна равняться ранее вычисленному массиву `cosine_similarities`. Проверим это (листинг 13.45).

Листинг 13.45. Получение косинусов из матричного произведения

```
cosine_matrix = unit_vectors @ unit_vectors.T
assert np.allclose(cosine_matrix, cosine_similarities)
```

Каждый элемент в `cosine_matrix` равен косинусу угла между двумя векторизованными текстами. Этот косинус можно преобразовать в коэффициент Танимото, который обычно отражает совпадение и различие слов в текстах. С помощью арифметики NumPy можно преобразовать `cosine_matrix` в матрицу коэффициентов Танимото, выполнив `cosine_matrix / (2 – cosine_matrix)` (листинг 13.46).

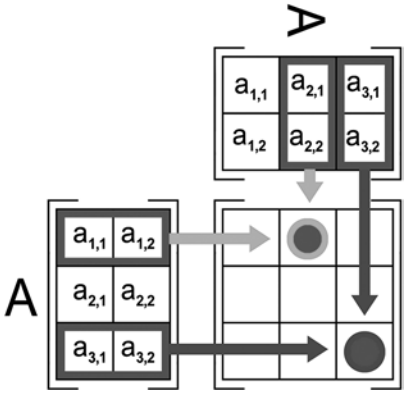


Рис. 13.17. Вычисление скалярного произведения A и транспонированной версии A. В результате этой операции получится новая матрица, каждый элемент *i*-й строки и *j*-го столбца которой будет равен скалярному произведению *i*-й строки и *j*-го столбца A. Таким образом, элемент в третьей строке третьего столбца итоговой матрицы равен скалярному произведению A[2] и ее самой. Если матрица A нормализована, то это скалярное произведение будет равно 1

Листинг 13.46. Преобразование косинусов в матрицу Танимото

```
tanimoto_matrix = cosine_matrix / (2 - cosine_matrix)
assert np.allclose(tanimoto_matrix, similarities)
```

Все коэффициенты Танимото вычислены всего в двух строках кода. Мы также можем получить эти коэффициенты, передав `unit_vectors` и `unit_vectors.T` непосредственно в функцию `normalized_tanimoto` (листинг 13.47). Напомню, что эта функция:

- получает два массива NumPy, размерность которых не ограничена;
- применяет оператор `@` к массивам NumPy. Если же массивы являются матрицами, эта операция возвращает матричное произведение;
- использует арифметику для изменения произведения. Арифметические операции можно применять как к числам, так и к матрицам.

Поэтому неудивительно, что `normalized_tanimoto(unit_vectors, unit_vectors.T)` возвращает результат, равный `tanimoto_matrix`.

Листинг 13.47. Передача матриц в normalized_tanimoto

```
output = normalized_tanimoto(unit_vectors, unit_vectors.T)
assert np.array_equal(output, tanimoto_matrix)
```

Имея матрицу нормализованных векторов TF, можно вычислить все их коэффициенты в одной строке кода.

ТИПИЧНЫЕ ОПЕРАЦИИ СРАВНЕНИЯ НОРМАЛИЗОВАННЫХ МАТРИЦ

- `orm_matrix @ norm_matrix.T` — возвращает матрицу косинусных коэффициентов между всеми векторами.
- `norm_matrix @ norm_matrix.T / ( 2 - norm_matrix @ norm_matrix.T)` — возвращает матрицу коэффициентов Танимото между всеми векторами.

13.4. ВЫЧИСЛИТЕЛЬНЫЕ ОГРАНИЧЕНИЯ МАТРИЧНОГО УМНОЖЕНИЯ

Скорость матричного умножения определяется размером матрицы. NumPy может обеспечивать оптимизацию быстродействия, но даже у этой библиотеки есть пределы возможностей. Они становятся очевидными при вычислении матричных произведений реальных текстов. Проблемы возникают из-за количества столбцов в матрице, которое определяется размером словаря. Общее число слов в словаре может выходить из-под контроля, когда мы начинаем сравнивать огромные тексты.

Представьте анализ романов. В среднем романе содержится от 5 000 до 10 000 уникальных слов. К примеру, в *«Хоббите»* использовано 6 175 уникальных слов, а в *«Повести о двух городах»* — 9 699. Некоторые слова этих романов совпадают, другие — нет. Совместно они формируют словарь, включающий 12 138 слов. В эту комбинацию можно добавить и третий роман. Если это будут *«Приключения Тома Сойера»*, то размер словаря расширится до 13 935 слов. Такими темпами добавление еще 27 романов приведет к разрастанию словаря примерно до 50 000 слов.

Предположим, что этим 30 романам требуется общий словарь, содержащий 50 000 слов. Кроме того, предположим, что мы вычисляем сходство между всеми этими 30 книгами. Сколько времени потребуется для подобной операции? Сейчас узнаем (листинг 13.48). Мы создадим `book_matrix` размером 30 книг на 50 000 слов. Все строки этой матрицы будут нормализованы. После этого мы измерим время выполнения `normalized_tanimoto(book_matrix, book_ matrix.T)`.

ПРИМЕЧАНИЕ

Цель нашего эксперимента — проверить влияние количества столбцов в матрице на продолжительность вычислений. В данном случае конкретное содержание матрицы значения не имеет, поэтому мы специально упростим ситуацию, установив количество вхождений каждого слова на 1. В результате нормализованные значения каждой строки будут равны 1/50 000. В реальных условиях так бы не получилось. Кроме того, заметьте, что можно оптимизировать время выполнения, если отследить все нулевые элементы матрицы. Здесь же мы не рассматриваем влияние нулевых значений на скорость матричного умножения.

Листинг 13.48. Оценка времени сравнения 30 романов между собой

```
vocabulary_size = 50000
normalized_vector = [1 / vocabulary_size] * vocabulary_size book_count = 30

def measure_run_time(book_count):
    book_matrix = np.array([normalized_vector] * book_count)
    start_time = time.time()
    normalized_tanimoto(book_matrix, book_matrix.T)
    return time.time() - start_time

run_time = measure_run_time(book_count)
print(f"It took {run_time:.4f} seconds to compute the similarities across a "
      f"{book_count}-book by {vocabulary_size}-word matrix")
```

Эта функция вычисляет время выполнения для матрицы размером book_count на 50 000. Она также повторно используется в следующих двух листингах

```
It took 0.0051 seconds to compute the similarities across a 30-book by 50000-
word matrix
```

На вычисление матрицы сходства ушло примерно 5 мс. Это вполне адекватное время выполнения. Останется ли оно таким же адекватным, если число анализируемых книг будет расти? Проверим. Мы построим график времени выполнения для разного количества книг, начиная с 30 и до примерно 1000 (листинг 13.49; рис. 13.18). В целях согласованности размер словаря мы оставим равным 50 000.

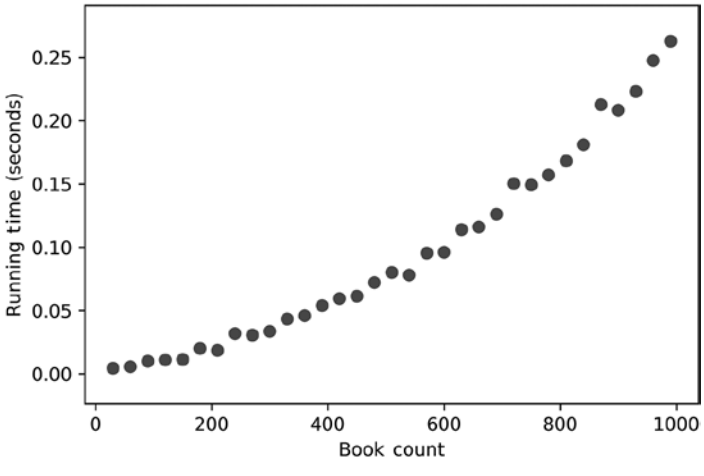


Рис. 13.18. График количества книг относительно продолжительности сравнения текстов. Время выполнения растет квадратично

Листинг 13.49. Построение графика количества книг относительно продолжительности сравнения

```
book_counts = range(30, 1000, 30)
run_times = [measure_run_time(book_count)
             for book_count in book_counts]
```

Мы не перебираем каждое возможное количество книг, иначе общее время выполнения получится огромным

```
plt.scatter(book_counts, run_times)
plt.xlabel('Book Count')
plt.ylabel('Running Time (Seconds)')
plt.show()
```

Генерирует не кривую, а точечный график. На рис. 13.18 раздельные точки соответствуют непрерывной параболической кривой

Время вычисления сходства с увеличением количества книг растет квадратично. При 1000 книг оно возрастает примерно до 0,27 с. Такая задержка еще вполне терпима. Однако, если продолжить увеличивать число книг, она выйдет за допустимые пределы. Это можно продемонстрировать с помощью простой математики (листинг 13.50). Построенная кривая принимает параболическую форму, определяемую формулой `y = n * (x ** 2)`. Когда `x` приближается к 1000, `y` равен примерно `0.27`. Таким образом, можно смоделировать время выполнения, используя уравнение `y = (0.27 / (1000 ** 2)) * (x ** 2)`. Убедимся в этом, построив график его результатов вместе с ранее вычисленными размерами (рис. 13.19). Два полученных графика должны преимущественно накладываться друг на друга.

Листинг 13.50. Моделирование времени выполнения с помощью квадратичной кривой

```
def y(x): return (0.27 / (1000 ** 2)) * (x ** 2)
plt.scatter(book_counts, run_times)
plt.plot(book_counts, y(np.array(book_counts)), c='k')
plt.xlabel('Book Count')
plt.ylabel('Running Time (Seconds)')
plt.show()
```

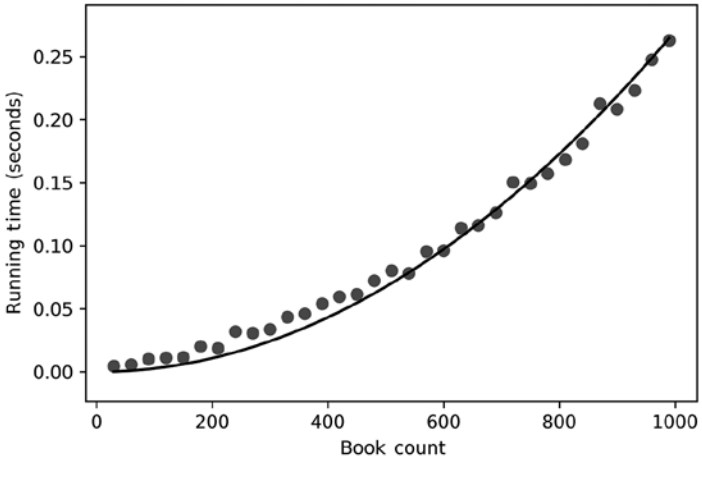


Рис. 13.19. Квадратичная кривая, построенная совместно с графиком времени выполнения. Форма кривой совпадает с точечным графиком времени выполнения

График нашего уравнения пересекается с графиком времени выполнения, значит, его можно использовать для прогнозирования скорости сравнения бóльшего числа книг. Посмотрим, сколько времени уйдет на измерение сходства между 300 000 книг (листинг 13.51).

ПРИМЕЧАНИЕ

Триста тысяч может показаться необычайно большим числом. Однако оно отражает от 200 000 до 300 000 ежегодно публикуемых англоязычных романов. Если мы хотим сравнить все вышедшие за год романы, то потребуется перемножить матрицы, содержащие более 200 000 строк.

Листинг 13.51. Прогнозирование времени выполнения для 300 000 книг

```
book_count = 300000
run_time = y(book_count) / 3600
print(f"It will take {run_time} hours to compute all-by-all similarities "
      f"from a {book_count}-book by {vocabulary_size}-word matrix")

It will take 6.75 hours to compute all-by-all similarities from a 300000-book
    by 50000-word matrix
```

Делит на 3600, чтобы преобразовать секунды в часы

На сравнение 300 000 книг уйдет примерно семь часов. Такая задержка явно неприемлема, особенно в промышленных NLP-системах, которые предназначены для обработки миллионов текстов в считаные секунды. Нам нужно сократить время выполнения, и одним из решений является уменьшение размера матрицы.

Наша матрица слишком велика, отчасти из-за количества столбцов. Каждая строка содержит 50 000 столбцов, соответствующих 50 000 слов. Однако в реальных условиях не все эти слова распределяются поровну. Притом что часть из них в текстах повторяется, другие могут встречаться единожды. Возьмем, к примеру, длинный роман *«Моби Дик»*: в нем 44 % слов употребляются всего раз. Некоторые из этих слов в других романах встречаются редко. Удалив их, мы уменьшим количество столбцов.

С противоположного края спектра находятся типичные слова, используемые в каждом романе. Слова вроде *the* не свидетельствуют о различии текстов, и их удаление также позволит сократить количество столбцов.

Таким образом, можно систематически уменьшать размер каждой строки матрицы с 50 000 до более адекватного значения. В следующей главе мы познакомимся с серией приемов сокращения размерности, позволяющих уменьшать любую входную матрицу. Уменьшение размерности матриц текстов существенно сокращает время выполнения типичных вычислений в NLP.

РЕЗЮМЕ

- Можно сравнивать тексты с помощью *коэффициента Жаккара*. Этот показатель сходства соответствует доле от общего числа уникальных слов, присутствующей в обоих текстах.
- Коэффициент Жаккара можно вычислить преобразованием текстов в двоичные векторы из 1 и 0. Получение *скалярного произведения* двух двоичных векторов текстов даст количество общих слов, встречающихся в обоих текстах. При этом скалярное произведение вектора текста с самим собой даст общее число слов в этом тексте. Этих значений будет достаточно для вычисления коэффициента Жаккара.
- *Коэффициент Танимото* расширяет коэффициент Жаккара до недвоичных векторов. Это позволяет сравнивать векторы количеств слов, называемые *векторами частоты встречаемости термина*, или *векторами TF*.
- Сходство векторов TF сильно зависит от размера текста. Эту зависимость можно устранить с помощью *нормализации*. Для нормализации вектора нужно вычислить *абсолютную величину*, являющуюся расстоянием этого вектора до начала координат, а затем разделить вектор на эту абсолютную величину, что даст нормализованный единичный вектор.
- Абсолютная величина *единичного вектора* равна 1. Кроме того, коэффициент Танимото отчасти зависит от абсолютной величины вектора, в связи с чем можно упростить функцию сходства, если она выполняется только для единичных векторов. Вдобавок сходство единичных векторов можно преобразовывать в другие распространенные параметры, такие как *косинусный коэффициент* и расстояние.
- Можно эффективно вычислить сходства всех со всеми, используя *матричное умножение*. *Матрица* — это просто двухмерная таблица чисел. Две матрицы можно перемножить, получив скалярное произведение каждой их строки и каждого столбца. Если умножить нормализованную матрицу на ее транспонированную версию, мы получим матрицу косинусных коэффициентов между каждой ее строкой и столбцом. Используя арифметику NumPy, можно преобразовать эти косинусные коэффициенты в коэффициенты Танимото.
- Матричное умножение выполняется в NumPy намного быстрее, чем в стандартном Python. Однако даже возможности NumPy имеют предел. Когда матрица становится чересчур большой, необходимо найти способ уменьшить ее.

11 Уменьшение размерности матричных данных

В этой главе

- ✓ Упрощение матриц с помощью геометрического вращения.
- ✓ Знакомство с анализом главных компонент.
- ✓ Продвинутые матричные операции для уменьшения размеров матрицы.
- ✓ Знакомство с сингулярной декомпозицией.
- ✓ Уменьшение размерности с помощью scikit-learn.

Уменьшение размерности — это серия приемов для уменьшения данных с сохранением информативности их содержания. Они задействуются во множестве наших повседневных цифровых занятий. Предположим, вы только что вернулись из отпуска в Белизе. У вас в телефоне есть десять фотографий, которые вы хотите отправить другу. К сожалению, размер этих фото довольно большой, а доступное беспроводное соединение очень медленное. Каждая фотография имеет размер 1200×1200 пикселов. Она занимает 5,5 Мбайт памяти и передается за 15 с. Получается, что передача всех десяти фотографий займет 2,5 мин. Но, к вашей удаче, мессенджер предоставляет альтернативный вариант, а именно возможность уменьшить каждое фото с 1200×1200 до 600×480 пикселов. Это сократит размеры каждого снимка в шесть раз. Уменьшая разрешение, вы жертвуете некоторыми деталями, но при этом изображения сохранят большую часть информации: пышные

джунгли, голубое море и сверкающий песок останутся полностью видимыми. Значит, компромисс того стоит. Уменьшение размерности в шесть раз увеличит скорость передачи в те же шесть раз, то есть теперь для передачи фотографий другу потребуется всего 25 с.

Но у сокращения размеров, естественно, есть и другие преимущества. Возьмем, к примеру, картирование, которое можно рассматривать как задачу по уменьшению размеров. Наша Земля — трехмерная сфера, которую можно точно смоделировать в виде глобуса. Его можно превратить в карту, спроецировав трехмерную форму на двухмерный лист бумаги. Бумажную карту проще переносить с места на место — в отличие от глобуса ее можно свернуть и положить в карман. Но двухмерная карта дает и другие преимущества. Предположим, что нас попросили найти все страны, которые граничат не менее чем с десятью другими территориями. Найти на карте такие богатые на соседей регионы легко — достаточно обратить внимание на густые кластеры городов. Если же использовать глобус, выполнить эту задачу будет уже не так просто — нам потребуется вращать и рассматривать его с разных сторон, так как невозможно видеть все страны сразу. В некотором смысле изгиб поверхности глобуса выступает в качестве уровня шума, мешающего решить поставленную задачу. Устранение кривой уменьшит сложность, но за это придется заплатить. В результате с карты, по сути, исчезнет материк Антарктида. Конечно же, никаких стран на нем нет, поэтому с точки зрения нашей задачи такой компромисс оправдан.

Аналогия с картой показывает следующие преимущества данных с уменьшенной размерностью:

- более компактные данные проще передавать и хранить;
- алгоритмические задачи выполняются быстрее, когда данных меньше;
- некоторые сложные задачи наподобие обнаружения кластеров можно упростить, удалив ненужную информацию.

Последние два пункта тесно связаны с текущим практическим заданием. Нам нужно кластеризовать тысячи документов по темам. Такая кластеризация подразумевает вычисление матрицы сходства всех документов между собой. Как говорилось в предыдущей главе, вычисление может оказаться медленным. Уменьшение размерности позволит ускорить этот процесс за счет сокращения количества столбцов данных в матрице. В качестве дополнительного бонуса текстовые данные уменьшенной размерности дают более качественные тематические кластеры.

Далее мы углубимся в связи между уменьшением размерности и кластеризацией данных, начав с простой задачи по кластеризации двухмерных данных в одном измерении.

14.1. КЛАСТЕРИЗАЦИЯ ДВУХМЕРНЫХ ДАННЫХ В ОДНОМ ИЗМЕРЕНИИ

Уменьшение размерности имеет много областей применения, включая лучше интерпретируемую кластеризацию. Представьте сценарий, в котором мы управляем онлайн-магазином одежды. Когда клиенты посещают наш сайт, их просят сообщить свой рост и вес. Эти характеристики вносятся в клиентскую базу данных. Для их сохранения требуются два столбца, поэтому такие данные являются двухмерными. Эти характеристики используются для того, чтобы рекомендовать клиентам подходящую по размеру одежду из доступного ассортимента. Наш ассортимент представлен в трех размерах: малом, среднем и большом. Располагая характеристиками 180 клиентов, нам нужно проделать следующее:

- сгруппировать клиентов в три отдельных кластера по размеру;
- создать интерпретируемую модель для определения категории размера одежды каждого нового клиента на основе вычисленных кластеров;
- сделать кластеризацию достаточно простой, чтобы ее могли понять наши не обладающие техническими знаниями инвесторы.

Больше всего спектр возможных решений ограничивает третий пункт. Кластеризация не может опираться на технические параметры, такие как центр масс или расстояние до среднего. В идеале нам нужно иметь возможность объяснить предлагаемую модель в одном рисунке. Достичь такого уровня простоты можно с помощью уменьшения размерности. Однако сначала нужно симулировать двухмерные данные характеристик для 180 клиентов. Начнем с их роста, который составляет от 60 до 78 дюймов. Характеристики роста мы симулируем вызовом `np.arange(60, 78, 0.1)`, которая вернет массив ростов от 60 до 78 дюймов, где каждый последующий рост увеличивается на 0,1 дюйма (листинг 14.1).

Листинг 14.1. Симуляция диапазона ростов

```
import numpy as np
heights = np.arange(60, 78, 0.1)
```

При этом симулируемые веса в значительной степени зависят от роста. Высокий человек скорее будет иметь бóльший вес, чем меньший. Было доказано, что в среднем вес человека в фунтах составляет `4 * height – 130`. Естественно, вес каждого отдельного человека колеблется вокруг этого среднего значения. Мы смоделируем эти случайные колебания при помощи нормального распределения со стандартным отклонением 10 фунтов. В результате, имея переменную `height`, можно смоделировать вес как `4 * height – 130 + np.random.normal(scale=10)`. Далее, используя эту связь, мы симулируем веса для каждого из 180 ростов (листинг 14.2).

Листинг 14.2. Симулирование весов на основе роста

```
np.random.seed(0)
random_fluctuations = np.random.normal(scale=10, size=heights.size)
weights = 4 * heights - 130 + random_fluctuations
```

Параметр scale устанавливает стандартное отклонение

Рост и вес можно рассматривать как двухмерные координаты в матрице `measurements`. Сохраним эти измеренные координаты и построим их график (листинг 14.3; рис. 14.1).

Листинг 14.3. Построение графика 2D-характеристик

```
import matplotlib.pyplot as plt
measurements = np.array([heights, weights])
plt.scatter(measurements[0], measurements[1])
plt.xlabel('Height (in)')
plt.ylabel('Weight (lb)')
plt.show()
```

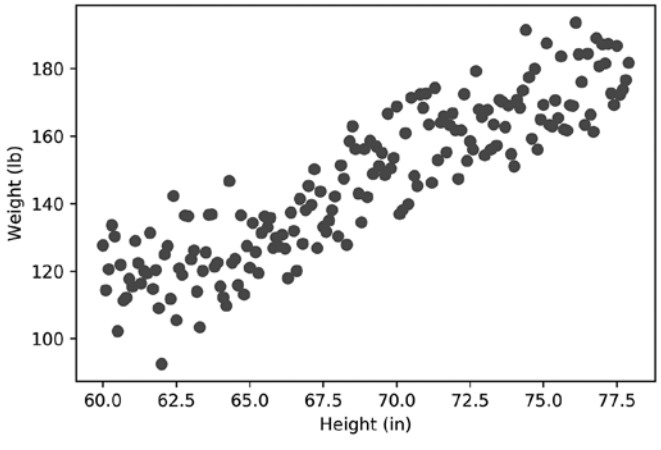


Рис. 14.1. График соотношения веса и роста. Здесь очевидна их линейная связь

На графике явно просматривается линейная связь между ростом и весом. Кроме того, как и ожидалось, оси роста и веса масштабируются по-разному. Напомню, что Matplotlib подстраивает 2D-оси, делая итоговый график более эстетичным. Обычно это хорошо, но вскоре мы будем вращать график для упрощения данных. Вращение нарушит масштабирование осей, усложнив сравнение повернутых данных с оригинальным графиком. Поэтому нам нужно уравнять оси для получения согласованного визуального вывода. Сделаем это вызовом `plt.axis('equal')`, после чего сгенерируем график еще раз (листинг 14.4; рис. 14.2).

Листинг 14.4. Построение графика двухмерных характеристик с использованием одинаково масштабированных осей

```
plt.scatter(measurements[0], measurements[1])
plt.xlabel('Height (in)')
plt.ylabel('Weight (lb)')
plt.axis('equal')
plt.show()
```

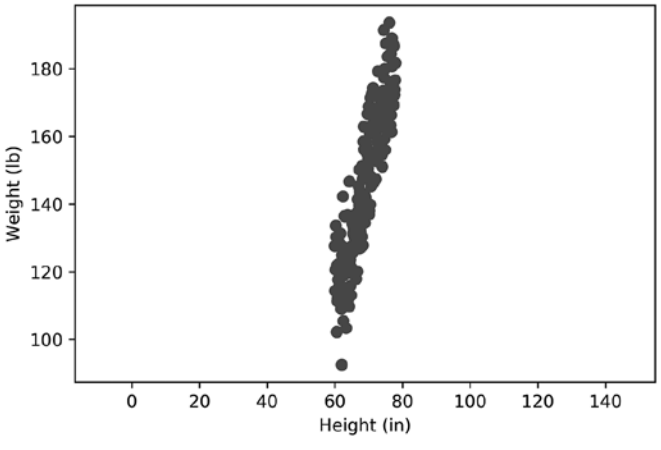


Рис. 14.2. График соотношения роста и веса с одинаковым масштабированием осей

Наш график формирует тонкую фигуру в форме сигары. Ее можно кластеризовать по размеру, если разрезать на три равные части. Один из способов получить кластеры — это метод *K*-средних. Естественно, интерпретация результата потребует понимания принципа работы метода *K*-средних. Менее техническое решение — просто положить сигару на бок. Если сигарообразный график будет расположен горизонтально, то мы разделим его на три части с помощью двух вертикальных срезов, как показано на рис. 14.3. Первый срез отделит 60 точек данных слева, а второй — 60 точек справа. Эти операции сегментируют наших клиентов таким способом, который будет несложно объяснить даже технически не подкованному человеку.

ПРИМЕЧАНИЕ

В этом упражнении мы предположим, что малый, средний и большой размеры распределены равномерно. В реальной индустрии одежды так будет вряд ли.

Если развернуть наши данные вдоль оси *X*, горизонтальных значений x должно быть достаточно для проведения различий между точками. Так что можно кластеризовать данные, не опираясь на вертикальные значения y. По сути, мы сможем удалить значения y с минимальной потерей информации. Это уменьшит размерность наших данных с 2D до 1D (рис. 14.4).

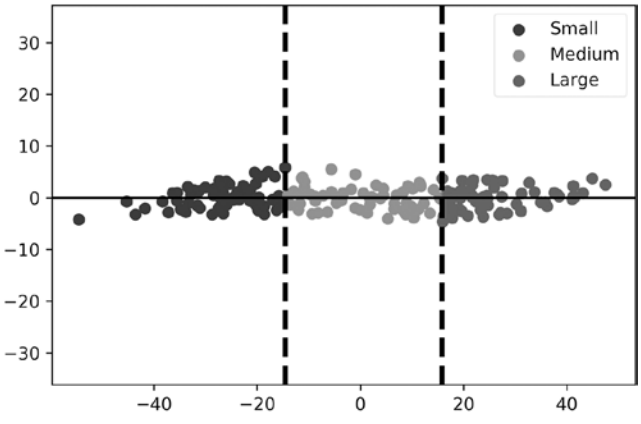


Рис. 14.3. Мы повернули линейные характеристики горизонтально, чтобы они лежали преимущественно вдоль оси X. Двух вертикальных срезов достаточно, чтобы разделить данные на три равных кластера: малый, средний и большой размеры. Здесь оси X достаточно для того, чтобы провести различие между характеристиками. Поэтому можно убрать ось Y с минимальной потерей информации. Заметьте, что этот рисунок был получен с помощью кода листинга 14.15

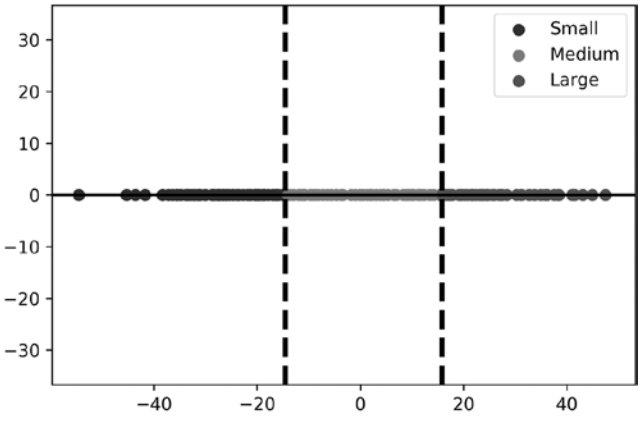


Рис. 14.4. Линейные характеристики, сведенные к одному измерению за счет поворота данных в горизонтальную плоскость. Точки данных развернуты в сторону оси X, а их y-координаты удалены. Тем не менее значений X достаточно для проведения различия между этими точками. Так что одномерный вывод по-прежнему позволяет разделить данные на три равных кластера

Теперь попробуем кластеризовать двухмерные данные, повернув их на бок. Поворот в горизонтальное положение позволит нам и кластеризовать данные, и уменьшить их размерность.

14.1.1. Уменьшение размерности с помощью вращения

Чтобы повернуть данные набок, нужно сделать два шага.

1. Сдвинуть все точки данных так, чтобы они оказались центрированы вокруг начала координат (0, 0). В результате будет проще повернуть график в сторону оси *X*.
2. Поворачивать график, пока общее расстояние точек данных до оси *X* не окажется минимальным.

Центрировать данные вокруг начала координат несложно. Центральная точка любого набора данных равна его среднему. Таким образом, нам нужно скорректировать координаты так, чтобы их среднее *x* и среднее *y* равнялись нулю. Это можно сделать вычитанием текущего среднего из каждой координаты. Иными словами, вычитание среднего роста из `heights` и среднего веса из `weights` даст набор данных, центрированный вокруг (0, 0).

Сместим координаты роста и веса, сохранив эти изменения в массиве `centered_data`. После этого построим график смещенных координат с целью убедиться, что они центрированы вокруг начала координат (листинг 14.5; рис. 14.5).

Листинг 14.5. Центрирование значений характеристик вокруг начала координат

```
centered_data = np.array([heights - heights.mean(),
                          weights - weights.mean()])
plt.scatter(centered_data[0], centered_data[1])
plt.axhline(0, c='black')
plt.axvline(0, c='black')
plt.xlabel('Centralized Height (in)')
plt.ylabel('Centralized Weight (lb)')
plt.axis('equal')
plt.show()
```

Визуализирует оси X и Y, чтобы отметить расположение начала координат

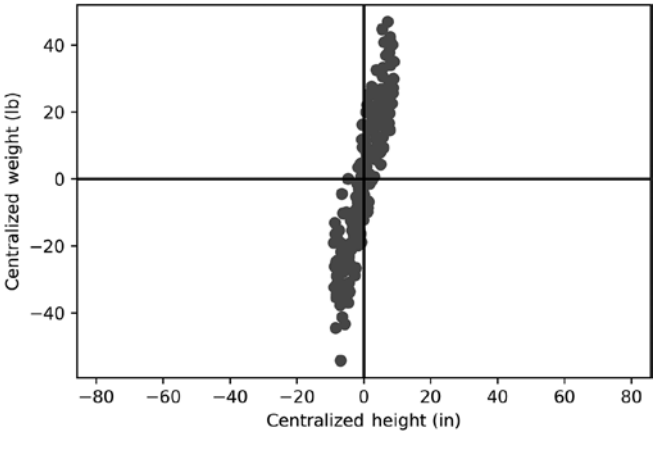


Рис. 14.5. График соотношения роста и веса, центрированный в начале координат. Такие центрированные данные можно вращать, как пропеллер

Теперь данные идеально центрированы вокруг начала координат. Однако ориентированы они больше вдоль оси *Y*, нежели *X*. Наша цель — скорректировать это направление вращением. Нужно поворачивать точки вокруг начала координат до тех пор, пока они не наложатся на ось *X*. Вращение двухмерного графика вокруг его центра требует использования *матрицы вращения* — массива 2 × 2 в форме `np.array([[cos(x), -sin(x)], [sin(x), cos(x)]])`, где `x` является углом вращения. Матричное умножение этого массива на `centered_data` повернет данные на `x` радиан. Вращение происходит против часовой стрелки. Для поворота данных по часовой стрелке нужно вместо `x` передать `-x`.

Теперь мы используем матрицу вращения для поворота `centered_data` по часовой стрелке на 90° (листинг 14.6), после чего построим график повернутых данных и исходного массива `centered_data` (рис. 14.6).

Листинг 14.6. Поворот centered_data на 90°

```
from math import sin, cos
angle = np.radians(-90)
rotation_matrix = np.array([[cos(angle), -sin(angle)],
                            [sin(angle), cos(angle)]])
rotated_data = rotation_matrix @ centered_data
plt.scatter(centered_data[0], centered_data[1], label='Original Data')
plt.scatter(rotated_data[0], rotated_data[1], c='y', label='Rotated Data')
plt.axhline(0, c='black')
plt.axvline(0, c='black')
plt.legend()
plt.axis('equal')
plt.show()
```

Преобразует угол из градусов в радианы

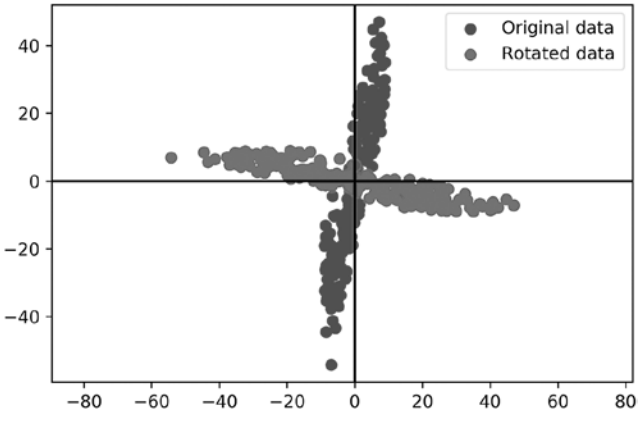


Рис. 14.6. График centered_data до и после поворота.
Данные были повернуты на 90° относительно начала координат.
Теперь они расположены ближе к оси X

Как и ожидалось, результат `rotated_data` оказался перпендикулярен графику `centered_data`. Мы успешно повернули график на 90°. Кроме того, вращение сместило его ближе к оси *X*. Теперь нам нужен способ оценить этот сдвиг количественно. Для этого сгенерируем штрафные баллы, которые по мере поворота точек данных в сторону оси *X* будут уменьшаться.

Мы будем накладывать штраф на все вертикальные значения оси *Y* (листинг 14.7). Штраф основан на принципе квадрата расстояния, описанном в главе 5. Квадрат штрафа равен среднему квадрату значения *y* в `rotated_data`. Поскольку значения *y* представляют расстояние до оси *X*, штраф равняется среднему квадрату расстояния до нее. Когда поворачиваемый набор данных сдвигается ближе к оси *X*, средний квадрат его значения *y* уменьшается.

Имея массив `y_values`, можно вычислить штраф, выполнив `sum([y ** 2 for y in y_values]) / y_values.size`. Также его можно вычислить с помощью `y_values @ y_values / y.size`. Два этих результата будут идентичны, но вычисление скалярного произведения более эффективно. Сравним штрафной балл для `rotated_data` и `centered_data`.

Листинг 14.7. Наложение штрафа на вертикальные значения y

```
data_labels = ['unrotated', 'rotated']
data_list = [centered_data, rotated_data]
for data_label, data in zip(data_labels, data_list):
    y_values = data[1]
    penalty = y_values @ y_values / y_values.size
    print(f"The penalty score for the {data_label} data is {penalty:.2f}")

The penalty score for the unrotated data is 519.82
The penalty score for the rotated data is 27.00
```

Поворот данных уменьшил штрафной балл более чем в 20 раз. Уменьшение интерпретируется статистически. При этом штраф можно связать с дисперсией, если учесть следующее:

- величина штрафа равна среднему квадрату расстояния *y* от 0 по массиву `y_values`;
- `y_values.mean()` равно `0`;
- квадрат штрафа равен среднему квадрату расстояния *y* от этого среднего;
- средний квадрат расстояния от среднего равен дисперсии;
- штрафной балл равен `y_values.var()`.

Мы сделали вывод, что штрафной балл равен дисперсии оси *Y*. Следовательно, поворот данных уменьшил дисперсию оси *Y* более чем в 20 раз. Проверим это (листинг 14.8).

Листинг 14.8. Уравнивание штрафа с дисперсией оси Y

```
for data_label, data in zip(data_labels, data_list):
    y_var = data[1].var()
    penalty = data[1] @ data[1] / data[0].size
    assert round(y_var, 14) == round(penalty, 14)
    print(f"The y-axis variance for the {data_label} data is {y_var:.2f}")

The y-axis variance for the unrotated data is 519.82
The y-axis variance for the rotated data is 27.00
```

Выполняет округление для компенсации погрешностей из-за плавающей точки

Оценить величину поворота можно на основе дисперсии. Вращение данных в сторону оси *X* уменьшает дисперсию вдоль оси *Y*. А как это вращение повлияет на дисперсию вдоль оси *X*? Сейчас выясним (листинг 14.9).

Листинг 14.9. Измерение вращательной дисперсии оси X

```
for data_label, data in zip(data_labels, data_list):
    x_var = data[0].var()
    print(f"The x-axis variance for the {data_label} data is {x_var:.2f}")

The x-axis variance for the unrotated data is 27.00
The x-axis variance for the rotated data is 519.82
```

Поворот поменял местами дисперсию осей *X* и *Y*. Однако общая сумма значений дисперсии осталась неизменной. Общая дисперсия сохраняется даже после вращения. Проверим этот факт (листинг 14.10).

Листинг 14.10. Подтверждение сохранения общей дисперсии

```
total_variance = centered_data[0].var() + centered_data[1].var()
assert total_variance == rotated_data[0].var() + rotated_data[1].var()
```

Сохранение дисперсии позволяет сделать следующие выводы.

- Дисперсию по оси *X* и оси *Y* можно объединить в один процентный показатель, где `x_values.var() / total_variance` равно `1 – y_values.var() / total_variance`.
- Поворот данных в сторону оси *X* ведет к увеличению дисперсии по этой оси при ее равнозначном уменьшении по оси *Y*. Уменьшение вертикальной дисперсии на *p* процентов увеличивает горизонтальную дисперсию на *p* процентов.

Код листинга 14.11 подтверждает эти выводы.

Листинг 14.11. Изучение процентного соотношения дисперсии вдоль осей

```
for data_label, data in zip(data_labels, data_list):
    percent_x_axis_var = 100 * data[0].var() / total_variance
    percent_y_axis_var = 100 * data[1].var() / total_variance
    print(f"In the {data_label} data, {percent_x_axis_var:.2f}% of the "
          "total variance is distributed across the x-axis")
    print(f"The remaining {percent_y_axis_var:.2f}% of the total "
          "variance is distributed across the y-axis\n")
```

```
In the unrotated data, 4.94% of the total variance is distributed across the x-axis
The remaining 95.06% of the total variance is distributed across the y-axis

In the rotated data, 95.06% of the total variance is distributed across the x-axis
The remaining 4.94% of the total variance is distributed across the y-axis
```

Поворот данных в сторону оси *X* увеличил дисперсию по этой оси на 90 процентных единиц. В то же время поворот уменьшил дисперсию по оси *Y* на те же 90 процентных единиц.

Повернем `centered_data` еще дальше, пока их расстояние до оси *X* не окажется минимальным. Минимизация этого расстояния равнозначна:

- минимизации процента общей дисперсии по оси *Y*. Это ведет к минимизации вертикального рассеяния;
- максимизации процента общей дисперсии по оси *X*. Это ведет к максимизации горизонтального рассеяния.

Мы поворачиваем `centered_data` в сторону оси *X* путем максимизации горизонтального рассеяния (листинг 14.12). Оно измеряется по всем углам от 1 до 180°. Эти измерения показываем на графике (рис. 14.7). Кроме того, вычитаем угол вращения, который максимизирует процент охвата дисперсии по оси *X*. Наш код не только выводит данный угол и процент, но и отмечает угол на графике.

Листинг 14.12. Максимизация горизонтального рассеяния

```
def rotate(angle, data=centered_data):
    angle = np.radians(-angle)
    rotation_matrix = np.array([[cos(angle), -sin(angle)],
                                [sin(angle), cos(angle)]])
    return rotation_matrix @ data

angles = np.arange(1, 180, 0.1)
x_variances = [(rotate(angle)[0].var()) for angle in angles]

percent_x_variances = 100 * np.array(x_variances) / total_variance
optimal_index = np.argmax(percent_x_variances)
optimal_angle = angles[optimal_index]
plt.plot(angles, percent_x_variances)
plt.axvline(optimal_angle, c='k')
plt.xlabel('Angle (degrees)')
plt.ylabel('% x-axis coverage')
plt.show()

max_coverage = percent_x_variances[optimal_index]
max_x_var = x_variances[optimal_index]

print("The horizontal variance is maximized to approximately "
      f"{int(max_x_var)} after a {optimal_angle:.1f} degree rotation.")
```

Поворачивает данные на входное число градусов. Переменная данных установлена на centered_data

Вычисляет дисперсию по оси X для поворота на каждый угол

Возвращает массив углов от 0 до 180°, где каждый последующий увеличивается на 0,1°

Вычисляет угол поворота, дающий максимальную дисперсию

Строит вертикальную линию через optimal_angle

```
print(f"That rotation distributes {max_coverage:.2f}% of the total "
       "variance onto the x-axis.")

The horizontal variance is maximized to approximately 541 after a 78.3 degree
     rotation.
That rotation distributes 99.08% of the total variance onto the x-axis.
```

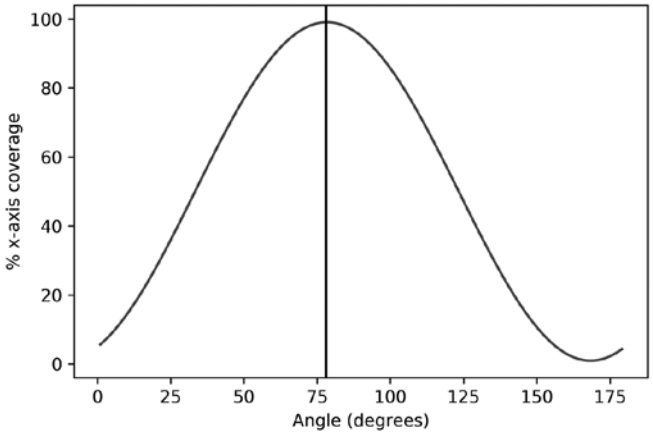


Рис. 14.7. График соотношения угла вращения и процента общей дисперсии, охватываемой осью X. Вертикальная линия обозначает угол, при котором дисперсия по оси X максимальна. Угол поворота в 78,3° смещает более 99 % общей дисперсии в сторону оси X. Поворот на этот угол позволит уменьшить размерность данных

Поворот `centered_data` на 78,3° максимально увеличит горизонтальное рассеяние. При таком угле поворота 99,08 % общей дисперсии будет распределено вдоль оси *X*. Таким образом, можно ожидать, что повернутые данные лягут преимущественно вдоль одномерной линии оси. Убедимся в этом, выполнив вращение и построив график результатов (листинг 14.13; рис. 14.8).

Листинг 14.13. Построение графика повернутых данных с высоким охватом дисперсии осью X

```
best_rotated_data = rotate(optimal_angle)
plt.scatter(best_rotated_data[0], best_rotated_data[1])
plt.axhline(0, c='black')
plt.axvline(0, c='black')
plt.axis('equal')
plt.show()
```

Бóльшая часть данных лежит близко к оси *X*, то есть рассеяние данных максимизируется в горизонтальном направлении. Сильно разбросанные точки по определению сильно разделены, а значит, хорошо различимы. В противоположность этому рассеяние вдоль оси *Y* было минимизировано. По вертикали точки данных различить сложно, поэтому можно удалить все координаты y с минимальной потерей информации. Это удаление должно вписываться в пределы 1 % общей

дисперсии, поэтому оставшихся значений по оси *X* для кластеризации наших характеристик будет достаточно.

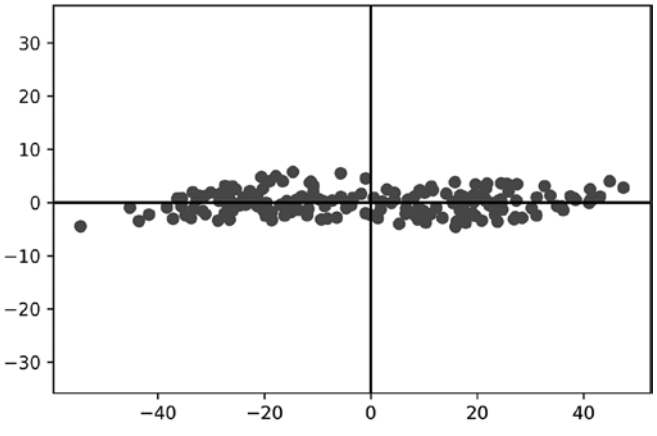


Рис. 14.8. График centered_data, повернутых на 78,3°. Такой поворот максимизирует дисперсию вдоль оси X и минимизирует — вдоль оси Y. Благодаря этому можно удалить координаты y с минимальной потерей информации

Теперь уменьшим `best_rotated_data` до одного измерения, избавившись от оси *Y* (листинг 14.14). После этого используем оставшийся одномерный массив для извлечения двух порогов кластеризации. Первый порог отделит клиентов с малым размером одежды от клиентов со средним, а второй — клиентов со средним размером от клиентов с большим. Вместе эти два порога разделят 180 клиентов на три кластера равного размера.

Листинг 14.14. Уменьшение повернутых данных до одного измерения для кластеризации

```
x_values = best_rotated_data[0]
sorted_x_values = sorted(x_values)
cluster_size = int(x_values.size / 3)
small_cutoff = max(sorted_x_values[:cluster_size])
large_cutoff = min(sorted_x_values[-cluster_size:])
print(f"A 1D threshold of {small_cutoff:.2f} separates the small-sized "
       "and medium-sized customers.")
print(f"A 1D threshold of {large_cutoff:.2f} separates the medium-sized "
       "and large-sized customers.")
```

```
A 1D threshold of -14.61 separates the small-sized and medium-sized customers.
A 1D threshold of 15.80 separates the medium-sized and large-sized customers.
```

Пороги можно визуализировать, задействовав их для вертикального разделения графика `best_reduced_data`. Эти два среза разделят график на три сегмента, каждый из которых будет соответствовать размеру одежды клиента (листинг 14.15). Затем мы визуализируем пороги и сегменты, причем каждый сегмент окрасим (рис. 14.9).

Листинг 14.15. Построение графика горизонтальных данных о клиентах, разделенных на три сегмента

```
def plot_customer_segments(horizontal_2d_data):
    small, medium, large = [], [], []
    cluster_labels = ['Small', 'Medium', 'Large']
    for x_value, y_value in horizontal_2d_data.T:
        if x_value <= small_cutoff:
            small.append([x_value, y_value])
        elif small_cutoff < x_value < large_cutoff:
            medium.append([x_value, y_value])
        else:
            large.append([x_value, y_value])
    for i, cluster in enumerate([small, medium, large]):
        cluster_x_values, cluster_y_values = np.array(cluster).T
        plt.scatter(cluster_x_values, cluster_y_values,
                    color=['g', 'b', 'y'][i],
                    label=cluster_labels[i])

    plt.axhline(0, c='black')
    plt.axvline(large_cutoff, c='black', linewidth=3, linestyle='--')
    plt.axvline(small_cutoff, c='black', linewidth=3, linestyle='--')
    plt.axis('equal')
    plt.legend()
    plt.show()

plot_customer_segments(best_rotated_data)
```

Получает на входе горизонтально расположенный набор данных о клиентах, сегментирует этот набор с помощью вертикальных порогов и строит графики сегментов по отдельности. Эта функция используется в данной главе неоднократно

Пороги одномерных значений x применяются для сегментации данных

Каждый сегмент клиентов отображается на графике отдельно

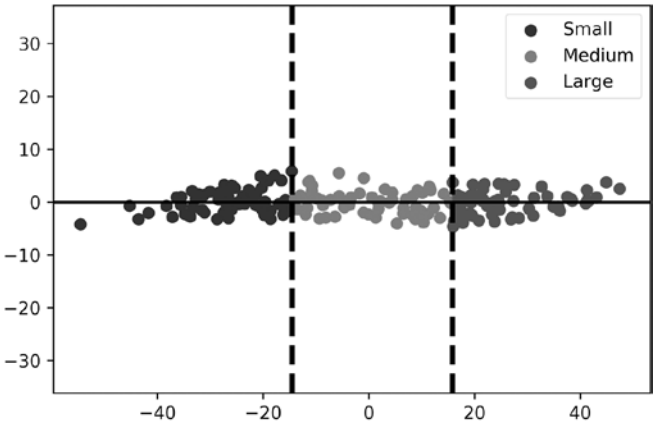


Рис. 14.9. Горизонтальный график centered_data, сегментированный при помощи двух вертикальных порогов. Сегментация разделяет его на три кластера клиентов: с малым, средним и большим размерами. Одномерной оси X достаточно для извлечения этих кластеров

Одномерного массива `x-values` вполне достаточно для сегментации информации о клиентах, потому что он охватывает 99,08 % дисперсии данных. Это значит, что массив можно использовать для воссоздания 99,08 % набора данных `centered_data` (рис. 14.10). Нужно лишь заново ввести измерение оси *Y*, добавив массив нулей. После этого останется повернуть полученный массив обратно в исходное положение. Все эти шаги выполняются в коде листинга 14.16.

Листинг 14.16. Воссоздание двухмерных данных из одномерного массива

```
zero_y_values = np.zeros(x_values.size) ◄—— Возвращает вектор нулей
reproduced_data = rotate(-optimal_angle, data=[x_values, zero_y_values])
```

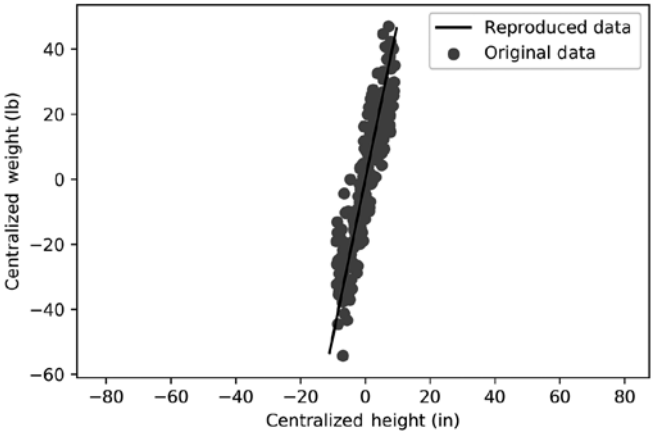


Рис. 14.10. График воссозданных данных вместе с оригинальными точками данных. Массив reproduced_data формирует одну линию, рассекающую точечный график centered_data. Эта линия представляет линейное направление, в котором дисперсия данных максимальна. Линия reproduced_data охватывает 99,08 % общей дисперсии

Теперь построим график `reproduced_data` вместе с матрицей `centered_data` для оценки качества воспроизведения (листинг 14.17).

Листинг 14.17. Построение графика воссозданных и исходных данных

```
plt.plot(reproduced_data[0], reproduced_data[1], c='k',
         label='Reproduced Data')
plt.scatter(centered_data[0], centered_data[1], c='y',
            label='Original Data')
plt.axis('equal')
plt.legend()
plt.show()
```

Воспроизведенные данные формируют линию, проходящую через середину точечного графика `centered_data`. Эта линия представляет *первое основное направление*, являющееся линейным направлением, в котором дисперсия данных максимальна.

Большинство двухмерных наборов данных содержат два основных направления. *Второе основное направление* перпендикулярно первому. Оно представляет оставшуюся дисперсию, не охваченную первым.

Первое основное направление можно использовать для обработки значений высоты и роста будущих клиентов. Мы предположим, что эти клиенты берутся из одного распределения, лежащего в основе содержащихся в `measurements` данных. Если да, то централизованные значения их высоты и роста также лежат вдоль первого основного направления, показанного на рис. 14.10. Эта параллельность в конечном счете позволит нам сегментировать данные о новых клиентах с помощью имеющихся порогов.

Теперь рассмотрим этот сценарий глубже, симулировав новые характеристики клиентов. После этого централизуем полученные данные и построим их график (листинг 14.18; рис. 14.11). Кроме того, мы построим линию, представляющую первое основное направление, — ожидается, что график характеристик будет совпадать с ней.

ПРИМЕЧАНИЕ

Первое основное направление пересекается с исходным. Значит, в целях выравнивания надо обеспечить, чтобы новые данные о клиентах пересекались с исходными. То есть эти данные необходимо централизовать.

Листинг 14.18. Симуляция и построение графика новых клиентских данных

```
np.random.seed(1)
new_heights = np.arange(60, 78, .11)
random_fluctuations = np.random.normal(scale=10, size=new_heights.size)
new_weights = 4 * new_heights - 130 + random_fluctuations
new_centered_data = np.array([new_heights - heights.mean(),
                              new_weights - weights.mean()])
plt.scatter(new_centered_data[0], new_centered_data[1], c='y',
            label='New Customer Data')
plt.plot(reproduced_data[0], reproduced_data[1], c='k',
         label='First Principal Direction')
plt.xlabel('Centralized Height (in)')
plt.ylabel('Centralized Weight (lb)')
plt.axis('equal')
plt.legend()
plt.show()
```

Отделяет все новые значения высоты на 0,11 дюйма, чтобы минимизировать пересечение с ее прежними значениями

Предположим, что новое распределение клиентов такое же, как и прежнее. Это позволит использовать для централизации данных имеющиеся средства

Новые данные о клиентах все так же лежат вдоль первого основного направления. Оно охватывает более 99 % дисперсии данных, формируя угол 78,3° относительно оси *X*. Следовательно, чтобы положить данные набок, нужно повернуть их на 78,3°. Полученные значения *x* охватывают более 99 % общей дисперсии. Такая высокая горизонтальная дисперсия позволит нам сегментировать клиентов, не опираясь на значения *y*. Для этого должно быть достаточно имеющихся одномерных сегментирующих порогов.

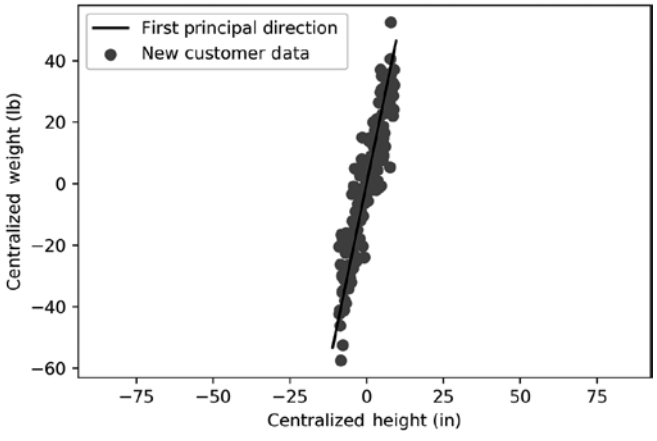


Рис. 14.11. Централизованный график, содержащий новые данные о клиентах и основное направление из исходного набора данных характеристик клиентов. Основное направление проходит прямо через ранее не встречавшийся график данных. Угол между этим направлением и осью X известен. Таким образом, можно уверенно положить эти данные набок, чтобы выполнить кластеризацию

Далее мы располагаем новые данные о клиентах горизонтально и сегментируем их с помощью функции `plot_customer_segments` (листинг 14.19; рис. 14.12).

Листинг 14.19. Поворот и сегментация новых данных о клиентах

```
new_horizontal_data = rotate(optimal_angle, data=new_centered_data)
plot_customer_segments(new_horizontal_data)
```

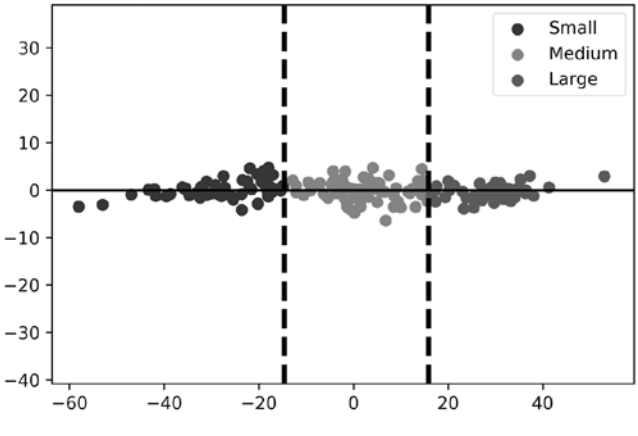


Рис. 14.12. Горизонтальный график новых данных о клиентах. Эти данные сегментированы при помощи двух вычисленных ранее вертикальных порогов. Сегментация разделяет график на три кластера клиентов: с малым, средним и большим размерами. Для извлечения этих кластеров достаточно одномерной оси X

Теперь вкратце подытожим наши наблюдения. Мы можем уменьшить любой двухмерный массив характеристик клиентов до одного измерения, повернув эти данные набок. Одномерных горизонтальных значений *x* должно быть достаточно для кластеризации клиентов по размеру одежды. Кроме того, данные проще повернуть, когда известно основное направление, вдоль которого дисперсия максимальна. Имея первое основное направление, можно уменьшить размерность данных о клиентах для удобства их кластеризации.

ПРИМЕЧАНИЕ

В качестве дополнительного бонуса уменьшение размерности позволяет упростить базу данных клиентов. Вместо сохранения и роста и веса достаточно просто сохранять горизонтальное значение x. Уменьшение хранилища базы данных с двух измерений до одного ускорит поиск клиентов и уменьшит затраты памяти.

До сих пор мы извлекали первое основное направление, вращая данные для максимизации дисперсии. К сожалению, этот прием не масштабируется на большее число измерений. Представьте, что вы анализируете 1000-мерный набор данных. Проверка каждого угла для 1000 разных осей окажется неоправданной с вычислительной точки зрения. Однако есть более простой способ извлечь все основные направления. Нужно лишь применить масштабируемый алгоритм, известный как *анализ главных компонент* (PCA).

PCA мы изучим в следующих нескольких разделах. Он прост в реализации, но понять его может быть нелегко. Поэтому будем рассматривать этот алгоритм по частям, начав с выполнения его реализации из scikit-learn. Мы применим PCA к нескольким наборам данных для более эффективной кластеризации и визуализации. Затем воссоздадим этот алгоритм с нуля, чтобы хорошо понять его слабые места, и в завершение эти слабости устраним.

14.2. УМЕНЬШЕНИЕ РАЗМЕРНОСТИ С ПОМОЩЬЮ PCA И SCIKIT-LEARN

Алгоритм PCA подстраивает оси набора данных так, чтобы бóльшая часть дисперсии распределялась вдоль небольшого числа измерений. В результате для различения точек данных потребуются не все измерения, а упрощенное различение данных приведет к упрощенной кластеризации. Поэтому хорошо, что в scikit-learn есть класс для анализа главных компонент, называемый `PCA`. Давайте импортируем PCA из `sklearn.decomposition` (листинг 14.20).

Листинг 14.20. Импорт PCA из scikit-learn

```
from sklearn.decomposition import PCA
```

Выполнение `PCA()` инициализирует объект `pca_model`, структурно схожий с объектами scikit-learn `cluster_model`, использованными в главе 10 (листинг 14.21). Тогда

мы создавали модели, способные кластеризовать входящие массивы. Теперь же создадим модель PCA, способную поворачивать массив `measurements` набок.

Листинг 14.21. Инициализация объекта pca_model

```
pca_object = PCA()
```

При помощи `pca_model` можно горизонтально повернуть двухмерную матрицу `data`, выполнив `model.fit_transform(data)`. Вызов этого метода ведет к присваиванию столбцам матрицы осей и затем переориентирует последние для максимизации дисперсии. Однако в массиве `measurements` оси хранятся в строках матрицы. Значит, нам нужно поменять местами строки и столбцы, получив транспонированную версию матрицы. При выполнении `pca_model.fit_transform(measurements.T)` возвращается матрица `pca_transformed_data`. Первый ее столбец представляет ось *X*, вдоль которой дисперсия максимальна, а второй — ось *Y*, вдоль которой она минимальна. График этих двух столбцов должен напоминать лежащую на боку сигару. Проверим это — выполним метод `fit_transform` для `measurements.T`, а затем построим график столбцов результата (листинг 14.22; рис. 14.13). Напомню, что к *i*-му столбцу матрицы NumPy можно обратиться через `M[:,i]`.

Листинг 14.22. Выполнение PCA с помощью scikit-learn

```
pca_transformed_data = pca_object.fit_transform(measurements.T)
plt.scatter(pca_transformed_data[:,0], pca_transformed_data[:,1])
plt.axhline(0, c='black')
plt.axvline(0, c='black')
plt.axis('equal')
plt.show()
```

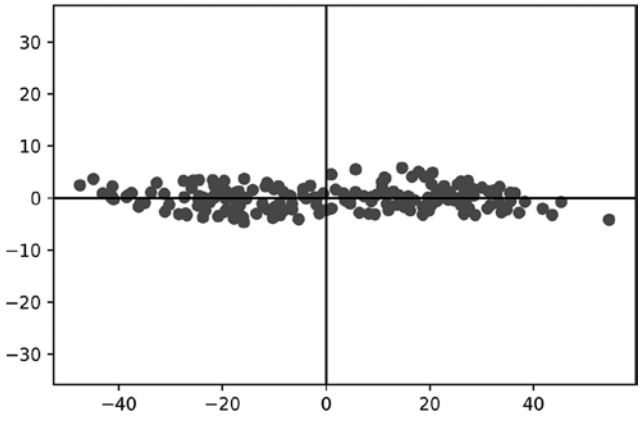


Рис. 14.13. График вывода алгоритма PCA из scikit-learn. Он представляет собой зеркальное отражение горизонтально расположенных данных клиентов с рис. 14.8. PCA переориентировал эти данные, в результате чего их дисперсия лежит преимущественно вдоль оси X. Так что ось Y можно удалить с минимальной потерей информации

Наш график является зеркальным представлением рис. 14.8 — значения *y* отражены вдоль оси *Y*. Выполнение PCA для двухмерного набора данных гарантированно расположит эти данные вдоль горизонтали — оси *X*. Однако фактическое отражение данных не ограничено одним конкретным направлением.

ПРИМЕЧАНИЕ

Мы можем воссоздать исходный горизонтальный график, умножив все значения y на –1. Следовательно, при выполнении plot_customer_segments((pca_transformed_data * np.array([1, -1])).T) будет сгенерирован график сегментированных размеров одежды клиентов, равнозначный рис. 14.9.

Несмотря на то что график наших данных ориентирован иначе, охват его дисперсии осью *X* должен согласовываться с прежними результатами. Убедиться в этом можно с помощью атрибута `explained_variance_ratio_` объекта `pca_object`. Этот атрибут содержит массив дробных величин дисперсии, охватываемой каждой осью. Таким образом, `100 * pca_object.explained_variance_ratio_[0]` должно равняться ранее полученному охвату оси *X*, то есть 99,08 %. Проверим (листинг 14.23).

Листинг 14.23. Извлечение дисперсии из вывода scikit-learn PCA

```
percent_variance_coverages = 100 * pca_object.explained_variance_ratio_
x_axis_coverage, y_axis_coverage = percent_variance_coverages
print(f"The x-axis of our PCA output covers {x_axis_coverage:.2f}% of "
       "the total variance")
```

Этот атрибут является массивом NumPy, содержащим дробный показатель охвата каждой осью. Умножение на 100 преобразует эти дроби в проценты

Каждый i-й элемент массива соответствует величине охвата дисперсии i-й осью

```
The x-axis of our PCA output covers 99.08% of the total variance
```

Наш `pca_object` максимизировал дисперсию вдоль оси *X*, раскрыв два основных направления набора данных. Эти направления хранятся в виде векторов в атрибуте `pca_components`, который является матрицей. Напомню, что векторы — это отрезки, указывающие в определенном направлении относительно начала координат. Кроме того, первое основное направление — это линия, исходящая из начала координат. Следовательно, его можно представить как вектор, называемый *первой главной компонентой*. К первой главной компоненте наших данных можно обратиться посредством вывода `pca_object.components[0]` (листинг 14.24). Далее мы выведем этот вектор вместе с его абсолютной величиной.

Листинг 14.24. Вывод первой главной компоненты

```
first_pc = pca_object.components_[0]
magnitude = norm(first_pc)
print(f"Vector {first_pc} points in a direction that covers "
      f"{x_axis_coverage:.2f}% of the total variance.")
print(f"The vector has a magnitude of {magnitude}")

Vector [-0.20223994 -0.979336 ] points in a direction that covers 99.08%
of the total variance.
The vector has a magnitude of 1.0
```

Первая главная компонента — это единичный вектор с абсолютной величиной 1,0. Его протяженность — целая единица длины от начала координат. Умножение этого вектора на число приводит к дальнейшему увеличению его протяженности. Если растянуть его таким образом достаточно далеко, то можно захватить все главное направление наших данных. Иными словами, можно растягивать вектор до тех пор, пока он полностью не пронзит наш сигаровидный график. Результат будет идентичен изображенному на рис. 14.10.

ПРИМЕЧАНИЕ

Если вектор pc — это главная компонента, тогда -pc — также главная компонента. Вектор -pc является зеркальным отражением pc. Они оба пролегают вдоль первого основного направления, хотя и указывают в противоположные стороны. Таким образом, pc и -pc в процессе уменьшения размерности можно использовать взаимозаменяемо.

Далее мы сгенерируем этот график (рис. 14.14). Во-первых, растянем вектор `first_pc`, пока он не охватит 50 единиц в положительном и отрицательном направлениях от начала координат (листинг 14.25). После этого построим график растянутого отрезка вдоль ранее вычисленной матрицы `centered_data`. Позднее мы используем этот отрезок, чтобы лучше понять принципы работы алгоритма PCA.

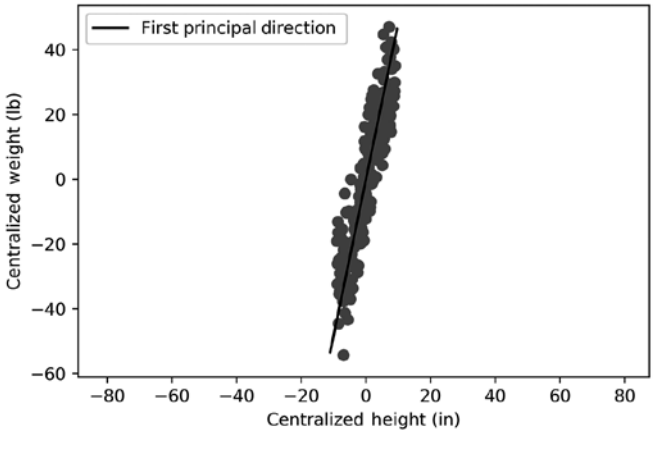


Рис. 14.14. Централизованный график, содержащий данные о клиентах и первое основное направление. Это направление было построено растягиванием первой главной компоненты

ПРИМЕЧАНИЕ

Мы строим график centered_data, а не measurements, потому что centered_data центрирован в начале координат. Наш растянутый вектор -out также центрирован в начале координат, что делает центрированную матрицу и вектор визуально сопоставимыми.

Листинг 14.25. Растягивание единичного вектора для охвата первого основного направления

```
def plot_stretched_vector(v, **kwargs):
    plt.plot([-50 * v[0], 50 * v[0]], [-50 * v[1], 50 * v[1]], **kwargs)

plt.plot(reproduced_data[0], reproduced_data[1], c='k',
         label='First Principal Direction')
plt.scatter(centered_data[0], centered_data[1], c='y')
plt.xlabel('Centralized Height (in)')
plt.ylabel('Centralized Weight (lb)')
plt.axis('equal')
plt.legend()
plt.show()
```

Эта функция растягивает входной вектор v. Растянутый отрезок простирается на 50 единиц в обоих направлениях от начала координат. Затем он отображается на графике. Вскоре эта функция будет использована повторно

Мы задействовали первую главную компоненту, чтобы направить набор данных вдоль его первого основного направления. Аналогичным образом можно растянуть и другой направленный единичный вектор, возвращаемый алгоритмом PCA. Как уже говорилось, большинство двухмерных наборов данных содержат два основных направления. Второе основное направление перпендикулярно первому, а его векторизованное представление называется *второй главной компонентой*. Эта компонента хранится во второй строке вычисленной нами матрицы `components`.

Почему нам интересна вторая главная компонента? В конце концов, она указывает в направлении, охватывающем менее 1 % дисперсии данных. Тем не менее для этой компоненты есть применение. И первая и вторая главные компоненты имеют особую связь с осями *X* и *Y* нашего набора данных. В связи с этим мы сейчас растянем и отразим на графике обе компоненты из матрицы `components` (листинг 14.26). Кроме того, построим на графике `centered_data`, а также обе оси. Итоговое визуальное представление позволит сделать ценные выводы (рис. 14.15).

Листинг 14.26. Построение графика основных направлений, осей и данных

```
principal_components = pca_object.components_
for i, pc in enumerate(principal_components):
    plot_stretched_vector(pc, c='k',
                          label='Principal Directions' if i == 0 else None)

for i, axis_vector in enumerate([np.array([0, 1]), np.array([1, 0])]):
    plot_stretched_vector(axis_vector, c='g', linestyle='-.',
                          label='Axes' if i == 0 else None)

plt.scatter(centered_data[0], centered_data[1], c='y')
plt.xlabel('Centralized Height (in)')
plt.ylabel('Centralized Weight (lb)')
plt.axis('equal')
plt.legend()
plt.show()
```

Строит оси x и y путем растягивания двух единичных векторов (вертикального и горизонтального) так, чтобы их абсолютные величины совпали с растянутыми главными компонентами. В результате растянутые оси и растянутые главные компоненты оказываются визуально сопоставимыми

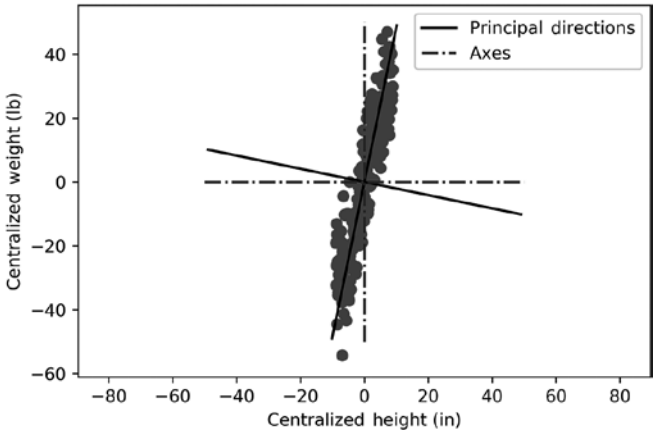


Рис. 14.15. Первое и второе основные направления, построенные вместе с графиком данных о клиентах. Эти направления перпендикулярны друг к другу. Если повернуть оси X и Y на 78,3°, то они идеально совпадут с основными направлениями. Таким образом, заменяя оси на основные направления, мы воссоздаем горизонтальный график, представленный на рис. 14.8

Согласно графику, два основных направления, по сути, являются повернутыми версиями осей *X* и *Y*. Представьте, что мы повернули их против часовой стрелки на 78,3°. После этого оси *X* и *Y* совпадут с двумя главными компонентами. Доли дисперсии, охватываемые этими осями, составят 99,02 и 0,08 % соответственно. Получается, что замена осей приведет к воссозданию горизонтального графика, приведенного на рис. 14.8.

ПРИМЕЧАНИЕ

Наклон головы влево при рассмотрении рис. 14.15 поможет представить результат.

Описанная подмена осей называется проекцией. Замена двух осей основными направлениями называется проекцией на основные направления. Используя тригонометрию, можно показать, что проекция `centered_data` на основные направления равна матричному произведению `centered_data` и двух главных компонент. Иными словами, `principal_components @ centered_data` изменяет положение координат *x* и *y* набора данных относительно основных направлений. Итоговый результат должен быть равен `pca_transformed_data.T`. Убедимся в этом с помощью кода из листинга 14.27.

ПРИМЕЧАНИЕ

В более широком смысле скалярное произведение между i-й главной компонентой и центрированной точкой данных проецирует эту точку на i-е основное направление. Значит, выполнение first_pc @ centered_data[i] проецирует i-ю точку данных на первое основное направление. Результат этой операции равен значению x, полученному при замене оси X на первое основное направление (pca_transformed_data[i][0]). Таким образом можно спроецировать несколько точек данных на несколько основных направлений, используя матричное умножение.

Листинг 14.27. Замена стандартных осей на основные направления с помощью проецирования

```
projections = principal_components @ centered_data
assert np.allclose(pca_transformed_data.T, projections)
```

Переориентированный вывод PCA зависит от проекции. В общем представлении алгоритм PCA работает так.

1. Централизует входные данные, вычитая их среднее из каждой точки данных.
2. Вычисляет главные компоненты набора данных. Подробности процесса вычисления будут рассмотрены в текущей главе позднее.
3. Получает матричное произведение между централизованными данными и главными компонентами. Эта операция заменяет стандартные оси данных на их основные направления.

Как правило, N-мерный набор данных имеет N основных направлений (по одному для каждой оси). При этом k-е основное направление максимизирует дисперсию, не охваченную первыми $k - 1$ направлениями. Таким образом, четырехмерный набор данных имеет четыре основных направления: первое максимизирует двунаправленную дисперсию, второе — всю двунаправленную дисперсию, не охваченную первым, а два последних включают всю оставшуюся дисперсию.

А вот здесь начинается самое интересное. Предположим, что мы проецируем четырехмерный набор данных на четыре его основных направления. Стандартные оси этого набора данных в результате заменяются на его основные направления. В соответствующих обстоятельствах две новые оси охватят значительную долю дисперсии. Следовательно, оставшиеся оси можно будет отбросить с минимальной потерей информации. Исключение этих двух осей уменьшит четырехмерный набор данных до двух измерений. Тогда можно будет визуализировать данные в виде двухмерного точечного графика. В идеале график сохранит достаточную дисперсию, чтобы правильно идентифицировать кластеры данных. Далее мы рассмотрим реальный сценарий, в котором визуализируем четырехмерные данные в двух измерениях.

КЛЮЧЕВАЯ ТЕРМИНОЛОГИЯ

- *Первое основное направление* — линейное направление, в котором дисперсия данных максимизирована. Замена оси X на первое основное направление переориентирует набор данных, чтобы максимизировать его распределение горизонтально. Переориентация позволяет выполнить более простую одномерную кластеризацию.
- *K-е основное направление* — линейное направление, максимизирующее дисперсию, не охваченную первыми $K - 1$ основными направлениями.

- *K-я главная компонента* — представление *K*-го основного направления в виде единичного вектора. Этот вектор можно задействовать для направленной проекции.
- *Проекция* — проецирование данных на основное направление аналогично замене стандартной оси этим направлением. Можно спроецировать набор данных на его *K* старших основных направлений, получив матричное произведение между централизованным набором данных и его *K* старшими главными компонентами.

14.3. КЛАСТЕРИЗАЦИЯ ЧЕТЫРЕХМЕРНЫХ ДАННЫХ В ДВУХ ИЗМЕРЕНИЯХ

Представьте, что мы ботаники, изучающие цветы на благоухающем лугу. Мы случайным образом отбираем 150 из них и для каждого записываем следующие характеристики:

- длину цветного лепестка;
- ширину цветного лепестка;
- длину зеленого листа под лепестком;
- ширину зеленого листа под лепестком.

Эти четырехмерные измерения уже существуют в scikit-learn и доступны для использования. Их можно получить, импортировав `load_iris` из `sklearn.datasets`. Вызов `load_iris()['data']` вернет матрицу, содержащую 150 строк и 4 столбца: каждая строка будет соответствовать цветку, а столбцы — размерам его листьев и лепестков. Далее мы загрузим эти данные и выведем соответствующие размеры для одного цветка (листинг 14.28). Все измерения приводятся в сантиметрах.

Листинг 14.28. Загрузка снятых с цветка размеров из scikit-learn

```
from sklearn.datasets import load_iris
flower_data = load_iris()
flower_measurements = flower_data['data']
num_flowers, num_measurements = flower_measurements.shape
print(f"{num_flowers} flowers have been measured.")
print(f"{num_measurements} measurements were recorded for every flower.")
print("The first flower has the following measurements (in cm): "
      f"{flower_measurements[0]}")
```

```
150 flowers have been measured.
4 measurements were recorded for every flower.
The first flower has the following measurements (in cm): [5.1 3.5 1.4 0.2]
```

Располагая матрицей цветочных размеров, мы ставим перед собой следующие цели:

- визуализировать данные о цветах в двухмерном пространстве;
- определить, имеются ли в этой визуализации кластеры;
- построить простую модель для различения между типами кластеров цветов (предполагается, что кластеры были найдены).

Начнем с визуализации данных. Они четырехмерные, но нам нужно отобразить их в 2D. Для уменьшения этих данных до двух измерений потребуется спроецировать их на первое и второе основные направления. Оставшиеся два направления можно будет отбросить. Таким образом, для анализа требуются лишь две первые главные компоненты.

Используя scikit-learn, можно ограничить анализ PCA до двух старших главных компонент. Потребуется лишь выполнить `PCA(n_components=2)` во время инициализации объекта PCA. Этот инициализированный объект сможет уменьшить входные данные до двухмерной проекции. Затем мы инициализируем двухкомпонентный объект PCA и используем `fit_transform`, чтобы уменьшить размерность цветов до 2D (листинг 14.29).

Листинг 14.29. Уменьшение размерности цветов до двух измерений

```
pca_object_2D = PCA(n_components=2)
transformed_data_2D = pca_object_2D.fit_transform(flower_measurements)
```

Вычисленная матрица `transformed_data_2D` должна быть двухмерной, состоящей всего из двух столбцов. Убедимся в этом (листинг 14.30).

Листинг 14.30. Проверка формы матрицы после уменьшения размерности

```
row_count, column_count = transformed_data_2D.shape
print(f"The matrix contains {row_count} rows, corresponding to "
      f"{row_count} recorded flowers.")
print(f"It also contains {column_count} columns, corresponding to "
      f"{column_count} dimensions.")
```

```
The matrix contains 150 rows, corresponding to 150 recorded flowers.
It also contains 2 columns, corresponding to 2 dimensions.
```

Какой объем от общей дисперсии данных охвачен итоговой матрицей? Выяснить это можно с помощью атрибута `explained_variance_ratio_` объекта `pca_object_2D` (листинг 14.31).

Листинг 14.31. Измерение охвата дисперсии матрицей после уменьшения ее размерности

Вычисляет охват дисперсии 2D-набора данных, ассоциируемого с pca_object. Эта функция используется в данной главе неоднократно

```
def print_2D_variance_coverage(pca_object):
    percent_var_coverages = 100 * pca_object.explained_variance_ratio_
```

```
    x_axis_coverage, y_axis_coverage = percent_var_coverages
    total_coverage = x_axis_coverage + y_axis_coverage
    print(f"The x-axis covers {x_axis_coverage:.2f}% "
          "of the total variance")
    print(f"The y-axis covers {y_axis_coverage:.2f}% "
          "of the total variance")
    print(f"Together, the 2 axes cover {total_coverage:.2f}% "
          "of the total variance")

print_2D_variance_coverage(pca_object_2D)

The x-axis covers 92.46% of the total variance
The y-axis covers 5.31% of the total variance
Together, the 2 axes cover 97.77% of the total variance
```

Матрица уменьшенной размерности охватывает более 97 % общей дисперсии данных. Таким образом, точечный график `transformed_data_2D` должен показывать бóльшую часть присутствующих в наборе данных паттернов (листинг 14.32; рис. 14.16).

Листинг 14.32. Построение графика данных о цветах в 2D

```
plt.scatter(transformed_data_2D[:,0], transformed_data_2D[:,1])
plt.show()
```

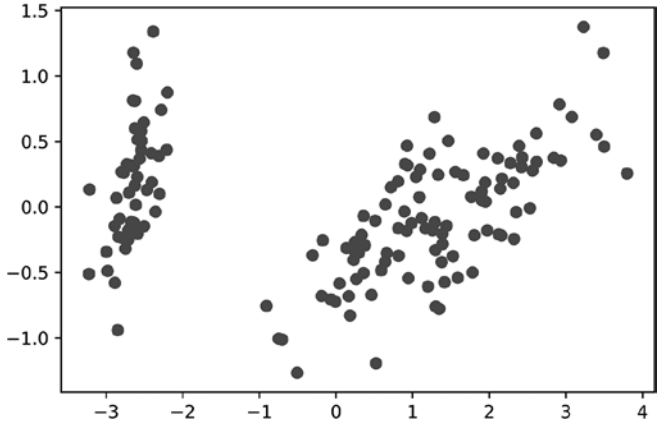


Рис. 14.16. Четырехмерные характеристики цветов, отраженные в двух измерениях. Размерность была уменьшена с помощью PCA. Уменьшенные данные охватывают более 97 % общей дисперсии. Полученный 2D-график информативен — на нем отчетливо видны два или три кластера цветов

Наши данные о цветах при построении в виде 2D-графика формируют кластеры. На основе этой кластеризации можно предположить, что имеются всего два или три типа цветов. В действительности же измеренные нами данные представляют три их уникальных вида. Информация о видах хранится в словаре `flower_data`.

Далее раскрасим наш график по видам цветов и убедимся, что цвета соотносятся с тремя отдельными кластерами (листинг 14.33; рис. 14.17).

Листинг 14.33. Раскрашивание графика по видам цветов

```
def visualize_flower_data(dim_reduced_data):
    species_names = flower_data['target_names']
    for i, species in enumerate(species_names):
        species_data = np.array([dim_reduced_data[j]
                                 for j in range(dim_reduced_data.shape[0])
                                 if flower_data['target'][j] == i]).T
        plt.scatter(species_data[0], species_data[1], label=species.title(),
                    color=['g', 'k', 'y'][i])
    plt.legend()
    plt.show()

visualize_flower_data(transformed_data_2D)
```

Строит график данных о цветах уменьшенной размерности, раскрашивая их в соответствии с видами

Эта функция используется в данной главе неоднократно. Возвращает названия трех видов цветов в наборе данных

Извлекает только те координаты, которые связаны с конкретными видами. Для фильтрации используем flower_data[target], которая сопоставляется со списком ID видов. Эти ID соответствуют трем названиям видов. Если j-й цветок соответствует species_name[i], то ID его вида равен j

Наносит на график каждый вид цветов, используя уникальный цвет

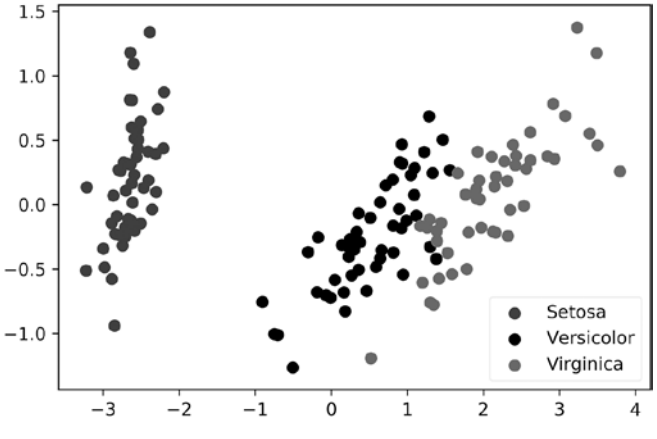


Рис. 14.17. Четырехмерные характеристики цветов, отраженные в виде двухмерного графика. Каждая точка закрашивается на основе вида соответствующего ей цветка. Три вида разделяются по трем кластерам. Таким образом, выполненное нами сокращение до двух измерений верно передает сигнал, необходимый для проведения различия между видами

По большей части эти три вида разделены в пространстве. Ирис разноцветный *(Versicolor)* и ирис виргинский *(Virginica)* немного пересекаются, значит, у их цветков есть общие характеристики. При этом ирис щетинистый *(Setosa)* формирует совершенно отдельный кластер. Вертикального порога значения x, равного -2, достаточно, чтобы изолировать ирис от других видов. В связи с этим можно

определить очень простую функцию обнаружения ириса. Она будет получать четырехэлементный массив `flower_sample`, содержащий четыре размера цветка, и делать следующее.

1. Централизовать образец вычитанием среднего по `flower_measurements` из `flower_sample`. Это среднее хранится в виде атрибута в `pca_object_2D` и равно `pca_object_2D.mean_`.
2. Проецировать централизованный образец на первое основное направление путем получения его скалярного произведения с первой главной компонентой. Напомню, что первая главная компонента хранится в `pca_object_2D.components_[0]`.
3. Проверять, меньше ли спроецированное значение, чем –2. Если да, то образец цветка будет рассматриваться как, возможно, относящийся к виду «ирис».

ПРИМЕЧАНИЕ

Функция обнаружения ириса не учитывает никакие иные виды цветов, кроме трех зафиксированных. Однако она должна успешно анализировать новые цветы на лугу, где новых видов не встречается.

Далее мы определим функцию `detect_setosa` и проанализируем образец цветка размерами (в сантиметрах) `[4.8, 3.7, 1.2, 0.24]` (листинг 14.34).

Листинг 14.34. Определение детектора ириса на основе данных уменьшенной размерности

```
def detect_setosa(flower_sample):
    centered_sample = flower_sample - pca_object_2D.mean_
    projection = pca_object_2D.components_[0] @ centered_sample
    if projection < -2:
        print("The sample could be a Setosa")
    else:
        print("The sample is not a Setosa")

new_flower_sample = np.array([4.8, 3.7, 1.2, 0.24])
detect_setosa(new_flower_sample)

The sample could be a Setosa
```

Если верить простому анализу относительно порога, который стал возможен благодаря PCA, данный образец цветка может являться ирисом. К преимуществам PCA относятся:

- визуализация сложных данных;
- упрощенная классификация и кластеризация данных;
- упрощенная классификация;

- меньшее потребление памяти. Сокращение размерности данных с четырех до двух измерений вдвое уменьшает количество байтов, необходимых для их хранения;
- ускорение вычислений. Уменьшение размерности данных с четырех до двух измерений в четыре раза сокращает время, необходимое для вычисления матричного сходства.

Итак, готовы ли мы использовать PCA для кластеризации текстовых данных? К сожалению, нет. Сначала необходимо рассмотреть ряд характерных для этого алгоритма недочетов и избавиться от них.

ТИПИЧНЫЕ МЕТОДЫ SCIKIT-LEARN PCA

- `pca_object = PCA()` — создает объект PCA, способный переориентировать входные данные так, чтобы их оси совпали с его основными направлениями.
- `pca_object = PCA(n_components=K)` — создает объект PCA, способный переориентировать входные данные так, чтобы *K* их осей совпали с *K* старших основных направлений. Все остальные оси игнорируются. Таким образом размерность данных сокращается до *K* измерений.
- `pca_transformed_data = pca_object.fit_transform(data)` — выполняет PCA для входных данных с использованием инициализированного объекта PCA. Метод `fit_transform` предполагает, что столбцы матрицы `data` соответствуют пространственным осям. Затем эти оси выравниваются с основным направлением данных, а полученный результат сохраняется в матрице `pca_transformed_data`.
- `pca_object.explained_variance_ratio_` — возвращает дробный показатель охвата дисперсии, связанный с каждым основным направлением скорректированного объекта PCA. Каждый *i*-й элемент соответствует дробному показателю охвата дисперсии вдоль *i*-го основного направления.
- `pca_object.mean_` — возвращает среднее входных данных, которые были скорректированы в объект PCA.
- `pca_object.components_` — возвращает главные компоненты входных данных, которые были скорректированы в объект PCA. Каждая *i*-я строка матрицы `components_` соответствует *i*-й главной компоненте. Выполнение `pca_object.components_[i] @ (data[j] - pca_object.mean_)` проецирует *j*-ю точку данных на *i*-ю главную компоненту. Спроецированный результат равен `pca_transformed_data[j][i]`.

14.3.1. Ограничения PCA

У PCA есть серьезные ограничения. Этот алгоритм излишне чувствителен к единицам измерения. К примеру, все размеры цветов выражены в сантиметрах, но можно представить, что мы преобразуем первую ось в миллиметры выполнением `10 * flower_measurements[0]`. Информационное содержание этой оси измениться не должно, однако ее дисперсия сместится. Преобразуем единицы измерения этой оси, чтобы понять, насколько изменится ее дисперсия (листинг 14.35).

Листинг 14.35. Оценка влияния изменения единиц измерения на дисперсию оси

```
first_axis_var = flower_measurements[:,0].var()
print(f"The variance of the first axis is: {first_axis_var:.2f}")

flower_measurements[:,0] *= 10
first_axis_var = flower_measurements[:,0].var()
print("We've converted the measurements from cm to mm.\nThat variance "
      f"now equals {first_axis_var:.2f}")

The variance of the first axis is: 0.68
We've converted the measurements from cm to mm.
That variance now equals 68.11
```

Дисперсия увеличилась в 100 раз. Теперь этот показатель для первой оси в нашем наборе данных доминирует. Рассмотрим последствия выполнения PCA для измененных размеров цветов: алгоритм попытается найти ось с максимальной дисперсией. Естественно, он найдет первую ось, где дисперсия возросла с 0,68 до 68. Следовательно, PCA спроецирует все данные на эту ось, в результате чего уменьшенный набор данных свернется в одно измерение. Подтвердить это можно повторным приведением `pca_object_2D` к `flower_measurements` с последующим выводом охвата дисперсии (листинг 14.36).

Листинг 14.36. Оценка влияния изменения единиц измерения на PCA

```
pca_object_2D.fit_transform(flower_measurements)
print_2D_variance_coverage(pca_object_2D)

The x-axis covers 98.49% of the total variance
The y-axis covers 1.32% of the total variance
Together, the 2 axes cover 99.82% of the total variance
```

Повторно подстраивает объект PCA под данные flower_measurements

Более 98 % дисперсии теперь лежит вдоль одной оси. Ранее для охвата ее 97 % требовались два измерения. Очевидно, что мы внесли в данные ошибку. Как же от нее избавиться? Первым на ум приходит вариант сделать так, чтобы на всех осях использовались одинаковые единицы измерения. Однако подобный практический подход не всегда возможен. Иногда единицы измерения просто недоступны. В других случаях оси соответствуют разным видам измерений (например, росту и весу), в связи с чем их единицы оказываются несовместимыми. Что же делать?

Рассмотрим основную причину смещения дисперсии. Мы сделали значения в `flower_measurements[:,0]` раз больше, значит, и их дисперсия стала больше. Различия в дисперсии осей обусловливаются различиями в значениях на этих осях. В предыдущей главе мы смогли избавиться от различий в размерах с помощью нормализации. Напомню, что во время нормализации вектор делится на свою абсолютную величину. В результате получается единичный вектор, абсолютная величина которого 1,0. Таким образом, если нормализовать наши оси, то все их значения будут лежать между 0 и 1, а значит, преобладание первой оси будет устранено. Давайте нормализуем `flower_measurements`, а затем уменьшим эти нормализованные данные до двух измерений (листинг 14.37). Полученный двухмерный охват дисперсии должен снова аппроксимироваться к 97 %.

Листинг 14.37. Нормализация данных для устранения различий в единицах измерения

```
for i in range(flower_measurements.shape[1]):
    flower_measurements[:,i] /= norm(flower_measurements[:,i])

transformed_data_2D = pca_object_2D.fit_transform(flower_measurements)
print_2D_variance_coverage(pca_object_2D)

The x-axis covers 94.00% of the total variance
The y-axis covers 3.67% of the total variance
Together, the 2 axes cover 97.67% of the total variance
```

Нормализация немного изменила данные. Теперь первое основное направление охватывает не 92,46 % общей дисперсии, а 94 %. При этом вторая главная компонента охватывает не 5,31 % общей дисперсии, а 3,67 %. Несмотря на эти изменения, общий двухмерный охват дисперсии по-прежнему суммируется приблизительно в 97 %. Мы еще раз построим график вывода PCA для подтверждения неизменности двухмерных паттернов кластеризации (листинг 14.38; рис. 14.18).

Листинг 14.38. Построение графика двухмерного вывода PCA после нормализации

```
visualize_flower_data(transformed_data_2D)
```

Полученный график немного отличается от предыдущих результатов. Однако три вида цветов по-прежнему разделяются на три кластера, и ирис сохраняет свою пространственную обособленность от других видов. Нормализация сохранила имеющееся разделение кластеров, устранив при этом ошибку, вызванную различием в единицах измерения.

К сожалению, нормализация приводит к непредвиденным последствиям. Нормализованные значения осей теперь лежат в диапазоне от 0 до 1, в связи с чем среднее каждой оси также находится между 0 и 1. Все значения находятся на расстоянии менее 1 от их среднего, и это проблема: PCA с целью централизации данных требует вычесть среднее из каждого значения оси. После этого централизованная матрица умножается на главные компоненты для выравнивания осей. К сожалению,

централизация данных не всегда оказывается достижима ввиду погрешностей из-за плавающей точки. Вычислительно сложно выполнить вычитание похожих значений со 100%-ной точностью, поэтому трудно вычесть среднее из значения, которое очень к нему близко. Предположим, мы анализируем массив с двумя точками данных, `1 + 1e-3` и `1 – 1e-3`. Среднее этого массива равно `1`. Вычитание из него `1` должно давать централизованное среднее `0`, но фактическое среднее не будет равно `0` из-за погрешности, которая продемонстрирована в листинге 14.39.

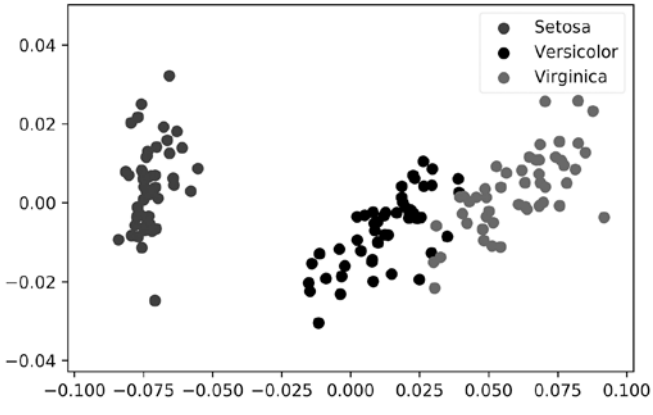


Рис. 14.18. Четырехмерные нормализованные размеры цветов, построенные в двухмерном пространстве. Каждая соответствующая цветку точка окрашена согласно виду этого цветка. Таким образом, сокращение до двух измерений правильно передает сигнал, необходимый для проведения различий между видами

Листинг 14.39. Демонстрация погрешностей, вызванных значениями, близкими к своим средним

```
data = np.array([1 + 1e-3, 1 - 1e-3])
mean = data.mean()
assert mean == 1
centralized_data = data - 2 * [mean]
assert centralized_data.mean() != 0
print(f"Actual mean is equal to {centralized_data.mean()}")
```

Среднее значение централизованных данных, вопреки предположению, не равно 0

```
Actual mean is equal to -5.551115123125783e-17
```

Невозможно надежно централизовать данные, которые близки к своему среднему. В связи с этим нельзя надежно выполнить для нормализованных данных PCA. Как быть в такой ситуации?

Решение этой проблемы есть, однако для его обнаружения придется углубленно разобрать механизм действия алгоритма PCA. Необходимо узнать, как вычислять главные компоненты с нуля без вращения данных. Этот процесс несколько абстрактен, но его можно понять, не изучая высшую математику. Разобравшись,

как строится алгоритм PCA с нуля, мы сможем слегка его скорректировать. Это позволит обойтись без централизации данных. Измененный алгоритм, известный как *сингулярная декомпозиция* (SVD), позволит эффективно кластеризовать текстовые данные.

ПРИМЕЧАНИЕ

Если вас не интересует, как создается SVD, можете сразу пролистать книгу до последнего раздела. В нем будет описано применение SVD в scikit-learn.

14.4. ВЫЧИСЛЕНИЕ ГЛАВНЫХ КОМПОНЕНТ БЕЗ ВРАЩЕНИЯ

В данном разделе мы узнаем, как извлекать главные компоненты с нуля. Чтобы лучше проиллюстрировать этот процесс, визуализируем наши векторы компонент. Естественно, проще всего отражать векторы, когда они двухмерны, поэтому начнем с возвращения к набору данных `measurements`, главные компоненты которого представлены в 2D. Напомню, что мы вычислили следующие результаты для этих данных:

- `centralized_data` — централизованная версия набора данных `measurements`. Среднее `centralized_data` равно `[0 0]`.
- `first_pc` — первая главная компонента набора данных `measurements`, представленная двухэлементным массивом.

Как уже говорилось, `first_pc` — это единичный вектор, указывающий в первом основном направлении. Это направление максимизирует дисперсию данных. Ранее мы находили первую главную компоненту, вращая двухмерный набор данных. Целью этого было либо максимизировать дисперсию по оси *X*, либо минимизировать ее по оси *Y*. Однако дисперсию вдоль оси можно более эффективно вычислить с помощью матричного умножения. Самое же важное то, что за счет сохранения дисперсии в матрице можно извлечь компоненты без вращения. Рассмотрим следующее:

- мы уже показали, что дисперсия массива `axis` равна `axis @ axis / axis.size` (см. листинг 14.8);
- таким образом, дисперсия оси `i` в `centered_data` равна `centered_data[i] @ centered_data[i] / centered_data.shape[1]`;
- следовательно, выполнение `centered_data @ centered_data.T / centered_data.shape[1]` даст матрицу `m`, где `m[i][i]` равно дисперсии вдоль оси `i`.

По сути, можно вычислить дисперсии по всем осям одной матричной операцией. Нужно лишь умножить матрицу на ее транспонированную версию и разделить на размер данных. В результате получится новая матрица, называемая *ковариационной матрицей*. Ее диагональ хранит дисперсию вдоль каждой оси.

ПРИМЕЧАНИЕ

Недиагональные элементы ковариационной матрицы также обладают информативными свойствами: они определяют направление линейного уклона между двух осей. В centered_data этот уклон между X и Y положителен. Следовательно, недиагональные элементы в centered_data @ centered_data.T также положительны.

Далее мы вычислим ковариационную матрицу `centered_data` (листинг 14.40) и присвоим ее переменной `cov_matrix`. После этого убедимся, что `cov_matrix[i][i]` равна дисперсии *i*-й оси для каждого `i`.

Листинг 14.40. Вычисление ковариационной матрицы

```
cov_matrix = centered_data @ centered_data.T / centered_data.shape[1]
print(f"Covariance matrix:\n {cov_matrix}")
for i in range(centered_data.shape[0]):
    variance = cov_matrix[i][i]
    assert round(variance, 10) == round(centered_data[i].var(), 10)

Covariance matrix:
[[ 26.99916667 106.30456732]
[106.30456732 519.8206294 ]]
```

Выполняет округление для компенсации погрешности из-за плавающей точки

Ковариационная матрица и главные компоненты обладают особой полезной связью: нормализованное произведение ковариационной матрицы и главной компоненты равно этой главной компоненте. То есть нормализация `cov_matrix @ first_pc` дает вектор, идентичный `first_pc`. Давайте проиллюстрируем эту связь, построив график `first_pc` и нормализованного произведения `cov_matrix` с `first_pc` (листинг 14.41; рис. 14.19).

ПРИМЕЧАНИЕ

При получении произведения матрицы и вектора мы рассматриваем вектор как таблицу с одним столбцом. Таким образом, вектор с x элементов рассматривается как матрица с x рядов и одним столбцом. После перестраивания вектора в матрицу выполняется стандартное матричное умножение. Оно дает матрицу с одним столбцом, равнозначную вектору. Следовательно, произведение матрицы M и вектора V равно np.array([row @ v for row in M]).

Листинг 14.41. Раскрытие связи между cov_matrix и first_pc

```
def plot_vector(vector, **kwargs):
    plt.plot([0, vector[0]], [0, vector[1]], **kwargs)

plot_vector(first_pc, c='y', label='First Principal Component')
product_vector = cov_matrix @ first_pc
product_vector /= norm(product_vector)
plot_vector(product_vector, c='k', linestyle='--',
            label='Normalized Product Vector')

plt.legend()
plt.axis('equal')
plt.show()
```

Эта вспомогательная функция строит 2D-вектор в виде отрезка, исходящего из начала координат

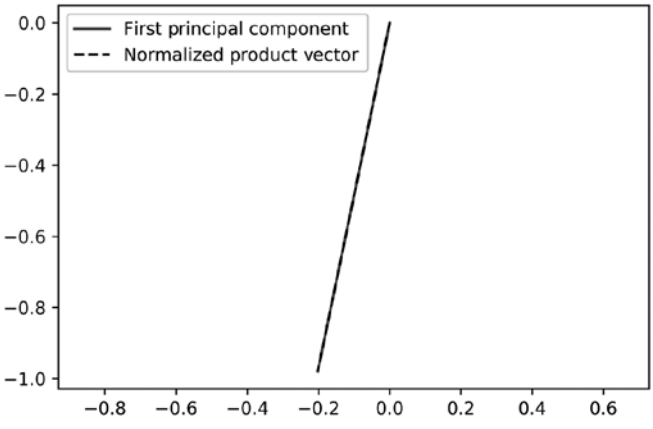


Рис. 14.19. График first_pc вместе с нормализованным произведением cov_matrix и first_pc. Два этих вектора идентичны. Главная компонента и произведение ковариационной матрицы и главной компоненты указывают в одном направлении

Два построенных вектора идентичны. Произведение `cov_matrix` и `first_pc` ориентировано в том же направлении, что и `first_pc`. Таким образом, по определению `first_pc` является *собственным вектором* `cov_matrix`. Сколько бы раз мы ни вычисляли произведение, его направление не изменится. Итак, `cov_matrix @ product_vector` указывает в том же направлении, что и `product_vector`, а угол между этими векторами равен нулю. Проверим это утверждение (листинг 14.42).

Листинг 14.42. Вычисление угла между произведениями собственных векторов

```
product_vector2 = cov_matrix @ product_vector
product_vector2 /= norm(product_vector2)
cosine_similarity = product_vector @ product_vector2
angle = np.degrees(np.arccos(cosine_similarity))
print(f"The angle between vectors equals {angle:.2f} degrees")

The angle between vectors equals 0.00 degrees
```

Оба вектора являются единичными. Как говорилось в главе 13, скалярное произведение двух единичных векторов равно косинусу их угла

Получение арккосинуса от косинусного коэффициента дает угол между векторами

Произведение матрицы и ее собственного вектора сохраняет направление этого вектора. Однако в большинстве случаев оно изменяет абсолютную величину собственного вектора. К примеру, `first-pc` является собственным вектором с абсолютной величиной 1. Умножение `first_pc` на ковариационную матрицу увеличит эту величину в *x* раз. Давайте выведем фактическое изменение абсолютной величины, выполнив `norm(cov_matrix @ first_pc)` (листинг 14.43).

Листинг 14.43. Оценка изменения абсолютной величины

```
new_magnitude = norm(cov_matrix @ first_pc)
print("Multiplication has stretched the first principal component by "
      f"approximately {new_magnitude:.1f} units.")

Multiplication has stretched the first principal component by approximately
541.8 units
```

Умножение растянуло `first_pc` на 541,8 единицы вдоль первого основного направления. Значит, `cov_matrix @ first_pc` равно `541.8 * first_pc`. При наличии матрицы `m` и ее собственного вектора `eigen_vec` произведение `m` и `eigen_vec` всегда будет равно `v * eigen_vec`, где `v` — это численное значение, формально называемое собственным значением. Собственное значение собственного вектора `first_pc` равно приблизительно 541. Оно может показаться вам знакомым, так как мы его уже видели — ранее в этой главе выводили максимизированную дисперсию по оси *X*, которая равнялась приблизительно 541. Значит, собственное значение равно дисперсии вдоль первого основного направления. Убедиться в этом можно, вызвав `(centered_data @ first_pc).var()` (листинг 14.44).

Листинг 14.44. Сравнение собственного значения с дисперсией

```
variance = (centered_data.T @ first_pc).var()
direction1_var = projections[0].var()
assert round(variance, 10) == round(direction1_var, 10)
print("The variance along the first principal direction is approximately"
       f" {variance:.1f}")

The variance along the first principal direction is approximately 541.8
```

Подытожим наши наблюдения.

- Первая главная компонента — это собственный вектор ковариационной матрицы.
- Связанное с этим вектором собственное значение равно дисперсии вдоль первого основного направления.

И эти результаты не случайны, так как математики доказали следующее.

- Главные компоненты набора данных равны нормализованным собственным векторам ковариационной матрицы этих данных.
- Дисперсия вдоль основного направления равна собственному значению связанной с ним главной компоненты.

Следовательно, чтобы найти первую главную компоненту, достаточно выполнить следующее.

1. Вычислить ковариационную матрицу.
2. Найти собственный вектор этой матрицы с наибольшим собственным значением. Этот вектор будет соответствовать направлению с максимальным охватом дисперсии.

Собственный вектор с наибольшим собственным значением можно извлечь при помощи простого итерационного алгоритма, называемого *степенным методом*.

КЛЮЧЕВАЯ ТЕРМИНОЛОГИЯ

- *Ковариационная матрица* — `m @ m.T / m.shape[1]`, где `m` — матрица со средним, равным нулю. Диагональ этой матрицы равна дисперсии вдоль каждой оси `m`.
- *Собственный вектор* — особый вид вектора, ассоциированный с матрицей. Если `m` — это матрица с собственным вектором `eigen_vec`, тогда `m@eigen_vec` указывает в том же направлении, что и `eigen_vec`. Кроме того, если `m` — это ковариационная матрица, то `eigen_vec` — это главная компонента.
- *Собственное значение* — численное значение, ассоциируемое с собственным вектором. Если `m` — это матрица с собственным вектором `eigen_vec`, тогда `m@eigen_vec` растягивает этот вектор на `eigenvalue` единиц. Следовательно, собственное значение равно `norm(m@eigen_vec)/norm(eigen_vec)`. Собственное значение главной компоненты равно дисперсии, охватываемой этой компонентой.

14.4.1. Извлечение собственных векторов с помощью степенного метода

Наша цель — получить собственные векторы `cov_matrix`. Процедура будет простой. Мы начнем с генерации случайного единичного вектора `random-vector` (листинг 14.45).

Листинг 14.45. Генерация случайного единичного вектора

```
np.random.seed(0)
random_vector = np.random.random(size=2)
random_vector /= norm(random_vector)
```

Далее вычислим `cov_matrix @ random_vector` (листинг 14.46). Это произведение матрицы и вектора поворачивает и растягивает наш случайный вектор. Итоговый вектор мы нормализуем так, чтобы его абсолютная величина была сопоставима с `random_vector`, после чего построим график нового вектора и исходного случайного (рис. 14.20). Ожидается, что эти два вектора будут указывать в разных направлениях.

Листинг 14.46. Получение произведения cov_matrix и random_vector

```
product_vector = cov_matrix @ random_vector
product_vector /= norm(product_vector)

plt.plot([0, random_vector[0]], [0, random_vector[1]],
          label='Random Vector')
plt.plot([0, product_vector[0]], [0, product_vector[1]], linestyle='--',
         c='k', label='Normalized Product Vector')

plt.legend()
plt.axis('equal')
plt.show()
```

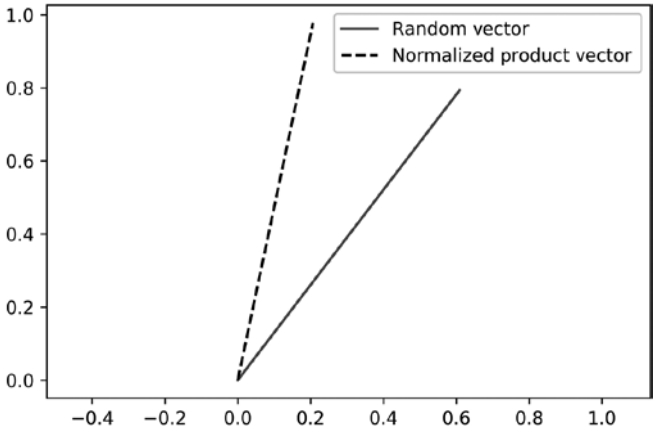


Рис. 14.20. График random_vector вместе с произведением cov_matrix и random_vector. Два этих вектора указывают в разных направлениях

У этих векторов нет ничего общего. Посмотрим, что произойдет при повторении предыдущего шага, выполнив `cov_matrix @ product_vector` (листинг 14.47). Затем нормализуем этот дополнительный вектор и нанесем на график вместе с ранее построенным `product_vector` (рис. 14.21).

Листинг 14.47. Получение произведения cov_matrix и product_vector

```
product_vector2 = cov_matrix @ product_vector
product_vector2 /= norm(product_vector2)

plt.plot([0, product_vector[0]], [0, product_vector[1]], linestyle='--',
         c='k', label='Normalized Product Vector')
plt.plot([0, product_vector2[0]], [0, product_vector2[1]], linestyle=':',
         c='r', label='Normalized Product Vector2')
plt.legend()
plt.axis('equal')
plt.show()
```

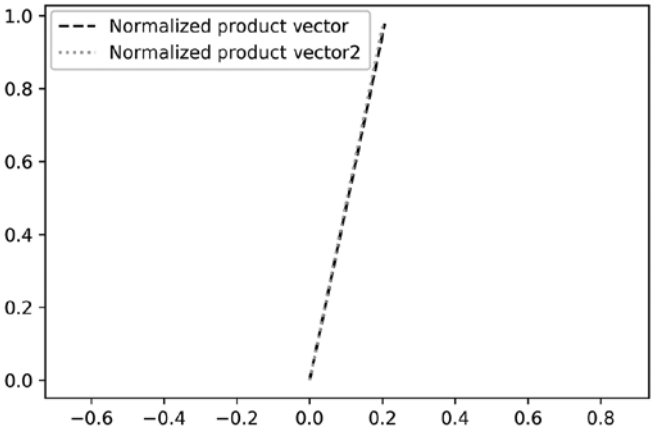


Рис. 14.21. График product_vector вместе с нормализованным произведением cov_matrix и product_vector. Эти два вектора идентичны, то есть мы нашли собственный вектор ковариационной матрицы

Векторы произведений указывают в одном направлении. Следовательно, `product_vector` — это собственный вектор `cov_matrix`. По сути, мы применили степенной метод, который является простым алгоритмом нахождения собственных векторов. С использованием этого алгоритма нам повезло: для обнаружения собственного вектора оказалось достаточно одной операции матричного умножения, хотя чаще требуется несколько итераций.

Степенной метод работает так.

1. Генерирует случайный единичный вектор.
2. Умножает этот вектор на нашу матрицу и нормализует результат. После единичный вектор поворачивается.
3. Итеративно повторяет предыдущий шаг, пока единичный вектор не прекратит вращаться. По определению теперь он является собственным вектором.

Итерация степенным методом гарантированно сходится к собственному вектору, если таковой существует. Как правило, десяти итераций оказывается вполне достаточно для достижения этого результата. Полученный собственный вектор имеет максимальное возможное собственное значение относительно других собственных векторов матрицы.

ПРИМЕЧАНИЕ

Собственные векторы некоторых матриц имеют отрицательные собственные значения. В таких случаях степенной метод возвращает собственный вектор с наибольшим абсолютным собственным значением.

Далее мы определим функцию power_iteration, которая получает матрицу, а возвращает собственный вектор и собственное значение. Ее мы протестируем выполнением power_ iteration(cov_matrix) (листинг 14.48).

Листинг 14.48. Реализация степенного метода

```
np.random.seed(0)
def power_iteration(matrix):
    random_vector = np.random.random(size=matrix.shape[0])
    random_vector = random_vector / norm(random_vector)
    old_rotated_vector = random_vector
    for _ in range(10):
        rotated_vector = matrix @ old_rotated_vector
        rotated_vector = rotated_vector / norm(rotated_vector)
        old_rotated_vector = rotated_vector

    eigenvector = rotated_vector
    eigenvalue = norm(matrix @ eigenvector)
    return eigenvector, eigenvalue

eigenvector, eigenvalue = power_iteration(cov_matrix)
print(f"The extracted eigenvector is {eigenvector}")
print(f"Its eigenvalue is approximately {eigenvalue: .1f}")

The extracted eigenvector is [0.20223994 0.979336 ]
Its eigenvalue is approximately 541.8
```

Функция power_iteration извлекла собственный вектор с собственным значением, равным приблизительно 541. Этот результат соответствует дисперсии вдоль первой основной оси. Получается, что собственный вектор равен первой главной компоненте.

ПРИМЕЧАНИЕ

Вы могли заметить, что извлеченный собственный вектор на рис. 14.21 направлен в сторону положительных значений. При этом первая главная компонента на рис. 14.19 направлена в сторону отрицательных. Как уже говорилось, главную компоненту pc при выполнении PCA можно использовать с -pc взаимозаменяемо — проекция на основные направления от этого не пострадает. Единственным заметным эффектом будет разница в отражении по проецируемым осям, как показано на рис. 14.13.

Наша функция возвращает один собственный вектор с наибольшим собственным значением. Следовательно, power_iteration(cov_matrix) возвращает главную компоненту с наибольшим охватом дисперсии. Как говорилось ранее, вторая главная компонента также является собственным вектором. Ее собственное значение соответствует дисперсии вдоль второго основного направления. Таким образом, эта компонента является собственным вектором со вторым по размеру собственным значением. А как это значение найти? Решение потребует всего нескольких строк

кода. Понять его будет несложно даже тем, кто не знает высшей математики, но мы все же разберем его основные этапы.

Для извлечения второго собственного вектора необходимо исключить из `cov_matrix` все следы первого. Этот процесс известен как понижение порядка матрицы. После понижения второе по размеру собственное значение матрицы становится наибольшим. Чтобы понизить порядок `cov_matrix`, необходимо получить внешнее произведение `eigenvector` с самим собой. Оно вычисляется путем получения парного произведения `eigenvector[i] * eigenvector[j]` для всех возможных значений `i` и `j` (листинг 14.49). Результаты этих парных произведений сохраняются в матрице `M`, где `M[i][j] = eigenvector[i] * eigenvector[j]`. Мы можем вычислить внешнее произведение, используя два вложенных цикла или с помощью NumPy, выполнив `np.outer(eigenvector, eigenvector)`.

ПРИМЕЧАНИЕ

Как правило, вычисляется внешнее произведение двух векторов, v1 и v2. В результате возвращается матрица m, в которой m[i][j] равно v1[i] * v2[j]. Во время понижения порядка матрицы и v1, и v2 оказываются равны eigenvector.

Листинг 14.49. Вычисление внешнего произведения собственного вектора с самим собой

```
outer_product = np.outer(eigenvector, eigenvector)
for i in range(eigenvector.size):
    for j in range(eigenvector.size):
        assert outer_product[i][j] == eigenvector[i] * eigenvector[j]
```

Располагая внешним произведением, можно понизить порядок `cov_matrix`, выполнив `cov_matrix – eigen-value * outer_product` (листинг 14.50). Эта базовая операция создаст матрицу, чей главный собственный вектор будет равен второй главной компоненте.

Листинг 14.50. Понижение порядка ковариационной матрицы

```
deflated_matrix = cov_matrix - eigenvalue * outer_product
```

Выполнение `product_iteration(deflated_matrix)` возвращает собственный вектор, который мы назовем `next_eigenvector`. Исходя из предыдущих рассуждений, будем считать верными следующие доводы:

- `next_eigenvector` равен второй главной компоненте;
- тогда `np.array([eigenvector, next_eigenvector])` равен матрице главных компонент, которую мы называем `components`;
- выполнение `components @ centered_data` проецирует наш набор данных на его основные направления;
- построение проекций должно давать горизонтальный сигаровидный график, похожий на те, что изображены на рис. 14.8 или 14.13.

Далее мы извлекаем `next_eigenvector` и выполняем вышеупомянутые проекции (листинг 14.51). После этого строим эти проекции на графике, подтверждая свои предположения (рис. 14.22).

Листинг 14.51. Извлечение второй главной компоненты из матрицы пониженного порядка

```
np.random.seed(0)
next_eigenvector, _ = power_iteration(deflated_matrix)
components = np.array([eigenvector, next_eigenvector])
projections = components @ centered_data
plt.scatter(projections[0], projections[1])
plt.axhline(0, c='black')
plt.axvline(0, c='black')
plt.axis('equal')
plt.show()
```

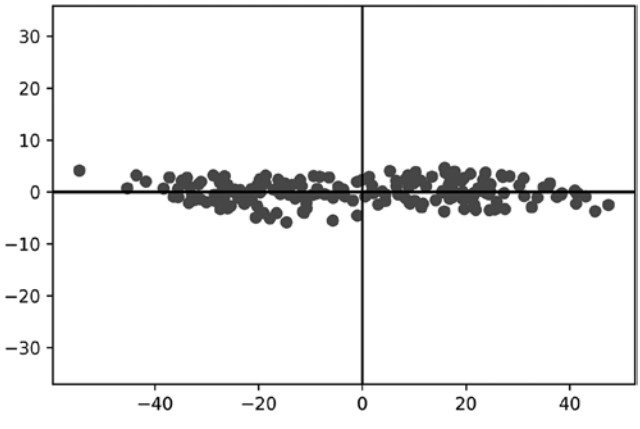


Рис. 14.22. График centered_data, спроецированных на их главные компоненты, вычисленные при помощи степенного метода. Этот график идентичен графику на рис. 14.13, сгенерированному с использованием scikit-learn PCA

ВЫЧИСЛЕНИЯ, СВЯЗАННЫЕ С ПОНИЖЕНИЕМ ПОРЯДКА МАТРИЦЫ В NUMPY

- `np.outer(eigenvector, eigenvector)` — вычисляет внешнее произведение собственного вектора с самим собой. Возвращает матрицу `m`, в которой `m[i][j]` равно `eigenvector[i] * eigenvector[j]`.
- `matrix -= eigenvalue * np.outer(eigenvector, eigenvector)` — понижает порядок матрицы путем удаления из нее всех следов собственного вектора с наибольшим собственным значением. Выполнение `power_iteration(matrix)` возвращает собственный вектор со следующим по размеру собственным значением.

По сути, мы разработали алгоритм для извлечения *K* старших главных компонент матрицы, чьи строки все усредняются в нуль. При получении любой подобной `centered_matrix` этот алгоритм выполняется следующим образом.

1. Вычисляет ковариационную матрицу `centered_matrix` выполнением `centered_matrix @ centered_matrix.T`.
2. Выполняет для полученной ковариационной матрицы `power_iteration`. Эта функция возвращает собственный вектор ковариационной матрицы (`eigenvector`), соответствующий максимально возможному собственному значению (`eigenvalue`). Этот собственный вектор равен первой главной компоненте.
3. Понижает порядок матрицы вычитанием `eigenvalue * np.outer(eigenvector, eigen-vector)`. Выполнение `power_iteration` для матрицы пониженного порядка ведет к извлечению из нее очередной главной компоненты.
4. Повторяет предыдущий шаг еще *K* – 2 раза для извлечения *K* старших главных компонент.

Реализуем этот алгоритм, определив `find_top_principal_components`. Эта функция извлекает *K* старших главных компонент из входной `centered_matrix` (листинг 14.52).

Листинг 14.52. Извлечение K старших главных компонент

Извлекает K старших главных компонент из матрицы, среднее значение строк которой равно нулю. Значение K устанавливается равным 2

```
def find_top_principal_components(centered_matrix, k=2):
    cov_matrix = centered_matrix @ centered_matrix.T
    cov_matrix /= centered_matrix[1].size
    return find_top_eigenvectors(cov_matrix, k=k)

def find_top_eigenvectors(matrix, k=2):
    matrix = matrix.copy()
    eigenvectors = []
    for _ in range(k):
        eigenvector, eigenvalue = power_iteration(matrix)
        eigenvectors.append(eigenvector)
        matrix -= eigenvalue * np.outer(eigenvector, eigenvector)

    return np.array(eigenvectors)
```

Главные компоненты — это просто старшие собственные векторы ковариационной матрицы, ранг которых определяется их собственными значениями. Чтобы подчеркнуть этот смысл, мы определяем отдельную функцию для извлечения K старших собственных векторов любой матрицы

Создает копию матрицы, чтобы можно было понизить разрядность копии, не изменяя оригинала

Мы определили функцию для извлечения *K* старших главных компонент набора данных. Эти компоненты позволят спроецировать данные на их *K* старших основных направлений, которые максимизируют дисперсию данных вдоль *K* осей. В результате оставшиеся оси данных можно будет отбросить, уменьшив количество столбцов координат до *K*. Следовательно, мы имеем возможность уменьшить любой набор данных до *K* измерений.

По сути, теперь у нас есть возможность выполнять PCA с нуля без использования scikit-learn. Мы можем применить этот алгоритм для уменьшения до *K* измерений *N*-мерного набора данных, где *N* — это количество столбцов входной матрицы данных. Для выполнения PCA необходимо сделать следующие шаги.

1. Вычислить среднее вдоль каждой из осей набора данных.
2. Вычесть это среднее из каждой оси, тем самым центрировав набор данных в начале координат.
3. Извлечь *K* старших главных компонент центрированного набора данных с помощью функции `find_ top_principal_components`.
4. Получить матричное произведение между главными компонентами и центрированным набором данных.

Далее мы реализуем эти шаги в одной функции, которую назовем `reduce_dimensions` (листинг 14.53). Почему бы не называть ее `pca`? Что ж, хотя первые два шага PCA и требуют централизации наших данных, вскоре мы узнаем, что уменьшить размерность можно и без централизации. В связи с этим передаем в функцию необязательный параметр `centralize_data` и устанавливаем для него значение `True`, чтобы функция выполнила PCA с установками по умолчанию.

Листинг 14.53. Определение функции reduce_dimensions

```
def reduce_dimensions(data, k=2, centralize_data=True):
    data = data.T.copy()
    if centralize_data:
        for i in range(data.shape[0]):
            data[i] -= data[i].mean()
    principal_components = find_top_principal_components(data)
    return (principal_components @ data).T
```

Функция получает матрицу данных, столбцы которой соответствуют осям. Это согласуется с входной ориентацией scikit-learn метода fit_transform. Затем она уменьшает матрицу с N до K столбцов

Данные транспонируются так, чтобы сохранить согласованность с ожидаемыми входными данными для find_principal_components

Опционально централизует данные, вычитая их среднее так, чтобы новое среднее равнялось 0

Теперь протестируем `reduce_dimensions`, применив ее к ранее проанализированным данным `flower_measurements`. Их мы сократим до двух измерений с помощью нашей собственной реализации PCA, после чего визуализируем результат (листинг 14.54; рис. 14.23). График должен согласовываться с графиком, изображенным на рис. 14.18.

Листинг 14.54. Уменьшение данных о цветах до двух измерений с помощью кастомной реализации PCA

```
np.random.seed(0)
dim_reduced_data = reduce_dimensions(flower_measurements)
visualize_flower_data(dim_reduced_data)
```

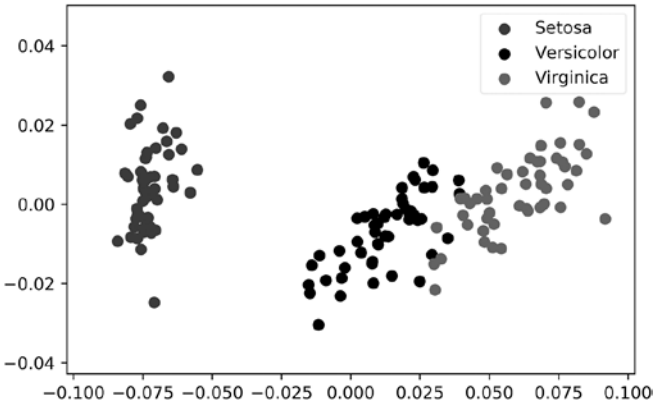


Рис. 14.23. Четырехмерные нормализованные размеры цветов, уменьшенные до двух измерений при помощи кастомной реализации PCA. Этот график идентичен выводу PCA, сгенерированному scikit-learn

Наш график в точности совпадает с выводом scikit-learn PCA. Мы воспроизвели реализацию scikit-learn, но с одним значительным отличием: в нашей функции централизация необязательна. И это окажется нам на руку. Как уже говорилось, мы не можем достоверно выполнить централизацию нормализованных данных. Кроме того, наш набор цветочных данных был нормализован для устранения различий в единицах измерения. Следовательно, невозможно достоверно выполнить PCA для `flower_measurements`. Одна из альтернатив — обойти централизацию, передав в функцию `reduce_dimensions` параметр `centralize-data=False`. Это, конечно, нарушит многие допущения алгоритма PCA, но результат все равно окажется полезным. Что произойдет, если мы уменьшим размерность `flower_measurements` без централизации? Далее мы это выясним, установив `centralize_data` на `False` и отобразив результаты на графике (листинг 14.55; рис. 14.24).

Листинг 14.55. Выполнение reduce_dimensions без централизации

Как говорилось ранее, случайность извлечения собственного вектора может повлиять на ориентацию двухмерного графика. Здесь мы задаем алгоритму начальное число, чтобы ориентация совпадала с другим графиком, который будет показан позже (рис. 14.25)

```
np.random.seed(3)
dim_reduced_data = reduce_dimensions(flower_measurements,
                                     centralize_data=False)
visualize_flower_data(dim_reduced_data)
```

В полученном выводе три вида цветов все так же распределяются по трем кластерам. Более того, ирис по-прежнему пространственно отделен от других видов. Однако в графике есть и изменения. Ирис формирует более плотный кластер, чем в прежнем результате PCA. Это наводит на вопрос: является ли последний график таким же полноценным, как и вывод PCA? Иными словами, представляет ли он по-прежнему 97 % общей дисперсии данных? Проверить это можно,

измерив дисперсию `dim_reduced_data` и разделив ее на общую дисперсию `flower_measurements` (листинг 14.56).

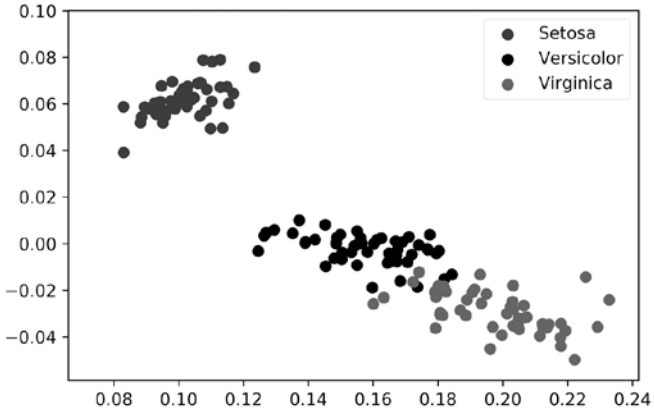


Рис. 14.24. Четырехмерные нормализованные размеры цветов, уменьшенные до двух измерений без централизации. Каждая точка окрашена в соответствии с видом цветка. Все три вида по-прежнему делятся на три кластера. Однако этот график больше не похож на вывод PCA

Листинг 14.56. Проверка дисперсии данных, уменьшенных без централизации

```
variances = [sum(data[:,i].var() for i in range(data.shape[1]))
             for data in [dim_reduced_data, flower_measurements]]
dim_reduced_var, total_var = variances
percent_coverege = 100 * dim_reduced_var / total_var
print(f"Our plot covers {percent_coverege:.2f}% of the total variance")
```

```
Our plot covers 97.29% of the total variance
```

Охват двухмерной дисперсии сохранен. Несмотря на то что его значение слегка колеблется, оно составляет примерно 97 %. Получается, что можно уменьшить размерность, не прибегая к централизации. Однако этот прием остается определяющей особенностью PCA, поэтому нашу измененную технику нужно назвать иначе. Официально, как уже говорилось, она называется *сингулярной декомпозицией* (SVD).

ПРЕДУПРЕЖДЕНИЕ

В отличие от PCA, SVD не гарантирует максимизацию дисперсии для каждой оси в уменьшенном выводе. Однако в большинстве реальных случаев она способна уменьшить размерность данных до вполне адекватной степени.

Математические свойства SVD довольно сложны, и их изучение выходит за рамки данной книги. Тем не менее специалисты по информатике могут использовать эти свойства для очень эффективного выполнения SVD, и найденные ими оптимизации были встроены в scikit-learn. В следующем разделе мы задействуем оптимизированную реализацию SVD из этой библиотеки.

14.5. ЭФФЕКТИВНОЕ УМЕНЬШЕНИЕ РАЗМЕРНОСТИ С ПОМОЩЬЮ SVD И SCIKIT-LEARN

Scikit-learn содержит класс для уменьшения размерности, `TruncatedSVD`, предназначенный для оптимального выполнения SVD. Для начала импортируем этот класс из `sklearn.decomposition` (листинг 14.57).

Листинг 14.57. Импорт TruncatedSVD из scikit-learn

```
from sklearn.decomposition import TruncatedSVD
```

Применить `TruncatedSVD` к нашим данным `flower_measurements` легко. Сначала нужно выполнить `TruncatedSVD(n_components=2)` для создания объекта `svd_object`, способного уменьшить данные до двух измерений. Затем можно будет задействовать SVD, выполнив `svd_object.fit_predict(flower_measurements)`. Вызов этого метода вернет матрицу `svd_transformed_data`. Далее мы применим `TruncatedSVD` и отразим результаты на графике (см. рис. 14.25). Он должен быть похож на кастомный вывод SVD, представленный на рис. 14.24.

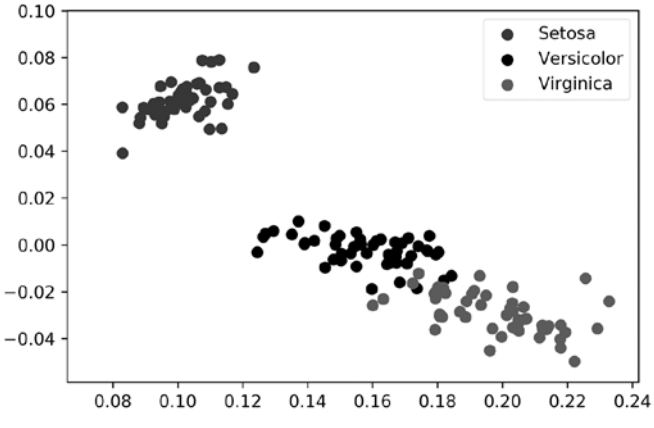


Рис. 14.25. Четырехмерные нормализованные размеры цветов, уменьшенные до двух измерений при помощи реализации SVD из scikit-learn. Этот результат идентичен рис. 14.24, сгенерированному с помощью нашей собственной реализации SVD

ПРИМЕЧАНИЕ

В отличие от класса PCA реализации TruncatedSVD в scikit-learn требуется входной параметр n_components (листинг 14.58). По умолчанию установлено значение этого параметра 2.

Листинг 14.58. Выполнение SVD с помощью scikit-learn

```
svd_object = TruncatedSVD(n_components=2)
svd_transformed_data = svd_object.fit_transform(flower_measurements)
visualize_flower_data(svd_transformed_data)
```

Неудивительно, что результат scikit-learn совпадает с результатом нашей собственной реализации SVD. Алгоритм scikit-learn быстрее и эффективнее в плане потребления памяти, но его результат от нашего не отличается.

ПРИМЕЧАНИЕ

Результат не будет отличаться при уменьшении количества измерений. Однако при его увеличении наша реализация станет менее точной, поскольку в вычисления собственных векторов просочатся мелкие ошибки. Причем с каждым вычислением собственного вектора они будут возрастать. Scikit-learn же, в свою очередь, компенсирует данные ошибки с помощью математических техник.

Можно дополнительно проверить совпадение выводов, сравнив охватываемую ими дисперсию. Наш `svd_object` содержит атрибут `explained_variance_ratio_`, в котором хранится массив дробных величин дисперсии, охватываемой каждым измерением после уменьшения. Суммирование `100 * explained_variance_ratio_` должно вернуть процент общей дисперсии, охватываемой в нашем двухмерном графике. На основе проведенного анализа мы ожидаем, что вывод будет аппроксимироваться к 97,29 %. Проверим это (листинг 14.59).

Листинг 14.59. Извлечение дисперсии из вывода scikit-learn SVD

Данный атрибут — это массив NumPy, содержащий дробный охват для каждой оси. Умножение на 100 преобразует эти дроби в проценты

```
percent_variance_coverages = 100 * svd_object.explained_variance_ratio_
x_axis_coverage, y_axis_coverage = percent_variance_coverages
total_2d_coverage = x_axis_coverage + y_axis_coverage
print(f"Our Scikit-Learn SVD output covers {total_2d_coverage:.2f}% of "
       "the total variance")
```

Каждый i-й элемент массива соответствует охвату дисперсии i-й осью

```
Our Scikit-Learn SVD output covers 97.29% of the total variance
```

ТИПИЧНЫЕ МЕТОДЫ SVD В SCIKIT-LEARN

- `svd_object = TruncatedSVD(n_components=K)` — создает объект SVD, способный уменьшить входные данные до *K* измерений.
- `svd_tranformed_data = svd_object.fit_transform(data)` — выполняет SVD для входных данных, используя инициализированный объект `TruncatedSVD`. Метод `fit_transform` предполагает, что столбцы матрицы `data` соответствуют пространственным осям. Результаты, уменьшенные в размере, сохраняются в матрице `svd_transformed_data`.
- `svd_object.explained_variance_ratio_` — возвращает дробный показатель охвата дисперсии, ассоциируемый с каждой уменьшенной в размере осью скорректированного объекта `TruncatedSVD`.

Оптимизированная реализация SVD из scikit-learn может уменьшать данные с тысяч измерений до всего нескольких сотен или даже десятков. Уменьшенные данные можно более эффективно хранить, передавать и обрабатывать предиктивными алгоритмами. Многие реальные задачи по анализу данных требуют применения SVD для их предварительного уменьшения. Диапазон применения сингулярной декомпозиции — от сжатия изображений и удаления акустического шума до обработки естественного языка. NLP, в частности, зависит от этого алгоритма ввиду характерной для текстовых данных раздутости. Как говорилось в предыдущей главе, реальные документы формируют огромные матрицы, количество столбцов в которых оказывается чрезмерно большим. Мы не можем умножить подобные матрицы эффективно, а значит, не можем вычислить сходство отраженных в них текстов. К счастью, SVD делает такие матрицы документов более управляемыми. Сингулярная декомпозиция позволяет уменьшать число их столбцов, сохраняя при этом большую часть дисперсии, в результате чего мы можем оперативно вычислять сходство между огромными текстами. Затем сходство текстов можно использовать для кластеризации входных документов.

В следующей главе мы наконец займемся анализом больших наборов данных документов. При этом научимся очищать и кластеризовать эти наборы данных, попутно визуализируя результаты. В этом анализе SVD проявит себя как фундаментальная техника, без которой невозможно обойтись.

РЕЗЮМЕ

- Уменьшение размерности набора данных может упрощать определенные задачи, например кластеризацию.
- Двухмерный набор данных можно уменьшить до одного измерения, поворачивая данные относительно начала координат до тех пор, пока их точки не окажутся поблизости от оси *X*. Этот прием максимизирует распределение данных вдоль этой оси, тем самым позволяя удалить ось *Y*. Однако для поворота данных сначала нужно централизовать их набор, чтобы его средние координаты лежали в начале координат.
- Поворот данных в сторону оси *X* аналогичен повороту оси *X* в сторону *первого основного направления*, представляющего линейное направление, вдоль которого дисперсия данных максимальна. *Второе основное направление* перпендикулярно первому. В двухмерном наборе данных оно представляет дисперсию, не охваченную первым основным направлением.
- Уменьшить размерность можно с помощью *анализа главных компонент* (PCA). PCA находит основные направления набора данных и представляет их как использующие единичные векторы, называемые *главными компонентами*. Перемножение матрицы централизованных данных и главных компонент ведет к замене стандартных осей данных основными направлениями. Это называется

проекцией. Проецируя данные на основные направления, мы максимизируем их дисперсию вдоль одних осей и минимизируем вдоль других. Оси с минимизированной дисперсией можно в итоге удалить.

- Главные компоненты можно извлечь путем вычисления *ковариационной матрицы* — матричного произведения центрированного набора данных с самим собой, разделенного на размер этого набора данных. Диагональ полученной матрицы представляет значения дисперсии вдоль осей.
- Главные компоненты — это *собственные векторы* ковариационной матрицы. Таким образом, по определению нормализованное произведение этой матрицы и каждой главной компоненты равно этой главной компоненте.
- Извлечь старший собственный вектор матрицы можно с помощью *степенного метода*. Этот итерационный алгоритм состоит из повторяющихся умножения и нормализации вектора с матрицей. Применение степенного метода к ковариационной матрице возвращает первую главную компоненту.
- Используя *понижение порядка матрицы*, мы можем устранить все следы собственного вектора. Понижение порядка ковариационной матрицы и повторное применение степенного метода ведет к возвращению второй главной компоненты. Итеративное повторение этого процесса возвращает все главные компоненты.
- PCA чувствителен к единицам измерения. Нормализация входных данных снижает чувствительность, но вследствие этого нормализованные данные приближаются к своему среднему. Это становится проблемой, поскольку PCA требует вычитания среднего из каждого значения оси для централизации данных, а выполнение вычитания между близкими величинами создает погрешности из-за плавающей точки.
- Избежать этих погрешностей можно, отказавшись от централизации данных перед уменьшением их размерности. Итоговый вывод будет охватывать дисперсию данных в достаточной степени. Эта измененная техника называется *сингулярной декомпозицией* (SVD).

15 NLP-анализ больших текстовых наборов данных

В этой главе

- ✓ Векторизация текстов с помощью scikit-learn.
- ✓ Уменьшение размерности векторизованных текстовых данных.
- ✓ Кластеризация больших текстовых наборов данных.
- ✓ Визуализация кластеров текстов
- ✓ Одновременное отображение нескольких визуализаций.

Ранее, обсуждая техники обработки естественного языка (NLP), мы рассматривали упрощенные примеры и небольшие наборы данных. В данной же главе применим NLP к большой коллекции реальных текстов. Учитывая представленные к этому моменту приемы, такой анализ может показаться простым. Предположим, что мы проводим исследование рынка среди нескольких дискуссионных форумов. На каждом из них сотни пользователей обсуждают определенную тему, к примеру политику, моду, технологии или машины. Нам нужно автоматически извлечь все темы обсуждения на основе содержимого диалогов. Отталкиваясь от этих тем, мы спланируем маркетинговую кампанию, нацеленную на пользователей в соответствии с их онлайн-интересами.

Как же кластеризовать беседы пользователей по темам? Один из подходов состоит в следующем.

1. Преобразовать тексты всех бесед в матрицу количеств слов, используя приемы из главы 13.
2. Уменьшить размерность матрицы с помощью SVD. Это позволит эффективно вычислить сходство между всеми парами текстов с помощью матричного умножения.
3. Использовать полученную матрицу сходства текстов для кластеризации бесед по темам.
4. Изучить кластеры тем и определить среди них полезные для нашей кампании.

Естественно, в реальной жизни анализ окажется не столь простым, как может показаться изначально. Здесь все еще остаются вопросы, на которые пока нет ответов. Как нам эффективно изучить кластеры тем, не прочитывая все относящиеся к ним тексты по очереди? Кроме того, какой алгоритм из представленных в главе 10 нужно взять для кластеризации бесед?

И даже на уровне попарного сравнения текстов мы сталкиваемся с определенными вопросами. Как обработать типичные неинформативные слова вроде *the*, *it* и *they*? Нужно ли накладывать на них штраф? Игнорировать? Или полностью отфильтровать? А как быть с распространенными специфичными для конкретного корпуса словами, такими как названия сайтов, где расположены форумы?

Ответы на все эти вопросы проще всего понять, изучив набор данных форума, содержащего тысячи текстов. И в scikit-learn есть один такой реальный набор данных. В текущей главе мы скачаем, изучим и кластеризуем его. В ходе анализа нам окажут бесценную помощь внешние библиотеки Python, а именно уже упомянутая scikit-learn и NumPy.

15.1. СКАЧИВАНИЕ ДИСКУССИЙ ОНЛАЙН-ФОРУМОВ С ПОМОЩЬЮ SCIKIT-LEARN

Scikit-learn предоставляет нам данные из Usenet — грамотно организованной онлайн-коллекции дискуссионных форумов. Эти форумы называются *новостными группами*. Каждая такая группа относится к отдельной теме обсуждения, которая кратко отражена в ее названии. Пользователи новостных групп общаются посредством размещения сообщений. Публикуемые ими посты не имеют ограничения по длине, поэтому некоторые оказываются довольно длинными. Разнообразие и различия в длине постов позволят нам расширить навыки NLP. В качестве тренировки scikit-learn предоставляет доступ более чем к 10 000 размещенных в сети сообщений. Эти новостные группы можно скачать, импортировав `fetch_20newsgroups` из `sklearn.datasets`. Вызов `fetch_20newsgroups()` вернет объект `newsgroups`,

содержащий текстовые данные. Кроме того, дополнительная передача в вызов этой функции параметра `remove=('headers', 'footers')` приведет к удалению из текста лишней информации. (Удаленные метаданные не имеют отношения к смыслу содержимого поста.) Код листинга 15.1 скачивает данные новостной группы, одновременно отфильтровывая лишнюю информацию.

ВНИМАНИЕ

Скачиваемый набор данных новостных групп очень большой. В связи с этим scikit-learn предварительно его не упаковывает. Выполнение fetch_20newsgroups даст scikit-learn команду скачать и сохранить этот набор данных на локальной машине, то есть вам потребуется активное интернет-соединение. Все последующие вызовы fetch_20newsgroups приведут к скачиванию набора данных локально, уже не требуя Интернета.

Листинг 15.1. Скачивание набора данных новостных групп

```
from sklearn.datasets import fetch_20newsgroups
newsgroups = fetch_20newsgroups(remove=('headers', 'footers'))
```

Объект `newsgroups` содержит посты из 20 разных новостных групп. Как уже говорилось, тема дискуссии каждой группы обозначена в ее названии. Просмотреть эти названия можно с помощью `newsgroups.target_names` (листинг 15.2).

Листинг 15.2. Вывод названий всех 20 новостных групп

```
print(newsgroups.target_names)
```

```
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x',
 'misc.forsale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball',
 'rec.sport.hockey', 'sci.crypt', 'sci.electronics', 'sci.med', 'sci.space',
 'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mideast',
 'talk.politics.misc', 'talk.religion.misc']
```

Категории этих новостных групп сильно разнятся: тут и изучение космоса (`sci.space`), и машины (`rec.cars`), и электроника (`sci.electronics`). Некоторые из этих категорий очень обширны. К примеру, политика (`talk.politics.misc`) может охватывать различные политические темы. Иные категории, напротив, очень узки: скажем, `comp.sys.mac.hardware` касается только оборудования для Mac, а `comp.sys.ibm.pc_hardware` относится только к оборудованию для ПК. В смысловом плане эти две новостные группы очень схожи — единственным различием является то, относится обсуждаемое оборудование к Mac или к ПК под Windows. Иногда различия между категориями бывают едва уловимы. Границы между темами текстов текучи и не обязательно устанавливаются жестко. И это нужно будет иметь в виду позднее, когда мы кластеризуем посты новостных групп.

Теперь перейдем к реальным текстам, которые хранятся в виде списка в атрибуте `newsgroups.data`. К примеру, `newsgroups.data[0]` содержит текст первого сохраненного поста новостных групп. Выведем его (листинг 15.3).

Листинг 15.3. Вывод первого поста новостных групп

```
print(newsgroups.data[0])
```

```
I was wondering if anyone out there could enlighten me on this car I saw the other
day. It was a 2-door sports car, looked to be from the late 60s/ early 70s. It was
called a Bricklin. The doors were really small.
In addition, the front bumper was separate from the rest of the body. This is all I
know. If anyone can tellme a model name, engine specs, years
of production, where this car is made, history, or whatever info you have on this
funky looking car, please e-mail.
```

Это пост о машине, и, вероятно, он был опубликован в новостной группе `rec.autos`. Убедиться в этом можно с помощью вывода `newsgroups.target_names[newsgroups.target[0]]` (листинг 15.4).

ПРИМЕЧАНИЕ

newsgroups.target[i] возвращает индекс названия новостной группы, связанной с i-м документом.

Листинг 15.4. Вывод названия новостной группы по индексу 0

```
origin = newsgroups.target_names[newsgroups.target[0]]
print(f"The post at index 0 first appeared in the '{origin}' group")
```

```
The post at index 0 first appeared in the 'rec.autos' group
```

Как мы и прогнозировали, связанный с машиной пост относится к группе обсуждения машин. Присутствия нескольких ключевых слов, таких как *car*, *bumper* и *engine*, оказалось достаточно для выявления его принадлежности. Естественно, это лишь один пост из множества. Разбить по категориям оставшиеся может быть не так просто.

Давайте повнимательнее взглянем на наш набор данных новостных групп, оценив его размер (листинг 15.5).

Листинг 15.5. Подсчет количества постов в новостных группах

```
dataset_size = len(newsgroups.data)
print(f"Our dataset contains {dataset_size} newsgroup posts")
```

```
Our dataset contains 11314 newsgroup posts
```

В наборе данных содержится 11 000 постов. Наша цель — кластеризовать их по темам, но данная задача в таких масштабах потребует вычислительной эффективности. То есть нам нужно эффективно вычислить сходство постов, представив текстовые данные в виде матрицы. Для этого потребуется трансформировать каждый пост в вектор частотности термина (вектор TF). Как говорилось в главе 13, индексы вектора TF сопоставляются с количеством слов в документе. Ранее мы вычисляли эти векторизованные количества слов с помощью кастомных функций. Теперь же используем для этого scikit-learn.

15.2. ВЕКТОРИЗАЦИЯ ДОКУМЕНТОВ С ПОМОЩЬЮ SCIKIT-LEARN

Scikit-learn предоставляет встроенный класс для преобразования входных текстов в векторы TF — `CountVectorizer`. Его инициализация ведет к созданию объекта `vectorizer`, способного векторизовать наши тексты. Код листинга 15.6 импортирует `CountVectorizer` из `sklearn.feature_extraction.text` и инициализирует этот класс.

Листинг 15.6. Инициализация объекта CountVectorizer

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
```

Теперь можно векторизовать тексты, сохраненные в списке `newsgroups.data`. Для этого достаточно выполнить `vectorizer.fit_transform(newsgroups.data)`. Вызов этого метода вернет матрицу TF, соответствующую векторизованным постам новостных групп. Напомню, что в матрице TF хранятся количества слов (в столбцах) по всем текстам (в строках). Давайте векторизуем эти посты и выведем полученную матрицу TF (листинг 15.7).

Листинг 15.7. Вычисление матрицы TF с помощью scikit-learn

```
tf_matrix = vectorizer.fit_transform(newsgroups.data)
print(tf_matrix)
```

```
(0, 108644)    4
(0, 110106)    1
(0, 57577)     2
(0, 24398)     2
(0, 79534)     1
(0, 100942)    1
(0, 37154)     1
(0, 45141)     1
(0, 70570)     1
(0, 78701)     2
(0, 101084)    4
(0, 32499)     4
(0, 92157)     1
(0, 100827)    6
(0, 79461)     1
(0, 39275)     1
(0, 60326)     2
(0, 42332)     1
(0, 96432)     1
(0, 67137)     1
(0, 101732)    1
(0, 27703)     1
(0, 49871)     2
(0, 65338)     1
(0, 14106)     1
```

```
:    :
(11313, 55901)     1
(11313, 93448)     1
(11313, 97535)     1
(11313, 93393)     1
(11313, 109366)    1
(11313, 102215)    1
(11313, 29148)     1
(11313, 26901)     1
(11313, 94401)     1
(11313, 89686)     1
(11313, 80827)     1
(11313, 72219)     1
(11313, 32984)     1
(11313, 82912)     1
(11313, 99934)     1
(11313, 96505)     1
(11313, 72102)     1
(11313, 32981)     1
(11313, 82692)     1
(11313, 101854)    1
(11313, 66399)     1
(11313, 63405)     1
(11313, 61366)     1
(11313, 7462)      1
(11313, 109600)    1
```

Полученная матрица `tf_matrix` не является массивом NumPy. Что же за структура данных перед нами? Проверить это можно с помощью вывода `type(tf_matrix)` (листинг 15.8).

Листинг 15.8. Проверка типа данных tf_matrix

```
print(type(tf_matrix))
```

```
<class 'scipy.sparse.csr.csr_matrix'>
```

Как оказалось, эта матрица является объектом SciPy, называемым `csr_matrix`. *CSR* расшифровывается как *сжатая разреженная матрица* и представляет собой формат хранения сжатых матриц, состоящих в основном из нулей. Такие преимущественно пустые матрицы называются *разреженными*. Их можно уменьшить за счет сохранения только ненулевых элементов. Это позволяет эффективнее использовать память и ускорить вычисления. Крупномасштабные текстовые матрицы обычно очень разрежены, поскольку один документ, как правило, содержит лишь небольшой процент общего словаря. Таким образом, scikit-learn автоматически преобразует векторизованный текст в формат CSR. Выполняется это преобразование с помощью класса `csr_matrix`, импортируемого из SciPy.

Такое взаимодействие между различными внешними библиотеками аналитики данных хотя и полезно, но несколько запутывает. В частности, новичкам может

быть непросто понять, в чем заключаются различия между массивом NumPy и CSR-матрицей SciPy. Дело в том, что массивы и CSR-матрицы обладают рядом общих атрибутов. Кроме того, и те и другие совместимы с некоторыми функциями NumPy, но не со всеми. Чтобы уменьшить эту путаницу, мы преобразуем `tf_matrix` в двухмерный массив NumPy. Бóльшая часть последующего анализа будет выполняться для этого массива, но иногда будем сравнивать использование массива с применением CSR-матрицы. Это позволит нам лучше понять, в чем сходство этих двух представлений матрицы и каковы различия между ними. Код листинга 15.9 преобразует `tf_matrix` в массив NumPy путем выполнения `tf_matrix.toarray()`, после чего выводит полученный результат.

ВНИМАНИЕ

Операция преобразования очень ресурсозатратна и требует почти 10 Гбайт памяти. Если на вашей машине объем памяти ограничен, рекомендуем выполнить этот код в облаке, используя Google Colaboratory (Colab) — бесплатную облачную среду Jupyter Notebook с 12 Гбайт доступной памяти. Google предоставляет для Colab подробную инструкцию, в которой описывается все необходимое для начала применения этой среды: https://colab.research.google.com/notebooks/welcome.ipynb.

Листинг 15.9. Преобразование CSR-матрицы в массив NumPy

```
tf_np_matrix = tf_matrix.toarray()
print(tf_np_matrix)

[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

Полученная матрица — это двухмерный массив NumPy. Все показанные в его превью элементы являются нулями, что подтверждает высокую степень разреженности. Каждый такой элемент соответствует количеству слов в посте. Строки матрицы представляют посты, а столбцы — отдельные слова. Таким образом, общее количество слов равно объему словаря нашего набора данных. Доступ к этому количеству мы получаем с помощью атрибута `shape`. Это общий атрибут для CSR-матрицы и массива NumPy. Далее используем `tf_np_matrix.shape` для вывода размера словаря (листинг 15.10).

Листинг 15.10. Проверка размера словаря

```
assert tf_np_matrix.shape == tf_matrix.shape
num_posts, vocabulary_size = tf_np_matrix.shape
print(f"Our collection of {num_posts} newsgroup posts contain a total of "
      f"{vocabulary_size} unique words")

Our collection of 11314 newsgroup posts contain a total of 114751 unique words
```

Наши данные содержат 114 751 уникальное слово. Однако в большинстве постов их содержится всего несколько сотен. Измерить это уникальное количество слов в посте по индексу `i` можно, подсчитав ненулевые элементы в строке `tf_np_matrix[i]`. Проще всего это сделать с помощью NumPy. Данная библиотека позволяет получить все ненулевые индексы вектора в `tf_np_matrix[i]` — нужно лишь передать интересующий вектор в функцию `np.flatnonzero`. В листинге 15.11 мы подсчитываем и выводим количество ненулевых индексов в посте про машину, соответствующем `newsgroups.data[0]`.

Листинг 15.11. Подсчет уникальных слов в посте про машину

Это равнозначно выполнению np.nonzero(tf_vector)[0]. Функция np.nonzero обобщает вычисление ненулевых индексов в x-мерном массиве. Она возвращает кортеж длиной x, в котором каждый i-й элемент представляет количество ненулевых индексов вдоль i-го измерения. Таким образом, получая одномерный массив tf_vector, функция np.nonzero возвращает кортеж в виде (non_zero_indices)

```
import numpy as np
tf_vector = tf_np_matrix[0]
non_zero_indices = np.flatnonzero(tf_vector)
num_unique_words = non_zero_indices.size
print(f"The newsgroup in row 0 contains {num_unique_words} unique words.")
print("The actual word counts map to the following column indices:\n")
print(non_zero_indices)

The newsgroup in row 0 contains 64 unique words.
The actual word-counts map to the following column indices:
[ 14106  15549  22088  23323  24398  27703  29357  30093  30629  32194
  32305  32499  37154  39275  42332  42333  43643  45089  45141  49871
  49881  50165  54442  55453  57577  58321  58842  60116  60326  64083
  65338  67137  67140  68931  69080  70570  72915  75280  78264  78701
  79055  79461  79534  82759  84398  87690  89161  92157  93304  95225
  96145  96432 100406 100827 100942 101084 101732 108644 109086 109254
 109294 110106 112936 113262]
```

Первый пост новостных групп содержит 64 уникальных слова. Что это за слова? Чтобы это выяснить, необходимо сопоставить индексы вектора TF и значения слов. Сопоставление можно сгенерировать вызовом `vectorizer.get_feature_names()`. Он вернет список слов, который мы назовем `words`. Каждый индекс `i` соответствует *i*-му слову в списке. Таким образом, выполнение `[words[i] for i in non_zero_indices]` вернет все уникальные слова в посте (листинг 15.12).

ПРИМЕЧАНИЕ

Эти слова можно получить и с помощью вызова vectorizer.inverse_transform(tf_vector). Метод inverse_transform вернет все слова, связанные с переданным ему вектором TF.

Листинг 15.12. Вывод уникальных слов из поста про машину

```
words = vectorizer.get_feature_names()
unique_words = [words[i] for i in non_zero_indices]
print(unique_words)

['60s', '70s', 'addition', 'all', 'anyone', 'be', 'body', 'bricklin',
'bumper', 'called', 'can', 'car', 'could', 'day', 'door', 'doors',
```

```
'early', 'engine', 'enlighten', 'from', 'front', 'funky', 'have',
'history', 'if', 'in', 'info', 'is', 'it', 'know', 'late', 'looked',
'looking', 'made', 'mail', 'me', 'model', 'name', 'of', 'on', 'or',
'other', 'out', 'please', 'production', 'really', 'rest', 'saw',
'separate', 'small', 'specs', 'sports', 'tellme', 'the', 'there',
'this', 'to', 'was', 'were', 'whatever', 'where', 'wondering', 'years', 'you']
```

Мы вывели все слова из `newsgroups.data[0]`. Естественно, не у всех них число упоминаний одинаковое — некоторые встречаются чаще. Можно предположить, что часто встречающиеся слова более тесно связаны с темой машин. Код листинга 15.13 выводит десять наиболее употребимых слов, а также их количество. Для наглядности представим вывод в виде таблицы Pandas.

ИЗВЛЕЧЕНИЕ НЕНУЛЕВЫХ ЭЛЕМЕНТОВ ОДНОМЕРНЫХ МАССИВОВ NUMPY

- `on_zero_indices = np.flatnonzero(np_vector)` — возвращает ненулевые индексы в одномерном массиве NumPy.
- `non_zero_vector = np_vector[non_zero_indices]` — выбирает ненулевые элементы одномерного массива NumPy (в предположении, что `non_zero_indices` соответствует его ненулевым индексам).

Листинг 15.13. Вывод наиболее частых слов в посте о машине

```
import pandas as pd
data = {'Word': unique_words,
        'Count': tf_vector[non_zero_indices]}

df = pd.DataFrame(data).sort_values('Count', ascending=False)
print(df[:10].to_string(index=False))
```

Упорядочивает таблицу Pandas на основе количества по убыванию

```
Word Count
   the     6
  this     4
   was     4
   car     4
    if     2
    is     2
    it     2
  from     2
    on     2
anyone     2
```

Четыре из 64 слов поста упоминаются не менее четырех раз. Одно из них — *car*, что неудивительно для поста в группе обсуждения машин. А вот остальные три слова никак с машинами не связаны: *the*, *this* и *was* относятся к числу наиболее распространенных слов английского языка. Они не несут в себе сигнала, дифференцирующего пост про машину и пост из другой темы, и являются лишь источником шума, повышая вероятность включения двух несвязанных документов в одну группу.

Специалисты по NLP называют такие слова *стоп-словами*, из векторизованных результатов они исключаются. Стоп-слова обычно удаляют из текста перед векторизацией. Именно поэтому в классе `CountVectorizer` есть встроенная опция для их удаления. Выполнение `CountVectorizer(stop_words='english')` инициализирует векторизатор, настроенный на удаление стоп-слов. Он игнорирует все наиболее распространенные английские слова в тексте.

Далее мы повторно инициализируем векторизатор, теперь уже с учетом удаления стоп-слов (листинг 15.14). После этого еще раз выполним `fit_transform` для повторного вычисления матрицы TF. Количество столбцов в этой матрице будет меньше, чем для ранее вычисленного словаря из 114 751 слова. Мы также еще раз сгенерируем список `words`, на этот раз без типичных стоп-слов вроде *the*, *this*, *of* и *it*.

Листинг 15.14. Удаление стоп-слов во время векторизации

```
vectorizer = CountVectorizer(stop_words='english')
tf_matrix = vectorizer.fit_transform(newsgroups.data)
assert tf_matrix.shape[1] < 114751

words = vectorizer.get_feature_names()
for common_word in ['the', 'this', 'was', 'if', 'it', 'on']:
    assert common_word not in words
```

Убеждаемся, что размер словаря уменьшился

Типичные стоп-слова были удалены

Из повторно вычисленной `tf_matrix` были удалены все стоп-слова. Теперь можно заново определить десять наиболее частых слов в `newsgroups.data[0]` (листинг 15.15). Заметьте, что в ходе этого процесса мы повторно вычисляем `tf_np_matrix`, `tf_vector`, `unique_words`, `non_zero_indices` и `df`.

ВНИМАНИЕ

Операция повторного определения наиболее часто используемых слов требует 2,5 Гбайт памяти.

Листинг 15.15. Повторный вывод наиболее частых слов после удаления стоп-слов

```
tf_np_matrix = tf_matrix.toarray()
tf_vector = tf_np_matrix[0]
non_zero_indices = np.flatnonzero(tf_vector)
unique_words = [words[index] for index in non_zero_indices]
data = {'Word': unique_words,
        'Count': tf_vector[non_zero_indices]}

df = pd.DataFrame(data).sort_values('Count', ascending=False)
print(f"After stop-word deletion, {df.shape[0]} unique words remain.")
print("The 10 most frequent words are:\n")
print(df[:10].to_string(index=False))

After stop-word deletion, 34 unique words remain.
The 10 most frequent words are:
```

```
      Word  Count
       car      4
       60s      1
       saw      1
   looking      1
      mail      1
     model      1
production      1
    really      1
      rest      1
  separate      1
```

После фильтрации остались 34 слова. Среди них только *car* упоминается более одного раза. Остальные 33 используются в посте всего раз и рассматриваются векторизатором как равнозначные. Однако стоит отметить, что не все слова равны по своей релевантности. Некоторые относятся к дискуссии о машинах в большей степени. К примеру, слово *model* описывает модель автомобиля (хотя оно, конечно, может относиться и к супермодели или модели машинного обучения). А слово *really* является более общим, так как ни к чему связанному с машиной не относится. Оно настолько незначительно и распространено, что почти попадает в категорию стоп-слов. На практике некоторые специалисты по NLP действительно вносят *really* в список стоп-слов. К сожалению, нет единого мнения насчет того, какие слова всегда бесполезны, а какие — нет. Однако все специалисты соглашаются с тем, что слово становится менее актуальным, если упоминается в слишком большом числе текстов. Таким образом, *really* менее релевантно, чем *model*, потому что упоминается в большом числе постов. Следовательно, при ранжировании слов по релевантности нужно учитывать не только общее число вхождений слова, но и частоту его встречаемости в других постах. Если два слова имеют одинаковое количество вхождений, то их нужно ранжировать по частоте встречаемости в других постах.

Теперь повторно упорядочим наши 34 слова уже не только на основе их количества в посте, но и с учетом числа вхождений в другие посты. После этого рассмотрим, как полученный порядок слов можно использовать для повышения качества векторизации текстов.

ТИПИЧНЫЕ МЕТОДЫ COUNTVECTORIZER В SCIKIT-LEARN

- `vectorizer = CountVectorizer()` — инициализирует объект `CountVectorizer`, способный векторизовать входные тексты на основе частотности используемых в них терминов.
- `vectorizer = CountVectorizer(stopwords='english')` — инициализирует объект, способный векторизовать входные тексты, одновременно отфильтровав типичные английские слова вроде *this* и *the*.

- `tf_matrix = vectorizer.fit_transform(texts)` — векторизует список входных текстов, используя инициализированный объект `vectorizer`, и возвращает CSR-матрицу частоты встречаемости в этих текстах терминов. Каждая *i*-я строка матрицы соответствует `texts[i]`. Каждый *j*-й столбец соответствует частоте встречаемости слова `j`.
- `vocabulary_list = vectorizer.get_feature_names()` — возвращает список-словарь, ассоциируемый со столбцами вычисленной матрицы TF. Каждый *j*-й столбец матрицы соответствует `vocabulary_list[j]`.

15.3. РАНЖИРОВАНИЕ СЛОВ ПО ЧИСЛУ ВХОЖДЕНИЙ И ЧАСТОТЕ ВСТРЕЧАЕМОСТИ В ПОСТАХ

Каждое из 34 слов в `df.Word` встречается в определенной доле постов новостных групп. В NLP она называется *частотой встречаемости слова в наборе документов* или DF. Мы предположим, что DF может повысить качество ранжирования слов, и проверим это предположение, изучив, как DF связана с важностью слов. Изначально ограничим область изучения одним документом, а затем обобщим наши выводы и на другие документы набора данных.

ПРИМЕЧАНИЕ

В аналитике данных техника расширяемого изучения используется часто. Мы начинаем с анализа небольшого среза данных, что позволяет понять основные паттерны набора данных. Следующим шагом проверяем точность этого понимания уже в бóльших масштабах.

Приступим к анализу. Нашей непосредственной целью является вычисление частотности 34 слов во всех документах, что позволит более качественно ранжировать слова по релевантности. Найти эти частоты можно с помощью матричных операций NumPy. Для начала нужно выбрать столбцы `tf_np_matrix`, соответствующие 34 ненулевым индексам в массиве `non_zero_indices`. Получить эту подматрицу можно выполнением `tf_np_matrix[:,non_zero_indices]` (листинг 15.16).

Листинг 15.16. Выборка столбцов матрицы с ненулевыми индексами

```
                  Обращается только к тем столбцам матрицы, которые
                        содержат в первой строке ненулевые значения
sub_matrix = tf_np_matrix[:,non_zero_indices] ◄
print("Our sub-matrix corresponds to the 34 words within post 0. "
      "The first row of the sub-matrix is:")
print(sub_matrix[0])
Our sub-matrix corresponds to the 34 words within post 0. The first row of
 the sub-matrix is:
[1 1 1 1 1 1 1 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

Первая строка `sub_matrix` соответствует 34 значениям количеств слов в `df`. Все вместе строки матрицы соответствуют количествам вхождений слов во всех постах. Однако сейчас нас не интересует точное число вхождений, нам нужно знать, присутствует ли каждое слово в каждом посте. Для этого необходимо преобразовать количества вхождений слов в двоичные значения (листинг 15.17). По сути, нам нужна двоичная матрица, в которой элемент `(i, j)` равен `1`, если слово `j` присутствует в посте `i`, и `0` — в противном случае. Для перевода матрицы в двоичное представление нужно импортировать из `sklearn.preprocessing` функцию `binarize`, а затем выполнить `binarize(sub_matrix)`.

Листинг 15.17. Преобразование количеств слов в двоичные значения

```
from sklearn.preprocessing import binarize
binary_matrix = binarize(sub_matrix)
print(binary_matrix)

[[1 1 1 ... 1 1 1]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 1 0]
...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

Функция binarize заменяет все ненулевые элементы в любом x-мерном массиве единицами

Теперь нужно сложить строки этой двоичной матрицы (листинг 15.18). Таким образом мы получим вектор целочисленных количеств. Каждый *i*-й элемент вектора будет представлять количество уникальных постов, в которых слово `i` присутствует. Для сложения строк двухмерного массива нужно лишь передать в его метод `sum` параметр `axis=0`. Выполнение `binary_matrix.sum(axis=0)` возвращает вектор уникальных количеств постов.

ПРИМЕЧАНИЕ

Двухмерный массив NumPy содержит две оси: ось 0 соответствует горизонтальным строкам, а ось 1 — вертикальным столбцам. Значит, выполнение binary_matrix.sum(axis=0) возвращает вектор суммированных строк, а выполнение binary_matrix.sum(axis=1) — вектор суммированных столбцов.

Листинг 15.18. Суммирование строк матрицы для получения количеств постов

Как правило, выполнение multi_dim_array.sum(axis=i) возвращает вектор суммированных значений вдоль i-й оси многомерного массива

```
unique_post_mentions = binary_matrix.sum(axis=0)
print("This vector counts the unique posts in which each word is "
      f"mentioned:\n {unique_post_mentions}")

This vector counts the unique posts in which each word is mentioned:
[ 18   21  202   314    4     26  802  536  842  154    67  348  184   25
   7  368  469  3093  238    268  780  901  292   95  1493  407  354  158
 574   95   98     2   295  1174]
```

Здесь нужно отметить, что предыдущие три процедуры можно объединить в одну строку кода, выполнив `binarize(tf_np_matrix[:,non_zero_indices]).sum(axis=0)` (листинг 15.19). Более того, при замене `tf_np_matrix` из NumPy на `tf_matrix` из SciPy мы все равно получим то же количество постов, в которых упоминаются термины.

Листинг 15.19. Вычисление количества постов с упоминанием терминов одной строкой кода

```
np_post_mentions = binarize(tf_np_matrix[:,non_zero_indices]).sum(axis=0)
csr_post_mentions = binarize(tf_matrix[:,non_zero_indices]).sum(axis=0)
print(f'NumPy matrix-generated counts:\n {np_post_mentions}\n')
print(f'CSR matrix-generated counts:\n {csr_post_mentions}')

NumPy matrix-generated counts:
[  18   21  202   314    4    26  802  536  842  154    67  348  184   25
    7  368  469  3093  238   268  780  901  292   95  1493  407  354  158
  574   95   98     2  295  1174]

CSR matrix-generated counts:
[[  18   21  202   314    4    26  802  536  842  154    67  348  184   25
    7  368  469  3093  238   268  780  901  292   95  1493  407  354  158
  574   95   98     2  295  1174]]
```

Числа в np_post_mentions и csr_post_mentions оказываются идентичными. Однако csr_post_mentions содержит дополнительный уровень скобок, поскольку в результате агрегирования суммы строк CSR-матрицы возвращается не массив NumPy, а особый матричный объект. В нем одномерный массив представлен матрицей с одной строкой и n столбцов. Для ее преобразования в одномерный массив NumPy необходимо выполнить np.asarray(csr_post_mentions)[0]

МЕТОДЫ АГРЕГИРОВАНИЯ СТРОК МАТРИЦЫ

- `vector_of_sums = np_matrix.sum(axis=0)` — суммирует строки матрицы NumPy. Если `np_matrix` является матрицей TF, то `vector_of_sums[i]` равен общему числу вхождений слова `i` в набор данных.
- `vector_of_sums = binarize( np_matrix).sum(axis=0)` — преобразует матрицу NumPy в двоичную матрицу, после чего суммирует ее строки. Если `np_matrix` является матрицей TF, то `vector_of_sums[i]` равен общему числу текстов, в которых упоминается слово `i`.
- `matrix_1D = binarize(csr_matrix).sum(axis=0)` — преобразует CSR-матрицу в двоичную, после чего суммирует ее строки. Возвращаемый результат является особым одномерным матричным объектом, а не вектором NumPy. Эту `matrix_1D` можно преобразовать в вектор NumPy, выполнив `np.asarray(matrix_1D)[0]`.

На основе полученного вектора количеств постов с упоминанием слов мы понимаем, что некоторые слова встречаются в тысячах постов. При этом определенная часть слов попадается менее чем в десяти публикациях. Давайте преобразуем эти количества в DF и сопоставим эти частоты с `df.word`. Затем выведем все слова, упоминаемые не менее чем в 10 % постов новостных групп (листинг 15.20). Эти слова, скорее всего, будут разбросаны по различным постам, поэтому мы предположим, что они не являются специфичными для какой-то конкретной темы. Если наше предположение окажется верным, релевантность этих слов будет низкой.

Листинг 15.20. Вывод слов с максимальной встречаемостью в документах

```
document_frequencies = unique_post_mentions / dataset_size
data = {'Word': unique_words,
        'Count': tf_vector[non_zero_indices],
        'Document Frequency': document_frequencies}

df = pd.DataFrame(data)
df_common_words = df[df['Document Frequency'] >= .1]
print(df_common_words.to_string(index=False))
```

Выбираются только слова с DF выше 1/10

```
  Word  Count  Document  Frequency
  know      1             0.273378
really      1             0.131960
 years      1             0.103765
```

Напомню, что Document frequency относится ко всем постам, а count — только к посту с индексом 0

Три из 34 слов имеют DF выше 0,1. Как и ожидалось, эти слова очень общие и не относятся конкретно к машинам. Значит, можно использовать DF для ранжирования. Далее мы распределим слова по релевантности следующим образом. Сначала упорядочим их по количеству в порядке убывания. Затем все слова с равным количеством также в порядке убывания упорядочим по частоте встречаемости в документах (листинг 15.21). В Pandas это упорядочение по двум столбцам можно произвести выполнением `df.sort_values(['Count', 'Document Frequency'], ascending=[False, True])`.

Листинг 15.21. Ранжирование слов по количеству и частоте встречаемости в наборе документов

```
df_sorted = df.sort_values(['Count', 'Document Frequency'],
                           ascending=[False, True])
print(df_sorted[:10].to_string(index=False))
```

```
      Word  Count  Document  Frequency
       car      4             0.047375
    tellme      1             0.000177
  bricklin      1             0.000354
     funky      1             0.000619
       60s      1             0.001591
       70s      1             0.001856
 enlighten      1             0.002210
    bumper      1             0.002298
     doors      1             0.005922
production      1             0.008397
```

Упорядочение прошло успешно. Новые связанные с машинами термины, такие как *bumper*, теперь представлены в списке слов с максимальной частотностью и релевантностью. Однако сама процедура сортировки была довольно запутанной, так как требовала отдельного упорядочения двух столбцов. Возможно, удастся упростить этот процесс, совместив количества слов и частоту их встречаемости в наборе документов в один показатель. Как это сделать? Один из вариантов — разделить TF каждого слова на связанную с ним DF. Итоговое значение будет увеличиваться при выполнении любого из следующих условий:

- при увеличении частотности слова;
- уменьшении частоты встречаемости слова в наборе документов.

Далее мы совместим TF и DF слов в один показатель (листинг 15.22). Начнем с вычисления `1 / document_frequencies`. Это действие даст массив значений *обратной частоты встречаемости термина в наборе документов* (IDF). Затем мы умножим `df.Count` на массив IDF, чтобы вычислить общий показатель. После этого добавим значения IDF и общие показатели в таблицу Pandas. В завершение упорядочим общие показатели и выведем топ результатов.

Листинг 15.22. Объединение TF и DF в один показатель

```
inverse_document_frequencies = 1 / document_frequencies
df['IDF'] = inverse_document_frequencies
df['Combined'] = df.Count * inverse_document_frequencies
df_sorted = df.sort_values('Combined', ascending=False)
print(df_sorted[:10].to_string(index=False))
```

```
      Word  Count  Document Frequency          IDF     Combined
    tellme      1            0.000177  5657.000000  5657.000000
  bricklin      1            0.000354  2828.500000  2828.500000
     funky      1            0.000619  1616.285714  1616.285714
       60s      1            0.001591   628.555556   628.555556
       70s      1            0.001856   538.761905   538.761905
 enlighten      1            0.002210   452.560000   452.560000
    bumper      1            0.002298   435.153846   435.153846
     doors      1            0.005922   168.865672   168.865672
     specs      1            0.008397   119.094737   119.094737
production      1            0.008397   119.094737   119.094737
```

Новое ранжирование провалилось! Слово *car* больше не отражается в списке ведущих терминов. Что же произошло? Заглянем в таблицу. Здесь у нас проблема со значениями IDF — некоторые из них огромны. В целом их диапазон — примерно от 100 до более чем 5000. При этом диапазон количества слов очень мал, всего от 1 до 4. Поэтому когда мы умножаем эти количества на значения IDF, то IDF доминируют и количества на конечный результат не влияют. Нужно каким-то образом уменьшить значения IDF. Но как?

Аналитики данных зачастую сталкиваются с излишне большими численными значениями. Один из способов уменьшить их — применить логарифмическую

функцию. К примеру, выполнение `np.log10(1000000)` вернет 6 (листинг 15.23). По сути, значение 1 000 000 заменяется количеством нулей в нем.

Листинг 15.23. Уменьшение большого значения через получение его логарифма

```
assert np.log10(1000000) == 6
```

Еще раз вычислим рейтинговый показатель, выполнив `df.Count * np.log10(df.IDF)`. Теперь перемножение количества слов и уменьшенных значений IDF должно давать более адекватный показатель рейтинга (листинг 15.24).

Листинг 15.24. Корректировка общего показателя с помощью логарифмов

```
df['Combined'] = df.Count * np.log10(df.IDF)
df_sorted = df.sort_values('Combined', ascending=False)
print(df_sorted[:10].to_string(index=False))
```

```
     Word  Count  Document  Frequency          IDF  Combined
      car      4             0.047375    21.108209  5.297806
   tellme      1             0.000177  5657.000000  3.752586
 bricklin      1             0.000354  2828.500000  3.451556
    funky      1             0.000619  1616.285714  3.208518
      60s      1             0.001591   628.555556  2.798344
      70s      1             0.001856   538.761905  2.731397
enlighten      1             0.002210   452.560000  2.655676
   bumper      1             0.002298   435.153846  2.638643
    doors      1             0.005922   168.865672  2.227541
    specs      1             0.008397   119.094737  2.075893
```

Скорректированные показатели демонстрируют хорошие результаты. Слово *car* снова присутствует в топе списка. Кроме того, *bumper* по-прежнему входит в топ-10 ранжированных слов, а слово *really* в нем отсутствует.

Полученный в итоге показатель называется *«частота встречаемости термина — обратная частота встречаемости термина в наборе документов»* (TF-IDF). Его можно вычислить умножением TF (количества слов) на логарифм IDF.

ПРИМЕЧАНИЕ

Математически np.log(1/x) равен -np.log(x). Следовательно, можно вычислить TF-IDF напрямую из DF. Также имейте в виду, что в литературе встречаются и другие, менее распространенные формулировки TF-IDF. К примеру, работая с большими документами, некоторые специалисты по NLP вычисляют TF-IDF как np.log(df.Count + 1) * -np.log10(document_frequences). Это ограничивает влияние любого сильно распространенного в документе слова.

TF-IDF — это простой, но вместе с тем мощный показатель для ранжирования слов в документе. Естественно, он оказывается подходящим, только если документ является частью большего набора документов. В противном случае полученные значения TF-IDF будут равны нулю. Этот показатель теряет свою эффективность

и тогда, когда применяется к небольшой коллекции похожих текстов. И все же для большинства реальных текстовых наборов данных он дает хорошие результаты ранжирования. Есть у этого показателя и дополнительное применение: его можно задействовать для векторизации слов в документе. Численное содержимое `df.Combined`, по сути, является вектором, созданным в результате изменения вектора TF, хранящегося в `df.Count`. Аналогичным образом можно преобразовать любой вектор TF в вектор TF-IDF. Для этого нужно лишь умножить вектор TF на логарифмы обратной частоты встречаемости терминов в наборе документов, то есть IDF.

Есть ли польза от преобразования векторов TF в более сложные векторы TF-IDF? О да! В более крупных наборах текстов векторы TF-IDF обеспечивают лучший сигнал сходства и расхождения этих текстов. Например, два текста, в которых речь идет о машинах, с большей вероятностью кластеризуются вместе, если значимость их не относящихся к делу элементов понизить. Таким образом, понижение значимости распространенных слов с помощью IDF повышает качество кластеризации больших коллекций текстов.

ПРИМЕЧАНИЕ

Это не обязательно актуально для небольших наборов данных, в которых количество документов мало, а DF терминов высока. В связи с этим IDF может оказаться слишком низкой для того, чтобы значительно улучшить результаты кластеризации.

Поэтому мы выигрываем, когда преобразуем нашу матрицу TF в матрицу TF-IDF. Такую трансформацию несложно выполнить и с помощью собственного кода, однако удобнее сделать это с помощью предлагаемого scikit-learn класса `TfidfVectorizer`.

15.3.1. Вычисление векторов TF-IDF с помощью scikit-learn

Класс `TfidfVectorizer` почти идентичен `CountVectorizer`, за исключением того, что в процессе векторизации учитывает IDF. Далее мы импортируем `TfidfVectorizer` из `sklearn.feature_extraction.text` и инициализируем этот класс, выполнив `TfidfVectorizer(stop_words='english')`. Полученный объект `tfidf_vectorizer` будет настроен на игнорирование всех стоп-слов. Последующее выполнение `tfidf_vectorizer.fit_transform(newsgroups.data)` возвращает матрицу векторизованных значений TF-IDF (листинг 15.25). Форма этой матрицы будет идентична `tf_matrix.shape`.

Листинг 15.25. Вычисление матрицы TF-IDF с помощью scikit-learn

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(stop_words='english')
tfidf_matrix = tfidf_vectorizer.fit_transform(newsgroups.data)
assert tfidf_matrix.shape == tf_matrix.shape
```

Наш `tfidf_vectorizer` выучил тот же словарь, что и более простой векторизатор TF. В действительности индексы слов в `tfidf_matrix` идентичны индексам в `tf_matrix`. Убедиться в этом можно с помощью `tfidf_vectorizer.get_feature_names()`(листинг 15.26). Этот метод вернет упорядоченный список слов, идентичный ранее вычисленному списку `words`.

Листинг 15.26. Подтверждение сохранения индексов векторизованных слов

```
assert tfidf_vectorizer.get_feature_names() == words
```

Поскольку порядок слов сохраняется, можно ожидать, что ненулевые индексы `tfidf_matrix[0]` будут равны ранее вычисленному массиву `non_zero_indices`. В этом мы убедимся после преобразования `tfidf_matrix` из структуры CSR в массив NumPy (листинг 15.27).

Листинг 15.27. Подтверждение сохранения ненулевых индексов

```
tfidf_np_matrix = tfidf_matrix.toarray()
tfidf_vector = tfidf_np_matrix[0]
tfidf_non_zero_indices = np.flatnonzero(tfidf_vector)
assert np.array_equal(tfidf_non_zero_indices,
                      non_zero_indices)
```

Индексы ненулевых значений `tf_vector` и `tfidf_vector` идентичны. Это значит, что можно добавить вектор TF-IDF в качестве столбца в нашу таблицу `df` (листинг 15.28). Внесение столбца `TF-IDF` позволит сравнить вывод scikit-learn с показателем, вычисленным вручную.

Листинг 15.28. Внесение вектора TF-IDF в имеющуюся таблицу Pandas

```
df['TFIDF'] = tfidf_vector[non_zero_indices]
```

Упорядочение по `df.TFIDF` дает рейтинг релевантности, согласующийся с прежними наблюдениями. Убедимся, что и `df.TFIDF`, и `df.Combined` после упорядочения дают одинаковый рейтинг слов (листинг 15.29).

Листинг 15.29. Упорядочение слов по df.TFIDF

```
df_sorted_old = df.sort_values('Combined', ascending=False)
df_sorted_new = df.sort_values('TFIDF', ascending=False)
assert np.array_equal(df_sorted_old['Word'].values,
                      df_sorted_new['Word'].values)
print(df_sorted_new[:10].to_string(index=False))
```

```
     Word  Count  Document  Frequency          IDF  Combined     TFIDF
      car      4             0.047375    21.108209  5.297806  0.459552
   tellme      1             0.000177  5657.000000  3.752586  0.262118
 bricklin      1             0.000354  2828.500000  3.451556  0.247619
    funky      1             0.000619  1616.285714  3.208518  0.234280
      60s      1             0.001591   628.555556  2.798344  0.209729
```

```
      70s      1            0.001856   538.761905  2.731397  0.205568
enlighten      1            0.002210   452.560000  2.655676  0.200827
   bumper      1            0.002298   435.153846  2.638643  0.199756
    doors      1            0.005922   168.865672  2.227541  0.173540
    specs      1            0.008397   119.094737  2.075893  0.163752
```

Рейтинг слов остался неизменным. Однако значения в столбцах `TF-IDF` и `Combined` не идентичны. Все топ-10 наших вручную вычисленных значений `Combined` оказались больше 1, а все значения TF-IDF из scikit-learn меньше 1. Почему так?

Оказывается, scikit-learn автоматически нормализует свои результаты вектора TF-IDF. Абсолютная величина `df.TFIDF` была приведена к равенству 1. Убедиться в этом можно, выполнив `norm(df.TFIDF.values)` (листинг 15.30).

ПРИМЕЧАНИЕ

Для отключения нормализации необходимо передать в функцию инициализации векторизатора аргумент norm=None. Выполнение TfidfVectorizer(norm=None, stop_words='english') приведет к возвращению векторизатора с отключенной нормализацией.

Листинг 15.30. Убеждаемся, что вектор TF-IDF нормализован

```
from numpy.linalg import norm
assert norm(df.TFIDF.values) == 1
```

Почему scikit-learn автоматически нормализует векторы? Ради нашего же блага! Как говорилось в главе 13, проще вычислять сходство векторов текстов, когда абсолютные величины всех векторов равны 1. Следовательно, наша нормализованная матрица TF-IDF подготовлена к анализу сходства.

ТИПИЧНЫЕ МЕТОДЫ TFIDFVECTORIZER В SCIKIT-LEARN

- `tfidf_vectorizer = TfidfVectorizer(stopwords='english')` — инициализирует объект `TfidfVectorizer`, способный векторизовать входные тексты на основе их значений TF-IDF. Этот объект настроен на исключение английских стоп-слов.
- `tfidf_matrix = tfidf_vectorizer.fit_transform(texts)` — выполняет векторизацию TF-IDF для списка входных текстов, используя инициализированный объект `vectorizer`, и возвращает CSR-матрицу нормализованных значений TF-IDF. Каждая ее строка автоматически нормализуется для упрощения вычисления сходства.
- `vocabulary_list = tfidf_vectorizer.get_feature_names()` — возвращает список-словарь, ассоциируемый со столбцами вычисленной матрицы TF-IDF. Каждый столбец `j` матрицы соответствует `vocabulary_list[j]`.

15.4. ВЫЧИСЛЕНИЕ СХОДСТВА СРЕДИ ОГРОМНЫХ НАБОРОВ ДОКУМЕНТОВ

Ответим на простой вопрос: «Какие из наших постов в новостных группах больше всего похожи на `newsgroups.post[0]`?» Получить ответ можно, вычислив все косинусные коэффициенты между `tfidf_np_matrix` и `tf_np_matrix[0]`. Как говорилось в главе 13, эти коэффициенты вычисляются умножением `tfidf_np_matrix` на `tfidf_matrix[0]` (листинг 15.31). Простого перемножения матрицы и вектора достаточно, поскольку все строки матрицы имеют абсолютную величину 1.

Листинг 15.31. Вычисление сходства с одним постом из новостных групп

```
cosine_similarities = tfidf_np_matrix @ tfidf_np_matrix[0]
print(cosine_similarities)

[1.        0.00834093 0.04448717 ... 0.        0.00270615 0.01968562]
```

На вычисление произведения между матрицей и вектором уходит несколько секунд. Итоговый вектор будет отражать косинусные коэффициенты: каждый его *i*-й индекс соответствует косинусному коэффициенту между `newsgroups.data[0]` и `newsgroups.data[i]`. Из вывода видно, что `cosine_similarities[0]` равен 1,0. И это неудивительно, поскольку `newsgroups_data[0]` будет иметь идеальное сходство с самим собой. А какой косинусный коэффициент в этом векторе будет следующим по старшинству? Выяснить это можно, выполнив `np.argsort(cosine_similarities)[-2]`. Вызов `argsort` упорядочивает индексы массива от меньших значений к бóльшим, поэтому предпоследний индекс будет соответствовать посту со вторым по величине косинусным коэффициентом.

ПРИМЕЧАНИЕ

Мы предполагаем, что в наборе данных больше нет постов с идеальным сходством 1. Имейте также в виду, что того же результата можно достичь с помощью вызова np.argmax(cosine_similarities[1:]) + 1, хотя такой подход работает только для постов по индексу 0.

Теперь извлечем этот индекс и выведем соответствующий ему коэффициент сходства (листинг 15.32). А также выведем связанный с ним текст, чтобы убедиться в совпадениях с постом про машину в `newsgroups.data[0]`.

Листинг 15.32. Поиск наиболее похожего поста в новостных группах

```
most_similar_index = np.argsort(cosine_similarities)[-2]
similarity = cosine_similarities[most_similar_index]
most_similar_post = newsgroups.data[most_similar_index]
print(f"The following post has a cosine similarity of {similarity:.2f} "
       "with newsgroups.data[0]:\n")
print(most_similar_post)
```

```
The following post has a cosine similarity of 0.64 with newsgroups.data[0]:

In article <1993Apr20.174246.14375@wam.umd.edu> lerxst@wam.umd.edu
(where's my
thing) writes:
>
>I was wondering if anyone out there could enlighten me on this car I saw
> the other day. It was a 2-door sports car, looked to be from the late
> 60s/ early 70s. It was called a Bricklin. The doors were really small. In
addition,
> the front bumper was separate from the rest of the body. This is
> all I know. If anyone can tellme a model name, engine specs, years
> of production, where this car is made, history, or whatever info you
> have on this funky looking car, please e-mail.

Bricklins were manufactured in the 70s with engines from Ford. They are rather
odd looking with the encased front bumper. There aren't a lot of them around, but
Hemmings (Motor News) ususally has ten or so listed.
Basically, they are a performance Ford with new styling slapped on top.

>    ---- brought to you by your neighborhood Lerxst ----

Rush fan?
```

Выведенный текст является повтором поста про машину по индексу 0. Этот ответ включает исходный пост, в котором задан вопрос о бренде машины. В нижней части вывода мы видим подробный ответ на этот вопрос. Ввиду сходства текстов оригинальный пост оказывается очень похож на ответ. При этом их косинусный коэффициент составляет 0,64, что не похоже на большое значение. Однако в обширных коллекциях текстов коэффициент сходства выше 0,6 — это хороший показатель пересекающегося содержания.

ПРИМЕЧАНИЕ

Из главы 13 мы помним, что косинусный коэффициент можно преобразовать в коэффициент Танимото, который имеет более глубокую теоретическую основу для сопоставления текстов. Превратить cosine_similarities в коэффициент Танимото можно выполнением cosine_similarities / (2 - cosine_similarities). Однако это превращение не изменит итогового результата. Выбор верхнего индекса в массиве Танимото вернет тот же ответ на пост. Поэтому, чтобы не усложнять, при рассмотрении следующих примеров сравнения текстов сосредоточимся на косинусном коэффициенте.

К настоящему моменту мы проанализировали только пост про машину, расположенный по индексу 0. Далее расширим анализ на еще один пост. Мы наобум выберем его среди новостных групп, найдем еще один, который будет больше всех на него похож, и выведем их оба вместе с коэффициентом сходства. А чтобы сделать этот пример поинтереснее, мы сначала вычислим матрицу косинусных коэффициентов между всеми постами, а затем используем ее для выбора случайной пары похожих постов.

ПРИМЕЧАНИЕ

Зачем вычислять матрицу сходств между всеми постами? В первую очередь чтобы попрактиковаться в применении навыков, полученных в предыдущей главе, хотя наличие доступа к этой матрице дает и некоторые преимущества. Предположим, мы хотим увеличить нашу сеть соседних постов с 2 до 10, а также включить соседа каждого соседа (по аналогии с кластеризацией алгоритмом DBSCAN, рассмотренной в главе 10). В таком случае будет гораздо эффективнее заранее вычислить все сходства.

Как вычислить матрицу косинусных сходств между всеми постами? Наивным подходом будет умножить `tfidf_np_matrix` на ее транспонированную версию. Однако по причинам, рассмотренным в главе 13, это окажется вычислительно неэффективным. В нашей матрице TF-IDF более 100 000 столбцов, поэтому, прежде чем выполнять умножение, нужно уменьшить ее размер. В предыдущей главе мы научились сокращать количество столбцов с помощью класса `TruncatedSVD` из scikit-learn. Он может уменьшить матрицу до заданного числа столбцов, которое определяется параметром `n_components`. При этом документация scikit-learn рекомендует для обработки текстовых данных значение `n_components`, равное `100`.

ПРИМЕЧАНИЕ

В документации scikit-learn временами приводятся полезные параметры для типичных случаев применения алгоритмов. Заглянем, к примеру, в мануал к TruncatedSVD на странице http://mng.bz/PXP9. Там написано: «TruncatedSVD работает для матриц TF/TF-IDF, возвращаемых векторизаторами в sklearn.feature_extraction.text. В данном контексте это называется латентным семантическим анализом (LSA)». Далее в документации параметр n_components описывается так: «Желательная размерность выходных данных. Должен быть строго меньше количества признаков. Для LSA его значение рекомендуется устанавливать равным 100».

Большинство специалистов по NLP согласны с тем, что передача `n_components=100` уменьшает матрицу TF-IDF до эффективного размера, сохраняя при этом полезную информацию столбцов. Мы последуем этой рекомендации, выполнив `TruncatedSVD(n_components=100).fit_transform(tfidf_matrix)` (листинг 15.33). Вызов этого метода вернет `shrunk_matrix` со 100 столбцами, которая будет являться двухмерным массивом NumPy, даже если мы передадим в качестве ввода `tfidf_matrix` из SciPy.

Листинг 15.33. Уменьшение размерности tfidf_matrix с помощью SVD

Итоговый вывод SVD зависит от направления вычисленных собственных векторов. Как мы видели в предыдущей главе, это направление определяется случайно. Следовательно, для обеспечения согласованности результатов мы выполняем np.random.seed(0)

```
np.random.seed(0)
from sklearn.decomposition import TruncatedSVD

shrunk_matrix = TruncatedSVD(n_components=100).fit_transform(tfidf_matrix)
print(f"We've dimensionally reduced a {tfidf_matrix.shape[1]}-column "
      f"{type(tfidf_matrix)} matrix.")
```

```
print(f"Our output is a {shrunk_matrix.shape[1]}-column "
      f"{type(shrunk_matrix)} matrix.")

We've dimensionally reduced a 114441-column
<class 'scipy.sparse.csr.csr_matrix'> matrix.
Our output is a 100-column <class 'numpy.ndarray'> matrix.
```

Уменьшенная матрица содержит всего 100 столбцов. Теперь можно эффективно вычислить ее косинусные сходства, выполнив `shrunk_matrix @ shrunk_matrix.T`. Однако сначала нужно убедиться, что строки этой матрицы остаются нормализованными. Проверим абсолютную величину `shrunk_matrix[0]` (листинг 15.34).

Листинг 15.34. Проверка абсолютной величины shrunk_matrix[0]

```
magnitude = norm(shrunk_matrix[0])
print(f"The magnitude of the first row is {magnitude:.2f}")

The magnitude of the first row is 0.49
```

Абсолютная величина строки меньше 1. Вывод SVD не был автоматически нормализован. Нам нужно вручную нормализовать эту матрицу, прежде чем вычислять косинусные коэффициенты. В этом поможет встроенная в scikit-learn функция `normalize`. Мы импортируем ее из `sklearn.preprocessing`, а затем выполним `normalize (shrunk_matrix)` (листинг 15.35). В итоге абсолютная величина строк в полученной нормализованной матрице будет равняться 1.

Листинг 15.35. Нормализация вывода SVD

```
from sklearn.preprocessing import normalize
shrunk_norm_matrix = normalize(shrunk_matrix)
magnitude = norm(shrunk_norm_matrix[0])
print(f"The magnitude of the first row is {magnitude:.2f}")

The magnitude of the first row is 1.00
```

Уменьшенная матрица была нормализована. Теперь выполнение `shrunk_norm_matrix @ shrunk_norm_matrix.T` должно дать матрицу косинусного сходства между всеми постами (листинг 15.36).

Листинг 15.36. Вычисление косинусного сходства всех со всеми

```
cosine_similarity_matrix = shrunk_norm_matrix @ shrunk_norm_matrix.T
```

Вот мы и получили матрицу сходства. Теперь с ее помощью выберем случайную пару очень похожих текстов (листинг 15.37). Начнем со случайного выбора поста по некоему `index1` (листинг 15.38). Затем выберем индекс `cosine_similarities[index1]`, обладающий вторым по величине косинусным коэффициентом. После этого, прежде чем отображать тексты, выведем оба индекса и коэффициент их сходства.

Листинг 15.37. Выбор случайной пары похожих постов

```
np.random.seed(1)
index1 = np.random.randint(dataset_size)

index2 = np.argsort(cosine_similarity_matrix[index1])[-2]
similarity = cosine_similarity_matrix[index1][index2]
print(f"The posts at indices {index1} and {index2} share a cosine "
      f"similarity of {similarity:.2f}")

The posts at indices 235 and 7805 share a cosine similarity of 0.91
```

Листинг 15.38. Вывод случайно выбранного поста

```
print(newsgroups.data[index2].replace('\n\n', '\n'))
Hello,
    Who can tell meWhere can I find the PD or ShareWare
Which can CAPTURE windows 3.1’s output of printer mananger?
    I want to capture the output of HP Laser Jet III.
    Though the PostScript can setup to print to file,but HP can't.
    I try DOS's redirect program,but they can’t work in Windows 3.1
        Thankx for any help....
--
Internet Address: u7911093@cc.nctu.edu.tw
    English Name: Erik Wang
    Chinese Name: Wang Jyh-Shyang
```

Этот пост содержит пустые строки. В целях экономии места мы их исключаем

И снова выведенный пост является вопросом. Можно смело предположить, что пост по `index1` — это ответ на него (листинг 15.39).

Листинг 15.39. Вывод наиболее похожего поста с ответом

```
print(newsgroups.data[index1].replace('\n\n', '\n'))

u7911093@cc.nctu.edu.tw ("By SWH ) writes:
>Who can tell me which program (PD or ShareWare) can redirect windows 3.1's
>output of printer manager to file?
>    I want to capture HP Laser Jet III's print output.
>     Though PostScript can setup print to file,but HP can't.
>    I use DOS's redirect program,but they can't work in windows.
>        Thankx for any help...
>--
> Internet Address: u7911093@cc.nctu.edu.tw
>    English Name: Erik Wang
>    Chinese Name: Wang Jyh-Shyang
> National Chiao-Tung University,Taiwan,R.O.C.
Try setting up another HPIII printer but when choosing what port to connect it to
choose FILE instead of like :LPT1. This will prompt you for a file name everytime
you print with that "HPIII on FILE" printer. Good Luck.
```

К этому моменту мы проверили две пары похожих постов. Каждая из них состояла из вопроса и ответа, в который был включен этот вопрос. Подобные скучные пары совпадающих текстов извлекать очень просто. Предлагаю повысить сложность и найти что-то поинтереснее. Далее мы займемся поиском кластеров похожих текстов, в которых посты имеют общее содержание, но повторяются не полностью.

15.5. КЛАСТЕРИЗАЦИЯ ПОСТОВ ПО ТЕМАМ

В главе 10 мы познакомились с двумя алгоритмами кластеризации: методом *K*-средних и DBSCAN. Метод *K*-средних способен кластеризовать данные только по евклидову расстоянию. DBSCAN же способен делать это на основе любого показателя расстояния. Одним из возможных является косинусное расстояние, равное 1 минус косинусный коэффициент.

ПРИМЕЧАНИЕ

Зачем использовать вместо косинусного коэффициента косинусное расстояние? Все алгоритмы кластеризации предполагают, что расстояние между двумя идентичными точками данных равно 0. И наоборот, косинусный коэффициент равен 0, если две точки данных не имеют ничего общего. В случае же их полной идентичности его значение равно 1. Это расхождение можно исправить, выполнив 1 - cosine_similarity_matrix и тем самым преобразовав результат в косинусный коэффициент. После преобразования для двух идентичных текстов косинусное расстояние будет равно 0.

Косинусное расстояние обычно используется совместно с DBSCAN. Именно поэтому реализация DBSCAN в scikit-learn позволяет указывать косинусное расстояние непосредственно во время инициализации объекта. Для этого достаточно передать `metric='cosine'` в конструктор класса. Так мы инициализируем объект `cluster_model`, настроенный выполнять кластеризацию на основе косинусного расстояния.

ПРИМЕЧАНИЕ

Реализация DBSCAN в scikit-learn вычисляет косинусное расстояние, начиная с повторного вычисления cosine_similarity_matrix. Однако повторного вычисления можно избежать, передав в конструктор metric='precomputed'. Таким образом инициализируется объект cluster_model, который будет кластеризовать данные на основе матрицы ранее вычисленных расстояний. Последующее выполнение cluster_model.fit_transform(1 – cosine_similarity_matrix) теоретически должно вернуть результат кластеризации. Однако в практическом смысле отрицательные значения в этой матрице расстояний, которые могут возникнуть вследствие погрешностей из-за плавающей точки, способны вызвать проблемы во время кластеризации. Все отрицательные значения в данной матрице перед кластеризацией необходимо заменить на нули. Эту операцию потребуется выполнить вручную в NumPy с помощью x[x < 0] = 0, где x = 1 – cosine_similarity_matrix.

Далее мы кластеризуем `shrunk_matrix` с помощью DBSCAN на основе косинусного расстояния. Во время этого примем следующие разумные допущения.

- Два поста в новостных группах попадают в кластер, если их косинусный коэффициент не менее 0,6 (что соответствует косинусному расстоянию более 0,4).
- Кластер содержит не менее 50 постов.

Исходя из этих допущений, параметры `eps` и `min_samples` алгоритма должны равняться 0,4 и 50 соответственно. Значит, мы инициализируем DBSCAN выполнением

`DBSCAN(eps=0.4, min_samples=50, metric='cosine')`. После этого с помощью инициализированного объекта `cluster_model` кластеризуем `shrunk_matrix` (листинг 15.40).

Листинг 15.40. Кластеризация постов с помощью DBSCAN

```
from sklearn.cluster import DBSCAN
cluster_model = DBSCAN(eps=0.4, min_samples=50, metric='cosine')

clusters = cluster_model.fit_predict(shrunk_matrix)
```

Мы сгенерировали массив кластеров. Давайте вкратце оценим качество его кластеризации. Нам уже известно, что набор данных в новостных группах охватывает 20 категорий. Названия некоторых из них между собой очень похожи, другие же, напротив, стоят особняком. Исходя из этого, разумно предположить, что весь набор данных охватывает от 10 до 25 действительно разных тем. Следовательно, можно ожидать, что наш массив `clusters` будет содержать от 10 до 25 кластеров — иной результат укажет на проблемы с входными параметрами кластеризации. Код листинга 15.41 подсчитывает количество кластеров.

Листинг 15.41. Подсчет количества кластеров DBSCAN

```
cluster_count = clusters.max() + 1
print(f"We've generated {cluster_count} DBSCAN clusters")

We've generated 3 DBSCAN clusters
```

Мы сгенерировали всего три кластера, что намного меньше ожидаемого количества. Очевидно, что параметры DBSCAN ошибочны. Есть ли какой-то алгоритмический способ скорректировать их? А может, в литературе имеются уже известные настройки DBSCAN, дающие приемлемые текстовые кластеры? Печально, но нет. Как выясняется, кластеризация текстов с помощью DBSCAN очень чувствительна к входным данным документов. Параметры этого алгоритма для кластеризации определенных типов текстов, таких как посты в новостных группах, вряд ли удачно преобразуются в параметры для других категорий документов, например новостных статей или электронных писем. Следовательно, в отличие от SVD, алгоритму DBSCAN недостает согласованных параметров NLP. Это не значит, что его нельзя применить к нашим текстовым данным, но подходящие значения для `eps` и `min_samples` необходимо подбирать путем проб и ошибок. К сожалению, в DBSCAN нет грамотного алгоритма для оптимизации этих двух важнейших параметров.

Метод *K*-средних, напротив, получает на входе один параметр *K*. Прикинуть его значение можно с помощью метода локтя, с которым мы познакомились в главе 10. Однако алгоритм *K*-средних способен выполнять кластеризацию только на основе евклидова расстояния, он не может обрабатывать косинусное расстояние. Проблема ли это? Не обязательно. Оказывается, нам повезло! Все строки в `shrunk_norm_matrix` являются нормализованными единичными векторами. В главе 13 мы видели, что евклидово расстояние двух нормализованных векторов `v1` и `v2` равно `(2 - 2 * v1 @ v2) ** 0.5`. Кроме того, косинусное расстояние между этими векторами

равно `1 – v1 @ v2`. С помощью простой алгебры можно легко показать, что евклидово расстояние между двумя нормализованными векторами пропорционально квадратному корню из их косинусного расстояния. Эти два параметра расстояния очень тесно связаны, что дает математическое обоснование для кластеризации `shrunk_norm_matrix` с помощью метода *K*-средних.

ВНИМАНИЕ

Если два вектора нормализованы, то евклидово расстояние между ними вполне может заменить собой косинусный коэффициент. Однако это не касается ненормализованных векторов. Значит, метод *K*-средних не нужно применять к текстовым матрицам, если они не нормализованы.

Исследования показали, что кластеризация методом *K*-средних обеспечивает адекватную сегментацию текстовых данных. Это может сбить с толку, поскольку в предыдущей главе DBSCAN дал более качественные результаты. К сожалению, в науке о данных правильный выбор алгоритма зависит от предметной области. Очень редко алгоритм может сработать универсально. Это можно сравнить с тем, как при ремонте мы не все задачи решаем с помощью молотка, иногда требуется, к примеру, отвертка или гаечный ключ. Аналитики данных должны гибко подходить к выбору подходящего для поставленной задачи инструмента.

ПРИМЕЧАНИЕ

Порой мы не знаем, какой алгоритм лучше использовать для конкретной задачи. В таких случаях стоит поискать известные решения в Сети. В частности, сайт scikit-learn предоставляет полезные решения для типичных случаев. Так, на нем есть пример кода для кластеризации текста: http://mng.bz/wQ9q. Примечательно, что этот задокументированный код показывает, как метод *K*-средних может кластеризовать текстовые векторы (после обработки SVD). В этой документации также указывается, что для достижения лучшего результата векторы должны быть нормализованы.

Далее мы задействуем метод *K*-средних для кластеризации `shrunk_norm_matrix` на *K* категорий новостных групп. Сначала потребуется выбрать значение *K*. Предположительно, наши тексты относятся к 20 категориям новостных групп. Однако, как говорилось ранее, реальное количество кластеров может не быть равным 20. Нам нужно прикинуть правильное значение *K*, сгенерировав график локтя. Для этого выполним метод *K*-средних для *K* значений от 1 до 60, а затем построим график результатов.

Вот только здесь есть проблема. Наш набор данных очень велик и содержит более 10 000 точек данных. Предоставляемая scikit-learn реализация `KMeans` подходит для одного выполнения кластеризации, но никак не для 60 разных, когда общее время выполнения может достичь нескольких минут. Как можно ускорить метод *K*-средних? Один из вариантов — сделать из нашего огромного набора данных случайную выборку. Можно отобрать 1000 случайных постов во время вычисления

центра масс, после чего выбрать еще 1000 случайных постов и потом обновить центры кластеров на основе их содержания. Таким образом можно итеративно оценить центры масс с помощью сэмплирования. В итоге нам не потребуется анализировать весь набор данных за раз. Эта измененная версия алгоритма *K*-средних называется *методом K-средних для мини-пакетов*. Scikit-learn позволяет выполнить такую мини-пакетную реализацию с помощью класса `MiniBatchKMeans`. В своих методах он практически идентичен стандартному классу `KMeans`. Далее мы импортируем обе реализации и сравним время их выполнения (листинг 15.42).

ПРИМЕЧАНИЕ

Нужно подчеркнуть, что даже при использовании MiniBatchKMeans мы добиваемся вычислительной эффективности только за счет того, что работаем с уменьшенными в размерах данными.

Листинг 15.42. Сравнение KMeans с MiniBatchKMeans

```
np.random.seed(0)
import time
from sklearn.cluster import KMeans, MiniBatchKMeans

k=20
times = []
for KMeans_class in [KMeans, MiniBatchKMeans]:
    start_time = time.time()
    KMeans_class(k).fit(shrunk_norm_matrix)
    times.append(time.time() - start_time)

running_time_ratio = times[0] / times[1]
print(f"Mini Batch K-means ran {running_time_ratio:.2f} times faster "
       "than regular K-means")

Mini Batch K-means ran 10.53 times faster than regular K-means
```

Вычисляет время выполнения каждой реализации алгоритма кластеризации

Выполнение time.time() возвращает текущее время в секундах

`MiniBatchKMeans` выполняется примерно в десять раз быстрее стандартного `KMeans`. Уменьшение времени выполнения обходится недорого — было доказано, что `MiniBatchKMeans` дает несколько менее качественные кластеры, чем `KMeans`. Однако сейчас нас беспокоит в первую очередь не качество кластеров, а определение подходящего *K* в диапазоне `range(1, 61)` при помощи метода локтя. Ускоренная реализация `MiniBatchKMeans` должна отлично сработать в качестве инструмента приблизительной оценки данного значения.

Далее мы сгенерируем график с помощью метода *K*-средних для мини-пакетов, а также добавим в этот график сетку, чтобы лучше определить потенциальные координаты локтя (листинг 15.43). Как было показано в главе 10, подобную сетку можно визуализировать вызовом `plt.grid(True)`. Наконец, нам нужно сравнить полученный локоть с официальным количеством категорий новостных групп. Для этого построим на графике вертикальную линию в точке *K*, равной 20 (рис. 15.1).

Листинг 15.43. Построение кривой локтя с помощью MiniBatchKMeans

```
np.random.seed(0)
import matplotlib.pyplot as plt

k_values = range(1, 61)
inertia_values = [MiniBatchKMeans(k).fit(shrunk_norm_matrix).inertia_
                  for k in k_values]
plt.plot(k_values, inertia_values)
plt.xlabel('K')
plt.ylabel('Inertia')
plt.axvline(20, c='k')
plt.grid(True)
plt.show()
```

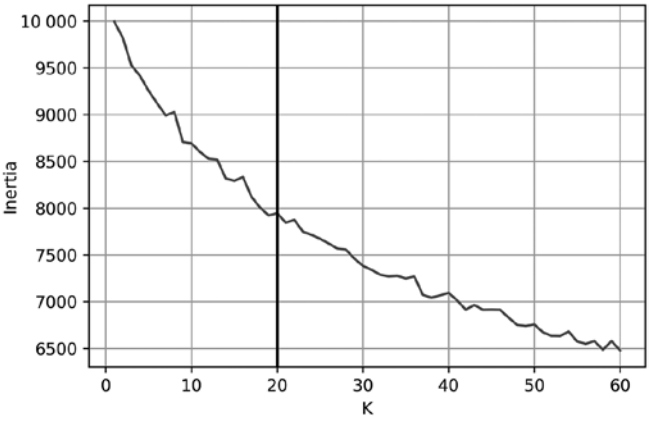


Рис. 15.1. График локтя, сгенерированный при помощи мини-пакетного метода K-средних для значений K от 1 до 61. Точное расположение локтя определить сложно. Однако уклон кривой заметно круче до точки K = 20. К тому же после этой точки она начинает уплощаться. На основании этого можно сделать вывод, что подходящим значением K будет примерно 20

Полученная кривая плавно снижается. Точное расположение изгиба локтя определить сложно. Мы видим, что эта кривая заметно круче, когда *K* меньше 20. Где-то после 20 кластеров она начинает уплощаться, но нет определенного места, где бы возникал резкий изгиб.

У нашего набора данных нет идеального *K*, при котором тексты разбивались бы на естественные кластеры. Почему? В первую очередь из-за того, что реальный текст беспорядочен и содержит множество нюансов. Категориальные границы в нем не всегда очевидны. К примеру, можно участвовать в беседе о технологиях или политике. Помимо этого, можно открыто обсуждать влияние технологии на политику. В итоге, казалось бы, различные темы дискуссии могут сливаться воедино, формируя новые темы. Ввиду этих сложностей между кластерами текстов

редко существует плавный переход. Следовательно, выяснить идеальное значение *K* трудно. Но можно сделать несколько полезных выводов: на основе графика локтя можно определить, что 20 — это адекватное приблизительное значение параметра *K*. Да, кривая нечеткая, и вполне возможно, что для него сгодится также 18 или 22. Как бы то ни было, нужно с чего-то начать, и именно 20 выглядит более разумным, чем 3 или 50. Наше решение неидеально, но приемлемо. Иногда при работе с реальными данными допустимое решение оказывается лучшим, на что можно рассчитывать.

ПРИМЕЧАНИЕ

Если вам неудобно выбирать точку локтя, разглядывая график, попробуйте использовать внешнюю библиотеку Yellowbrick. Она содержит класс KElbowVisualizer (http://mng.bz/7lV9), который для автоматического выделения локтя на графике задействует реализации мини-пакетного метода K-средних из Matplotlib и scikit-learn. Если мы используем KElbowVisualizer и применим его к данным, то соответствующий объект вернет K, равное 23. Помимо этого, Yellowbrick предлагает и более эффективные методы выбора значения K, такие как коэффициент силуэта, о котором упоминалось в главе 10. Установить эту библиотеку можно выполнением pip install yellowbrick.

Теперь разделим `shrunk_norm_matrix` на 20 кластеров (листинг 15.44). Сначала для большей точности мы выполним исходную реализацию `KMeans`, после чего для удобства анализа сохраним индексы текстов и ID кластеров в таблице Pandas.

Листинг 15.44. Кластеризация постов по 20 группам

```
np.random.seed(0)
cluster_model = KMeans(n_clusters=20)
clusters = cluster_model.fit_predict(shrunk_norm_matrix)
df = pd.DataFrame({'Index': range(clusters.size), 'Cluster': clusters})
```

Мы кластеризовали тексты и теперь готовы изучить содержимое полученных кластеров. Однако сначала нужно кратко рассмотреть одно важное следствие выполнения метода *K*-средних для больших матриц — на разных компьютерах итоговые кластеры могут несколько различаться, даже если выполнить `np.random.seed(0)`. Это расхождение вызывается тем, что разные машины округляют числа с плавающей точкой по-разному. Одни компьютеры округляют небольшие числа вверх, а другие — вниз. Обычно эта разница незаметна. К сожалению, в матрице размером 10 000 на 100 элементов даже небольшие различия могут повлиять на результат кластеризации. Метод *K*-средних недетерминированный, о чем мы говорили в главе 10, — он многими способами может сходиться к нескольким наборам равно допустимых кластеров. Таким образом, кластеры, полученные вами на своей машине, могут отличаться от приведенных в книге, но наблюдения и выводы должны быть аналогичными.

Учитывая все сказанное, приступим к анализу, начав с одного кластера и впоследствии проанализировав все одновременно.

15.5.1. Анализ одного кластера текстов

Один из наших 20 кластеров содержит пост об автомобиле, расположенный в `newsgroups.data` с индексом 0. Отделим и подсчитаем количество текстов, сгруппированных вместе с сообщением об автомобиле (листинг 15.45).

Листинг 15.45. Отделение кластера с обсуждениями машин

```
df_car = df[df.Cluster == clusters[0]]
cluster_size = df_car.shape[0]
print(f"{cluster_size} posts cluster together with the car-themed post "
       "at index 0")
```

```
393 posts cluster together with the car-themed post at index 0
```

ВНИМАНИЕ

Как только что говорилось, содержимое кластера на вашем локальном компьютере может несколько отличаться от рассматриваемого здесь. Общий его размер может минимально отклоняться от 393. Если так и будет, последующие несколько листингов могут давать иные результаты. Независимо от этих отличий у вас все равно должна быть возможность сделать на основе своих результатов аналогичные выводы.

Вместе с текстом об автомобиле с индексом 0 кластеризуются 393 других поста. Вероятно, они также посвящены обсуждению машин. Если так, то в случайно выбранном посте речь должна идти об автомобиле. Проверим это (листинг 15.46).

Листинг 15.46. Вывод случайного поста в кластере о машинах

```
np.random.seed(1)
def get_post_category(index):
    target_index = newsgroups.target[index]
    return newsgroups.target_names[target_index]

random_index = np.random.choice(df_car.Index.values)
post_category = get_post_category(random_index)

print(f"This post appeared in the {post_category} discussion group:\n")
print(newsgroups.data[random_index].replace('\n\n', '\n'))
```

Возвращает категорию поста, находящегося по индексу index. В текущей главе эта функция будет использоваться повторно

```
This post appeared in the rec.autos discussion group:

My wife and I looked at, and drove one last fall. This was a 1992 model. It was
WAYYYYYYYYY underpowered. I could not imagine driving it in the mountains here
in Colorado at anything approaching highway speeds. I have read that the new
1993 models have a newer, improved hp engine.
I'm quite serious that I laughed in the salesman face when he said "once it's
broken in it will feel more powerful". I had been used to driving a Jeep 4.0L
190hp engine. I believe the 92's Land Cruisers (Land Yachts) were 3.0L, the
sames as the 4Runner, which is also underpowered (in my own personal opinion).
They are big cars, very roomy, but nothing spectacular.
```

В этом случайном посте речь идет о модели Jeep. Он был размещен в группе `rec.autos`. Сколько же из почти 400 постов в кластере принадлежат к `rec.autos`? Сейчас выясним (листинг 15.47).

Листинг 15.47. Проверка принадлежности постов кластера к rec.autos

```
rec_autos_count = 0
for index in df_car.Index.values:
    if get_post_category(index) == 'rec.autos':
        rec_autos_count += 1

rec_autos_percent = 100 * rec_autos_count / cluster_size
print(f"{rec_autos_percent:.2f}% of posts within the cluster appeared "
      "in the rec.autos discussion group")
```

```
84.73% of posts within the cluster appeared in the rec.autos discussion group
```

В этом кластере 84 % постов относятся к `rec.autos`. Получается, что в данном кластере доминирует группа по обсуждению машин. А что насчет остальных 16 % постов? Попали ли они в кластер по ошибке или тоже относятся к теме автомобилей? Вскоре мы это узнаем. Отделим индексы постов в `df_car`, которые не принадлежат `rec.autos`. После этого мы выберем случайный индекс и выведем связанный с ним пост (листинг 15.48).

Листинг 15.48. Анализ поста, не относящегося к rec.autos

```
np.random.seed(1)
not_autos_indices = [index for index in df_car.Index.values
                     if get_post_category(index) != 'rec.autos']

random_index = np.random.choice(not_autos_indices)
post_category = get_post_category(random_index)

print(f"This post appeared in the {post_category} discussion group:\n")
print(newsgroups.data[random_index].replace('\n\n', '\n'))
```

```
This post appeared in the sci.electronics discussion group:

>The father of a friend of mine is a police officer in West Virginia. Not
>only is his word as a skilled observer good in court, but his skill as an
>observer has been tested to be more accurate than the radar gun in some
>cases . . ..  No foolin!  He can guess a car's speed to within 2-3mph just
>by watching it blow by - whether he's standing still or moving too! (Yes,
1) How was this testing done, and how many times? (Calibrated speedometer?)
2) It's not the "some cases" that worry me, it's the "other cases" :-)

They are big cars, very roomy, but nothing spectacular.
```

Этот случайный пост относится к группе по обсуждению электроники. В нем описывается использование радара для измерения скорости автомобиля. Тематически он посвящен машинам, значит, был кластеризован верно. А что насчет остальных

примерно 60 постов, попавших в список `not_autos_indices`? Как оценить их релевантность? Можно прочесть каждый по очереди, но это не масштабируемое решение. Эффективнее будет агрегировать их содержимое, отобразив ведущие слова всех постов. Мы ранжируем каждое слово, просуммировав его TF-IDF в каждом индексе `not_autos_indices`, после чего упорядочим эти слова на основе их агрегированного значения TF-IDF. Вывод топ-10 слов поможет нам понять, относится ли анализируемое содержание к машинам.

Далее мы определим функцию `rank_words_by_tfidf`, которая будет получать список индексов и ранжировать слова по этим индексам, используя ранее описанный подход. Ранжированные слова будут сохраняться в таблице Pandas для удобства отображения. Суммированные значения TF-IDF, применяемые для ранжирования слов, также будут сохранены в этой таблице. После определения функции мы выполним `rank_words_by_tfidf(not_autos_indices)` и выведем топ-10 результатов (листинг 15.49).

ПРИМЕЧАНИЕ

Располагая массивом indices, мы хотим агрегировать строки tfidf_np_matrix[indices]. Как уже говорилось, можно выполнить суммирование слов по всем строкам с помощью tfidf_np_matrix[indices].sum(axis=0). Кроме того, эту сумму можно сгенерировать, выполнив tfidf_matrix[indices].sum(axis=0), где tfidf_matrix является CSR-объектом SciPy. Суммирование по строкам разреженной CSR-матрицы вычислительно существенно быстрее, но эта операция вернет матрицу размером $1 \times n$, не являющуюся объектом NumPy. Этот результат нужно преобразовать в массив NumPy, выполнив np.asarray(tfidf_matrix[indices].sum(axis=0))[0].

Листинг 15.49. Ранжирование топ-10 слов с помощью TF-IDF

```
def rank_words_by_tfidf(indices, word_list=words):
    summed_tfidf = np.asarray(tfidf_matrix[indices].sum(axis=0))[0]
    data = {'Word': word_list,
            'Summed TFIDF': summed_tfidf}
    return pd.DataFrame(data).sort_values('Summed TFIDF', ascending=False)

df_ranked_words = rank_words_by_tfidf(not_autos_indices)
print(df_ranked_words[:10].to_string(index=False))
```

```
    Word  Summed  TFIDF
     car       8.026003
    cars       1.842831
   radar       1.408331
   radio       1.365664
     ham       1.273830
     com       1.164511
odometer       1.162576
   speed       1.145510
    just       1.144489
  writes       1.070528
```

Это суммирование равнозначно выполнению tfidf_np_matrix[indices].sum(axis=0). Такая упрощенная агрегация массива NumPy выполняется примерно за 1 с. Это вроде бы немного, но при повторении вычислений для 20 кластеров время выполнения составит уже 20 с. В свою очередь, суммирование по строкам разреженной матрицы происходит значительно быстрее

Двумя ведущими словами оказались *car* и *cars*.

ПРИМЕЧАНИЕ

Слово cars является множественной формой car. Можно объединить их на основе окончания s слова cars. Процесс сокращения множественной формы слова до его корневой формы называется стеммингом. Полезные функции для эффективного стемминга можно найти во внешней библиотеке Natural Language Toolkit (https://www.nltk.org).

В прочих строчках рейтинга мы видим слова *radar*, *odometer* и *speed*. Некоторые из этих терминов встречались в случайно выбранном посте из `sci.electronics`. Использование радара для измерения скорости автомобиля — типичная тема в текстах, представленных в списке `not_autos_indices`. А как эти связанные со скоростью ключевые слова сопоставляются с остальными постами в кластере про автомобили? Проверить это можно, передав `df_car.Index.values` в `rank_words_by_tfidf` (листинг 15.50).

Листинг 15.50. Ранжирование топ-10 слов в кластере по обсуждению машины

```
df_ranked_words = rank_words_by_tfidf(df_car.Index.values)
print(df_ranked_words[:10].to_string(index=False))
```

```
   Word  Summed  TFIDF
    car      47.824319
   cars      17.875903
 engine      10.947385
 dealer       8.416367
    com       7.902425
   just       7.303276
 writes       7.272754
    edu       7.216044
article       6.768039
   good       6.685494
```

Как правило, посты в кластере `df_car` крутятся вокруг автомобильных двигателей и дилеров. Однако в небольшой их части обсуждаются методы измерения скорости машин с помощью радара. Посты о радаре чаще всего относятся к новостной группе `sci.electronics`. Тем не менее в них действительно идет речь о машинах, а не о политике, программном обеспечении или медицине. Выходит, кластер `df_car` оказывается достоверным. Путем анализа ведущих ключевых слов мы смогли проверить кластер, не прочитывая каждый его пост.

Аналогичным образом можно использовать `rank_words_by_tfidf` для получения ведущих слов в каждом из 20 кластеров. Выявленные ключевые слова позволят понять тему этих кластеров. К сожалению, показывать в книге 20 разных таблиц слов будет нерационально — они займут слишком много места, добавив в нее лишние страницы. В качестве альтернативы можно визуализировать эти слова в виде изображений на одном согласованном графике. Далее мы узнаем, как визуализировать содержимое нескольких кластеров текстов.

15.6. ВИЗУАЛИЗАЦИЯ КЛАСТЕРОВ ТЕКСТОВ

Наша задача — визуализировать ранжированные ключевые слова для нескольких кластеров текстов. Вначале нужно разобраться с простой задачей: как визуализировать важные ключевые слова из одного кластера? В качестве одного из вариантов можно просто вывести их в порядке значимости. К сожалению, такая сортировка никак не отразит относительную значимость. К примеру, в таблице `df_ranked_words` за словом *cars* следует слово *engine*. Тем не менее суммированный показатель TF-IDF для *cars* равен 17,8, а для *engine* — 10,9. Таким образом, *cars* в контексте кластера текстов про автомобили примерно в 1,6 раза значительнее *engine*. Как же внести в визуализацию относительную значимость? Можно обозначить ее с помощью размера шрифта: использовать для *cars* шрифт размером 17,8, а для *engine* — размером 10,9. В таком случае слово *cars* будет в 1,6 раза больше, а значит, в 1,6 раза важнее. Естественно, размер шрифта 10,9 может оказаться слишком мал, чтобы было удобно читать. Тогда можно его увеличить, удвоив суммированные показатели значительности TF-IDF. Python не позволяет нам изменять размер шрифта непосредственно во время вывода. Однако это можно сделать при помощи Matplotlib-функции `plt.text`. Выполнение `plt.text(x, y, word, fontsize=z)` приведет к отображению слова по координатам `(x, y)` и установке размера шрифта равным `z`. Эта функция позволяет визуализировать слова в двухмерной сетке, где их размер устанавливается пропорционально значимости. Такой вид визуализации называется *облаком слов*. Давайте задействуем `plt.text` для генерации облака ведущих слов из `df_ranked_words`. Мы построим облако в виде сетки 5×5 слов (листинг 15.51; рис. 15.2). Размер шрифта каждого слова при этом будет равен его удвоенному показателю значимости.

Листинг 15.51. Построение облака слов с помощью Matplotlib

```
i = 0
for x_coord in np.arange(0, 1, .2):
    for y_coord in np.arange(0, 1, .2):
        word, significance = df_ranked_words.iloc[i].values
        plt.text(y_coord, x_coord, word, fontsize=2*significance)
        i += 1

plt.show()
```

В визуализации получилась мешанина. Большие слова вроде *car* занимают слишком много места. Они пересекаются с другими словами, делая изображение неразборчивым. Нужно построить облако более продуманно — без наложения слов. Удаление наложения построенных в двухмерном пространстве слов — непростая задача. К счастью, над ее решением до нас уже постарались создатели внешней библиотеки Wordcloud. Она способна генерировать облака слов визуально привлекательным образом. Далее мы ее установим, после чего импортируем и инициализируем класс `WordCloud` (листинг 15.52).

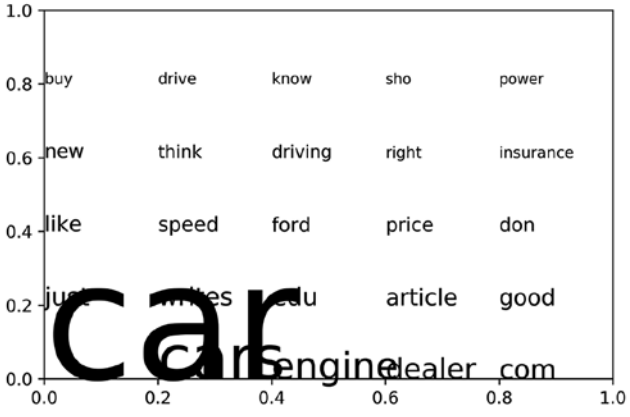


Рис. 15.2. Облако слов, сгенерированное при помощи Matplotlib. Из-за наложения слов выглядит оно несколько беспорядочно

ПРИМЕЧАНИЕ

Для установки Wordcloud выполните из терминала pip install wordcloud.

Листинг 15.52. Инициализация класса WordCloud

Точки расположения слов в облаке генерируются случайно. Для сохранения согласованности вывода необходимо передать случайное начальное значение с помощью параметра random_state

```
from wordcloud import WordCloud
cloud_generator = WordCloud(random_state=1)
```

Выполнение `WordCloud()` возвращает объект `cloud_generator`. Для генерации облака слов мы используем его метод `fit_words`. Выполнение `cloud_generator.fit_words(words_to_score)` приведет к созданию изображения на основе `words_to_score`, являющегося словарем, в котором слова сопоставлены с показателями их значимости.

ПРИМЕЧАНИЕ

Имейте в виду, что выполнение cloud_generator.generate_from_frequencies(word_to_score) даст тот же результат.

Далее мы создадим изображение из наиболее значимых слов в `df_ranked_words` (листинг 15.53). Сохраним его в переменной `wordcloud_image`, но пока отрисовывать не станем.

Листинг 15.53. Генерация изображения облака слов

```
words_to_score = {word: score
                  for word, score in df_ranked_words[:10].values}
wordcloud_image = cloud_generator.fit_words(words_to_score)
```

Теперь можно визуализировать `wordcloud_image`. Matplotlib-функция `plt.imshow` может отрисовывать изображения на основе различных входных форматов. Выполнение `plt.imshow(wordcloud_image)` приведет к отображению сгенерированного нами облака слов (листинг 15.54; рис. 15.3).

ПРИМЕЧАНИЕ

В Python есть несколько способов вывода изображений. Один из них подразумевает сохранение картинки в виде двухмерного массива NumPy. В качестве альтернативы можно сохранить его с помощью особого класса из Python Imaging Library (PLI). Функция plt.imshow может выводить изображения, хранящиеся в виде объектов NumPy или объектов PIL Image. Она может также отображать кастомные объекты изображений, включающие метод to_image, но вывод последнего должен возвращать массив NumPy или объект PIL Image.

Листинг 15.54. Построение изображения с помощью plt.imshow

```
plt.imshow(wordcloud_image)
plt.show()
```

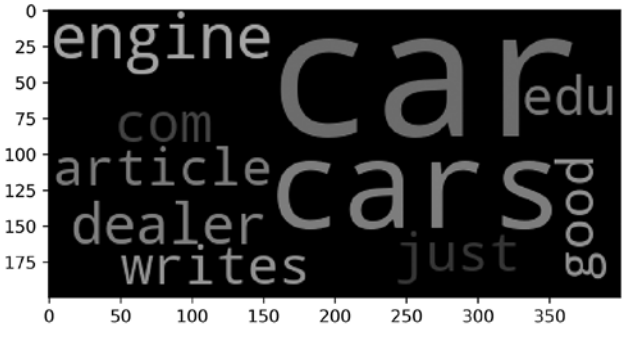


Рис. 15.3. Облако слов, сгенерированное с помощью класса WordCloud. Теперь слова не перекрывают друг друга. Однако фон здесь слишком темный, а края некоторых букв получились шероховатыми

Мы визуализировали облако слов, но изображение неидеально: темный фон затрудняет чтение слов. Для его изменения с черного на белый во время инициализации нужно выполнить `WordCloud(background_color='white')`. Кроме того, края некоторых букв выглядят пиксельными и угловатыми. Чтобы их сгладить, нужно передать в `plt.imshow` параметр `interpolation="bilinear"`. Давайте заново сгенерируем облако слов, теперь уже со светлым фоном и сглаживанием букв (листинг 15.55; рис. 15.4).

Листинг 15.55. Улучшение качества изображения облака слов

```
cloud_generator = WordCloud(background_color='white',
                            random_state=1)
wordcloud_image = cloud_generator.fit_words(words_to_score)
plt.imshow(wordcloud_image, interpolation="bilinear")
plt.show()
```

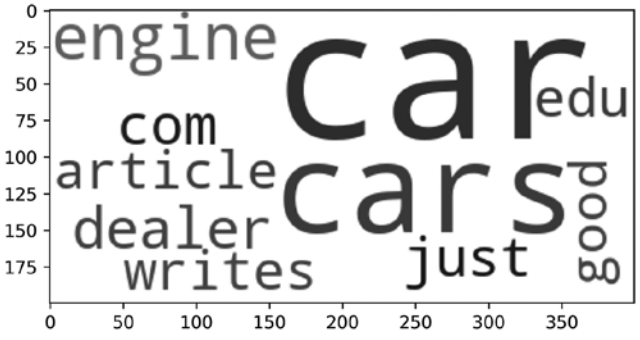


Рис. 15.4. Облако слов, сгенерированное с помощью класса WordCloud. Для лучшей видимости его фон задан белым, а края букв сглажены

Мы успешно визуализировали ведущие слова из кластера об автомобилях. Слова *car* и *cars* явно доминируют над менее значимыми терминами вроде *engine* и *dealer*. Теперь можно интерпретировать содержимое кластера, просто взглянув на облако слов. Естественно, мы уже хорошо его изучили и из этой визуализации ничего нового не почерпнем. Давайте лучше применим построение облака слов к случайно выбранному кластеру (листинг 15.56; рис. 15.5). Это облако отобразит 15 наиболее значимых слов кластера и позволит определить его основную тему.

ПРИМЕЧАНИЕ

Цвета слов в облаке генерируются случайно, и некоторые из них выглядели бы в черно-белой версии книги плохо. По этой причине мы специально ограничили выбор цветов до небольшого набора, используя параметр color_func из класса WorldCloud.

Листинг 15.56. Построение облака слов для случайного кластера

Получает на входе таблицу df_cluster и возвращает изображение облака слов для ведущих max_words слов кластера. Ранжирование слов выполняется с помощью ранее определенной функции rank_words_by_tfidf

Класс WordCloud включает опциональный параметр color_func. Он ожидает функцию выбора цвета, присваивающую каждому слову цвет. Здесь для управления настройками цвета мы определяем собственную функцию

```
np.random.seed(1)

def cluster_to_image(df_cluster, max_words=15):
    indices = df_cluster.Index.values
    df_ranked_words = rank_words_by_tfidf(indices)[:max_words]
    words_to_score = {word: score
                      for word, score in df_ranked_words[:max_words].values}
    cloud_generator = WordCloud(background_color='white',
                                color_func=_color_func,
                                random_state=1)
    wordcloud_image = cloud_generator.fit_words(words_to_score)
    return wordcloud_image

def _color_func(*args, **kwargs):
    return np.random.choice(['black', 'blue', 'teal', 'purple', 'brown'])
```

Вспомогательная функция для случайного присваивания каждому слову одного из пяти допустимых цветов

```
cluster_id = np.random.randint(0, 20)
df_random_cluster = df[df.Cluster == cluster_id]
wordcloud_image = cluster_to_image(df_random_cluster)
plt.imshow(wordcloud_image, interpolation="bilinear")
plt.show()
```

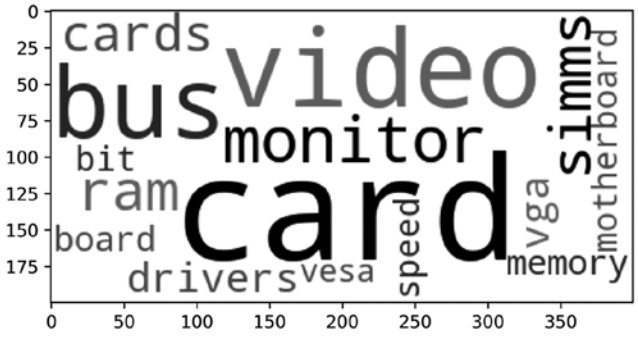


Рис. 15.5. Облако слов случайного кластера. Похоже, что его темами являются технологии и компьютерное железо

Наш случайно выбранный кластер включает ведущие слова, такие как *monitor*, *video*, *memory*, *card*, *motherboard*, *bit* и *ram*. Похоже, он связан с технологиями и компьютерным железом. Убедиться в этом можно с помощью вывода наиболее распространенной категории новостных групп в кластере (листинг 15.57).

ПРИМЕЧАНИЕ

Присутствие слов card, video и memory говорит о том, что card относится либо к video card, либо к memory card. В NLP подобные последовательности из нескольких слов называются биграммами. В целом последовательность из n слов называется N-граммой. TfidfVectorizer может выполнять векторизацию N-грамм произвольной длины. Для этого достаточно передать ему при инициализации параметр ngram_range. Выполнение TfidfVectorizer(ngram_range(1, 3)) приведет к созданию векторизатора, отслеживающего все 1-граммы (одинарные слова), 2-граммы (такие как video card) и 3-граммы (такие как natural language processing). Естественно, N-граммы приводят к увеличению словаря до миллионов записей. Но мы можем ограничить его размер до 100 000 ведущих N-грамм, передав в метод инициализации векторизатора параметр max_features=100000.

Листинг 15.57. Определение наиболее распространенной категории в кластере

```
from collections import Counter

def get_top_category(df_cluster):
    categories = [get_post_category(index)
                  for index in df_cluster.Index.values]
    top_category, _ = Counter(categories).most_common()[0]
    return top_category
```

```
top_category = get_top_category(df_random_cluster)
print("The posts within the cluster commonly appear in the "
      f"'{top_category}' newsgroup")
```

```
The posts within the cluster commonly appear in the
'comp.sys.ibm.pc.hardware' newsgroup
```

Многие посты кластера относятся к новостной группе `comp.sys.ibm.pc.hardware`. Значит, мы успешно определили, что темой кластера является аппаратное обеспечение. Для этого оказалось достаточно взглянуть на облако слов.

К этому моменту мы сгенерировали два отдельных облака слов для двух отдельных кластеров. Однако конечной целью является одновременное отображение нескольких облаков слов. Далее мы визуализируем все облака на одном рисунке, используя принцип Matplotlib под названием *«подграфик»*.

ТИПИЧНЫЕ МЕТОДЫ ДЛЯ ВИЗУАЛИЗАЦИИ СЛОВ

- `plt.text(word, x, y, fontsize=z)` — отображает слово по координатам `(x, y)`, размер шрифта `z`.
- `cloud_generator = WordCloud()` — инициализирует объект, способный сгенерировать облако слов. Фон облака будет черным.
- `cloud_generator = WordCloud(background_color='white')` — инициализирует объект, способный сгенерировать облако слов. Фон облака будет белым.
- `wordcloud_image = cloud_generator.fit_words(words_to_score)` — генерирует изображение облака слов на основе словаря `words_to_score`, в котором слова сопоставляются с показателями их значимости. Размер каждого слова в `wordcloud_image` вычисляется в соответствии с его значимостью.
- `plt.imshow(wordcloud_image)` — отрисовывает вычисленное `wordcloud_image`.
- `plt.imshow(wordcloud_image, interpolation="bilinear")` — отрисовывает вычисленное `wordcloud_image`, сглаживая буквы.

15.6.1. Использование подграфиков для визуализации нескольких облаков слов

Matplotlib позволяет включить несколько графиков в одно изображение. Каждый отдельный график называется *подграфиком*. Подграфики можно организовать несколькими способами, но чаще всего они упорядочиваются по сетке. Можно создать сетку подграфиков, содержащую *r* строк и *c* столбцов, выполнив `plt.subplots(r, c)`. Функция `plt.subplots` генерирует эту сетку, параллельно возвращая кортеж

(`figure, axes`). Переменная `figure` является особым классом, который отслеживает основное изображение, заключающее в себе сетку. В свою очередь, переменная `axes` — это двухмерный список, содержащий *r* строк и *c* столбцов. Каждый элемент `axes` представляет собой Matplotlib-объект `AxesSubplot`. Каждый объект подграфика можно использовать для вывода уникальной визуализации: выполнение `axes[i][j].plot(x, y)` строит график `x` относительно `y` в подграфике, расположенном в *i*-й строке и *j*-м столбце сетки.

ВНИМАНИЕ

Выполнение subplots(1, z) для subplots(z, 1) возвращает одномерный список axes, в котором len(axes) == z, а не двухмерной сетке.

Давайте рассмотрим, как используется `plt.subplots`. Мы сгенерируем сетку подграфиков размером 2×2, выполнив `plt.subplots(2, 2)`, после чего переберем в ней каждые строку *r* и столбец *c*. Для каждого уникального подграфика, расположенного в `(r, c)`, построим квадратичную кривую, у которой `y = r * x * x + c * x`. Связав параметры этой кривой с позицией сетки, сгенерируем четыре отдельные кривые, которые будут располагаться в границах одного рисунка (листинг 15.58; рис. 15.6).

Листинг 15.58. Генерация четырех подграфиков с помощью Matplotlib

```
figure, axes = plt.subplots(2, 2)
for r in range(2):
    for c in range(2):
        x = np.arange(0, 1, .2)
        y = r * x * x + c * x
        axes[r][c].plot(x, y)

plt.show()
```

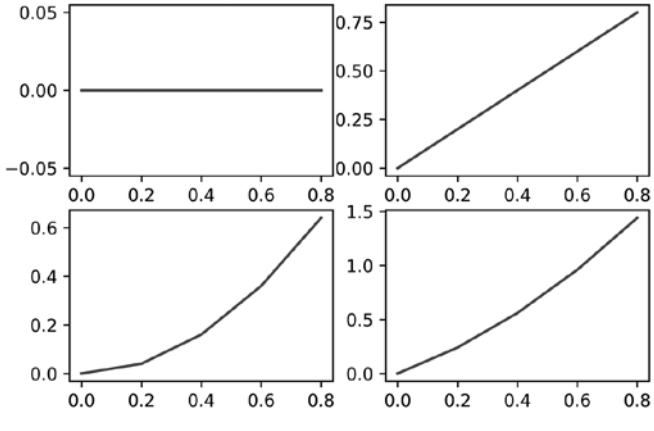


Рис. 15.6. Четыре кривые, отрисованные относительно четырех подграфиков на одном рисунке

В подграфиках нашей сетки отображены четыре кривые. Любую из них можно заменить облаком слов. Давайте визуализируем `wordcloud_image` в нижнем левом квадрате сетки, выполнив `axes[1][0].imshow(wordcloud_image)` (листинг 15.59; рис. 15.7). При этом мы дополнительно присвоим этому подграфику название: оно будет равно `top_category`, которой является `comp.sys.ibm.pc.hardware`. Название подграфика установим посредством выполнения `axes[r][c].set_title(top_category)`.

Листинг 15.59. Построение облака слов в подграфике

```
figure, axes = plt.subplots(2, 2)
for r in range(2):
    for c in range(2):
        if (r, c) == (1, 0):
            axes[r][c].set_title(top_category)
            axes[r][c].imshow(wordcloud_image,
                              interpolation="bilinear")
        else:
            x = np.arange(0, 1, .2)
            y = r * x * x + c * x
        axes[r][c].plot(x, y)

plt.show()
```

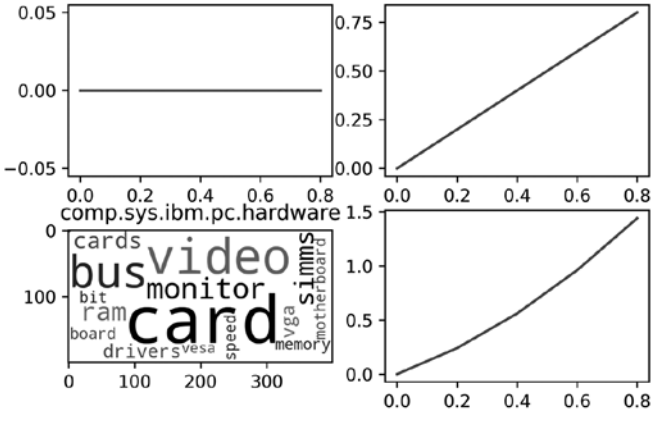


Рис. 15.7. Три кривые и облако слов отрисованы в четырех подграфиках. Из-за проблем с форматированием и размером рисунка читаемость облака слов страдает

Мы визуализировали облако слов в сетке подграфиков, но у его представления есть недостатки. Слова в облаке сложно читать, поскольку их подграфик слишком мал. Нужно его увеличить, для чего потребуется изменить размер рисунка. Сделать это можно с помощью параметра `figsize`. Передача `figsize=(width, height)` в `plt.subplots` приведет к созданию рисунка размером `width` дюймов в ширину

и `height` дюймов в высоту. В итоге каждый подграфик рисунка также подстроится под новый размер.

Помимо этого, график можно улучшить, внеся еще кое-какие изменения. Уменьшение количества визуализируемых слов с 15 до 10 упростит чтение уменьшенной версии облака слов.

Также нужно удалить из графика разметку осей, поскольку она занимает слишком много места, не неся никакой полезной информации. Для удаления разметки осей *X* и *Y* из `axis[r][c]` нужно вызвать `axis[r][c].set_xticks([])` и `axis[r][c].set_yticks([])` соответственно. С учетом этого мы сгенерируем изображение шириной 20 и высотой 15 дюймов (листинг 15.60).

Листинг 15.60. Визуализация всех кластеров с помощью 20 подграфиков

```
np.random.seed(0)

def get_title(df_cluster):
    top_category = get_top_category(df_cluster)
    cluster_id = df_cluster.Cluster.values[0]
    return f"{cluster_id}: {top_category}"

figure, axes = plt.subplots(5, 4, figsize=(20, 15))
cluster_groups = list(df.groupby('Cluster'))
for r in range(5):
    for c in range(4):
        _, df_cluster = cluster_groups.pop(0)
        wordcloud_image = cluster_to_image(df_cluster, max_words=10)
        ax = axes[r][c]
        ax.imshow(wordcloud_image,
                  interpolation="bilinear")
        ax.set_title(get_title(df_cluster), fontsize=20)
        ax.set_xticks([])
        ax.set_yticks([])

plt.show()
```

Генерирует название подграфиков, совмещая ID кластера с наиболее распространенной категорией в кластере

Увеличивает шрифт названия до 20 для лучшей читаемости

ПРИМЕЧАНИЕ

Фактические размеры изображения в книге составляют не 20×15 дюймов.

На большом изображении будут представлены 20 подграфиков, распределенных по сетке 5×4. Каждый из них содержит облако слов, соответствующее одному из наших кластеров, а в качестве названия каждого подграфика задана доминирующая в кластере новостная группа. Мы также включили во все названия индексы кластеров, чтобы в дальнейшем к ним обращаться. Наконец, мы удалили со всех графиков разметку осей. Итоговая визуализация дает общее представление всех паттернов доминирующих слов для всех 20 кластеров (рис. 15.8).

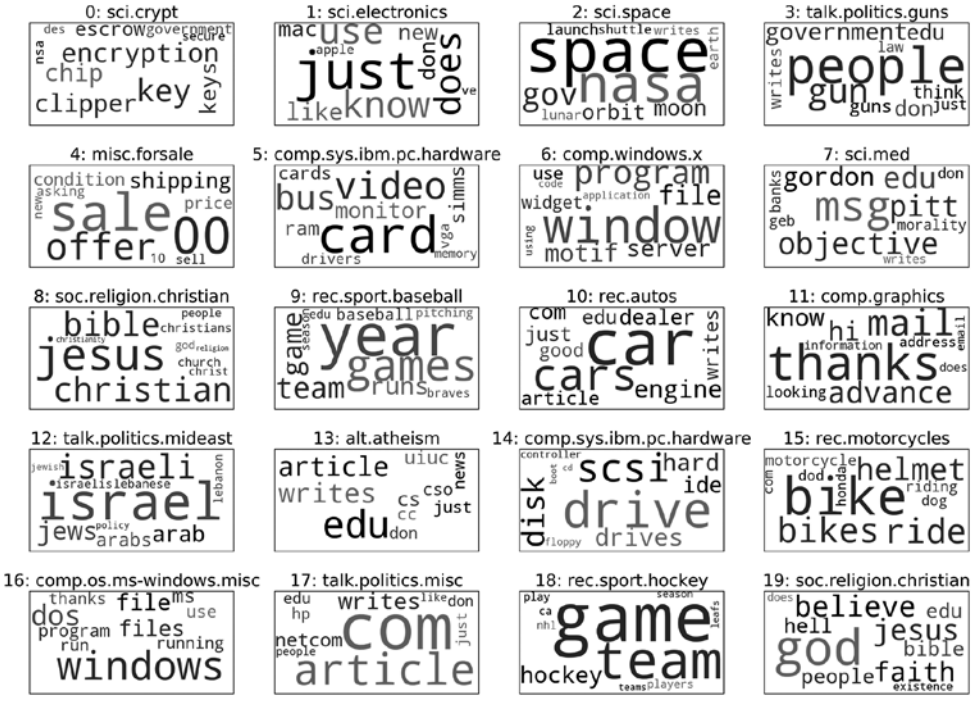


Рис. 15.8. Изображение 20 облаков слов, визуализированных в 20 подграфиках. Каждое облако соответствует одному из 20 кластеров. В качестве названия каждого подграфика установлена ведущая категория кластера. Название большинства облаков слов соответствует их отображаемому содержанию, однако некоторые облака (например, для кластеров 1 и 7) не предлагают информативного содержания

ТИПИЧНЫЕ МЕТОДЫ ДЛЯ РАБОТЫ С ПОДГРАФИКАМИ

- `figure, axes = plt.subplots(x, y)` — создает изображение, содержащее сетку подграфиков размером $x \times y$. Если $x > 1$ и $y > 1$, тогда `axes[r][c]` соответствует подграфику, расположенному в строке *r* и столбце *c* сетки.
- `figure, axes = plt.subplots(x, y, figsize=(width, height))` — создает изображение, содержащее сетку подграфиков размером $x \times y$. Это изображение `width` дюймов в ширину и `height` дюймов в высоту.
- `axes[r][c].plot(x_values, y_values)` — отрисовывает данные в подграфике, расположенном в строке *r* и столбце `c`.
- `axes[r][c].set_title(title)` — добавляет название в подграфик, расположенный в строке *r* и столбце *c*.

Мы визуализировали ведущие слова всех 20 кластеров. По большей части эти визуализации имеют смысл. Основной темой кластера 0 является криптография, так как его ведущие слова включают *encryption*, *secure*, *keys* и *nsa*. Во втором кластере основная тема касается космоса: среди его ведущих слов присутствуют *space*, *nasa*, *shuttle, moon* и *orbit*. Кластер 4 сосредоточен вокруг шопинга и содержит ведущие слова вроде *sale*, *offer*, *shipping* и *condition*. Кластеры 9 и 18 относятся к спорту: их основные темы — это *baseball* и *hockey* соответственно. В постах кластера 9 зачастую упоминаются *games*, *runs*, *baseball*, *pitching* и *team*. В кластере 18 посты содержат много слов вроде *game*, *team*, *players*, *hockey* и *nhl*. Тематику большинства кластеров на основе их облаков слов понять несложно. Примерно 75 % кластеров содержат ведущие слова, соответствующие доминирующим в них категориям.

Естественно, есть в нашем выводе и проблемы. Несколько облаков слов оказались бессмысленными: например, название подграфика в кластере 1 — `sci.electronics`, хотя его облако состоит из таких слов, как *just*, *like*, *does* и *know*. В то же время кластер 7 содержит подграфик с названием `sci.med`, хотя его облако слов состоит из слов вроде *pitt*, *msg* и *gordon*. К сожалению, визуализация облаков слов не всегда дает идеальный результат. Иногда определяющие ее кластеры сформированы неудачно либо доминирующий язык кластеров смещен в сторону неожиданных текстовых паттернов.

ПРИМЕЧАНИЕ

Прочитывание некоторых отобранных постов в кластерах поможет обнаружить смещения. К примеру, многие вопросы по электронике в кластере 1 содержат вопрос: «Does anyone know?» Кроме того, многие посты в кластере 7 были написаны студентом по имени Gordon, который обучался в University of Pittsburgh (pitt).

К счастью, есть ряд приемов, который позволит прояснить неопределенные облака слов. Например, можно отфильтровать очевидно бесполезные слова, а затем заново сгенерировать облако. Либо можно просто отбросить старшие x слов в кластере и визуализировать облако, используя следующие по рангу термины. Давайте перейдем к топ-10 слов из кластера 7 и еще раз вычислим их облако (листинг 15.61; рис. 15.9).

Листинг 15.61. Повторное вычисление облака слов после фильтрации

```
np.random.seed(3)
df_cluster= df[df.Cluster == 7]
df_ranked_words = rank_words_by_tfidf(df_cluster.Index.values)

words_to_score = {word: score
                  for word, score in df_ranked_words[10:25].values}
cloud_generator = WordCloud(background_color='white',
                            color_func=_color_func,
                            random_state=1)
wordcloud_image = cloud_generator.fit_words(words_to_score)
```

Мы визуализируем топ-15 слов, поскольку пространственно подграфиком не ограничены

```
plt.imshow(wordcloud_image, interpolation="bilinear")
plt.title(get_title(df_cluster), fontsize=20)
plt.xticks([])
plt.yticks([])
plt.show()
```

Заметьте, что в plt нет методов подграфика set_title, set_xticks и set_yticks. Вместо этого для достижения тех же результатов нам нужно вызвать plt.title, plt.xticks и plt.yticks

Рис. 15.9. Облако слов кластера 7, повторно вычисленное после фильтрации. Теперь видно, что основная тема этого кластера — медицина

В кластере 7 доминируют слова вроде *disease*, *medical*, *doctor*, *food*, *pain* и *patients*. Его медицинская тема теперь очевидна. Не принимая во внимание бесполезные слова, мы смогли определить истинное содержимое кластера. Естественно, этот простой подход сработает не всегда. NLP — запутанная наука, и здесь нет универсального средства, которое решило бы все наши проблемы. И тем не менее можно немалого достичь, даже несмотря на неструктурированную природу сложных текстов. Подумайте о том, чего мы смогли добиться: мы взяли 10 000 разнообразных реальных текстов и кластеризовали их на множество осмысленных тем. Более того, мы визуализировали эти темы в одном изображении, и интерпретировать большинство из них оказалось легко. Этих результатов мы добились, выполнив простую серию шагов, которые можно применить к любому огромному набору текстов. По сути, мы выработали конвейер для кластеризации и визуализации неструктурированных текстовых данных. Работает он так.

1. Трансформировать текст в нормализованную матрицу TF-IDF, используя класс `TfidfVectorizer`.
2. Уменьшить матрицу до 100 измерений с помощью алгоритма SVD.
3. Нормализовать уменьшенный в размере вывод для кластеризации.
4. Кластеризовать нормализованный вывод с помощью метода *K*-средних. Примерно прикинуть величину *K* можно, сгенерировав график локтя при помощи метода *K*-средних для мини-пакетов, который оптимизирован для скорости.

5. Визуализировать ведущие слова в каждом кластере, используя облако слов. Все облака слов отображаются в качестве подграфиков на одном рисунке. При этом слова ранжируются на основе их суммированных значений TF-IDF среди всех текстов кластера.
6. Интерпретировать тему каждого кластера, визуализировав облака слов. Любые неинтерпретируемые кластеры анализируются более подробно.

Имея такой конвейер для анализа текстов, можно эффективно кластеризовать и интерпретировать практически любой набор реальных текстов.

РЕЗЮМЕ

- Набор данных новостных групп из scikit-learn содержит более 10 000 постов, разделенных на 20 категорий.
- Эти посты можно преобразовать в матрицу TF, используя класс `CountVectorizer` из scikit-learn. Полученная матрица будет сохранена в формате *CSR*, применяемом для эффективного анализа разреженных матриц, состоящих преимущественно из нулей.
- Как правило, матрицы TF разреженные. Одна строка в них может ссылаться всего на несколько десятков слов словаря набора данных. Для обращения к ненулевым словам используется функция `np.flatnonzero`.
- Самые часто встречающиеся в тексте слова обычно оказываются *стоп-словами* — английскими словами вроде *the* или *this*. Стоп-слова необходимо убирать из наборов текстов перед векторизацией.
- Даже после отфильтровывания стоп-слов некоторые особо популярные слова все равно остаются. Минимизировать их влияние можно с помощью вычисления частоты их встречаемости в наборе документов (DF). DF слова равна общей доле текстов, в которых оно встречается. Более распространенные слова оказываются менее значимыми. Получается, что менее значимые слова имеют более высокую DF.
- Частотность термина и частоту его встречаемости в наборе документов можно совмещать в один показатель значимости, называемый *TF-IDF*. Как правило, векторы TF-IDF оказываются более информативными, чем векторы TF. Преобразовать тексты в векторы TF-IDF можно с помощью предоставляемого scikit-learn класса `TfidfVectorizer`. Этот векторизатор возвращает матрицу TF-IDF, чьи строки автоматически нормализуются для простоты вычисления сходства.
- Размерность больших матриц TF-IDF перед кластеризацией необходимо уменьшать. Рекомендуемое количество измерений — 100. Вывод SVD с уменьшенной размерностью перед последующим анализом необходимо нормализовать.

- Кластеризовать нормализованные текстовые данные с уменьшенной размерностью можно с помощью метода *K*-средних либо DBSCAN. Таким образом, *K*-средних остается предпочтительным алгоритмом кластеризации. Приблизительно оценить оптимальное значение *K* можно с помощью графика локтя. В случае больших наборов данных ради повышения скорости следует генерировать этот график, используя `MiniBatchKMeans`.
- Имея любой кластер текстов, нам необходимо определить слова, наиболее релевантные ему. Для этого можно ранжировать каждое слово суммированием его значений TF-IDF по всем строкам матрицы, представленным кластером. Более того, ранжированные слова можно визуализировать в виде *облака слов* — двухмерного, состоящего из слов изображения, в котором размер каждого слова пропорционален его значимости.
- Можно уместить множество облаков слов в одно изображение, используя функцию `plt.subplots`. Такая визуализация дает общий вид всех доминирующих паттернов слов среди всех кластеров.

16 Извлечение текстов с веб-страниц

В этой главе

- ✓ Отрисовка веб-страниц с помощью HTML.
- ✓ Базовая структура HTML-файлов.
- ✓ Извлечение текстов из HTML-файлов с помощью библиотеки Beautiful Soup.
- ✓ Скачивание HTML-файлов с онлайн-ресурсов.

Интернет является великим источником текстовых данных. Миллионы веб-страниц предлагают безграничные объемы текстового контента в форме новостных статей, страниц энциклопедии, научных работ, обзоров ресторанов, политических дискуссий, патентов, корпоративных финансовых заявлений, вакансий и т. д. Все эти страницы можно проанализировать, если скачать их файлы в формате языка разметки гипертекста (HTML). *Язык разметки* — это система для аннотации документов, отличающая эти аннотации от содержимого документа. В случае HTML аннотации являются инструкциями по визуализации веб-страницы.

Обычно визуализацию веб-контента выполняет браузер. Сначала он скачивает HTML-документ страницы на основе ее веб-адреса — URL. Затем парсит этот документ для получения инструкций по его отображению. Наконец, встроенный в браузер движок отрисовки форматирует и отображает все изображения и текст согласно тем самым указаниям разметки. В итоге отображенная страница оказывается пригодной для восприятия человеком.

Естественно, во время масштабного анализа данных нам не нужно отрисовывать каждую страницу. Компьютеры могут обращаться к текстам документов без визуализации. Поэтому при анализе HTML-документов можно сосредоточиться на тексте, пропустив инструкции по его отображению. И все же полностью игнорировать аннотации не следует, так как они предоставляют ценную информацию. Например, размеченный заголовок документа может вкратце обобщать его содержимое. Следовательно, отделение заголовка от основного абзаца может принести нам пользу. Если мы сможем различать отдельные части документа, то нам удастся провести более тщательный анализ. Так что при анализе онлайн-текстов базовое знание HTML-структуры необходимо. С учетом этого текущая глава начнется с обзора HTML-структуры, после чего мы научимся парсить ее при помощи библиотек Python.

ПРИМЕЧАНИЕ

Если вы уже знаете основы HTML, то можете сразу перейти к разделу 16.2.

16.1. СТРУКТУРА HTML-ДОКУМЕНТОВ

HTML-документ состоит из HTML-элементов. Каждый из них соответствует компоненту документа. К примеру, заголовок документа — это элемент, каждый абзац в нем тоже является элементом. Начало элемента обозначается открывающим тегом: например, открывающим тегом заголовка является `<title>`, а абзаца — `<p>`. Любой открывающий тег начинается и заканчивается угловыми скобками, `<>`. Добавление прямого слеша к тегу преобразует его в закрывающий. Закрывающими тегами обозначаются конечные точки большинства элементов, то есть непосредственный текст заголовка заканчивается `</title>`, а текст абзаца — `</p>`.

Вскоре мы изучим многие типичные HTML-теги, но сначала нужно познакомиться с самым важным — `<html>`, который указывает на начало всего HTML-документа. Используем этот тег для создания документа, состоящего всего из одного слова *Hello* (листинг 16.1). Содержимое документа мы сгенерируем инструкцией `html_contents = "<html>Hello</html>"`.

Листинг 16.1. Определение простой HTML-строки

```
html_contents = "<html>Hello</html>"
```

HTML-содержимое должно отображаться в браузере. Значит, `html_contents` можно визуализировать, сохранив его в файл и загрузив в предпочтительный браузер. В качестве альтернативы можно отобразить `html_contents` непосредственно в IPython Jupyter Notebook. Нужно будет просто импортировать HTML и `display` из `IPython.core.display`. Затем выполнение `display(HTML(html_contents))` приведет к отображению отрисованного вывода (листинг 16.2; рис. 16.1).

Мы отобразили свой HTML-документ, но пока он не особо впечатляет — его тело состоит всего из одного слова. Более того, у документа нет заголовка. Присвоим

ему заголовок, используя тег `<title>`. Установим в качестве него что-нибудь простое вроде *Data Science is Fun*. Для этого сначала создадим строку заголовка `"<title>Data Science is Fun</title>"` (листинг 16.3).

Листинг 16.2. Отрисовка HTML-строки

```
from IPython.core.display import display, HTML
def render(html_contents): display(HTML(html_contents))
render(html_contents)
```

Определяет однострочную функцию отрисовки для повторной визуализации HTML с помощью меньшего количества кода

Hello

Рис. 16.1. Отрисованный HTML-документ, содержащий одно слово Hello

Листинг 16.3. Определение заголовка в HTML

```
title = "<title>Data Science is Fun</title>"
```

Теперь мы заключим этот заголовок в `<html>` и `</html>`, выполнив `html_contents = f"<html>{title}Hello</html>"`, после чего отобразим обновленное содержание (листинг 16.4; рис. 16.2).

Листинг 16.4. Добавление заголовка к HTML-строке

```
html_contents = f"<html>{title}Hello</html>"
render(html_contents)
```

Hello

Рис. 16.2. Отрисованный HTML-документ. Его заголовок в выводе не отображается — мы видим только слово Hello

Вывод идентичен прежнему. Заголовок не появляется в теле отрисованного HTML, он показан только в строке заголовка браузера (рис. 16.3).

Рис. 16.3. Отображение HTML-документа в браузере. Заголовок показан лишь в строке заголовка браузера

Несмотря на свою частичную видимость, заголовок дает нам очень важную информацию — отражает общее содержание документа. К примеру, в объявлении о вакансии заголовок прямо сообщает о сути предполагаемой работы. Таким образом, несмотря на отсутствие в теле документа, он отражает необходимую информацию. Это важное различие обычно подчеркивается с помощью тегов `<head>` и `<body>`. Содержание, размеченное тегом `<body>`, будет отображаться в теле вывода. При этом тег `<head>` обозначает жизненно важную информацию, которая в теле не отображается. Далее мы подчеркнем это различие, заключив `title` в элемент `head` HTML-документа. А также заключим видимое слово *Hello* в элемент `body` содержания (листинг 16.5).

Листинг 16.5. Добавление в HTML-строку элементов head и body

```
head = f"<head>{title}</head>"
body = "<body>Hello</body>"
html_contents = f"<html> {title} {body}</html>"
```

Бывают случаи, когда нужно отобразить заголовок документа в теле страницы. К примеру, в объявлении о вакансии работодатель наверняка захочет показать название должности. Такой визуализированный заголовок называется *header* (шапка) и обозначается тегом `<h1>`. Естественно, этот тег заключается в `<body>`, где содержится все визуализируемое содержимое. Код листинга 16.6 добавляет шапку в тело нашего HTML (рис. 16.4).

Листинг 16.6. Добавление шапки в HTML-строку

```
header = "<h1>Data Science is Fun</h1>"
body = f"<body>{header}Hello</body>"
html_contents = f"<html> {title} {body}</html>"
render(html_contents)
```

HTML-элементы можно вкладывать друг в друга подобно матрешке. Здесь мы вкладываем элемент шапки в элемент тела, а тело и заголовок — в теги <html> и </html>

Рис. 16.4. Отрисованный HTML-документ. Здесь мы видим большую шапку

Единственное слово странно выглядит рядом с огромным заголовком. Как правило, HTML-документы содержат в своем теле более одного слова — обычно это множество предложений, разбитых на множество абзацев. Как уже говорилось, абзацы размечаются тегом `<p>`.

Добавим в наш HTML-документ два абзаца (листинг 16.7; рис. 16.5). Эти искусственные абзацы мы сформируем из предложений, состоящих из повторяющихся

слов: первый будет содержать слово *Paragraph 0*, повторяющееся 40 раз, а следующий — *Paragraph 1*.

Листинг 16.7. Добавление абзацев в HTML-строку

```
paragraphs = ''
for i in range(2):
    paragraph_string = f"Paragraph {i} " * 40
    paragraphs += f"<p>{paragraph_string}</p>"

body = f"<body>{header}{paragraphs}</body>"
html_contents = f"<html> {title} {body}</html>"
render(html_contents)
```

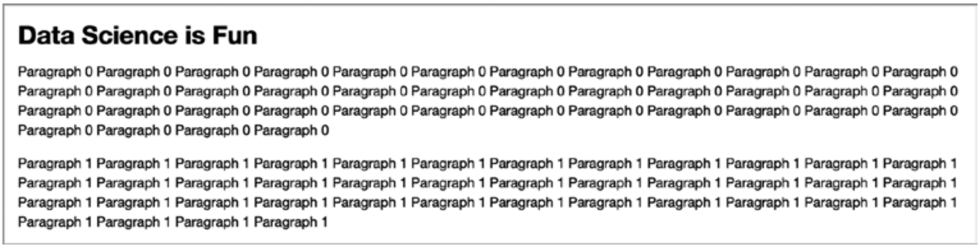


Рис. 16.5. Отрисованный HTML-документ. В нем два абзаца

Мы вставили в свой HTML-документ элементы абзаца. Текст в них различается, однако оба они заключены в одинаковые теги <p>. HTML-парсеру трудно проводить различие между первым и вторым тегами. Иногда есть смысл сделать различие между тегами более явно выраженным, в частности, если каждый абзац имеет уникальное форматирование. Провести такое различие между тегами <p> можно, присвоив каждому собственный ID, который вставляется прямо внутрь скобок тега. Например, мы обозначим первый абзац `paragraph 0`, написав <p id="paragraph 0">. Добавленный `id` называется *атрибутом* элемента абзаца. Атрибуты вставляются в открывающие теги элементов, позволяя отслеживать содержащуюся в них полезную информацию.

Код листинга 16.8 добавляет в теги абзацев атрибуты `id`, на основе которых мы их будем впоследствии различать.

Листинг 16.8. Добавление в абзацы атрибутов id

```
paragraphs = ''
for i in range(2):
    paragraph_string = f"Paragraph {i} " * 40
    attribute = f"id='paragraph {i}'"
    paragraphs += f"<p {attribute}>{paragraph_string}</p>"

body = f"<body>{header}{paragraphs}</body>"
html_contents = f"<html> {title} {body}</html>"
```

HTML-атрибуты играют множество значимых ролей. Особенно важны они при линковке документов. Интернет построен поверх *гиперссылок*, представляющих собой кликабельные тексты, связывающие веб-страницы. Щелчок на гиперссылке переносит вас в новый HTML-документ. Каждая гиперссылка обозначается тегом якоря `<a>`, который и делает ее текст кликабельным.

Тем не менее для указания адреса связанного документа необходима дополнительная информация. Предоставить ее можно с помощью атрибута `href`, означающего гипертекстовую ссылку. Например, разметка текста с использованием `<a href="https://www.manning.com">` привяжет этот текст к сайту Manning.

Далее мы создадим гиперссылку, которая будет выглядеть как *Data Science Bookcamp*, и привяжем ее кликабельный текст к сайту этой книги. Затем вставим гиперссылку в новый абзац, которому присвоим ID `paragraph 3` (листинг 16.9; рис. 16.6).

Листинг 16.9. Добавление гиперссылки в HTML-строку

```
link_text = "Data Science Bookcamp"
url = "https://www.manning.com/books/data-science-bookcamp"
hyperlink = f"<a href='{url}'>{link_text}</a>"
new_paragraph = f"<p id='paragraph 2'>Here is a link to {hyperlink}</p>"
paragraphs += new_paragraph
body = f"<body>{header}{paragraphs}</body>"
html_contents = f"<html> {title} {body}</html>"
render(html_contents)
```

Создает кликабельную гиперссылку. Щелчок на словах Data Science Bookcamp будет перенаправлять пользователя по URL-адресу с электронной версией книги

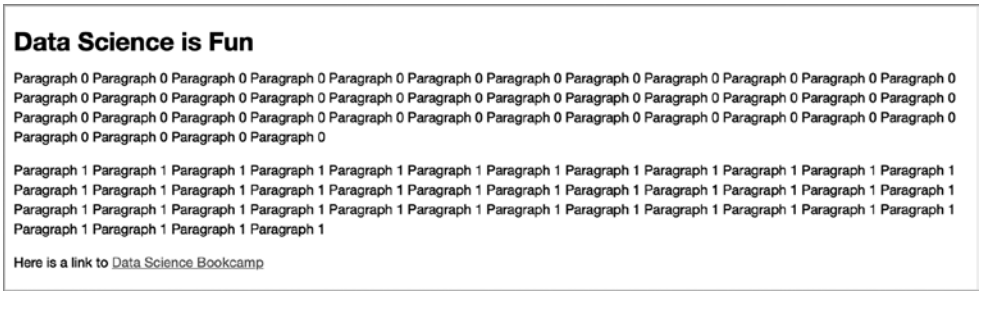


Рис. 16.6. Отрисованный HTML-документ. В него был добавлен еще один абзац, содержащий кликабельную ссылку на Data Science Bookcamp

HTML-элементы могут иметь разную сложность. Помимо шапок и абзацев, можно визуализировать также списки текстов. Предположим, что мы хотим отобразить список популярных библиотек для аналитики данных. Начнем с определения этого списка в Python (листинг 16.10).

Листинг 16.10. Определение списка библиотек по аналитике данных

```
libraries = ['NumPy', 'SciPy', 'Pandas', 'Scikit-Learn']
```

Теперь разметим каждый элемент тегом `<li>`, обозначающим *элемент списка* (листинг 16.11). Сохраним эти элементы в строке `items`.

Листинг 16.11. Разметка списка элементов тегом

```
items = ''
for library in libraries:
    items += f"<li>{library}</li>"
```

Наконец, вложим строку `items` в тег `<ul>`, в котором `ul` означает *неструктурированный список*. Затем внесем этот неструктурированный список в тело нашего документа. При этом мы также вставим между абзацами и списком вторую шапку (заголовок) *Common Data Science Libraries* (листинг 16.12). Для проведения различия между первым и вторым заголовками используем тег `<h2>` (рис. 16.7).

Листинг 16.12. Добавление неструктурированного списка в HTML-строку

```
unstructured_list = f"<ul>{items}</ul>"
header2 = '<h2>Common Data Science Libraries</h2>'
body = f"<body>{header}{paragraphs}{header2}{unstructured_list}</body>"
html_contents = f"<html> {title} {body}</html>"
render(html_contents)
```

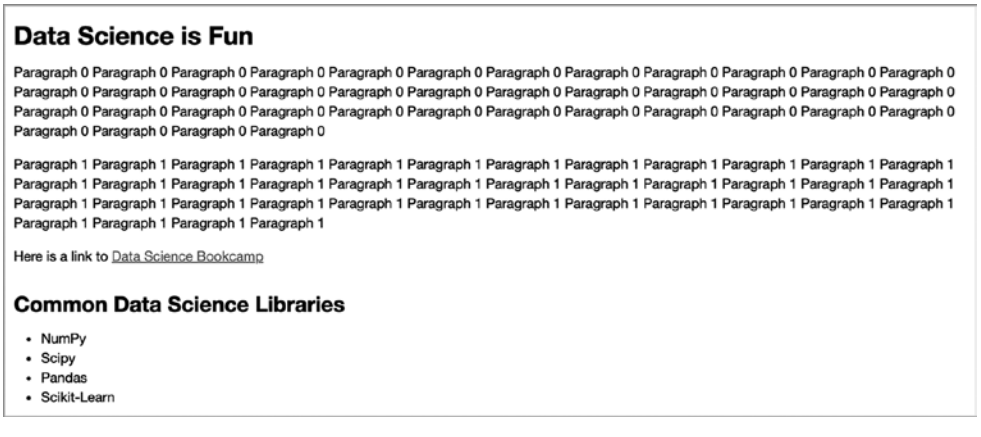


Рис. 16.7. Отрисованный HTML-документ. Обновленный документ содержит маркированный список распространенных библиотек для аналитики данных

Библиотеки были отображены в виде маркированного списка. Каждый его пункт занимает отдельную строку. Традиционно так обозначаются различные категории — от библиотек для анализа данных до требуемых навыков в объявлениях о вакансиях.

Здесь стоит отметить, что HTML-тело разделено на две части: первая соответствует тексту из трех абзацев, а вторая — маркированному списку. Как правило, подобное разделение выполняется при помощи специальных тегов `<div>`, которые позволяют инженерам фронтенда отслеживать отдельные элементы и соответствующим образом настраивать их форматирование. Обычно каждый тег `<div>` различается по некоему атрибуту. Если этот атрибут уникален для подраздела, то им выступает `id`. Если же он используется не в одном подразделе, то добавляется уточняющий `class`.

Ради согласованности мы отделим два наших раздела друг от друга, вложив их в два разных подраздела. Первый подраздел получит ID `paragraph`, а второй — ID `list`. Кроме того, поскольку оба подраздела содержат лишь текст, присвоим каждому атрибут класса `text`. Мы также добавим в тело третий, пустой подраздел, который обновим позднее. ID и класс пустого подраздела будут установлены как `empty` (листинг 16.13).

Листинг 16.13. Добавление подразделов в HTML-строку

```
div1 = f"<div id='paragraphs' class='text'>{paragraphs}</div>"
div2 = f"<div id='list' class='text'>{header2}{unstructured_list}</div>"
div3 = "<div id='empty' class='empty'></div>"
body = f"<body>{header}{div1}{div2}{div3}</body>"
html_contents = f"<html> {title}{body}</html>"
```

Третий подраздел пуст, но к нему все равно можно обращаться по классу и ID. Позднее мы вернемся к этому подразделу, чтобы добавить текст

ТИПИЧНЫЕ HTML-ЭЛЕМЕНТЫ И АТРИБУТЫ

- `<html>..</html>` — ограничивает весь HTML-документ.
- `<title>..</title>` — заголовок документа. Он отображается в браузерной строке заголовка, но не в отрисовываемом браузером содержимом документа.
- `<head>..</head>` — шапка документа. Информация шапки отображается в отрисовываемом браузером содержимом.
- `<body>..</body>` — тело документа. Информация, содержащаяся в нем, отображается в отрисовываемом браузером содержимом.
- `<h1>..</h1>` — шапка (заголовок) документа. Обычно отрисовывается большими жирными буквами.
- `<h2>..</h2>` — шапка (заголовок) документа. Ее форматирование слегка отличается от форматирования `<h1>`.
- `<p>..</p>` — отдельный абзац документа.
- `<p id="unique_id">..</p>` — отдельный абзац документа, содержащий уникальный атрибут `id`, который не используется никакими другими элементами документа.

- `<a href="http://www.manning.com">..</a>` — кликабельная текстовая гиперссылка. Щелчок на ее тексте перенаправляет пользователя по URL, указанному в атрибуте `href`.
- `<ul>..</ul>` — неструктурированный список, состоящий из отдельных элементов, отрисовываемых браузером в содержании документа в виде маркированных пунктов.
- `<li>..</li>` — отдельный элемент неструктурированного списка.
- `<div>..</div>` — теги, ограничивающие определенный подраздел документа.
- `<div class="category_class">..</div>` — теги, ограничивающие определенный подраздел документа. Этому подразделу с помощью атрибута `class` присваивается категория. В отличие от уникального ID, класс может использоваться не одним подразделом документа.

После внесения в строку `html_contents` всех этих изменений впору оценить, как теперь выглядит ее содержимое (листинг 16.14).

Листинг 16.14. Вывод измененной HTML-строки

```
print(html_contents)
```

```
<html> <title>Data Science is Fun</title><body><h1>Data Science is Fun</h1>
<div id='paragraphs' class='text'><p id='paragraph 0'>Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 </p><p id='paragraph 1'>Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 </p><p id='paragraph 2'>
Here is a link to <a href='https://www.manning.com/books/data-science-bookcamp'>
Data Science Bookcamp</a></p></div><div id='list' class='text'>
<h2>Common Data Science Libraries</h2><ul><li>NumPy</li>
<li>SciPy</li><li>Pandas</li><li>Scikit-Learn</li>
</ul></div><div id='empty' class='empty'></div></body></html>
```

Получился полный бардак! Содержимое документа почти невозможно прочитать. Кроме того, извлечь отдельные элементы из `html_contents` исключительно трудно. Представьте, что нам нужно извлечь заголовок документа. Для этого сначала

придется разделить `html_contents` по скобкам `>`, а затем перебрать все результаты этого разделения, остановившись на строке `<title`. Затем потребуется переместиться на один индекс далее и извлечь строку, содержащую текст заголовка. Наконец, нужно будет очистить строку заголовка, выполнив `split` по скобке `<`. Замысловатый процесс извлечения заголовка показан в листинге 16.15.

Листинг 16.15. Извлечение HTML-заголовка с помощью простого Python

```
split_contents = html_contents.split('>')
for i, substring in enumerate(split_contents):
    if substring.endswith('<title'):
        next_string = split_contents[i + 1]
        title = next_string.split('<')[0]
        print(title)
        break
```

Перебирает все подстроки, предшествующие >

Эта подстрока оканчивается открывающим тегом заголовка, значит, следующая подстрока и есть заголовок

```
Data Science is Fun
```

Есть ли более ясный способ извлечения элементов из HTML-документов? Да! Для этого не обязательно парсить документ вручную — можно использовать внешнюю библиотеку Beautiful Soup.

16.2. ПАРСИНГ HTML С ПОМОЩЬЮ BEAUTIFUL SOUP

Начнем с установки Beautiful Soup, после чего импортируем из `bs4` класс `BeautifulSoup` (листинг 16.16). Следуя традиционному соглашению, мы импортируем `BeautifulSoup` как `bs`.

ПРИМЕЧАНИЕ

Для установки библиотеки Beautiful Soup выполните из терминала pip install bs4.

Листинг 16.16. Импорт класса BeautifulSoup

```
from bs4 import BeautifulSoup as bs
```

Теперь мы инициализируем класс `BeautifulSoup`, выполнив `bs(html_contents)`. Следуя соглашению, присвоим инициализируемый объект переменной `soup` (листинг 16.17).

ПРИМЕЧАНИЕ

По умолчанию класс bs использует для извлечения HTML-содержимого встроенный в Python HTML-парсер. Однако внешние библиотеки предлагают более эффективные инструменты. Одна из таких популярных библиотек называется lxml, и установить ее можно, выполнив pip install lxml. После установки парсер этой библиотеки можно будет применять во время инициализации bs. Для этого достаточно выполнить bs(html_contents, 'lxml').

Листинг 16.17. Инициализация BeautifulSoup с помощью HTML-строки

```
soup = bs(html_contents)
```

Наш объект soup отслеживает все элементы в считанном HTML. Их можно вывести в чистом, читаемом формате, выполнив метод `soup.prettify()` (листинг 16.18).

Листинг 16.18. Вывод читаемого HTML с помощью Beautiful Soup

```
print(soup.prettify())
```

```
<html>
 <title>
  Data Science is Fun
 </title>
 <body>
  <h1>
   Data Science is Fun
  </h1>
  <div class="text" id="paragraphs">
   <p id="paragraph 0">
    Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
    Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
    Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
    Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
    Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
    Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
    Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
    </p>
    <p id="paragraph 1">
    Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
    Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
    Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
    Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
    Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
    Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
    Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
    </p>
    <p id="paragraph 2">
     Here is a link to
     <a href="https://www.manning.com/books/data-science-bookcamp">
      Data Science Bookcamp
     </a>
    </p>
   </div>
   <div class="text" id="list">
    <h2>
     Common Data Science Libraries
    </h2>
    <ul>
     <li>
      NumPy
     </li>
```

```
     <li>
      SciPy
     </li>
     <li>
      Pandas
     </li>
     <li>
      Scikit-Learn
     </li>
    </ul>
   </div>
  <div class="empty" id="empty">
  </div>
 </body>
</html>
```

Предположим, мы хотим обратиться к отдельному элементу, например к заголовку. Объект `soup` предоставляет такой доступ с помощью метода `find`. Выполнение `soup.find('title')` приведет к возвращению всего содержания, заключенного в открывающий и закрывающий теги заголовка (листинг 16.19).

Листинг 16.19. Извлечение заголовка с помощью Beautiful Soup

```
title = soup.find('title')
print(title)
```

```
<title>Data Science is Fun</title>
```

Полученный `title` оказывается HTML-строкой, ограниченной тегами заголовка. Однако переменная `title` не является строкой — это инициализированный Beautiful Soup класс `Tag`. Убедиться в этом можно с помощью вывода `type(title)` (листинг 16.20).

Листинг 16.20. Вывод типа данных заголовка

```
print(type(title))
```

```
<class 'bs4.element.Tag'>
```

Каждый объект `Tag` содержит атрибут `text`, который сопоставляется с текстом в теге. Таким образом, вывод `title.text` ведет к возвращению *Data Science is Fun* (листинг 16.21).

Листинг 16.21. Вывод атрибута text заголовка

```
print(title.text)
```

```
Data Science is Fun
```

Мы обратились к тегу `title`, выполнив `soup.find('title')`. Иначе к нему можно обратиться простым выполнением `soup.title`. Получается, что `soup.title.text` возвращает строку, равнозначную `title.text` (листинг 16.22).

Листинг 16.22. Обращение к атрибуту text заголовка из soup

```
assert soup.title.text == title.text
```

Аналогичным образом можно обратиться к телу документа, выполнив `soup.body`. Далее мы выведем весь текст из тела нашего HTML (листинг 16.23).

Листинг 16.23. Обращение к атрибуту text тела из soup

```
body = soup.body
print(body.text)
```

```
Data Science is FunParagraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Here is a link to Data Science BookcampCommon Data Science
        LibrariesNumPySciPyPandasScikit-Learn
```

В качестве вывода мы получили все тексты из тела в объединенном виде. В этот массив входят заголовки, содержимое списков и абзацы. Читать очень сложно. Вместо того чтобы выводить весь текст, следует сузить интересующую нас область. Давайте выведем текст только первого абзаца, используя `body.p.text` (листинг 16.24). Тот же результат можно получить с помощью `soup.p.text`.

Листинг 16.24. Обращение к тексту первого абзаца

```
assert body.p.text == soup.p.text
print(soup.p.text)
```

```
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
```

Обращаясь к `body.p`, мы возвращаем первый абзац `body`. А как обратиться к остальным двум? Для этого можно задействовать метод `find_all`. Выполнение `body.find_all('p')` приведет к возвращению списка всех тегов абзацев в тексте (листинг 16.25).

Листинг 16.25. Обращение ко всем абзацам в теле

```
paragraphs = body.find_all('p')
for i, paragraph in enumerate(paragraphs):
    print(f"\nPARAGRAPH {i}:")
    print(paragraph.text)
```

```
PARAGRAPH 0:
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0

PARAGRAPH 1:
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragrap

PARAGRAPH 2:
Here is a link to Data Science Bookcamp
```

Аналогичным образом мы обращаемся к маркированному списку посредством выполнения `body.find_all('li')`. Давайте используем `find_all` для вывода всех библиотек, перечисленных в маркированных списках тела (листинг 16.26).

Листинг 16.26. Обращение к маркированным спискам тела

```
print([bullet.text for bullet
       in body.find_all('li')])
```

```
['NumPy', 'Scipy', 'Pandas', 'Scikit-Learn']
```

Методы `find` и `find_all` позволяют искать элементы по типу тега и атрибуту. Предположим, мы хотим обратиться к элементу с уникальным ID `x`. Для поиска по ID нужно просто выполнить `find(id='x')`. С учетом этого выведем текст последнего абзаца с ID `paragraph_2` (листинг 16.27).

Листинг 16.27. Нахождение абзаца по ID

```
paragraph_2 = soup.find(id='paragraph 2')
print(paragraph_2.text)
```

```
Here is a link to Data Science Bookcamp
```

Содержимое `paragraph_2` включает веб-ссылку на *Data Science Bookcamp*. Фактический URL хранится в атрибуте `href`. Beautiful Soup позволяет обращаться к любому атрибуту, используя метод `get` (листинг 16.28). Таким образом, выполнение `paragraph_2.get('id')` вернет `paragraph 2`, а выполнение `paragraph_2.a.get('href')` вернет URL.

Листинг 16.28. Обращение к атрибуту в теге

```
assert paragraph_2.get('id') == 'paragraph 2'
print(paragraph_2.a.get('href'))

https://www.manning.com/books/data-science-bookcamp
```

Всем атрибутам ID в нашем HTML присвоены уникальные значения. Однако не все эти атрибуты уникальны. Например, два из трех элементов подразделов имеют атрибут `class`, установленный как `text`. При этом третий элемент подраздела содержит уникальный класс, установленный как `empty`. Выполнение `body.find_all('div')` вернет все три этих подразделяющих элемента. А как получить только те, чей класс установлен как `text`? Для этого достаточно выполнить `body.find_all('div', class_='text')`. Добавленный параметр `class_` ограничит результаты подразделами с соответствующим классом. Код листинга 16.29 находит эти подразделы и выводит их текстовое содержимое.

ПРИМЕЧАНИЕ

Почему мы выполняем find_all для class_, а не class? Дело в том, что в Python ключевое слово class является зарезервированным идентификатором, используемым для определения новых классов. Чтобы обойти это ограничение, в Beautiful Soup предоставляется особый параметр class_.

Листинг 16.29. Обращение к подразделам по их общему атрибуту class

```
for division in soup.find_all('div', class_='text'):
    id_ = division.get('id')

    print(f"\nDivision with id '{id_}':")
    print(division.text)

Division with id 'paragraphs':
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0
Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 0 Paragraph 1 Paragraph 1
```

```
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1 Paragraph 1
Paragraph 1 Paragraph 1 Here is a link to Data Science Bookcamp

Division with id 'list':
Common Data Science LibrariesNumPyScipyPandasScikit-Learn
```

К этому моменту мы использовали Beautiful Soup для обращения к элементам HTML. Эта библиотека позволяет также редактировать отдельные элементы. Например, при наличии объекта `tag` его можно удалить, выполнив `tag.decompose()`. Метод `decompose` удалит этот элемент из всех наших структур данных, включая `soup`. Таким образом, вызов `body.find (id='paragraph 0').decompose()` приведет к удалению всех следов этого абзаца. Кроме того, вызов `soup.find(id='paragraph 1').decompose()` приведет к удалению второго абзаца из объектов `soup` и `body`. После этих операций удаления останется лишь третий абзац. Убедимся в этом (листинг 16.30).

Листинг 16.30. Удаление абзацев с помощью Beautiful Soup

```
body.find(id='paragraph 0').decompose()
soup.find(id='paragraph 1').decompose()
print(body.find(id='paragraphs').text)

Here is a link to Data Science Bookcamp
```

Метод decompose удаляет абзац из всех вложенных объектов тега. Удаление абзаца из soup приводит также к его удалению из body и наоборот

Кроме того, в HTML можно вставлять теги. Предположим, что мы хотим вставить в последний, пустой подраздел новый абзац. Для этого сначала нужно создать новый элемент абзаца. Выполнение `soup.new_tag('p')` приведет к возвращению объекта `Tag` пустого абзаца (листинг 16.31).

Листинг 16.31. Инициализация Tag пустого абзаца

```
new_paragraph = soup.new_tag('p')
print(new_paragraph)

<p></p>
```

Затем нужно обновить текст созданного абзаца, присвоив его `new_paragraph.string` (листинг 16.32). Выполнение `new_paragraph.string = x` установит текст абзаца равным `x`.

Листинг 16.32. Обновление текста пустого абзаца

```
new_paragraph.string = "This paragraph is new"
print(new_paragraph)

<p>This paragraph is new</p>
```

Наконец, мы добавим обновленный `new_paragraph` к существующему объекту `Tag`. Имея два объекта `Tag`, `tag1` и `tag2`, можно вставить `tag1` в `tag2`, выполнив `tag2.append(tag1)`. Значит, выполнение `soup.find(id='empty').append(new_paragraph)` должно привести к добавлению абзаца в пустой подраздел (листинг 16.33). Давайте обновим наш HTML и подтвердим внесение изменений, отобразив результат (рис. 16.8).

Листинг 16.33. Вставка абзаца с помощью Beautiful Soup

```
soup.find(id='empty').append(new_paragraph) r
ender(soup.prettify())
```

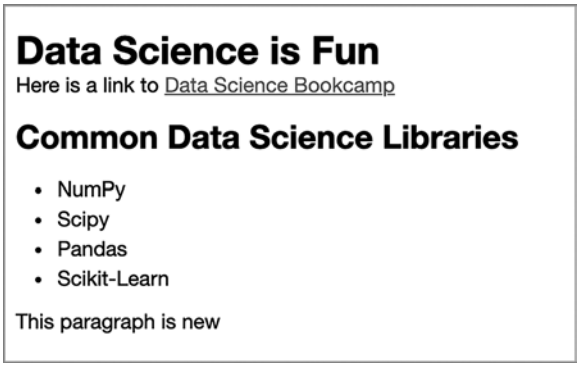


Рис. 16.8. Отрисованный HTML-документ. Он отредактирован — из него удалены два изначальных абзаца и добавлен один новый

ТИПИЧНЫЕ МЕТОДЫ BEAUTIFUL SOUP

- `soup = bs(html_contents)` — инициализирует объект `BeautifulSoup` из HTML-элементов в спарсенном `html_contents`.
- `soup.prettify()` — возвращает спарсенный HTML-документ в чистом, читаемом формате.
- `title = soup.title` — возвращает объект `Tag`, связанный с элементом заголовка спарсенного документа.
- `title = soup.find('title')` — возвращает объект `Tag`, связанный с элементом заголовка спарсенного документа.
- `tag_object = soup.find('element_tag')` — возвращает объект `Tag`, связанный с первым HTML-элементом, ограниченным указанным тегом `element_tag`.

- `tag_objects = soup.find_all('element_tag')` — возвращает список всех объектов `Tag`, ограниченных указанным тегом `element_tag`.
- `tag_object = soup.find(id='unique_id')` — возвращает объект `Tag`, содержащий указанный уникальный атрибут `id`.
- `tag_objects = soup.find_all('element_tag', class_='category_ class')` — возвращает список объектов `Tag`, ограниченных указанным тегом `element_tag` и содержащих указанный атрибут класса.
- `tag_object = soup.new_tag('element_tag')` — создает новый объект `Tag`, чей тип HTML-элемента указывается тегом `element`.
- `tag_object.decompose()` — удаляет объект `Tag` из `soup`.
- `tab_object.append(tag_object2)` — получая два объекта `Tag`, `tag_object` и `tag_object2`, вставляет `tag_object2` в `tag_object`.
- `tag_object.text` — возвращает весь видимый в объекте `Tag` текст.
- `tag_object.get('attribute')` — возвращает HTML-атрибут, присвоенный объекту `Tag`.

16.3. СКАЧИВАНИЕ И ПАРСИНГ ОНЛАЙН-ДАННЫХ

Библиотека Beautiful Soup позволяет с легкостью парсить, анализировать и редактировать HTML-документы. В большинстве случаев их необходимо скачивать напрямую из Интернета. Давайте вкратце рассмотрим выполнение этой процедуры с помощью встроенного в Python модуля `urllib`. Начнем с импорта функции `urlopen` из `urllib.request` (листинг 16.34).

ПРИМЕЧАНИЕ

Функции urlopen будет достаточно при скачивании одного HTML-документа с одной незащищенной онлайн-страницы. Для более сложных случаев следует рассмотреть применение внешней библиотеки Requests (https://requests.readthedocs.io).

Листинг 16.34. Импорт функции urlopen

```
from urllib.request import urlopen
```

Имея URL-адрес онлайн-документа, можно скачать его HTML-содержимое, выполнив `urlopen(url).read()`. Далее с помощью `urlopen` мы скачаем содержимое

сайта Manning, связанное с этой книгой. Затем выведем первые 1000 символов скачанного HTML.

ВНИМАНИЕ

Код листинга 16.35 сработает только при активном интернет-соединении. Кроме того, в результате корректировки сайта скачанный HTML может изменяться.

Листинг 16.35. Скачивание HTML-документа

Функция urlopen устанавливает сетевое соединение с указанным URL. Это соединение отслеживается с помощью специального объекта URLopener. Вызов метода read этого объекта приведет к скачиванию по установленному соединению текста

```
url = "https://www.manning.com/books/data-science-bookcamp"
html_contents = urlopen(url).read()
print(html_contents[:1000])
```

```
b'\n<!DOCTYPE html>\n<!--[if lt IE 7 ]> <html lang="en" class="no-js ie6
ie"> <![endif]-->\n<!--[if IE 7 ]><html lang="en" class="no-js ie7
ie"> <![endif]-->\n<!--[if IE 8 ]><html lang="en" class="no-js ie8
ie"> <![endif]-->\n<!--[if IE 9 ]><html lang="en" class="no-js ie9
ie"> <![endif]-->\n<!--[if (gt IE 9)|!(IE)]><!--> <html lang="en"
class="no-js"><!--<![endif]-->\n<head>\n
<title>Manning | Data Science Bookcamp</title>\n\n
<meta name="msapplication-TileColor" content=" #343434"/>\n
<meta name="msapplication-square70x70logo" content="/assets/favicon/windows-
    small-tile-6f6b7c9200a7af9169e488a11d13a7d3.png"/>\n
<meta name="msapplication-square150x150logo"
    content="/assets/favicon/windows-medium-tile-
    8fae4270fe3f1a6398f15015221501fb.png"/>\n
<meta name="msapplication-wide310x150logo" content="/assets/favicon/windows-
    wide-tile-a856d33fb5e508f52f09495e2f412453.png"/>\n
<meta name="msapplication-square310x310logo"
content="/assets/favicon/windows-large-tile-072d5381c2c83afa'
```

Далее при помощи Beautiful Soup извлечем из этого беспорядочного HTML-документа заголовок (листинг 16.36).

Листинг 16.36. Обращение к заголовку с помощью Beautiful Soup

```
soup = bs(html_contents)
print(soup.title.text)
```

```
Manning | Data Science Bookcamp
```

Используя объект soup, можно углубленно проанализировать страницу. Например, извлечь подраздел, содержащий шапку *about this book*, чтобы вывести описание этой книги (листинг 16.37).

ВНИМАНИЕ

Онлайн-HTML постоянно обновляется. Будущие изменения сайта Manning могут привести к нарушению работоспособности этого кода. Тем из вас, кто столкнется с подобными сложностями, рекомендуется проанализировать HTML вручную, чтобы извлечь описание книги.

Листинг 16.37. Обращение к описанию данной книги

```
for division in soup.find_all('div'):
    header = division.h2
    if header is None:
        continue

    if header.text.lower() == 'about the book':
        print(division.text)
```

Перебирает подразделы страницы

Проверяет наличие заголовка подраздела

После обнаружения раздела about выводит его содержимое

```
about the book

Data Science Bookcamp is a comprehensive set of challenging projects
carefully designed to grow your data science skills from novice to master.
Veteran data scientist Leonard Apeltsin sets five increasingly difficult
exercises that test your abilities against the kind of problems you'd encounter
in the real world. As you solve each challenge, you'll acquire and expand the
data science and Python skills you'll use as a professional data scientist.
Ranging from text processing to machine learning, each project comes complete
with a unique, downloadable data set and a fully explained step-by-step
solution. Because these projects come from Dr.
Apeltsin's vast experience, each solution highlights the most likely failure
points along with practical advice for getting past unexpected pitfalls. When
you wrap up these five awesome exercises, you'll have a diverse, relevant skill
set that's transferable to working in industry.
```

Теперь мы готовы использовать Beautiful Soup для парсинга вакансий в рамках нашего практического задания.

РЕЗЮМЕ

- HTML-документы состоят из вложенных элементов, которые предоставляют дополнительную информацию о тексте. Большинство таких элементов определяются открывающим и закрывающим тегами.
- Некоторые элементы подразумевают отображение содержащегося в них текста в браузере. Традиционно отображаемая информация вкладывается в элемент `body`. Прочие, неотображаемые, тексты, такие как заголовок документа, вкладываются в элемент `head`.

- Атрибуты можно вставлять в открывающие и закрывающие HTML-теги для отслеживания дополнительной информации. Теги одного типа помогают различать уникальные атрибуты `id`. Более того, для отслеживания элементов по категориям можно применять атрибуты `class`. В отличие от уникальных `id`, один атрибут `class` может использоваться несколькими элементами.
- Вручную извлекать тексты из HTML с помощью базовых возможностей Python сложно. К счастью, библиотека Beautiful Soup позволяет запрашивать элементы по типу тега и присвоенным значениям атрибутов. Она также дает возможность редактировать сам HTML.
- С помощью встроенной Python-функции `urlopen` можно скачивать HTML-файлы прямо из Сети. Затем их можно анализировать, используя Beautiful Soup.

17 Решение практического задания 4

В этой главе

- ✓ Парсинг текста из HTML.
- ✓ Вычисление коэффициента сходства текстов.
- ✓ Кластеризация и анализ больших текстовых наборов данных.

Мы скачали тысячи объявлений о вакансиях, использовав для поиска часть содержания этой книги, от практического задания 1 до практического задания 4 (подробности описаны в условии задачи). Помимо скачанных объявлений, в нашем распоряжении есть два текстовых файла: `resume.txt` и `table_of_contents.txt`. Первый содержит образец резюме, а второй — урезанную версию содержания, использованную для запроса соответствующих объявлений. Наша цель — определить по скачанным объявлениям типичные навыки аналитика данных. После этого мы сравним их с нашим резюме, чтобы понять, каких из них в нем недостает. Сделаем мы это по такому алгоритму.

1. Спарсим из скачанных HTML-файлов весь текст.
2. Проанализируем спарсенный вывод, чтобы понять, как описываются искомые навыки в онлайн-объявлениях. Особое внимание уделим тому, связаны ли определенные HTML-теги с описаниями навыков.
3. Отфильтруем из набора данных все неподходящие вакансии.
4. Кластеризуем навыки на основе сходства текстов.
5. Визуализируем кластеры с помощью облаков слов.

6. При необходимости скорректируем параметры кластеризации, чтобы улучшить визуализируемый вывод.
7. Сравним кластеризованные навыки с нашим резюме, чтобы определить недостающие.

ВНИМАНИЕ

Спойлер! Далее описывается решение для практического задания 4. Я настоятельно рекомендую попробовать решить эту задачу самим, прежде чем читать ее решение. Условие задачи можно найти в начале практического задания.

17.1. ИЗВЛЕЧЕНИЕ ТРЕБУЕМЫХ НАВЫКОВ ИЗ ОБЪЯВЛЕНИЙ О ВАКАНСИЯХ

Начнем со скачивания всех HTML-файлов в каталоге `job_postings`. Содержимое этих файлов мы сохраним в списке `html_contents`.

ВНИМАНИЕ

Прежде чем выполнять код листинга 17.1, не забудьте вручную распаковать каталог job_postings.zip.

Листинг 17.1. Скачивание HTML-файлов

Мы используем модуль glob из Python 3, чтобы получить имена файлов в каталоге job_postings с расширениями HTML. Эти имена файлов упорядочены с целью сохранения согласованности вывода на разных машинах. Так мы гарантируем, что первые два отобранных файла у всех читателей будут одинаковыми

```
import glob
html_contents = []

for file_name in sorted(glob.glob('job_postings/*.html')):
    with open(file_name, 'r') as f:
        html_contents.append(f.read())

print(f"We've loaded {len(html_contents)} HTML files.")
```

```
We've loaded 1458 HTML files.
```

Каждый из 1458 скачанных HTML-файлов можно спарсить при помощи Beautiful Soup. В листинге 17.2 мы выполним эту операцию и сохраним результаты в списке `soup_objects`. При этом убедимся, что каждый спарсенный HTML-файл содержит заголовок и тело.

Листинг 17.2. Парсинг HTML-файлов

```
from bs4 import BeautifulSoup as bs

soup_objects = []
for html in html_contents:
```

```
    soup = bs(html)
    assert soup.title is not None
    assert soup.body is not None
    soup_objects.append(soup)
```

Каждый спарсенный файл действительно имеет заголовок и тело. А есть ли среди заголовков и тел этих файлов какие-либо повторы? Выяснить это можно, упорядочив текст заголовка и тела в двух столбцах таблицы Pandas. Вызов Pandas метода `describe` покажет присутствие в текстах любых повторов (листинг 17.3).

Листинг 17.3. Проверка текстов тела и заголовка на наличие повторов

```
import pandas as pd
html_dict = {'Title': [], 'Body': []}

for soup in soup_objects:
    title = soup.find('title').text
    body = soup.find('body').text
    html_dict['Title'].append(title)
    html_dict['Body'].append(body)

df_jobs = pd.DataFrame(html_dict)
summary = df_jobs.describe()
print(summary)
```

```
Title  \
Count                            1458
unique                           1364
top     Data Scientist - New York, NY
freq                               13

                                                    Body
Count                                               1458
unique                                              1458
top     Data Scientist - New York, NY 10011\nAbout the...
freq                                                   1
```

В результате 1364 из 1458 заголовков оказались уникальными. Оставшиеся 94 — это повторы. Самый распространенный заголовок повторяется 13 раз, он связан с вакансией аналитика данных в Нью-Йорке. Можно без проблем определить, что все повторяющиеся заголовки соответствуют уникальному тексту тела. Все 1458 тел уникальны, значит, ни одно из объявлений не повторяется, даже если у некоторых одинаковые заголовки.

Мы убедились, что в HTML дубликатов нет. Теперь проанализируем его содержимое подробнее, чтобы определить, как в нем описываются требуемые навыки.

17.1.1. Анализ HTML на предмет описания навыков

Начнем анализ с отображения HTML с индексом 0 в `html_contents` (листинг 17.4; рис. 17.1).

Листинг 17.4. Отображение HTML первой вакансии

```
from IPython.core.display import display, HTML
assert len(set(html_contents)) == len(html_contents)
display(HTML(html_contents[0]))
```

Это вакансия на должность аналитика данных. Объявление начинается с краткого описания, из которого мы узнаем, что работа подразумевает извлечение полезной информации из правительственных данных. К необходимым навыкам относятся построение моделей, формирование статистики и визуализация.

Data Scientist - Beavercreek, OH

Data Scientist

Position Overview:

Centauri is looking for a detail oriented, motivated, and organized Data Scientist to work as part of a team to clean, analyze, and produce insightful reporting on government data. The ideal candidate is adept at using large data sets to find trends for intelligence reporting and will be proficient in process optimization and using models to test the effectiveness of different courses of action. They must have strong experience using a variety of data mining/data analysis methods, using a variety of data tools, building and implementing models, using/creating algorithms and producing easily understood visuals to represent findings. Candidate will work closely with Data Managers and stakeholders to tailor their analysis to answer key questions. The candidate must have a strong understanding of Geographic Information Systems (GIS) and statistical analysis.

Responsibilities:

- Use statistical research methods to analyze datasets produced through multiple sources of intelligence production
- Mine and analyze data from databases to answer key intelligence questions
- Assess the effectiveness and accuracy of new data sources and data gathering techniques
- Develop custom data models and algorithms to apply to data sets
- Use predictive modeling to produce reporting about future trends based on historical data
- Spatially analyze geographic data using GIS tools
- Visualize findings in easily understood graphics and aesthetically appealing finished reports

Qualifications for Data Scientist:

- Experience using statistical computer languages (R, Python, SLQ, etc.) to manipulate data and draw insights from large data sets
- Experience in basic visualization methods, especially using tools such as Tableau, ggplot, and matplotlib
- Knowledge of a variety of machine learning techniques (clustering, decision tree learning, artificial neural networks, etc.) and their real-world advantages/drawbacks
- Knowledge of advanced statistical techniques and concepts (regression, properties of distributions, statistical tests and proper usage, etc.) and experience with applications

Рис. 17.1. Отображенный HTML-документ первой вакансии. Первый абзац описывает предлагаемую должность аналитика данных. За ним следует маркированный список, перечисляющий необходимые навыки

Требуемые навыки дополнительно уточняются в двух подразделах, выделенных жирным: Responsibilities (Обязанности) и Qualifications (Опыт). Каждый подраздел — это список из нескольких пунктов, состоящих из одного предложения. Среди обязанностей перечисляются такие: применение статистических методов (п. 1),

выявление трендов (п. 5), пространственный анализ географических данных (п. 6) и эстетически привлекательная визуализация (п. 7). Требования подраздела Qualifications включают знание языков программирования R или Python (п. 1), инструментов визуализации, таких как Matplotlib (п. 2), приемов машинного обучения, включая кластеризацию (п. 3), и понимание продвинутых принципов статистики (п. 4).

Стоит сказать, что требования к опыту несильно отличаются от обязанностей. Да, они ориентированы на знание инструментов и принципов, в то время как обязанности больше связаны с реальной деятельностью в должности. Но в некотором смысле пункты этих подразделов взаимозаменяемы. Каждый пункт описывает навык, которым соискатель должен обладать, чтобы успешно справиться с работой. Значит, можно подразделить `html_contents[0]` на две принципиально разные части:

- первичное описание вакансии;
- список навыков, необходимых для ее получения.

Будет ли следующая вакансия структурирована аналогичным образом? Далее мы это выясним, отобразив `html_contents[1]` (листинг 17.5; рис. 17.2).

Листинг 17.5. Отрисовка HTML второй вакансии

```
display(HTML(html_contents[1]))
```

Data Scientist - Seattle, WA 98101

Are you interested in being a part of an Artificial Intelligence Marketing (AIM) company that is transforming how B2C enterprises engage with their customers; improving customer experience, marketing throughput and for the first time directly optimizing key business KPIs? Do you want to join a startup company backed by the top firms in the venture capital and SaaS industries? Would you like to be part of a company that prides itself on being a meritocracy, where passion, innovation, integrity, and our customers are at the heart of all that we do? Then, consider joining us at Amplero, an Artificial Intelligence Marketing company that leverages machine learning and multi-armed bandit experimentation to dynamically test thousands of permutations to adaptively optimize every customer interaction and maximize customer lifetime value and loyalty. We are growing our customer base and are looking for Data Scientists to join our innovative and energetic team! This is a unique opportunity to both drive innovations for our technology and to realize their impact as you work closely with our client engagement teams to best leverage our scientific capabilities within the Amplero product for marketing optimization and customer insights.

As an Amplero Data Scientist you would:

- Interface with our internal engagement teams and clients to understand business questions, and perform analytical "deep dives" to develop relevant and interpretive insights in support of our client engagements
- Smartly leverage appropriate technologies to answer tough questions or understand root causes of unexpected outcomes and statistical anomalies
- Develop analysis tools which will influence both our products and clients; including python pipelines focused on the productization of data science and insights tools for marketing performance and optimization
- Feature generation and selection from a wide variety of raw data types including time series and graphs
- Work with the Amplero Product Team to provide ongoing feedback to the features and priorities most aligned with our clients' current and future needs to inform the product roadmap, test product hypotheses as well as to help plan the product lifecycle

We'd love to hear from you if:

- You're an expert with data analysis and visualization tools including Python (including NumPy, SciPy, Pandas, scikit-learn) and other packages that enable data mining and machine learning
- You have a proven track record of applying data science to solve difficult real-world business problems
- You're familiar with areas of marketing data science where beyond-human scale, advanced experimentation and machine learning capabilities are used for achieving marketing performance, for example, DMP's in display advertising, Multivariate Testing, Statistical Significance Evaluation
- You've got excellent written and verbal communication skills for team and customer interactions - specifically, you're a genius at communicating results and the value of complex technical solutions to a non-technical audience

Рис. 17.2. Отображенный HTML-текст второй вакансии. Как и в рассмотренном объявлении, первый абзац дает общее описание, а далее список уточняет необходимые навыки

Перед нами вакансия аналитика данных в компании, занимающейся ИИ-маркетингом. Структура этого объявления аналогична построению предыдущего, из `html_contents[0]`: в первом абзаце приводится описание должности, а затем в виде списков перечисляются необходимые навыки. Эти навыки отличаются в плане технических требований и деталей. Например, в четвертом пункте снизу указано знание набора инструментов Python для аналитики данных (NumPy, SciPy, Pandas, scikit-learn), следующий пункт требует опыта решения сложных бизнес-задач, а в последнем говорится о необходимости превосходных письменных и вербальных навыков. Три перечисленных пункта очень разные, и такая разница показана намеренно — автор объявления подчеркивает различные требования, необходимые для получения должности. Выходит, что пункты перечней в `html_contents[0]` и `html_contents[1]` служат одной цели, предлагая краткие описания уникальных навыков, необходимых для каждой должности.

Встречаются ли эти типы перечисляемых в списке описаний в других объявлениях? Далее мы это выясним, а начнем с извлечения пунктов из каждого спарсенного HTML-файла. Напомню, что любой пункт представлен HTML-тегом `<li>`. Каждый файл с подобным списком будет содержать несколько таких тегов, значит, можно извлечь список пунктов из объекта `soup`, вызвав `soup.find_all('li')`. Далее мы переберем полученный список `soup_objects` и извлечем все пункты из каждого его элемента (листинг 17.6). Результат будет сохранен в столбце `Bullets` уже имеющейся у нас таблицы `df_jobs`.

Листинг 17.6. Извлечение пунктов из HTML

```
df_jobs['Bullets'] = [[bullet.text.strip()
                       for bullet in soup.find_all('li')]
                       for soup in soup_objects]
```

Убирает из каждого пункта разрывы строк, чтобы избежать их вывода в последующем анализе

Пункты из каждого объявления сохраняются в `df_jobs.Bullets`. Тем не менее есть вероятность, что некоторые или даже многие вакансии не будут содержать списков. Какой процент объявлений действительно их содержит? Код листинга 17.7 позволит это выяснить. Если этот процент окажется слишком мал, то дальнейший анализ пунктов не будет стоить затраченного времени.

Листинг 17.7. Измерение доли объявлений со списками

```
bulleted_post_count = 0
for bullet_list in df_jobs.Bullets:
    if bullet_list:
        bulleted_post_count += 1

percent_bulleted = 100 * bulleted_post_count / df_jobs.shape[0]
print(f"{percent_bulleted:.2f}% of the postings contain bullets")
```

```
90.53% of the postings contain bullets
```

Списки имеются в 90 % объявлений. Все ли они (или большинство) относятся к требуемым навыкам? Пока это неизвестно. Однако можно дополнительно уточнить содержание пунктов с помощью вывода ведущих слов их текстов. Мы ранжируем эти слова по количеству вхождений. В качестве альтернативы можно выполнить ранжирование, используя значения TF-IDF, а не просто количество. Как говорилось в главе 15, показатели TF-IDF с меньшей вероятностью будут содержать нерелевантные слова.

Код листинга 17.8 ранжирует слова на основе суммированных значений TF-IDF. Сначала он вычисляет матрицу TF-IDF, в которой строки соответствуют отдельным пунктам, после чего выполняет по этим строкам суммирование. Полученные суммы затем используются для ранжирования слов, соответствующих столбцам матрицы. Наконец, выполняется проверка, действительно ли полученные топ-5 слов связаны с навыками.

Листинг 17.8. Анализ ведущих слов в пунктах списков HTML-документа

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer

def rank_words(text_list):
    vectorizer = TfidfVectorizer(stop_words='english')
    tfidf_matrix = vectorizer.fit_transform(text_list).toarray()
    df = pd.DataFrame({'Words': vectorizer.get_feature_names(),
                       'Summed TFIDF': tfidf_matrix.sum(axis=0)})
    sorted_df = df.sort_values('Summed TFIDF', ascending=False)
    return sorted_df

all_bullets = []
for bullet_list in df_jobs.Bullets:
    all_bullets.extend(bullet_list)

sorted_df = rank_words(all_bullets)
print(sorted_df[:5].to_string(index=False))
```

```
     Words  Summed  TFIDF
experience     878.030398
      data     842.978780
    skills     440.780236
      work     371.684232
   ability     370.969638
```

Возвращает упорядоченную таблицу Pandas, содержащую ведущие слова

Эти слова упорядочиваются на основе их суммированных по строкам матрицы tfidf_matrix значений TF-IDF

Среди топ-5 слов встречаются термины вроде *skills* и *ability*, которые явно указывают на связь таких пунктов с отдельными навыками. А как эти слова сопоставляются с остальными терминами каждого объявления? Чтобы это выяснить, мы переберем тело каждого объявления и удалим все маркированные списки с помощью метода Beautiful Soup `decompose`. Затем извлечем оставшийся текст тела и сохраним его в списке `non_bullets`. В последнюю же очередь мы применим к этому списку функцию `rank_words` и отобразим топ-5 слов, не входящих в списки (листинг 17.9).

Листинг 17.9. Анализ ведущих слов в телах HTML-документов

```
non_bullets = []
for soup in soup_objects:
    body = soup.body
    for tag in body.find_all('li'):
        tag.decompose()

    non_bullets.append(body.text)

sorted_df = rank_words(non_bullets)
print(sorted_df[:5].to_string(index=False))
```

В качестве альтернативы вызов body.find_all('ul') даст такой же результат

```
     Words  Summed  TFIDF
      data      99.111312
      team      39.175041
      work      38.928948
experience      36.820836
  business      36.140488
```

Слов *skills* и *ability* теперь в ранжированном выводе нет. Их заменили слова *business* и *team*. Значит, текст вне списков в меньшей степени ориентирован на навыки, чем тот, из которого они сформированы. Однако все равно стоит отметить, что в `bullets` и `non_bullets` есть определенные общие слова: *data*, *experience* и *work*. Странно, но термины *scientist* и *science* в списке отсутствуют. Может, некоторые объявления относятся к связанным с данными вакансиям, которые не соотносятся напрямую с аналитикой? Нужно изучить эту возможность (листинг 17.10). Мы начнем с перебора всех заголовков во всех вакансиях и проверим, упоминается ли в них должность аналитика данных. Затем определим процент вакансий, в заголовках которых термины *data science* и *data scientist* отсутствуют. И наконец, выведем выборку из десяти таких заголовков с целью общей оценки.

ПРИМЕЧАНИЕ

Как говорилось в главе 11, мы сопоставляем термины с текстом заголовков, используя регулярные выражения.

Листинг 17.10. Проверка упоминаний в заголовках должности аналитика данных

```
regex = r'Data Scien(ce|tist)'
df_non_ds_jobs = df_jobs[~df_jobs.Title.str.contains(regex, case=False)]

percent_non_ds = 100 * df_non_ds_jobs.shape[0] / df_jobs.shape[0]
print(f"{percent_non_ds:.2f}% of the job posting titles do not mention a "
      "data science position. Below is a sample of such titles:\n")

for title in df_non_ds_jobs.Title[:10]:
    print(title)
```

Pandas-метод str.contains может сопоставлять регулярное выражение с текстом столбца. Передача в него case=False отключает при сопоставлении чувствительность к регистру

```
64.81% of the job posting titles do not mention a data science position. Below
is a sample of such titles:
```

```
Patient Care Assistant / PCA - Med/Surg (Fayette, AL) - Fayette, AL
Data Manager / Analyst - Oakland, CA
Scientific Programmer - Berkeley, CA
JD Digits - AI Lab Research Intern - Mountain View, CA
Operations and Technology Summer 2020 Internship-West Coast - Universal City, CA
Data and Reporting Analyst - Olympia, WA 98501
Senior Manager Advanced Analytics - Walmart Media Group - San Bruno, CA
Data Specialist, Product Support Operations - Sunnyvale, CA
Deep Learning Engineer - Westlake, TX
Research Intern, 2020 - San Francisco, CA 94105
```

Почти в 65 % заголовков объявлений должность аналитика данных не упоминается. Однако на основе нашей выборки можно выявить другие термины, подходящие для описания подобных должностей. Объявления могут обращаться к *data specialist*, *data analyst* или *scientific programmer*. Более того, в определенных объявлениях говорится об исследовательской преддипломной практике, что тоже можно счесть относящимся к данным. Однако не все отобранные должности полностью соответствуют сути: в нескольких объявлениях указаны руководящие вакансии, которые не соотносятся с нашими карьерными целями. Нужно подумать об исключении вакансий управленцев из анализа.

Более серьезная проблема заключается в том, что первое объявление списка относится к *Patient Care Assistant (PCA)*. Очевидно, что оно закралось в него ошибочно. Возможно, алгоритм спутал заголовок вакансии с приемом сокращения данных под названием PCA. Это ошибочное объявление содержит навыки, которых у нас нет и которые мы не хотим получать. Они представляют опасность для нашего анализа и будут выступать источником шума, если их не удалить. Продемонстрировать эту опасность можно путем вывода первых пяти пунктов `df_non_ds_jobs[0]` (листинг 17.11).

Листинг 17.11. Выборка пунктов списка из вакансии, не связанной с аналитикой данных

```
bullets = df_non_ds_jobs.Bullets.iloc[0]
for i, bullet in enumerate(bullets[:5]):
    print(f"{i}: {bullet.strip()}")
```

```
0: Provides all personal care services in accordance with the plan of treatment
assigned by the registered nurse
1: Accurately documents care provided
2: Applies safety principles and proper body mechanics to the performance of
specific techniques of personal and supportive care, such as ambulation of
patients, transferring patients, assisting with normal range of motions and
positioning
3: Participates in economical utilization of supplies and ensures that equipment
and nursing units are maintained in a clean, safe manner
4: Routinely follows and adheres to all policies and procedures
```

Мы специалисты по работе с данными, и наша основная задача не связана с заботой о пациентах (индекс 0) или обслуживанием санитарного оборудования (индекс 4).

Нужно удалить эти навыки из набора данных, но как? Один из вариантов — использовать сходство текстов. Можно сравнить объявления с нашим резюме и удалить вакансии, которые не соответствуют его содержанию. Помимо этого, в качестве дополнительного сигнала стоит сопоставить объявления с содержанием книги. По сути, нам нужно оценить релевантность каждой вакансии относительно как нашего резюме, так и материалов книги. Это позволит отфильтровать ошибочные объявления, сохранив лишь актуальные.

В качестве альтернативы можно отфильтровать отдельные навыки, описываемые в пунктах списков. По существу, это подразумевает ранжирование отдельных пунктов, а не вакансий. Но тут есть проблема. Представьте: если мы отфильтруем все пункты, которые не соответствуют нашему резюме или содержанию книги, то оставшиеся будут охватывать навыки, которыми мы уже обладаем. Такой исход противоречит нашей цели обнаружения недостающих навыков на основе релевантных объявлений о вакансиях аналитиков данных. Вместо этого следует реализовать нашу цель так.

1. Определить релевантные объявления, которые соответствуют нашему набору навыков частично.
2. Проанализировать, каких пунктов объявлений недостает в наборе наших навыков.

Опираясь на эту стратегию, далее мы отфильтруем вакансии по релевантности.

17.2. ФИЛЬТРАЦИЯ ВАКАНСИЙ ПО РЕЛЕВАНТНОСТИ

Наша задача — оценить релевантность вакансий, используя сопоставление текстов. Мы хотим сравнить текст каждого объявления со своим резюме и/или содержанием книги. Для начала сохраним резюме в строке `resume` (листинг 17.12).

Листинг 17.12. Загрузка резюме

```
resume = open('resume.txt', 'r').read()
print(resume)

Experience

1. Developed probability simulations using NumPy.
2. Assessed online ad-clicks for statistical significance using Permutation
testing.
3. Analyzed disease outbreaks using common clustering algorithms.

Additional Skills

1. Data visualization using Matplotlib.
2. Statistical analysis using SciPy.
```

```
3. Processing structured tables using Pandas.
4. Executing K-Means clustering and DBSCAN clustering using Scikit-Learn.
5. Extracting locations from text using GeonamesCache.
6. Location analysis and visualization using GeonamesCache and Cartopy.
7. Dimensionality reduction with PCA and SVD, using Scikit-Learn.
8. NLP analysis and text topic detection using Scikit-Learn.
```

Аналогичным образом можно сохранить содержание в строке `table_of_contents` (листинг 17.13).

Листинг 17.13. Загрузка содержания

```
table_of_contents = open('table_of_contents.txt', 'r').read()
```

Вместе `resume` и `table_of_contents` обобщают имеющийся у нас набор навыков. Далее внесем эти навыки в строку `existing_skills` (листинг 17.14).

Листинг 17.14. Объединение навыков в одной строке

```
existing_skills = resume + table_of_contents
```

Нам нужно вычислить сходство между каждым объявлением и своими навыками. Иными словами, мы хотим определить все, в чем схожи между собой `df_jobs.Body` и `existing_skills`. Для этого сначала потребуется векторизовать все тексты. Причем нужно векторизовать `df_jobs.Body` вместе с `existing_skills`, чтобы все векторы использовали один словарь. Далее мы совместим строку объявлений и строку навыков в один список текстов и векторизуем их с помощью реализации `TfidfVectorizer` из scikit-learn (листинг 17.15).

Листинг 17.15. Векторизация навыков и данных объявлений

```
text_list = df_jobs.Body.values.tolist() + [existing_skills]
vectorizer = TfidfVectorizer(stop_words='english')
tfidf_matrix = vectorizer.fit_transform(text_list).toarray()
```

Векторизованные тексты сохранены в матрице `tfidf_matrix`. Последняя ее строка (`tfidf_matrix[-1]`) соответствует набору наших навыков, а все остальные — объявлениям. Таким образом, можно легко вычислить косинусные коэффициенты между вакансиями и `existing_skills`, выполнив `tfdf_matrix[:-1]@tfidf_matrix[-1]`, — умножение матрицы на вектор вернет массив косинусных коэффициентов. Код листинга 17.16 вычисляет массив `cosine_similarities`.

ПРИМЕЧАНИЕ

Вам интересно, есть ли смысл визуализировать рейтинг слов? Ответ — да! Мы вскоре построим это распределение для получения ценной информации, но сначала нужно выполнить простую проверку разумности с помощью вывода ведущих заголовков вакансий. Это действие подтвердит корректность нашего предположения, что все выведенные вакансии релевантны.

Листинг 17.16. Вычисление косинусных коэффициентов для навыков

```
cosine_similarities = tfidf_matrix[:-1] @ tfidf_matrix[-1]
```

Полученные косинусные коэффициенты отражают текстовое сходство между нашими навыками и навыками в вакансиях. Вакансии с бóльшим сходством являются более подходящими и наоборот. Теперь с помощью этих коэффициентов можно ранжировать вакансии по релевантности, чем мы далее и займемся. Для начала нужно будет сохранить коэффициенты в столбце `Relevance` таблицы `df_jobs`. Затем мы упорядочим содержимое этой таблицы по `df_jobs.Relevance` по убыванию. И наконец, выведем 20 наименее релевантных вакансий и проверим, связаны ли они с наукой о данных (листинг 17.17).

Листинг 17.17. Вывод 20 наименее релевантных вакансий

```
df_jobs['Relevance'] = cosine_similarities
sorted_df_jobs = df_jobs.sort_values('Relevance', ascending=False)
for title in sorted_df_jobs[-20:].Title:
    print(title)
```

```
Data Analyst Internship (8 month minimum) - San Francisco, CA
Leadership and Advocacy Coordinator - Oakland, CA 94607
Finance Consultant - Audi Palo Alto - Palo Alto, CA
RN - Hattiesburg, MS
Configuration Management Specialist - Dahlgren, VA
Deal Desk Analyst - Mountain View, CA
Dev Ops Engineer AWS - Rockville, MD
Web Development Teaching Assistant - UC Berkeley (Berkeley) - Berkeley, CA
Scorekeeper - Oakland, CA 94612
Direct Care - All Experience Levels (CNA, HHA, PCA Welcome) - Norwell, MA 02061
Director of Marketing - Cambridge, MA
Certified Strength and Conditioning Specialist - United States
PCA - PCU Full Time - Festus, MO 63028
Performance Improvement Consultant - Los Angeles, CA
Patient Services Rep II - Oakland, CA
Lab Researcher I - Richmond, CA
Part-time instructor of Statistics for Data Science and Machine Learning - San
Francisco, CA 94105
Plant Engineering Specialist - San Pablo, CA
Page Not Found - Indeed Mobile
Director of Econometric Modeling - External Careers
```

Большинство полученных вакансий совершенно из другой области. К ним относятся такие должности, как *Leadership and Advocacy Coordinator*, *Financial Consultant, RN* и *Scorekeeper*. Одна из вакансий даже гласит: *Page Not Found*, указывая на ошибку при скачивании веб-страницы. Тем не менее несколько объявлений из 20 выведенных все-таки с наукой о данных связаны: к примеру, в одном приглашают *Part-time Instructor of Statistics and Data Science and Machine Learning*. Правда, эта вакансия не совсем соответствует тому, что мы ищем. Все же наша

непосредственная цель — практиковать аналитику данных, а не обучать ей. Таким образом, 20 наименее релевантных вакансий можно выбросить из таблицы. Теперь же для сравнения выведем из `sorted_ds_jobs` 20 наиболее подходящих объявлений (листинг 17.18).

Листинг 17.18. Вывод 20 наиболее релевантных вакансий

```
for title in sorted_df_jobs[:20].Title:
    print(title)

Chief Data Officer - Culver City, CA 90230
Data Scientist - Beavercreek, OH
Data Scientist Population Health - Los Angeles, CA 90059
Data Scientist - San Diego, CA
Data Scientist - Beavercreek, OH
Senior Data Scientist - New York, NY 10018
Data Architect - Raleigh, NC 27609
Data Scientist (PhD) - Spring, TX
Data Science Analyst - Chicago, IL 60612
Associate Data Scientist (BS / MS) - Spring, TX
Data Scientist - Streetsboro, OH 44241
Data Scientist - Los Angeles, CA
Sr Director of Data Science - Elkridge, MD
2019-57 Sr. Data Scientist - Reston, VA 20191
Data Scientist (PhD) - Intern - Spring, TX
Sr Data Scientist. - Alpharetta, GA 30004
Data Scientist GS 13/14 - Clarksburg, WV 26301
Data Science Intern (BS / MS) - Intern - Spring, TX
Senior Data Scientist - New York, NY 10038
Data Scientist - United States
```

Почти все они связаны с аналитикой данных. Некоторые, такие как *Chief Data Officer*, пожалуй, превышают наш уровень навыков, но основная масса вакансий выглядят вполне соответствующими нашей карьере в этой сфере.

ПРИМЕЧАНИЕ

Судя по названию, должность Chief Data Officer является руководящей. Как уже говорилось, такие должности требуют особого набора навыков. Однако если вывести тело объявления (sorted_df_jobs.iloc[0].Body), то мы сразу обнаружим, что эта вакансия совсем не связана с управлением. Компания просто ищет высококвалифицированного специалиста, чтобы доверить ему все вопросы, касающиеся работы с данными. Иногда заголовки вакансий могут сбивать с толку: беглый взгляд на название не способен заменить внимательное прочтение текста объявления.

Очевидно, что при высокой `df_jobs.Relevance` связанные объявления оказываются релевантными. По мере же уменьшения `df_jobs.Relevance` релевантность соответствующих вакансий также падает. Таким образом, можно предположить, что существует некая переломная точка, отделяющая подходящие вакансии от неподходящих. Далее мы попробуем ее найти и для начала визуализируем кривую

упорядоченного распределения релевантности относительно ранга. Иными словами, построим график `range(df_jobs.shape[0])` относительно `sorted_df_jobs.Relevance` (листинг 17.19; рис. 17.3). Ожидается, что кривая релевантности будет непрерывно снижаться. При этом любое резкое снижение укажет на ту самую точку разделения между релевантными и нерелевантными вакансиями.

Листинг 17.19. Построение графика ранжированных вакансий относительно их релевантности

```
import matplotlib.pyplot as plt
plt.plot(range(df_jobs.shape[0]), sorted_df_jobs.Relevance.values)
plt.xlabel('Index')
plt.ylabel('Relevance')
plt.show()
```

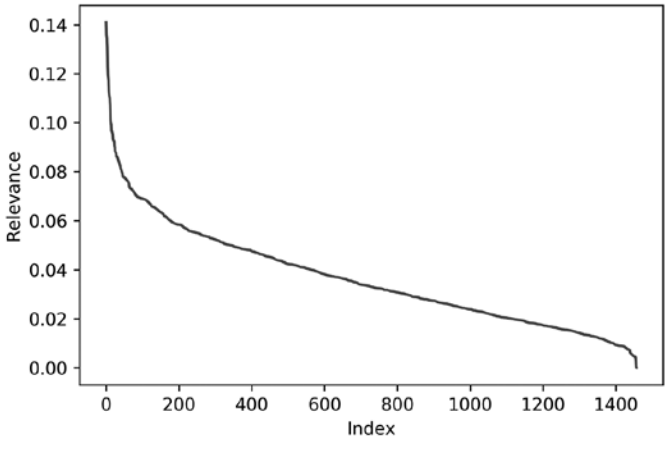


Рис. 17.3. График индексов ранжированных вакансий относительно их релевантности. Меньшие индексы указывают на более высокую релевантность. Релевантность равна косинусному коэффициенту между каждой вакансией и existing_skills. Примерно на индексе 60 ее значение резко снижается

Наша кривая релевантности напоминает график локтя при использовании метода *K*-средних. Изначально она резко опускается, после чего примерно в районе $x = 60$ начинает выпрямляться. Давайте обозначим этот переход, построив вертикальную линию в точке $x = 60$ (листинг 17.20; рис. 17.4).

Листинг 17.20. Добавление в график релевантности вертикального разделителя

```
plt.plot(range(df_jobs.shape[0]), sorted_df_jobs.Relevance.values)
plt.xlabel('Index')
plt.ylabel('Relevance')
plt.axvline(60, c='k')
plt.show()
```

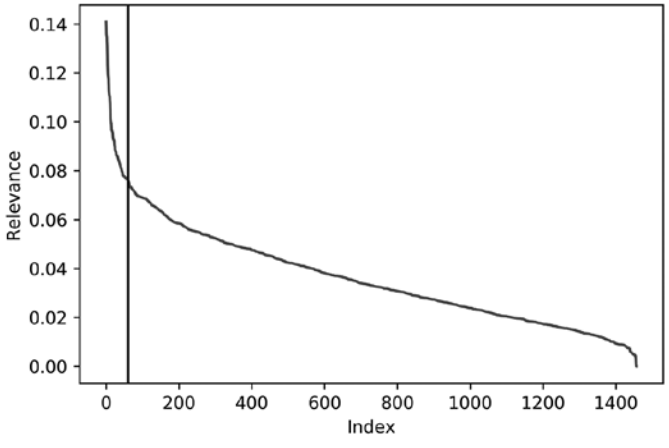


Рис. 17.4. График ранжирования вакансий относительно их релевантности с добавлением вертикальной линии разделения на индексе 60. Индексы менее 60 соответствуют существенно более высоким значениям релевантности

Наш график подразумевает, что первые 60 вакансий заметно более релевантны, чем все последующие. Далее мы это проверим. Как мы уже видели, первые 20 вакансий имеют высокую степень релевантности. Согласно той же логике можно сделать вывод, что вакансии с 40-й по 60-ю также очень релевантны. Выведем `sorted_ds_jobs[40: 60].Title` для оценки (листинг 17.21).

Листинг 17.21. Вывод вакансий ниже порога релевантности

```
for title in sorted_df_jobs[40: 60].Title.values:
    print(title)
```

```
Data Scientist III - Pasadena, CA 91101
Global Data Engineer - Boston, MA
Data Analyst and Data Scientist - Summit, NJ
Data Scientist - Generalist - Glendale, CA
Data Scientist - Seattle, WA
IT Data Scientist - Contract - Riverton, UT
Data Scientist (Analytic Consultant 4) - San Francisco, CA
Data Scientist - Seattle, WA
Data Science & Tagging Analyst - Bethesda, MD 20814
Data Scientist - New York, NY
Senior Data Scientist - Los Angeles, CA
Principal Statistician - Los Angeles, CA
Senior Data Analyst - Los Angeles, CA
Data Scientist - Aliso Viejo, CA 92656
Data Engineer - Seattle, WA
Data Scientist - Digital Factory - Tampa, FL 33607
Data Scientist - Grapevine, TX 76051
```

```
Data Scientist - Bioinformatics - Denver, CO 80221
EPIDEMIOLOGIST - Los Angeles, CA
Data Scientist - Bellevue, WA
```

Почти все полученные вакансии относятся к должностям ученых либо аналитиков, работающих с данными. Единственным кандидатом на выброс здесь является эпидемиолог, который просочился, вероятно, из-за заявленного опыта в отслеживании эпидемий заболеваний. За исключением этого остальные вакансии отлично подходят. По нашей логике при выведении очередных 20 вакансий показатель их релевантности должен снижаться, поскольку они выходят за границу индекса 60. Проверим это (листинг 17.22).

Листинг 17.22. Вывод вакансий за пределами порога релевантности

```
for title in sorted_df_jobs[60: 80].Title.values:
    print(title)

Data Scientist - Aliso Viejo, CA
Data Scientist and Visualization Specialist - Santa Clara Valley, CA 95014
Data Scientist - Los Angeles, CA
Data Scientist Manager - NEW YORK LOCATION! - New York, NY 10036
Data Science Intern - San Francisco, CA 94105
Research Data Analyst - San Francisco, CA
Sr Data Scientist (Analytic Consultant 5) - San Francisco, CA
Data Scientist, Media Manipulation - Cambridge, MA
Manager, Data Science, Programming and Visualization - Boston, MA
Data Scientist in Broomfield, CO - Broomfield, CO
Senior Data Scientist - Executive Projects and New Solutions - Foster City, CA
Manager of Data Science - Burbank California - Burbank, CA
Data Scientist Manager - Hiring in Burbank! - Burbank, CA
Data Scientists needed in NY - Senior Consultants and Managers! - New York, NY
10036
Data Scientist - Menlo Park, CA
Data Engineer - Santa Clara, CA
Data Scientist - Remote
Data Scientist I-III - Phoenix, AZ 85021
SWE Data Scientist - Santa Clara Valley, CA 95014
Health Science Specialist - San Francisco, CA 94102
```

Несколько вакансий среди объявлений с 60-го по 80-е заметно менее релевантны, чем рассмотренные ранее. Некоторые из них относятся к управленческим должностям, а одна — к должности специалиста по здравоохранению. Тем не менее основная часть вакансий связана с ролью аналитика данных вне области здравоохранения или управления. Это наблюдение можно легко оценить количественно при помощи регулярных выражений. Мы определим функцию `percent_relevant_titles`, возвращающую процент неуправленческих вакансий по аналитике данных в срезе датафрейма, и применим ее к `sorted_df_jobs[60: 80]`. Результат даст очень простую альтернативную меру релевантности на основе заголовков вакансий (листинг 17.23).

Листинг 17.23. Измерение релевантности заголовков в подмножестве вакансий

```
import re
def percent_relevant_titles(df):
    regex_relevant = re.compile(r'Data (Scien|Analy)',
                                flags=re.IGNORECASE)
    regex_irrelevant = re.compile(r'\b(Manage)',
                                  flags=re.IGNORECASE)
    match_count = len([title for title in df.Title
                       if regex_relevant.search(title)
                       and not regex_irrelevant.search(title)])
    percent = 100 * match_count / df.shape[0]
    return percent

percent = percent_relevant_titles(sorted_df_jobs[60: 80])
print(f"Approximately {percent:.2f}% of job titles between indices "
      "60 - 80 are relevant")
```

Сопоставляет релевантные заголовки вакансий, в которых упоминается должность аналитика данных

Сопоставляет нерелевантные заголовки вакансий, где упоминаются управляющие должности

Подсчитывает число неуправленческих должностей аналитика данных

```
Approximately 65.00% of job titles between indices 60 - 80 are relevant
```

Примерно две трети заголовков вакансий из `sorted_df_jobs[60: 80]` релевантны. И хотя релевантность после индекса 60 снизилась, более 50 % вакансий по-прежнему связаны с наукой о данных. Возможно, этот процент упадет, если вывести следующие 20 вакансий в диапазоне от 80 до 100. Проверим это (листинг 17.24).

Листинг 17.24. Измерение релевантности следующего подмножества вакансий

```
percent = percent_relevant_titles(sorted_df_jobs[80: 100])
print(f"Approximately {percent:.2f}% of job titles between indices "
      "80 - 100 are relevant")
```

```
Approximately 80.00% of job titles between indices 80 - 100 are relevant
```

А вот и нет! Процент заголовков, связанных с наукой о данных, возрос до 80 %. Когда же этот показатель упадет ниже 50 %? Далее мы выполним серию итераций с вычислением процента релевантности, после чего отобразим эти проценты на графике (листинг 17.25; рис. 17.5). А также построим горизонтальную линию в районе 50 %, чтобы можно было определить индекс, в котором релевантных заголовков вакансий будет меньшинство.

Листинг 17.25. Построение графика релевантности для групп заголовков вакансий

```
def relevant_title_plot(index_range=20):
    percentages = []

    start_indices = range(df_jobs.shape[0] - index_range)
    for i in start_indices:
```

Эта функция выполняет percent_relevant_titles для каждого последующего среза из index_range вакансий, после чего все полученные показатели отображаются на графике. Позднее мы скорректируем значение этого параметра

Анализирует sorted_df_jobs[i: i + index_range], где i — это диапазон от 0 до общего числа объявлений минус диапазон индексов

```
        df_slice = sorted_df_jobs[i: i + index_range]
        percent = percent_relevant_titles(df_slice)
        percentages.append(percent)
    plt.plot(start_indices, percentages)
    plt.axhline(50, c='k')
    plt.xlabel('Index')
    plt.ylabel('% Relevant Titles')

relevant_title_plot()
plt.show()
```

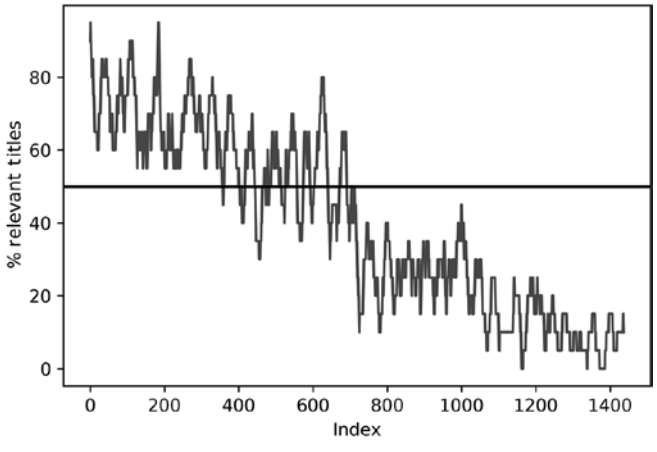


Рис. 17.5. График индексов ранжированных вакансий относительно релевантности их заголовков. Релевантность заголовка равна проценту связанных с наукой о данных заголовков среди 20 последовательных вакансий, начиная с некоторого индекса. Горизонтальной линией обозначена релевантность 50 %. Ниже 50 % она оказывается примерно на индексе 700

График сильно колеблется. Но, несмотря на эти колебания, видно, что релевантных заголовков становится менее 50 % примерно на индексе 700. Естественно, этот порог может быть характерным лишь для выбранного нами диапазона индексов. Сохранится ли он, если диапазон удвоить? Чтобы это выяснить, выполним `relevant_title_plot(index_range=40)` (листинг 17.26; рис. 17.6). Мы также построим вертикальную линию по индексу 700, которая позволит убедиться в том, что после нее релевантных заголовков становится менее 50 %.

Листинг 17.26. Построение графика процентной релевантности при увеличенном диапазоне индексов

```
relevant_title_plot(index_range=40)
plt.axvline(700, c='k')
plt.show()
```

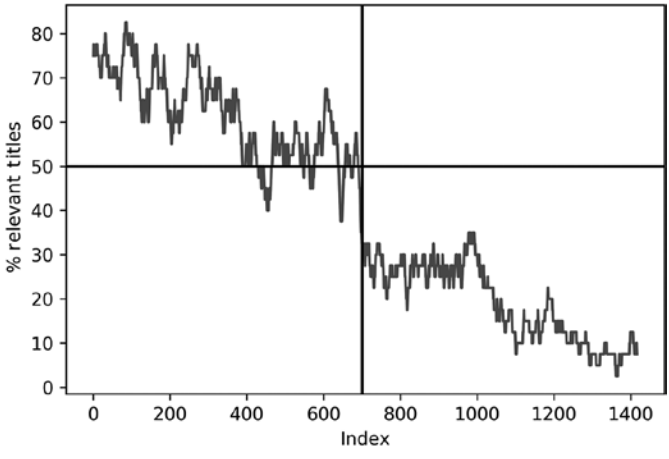


Рис. 17.6. График индексов ранжированных вакансий относительно релевантности заголовков. Релевантность заголовков равна проценту связанных с наукой о данных заголовков среди 40 последовательных вакансий, начиная с некоторого индекса. Горизонтальная линия обозначает релевантность 50 %, а вертикальная — индекс 700. После нее релевантность опускается ниже 50 %

Наш обновленный график после индекса 700 продолжает опускаться ниже 50 %.

ПРИМЕЧАНИЕ

Аппроксимировать этот порог можно разными способами. Предположим, мы упрощаем регулярное выражение до r'Data (Science|Scientist)'. Таким образом игнорируем все упоминания аналитиков или менеджеров. Также предположим, что мы исключаем диапазоны индексов и вместо этого подсчитываем общее число заголовков с упоминанием науки о данных, встречающихся под каждым индексом. Отобразив на графике эти простые результаты, увидим кривую, выравнивающуюся на индексе 700. Несмотря на упрощение, мы получили очень похожие результаты. В науке о данных зачастую имеется более одного пути к получению искомых выводов.

На этом этапе перед нами стоит выбор между двух порогов релевантности. Первый порог, индекс 60, очень точен: большинство вакансий под ним связаны с должностями специалиста по данным. Однако у него очень ограничен охват: сотни вакансий аналитиков данных оказываются за индексом 60. При этом второй порог по индексу 700 охватывает намного больше должностей, связанных с обработкой данных, но в него попадают и некоторые нерелевантные вакансии. Между этими порогами разница почти 12-кратная, так какой же использовать? Что предпочесть, более высокую точность или охват? Если выбрать охват, то сильно ли повредит нашему анализу лишний шум? А если склониться к точности, то не окажется ли анализ неполноценным из-за ограниченного разнообразия упоминаемых навыков? Это серьезные вопросы, и, к сожалению, простого ответа на них нет. Более высокая точность ценой меньшего охвата может навредить анализу, но и обратный выбор — тоже. Что же делать?

А может, попробовать использовать оба порога? Таким образом нам удастся сравнить недостатки и преимущества каждого. Сначала мы кластеризуем наборы навыков из вакансий ниже индекса 60, а затем повторим анализ для вакансий ниже индекса 700. После этого интегрируем два результата в единое заключение.

17.3. КЛАСТЕРИЗАЦИЯ НАВЫКОВ В РЕЛЕВАНТНЫХ ОБЪЯВЛЕНИЯХ О ВАКАНСИЯХ

Наша цель — кластеризовать навыки в 60 наиболее релевантных вакансиях. В каждой из них они различаются и отчасти представлены маркированными списками. Получается, перед нами следующий выбор:

- кластеризовать 60 текстов в `sorted_df_jobs[:60].Body`;
- кластеризовать сотни отдельных пунктов списков в `sorted_df_jobs[:60].Bullets`.

Второй вариант предпочтительнее по двум причинам.

- Наша цель — определить недостающие навыки, а пункты списков в большей степени сосредоточены на отдельных навыках, чем разнородное содержимое самого объявления.
- Короткие пункты списков легко выводить и читать, чего не скажешь о длинных объявлениях. Таким образом, кластеризация пунктов позволит проанализировать каждый кластер путем вывода текста некоторых из кластеризованных пунктов.

Мы кластеризуем выделенные из текстов пункты, начав с сохранения `sorted_df_jobs[:60].Bullets` в одном списке (листинг 17.27).

Листинг 17.27. Извлечение пунктов из 60 наиболее релевантных вакансий

```
total_bullets = []
for bullets in sorted_df_jobs[:60].Bullets:
    total_bullets.extend(bullets)
```

Сколько пунктов в списке? Есть ли среди них повторы? Проверить это можно, загрузив `total_bullets` в таблицу Pandas и применив метод `describe` (листинг 17.28).

Листинг 17.28. Сбор общей статистики по пунктам

```
df_bullets = pd.DataFrame({'Bullet': total_bullets})
print(df_bullets.describe())

Bullet
count                                                1091
unique                                                900
top     Knowledge of advanced statistical techniques a...
freq                                                    9
```

Всего в списке 1091 пункт, но лишь 900 из них уникальны — остальные 91 являются повторами. Самый частый повтор встречается девять раз. Если с этой проблемой не разобраться, она повлияет на результат кластеризации. Следует удалить все повторы, прежде чем продолжить анализ.

ПРИМЕЧАНИЕ

Откуда берутся повторы? Выяснить это можно, проследив, к каким объявлениям относятся некоторые из них. Из соображений экономии места этот анализ в книге не приводится, но мы рекомендуем вам выполнить его самостоятельно. Результат покажет, что некоторые компании используют одни и те же шаблоны объявлений для разных вакансий. Каждый шаблон корректируется под соответствующую должность, но определенные пункты списков остаются. Эти повторы могут увести кластеры в сторону специфичных для определенных компаний навыков, поэтому лучше их из total_bullets удалить.

Код листинга 17.29 отфильтровывает пустые строки и повторы из списка пунктов, после чего векторизует этот список с помощью векторизатора TF-IDF.

Листинг 17.29. Удаление повторов и векторизация пунктов

```
total_bullets = sorted(set(total_bullets))
vectorizer = TfidfVectorizer(stop_words='english')
tfidf_matrix = vectorizer.fit_transform(total_bullets)
num_rows, num_columns = tfidf_matrix.shape
print(f"Our matrix has {num_rows} rows and {num_columns} columns")
```

Преобразует total_bullets в множество, чтобы удалить 91 повтор. Это множество мы упорядочиваем для обеспечения согласованной последовательности его элементов, а значит, и согласованного вывода. В качестве альтернативы можно исключить повторы непосредственно из таблицы Pandas, выполнив df_bullets.drop_duplicates(inplace=True)

Векторизация списка без повторов выполнена. Полученная матрица TF-IDF содержит 900 строк и более 2000 столбцов. Итого в ней более 1,8 млн элементов. Такая матрица слишком велика для эффективной кластеризации, поэтому уменьшим ее с помощью процедуры, описанной в главе 15. Мы сократим матрицу до 100 измерений с помощью SVD, после чего нормализуем ее (листинг 17.30).

Листинг 17.30. Уменьшение размерности матрицы TF-IDF

```
import numpy as np
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import normalize
np.random.seed(0)

def shrink_matrix(tfidf_matrix):
    svd_object = TruncatedSVD(n_components=100)
    shrunk_matrix = svd_object.fit_transform(tfidf_matrix)
    return normalize(shrunk_matrix)

shrunk_norm_matrix = shrink_matrix(tfidf_matrix)
```

Применяет SVD к входной матрице TF-IDF. Матрица уменьшается до 100 измерений, нормализуется и возвращается

Мы почти готовы кластеризовать нормализованную матрицу с помощью метода *K*-средних. Однако сначала нужно подобрать *K*. Сгенерируем график локтя, используя мини-пакетную версию метода *K*-средних, оптимизированную под скорость (листинг 17.31; рис. 17.7).

Листинг 17.31. Построение графика локтя методом K-средних для мини-пакетов

```
np.random.seed(0)
from sklearn.cluster import MiniBatchKMeans
def generate_elbow_plot(matrix):
    k_values = range(1, 61)
    inertia_values = [MiniBatchKMeans(k).fit(matrix).inertia_
                      for k in k_values]
    plt.plot(k_values, inertia_values)
    plt.xlabel('K')
    plt.ylabel('Inertia')
    plt.grid(True)
    plt.show()

generate_elbow_plot(shrunk_norm_matrix)
```

Генерирует график локтя для входной матрицы данных, используя метод K-средних для мини-пакетов

Число кластеров — от 1 до 60

Строит сетку, чтобы проще было определить нахождение локтя на оси X

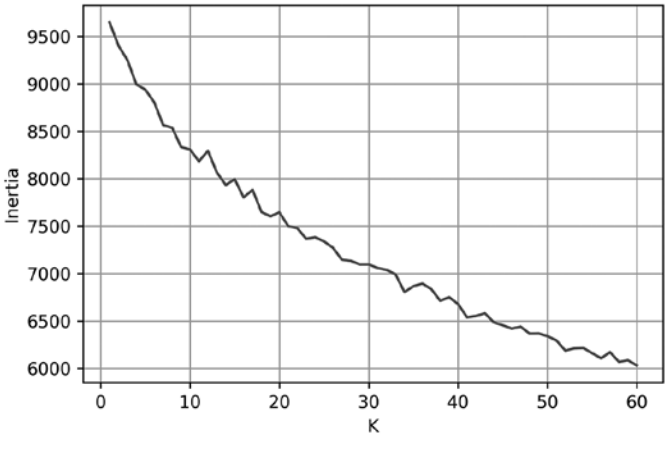


Рис. 17.7. График локтя, построенный с помощью мини-пакетного метода K-средних для диапазона значений K от 1 до 60. Точное расположение локтя определить сложно

Полученная кривая опускается плавно. Точное расположение изогнутого локтеобразного перехода определить сложно: кривая резко опускается при $K = 10$ и затем постепенно изгибается где-то между $K = 10$ и $K = 25$. Какое же значение K нам следует выбрать — 10, 25 или нечто среднее вроде 15 либо 20? Правильный ответ здесь неочевиден, так почему бы не попробовать несколько значений K? Мы кластеризуем данные несколько раз, используя K, равное 10, 15, 20 и 25, а затем сопоставим результаты. При необходимости подумаем о выборе для кластеризации другого K. Начнем с группировки навыков по 15 кластерам.

ПРИМЕЧАНИЕ

Наша цель — изучить результаты при четырех разных значениях K. Генерировать эти результаты можно в произвольном порядке. Здесь мы начнем с K = 15, потому что такое число итоговых кластеров не слишком велико, но и не слишком мало. Это позволит установить хорошую основу для последующего анализа результатов.

17.3.1. Группировка навыков по 15 кластерам

Мы выполним метод *K*-средних при *K* = 15, после чего сохраним индексы текстов и ID кластеров в таблице Pandas. Мы также сохраним фактическое содержимое пунктов списков для дальнейшего обращения к нему. Наконец, с помощью Pandas метода `groupby` мы разделим таблицу на кластеры (листинг 17.32).

Листинг 17.32. Разнесение пунктов по 15 кластерам

```
np.random.seed(0)
from sklearn.cluster import KMeans

def compute_cluster_groups(shrunk_norm_matrix, k=15,
                           bullets=total_bullets):
    cluster_model = KMeans(n_clusters=k)
    clusters = cluster_model.fit_predict(shrunk_norm_matrix)
    df = pd.DataFrame({'Index': range(clusters.size), 'Cluster': clusters,
                       'Bullet': bullets})
    return [df_cluster for _, df_cluster in df.groupby('Cluster')]

cluster_groups = compute_cluster_groups(shrunk_norm_matrix)
```

Выполняет для входной shrunk_norm_matrix кластеризацию методом K-средних. Параметр K установлен на 15. Эта функция возвращает список таблиц Pandas, в котором каждая таблица представляет кластер. В эти таблицы включены кластеризованные пункты, переданные через дополнительный параметр bullets

Отслеживает индекс каждого кластеризованного пункта в кластере, ID кластера и текст

Каждый из наших текстовых кластеров сохранен в таблице Pandas в списке `cluster_groups`. Мы можем визуализировать эти кластеры с помощью облаков слов. В главе 15 мы определили для такой визуализации собственную функцию `cluster_to_image`. Она получала таблицу кластеров и возвращала изображение облака слов. Код листинга 17.33 переопределяет ту самую функцию и применяет ее к `cluster_groups[0]` (рис. 17.8).

ПРИМЕЧАНИЕ

Почему нужно переопределять функцию? Дело в том, что в главе 15 cluster_to_image зависела от фиксированной матрицы TF-IDF и списка-словаря. В текущем же анализе эти параметры не фиксированы — и матрица, и словарь по мере подстройки индекса релевантности будут смещаться. В связи с этим нужно обновить функцию, чтобы получить более динамический вывод.

Листинг 17.33. Визуализация первого кластера

```
from wordcloud import WordCloud
np.random.seed(0)

def cluster_to_image(df_cluster, max_words=10, tfidf_matrix=tfidf_matrix,
                     vectorizer=vectorizer):
    indices = df_cluster.Index.values
    summed_tfidf = np.asarray(tfidf_matrix[indices].sum(axis=0))[0]
    data = {'Word': vectorizer.get_feature_names(),'Summed TFIDF':
        summed_tfidf}
    df_ranked_words = pd.DataFrame(data).sort_values('Summed TFIDF',
        ascending=False)
    words_to_score = {word: score
                      for word, score in df_ranked_words[:max_words].values
                      if score != 0}
    cloud_generator = WordCloud(background_color='white',
                                color_func=_color_func,
                                random_state=1)
    wordcloud_image = cloud_generator.fit_words(words_to_score)
    return wordcloud_image

def _color_func(*args, **kwargs):
    return np.random.choice(['black', 'blue', 'teal', 'purple', 'brown'])

wordcloud_image = cluster_to_image(cluster_groups[0])
plt.imshow(wordcloud_image, interpolation="bilinear")
plt.show()
```

Получает таблицу df_cluster и возвращает облако слов для ведущих max_words слов, соответствующих этому кластеру. Слова берутся из входного класса vectorizer. Они суммируются по строкам входной tfidf_matrix. Когда мы увеличим порог вакансий с 60 до 700, vectorizer и tfidf_matrix нужно будет соответствующим образом подстроить

Вспомогательная функция для случайного присваивания каждому слову одного из пяти цветов

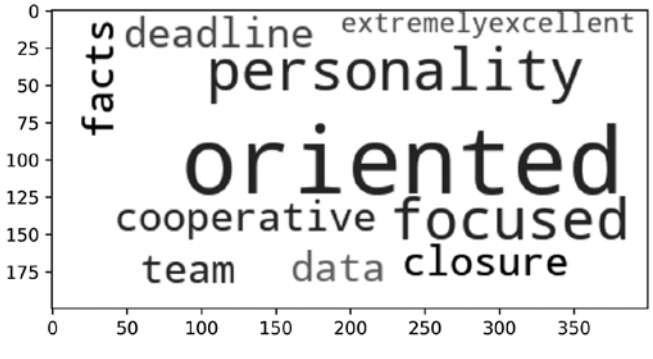


Рис. 17.8. Облако слов, сгенерированное для кластера по индексу 0. Формулировка в нем получилась немного расплывчатой. Она описывает внимательного человека, ориентированного на работу с данными

Похоже, что содержание облака слов описывает внимательного человека, ориентированного на работу с данными, но это не совсем ясная формулировка. Возможно,

удастся узнать о кластере больше, если вывести из `cluster_group[0]` некоторые пункты (листинг 17.34).

ПРИМЕЧАНИЕ

Мы выведем случайную выборку пунктов. Она должна оказаться достаточно информативной для понимания всего кластера. Однако стоит подчеркнуть, что не все пункты равнозначны — некоторые находятся ближе к центру масс их кластера, а значит, отражают его смысл более явно. Получается, что мы можем при желании упорядочить пункты на основе их расстояния до среднего значения кластера. В книге из соображений экономии места мы эту операцию опустим, но вам рекомендуем выполнить ее самостоятельно.

Листинг 17.34. Вывод некоторых пунктов из кластера 0

```
np.random.seed(1)
def print_cluster_sample(cluster_id):
    df_cluster = cluster_groups[cluster_id]
    for bullet in np.random.choice(df_cluster.Bullet.values, 5,
                                   eplace=False):
        print(bullet)

print_cluster_sample(0)

Data-oriented personality
Detail-oriented
Detail-oriented – quality and precision-focused
Should be extremelyExcellent facts and data oriented
Data oriented personality
```

Выводит пять случайных пунктов из cluster_groups[cluster_id]

Все выведенные пункты похожи и нацелены на соискателя, внимательного к деталям и ориентированного на работу с данными. Чисто лингвистически этот кластер допустим, но он отражает навык, который сложно оценить. Внимание к деталям очень общее качество — его трудно измерить количественно, продемонстрировать и освоить. По-хорошему, другие кластеры должны содержать более конкретные технические навыки.

Далее мы проанализируем все 15 кластеров одновременно, используя облака слов. Они будут отображены в 15 подграфиках, расположенных в сетке 5×3 (листинг 17.35; рис. 17.9).

Листинг 17.35. Визуализация 15 кластеров

```
def plot_wordcloud_grid(cluster_groups, num_rows=5, num_columns=3,
                        **kwargs):
    figure, axes = plt.subplots(num_rows, num_columns, figsize=(20, 15))
    cluster_groups_copy = cluster_groups[:]
    for r in range(num_rows):
        for c in range(num_columns):
            if not cluster_groups_copy:
                break
```

Строит облако слов для каждого кластера в cluster_groups. Они отображаются в сетке размером num_rows на num_columns

Синтаксис **kwargs позволяет передать в функцию cluster_to_image дополнительные параметры. Таким образом можно легко изменять и vectorizer, и tfidf_matrix

```
            df_cluster = cluster_groups_copy.pop(0)
            wordcloud_image = cluster_to_image(df_cluster, **kwargs)
            ax = axes[r][c]
            ax.imshow(wordcloud_image, interpolation="bilinear")
            ax.set_title(f"Cluster {df_cluster.Cluster.iloc[0]}")
            ax.set_xticks([])
            ax.set_yticks([])

plot_wordcloud_grid(cluster_groups)
plt.show()
```

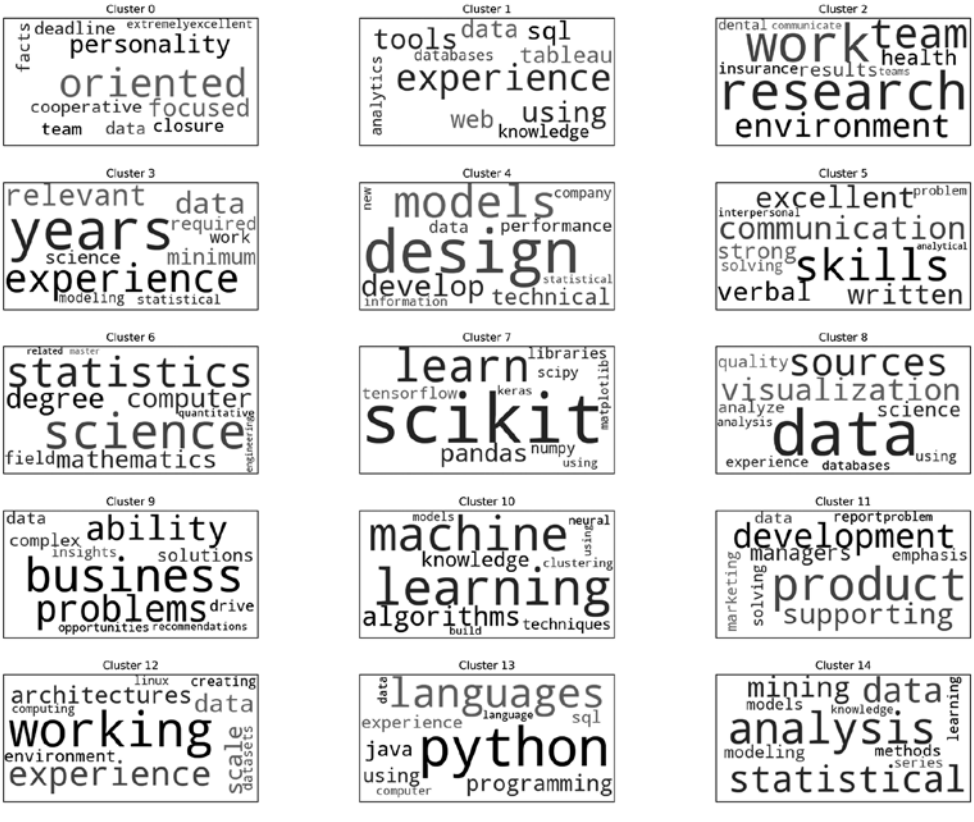


Рис. 17.9. Пятнадцать облаков слов визуализированы в 15 подграфиках. Каждое облако соответствует одному из 15 кластеров. Заголовки подграфиков соответствуют ID кластеров. Некоторые кластеры, например кластер 7, описывают технические навыки. Другие же, например кластер 0, менее технические

Наши 15 кластеров навыков демонстрируют разноплановый набор тематик. Некоторые кластеры преимущественно технического характера. К примеру, кластер 7 сосредоточен на внешних библиотеках аналитики данных, таких как scikit-learn, Pandas, NumPy, Matplotlib и SciPy.

Причем scikit-learn явно доминирует. Большинство из этих библиотек присутствуют в нашем резюме и обсуждались на протяжении книги. Давайте выведем выборку пунктов из кластера 7 и убедимся в том, что в них говорится о библиотеках для аналитики данных (листинг 17.36).

Листинг 17.36. Вывод некоторых пунктов из кластера 7

```
np.random.seed(1)
print_cluster_sample(7)

Experience using one or more of the following software packages:
scikit-learn, numpy, pandas, jupyter, matplotlib, scipy, nltk, spacy, keras,
tensorflow
Using one or more of the following software packages: scikit-learn, numpy, pandas,
jupyter, matplotlib, scipy, nltk, spacy, keras, tensorflow Experience with machine
learning libraries and platforms, like Scikit-learn and Tensorflow

Proficiency in incorporating the use of external proprietary and open-source
libraries such as, but not limited to, Pandas, Scikit- learn, Matplotlib, Seaborn,
GDAL, GeoPandas, and ArcPy
Experience using ML libraries, such as scikit-learn, caret, mlr, mllib
```

При этом другие кластеры, например кластер 0, охватывают в основном нетехнические навыки. Эти личностные качества, включающие бизнес-проницательность, внимательность, стратегическое мышление, коммуникабельность и умение сотрудничать, в нашем резюме отсутствуют. Значит, в среднем нетехнические кластеры должны иметь меньшее сходство с резюме. Это подталкивает к интересной мысли: почему бы не разделить технические кластеры и кластеры личностных качеств на основе сходства их текстов? Такое разделение позволит более систематически оценить каждый тип навыков. Давайте попробуем! Начнем с вычисления косинусного коэффициента между каждым пунктом в `total_bullets` и нашем резюме (листинг 17.37).

ПРИМЕЧАНИЕ

Почему мы используем только резюме, а не комбинацию навыков из резюме и содержания, упорядоченных в переменной existing_skills? Дело в том, что наша конечная цель — определить, какие кластеры навыков отсутствуют в резюме. А в этом должно помочь вычисление прямого сходства между резюме и каждым кластером. Если коэффициент сходства окажется слишком низким, значит, кластеризованные навыки представлены в резюме недостаточно явно.

Листинг 17.37. Вычисление сходства между пунктами кластеров и нашим резюме

```
def compute_bullet_similarity(bullet_texts):
    bullet_vectorizer = TfidfVectorizer(stop_words='english')
    matrix = bullet_vectorizer.fit_transform(bullet_texts + [resume])
    matrix = matrix.toarray()
    return matrix[:-1] @ matrix[-1]

bullet_cosine_similarities = compute_bullet_similarity(total_bullets)
```

Вычисляет косинусный коэффициент между входным bullet_texts и переменной resume

Наш массив `bullet_cosine_similarities` содержит коэффициенты сходства текстов между всеми кластеризованными пунктами. В любом отдельном кластере можно совместить эти коэффициенты в единый показатель, вычислив их среднее. Согласно нашей гипотезе, технический кластер должен иметь более высокое среднее сходство с резюме, чем кластер личностных качеств. Далее мы проверим, так ли это, на примере уже знакомых кластеров 7 и 0 (листинг 17.38).

Листинг 17.38. Сравнение средних коэффициентов сходства с резюме

```
def compute_mean_similarity(df_cluster):
    indices = df_cluster.Index.values
    return bullet_cosine_similarities[indices].mean()

tech_mean = compute_mean_similarity(cluster_groups[7])
soft_mean = compute_mean_similarity(cluster_groups[0])
print(f"Technical cluster 7 has a mean similarity of {tech_mean:.3f}")
print(f"Soft-skill cluster 3 has a mean similarity of {soft_mean:.3f}")

Technical cluster 7 has a mean similarity of 0.203 Soft-skill cluster 3 has a mean
similarity of 0.002
```

Технический кластер в 100 раз ближе к нашему резюме, чем кластер с личностными качествами. Получается, мы на правильном пути! Далее вычислим средний коэффициент сходства для всех 15 кластеров, после чего упорядочим их по убыванию согласно полученным результатам. Если наше предположение будет верным, технические кластеры окажутся в начале упорядоченного списка. Подтвердить мы это сможем, заново отобразив сетку подграфиков с облаками слов. Код листинга 17.39 выполняет упорядочение и визуализирует кластеры (рис. 17.10).

ПРИМЕЧАНИЕ

Мы собираемся упорядочить кластеры по соответствию технической сфере. Для завершения практического задания это не обязательно — каждый кластер можно оценить отдельно в произвольном порядке. Однако за счет переупорядочения кластеров мы можем быстрее выполнить это, поэтому для упрощения рабочего процесса данный подход предпочтительнее.

Листинг 17.39. Упорядочение подграфиков по сходству с резюме

```
def sort_cluster_groups(cluster_groups):
    mean_similarities = [compute_mean_similarity(df_cluster)
                         for df_cluster in cluster_groups]

    sorted_indices = sorted(range(len(cluster_groups)),
                            key=lambda i: mean_similarities[i],
                            reverse=True)
    return [cluster_groups[i] for i in sorted_indices]

sorted_cluster_groups = sort_cluster_groups(cluster_groups)
plot_wordcloud_grid(sorted_cluster_groups)
plt.show()
```

Упорядочивает входной массив cluster_groups по среднему коэффициенту сходства с резюме

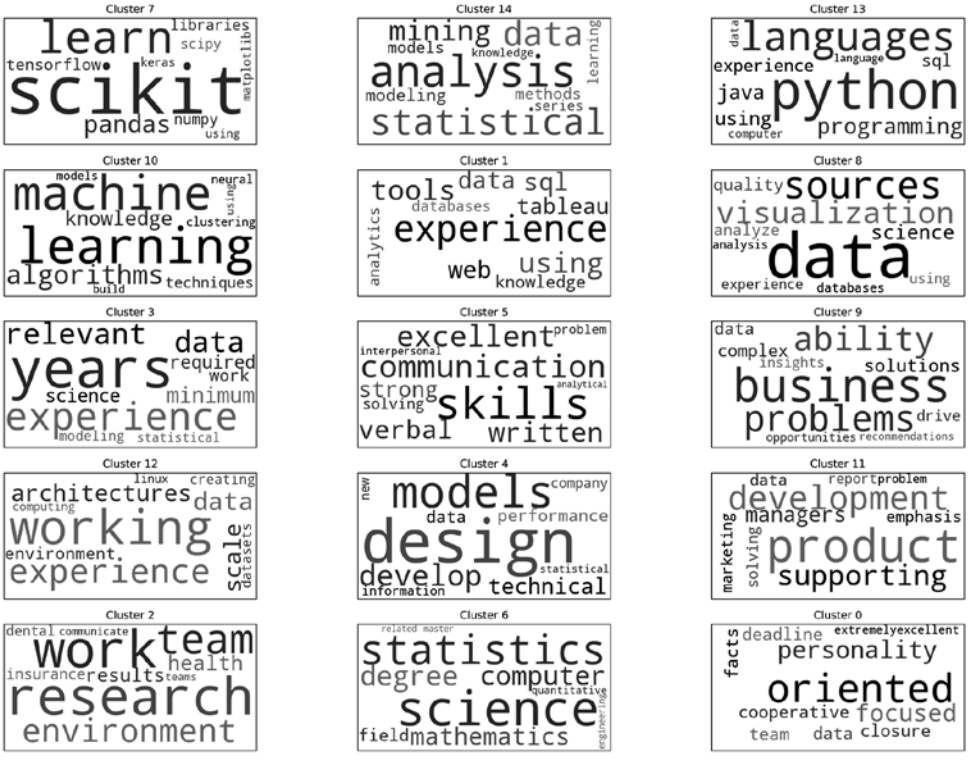


Рис. 17.10. Пятнадцать облаков слов визуализированы в 15 подграфиках. Каждое облако слов соответствует одному из 15 кластеров. Кластеры упорядочены по среднему сходству с резюме. Первые два ряда сетки подграфиков содержат более технические кластеры

Наша гипотеза оказалась верна! Первые два ряда обновленной сетки подграфиков явно соответствуют техническим навыкам. Более того, эти технические навыки теперь удобно упорядочены на основе степени сходства с нашим резюме. Это позволяет систематически ранжировать навыки от наиболее схожих (то есть представленных в резюме) до наименее схожих (то есть, скорее всего, в резюме отсутствующих).

17.3.2. Анализ кластеров технических навыков

Теперь мы перейдем к шести техническим кластерам, отображенным в первых двух рядах сетки подграфиков, и заново построим связанные с ними облака слов, но уже в сетке, состоящей из двух рядов и трех столбцов (листинг 17.40; рис. 17.11). Эта визуализация технических навыков позволит расширить размер облака слов. Позднее вернемся к оставшимся облакам, связанным с личностными качествами (см. рис. 17.10).

Листинг 17.40. Построение облаков слов для шести технических кластеров

```
plot_wordcloud_gri(sorted_cluster_groups[:6], num_rows=3, num_columns=2)
plt.show()
```

Рис. 17.11. Шесть облаков слов, связанных с шестью техническими кластерами. Они упорядочены по среднему сходству с резюме. Первые четыре облака информативны: они сосредоточены на научных библиотеках, статистическом анализе, программировании с помощью Python и машинном обучении. Оставшиеся два облака слов неконкретны и неинформативны

Первые четыре технических кластера в сетке очень информативны. Далее мы поочередно их проанализируем, начав с левого верхнего квадрата сетки. Для краткости изложения задействуем только облака слов. Их содержимого должно быть достаточно для охвата навыков, представленных каждым кластером. Однако, если вы захотите изучить какой-либо кластер подробнее, можете дополнительно вывести некоторые из его пунктов.

Первые четыре кластера можно описать так.

- *Кластер 7 (ряд 0, столбец 0)* — ориентирован на библиотеки для аналитики данных, и мы его уже обсуждали. Библиотеки scikit-learn, NumPy, SciPy и Pandas в данной книге рассматривались.

А вот о TensorFlow и Keras мы не говорили. Это библиотеки для глубокого обучения, которые специалисты по ИИ используют при обучении сложных предиктивных моделей на мощном оборудовании. Граница между должностями специалиста по данным и специалиста по ИИ не всегда явная.

И хотя знание техник глубокого обучения обычно к требованиям не относится, иногда знакомство с ними помогает получить желаемую работу. Так что, если вас заинтересует изучение данных библиотек, рекомендую книгу Нишанта Шуклы (Nishant Shukla) *Machine Learning with TensotFlow* (Manning, 2018, www.manning.com/books/machine-learning-with-tensorflow) или Франсуа Шолле (François Chollet) *Deep Learning with Python, Second Edition*[1] (Manning, 2021, www.manning.com/books/deep-learning-with-python-second-edition).

- *Кластер 14 (ряд 0, столбец 1)* — сосредоточен на статистическом анализе, который в нашем резюме указан. Статистические методы рассматривались во втором практическом задании книги.
- *Кластер 13 (ряд 1, столбец 0)* — описывает навыки владения языками программирования. Среди необходимых языков явно доминирует Python. Если учесть наш опыт работы с Python, почему этот кластер не оказался в списке выше? Все дело в том, что в нашем резюме Python не назван. Да, мы работаем с множеством его библиотек, что подразумевает знакомство с этим языком, но сами навыки использования Python, которые мы получили при прочтении книги, нигде явно не указаны. Возможно, стоит дополнить ими наше резюме.
- *Кластер 10 (ряд 1, столбец 2)* — ориентирован на машинное обучение. Область МО включает различные предиктивные алгоритмы на основе данных. Многие из них представлены в следующем практическом задании, но пока мы не закончили текущее, вносить их в резюме рано.

  Дополнительно стоит отметить, что техники кластеризации иногда относят к алгоритмам МО *без учителя (unsupervised learning)*. То есть такую формулировку мы в резюме указать можем, но любое упоминание более общих навыков работы с машинным обучением даст ложное представление о наших компетенциях.

Последние два технических кластера туманны и неинформативны. В них перечисляются несвязанные инструменты и техники анализа. Код листинга 17.41 отбирает пункты из этих кластеров (8 и 1), чтобы подтвердить отсутствие в них явного паттерна.

ПРИМЕЧАНИЕ

В обоих кластерах упоминается база данных. Умение использовать базы данных — полезный навык, но в этих кластерах он неосновной. Чуть позже в текущей главе мы встретим кластер, связанный с базами данных, когда увеличим значение K.

---

[1] *Шолле Ф.* Глубокое обучение на Python. 2-е межд. изд. — СПб.: Питер, 2023.

Листинг 17.41. Вывод нескольких пунктов из кластеров 8 и 1

```
np.random.seed(1)
for cluster_id in [8, 1]:
    print(f'\nCluster {cluster_id}:')
    print_cluster_sample(cluster_id)
```

```
Cluster 8:
Use data to inform and label customer outcomes and processes
Perform exploratory data analysis for quality control and improved understanding
Champion a data-driven culture and help develop best-in-class data science
capabilities
Work with data engineers to plan, implement, and automate integration of external
data sources across a variety of architectures, including local databases, web
APIs, CRM systems, etc
Design, implement, and maintain a cutting-edge cloud-based data-infrastructure for
large data-sets

Cluster 1:
Have good knowledge on Project management tools JIRA, Redmine, and Bugzilla
Using common cloud computing platforms including AWS and GCP in addition to their
respective utilities for managing and manipulating large data sources, model,
development, and deployment
Experience in project deployment using Heroku/Jenkins and using web Services like
Amazon Web Services (AWS)
Expert level data analytics experience with T-SQL and Tableau
Experience reviewing and assessing military ground technologies
```

На этом анализ технических кластеров закончен. Четыре из них оказались релевантными, а два — нет. Теперь перейдем к оставшимся кластерам, связанным с личностными качествами. Нам нужно понять, присутствуют ли в этих данных релевантные кластеры.

17.3.3. Анализ кластеров личностных качеств

Начнем с визуализации оставшихся девяти кластеров в сетке размером 3×3 (листинг 17.42; рис. 17.12).

Листинг 17.42. Построение облаков слов для оставшихся девяти кластеров личностных качеств

```
plot_wordcloud_grid(sorted_cluster_groups[:6], num_rows=3, num_columns=3)
plt.show()
```

Оставшиеся кластеры выглядят куда менее однозначно в сравнении с первыми четырьмя техническими. Интерпретировать их сложнее. Например, в кластере 2 (ряд 2, столбец 0) присутствуют туманные слова вроде *work*, *team*, *research* и *environment*. Кластер 12 (ряд 1, столбец 0) также очень абстрактен и состоит из терминов вроде *environment*, *working* и *experience*. Более того, вывод дополнительно усложняется кластерами, которые вообще не представляют реальные навыки.

К примеру, кластер 3 (ряд 0, столбец 0) состоит не из навыков, а из стажа работы — в нем имеются пункты, указывающие необходимый срок трудовой деятельности в данной сфере в годах. Аналогично кластер 6 (ряд 2, столбец 1) также не содержит навыков, а предъявляет требования к определенному уровню образования для приглашения на собеседование. Мы немного ошиблись в своих предположениях — не все пункты представляют реальные профессиональные навыки. Убедиться в том, что была допущена ошибка, можно с помощью выборки пунктов из кластеров 6 и 3 (листинг 17.43).

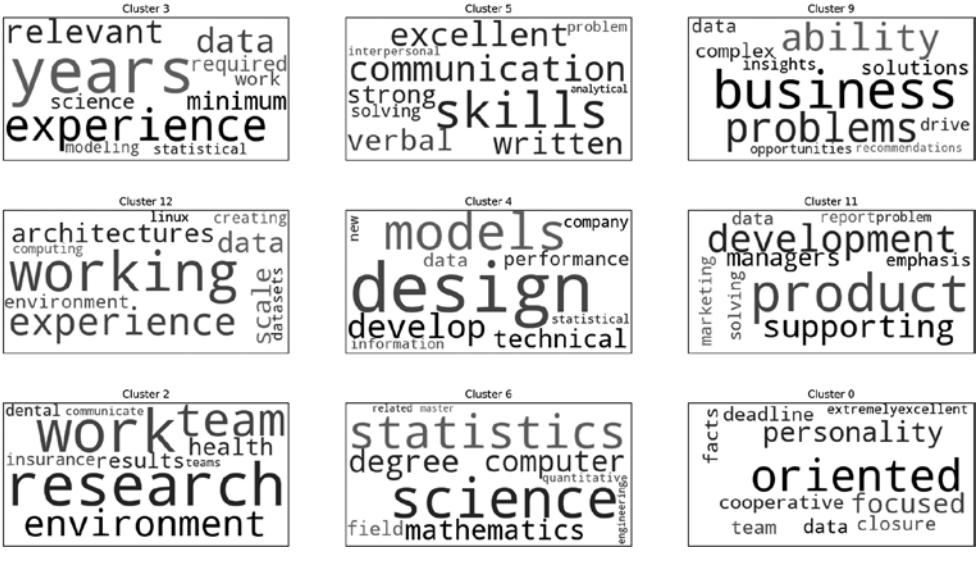


Рис. 17.12. Девять облаков слов, связанных с девятью кластерами личностных качеств, упорядочены по среднему сходству с резюме. Большинство кластеров неконкретны и неинформативны, но кластеры коммуникативных навыков в первом ряду внимания все же заслуживают

Листинг 17.43. Вывод выборочных пунктов из кластеров 6 и 3

```
np.random.seed(1)
for cluster_id in [6, 3]:
    print(f'\nCluster {cluster_id}:')
    print_cluster_sample(cluster_id)

Cluster 6:
MS in a quantitative research discipline (e.g., Artificial Intelligence, Computer
Science, Machine Learning, Statistics, Applied Math, Operations Research)
Master's degree in data science, applied mathematics, or bioinformatics preferred.
PhD degree preferred
Ph.D. in a quantitative discipline (e.g., statistics, computer science, economics,
mathematics, physics, electrical engineering, industrial engineering or other STEM
fields)
```

```
7+ years of experience manipulating data sets and building statistical models,
has advanced education in Statistics, Mathematics, Computer Science or another
quantitative field, and is familiar with the following
    software/tools:

Cluster 3:
Minimum 6 years relevant work experience (if Bachelor's degree) or minimum 3 years
relevant work experience (if Master's degree) with a proven track record in driving
value in a commercial setting using data science skills.
Minimum five (5) years of experience manipulating data sets and building
statistical models, and familiarity with:
5+ years of relevant work experience in data analysis or related field.
(e.g., as a statistician / data scientist / scientific researcher)
3+ years of statistical modeling experience
Data Science: 2 years (Required)
```

Один из кластеров личностных качеств интерпретировать очень легко — кластер 5 (ряд 0, столбец 1) сосредоточен на навыках межличностной коммуникации, как письменной, так и устной. Хорошие коммуникативные навыки очень важны в карьере того, кто работает с данными. Результаты анализа сложных данных точно и корректно должны быть донесены до заинтересованных сторон. Впоследствии на основе убедительно переданной информации ими будут приняты нужные решения. Если же мы не сможем должным образом представить собранные результаты, то весь наш тяжкий труд окажется напрасным.

К сожалению, коммуникативные навыки осваивать трудно. Здесь недостаточно просто почитать книги, необходима практика прямого взаимодействия и сотрудничества с другими людьми. Если вы захотите расширить свои коммуникативные способности, то лучше всего будет подумать о работе с другими аналитиками данных локально или хотя бы удаленно. Выберите проект по обработке данных и завершите его в составе команды. Это позволит вам смело акцентировать в резюме умение работать в коллективе.

17.3.4. Анализ кластеров при других значениях K

Кластеризация методом K-средних при $K = 15$ дала прекрасные результаты. Тем не менее выбор параметра был отчасти произвольным, поскольку мы не могли определить оптимальное значение K. Произвольная природа полученных результатов вызывает некоторые проблемы: возможно, нам просто повезло и другое K не дало бы никакой толковой информации. А может, мы ошиблись с выбором его значения и упустили критически важные кластеры? Проблемой в данном случае является согласованность кластеров. Сколько информативных кластеров сохранится, если изменить K? Чтобы это выяснить, мы сгенерируем их, используя другие значения K. Начнем с установки $K = 25$ и построения результатов в сетке подграфиков размером 5×5 (листинг 17.44; рис. 17.13). Они будут упорядочены на основе сходства их кластеров с нашим резюме.

Листинг 17.44. Визуализация 25 упорядоченных кластеров

```
np.random.seed(0)
cluster_groups = compute_cluster_groups(shrunk_norm_matrix, k=25)
sorted_cluster_groups = sort_cluster_groups(cluster_groups)
plot_wordcloud_grid(sorted_cluster_groups, num_rows=5, num_columns=5)
plt.show()
```

Рис. 17.13. Облака слов, связанные с 25 кластерами навыков. Ранее рассмотренные навыки по-прежнему присутствуют, несмотря на увеличение K. Помимо этого, мы наблюдаем новые технические навыки, достойные упоминания, включая знакомство с базами данных и умение работать с веб-сервисами

В новом выводе большинство ранее полученных кластеров сохранились. К ним относятся кластеры библиотек для аналитики данных (ряд 0, столбец 1), машинного обучения (ряд 1, столбец 2) и коммуникативных способностей (ряд 2, столбец 0). Кроме того, мы получили три полезных кластера технических навыков, которые находятся в первых двух рядах сетки.

- *Кластер 8 (ряд 0, столбец 4)* — сосредоточен на веб-сервисах. С помощью этих инструментов обеспечивается взаимодействие между клиентом и удаленным сервером. В большинстве промышленных конфигураций для аналитики данные хранятся удаленно на сервере и передаются через кастомные API. В Python эти протоколы API обычно создаются при помощи фреймворка Django. Для

специалистов по данным знакомство с этими инструментами желательно, но не обязательно. Узнать больше о веб-сервисах и передачах с помощью API вы можете в книге Майкла Виттига (Michael Wittig) и Андреа Виттига (Andreas Wittig) *Amazon Web Services in Action* (Manning, 2018, https://www.manning.com/books/amazon-web-services-in-action-second-edition), а также в *The Design of Web APIs*[1] Арно Лоре (Arnaud Lauret) (Manning, 2019, https://www.manning.com/books/the-design-of-web-apis).

- *Кластер 23 (ряд 1, столбец 3)* — сосредоточен вокруг различных типов баз данных. Масштабные структурированные данные обычно хранятся в реляционных БД и запрашиваются с помощью языка структурированных запросов (SQL). Однако не все БД реляционные. Иногда данные хранятся в альтернативных неструктурированных хранилищах, таких как MongoDB. Из таких БД данные запрашиваются при помощи языка запросов NoSQL. Понимание различных типов баз данных может очень пригодиться в карьере аналитика. Если вы хотите получше разобраться в этой теме, рекомендую книгу *Understanding Databases* (Manning, 2019, www.manning.com/books/understanding-databases). Если же вас интересует MongoDB, читайте *MongoDB in Action, Second Edition*[2] Кайла Бэнкера (Kyle Banker) и др. (Manning, 2016, www.manning.com/books/mongodb-in-action-second-edition).
- *Кластер 2 (ряд 1, столбец 1)* — в основном содержит инструменты визуализации, не относящиеся к Python, например Tableau и ggplot. Tableau — это платная программа компании Salesforce, обычно ее используют те, кто может позволить себе контракт с Salesforce. Подробнее об этом инструменте можете узнать в книге *Practical Tableau* Райана Слипера (Ryan Sleeper) (O'Reilly, 2018, http://mng.bz/Xrdv). В свою очередь, ggplot — это пакет визуализации данных для языка статистического программирования R. Если вас интересует эта тема, то обратитесь к книге Нины Зумель (Nina Zumel) и Джона Маунта (John Mount) *Practical Data Science with R, Second Edition* (Manning, 2019, www.manning.com/books/practical-data-science-with-r-second-edition).

Наш график содержит также семь новых кластеров, в которых отражены преимущественно общие навыки вроде решения задач (ряд 3, столбец 0) и командной работы (ряд 2, столбец 3). Кроме того, по меньшей мере один из новых кластеров не содержит реальных навыков, например кластер про выгоду от страхования здоровья (ряд 3, столбец 4).

При увеличении K с 15 до 25 сохранились все ранее полученные полезные кластеры и появились интересные новые. Останется ли неизменной стабильность этих кластеров, если сдвинуть K на промежуточное значение 20? Мы это выясним далее, построив облака слов для 20 упорядоченных кластеров в сетке, имеющей четыре ряда и пять столбцов (листинг 17.45; рис. 17.14).

---

[1] *Лоре А.* Проектирование веб-API.

[2] *Бэнкер К.* MongoDB в действии.

Листинг 17.45. Визуализация 20 упорядоченных кластеров

```
np.random.seed(0)
cluster_groups = compute_cluster_groups(shrunk_norm_matrix, k=20)
sorted_cluster_groups = sort_cluster_groups(cluster_groups)
plot_wordcloud_grid(sorted_cluster_groups, num_rows=4, num_columns=5)
plt.show()
```

Рис. 17.14. Облака слов, связанные с 20 кластерами навыков. Большинство ранее выведенных навыков сохранились, но кластер статистического анализа отсутствует

При $K = 20$ большинство ранее полученных полезных кластеров сохранились, включая связанные с библиотеками для аналитики данных (ряд 0, столбец 0), программированием на Python (ряд 0, столбец 3), машинным обучением (ряд 1, столбец 0), коммуникативными навыками (ряд 1, столбец 4), веб-сервисами (ряд 0, столбец 1) и работой с базами данных (ряд 0, столбец 4). Однако кластер инструментов визуализации, не связанных с Python, исчез. Самое же печальное, что пропал и кластер статистического анализа, который присутствовал при $K = 15$ и $K = 25$.

ПРИМЕЧАНИЕ

Кластер статистического анализа был заменен кластером статистических алгоритмов (ряд 0, столбец 2). В нем доминируют три термина: algorithms, clustering и regression. Естественно, сейчас мы уже хорошо знакомы с кластеризацией, но все же техники регрессии в нашем резюме не упоминаются, поскольку с ними мы еще не познакомились. Освоив эти приемы в практическом задании 5, сможем смело внести их в резюме.

Казалось бы стабильный кластер исчез. К сожалению, подобные колебания вполне типичны. Ввиду сложной природы человеческого языка кластеризация текстов чувствительна к изменению параметров, которые из-за возможности широкой трактовки тематик очень трудно подобрать идеально. Кластеры, которые формируются при одном наборе параметров, могут исчезать при их изменении. Если выполнить кластеризацию для всего одного значения K, возникнет риск упустить полезные результаты. Поэтому во время текстового анализа предпочтительнее визуализировать выводы для нескольких значений K. Отталкиваясь от этого, мы далее посмотрим, что произойдет при установке $K = 10$ (листинг 17.46; рис. 17.15).

Листинг 17.46. Визуализация десяти упорядоченных кластеров

```
np.random.seed(0)
cluster_groups = compute_cluster_groups(shrunk_norm_matrix, k=10)
sorted_cluster_groups = sort_cluster_groups(cluster_groups)
plot_wordcloud_grid(sorted_cluster_groups, num_rows=5, num_columns=2)
plt.show()
```

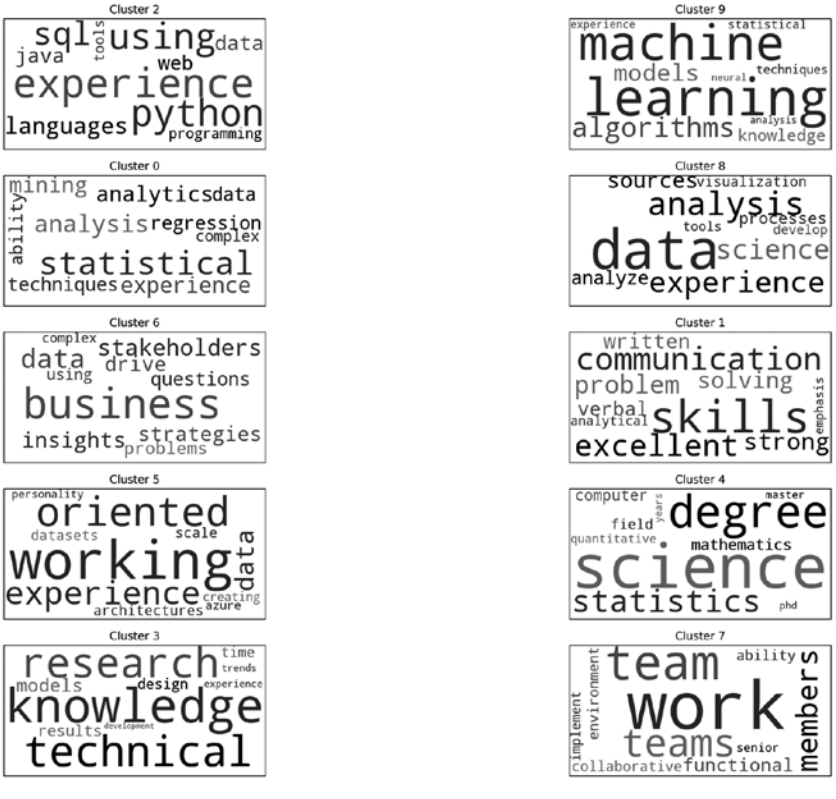


Рис. 17.15. Облака слов, связанные с десятью кластерами навыков. Четыре из ранее рассмотренных навыков сохранились, несмотря на низкое значение K

Десять полученных кластеров оказались довольно ограниченными. Тем не менее четыре из них содержат уже знакомые нам критические навыки: программирование на Python (ряд 0, столбец 0), машинное обучение (ряд 0, столбец 1) и коммуникативные способности (ряд 2, столбец 1). Появился и кластер статистического анализа (ряд 1, столбец 0). На удивление, некоторые из кластеров навыков являются универсальными и формируются даже тогда, когда *K* сильно изменяется. Несмотря на некоторую стохастичность нашей кластеризации, уровень согласованности сохраняется. Таким образом, наблюдаемые результаты оказываются не просто случайными выводами — это конкретные паттерны, которые мы обнаружили среди сложных, запутанных реальных текстов.

До сих пор наши наблюдения ограничивались 60 наиболее релевантными объявлениями. Однако, как мы видели, в этом наборе данных присутствует шум. А что произойдет, если расширить анализ до 700 объявлений? Изменятся ли результаты или останутся теми же? Далее мы это выясним.

17.3.5. Анализ 700 наиболее релевантных вакансий

Начнем с подготовки `sorted_df_jobs[:700].Bullets` для кластеризации (листинг 17.4).

1. Извлечем все пункты и удалим повторы.
2. Векторизуем тексты пунктов.
3. Уменьшим размерность векторизованных текстов и нормализуем полученную матрицу.

Листинг 17.47. Подготовка sorted_df_jobs[:700] для анализа кластеров

```
np.random.seed(0)
total_bullets_700 = set()
for bullets in sorted_df_jobs[:700].Bullets:
    total_bullets_700.update([bullet.strip()
                              for bullet in bullets])

total_bullets_700 = sorted(total_bullets_700)
vectorizer_700 = TfidfVectorizer(stop_words='english')
tfidf_matrix_700 = vectorizer_700.fit_transform(total_bullets_700)
shrunk_norm_matrix_700 = shrink_matrix(tfidf_matrix_700)
print(f"We've vectorized {shrunk_norm_matrix_700.shape[0]} bullets")

We've vectorized 10194 bullets
```

Мы векторизовали 10 194 пункта списков. Теперь сгенерируем для этих результатов график локтя (листинг 17.48). Исходя из прежних наблюдений, мы не ожидаем, что он будет особо информативен, но все равно создаем его, чтобы сохранить согласованность с предыдущим анализом (рис. 17.16).

Листинг 17.48. Построение графика локтя для 10 194 пунктов

```
np.random.seed(0)
generate_elbow_plot(shrunk_norm_matrix_700)
plt.show()
```

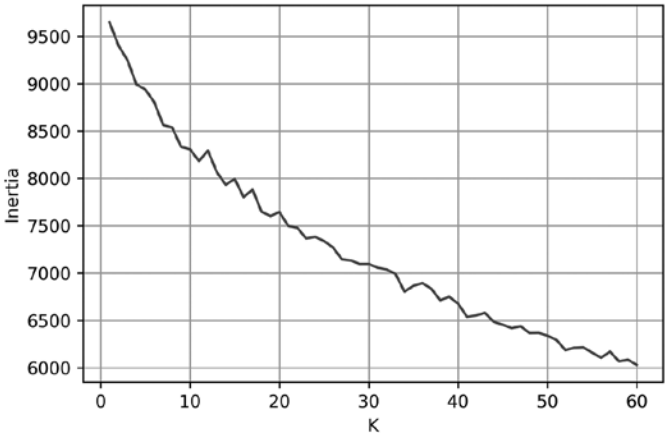


Рис. 17.16. График локтя, сгенерированный с использованием пунктов списков из 700 наиболее релевантных объявлений. Точное расположение локтя определить сложно

Как и ожидалось, точное расположение локтя на графике неочевидно. Он распределен между $K = 10$ и $K = 25$. Эту неопределенность мы устраним, установив $K = 20$. Сгенерируем и визуализируем 20 кластеров (листинг 17.49) и при необходимости скорректируем значение K для сравнительной кластеризации (рис. 17.17).

ВНИМАНИЕ

Как говорилось в главе 15, результат выполнения метода K-средних для огромных матриц, содержащих $10\,000 \times 100$ элементов, на разных компьютерах может быть различным. Итоги выполненной вами кластеризации могут отличаться от вывода, приведенного здесь, но на их основе у вас все равно должны получиться аналогичные заключения.

Листинг 17.49. Визуализация 20 упорядоченных кластеров для 10 194 пунктов

```
np.random.seed(0)
cluster_groups_700 = compute_cluster_groups(shrunk_norm_matrix_700, k=20,
                                            bullets=total_bullets_700)
bullet_cosine_similarities = compute_bullet_similarity(total_bullets_700)
sorted_cluster_groups_700 = sort_cluster_groups(cluster_groups_700)
plot_wordcloud_grid(sorted_cluster_groups_700, num_rows=4, num_columns=5,
                    vectorizer=vectorizer_700,
                    tfidf_matrix=tfidf_matrix_700)
```

Повторно вычисляет bullet_cosine_similarities для упорядочения

Нужно передать обновленные матрицу TF-IDF и vectorizer в функцию построения облаков слов

Рис. 17.17. Облака слов, сгенерированные кластеризацией более чем 10 000 пунктов. Несмотря на десятикратное увеличение количества пунктов, итоговые навыки в основном остались прежними

Результат кластеризации очень похож на полученный ранее. Сохранились ключевые информативные кластеры, которые мы наблюдали при анализе 60 объявлений, в том числе отражающие навыки работы с библиотеками для аналитики данных (ряд 0, столбец 0), статистический анализ (ряд 0, столбец 1), программирование на Python (ряд 0, столбец 4), машинное обучение (ряд 0, столбец 3) и коммуникативные навыки (ряд 1, столбец 3). Есть, конечно, и небольшие изменения, но в основном результат остался прежним.

ПРИМЕЧАНИЕ

К интересным изменениям относится появление обобщенного кластера визуализации (ряд 0, столбец 2). Он включает различные инструменты визуализации, в том числе Matplotlib. Кроме того, в его облаке слов упоминается бесплатная JavaScript-библиотека D3.js. С ее помощью некоторые аналитики данных создают интерактивные визуализации. Подробнее узнать об этой библиотеке можно в книге Элайджи Микса (Elijah Meeks) D3.js in Action, Second Edition (Manning, 2017, www.manning.com/books/d3js-in-action-second-edition).

Некоторые навыки встречаются в объявлениях постоянно. Они не особо чувствительны к выбранному нами порогу релевантности, поэтому их можно выявить, даже если порог не будет определен.

17.4. ЗАКЛЮЧЕНИЕ

Теперь мы готовы обновить образец нашего резюме. Первым делом нужно подчеркнуть навыки работы с Python. Для этого достаточно будет одной строки, сообщающей, что мы *профессионально владеем Python*. Кроме того, следует заявить о наших коммуникативных навыках. Как показать, что мы хорошо умеем взаимодействовать с другими людьми? Вопрос непростой. Одного указания, что мы умеем *ясно доносить сложные результаты до различной аудитории*, будет недостаточно. Вместо этого следует описать личный проект, в котором мы делали следующее:

- сотрудничали с членами команды, решая сложную задачу по обработке данных;
- в письменной или устной форме доносили сложные результаты до аудитории, не обладающей техническими знаниями.

ПРИМЕЧАНИЕ

Если вы работали над подобным проектом, то определенно должны внести это в резюме. Если же нет, то желательно специально искать возможности в таком проекте поучаствовать. Полученные в итоге навыки окажутся бесценными и повысят шанс успешного найма.

Помимо этого, прежде чем мы закончим свое резюме, нужно разобраться с остальными пробелами в навыках. Опыт работы в сфере машинного обучения очень важен для успешной карьеры аналитика данных. Мы еще не изучали машинное обучение, но займемся этим в следующем практическом задании. После этого можно будет гордо описать обретенные навыки в резюме.

Наконец, в нем следует продемонстрировать опыт работы с инструментами для скачивания и сохранения удаленных данных. К ним относятся базы данных и веб-сервисы. И хотя их использование выходит за область данной книги, освоить эти инструменты можно с помощью независимого практического проекта. Умение работать с базами данных и веб-сервисами не всегда необходимо для получения должности в сфере обработки данных, тем не менее потенциальные работодатели приветствуют даже небольшие такие навыки у своих соискателей.

РЕЗЮМЕ

- Текстовые данные нельзя анализировать вслепую. Всегда нужно отбирать и прочитывать некоторые тексты, прежде чем прогонять какие-либо алгоритмы. Это особенно актуально для HTML-файлов, в которых теги разграничивают в тексте уникальные сигналы. Отобразив выборку объявлений о вакансиях, мы выяснили, что уникальные навыки приводятся в каждом HTML-файле в виде маркированных списков. Если бы мы вслепую кластеризовали тело каждого файла, то результаты оказались бы менее информативными.

- Кластеризация текстов — непростая задача. В редких случаях есть некое идеальное количество кластеров, потому что трактовка языка может быть очень расплывчатой, а значит, расплывчаты и границы тематик. Но, несмотря на эту неопределенность, ряд тем неизменно проявляется при выводе разного количества кластеров. Поэтому, даже если график локтя не раскрывает точное число кластеров, у такой ситуации есть решение. Выполнение анализа при нескольких параметрах кластеризации позволит выявить темы, устойчиво присутствующие в тексте.
- Выбирать значения параметров не всегда легко. Эта проблема выходит далеко за рамки простой кластеризации. При выборе порога релевантности мы метались между двумя значениями — 60 и 700. Ни одно из них не выглядело более удачным, чем другое, поэтому мы попробовали оба. В науке о данных некоторые задачи не имеют идеального значения порога или параметров. Тем не менее не следует сдаваться и игнорировать подобные задачи. Напротив, нужно экспериментировать. Ученые получают информацию, анализируя результаты при разных входных параметрах. Так что и мы, будучи специалистами в сфере данных, можем получать бесценные результаты, перебирая и корректируя используемые параметры.

Практическое задание 5

Прогнозирование будущих знакомств на основе данных социальной сети

УСЛОВИЕ ЗАДАЧИ

Добро пожаловать во FriendHook, крутейший стартап Кремниевой долины. FriendHook — это приложение социальной сети для студентов колледжа. Чтобы к нему подключиться, студент должен отсканировать свой учебный ID, подтвердив принадлежность к колледжу. После подтверждения новый участник может создать профиль FriendHook, указав в нем свое общежитие и учебные интересы. Создав профиль, студент может отправлять *запросы на дружбу* другим студентам своего колледжа, которые, в свою очередь, имеют возможность его запрос принять либо отклонить. При подтверждении такого запроса двое студентов официально становятся *друзьями во FriendHook*. Используя новую цифровую связь, друзья могут обмениваться фотографиями, совместно трудиться над курсовыми и держать друг друга в курсе последних слухов кампуса.

Приложение FriendHook — это хит. Оно применяется в сотнях колледжей по всему миру. Пользовательская база растет, а с ней и сама компания. Вы первый специалист по данным, нанятый стартапом FriendHook, и вашим начальным заданием будет работа над алгоритмом рекомендации друзей в этой сети.

Внедрение алгоритма рекомендации друзей друзей

Иногда пользователи FriendHook испытывают сложности с поиском в цифровом приложении своих друзей из реальной жизни. Чтобы поспособствовать образованию новых связей, команда инженеров реализовала простой механизм рекомендации друзей. Раз в неделю все пользователи получают электронное письмо, рекомендующее нового друга, которого в их сети еще нет. Получатель может проигнорировать это письмо либо отправить потенциальному другу запрос, который тот примет или отклонит.

Сейчас этот механизм рекомендаций следует простому *алгоритму рекомендации друга моего друга*. Работает алгоритм так. Предположим, что мы хотим порекомендовать нового друга студенту A. Для этого выбираем случайного студента B, который уже является другом студента A. После этого случайно выбираем студента C, который является другом студента B, но не студента A, и рекомендуем его последнему в качестве друга (рис. П35.1).

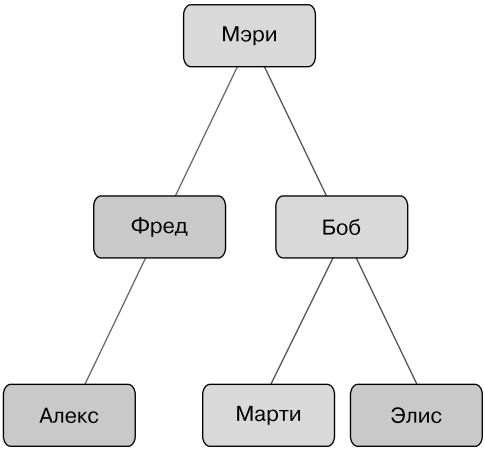


Рис. П35.1. Алгоритм рекомендации друга моего друга в действии. У Мэри есть два друга — Фред и Боб. Случайным образом из них выбирается один (Боб). У Боба тоже есть два друга — Марти и Элис. Ни Марти, ни Элис в друзьях у Мэри не числятся. Далее случайно выбирается один из друзей ее друга (Марти), и Мэри получает письмо с предложением отправить Марти запрос на дружбу

По сути, этот алгоритм предполагает, что друг твоего друга тоже вполне может являться твоим другом. Это разумное, но несколько примитивное предположение. Какова вероятность того, что это предположение окажется верным? Никто не знает. Однако это предстоит выяснить вам, как первому аналитику данных в компании. Вам поставили задачу построить модель, прогнозирующую поведение студента в ответ на рекомендации алгоритма.

Прогнозирование поведения пользователя

Механизм рекомендации друга моего друга может вызвать три типа поведения:

- пользователь прочитывает пришедшую по почте рекомендацию и либо ее отклоняет, либо игнорирует;
- пользователь отправляет по этой рекомендации запрос на дружбу, который его получатель отклоняет либо игнорирует;
- пользователь отправляет по этой рекомендации запрос на дружбу, который получатель принимает, и в сети FriendHook устанавливается новая связь.

Можно ли спрогнозировать эти три варианта поведения? Технический директор FriendHook хочет, чтобы вы в этом разобрались. Он предоставил вам данные сети FriendHook из случайно выбранного университета. Они охватывают всех его пользователей, включая их зарегистрированное поведение в ответ на еженедельные рекомендации друзей. Помимо этого, там есть информация из профиля каждого пользователя, раскрывающая его специальность и название общежития, где он проживает. Для защиты личных данных пользователей эта информация зашифрована (чуть позже об этом будет сказано подробнее). Наконец, данные включают сеть существующих связей FriendHook в университете, составленную как раз перед рассылкой рекомендаций.

Ваша задача — создать модель, прогнозирующую поведение пользователей на основе их профилей и данных из социальной сети. Она должна обобщаться как на колледжи, так и на университеты. Причем обобщаемость очень важна — модель, которую нельзя применить в других колледжах, окажется бесполезной для команды разработчиков. Представьте, к примеру, модель, которая точно прогнозирует поведение в одном или двух общежитиях случайного университета. Иными словами, для составления точных прогнозов ей требуются конкретные названия общежитий. Такая модель окажется бесполезной, поскольку в других университетах общежития будут называться иначе. В идеале она должна обобщаться на все общежития всех университетов планеты.

После создания обобщенной модели нужно будет проанализировать ее внутренние функции. Ваша задача — разобраться, как жизнь университета способствует образованию новых связей во FriendHook.

Цели проекта амбициозны, но вполне выполнимы. Их можно достичь, придерживаясь следующего алгоритма действий.

1. Загрузить три набора данных, описывающих поведение пользователей, их профили и сеть друзей. Изучить все наборы и при необходимости очистить их от шума.

2. Создать и оценить модель, прогнозирующую поведение пользователей на основе их профилей и имеющихся дружеских связей. Это задание по желанию можно разбить на две части: создать модель, задействуя только сеть друзей, а затем добавить информацию профилей и проверить, повысит ли это ее эффективность.
3. Определить, достаточно ли хорошо модель обобщается на другие университеты.
4. Проанализировать внутренние механизмы модели, чтобы лучше понять поведение студентов.

ОПИСАНИЕ НАБОРА ДАННЫХ

Наши данные содержат три файла, хранящиеся в каталоге `friendhook`. Эти файлы являются CSV-таблицами и называются `Profiles.csv`, `Observations.csv` и `Friendships.csv`. Разберем каждую таблицу по-отдельности.

Таблица Profiles

`Profies.csv` содержит информацию профилей всех студентов выбранного университета. Она распределена по шести столбцам: `Profile_ID`, `Sex`, `Relationship_Status`, `Major`, `Dorm` и `Year`. Команда FriendHook очень обеспокоена сохранением конфиденциальности данных каждого студента, поэтому вся информация профилей тщательно зашифрована.

Используемый во FriendHook алгоритм шифрования получает описательный текст и возвращает уникальный зашифрованный 12-символьный *хеш-код*. Предположим, что студент указал в качестве своей специализации физику. Слово *physics* в итоге шифруется и заменяется хеш-кодом вроде `b90a1221d2bc`. Если другой студент укажет в качестве специализации историю, то будет сгенерирован другой хеш-код (например, `983a9b1dc2ef`). Это позволяет проверять, обучаются ли двое студентов по одному направлению, даже не зная его точного названия. Из соображений сохранения личной тайны зашифрованы были все шесть столбцов профиля. Давайте рассмотрим их подробнее.

- `Profile_ID` — уникальный идентификатор для отслеживания каждого студента. Он может ассоциироваться с поведением пользователей в таблице `Observations`, а также со связями в сети FriendHook через таблицу `Friendships`.
- `Sex` — это необязательное поле описывает пол студента как `Male` или `Female`. Студенты, не желающие указывать свой пол, могут оставлять это поле незаполненным. Такие поля сохраняются в таблице в виде пустых значений.
- `Relationship_Status` — дополнительное поле, указывающее статус личных отношений студента. Здесь каждый студент может выбирать одну из трех категорий: `Single` (Свободен) `In a Relationship` (В отношениях) или `It's Complicated` (Все

сложно). У студентов также есть возможность оставить это поле незаполненным. Такие поля сохраняются в таблице в виде пустых значений.

- `Major` — выбранная специализация, например физика, история, экономика и т. д. Это поле необходимо для активации аккаунта FriendHook. Студенты, еще не определившиеся со специализацией, могут выбирать опцию `Undecided`.
- `Dorm` — название общежития студента. Это поле необходимо для активации аккаунта FriendHook. Учащиеся, проживающие вне кампуса, могут выбирать опцию `Off-Campus Housing`.
- `Year` — год обучения. Здесь выбирается один из четырех вариантов: `Freshman` (Первый курс), `Sophomore` (Второй курс), `Junior` (Третий курс) или `Senior` (Выпускной курс).

Таблица Observations

Таблица `Observations.csv` содержит зарегистрированное поведение пользователей в ответ на полученные по почте рекомендации друзей. Состоит она из пяти полей.

- `Profile_ID` — ID пользователя, получившего рекомендацию. Этот ID соответствует ID профиля в таблице `Profiles`.
- `Selected_Friend` — уже имеющийся друг пользователя в столбце `Profile_ID`.
- `Selected_Friend_of_Friend` — случайно выбранный друг `Selected_Friend`, который еще не является другом `Profile_ID`. Этот случайный друг друга рекомендуется пользователю в качестве друга в рассылке электронной почты.
- `Friend_Request_Sent` — булев столбец, который устанавливается `True`, если пользователь отправляет предлагаемому другу друга запрос на дружбу, и `False` — в противном случае.
- `Friend_Request_Accepted` — булев столбец, который устанавливается `True`, только если отправленный пользователем запрос на дружбу был принят.

В этой таблице хранится все зарегистрированное поведение пользователей в отношении еженедельных рекомендаций друзей. Наша цель — спрогнозировать булевы результаты последних двух столбцов таблицы на основе данных профиля и социальной сети.

Таблица Friendships

`Feirndships.csv` содержит сеть FriendHook выбранного университета. Эта сеть использовалась в качестве ввода для алгоритма рекомендации друзей. Таблица `Friendships` содержит всего два столбца: `Friend A` и `Friend B`. В них указаны ID профилей, сопоставляющиеся со столбцами `Profile_ID` в таблицах `Profiles`

и `Observations`. Каждая строка соответствует паре друзей во FriendHook. Например, первая строка содержит ID `b8bc075e54b9` и `49194b3720b6`. Из этих идентификаторов можно сделать вывод, что связанные с ними студенты имеют дружескую связь в этой сети. По ID можно находить профиль каждого студента. В итоге профили позволяют нам выяснять, совпадает ли учебная специальность студентов и проживают ли они в одном общежитии.

ОБЗОР

Чтобы решить поставленную задачу, нам нужно уметь:

- анализировать данные сети с помощью Python;
- находить кластеры друзей в социальных сетях;
- обучать модели МО с учителем и оценивать их;
- анализировать внутренние механизмы обученных моделей для получения из данных полезной информации.

18 Знакомство с теорией графов и анализом сетей

В этой главе

- ✓ Представление разных наборов данных в виде сетей.
- ✓ Анализ сетей с помощью библиотеки NetworkX.
- ✓ Оптимизация маршрутов в сети.

Исследование связей потенциально может принести миллиарды долларов. В 1990-е годы двое выпускников проанализировали свойства взаимосвязанных веб-страниц и в результате основали Google. В начале 2000-х один студент начал в цифровом виде отслеживать связи между людьми, в результате чего основал Facebook. Анализ связей может привести не только к обретению неслыханного богатства, но и к спасению бесчисленного числа жизней. Отслеживание связей между белками в раковых клетках способно помочь в разработке целевых лекарств, излечивающих рак. Анализ связей между подозреваемыми в терроризме может раскрыть и предотвратить их коварные планы. И у этих, казалось бы, совершенно разных сценариев есть одна общая деталь: их можно изучить с помощью математической концепции, которую одни называют *теорией сетей*, а другие — *теорией графов*.

Теория графов занимается исследованием связей между объектами. Этими объектами может быть что угодно: связанные отношениями люди, связанные ссылками веб-страницы или соединяемые дорогами города. Набор объектов и их разветвленные связи называется *сетью* или *графом*, смотря кого спросить. Инженеры предпочитают термин *«сеть»*, а математики — *«граф»*. Мы же для своих целей

будем использовать оба. Графы — это простые абстракции, отражающие сложность нашего пронизанного взаимосвязями мира. При этом свойства графов остаются на удивление согласованными в системах, относящихся как к обществу, так и к природе. Теория графов — это структура для математического отслеживания согласованностей. Она совмещает в себе идеи из разных разделов математики, включая теорию вероятностей и матричный анализ. С их помощью можно делать полезные выводы из реальных данных, начиная с ранжирования выдачи страниц поискового движка и заканчивая кластеризацией социальных кругов. Так что для качественной аналитики данных знание теории графов необходимо.

В следующих двух главах познакомимся с принципами теории графов, взяв за основу уже изученные концепции науки о данных и библиотеки. Мы начнем с решения простых задач, изучая графы из ссылок на веб-страницы и дорог. Затем в главе 19 используем более продвинутые приемы для обнаружения кластеров друзей в социальных графах. Однако первым будет более простое задание, требующее ранжировать сайты по популярности.

18.1. ИСПОЛЬЗОВАНИЕ БАЗОВОЙ ТЕОРИИ ГРАФОВ ДЛЯ РАНЖИРОВАНИЯ САЙТОВ ПО ПОПУЛЯРНОСТИ

В Интернете существует множество сайтов, посвященных науке о данных, одни популярнее других. Предположим, что вы хотите оценить наиболее популярные из них с помощью общедоступных данных. Это исключает возможность скрытного отслеживания данных трафика. Что же делать? Теория сетей предлагает простой способ ранжирования сайтов на основе их публичных ссылок. Чтобы понять, как это работает, мы построим простую сеть, состоящую из двух сайтов, посвященных науке о данных, — руководства по NumPy и руководства по SciPy. В теории графов эти сайты будут называться *узлами* графа. Узлы — это точки сети, которые могут формировать между собой связи, называемые *ребрами*. Два наших сайта-узла сформируют ребро, если один будет ссылаться на другой.

Мы начнем с сохранения двух узлов в двухэлементном списке (листинг 18.1). Эти элементы будут представлять `'NumPy'` и `'SciPy'` соответственно.

Листинг 18.1. Определение списка узлов

```
nodes = ['NumPy', 'SciPy']
```

Предположим, что на сайте SciPy говорится про зависимости NumPy и при этом присутствует ссылка на страницу NumPy. Щелчок на ней будет переносить читателя с сайта, представленного `nodes[1]`, на сайт, представленный `nodes[0]`. Мы рассматриваем эту связь как ребро, ведущее от индекса 1 к индексу 0 (рис. 18.1).

Это ребро можно выразить как кортеж `(1, 0)`. Здесь мы формируем ребро, сохраняя `(1,0)` в списке `edges` (листинг 18.2).

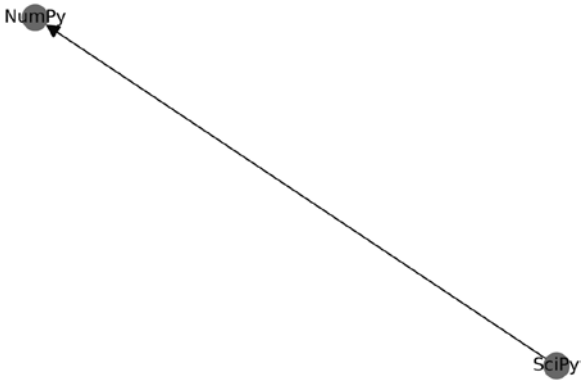


Рис. 18.1. Два сайта, NumPy и SciPy, представлены в виде кружков — узлов. Направленное ребро ведет от SciPy к NumPy, указывая на наличие между этими сайтами направленной связи. Если NumPy и SciPy сохранены в виде индексов узлов 0 и 1, их ребро можно выразить как кортеж (1, 0). Далее мы узнаем, как генерировать схему сети, изображенной на этом рисунке

Листинг 18.2. Определение списка ребер

```
edges = [(1, 0)]
```

Единственное ребро `(1, 0)` представляет связь, которая переносит пользователя из `nodes[1]` в `nodes[0]`. У этого ребра есть определенное направление, в связи с чем оно называется *направленным ребром*. Графы, содержащие направленные ребра, называются *направленными графами*. В направленном графе ребро `(i, j)` не равнозначно ребру `(j, i)`. Присутствие `(i, j)` в списке `edges` не подразумевает присутствия в нем `(j, i)`. К примеру, в нашей сети страница NumPy пока не имеет ссылки на страницу SciPy, значит, кортеж этого ребра в списке `edges` отсутствует.

Располагая списком направленных ребер `edges`, можно легко проверить, имеет ли страница по индексу `i` ссылку на страницу по индексу `j`. Такая связь существует, если `(i, j) in edges` равно `True`. Таким образом, можно определить однострочную функцию `edge_exists`, проверяющую наличие ребер между индексами `i` и `j` (листинг 18.3).

Листинг 18.3. Проверка наличия ребра

```
def edge_exists(i, j): return (i, j) in edges

assert edge_exists(1, 0)
assert not edge_exists(0, 1)
```

Функция `edge_exists` работает, но она неэффективна. Она должна обходить список, проверяя наличие ребра. Такой обход не проблема для списка, содержащего одно ребро. А вот если увеличить размер нашей сети до 1000 страниц, тогда количество ребер в списке может достичь миллиона. Обход такого списка будет вычислительно неоправданным, так что нам нужно альтернативное решение.

Один из вариантов — сохранять наличие или отсутствие каждого ребра `(i, j)` в *i*-й строке и *j*-м столбце таблицы. По сути, можно построить таблицу `t`, в которой `t[i][j] = edge_exists(i, j)`, в результате чего поиск ребер станет мгновенным. Более того, эту таблицу можно представить как двухмерный двоичный массив, если сохранять `not edge_exists(i, j)` как `0`, а `edge_exists(i, j)` как `1`, что позволит представить наш граф в виде двоичной матрицы `M`, в которой `M[i][j] = 1`, если ребро между узлами `i` и `j` существует. Такое матричное представление сети называется *матрицей смежности*. Далее мы вычислим и выведем матрицу смежности для нашего направленного графа с двумя узлами и одним ребром. Изначально матрица будет содержать только `0`. Затем мы переберем каждое ребро `(i, j)` в `edges` и установим `adjacency_matrix[i][j]` на `1` (листинг 18.4).

Листинг 18.4. Отслеживание узлов и ребер с помощью матрицы

```
import numpy as np
adjacency_matrix = np.zeros((len(nodes), len(nodes)))
for i, j in edges:
    adjacency_matrix[i][j] = 1

assert adjacency_matrix[1][0]
assert not adjacency_matrix[0][1]

print(adjacency_matrix)
```

```
[[0. 0.]
 [1. 0.]]
```

Вывод матрицы позволяет увидеть представленные в сети ребра. Помимо этого, можно обнаружить потенциальные ребра, которых в сети нет. К примеру, мы явно видим ребро, ведущее от узла 1 к узлу 0. При этом возможные ребра `(0, 0)`, `(0, 1)` и `(1, 1)` в графе отсутствуют. В нем также нет ссылки, ведущей от узла 0 к узлу 0, то есть страница NumPy на саму себя не ссылается, хотя в теории могла бы. Можно представить плохо построенную страницу, где гиперссылка указывает на саму себя, — такая ссылка окажется бесполезной, поскольку щелчок на ней будет вести вас туда, откуда вы начали, хотя сам по себе такой вариант возможен. В теории графов подобные ссылающиеся сами на себя страницы называются *петлями*. В следующей главе мы познакомимся с алгоритмом, который можно улучшить добавлением петель, но пока ограничим анализ ребрами между парами разных узлов.

Добавим недостающее ребро от узла 0 к узлу 1. Это будет означать, что теперь страница NumPy ссылается на страницу SciPy (листинг 18.5).

Листинг 18.5. Добавление ребра в матрицу смежности

```
adjacency_matrix[0][1] = 1
print(adjacency_matrix)

[[0. 1.]
 [1. 0.]]
```

Предположим, что мы хотим расширить сеть сайтов, добавив два ресурса по науке о данных, посвященные Pandas и Matplotlib. Их добавление увеличит число узлов с двух до четырех, значит, нам потребуется также расширить матрицу смежности с размера 2×2 до 4×4. В ходе этого мы сохраним и все существующие связи между узлом 0 и узлом 1. К сожалению, в NumPy сложно изменять размер матрицы с сохранением всех существующих в ней значений — в эту библиотеку не закладывалось удобство обработки растущих массивов, форма которых постоянно увеличивается. Это противоречит разрастающейся природе Интернета, в котором непрерывно появляются новые сайты. Следовательно, NumPy не лучший инструмент для анализа растущих сетей. Как же быть?

ПРИМЕЧАНИЕ

NumPy неудобна для отслеживания добавления узлов и ребер. Тем не менее, как уже говорилось, эта библиотека необходима для выполнения матричного умножения. В следующей главе мы будем умножать матрицу смежности для анализа социальных графов, то есть использование NumPy окажется необходимым для продвинутого анализа сетей. Пока же с целью упрощения построения сетей мы будем опираться на альтернативную библиотеку Python.

Нужно переключиться на другую библиотеку Python, а именно NetworkX, которая позволяет с легкостью изменять сети. Помимо этого, она предоставляет дополнительные полезные возможности, включая визуализацию сетей. Так что далее мы перейдем к анализу сайтов с помощью NetwotkX.

18.1.1. Анализ веб-сетей при помощи NetworkX

Начнем с установки NetworkX (листинг 18.1), после чего по общепринятому соглашению использования этой библиотеки импортируем `networkx` как `nx`.

ПРИМЕЧАНИЕ

Для установки NetwotkX выполните в терминале pip install networkx.

Листинг 18.6. Импорт библиотеки NetworkX

```
import networkx as nx
```

Теперь используем `nx` для генерации направленного графа. В NetworkX направленные графы отслеживаются с помощью класса `nx.DiGraph`. Вызов `nx.DiGraph()`

инициализирует новый объект направленного графа, содержащий ноль узлов и ноль ребер. Код листинга 18.7 инициализирует этот граф. Следуя соглашению NetworkX, мы называем инициализированный граф `G`.

Листинг 18.7. Инициализация объекта направленного графа

```
G = nx.DiGraph()
```

Теперь мы медленно расширим этот направленный граф, начав с добавления одного узла. NetworkX позволяет добавлять узлы в объект графа с помощью метода `add_node`. Вызов `G.add_node(0)` приведет к созданию одного узла, чей индекс в матрице смежности будет равен `0` (листинг 18.8). Для просмотра этой матрицы нужно выполнить `nx.to_numpy_array(G)`.

ВНИМАНИЕ

Метод add_node всегда расширяет матрицу смежности графа на один узел. Это расширение происходит независимо от ввода метода. Поэтому G.add_node(1000) также создает узел, размещенный в матрице смежности по индексу 0. Однако этот узел будет отслеживаться в том числе с помощью вторичного индекса 1000, что неизбежно приведет к путанице. В этом случае желательно обеспечить, чтобы численные входные данные для add_node соответствовали индексам, добавляемым в матрицу смежности.

Листинг 18.8. Добавление узла в объект графа

```
G.add_node(0)
print(nx.to_numpy_array(G))
```

```
[[0.]]
```

Добавленный узел связан со страницей NumPy. Эту связь можно записать явно, выполнив `G.nodes[0]['webpage'] = 'NumPy'`. Тип данных `G.nodes` представляет собой особый класс для отслеживания всех узлов в `G`. Он структурирован в виде списка. Выполнение `G[i]` ведет к возвращению словаря атрибутов, связанных с узлом в `i`. Эти атрибуты помогут нам отслеживать идентичность узла. В данном случае мы хотим присвоить ему веб-страницу, поэтому отображаем значение в `G.nodes[i]['webpage']`.

Код листинга 18.9 перебирает `G.nodes` и выводит словарь атрибутов в `G.nodes[i]`. Начальный ввод представляет один узел, чей словарь атрибутов пуст: мы присваиваем этому узлу веб-страницу и снова выводим его словарь.

Листинг 18.9. Добавление атрибута существующему узлу

```
def print_node_attributes():
    for i in G.nodes:
        print(f"The attribute dictionary at node {i} is {G.nodes[i]}")
```

```
print_node_attributes()
G.nodes[0]['webpage'] = 'NumPy'
print("\nWe've added a webpage to node 0")
print_node_attributes()

The attribute dictionary at node 0 is {}

We've added a webpage to node 0
The attribute dictionary at node 0 is {'webpage': 'NumPy'}
```

Атрибуты можно присваивать непосредственно при добавлении узла в граф. Для этого нужно лишь передать `attribute=some_value` в метод `G.add_node`. К примеру, мы собираемся вставить узел с индексом 1, связанный с веб-страницей SciPy. Выполнение `G.add_node(1, webpage='SciPy')` приведет к добавлению этого узла и его атрибута (листинг 18.10).

Листинг 18.10. Добавление узла с атрибутом

```
G.add_node(1, webpage='SciPy')
print_node_attributes()

The attribute dictionary at node 0 is {'webpage': 'NumPy'}
The attribute dictionary at node 1 is {'webpage': 'SciPy'}
```

Заметьте, что можно вывести все узлы вместе с их атрибутами, просто выполнив `G.nodes(data=True)` (листинг 18.11).

Листинг 18.11. Вывод узлов вместе с их атрибутами

```
print(G.nodes(data=True))

[(0, {'webpage': 'NumPy'}), (1, {'webpage': 'SciPy'})]
```

А теперь добавим веб-ссылку из узла 1 (SciPy) на узел 0 (NumPy). При использовании направленного графа можно внести в него ребро, ведущее от `i` к `j`, выполнив `G.add_edge(i, j)` (листинг 18.12).

Листинг 18.12. Добавление ребра в объект графа

```
G.add_edge(1, 0)
print(nx.to_numpy_array(G))

[[0. 0.]
 [1. 0.]]
```

Из полученной матрицы смежности видно ребро, идущее от узла 1 к узлу 0. К сожалению, наш вывод матрицы по мере добавления других узлов станет очень громоздким. Отслеживание 1 и 0 в двухмерной таблице — это не самый интуитивный способ отображения сети. А что, если вместо этого построить сеть непосредственно?

Два наших узла можно отобразить в виде двух точек в двухмерном пространстве, а ребро — в виде отрезка, их соединяющего. Подобный график легко сгенерировать с помощью Matplotlib. Именно поэтому наш объект `G` имеет встроенный метод `draw()` для построения графа с помощью библиотеки Matplotlib. Для визуализации графа мы вызываем `G.draw()` (листинг 18.13; рис. 18.2).

Листинг 18.13. Построение объекта графа

```
import matplotlib.pyplot as plt
np.random.seed(0)
nx.draw(G)
plt.show()
```

Расположение узлов определяется с помощью рандомизированного алгоритма. Для обеспечения согласованной визуализации мы задаем при рандомизации начальное число

Согласно требованиям Matplotlib, для отображения результатов нужно вызвать plt.show()

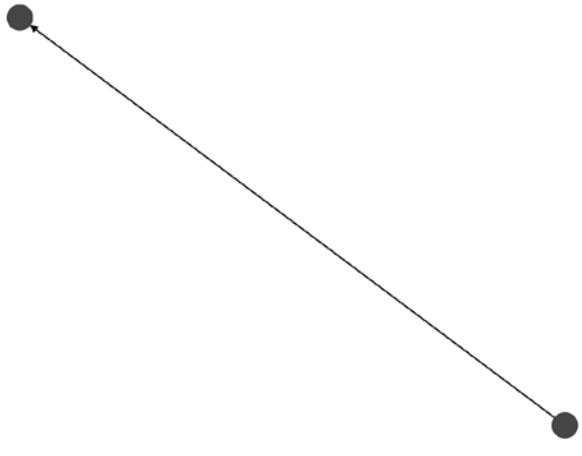


Рис. 18.2. Визуализированный направленный граф из двух узлов. Не особо приметная стрелка направлена от нижнего узла к верхнему

Получившемуся графу однозначно не помешает доработка. Во-первых, нужно увеличить стрелку, что можно сделать с помощью параметра `arrowsize`. Передав `arrowsize=20` в `G.draw`, мы удвоим длину и ширину стрелки. Нужно также добавить к узлам метки. Это можно сделать с помощью параметра `labels`, который получает словарь сопоставлений между ID узлов и нужными метками.

Код листинга 18.14 генерирует это сопоставление, выполняя `{i:G.nodes[i]['webpage'].for i in G.nodes}`, после чего заново строит сеть с метками узлов и увеличенной стрелкой (рис. 18.3).

ПРИМЕЧАНИЕ

В качестве дополнения можно изменить размер узла, передав в nx.draw параметр node_size, но пока наши узлы имеют подобающий размер, определяемый значением 300.

Листинг 18.14. Корректировка отображения графа

```
np.random.seed(0)
labels = {i: G.nodes[i]['webpage'] for i in G.nodes}
nx.draw(G, labels=labels, arrowsize=20)
plt.show()
```

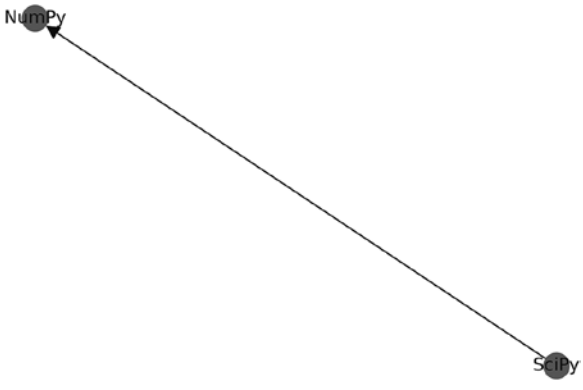


Рис. 18.3. Визуализированный направленный граф из двух узлов. Стрелка направлена от нижнего узла к верхнему. Оба узла обозначены, но метки трудноразличимы

Стрелка стала больше, но подписи видимы лишь частично, так как сливаются с темным цветом узлов. Чтобы сделать подписи более отчетливыми, можно заменить цвет узлов на светлый, например бирюзовый. Для изменения цвета передадим в `G.draw` параметр `node_color="cyan"` (листинг 18.15; рис. 18.4).

Листинг 18.15. Изменение цвета узлов

```
np.random.seed(0)
nx.draw(G, labels=labels, node_color="cyan", arrowsize=20)
plt.show()
```

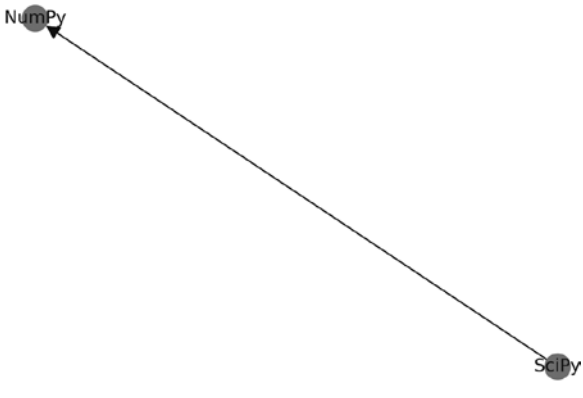


Рис. 18.4. Визуализированный направленный граф из двух узлов. Оба узла обозначены, а их цвет изменен для лучшей видимости подписей

На последнем графе подписи видно значительно лучше. Здесь уже есть направленная связь от SciPy к NumPy, так что пора добавить обратную ссылку от NumPy к SciPy (листинг 18.16; рис. 18.5).

Листинг 18.16. Добавление обратной ссылки между веб-страницами

```
np.random.seed(0)
G.add_edge(0, 1)
nx.draw(G, labels=labels, node_color="cyan", arrowsize=20)
plt.show()
```

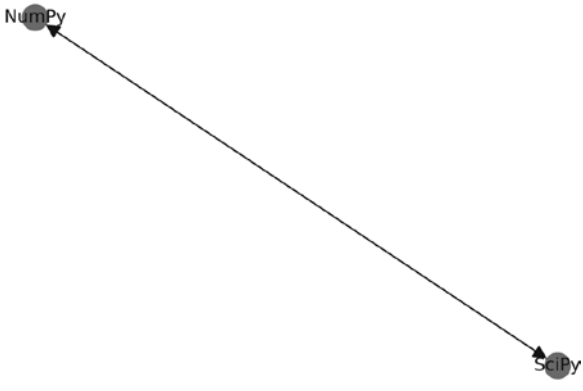


Рис. 18.5. Визуализированный направленный граф из двух узлов. Стрелки имеются с обоих концов ребра между узлами, указывая на его двунаправленность

Теперь можно расширить сеть, добавив еще две веб-страницы — Pandas и Matplotlib. Они будут отвечать узлам с ID 2 и 3 соответственно. Мы вставим два узла по отдельности, выполнив сначала `G.add_node(2)`, а затем `G.add_node(3)`. В качестве альтернативы их можно вставить одновременно с помощью метода `G.add_nodes_from`, получающего список узлов, которые нужно вставить в граф. Таким образом, выполнение `G.add_nodes_from([2, 3])` приведет к добавлению в сеть указанных ID узлов. Однако этим новым узлам пока не будут присвоены никакие атрибуты веб-страниц.

К счастью, метод `G.add_nodes_from` позволяет нам указывать значения атрибутов вместе с ID узлов. Для этого нужно лишь передать в метод `[(2, attributes_2), [(3, attributes_3)]`. По сути, необходимо передать список кортежей, соответствующий ID узлов и атрибутам. Атрибуты сохраняются в словаре, где их имена сопоставляются со значениями. К примеру, словарь Pandas `attributes_2` будет соответствовать `{'webpage': 'Pandas')`. Далее мы вставим эти узлы вместе с их атрибутами и выведем `G.nodes(data=True)`, чтобы убедиться в присутствии новых узлов (листинг 18.17).

Листинг 18.17. Добавление в объект графа нескольких узлов

```
webpages = ['Pandas', 'Matplotlib']
new_nodes = [(i, {'webpage': webpage})
             for i, webpage in enumerate(webpages, 2)]
G.add_nodes_from(new_nodes)

print(f"We've added these nodes to our graph:\n{new_nodes}")
print('\nOur updated list of nodes is:')
print(G.nodes(data=True))

We've added these nodes to our graph:
[(2, {'webpage': 'Pandas'}), (3, {'webpage': 'Matplotlib'})]

Our updated list of nodes is:
[(0, {'webpage': 'NumPy'}), (1, {'webpage': 'SciPy'}), (2, {'webpage':
     'Pandas'}), (3, {'webpage': 'Matplotlib'})]
```

Мы добавили еще два узла. Теперь можно визуализировать обновленный граф (листинг 18.18; рис. 18.6).

Листинг 18.18. Построение обновленного графа из четырех узлов

```
np.random.seed(0)
labels = {i: G.nodes[i]['webpage'] for i in G.nodes}
nx.draw(G, labels=labels, node_color="cyan", arrowsize=20)
plt.show()
```

Рис. 18.6. Визуализированный направленный граф веб-страниц. Страницы Pandas и Matplotlib пока отделены

Пока у нас получилась несвязанная сеть. Далее добавим две новые ссылки: одну — от Matplotlib (узел 3) к NumPy (узел 0), другую — от NumPy (узел 0) к Pandas (узел 2). Для этого нужно вызвать сначала `G.add_edge(3, 0)`, а затем `G.add_edge(0, 2)`.

Одновременно же добавить их можно с помощью метода `G.add_edges_from`, который получает список ребер, каждое из которых представлено кортежем в форме `(i, j)`. Таким образом, выполнение `G.add_edges_from([(0, 2), (3, 0)])` должно привести к добавлению в граф двух новых ребер. Код листинга 18.19 добавляет эти ребра и заново генерирует график (рис. 18.7).

Листинг 18.19. Добавление в объект графа нескольких ребер

```
np.random.seed(1)
G.add_edges_from([(0, 2), (3, 0)])
nx.draw(G, labels=labels, node_color="cyan", arrowsize=20)
plt.show()
```

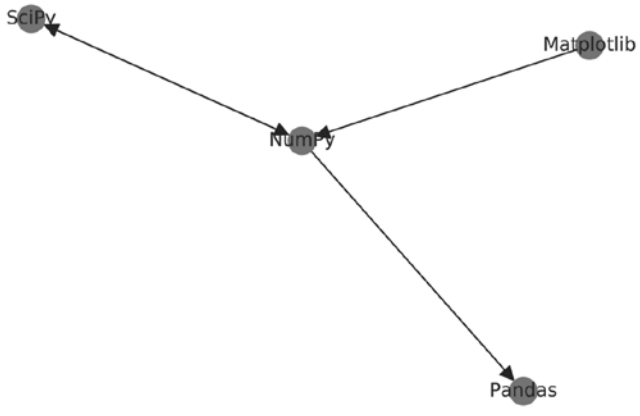


Рис. 18.7. Визуализированный направленный граф веб-страниц. Направленные в центр ссылки указывают на страницу NumPy. Все остальные страницы имеют не более одной входящей ссылки

ПРИМЕЧАНИЕ

В нашей визуализации графа узлы были разбросаны с целью подчеркнуть возможность соединения их ребер. Достичь этого эффекта помогла техника силовой визуализации графов, в основе которой лежит физика. Узлы в ней моделируются как отрицательно заряженные частицы, которые друг от друга отталкиваются, а ребра — как пружины, эти частицы соединяющие. По мере расхождения связанных узлов пружины начинают стягивать их обратно. Моделирование физических уравнений в этой системе и дает нашу визуализацию графа.

Страница NumPy находится в центре обновленного графа. Две веб-страницы, SciPy и Matplotlib, содержат ссылки, ведущие к NumPy. Все остальные веб-страницы имеют не более одной входящей ссылки. На ресурс NumPy создатели контента сделали бóльший акцент, чем на любой другой. Можно сделать вывод, что сайт этой библиотеки самый популярный, поскольку на него ведет больше всего ссылок. По сути, мы разработали простой показатель ранжирования сайтов Интернета.

Он соответствует количеству входящих ребер, указывающих на сайт, и называется *входящей степенью* узла. Этот показатель противоположен *исходящей степени*, которая соответствует количеству ребер, исходящих из сайта. Глядя на наш граф, можно вывести входящую степень каждого сайта автоматически. Ее можно вычислить и непосредственно из матрицы смежности графа. Чтобы показать, как это делается, мы сначала выведем обновленную матрицу смежности (листинг 18.20).

Листинг 18.20. Вывод обновленной матрицы смежности

```
adjacency_matrix = nx.to_numpy_array(G)
print(adjacency_matrix)

[[0. 1. 1. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 0.]
 [1. 0. 0. 0.]]
```

Напомню, что *i*-й столбец матрицы отслеживает входящие ребра узла `i`. Общее их число равно количеству единиц в этом столбце. Следовательно, сумма значений в нем равна входящей степени узла. Например, столбец 0 нашей матрицы равен `[0, 1, 0, 1]`. Сумма этих значений составит входящую степень 2, соответствующую странице NumPy. Если брать в целом, то выполнение `adjacency_matrix.sum(axis=0)` возвращает вектор входящих степеней (листинг 18.21). Наибольший элемент этого вектора соответствует наиболее популярной странице в рассматриваемом интернет-графе.

ПРИМЕЧАНИЕ

Наша простая система ранжирования предполагает, что все входящие ссылки имеют равный вес, но на деле это не так. Входящая ссылка с очень популярного сайта имеет больший вес, поскольку привлекает больше трафика. В следующей главе мы познакомимся с более сложным алгоритмом ранжирования, называемым PageRank, который учитывает популярность сайтов, обеспечивающих поток трафика.

Листинг 18.21. Вычисление входящих степеней с помощью матрицы смежности

```
in_degrees = adjacency_matrix.sum(axis=0)
for i, in_degree in enumerate(in_degrees):
    page = G.nodes[i]['webpage']
    print(f"{page} has an in-degree of {in_degree}")

top_page = G.nodes[in_degrees.argmax()]['webpage']
print(f"\n{top_page} is the most popular page.")

NumPy has an in-degree of 2.0
SciPy has an in-degree of 1.0
Pandas has an in-degree of 1.0
Matplotlib has an in-degree of 0.0

NumPy is the most popular page.
```

В качестве альтернативы входящие степени можно вычислить при помощи NetworkX-метода `in_degree`. Вызов `G.in_degree(i)` вернет входящую степень узла `i`. Тогда можно ожидать, что `G.in_degree(0)` будет равно `2`. Проверим это (листинг 18.22).

Листинг 18.22. Вычисление входящих степеней с помощью NetworkX

```
assert G.in_degree(0) == 2
```

Нужно помнить, что в этом коде `G.nodes[0]` соответствует странице NumPy. Отслеживание сопоставления между ID узлов и именами страниц представляет некоторое неудобство, которое можно обойти, присвоив отдельным узлам строковые идентификаторы. К примеру, имея пустой граф `G2`, мы добавим в него ID наших узлов, выполнив `G2.add_nodes_from(['NumPy', 'SciPy', 'Matplotlib', 'Pandas'])`. Тогда выполнение `G2.in_degree('NumPy')` вернет входящую степень страницы NumPy (листинг 18.23).

ПРИМЕЧАНИЕ

Сохранение ID узлов в виде строк упрощает доступ к определенным узлам графа. Однако за это удобство мы платим несоответствием между ID узлов и индексами в матрице смежности. Как мы вскоре узнаем, эта матрица необходима для ряда задач при работе с сетями, поэтому обычно рекомендуется сохранять ID узлов в виде индексов, а не строк.

Листинг 18.23. Использование строк в качестве ID узлов в графе

```
G2 = nx.DiGraph()
G2.add_nodes_from(['NumPy', 'SciPy', 'Matplotlib', 'Pandas'])
G2.add_edges_from([('SciPy', 'NumPy'), ('SciPy', 'NumPy'),
                   ('NumPy', 'Pandas'), ('Matplotlib', 'NumPy')])
assert G2.in_degree('NumPy') == 2
```

Имея набор атрибутов узлов и набор ребер, можно сгенерировать граф всего в трех строках кода. Этот паттерн пригождается во многих сетевых задачах. Как правило, работая с данными графа, аналитики имеют два файла: один содержит все атрибуты узлов, а другой — информацию о связи между этими узлами. К примеру, в нашем практическом задании дана таблица профилей из FriendHook и таблица существующих отношений между ними. Отношения здесь выступают в качестве ребер, которые можно загрузить вызовом `add_edges_from`. При этом информация профилей отражает атрибуты каждого пользователя в графе дружеских связей. После должной подготовки эти профили можно отобразить обратно в узлы, вызвав `add_nodes_from`. Таким образом, будет легко загрузить граф FriendHook в NetworkX для дальнейшего анализа.

БАЗОВЫЕ МЕТОДЫ ДЛЯ РАБОТЫ С ГРАФАМИ В NETWORKX

- `G = nx.DiGraph()` — инициализирует новый направленный граф.
- `G.add_node(i)` — создает узел с индексом `i`.
- `G.nodes[i]['attribute'] = x` — присваивает узлу `i` атрибут `x`.
- `G.add_node(i, attribute=x)` — создает узел `i` с атрибутом `x`.
- `G.add_nodes_from([i, j])` — создает узлы с индексами `i` и `j`.
- `G.add_nodes_from([(i, {'a': x}), (j, {'a': y})]` — создает узлы с индексами `i` и `j`. Атрибут `a` каждого такого узла устанавливается равным `x` и `y` соответственно.
- `G.add_edge(i, j)` — создает ребро, исходящее из `i` к `j`.
- `G.add_edges_from([(i, j), (k, m)])` — создает новые ребра, исходящие из `i` к `j` и от `k` к `m`.
- `nx.draw(G)` — строит граф `G`.

Пока что мы говорили о направленных графах, в которых маршрут обхода узлов строго определен. Каждое направленное ребро подобно улице с односторонним движением. А что, если вместо этого рассматривать каждое ребро так, будто по нему разрешено двигаться в двух направлениях? Тогда ребра станут *ненаправленными*, и мы получим *ненаправленный граф*. В нем можно перемещаться между связанными узлами в любом направлении. Эта парадигма неприменима к направленной сети, лежащей в основе Интернета, но зато применима к ненаправленной сети дорог, связывающих города во всем мире. В следующей главе мы проанализируем поездку по дорогам, используя ненаправленные графы, а впоследствии задействуем их для оптимизации времени движения между городами.

18.2. ИСПОЛЬЗОВАНИЕ НЕНАПРАВЛЕННЫХ ГРАФОВ ДЛЯ ОПТИМИЗАЦИИ ПОЕЗДКИ МЕЖДУ ГОРОДАМИ

В бизнес-логистике время доставки товара может повлиять на определенные критически важные решения. Представьте следующий сценарий: вы открыли собственный цех по приготовлению комбучи. В планах — поставлять партии вкуснейшего ферментированного чая во все ближайшие города. Если говорить

точнее, то вы будете отправлять напиток только в те из них, которые расположены на расстоянии двух часов езды от цеха. В противном случае расходы на горючее не оправдают выручку от проданного чая. Магазин в соседнем округе заинтересован в регулярных поставках вашего продукта. Каково будет кратчайшее время доставки между вашим цехом и этим магазином?

Обычно ответ на этот вопрос вы бы получили, прокладывая возможные маршруты в приложении на смартфоне, но мы предположим, что существующие технологические решения недоступны (возможно, эта область удалена и ее карты еще не были внесены в базу данных в Сети). Иными словами, вам нужно воссоздать вычисления длительности поездки, выполняемые современными инструментами на смартфонах. Для этого вы обращаетесь к бумажной карте региона. На ней многие дороги между городами извилистые, а некоторые — прямые. Очень удобно, что длительность поездки между соединенными дорогами городами на карте указана явно. Такие связи городов можно смоделировать с помощью ненаправленных графов.

Предположим, что дорога связывает два города, 0 и 1. Время следования между ними составляет 20 мин. Эту информацию нужно внести в ненаправленный граф. Для начала мы сгенерируем этот граф в NetworkX, выполнив `nx.Graph()`. Затем добавим в него ненаправленное ребро, выполнив `G.add_edge(0, 1)`. Наконец, внесем в качестве атрибута этого ребра время следования, выполнив `G[0][1]['travel_time'] = 20` (листинг 18.24).

Листинг 18.24. Создание ненаправленного графа из двух узлов

```
G = nx.Graph()
G.add_edge(0, 1)
G[0][1]['travel_time'] = 20
```

Время следования — это атрибут ребра `(0, 1)`. Имея атрибут `k` ребра `(i, j)`, можно обратиться к нему, выполнив `G[i][j][k]` (листинг 18.25). Значит, выяснить время следования можно, выполнив `G[0][1]['travel_time']`. В нашем ненаправленном графе продолжительность поездки между городами от направления не зависит, значит, `G[1][0]['travel_time']` также равно 20.

Листинг 18.25. Проверка атрибута ребра графа

```
for i, j in [(0, 1), (1, 0)]:
    travel_time = G[i][j]['travel_time']
    print(f"It takes {travel_time} minutes to drive from Town {i} to Town {j}.")

It takes 20 minutes to drive from Town 0 to Town 1.
It takes 20 minutes to drive from Town 1 to Town 0.
```

Города 1 и 0 на нашей карте связаны. Однако не все они связаны напрямую. Представьте себе город 2, соединенный с городом 1, но не с городом 0. Между городом 0

и городом 2 дорог нет, но есть дорога между 1 и 2. Время следования по ней составляет 15 мин. Добавим новую связь в наш граф. Внесение этого ребра и соответствующего времени следования выполняется в одной строке кода с помощью `G.add_edge(1, 2, travel_time=15)`, после чего мы визуализируем граф посредством `nx.draw` (листинг 18.26). Мы также устанавливаем подписи узлов равными их ID, для чего передаем `with_labels=True` в функцию `draw` (рис. 18.8).

Листинг 18.26. Визуализация пути между несколькими городами

```
np.random.seed(0)
G.add_edge(1, 2, travel_time=15)
nx.draw(G, with_labels=True, node_color='khaki')
plt.show()
```

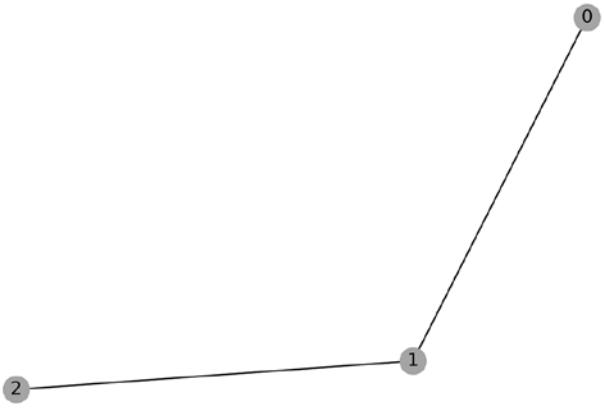


Рис. 18.8. Визуализированный путь из города 0 в город 2 через город 1

Чтобы добраться из города 0 в город 2, необходимо проехать через город 1. В связи с этим общее время в пути оказывается равно сумме `G[0][1]['travel_time']` и `G[1][2]['travel_time']`. Далее мы это время вычислим (листинг 18.27).

Листинг 18.27. Вычисление времени поездки между городами

```
travel_time = sum(G[i][1]['travel_time'] for i in [0, 2])
print(f"It takes {travel_time} minutes to drive from Town 0 to Town 2.")
```

```
It takes 35 minutes to drive from Town 0 to Town 2.
```

Мы нашли минимальное время в пути между двумя городами. Вычисления оказались простыми, поскольку между городом 0 и городом 2 есть всего один маршрут. Но в реальной жизни таких маршрутов может существовать несколько, в связи с чем оптимизировать время движения между несколькими городами

уже не так легко. Для наглядности построим граф, содержащий более десятка городов, разбросанных по нескольким округам. В этой модели время следования между городами, находящимися в разных округах, будет возрастать. Мы примем следующие допущения:

- города расположены в шести разных округах;
- в каждом округе находятся от трех до десяти городов;
- 90 % городов одного округа связаны дорогами напрямую. Среднее время следования по дороге между округами составляет 20 мин;
- между 5 % городов разных округов есть прямая дорога. Среднее время следования по дороге внутри округа составляет 45 мин.

Теперь смоделируем этот сценарий, после чего построим алгоритм для вычисления кратчайшего времени следования между любыми двумя городами нашей сложной сети.

ТИПИЧНЫЕ МЕТОДЫ NETWORKX И ПРИСВАИВАНИЕ АТРИБУТОВ

- `G = nx.Graph()` — инициализирует новый ненаправленный граф.
- `G.nodes[i]['attribute'] = x` — присваивает узлу `i` атрибут `x`.
- `G[i][j]['attribute'] = x` — присваивает атрибут `x` ребру `(i, j)`.

18.2.1. Моделирование сложной сети из городов и округов

Начнем с моделирования одного округа с пятью городами (листинг 18.28). Сперва добавим в пустой граф пять узлов. Каждому узлу будет присвоен атрибут `county_id`, равный `0`, говорящий о том, что все узлы принадлежат к одному округу.

Листинг 18.28. Моделирование пяти городов в одном округе

```
G = nx.Graph()
G.add_nodes_from((i, {'county_id': 0}) for i in range(5))
```

Далее присвоим этим пяти городам случайные дороги (листинг 18.29; рис. 18.9), перебрав каждую возможную пару узлов и подбросив монету со смещенным центром тяжести. В 90 % случаев она будет падать орлом вверх, и при таком исходе мы будем добавлять между парой узлов ребро. Параметр `travel_time` каждого ребра выбирается случайно путем выборки из нормального распределения, среднее значение которого равно 20.

ПРИМЕЧАНИЕ

Напомню, что нормальное распределение — это колоколообразная кривая, обычно используемая для анализа случайных процессов в теории вероятности и статистике. Подробнее о ней можно почитать в главе 6.

Листинг 18.29. Моделирование случайных дорог внутри округа

```
import numpy as np
np.random.seed(0)

def add_random_edge(G, node1, node2, prob_road=0.9,
                    mean_drive_time=20):
    if np.random.binomial(1, prob_road):
        drive_time = np.random.normal(mean_drive_time)
        G.add_edge(node1, node2, travel_time=round(drive_time, 2))

nodes = list(G.nodes())
for node1 in nodes[:-1]:
    for node2 in nodes[node1 + 1:]:
        add_random_edge(G, node1, node2)

nx.draw(G, with_labels=True, node_color='khaki')
plt.show()
```

Эта функция пробует сгенерировать в графе G случайное ребро между node1 и node2. Вероятность добавления ребра равна prob_road. Добавленному ребру присваивается случайный атрибут времени следования, которое выбирается из нормального распределения со средним значением, равным mean_travel_time

Подбрасывает монету, определяя, нужно ли вставлять ребро

Выбирает время следования из нормального распределения

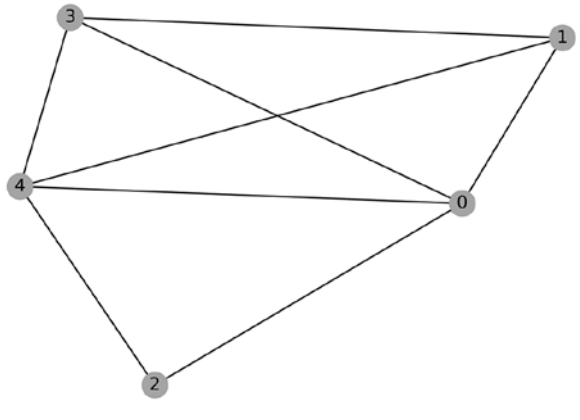


Рис. 18.9. Случайно сгенерированная сеть дорог в округе с пятью городами

Мы соединили большинство городов в округе 0. Аналогичным образом можно случайно сгенерировать дороги и города для округа 1. Здесь мы сгенерируем округ 1 и сохраним его в отдельном графе (листинг 18.30; рис. 18.10). Количество городов в нем будет случайно выбрано между 3 и 10.

Листинг 18.30. Моделирование второго случайного округа

```
np.random.seed(0)
def random_county(county_id):
    numTowns = np.random.randint(3, 10)
    G = nx.Graph()
    nodes = [(node_id, {'county_id': county_id})
             for node_id in range(numTowns)]

    G.add_nodes_from(nodes)
    for node1, _ in nodes[:-1]:
        for node2, _ in nodes[node1 + 1:]:
            add_random_edge(G, node1, node2)

    return G

G2 = random_county(1)
nx.draw(G2, with_labels=True, node_color='khaki')
plt.show()
```

Генерирует случайный граф округа

Случайно выбирает количество городов в округе в диапазоне от 3 до 10

Случайно добавляет дороги внутри округа

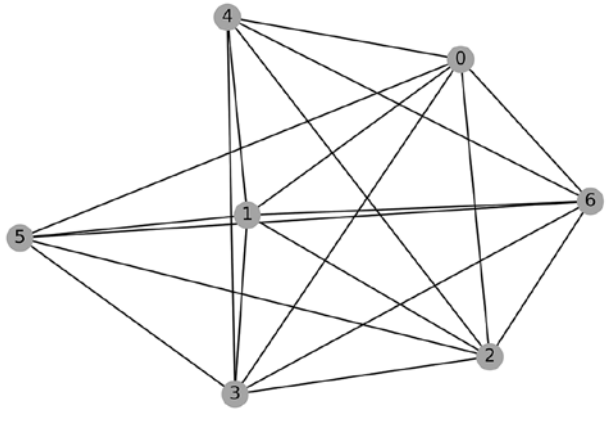


Рис. 18.10. Случайно сгенерированная сеть дорог во втором округе. Количество городов в нем также было выбрано случайно

Сейчас округа 1 и 2 сохранены в двух отдельных графах, `G` и `G2`. Теперь нужно их как-то совместить. Слияние графов усложняется тем фактом, что в них используются одинаковые ID узлов. К счастью, задачу может упростить функция `nx.disjoint_union`, которая получает два графа, `G` и `G2`, а затем устанавливает ID каждого узла на уникальное значение между 0 и общим количеством узлов. В завершение она реализует слияние двух графов. Код листинга 18.31 выполняет `nx.disjoint_union(G, G2)`, после чего строит график результатов (рис. 18.11).

Листинг 18.31. Слияние двух отдельных графов

```
np.random.seed(0)
G = nx.disjoint_union(G, G2)
nx.draw(G, with_labels=True, node_color='khaki')
plt.show()
```

Рис. 18.11. Сети двух округов объединены

Теперь два округа отражены в одном графе и каждому городу в нем присвоен уникальный ID. Далее можно генерировать между этими округами случайные дороги (листинг 18.32; рис. 18.12). Мы перебираем все пары узлов между округами, в которых `G[n1]['county_id']!=G[n2]['county_id']`. Для каждой пары узлов применяем `add_random_edge`. Установлена вероятность построения ребра 0,05, а среднее время следования — 90 мин.

Листинг 18.32. Добавление случайных дорог между округами

```
np.random.seed(0)
def add_intercounty_edges(G):
    nodes = list(G.nodes(data=True))
    for node1, attributes1 in nodes[:-1]:
        county1 = attributes1['county_id']
        for node2, attributes2 in nodes[node1:]:
            if county1 != attributes2['county_id']:
                add_random_edge(G, node1, node2,
                                prob_road=0.05, mean_drive_time=45)
    return G

G = add_intercounty_edges(G)
np.random.seed(0)
nx.draw(G, with_labels=True, node_color='khaki')
```

Добавляет случайное ребро между теми узлами в графе G, чьи ID округов не совпадают

Перебирает каждый узел и связанные с ним атрибуты

Перебирает пары узлов, которые еще не сравнивались

Пытается добавить случайное ребро между округами

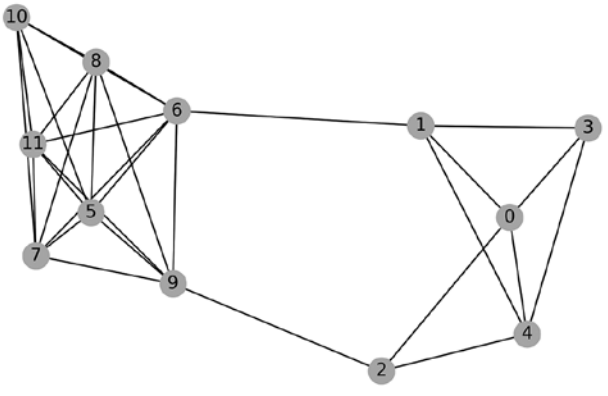


Рис. 18.12. Сети двух округов соединены случайными дорогами

Мы успешно симулировали два взаимосвязанных округа. Теперь симулируем шесть таких округов (листинг 18.33; рис. 18.13).

Листинг 18.33. Симулирование шести взаимосвязанных округов

```
np.random.seed(1)
G = random_county(0)
for county_id in range(1, 6):
    G2 = random_county(county_id)
    G = nx.disjoint_union(G, G2)

G = add_intercounty_edges(G)
np.random.seed(1)
nx.draw(G, with_labels=True, node_color='khaki')
plt.show()
```

Мы визуализировали граф из шести округов, но отдельные округа в нем выделить сложно. К счастью, можно улучшить график, окрасив каждый узел на основе ID его округа. Для этого потребуется изменить входные значения параметра `node_color` — вместо передачи одной строки цвета мы передадим их список. В этом списке *i*-й цвет будет соответствовать присвоенному цвету узла по индексу `i`. Код листинга 18.34 делает так, чтобы узлам в разных округах были присвоены разные цвета, но при этом все узлы одного округа имели один цвет (рис. 18.14)

Листинг 18.34. Окрашивание узлов по округам

```
np.random.seed(1)
county_colors = ['salmon', 'khaki', 'pink', 'beige', 'cyan', 'lavender']
county_ids = [G.nodes[n]['county_id']
              for n in G.nodes]
```

```
node_colors = [county_colors[id_]
               for id_ in county_ids]
nx.draw(G, with_labels=True, node_color=node_colors)
plt.show()
```

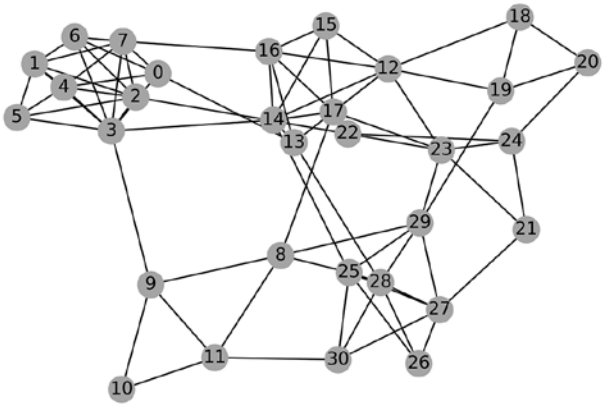


Рис. 18.13. Шесть сетей округов, связанных случайными дорогами

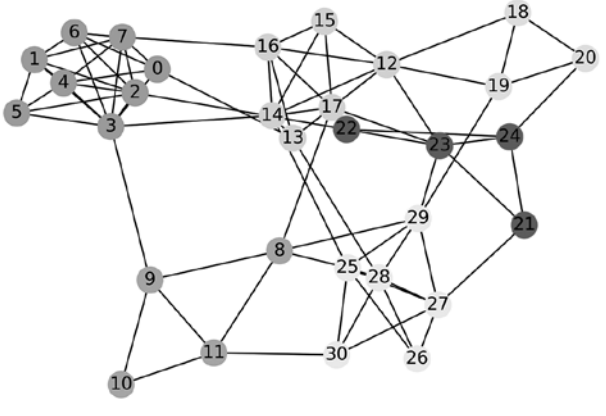


Рис. 18.14. Сети шести округов, связанные случайными дорогами. Отдельные города окрашены в соответствии с ID их округа

Теперь отдельные округа отчетливо видны. Большинство из них формируют в сети плотные группы. Позднее мы будем извлекать эти группы автоматически с помощью кластеризации сети. Сейчас же сосредоточимся на вычислении кратчайшего времени следования между узлами.

ТИПИЧНЫЕ ФУНКЦИИ ВИЗУАЛИЗАЦИИ ГРАФА В NETWORKX

- `nx.draw(G)` — строит граф `G`.
- `nx.draw(G, labels=True)` — строит граф `G` с подписями узлов. Подписи соответствуют ID узлов.
- `nx.draw(G, labels=ids_to_labels)` — строит граф `G` с подписями узлов. Последние обозначаются путем сопоставления ID узлов и их подписей. Это сопоставление определяется словарем `ids_to_labels`.
- `nx.draw(G, node_color=c)` — строит граф `G`, все узлы которого окрашиваются с помощью `color c`.
- `nx.draw(G, node_color=ids_to_colors)` — строит граф `G`, все узлы которого окрашиваются на основе сопоставления ID узлов и цветов. Это сопоставление определяется словарем `ids_to_colors`.
- `nx.draw(G, arrowsize=20)` — строит направленный граф `G`, увеличивая размер стрелок на его ребрах.
- `nx.draw(G, node_size=20)` — строит граф `G`, уменьшая размер узлов с предустановленного значения 300 до 20.

18.2.2. Вычисление кратчайшего времени следования между узлами

Предположим, что наш чайный цех расположен в городе 0, а потенциальный клиент — в городе 30. Нужно определить кратчайшее время следования из города 0 в город 30. В ходе этого потребуется вычислить минимальное время следования от города 0 до всех остальных городов. Как это сделать? Изначально нам известно лишь очевидное время следования от города 0 до него самого — 0 мин. Давайте зарегистрируем это время в словаре `fastest_times` (листинг 18.35). Позднее мы заполним этот словарь длительностями поездок до каждого города.

Листинг 18.35. Отслеживание кратчайшего известного времени следования

```
fastest_times = {0: 0}
```

Далее мы можем ответить на простой вопрос: «Каковы известные расстояния между городом 0 и соседними с ним городами?» Соседями в этом случае являются города, соединенные дорогами с городом 0. В NetworkX к соседям города 0 можно обратиться с помощью выполнения `G.neighbors(0)`. Этот метод вернет результат итерации, отображающий ID узлов, связанных с узлом 0. В качестве альтернативы соседей можно вычислить выполнением `G[0]`. Код листинга 18.36 выводит ID всех соседних городов.

Листинг 18.36. Вывод соседей города 0

```
neighbors = list(G.neighbors(0))
assert list(neighbors) == list(G[0])
print(f"The following towns connect directly with Town 0:\n{neighbors}")

The following towns connect directly with Town 0:
[3, 4, 6, 7, 13]
```

Запишем продолжительность поездки между городом 0 и каждым из этих пяти его соседей, а затем на основе этих значений обновим `fastest_times` (листинг 18.37). Кроме того, выведем эти значения в упорядоченном виде для последующего анализа.

Листинг 18.37. Определение времени следования до соседних городов

```
time_to_neighbor = {n: G[0][n]['travel_time'] for n in neighbors} fastest_times.
update(time_to_neighbor)
for neighbor, travel_time in sorted(time_to_neighbor.items(),
                                    key=lambda x: x[1]):
    print(f"It takes {travel_time} minutes to drive from Town 0 to Town "
          f"{neighbor}.")

It takes 18.04 minutes to drive from Town 0 to Town 7.
It takes 18.4 minutes to drive from Town 0 to Town 3.
It takes 18.52 minutes to drive from Town 0 to Town 4.
It takes 20.26 minutes to drive from Town 0 to Town 6.
It takes 44.75 minutes to drive from Town 0 to Town 13.
```

Чтобы доехать из города 0 до города 13, потребуется примерно 45 мин. Является ли это время следования кратчайшим из возможных между этими городами? Не обязательно. Объезд через другой город может укоротить маршрут, как это показано на рис. 18.15.

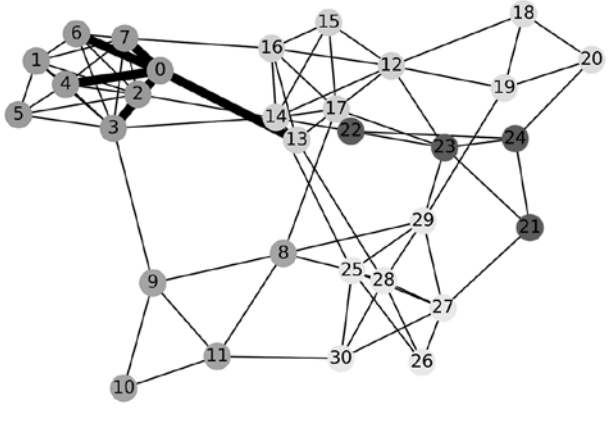


Рис. 18.15. Дороги, связывающие город 0 с его соседями, выделены толстыми темными ребрами. Время следования по этим пяти дорогам известно. Есть вероятность, что существуют более короткие маршруты, но они потребуют проезда через другие города

Возьмем, к примеру, объезд через город 7. Это самый близкий населенный пункт, добраться до которого можно всего за 18 мин. А что, если между городами 7 и 13 есть дорога? Если она существует и время следования по ней составляет менее 27 мин, тогда у нас получится более короткий маршрут до города 13. Аналогичная логика применима к городам 3, 4 и 6. Есть вероятность, что изучение маршрутов между городом 7 и его соседями позволит нам сэкономить время. Для этого проделаем следующее.

1. Определим соседей города 7.
2. Определим время следования между городом 7 и каждым соседним с ним городом *N*.
3. Прибавим к полученному времени 18,04 мин, получив длительность поездки между городом 0 и городом *N* при объезде через город 7.
4. Если *N* присутствует в `fastest_times`, проверим, короче ли этот маршрут с объездом, чем `fastest_times[N]`. В случае обнаружения более оптимального варианта обновим `fastest_times` и выведем кратчайшее время следования.
5. Если *N* в `fastest_times` отсутствует, обновим словарь временем следования, вычисленным на шаге 3, получив длительность маршрута между городом 0 и городом *N*, когда между ними нет прямой дороги.

Все описанные шаги проделываются в листинге 18.38.

Листинг 18.38. Поиск кратчайших объездов через город 7

```
def examine_detour(town_id):
    detour_found = False

    travel_time = fastest_times[town_id]
    for n in G[town_id]:
        detour_time = travel_time + G[town_id][n]['travel_time']
        if n in fastest_times:
            if detour_time < fastest_times[n]:
                detour_found = True
                print(f"A detour through Town {town_id} reduces "
                    f"travel-time to Town {n} from "
                    f"{fastest_times[n]:.2f} to "
                    f"{detour_time:.2f} minutes.")
                fastest_times[n] = detour_time

        else:
            fastest_times[n] = detour_time

    return detour_found

if not examine_detour(7):
    print("No detours were found.")
```

Проверяет, изменяет ли объезд через town_id кратчайшее известное время следования из города 0 в другие города

Время следования между городом 0 и town_id

Время объезда от города 0 до соседа town_id

Проверяет, улучшил ли объезд кратчайшее известное время следования от города 0 до N

Записывает кратчайшее известное время следования от города 0 до N

```
addedTowns = len(fastest_times) - 6
print(f"We've computed travel-times to {addedTowns} additional towns.")
```

Проверяет, сколько новых городов было добавлено в словарь fastest_times к шести изначально присутствующим

```
No detours were found.
We've computed travel-times to 3 additional towns.
```

Мы выяснили время следования до трех дополнительных городов, но пока не нашли более коротких маршрутов к соседям от города 0. Хотя это все еще возможно. Давайте выберем в качестве точки объезда другого перспективного кандидата — близкий к городу 0 пункт, соседей которого мы еще не исследовали. Для этого потребуется проделать следующее.

1. Объединить соседей городов 0 и 7 в пул кандидатов на объезд. Заметьте, что оба эти города в него также войдут, в связи с чем потребуется следующий шаг.
2. Удалить города 0 и 7 из пула кандидатов, оставить набор непроанализированных городов.
3. Выбрать неизученный город с кратчайшим известным временем следования до города 0.

Далее мы выполним эти шаги для выбора следующего кандидата на объезд (листинг 18.39), используя логику, представленную на рис. 18.16.

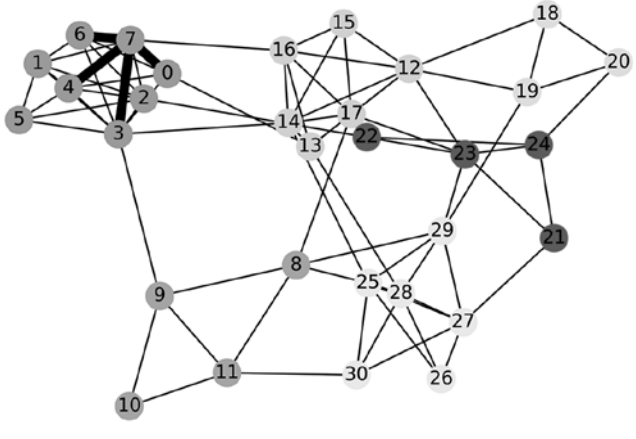


Рис. 18.16. Прямые объезды через город 7 для посещения соседей города 0 выделены темными жирными ребрами. Они не сократили общее время следования. Возможно, улучшить результаты позволит объезд через город 3

Листинг 18.39. Выбор альтернативного кандидата на объезд

Пул кандидатов на объезд совмещает соседей городов 0 и 7. Заметьте, что сами эти города соседствуют друг с другом, поэтому их нужно из списка удалить

```
candidate_pool = set(G[0]) | set(G[7])
examinedTowns = {0, 7}
unexaminedTowns = candidate_pool - examinedTowns
```

Удаляет из набора кандидатов все ранее изученные города

```
detour_candidate = min(unexaminedTowns,
                       key=lambda x: fastest_times[x])
travel_time = fastest_times[detour_candidate]
print(f"Our next detour candidate is Town {detour_candidate}, "
      f"which is located {travel_time} minutes from Town 0.")
```

Выбирает кандидата на объезд с кратчайшим известным временем следования до города 0

```
Our next detour candidate is Town 3, which is located 18.4 minutes from Town 0.
```

Следующим кандидатом на объезд будет город 3. Проверка его соседей может раскрыть новые, еще не рассмотренные города. Все подобные города мы добавим в `unexaminedTowns`, который позволит отслеживать оставшихся кандидатов на объезд для дальнейшего анализа (листинг 18.40). Заметьте, что с целью отслеживания кандидатов город 3 после изучения необходимо перенести из `unexaminedTowns` в `examinedTowns`.

Листинг 18.40. Поиск более коротких объездов через город 3

```
if not examine_detour(detour_candidate):
    print("No detours were found.")

def new_neighbors(town_id):
    return set(G[town_id]) - examinedTowns

def shift_to_examined(town_id):
    unexaminedTowns.remove(town_id)
    examinedTowns.add(town_id)
```

Анализирует город 3 на предмет возможных объездов

Эта вспомогательная функция получает соседей города 3, которые еще не были включены в список кандидатов на объезд

Эта вспомогательная функция после анализа перемещает город 3 в examinedTowns

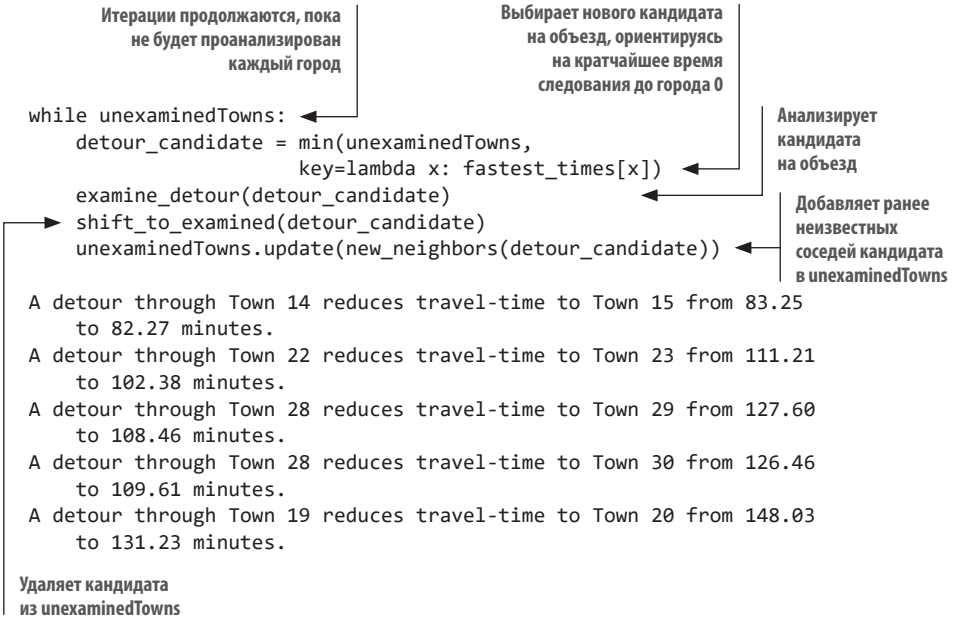
```
unexaminedTowns.update(new_neighbors(detour_candidate))
shift_to_examined(detour_candidate)
num_candidates = len(unexaminedTowns)
print(f"{num_candidates} detour candidates remain.")

No detours were found.
9 detour candidates remain.
```

И снова мы не нашли никаких маршрутов объезда. Однако в наборе `unexaminedTowns` остаются еще девять кандидатов, которых мы проверим в дальнейшем. Код листинга 18.41 итеративно проделывает следующее.

1. Выбирает неизученный город с кратчайшим известным временем следования до города 0.
2. Проверяет его на предмет возможных объездов с помощью `examine_detour`.
3. Переносит ID этого города из `unexaminedTowns` в `examinedTowns`.
4. Если в списке остаются неизученные города, повторяет шаг 1. В противном случае завершается.

Мы проанализировали время следования до каждого города и обнаружили пять возможных маршрутов объезда, два из которых проходят через город 28. Они сокращают время следования до городов 29 и 30 с 2,1 до 1,8 ч, значит, оба города попадают в приемлемый диапазон длительности поездки от нашего чайного цеха.

Листинг 18.41. Оценка каждого города на предмет более короткого объезда




```
while unexaminedTowns:
    detour_candidate = min(unexaminedTowns,
                           key=lambda x: fastest_times[x])
    examine_detour(detour_candidate)
    shift_to_examined(detour_candidate)
    unexaminedTowns.update(new_neighbors(detour_candidate))

A detour through Town 14 reduces travel-time to Town 15 from 83.25
    to 82.27 minutes.
A detour through Town 22 reduces travel-time to Town 23 from 111.21
    to 102.38 minutes.
A detour through Town 28 reduces travel-time to Town 29 from 127.60
    to 108.46 minutes.
A detour through Town 28 reduces travel-time to Town 30 from 126.46
    to 109.61 minutes.
A detour through Town 19 reduces travel-time to Town 20 from 148.03
    to 131.23 minutes.
```

Сколько еще городов расположены в пределах двух часов езды от города 0? Сейчас выясним (листинг 18.42).

Листинг 18.42. Подсчет всех городов в диапазоне двух часов езды

```
closeTowns = {town for town, drive_time in fastest_times.items()
              if drive_time <= 2 * 60}

num_closeTowns = len(closeTowns)
totalTowns = len(G.nodes)
print(f"{num_closeTowns} of our {totalTowns} towns are within two "
      "hours of our brewery.")
```

```
29 of our 31 towns are within two hours of our brewery.
```

Все города, за исключением двух, расположены в пределах двух часов езды от цеха. Это мы выяснили, решив *задачу нахождения кратчайшего пути*. Она применима к графам, чьи ребра имеют численные атрибуты, называемые *весами ребер*. Общая же последовательность требуемых переходов между узлами называется *путем*. Каждый путь пролегает через последовательность ребер. Сумма весов этих ребер называется *длиной пути*. Задача требует от нас вычисления наименьшей длины пути между узлом N и каждым узлом графа. Если все веса ребер положительны, то длину путей можно вычислить так.

1. Создать словарь кратчайших длин путей, который изначально будет выглядеть как `{N: 0}`.

2. Создать множество узлов для анализа. Изначально оно будет пусто.
3. Создать множество узлов, которые мы хотим проанализировать. Изначально в нем будет находиться только *N*.
4. Удалить из множества неисследованных узлов неисследованный узел *U*. Мы выбираем *U* с минимальной длиной пути до *N*.
5. Найти всех соседей *U*.
6. Вычислить длину пути между каждым этим соседом и *N*. Обновить словарь кратчайших длин путей.
7. Добавить во множество неисследованных узлов каждого еще не рассмотренного соседа.
8. Добавить *U* во множество исследованных узлов.
9. Если остались неисследованные узлы, повторить шаг 4. В противном случае закончить.

Этот алгоритм поиска кратчайшего пути включен в NetworkX. Имея граф `G` с атрибутом весов ребер `weight`, можно вычислить все кратчайшие длины путей от узла *N*, выполнив `nx.shortest_path_length(G, weight='weight', source=N)`. Здесь мы задействуем функцию `shortest_path_length` для вычисления `fastest_times` в одной строке кода (листинг 18.43).

Листинг 18.43. Вычисление длин кратчайших путей с помощью NetworkX

```
shortest_lengths = nx.shortest_path_length(G, weight='travel_time',
                                           source=0)
for town, path_length in shortest_lengths.items():
    assert fastest_times[town] == path_length
```

Наш алгоритм вычисления кратчайшего пути на деле этот кратчайший путь не возвращает. Тем не менее в реальных случаях нам нужно знать путь, который минимизирует расстояние между узлами. К примеру, просто знать кратчайшее время следования между городом 0 и городом 30 недостаточно. Нам также нужно знать направления движения, которые позволят преодолеть этот путь менее чем за два часа. К счастью, рассмотренный алгоритм можно легко скорректировать для отслеживания кратчайшего пути. Нужно лишь добавить структуру словаря, отслеживающую переход между узлами. Фактическую последовательность пройденных узлов можно представить списком. Для краткости изложения мы не станем определять функцию отслеживания кратчайшего пути с нуля. Однако рекомендуем вам написать эту функцию и сравнить ее вывод с выводом функции NetworkX `shortest_path`. Вызов `nx.shortest_path_length(G, weight='weight', source=N)` вычислит все кратчайшие пути от узла *N* до каждого узла в `G` (листинг 18.44). Таким образом, выполнение `nx.shortest_path(G, weight='travel_time', source=0)[30]` должно привести к возвращению кратчайшего маршрута между городами 0 и 30. Далее мы выведем этот маршрут (рис. 18.17).

Листинг 18.44. Вычисление кратчайших путей с помощью NetworkX

```
shortest_path = nx.shortest_path(G, weight='travel_time', source=0)[30]
print(shortest_path)

[0, 13, 28, 30]
```

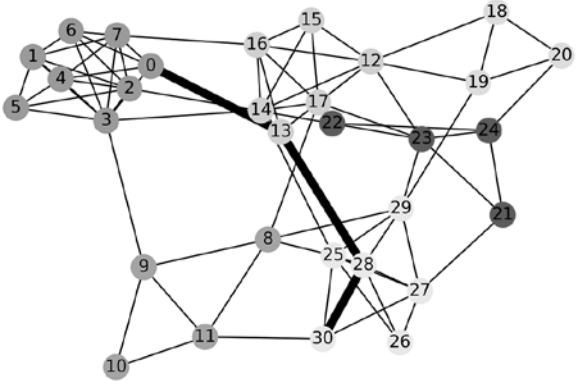


Рис. 18.17. Кратчайший путь между городами 0 и 30 выделен темными жирными ребрами. Он проходит от города 0 через город 13, затем город 28 и достигает города 30. Стоит отметить, что в графе есть и альтернативные пути: например, можно проехать от города 13 до города 25, а затем до города 30. Однако выделенный маршрут гарантированно имеет наименьшую возможную длину

Время поездки минимизируется, если ехать из города 0 до города 13, затем до города 28 и, наконец, до города 30. В таком случае ожидается, что время следования будет равно `fastest_times[30]`. Проверим это (листинг 18.45).

Листинг 18.45. Проверка длины кратчайшего пути

```
travel_time = 0
for i, town_a in enumerate(shortest_path[:-1]):
    town_b = shortest_path[i + 1]
    travel_time += G[town_a][town_b]['travel_time']

print("The fastest travel time between Town 0 and Town 30 is "
      f"{travel_time} minutes.")
assert travel_time == fastest_times[30]

The fastest travel time between Town 0 and Town 30 is 109.61 minutes.
```

Базовая теория сетей позволяет оптимизировать маршруты между географическими точками. В следующей главе мы на основе этой теории разработаем более продвинутые приемы, а именно симулируем поток трафика через сеть городов. Это позволит нам обнаружить самые центральные города в графе. Позднее мы используем ее для кластеризации городов по отдельным округам и покажем, как можно применить эту технику для обнаружения групп друзей в социальном графе.

ТИПИЧНЫЕ ТЕХНИКИ NETWORKX ДЛЯ РАБОТЫ С ПУТЯМИ В СЕТИ

- `G.neighbors(i)` — возвращает всех соседей узла `i`.
- `G[i]` — возвращает всех соседей узла `i`.
- `G[i][j]['weight']` — возвращает протяженность одного пути между соседними узлами `i` и `j`.
- `nx.shortest_path_length(G, weight='weight', source=N)` — возвращает словарь кратчайших путей от узла *N* до всех доступных узлов в графе. Для измерения пути используется атрибут `weight`.
- `nx.shortest_path(G, weight='weight', source=N)` — возвращает словарь кратчайших путей от узла *N* до всех доступных узлов графа.

РЕЗЮМЕ

- *Теория сетей* занимается исследованием связей между объектами. Коллекция объектов вместе с их разветвленными связями называется *сетью* либо *графом*. Сами объекты называются *узлами*, а их связи — *ребрами*.
- Если для ребра определено конкретное направление, оно называется *направленным ребром*. Графы с направленными ребрами называются *направленными графами*. Если граф состоит из ненаправленных ребер, он называется *ненаправленным графом*.
- Граф можно представить как двоичную матрицу `M`, в которой `M[i][j] = 1`, если между узлами `i` и `j` существует ребро. Такое матричное представление называется *матрицей смежности*.
- В направленном графе можно подсчитать входящие и исходящие ребра для каждого узла. Количество входящих называется *входящей степенью*, а количество исходящих — *исходящей степенью*. В определенных графах входящая степень служит мерой популярности узла. Вычислить эту степень можно суммированием строк матрицы смежности.
- Теорию графов можно использовать для оптимизации маршрута между узлами. Последовательность переходов между узлами называется *путем*. С этим путем можно связать длину, если каждому ребру будет присвоен численный атрибут, называемый *весом ребра*. Сумма весов ребер вдоль всей последовательности переходов между узлами пути называется *длиной пути*. Задача нахождения *кратчайшей длины пути* состоит в минимизации длин путей от узла *N* до всех других узлов графа. Если веса ребер положительны, то длины путей можно минимизировать алгоритмически.

19 Динамическое применение теории графов для ранжирования узлов и анализа социальных сетей

В этой главе

- ✓ Поиск самых центральных точек в сети.
- ✓ Кластеризация связей в сети.
- ✓ Освоение анализа социальных графов.

В предыдущей главе мы познакомились с несколькими типами графов. Мы изучили веб-страницы, связанные прямыми ссылками, а также сеть дорог, охватывающую несколько округов. В ходе анализа в основном рассматривали эту сеть как состоящую из застывших, статических объектов, подсчитывая соседние узлы так, будто они являются неподвижными облаками на фотографии. В реальности же облака находятся в постоянном движении, это касается и множества сетей. Большинство заслуживающих изучения сетей постоянно кипят активностью. Машины едут по сетям дорог, вызывая заторы вблизи часто посещаемых городов. По аналогии формируется и поток трафика в Интернете, где миллионы пользователей постоянно переходят по всевозможным ссылкам. В социальных сетях велика активность в виде различных обсуждений, слухов или культурных мемов, распространяемых среди узкого круга друзей. Понимание того, по каким законам существует этот динамический поток, поможет автоматически обнаруживать группы друзей. Оно также поможет определять наиболее загруженные трафиком страницы Интернета. Подобное моделирование динамической сетевой активности необходимо для функционирования множества крупных технологических компаний. В реальности один из методов моделирования, представленный в данной главе, привел к основанию компании, которая в итоге выросла до оборотов в триллионы долларов.

Динамический поток (людей, машин и т. д.) в графе по своей природе является случайным процессом, то есть его можно анализировать с помощью случайных симуляций, аналогичных рассмотренным в главе 3. В начале же текущей главы мы с помощью таких симуляций изучим трафик машин. Затем, используя матричное умножение, попробуем вычислить вероятность организации более эффективного трафика, после чего посредством матричного анализа обнаружим кластеры сообществ с его высокими показателями. В завершение применим созданную ранее технику кластеризации для поиска групп друзей в социальных сетях.

Что ж, приступим. Начнем с простой задачи по выявлению городов с интенсивным движением транспорта, используя симуляцию трафика.

19.1. НАХОЖДЕНИЕ ЦЕНТРАЛЬНЫХ УЗЛОВ НА ОСНОВЕ ОЖИДАЕМОГО ТРАФИКА В СЕТИ

В предыдущей главе мы симулировали сеть дорог, соединяющую 31 город, эти города расположены в шести округах (рис. 19.1). Эту сеть мы сохранили в графе `G`. Нашей целью было оптимизировать время бизнес-доставки во все эти города. Давайте разовьем сценарий. Предположим, наш бизнес растет впечатляющими темпами. Мы хотим расширить клиентскую базу, разместив рекламный баннер в одном из городов, представленных `G.nodes`. Чтобы максимально увеличить число просмотров этого баннера, мы выберем город с самым активным трафиком. Чисто интуитивно это определяется количеством ежедневно проезжающих через город машин. Можем ли мы ранжировать 31 город в `G.nodes` на основе ожидаемого ежедневного трафика? Да! С помощью простого моделирования можно спрогнозировать поток машин, движущихся по сети дорог между городами. Позже мы расширим приемы вычисления трафика для автоматического обнаружения местных округов.

Нам нужен способ ранжирования городов на основе ожидаемого трафика. Можно примитивно посчитать дороги, входящие в каждый город. Город с пятью дорогами может получать трафик с пяти направлений, при этом поток в город с одной входящей дорогой окажется значительно меньше. Подсчет дорог выполняется аналогично ранжированию сайтов по их входящим степеням, чем мы занимались в главе 18. Напомню, что входящая степень узла — это число направленных к нему ребер. Однако, в отличие от графа сайтов, сеть дорог ненаправленная, то есть в ней нет разницы между входящими и исходящими ребрами. Получается, что между входящей и исходящей степенями узла разницы тоже нет. Оба этих значения равны, значит, количество ребер узла в ненаправленном графе называется

просто его *степенью*. Степень любого узла `i` можно вычислить, сложив содержимое *i*-го столбца матрицы смежности графа или выполнив `len(G.nodes[i])`. В качестве альтернативы можно задействовать метод NetworkX `degree` вызовом `G.degree(i)`. В коде листинга 19.1 мы используем все эти способы для подсчета дорог, проходящих через город 0.

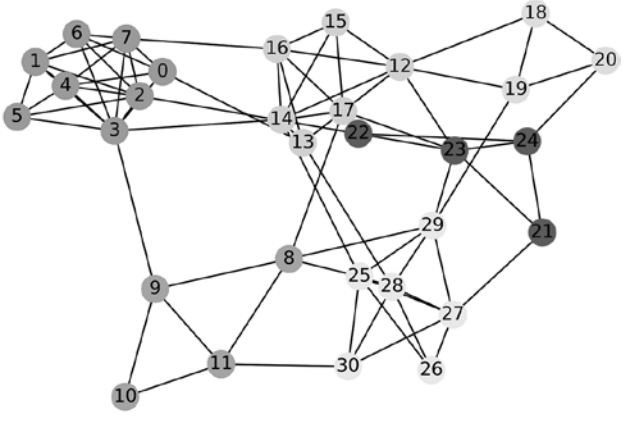


Рис. 19.1. Симулированная сеть узлов из главы 18, сохраненная в графе G. Дороги соединяют 31 город, города разбросаны по шести округам. Каждый город закрашен на основе ID его округа

Листинг 19.1. Вычисление степени одного узла

```
adjacency_matrix = nx.to_numpy_array(G)
degree_town_0 = adjacency_matrix[:,0].sum()
assert degree_town_0 == len(G[0])
assert degree_town_0 == G.degree(0)
print(f"Town 0 is connected by {degree_town_0:.0f} roads.")

Town 0 is connected by 5 roads.
```

Используя степени узлов, можно ранжировать их по важности. В теории графов любая мера значимости узла обычно называется *центральностью узла*, а ранжированная значимость, основанная на степени узла, — *степенью центральности*. Выберем в `G` узел с наивысшей степенью центральности (листинг 19.2), он станет начальным кандидатом на размещение рекламного баннера (рис. 19.2).

Листинг 19.2. Выбор центрального узла на основе степени центральности

```
np.random.seed(1)
central_town = adjacency_matrix.sum(axis=0).argmax()
degree = G.degree(central_town)
```

```
print(f"Town {central_town} is our most central town. It has {degree} "
      "connecting roads.")
node_colors[central_town] = 'k'
nx.draw(G, with_labels=True, node_color=node_colors)
plt.show()
```

```
Town 3 is our most central town. It has 9 connecting roads.
```

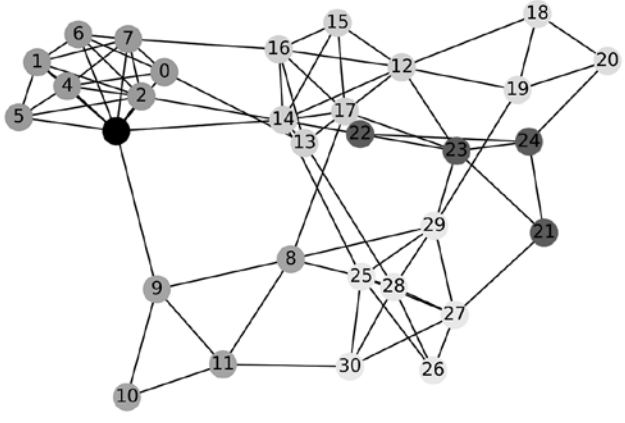


Рис. 19.2. Сеть дорог между городами. Город 3 имеет наивысшую степень центральности и закрашен черным

Самым центральным у нас получился город 3. Дороги соединяют его с девятью городами, относящимися к трем округам. А как город 3 сопоставляется со вторым по степени центральности городом? Это можно быстро проверить с помощью вывода второй по старшинству степени в `G` (листинг 19.3).

Листинг 19.3. Выбор узла со второй по величине степенью центральности

```
second_town = sorted(G.nodes, key=lambda x: G.degree(x), reverse=True)[1]
second_degree = G.degree(second_town)
print(f"Town {second_town} has {second_degree} connecting roads.")
```

```
Town 12 has 8 connecting roads.
```

К городу 12 примыкают 12 дорог, то есть он отстает от города 3 всего на одну. А что бы мы делали, если бы у этих двух городов были одинаковые степени? Попробуем выяснить. На рис. 19.2 мы видим дорогу, соединяющую города 3 и 9. Предположим, что она закрыта на ремонт. Это вынуждает нас убрать из `G` одно ребро (листинг 19.4). Выполнение `G.remove(3, 9)` приведет к удалению ребра между узлами 3 и 0, в результате чего степень города 3 сравняется со степенью города 12. В сети есть и другие важные структурные изменения, которые мы визуализируем на рис. 19.3.

Листинг 19.4. Удаление ребра у наиболее центрального узла

```
np.random.seed(1)
G.remove_edge(3, 9)
assert G.degree(3) == G.degree(12)
nx.draw(G, with_labels=True, node_color=node_colors)
 plt.show()
```

После удаления ребра центральность городов 3 и 12 стала одинаковой

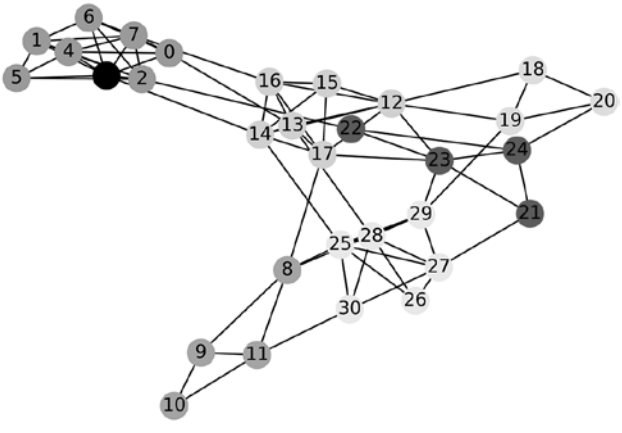


Рис. 19.3. Сеть дорог между городами после закрытия одной из них. Города 3 и 12 теперь имеют одинаковую степень центральности, но при этом город 12 находится в более центральной позиции графа. Его округ граничит со многими другими округами, в то время как закрытая дорога частично изолировала город 3 от остального мира

Удаление дороги частично изолировало город 3, а также его соседей. Этот город находится в округе 0, который включает в себя города с 0-го по 7-й. До этого одна дорога, проходящая через город 3, связывала округ 0 с округом 1. Теперь же ее нет, значит, город 3 стал менее доступным. А вот город 12, напротив, продолжает оставаться соседом многих округов.

Теперь город 3 является менее центральным, чем город 12, но степени обоих равны. Мы раскрыли серьезный недочет степени центральности: связующие дороги не имеют значения, если они не ведут к важным точкам.

Представьте, что у города есть 1000 дорог и все они ведут в тупик. А теперь представьте город всего с четырьмя дорогами, каждая из которых ведет к более крупному мегаполису. В такой ситуации, несмотря на разительную разницу в степенях в пользу первого города, логично ожидать, что трафик через второй город будет активнее. По той же логике предполагается, что город 12 привлечет больший поток трафика, чем город 3, хотя они и имеют равные степени центральности. В действительности можно даже количественно оценить эти различия, применяя случайные симуляции. В следующем разделе мы измерим центральность городов путем симуляции трафика между ними.

19.1.1. Измерение центральности с помощью симуляции трафика

Вскоре мы смоделируем трафик в своей сети, запрограммировав 20 000 автомобилей на произвольное движение между 31 городом. Однако сначала нужно симулировать случайный путь одной машины. Она начнет путешествие в произвольном городе *i*. Затем водитель случайным образом выберет одну из проходящих через него дорог `G.degree(i)` и посетит случайный город, соседствующий с *i*. Далее произойдет выбор очередной случайной дороги. Этот процесс будет повторяться до тех пор, пока машина не объедет десять городов. В листинге 19.5 мы определим функцию `random_drive` для выполнения описанной симуляции на графе `G`. Эта функция будет возвращать конечное местоположение машины.

ПРИМЕЧАНИЕ

В теории графов этот тип случайного обхода узлов называется случайным блужданием.

Листинг 19.5. Симуляция случайного маршрута одной машины

```
np.random.seed(0)
def random_drive(num_stops=10):
    town = np.random.choice(G.nodes)
    for _ in range(num_stops):
        town = np.random.choice(G[town])

    return town

destination = random_drive()
print(f"After driving randomly, the car has reached Town {destination}.")

After driving randomly, the car has reached Town 24.
```

Эта функция симулирует случайный путь машины через num_paths городов

Место старта машины выбирается случайно

Машина едет в случайный соседний город

Листинг 19.6 повторяет эту симуляцию с 20 000 машин и подсчитывает их количество в каждом из 31 города. Полученное число машин представляет собой степень загруженности дорог в городе. В итоге выводим показатель трафика в наиболее посещаемом городе. Мы также замеряем время выполнения 20 000 итераций, чтобы иметь представление о затратах времени на выполнение симуляции.

ПРИМЕЧАНИЕ

Наша симуляция сильно упрощена. В реальности люди не ездят случайным образом из города в город. Трафик возрастает на определенных участках, потому что они расположены между пунктами, которые люди посещают часто, где они находят обширный рынок жилья, рынок труда, множество магазинов и прочие удобства. Но упрощение не несет вреда. Оно даже полезно! Построенная нами модель обобщается за пределы движения автомобилей. Ее можно применить к веб-трафику, а также к потоку социальных взаимодействий. Вскоре мы расширим свой анализ на эти категории графов, а это оказалось бы невозможным, будь наша модель менее простой и более конкретной.

Листинг 19.6. Симуляция трафика с использованием 20 000 машин

```
import time
np.random.seed(0)
car_counts = np.zeros(len(G.nodes))
num_cars = 20000

start_time = time.time()
for _ in range(num_cars):
    destination = random_drive()
    car_counts[destination] += 1

central_town = car_counts.argmax()
traffic = car_counts[central_town]
running_time = time.time() - start_time
print(f"We ran a {running_time:.2f} second simulation.")
print(f"Town {central_town} has the most traffic.")
print(f"There are {traffic:.0f} cars in that town.")

We ran a 3.47 second simulation.
Town 12 has the most traffic.
There are 1015 cars in that town.
```

Сохраняет показатели трафика не в словаре, а в массиве для упрощения их векторизации в последующем коде

В городе 12 зарегистрирован максимальный трафик из более чем 1000 машин. И это неудивительно, учитывая, что города 12 и 3 имеют максимальную степень центральности. Исходя из предыдущей дискуссии, мы также ожидаем, что город 12 будет иметь более насыщенный трафик, чем город 3. Проверим это (листинг 19.7).

Листинг 19.7. Проверка трафика в городе 3

```
print(f"There are {car_counts[3]:.0f} cars in Town 3.")

There are 934 cars in Town 3.
```

Наши ожидания подтвердились. В городе 3 зафиксировано менее 1000 машин. Здесь важно отметить, что сравнить количество машин может быть трудно, особенно при больших значениях `num_cars`. В связи с этим лучше будет заменить прямые подсчеты вероятностями, разделив их на число симуляций. Выполнив `car_counts / num_cars`, мы получим массив вероятностей — каждая *i*-я вероятность в нем будет равна вероятности того, что случайно путешествующая машина приедет в город *i*. Далее выведем эти вероятности для городов 12 и 3 (листинг 19.8).

Листинг 19.8. Преобразование показателей трафика в вероятности

```
probabilities = car_counts / num_cars
for i in [12, 3]:
    prob = probabilities[i]
    print(f"The probability of winding up in Town {i} is {prob:.3f}.")

The probability of winding up in Town 12 is 0.051.
The probability of winding up in Town 3 is 0.047.
```

Согласно нашей случайной симуляции в городе 12 машина оказывается в 5,1 % случаев, а в городе 3 — в 4,7 %. Таким образом, мы показали, что город 12 является более центральным, чем город 3. К сожалению, процесс симуляции медленный и плохо масштабируется на более обширные графы.

ПРИМЕЧАНИЕ

Выполнение симуляций заняло у нас 3,47 с. Это время выглядит вполне адекватным, но в случае больших графов для оценки вероятностей потребуется больше симуляций. Все дело в законе больших чисел, с которым мы познакомились в главе 4. Граф, содержащий в 1000 раз больше узлов, потребует в 1000 раз больше симуляций, что приведет к увеличению времени выполнения примерно до часа.

Можно ли вычислить эти вероятности без симуляции потока 20 000 машин? Да! В следующем разделе мы покажем, как вычислять вероятности для трафика, используя простое матричное умножение.

19.2. ВЫЧИСЛЕНИЕ ВЕРОЯТНОСТИ ПУТЕШЕСТВИЯ В ТОТ ИЛИ ИНОЙ ГОРОД С ПОМОЩЬЮ МАТРИЧНОГО УМНОЖЕНИЯ

Симуляцию трафика можно смоделировать математически, используя матрицы и векторы. Мы разделим этот процесс на простые и удобные для выполнения части. Представьте себе машину, которая собирается выехать из города 0 в направлении одного из соседних городов. Перед ней стоит выбор из `G.degree(0)` городов, значит, вероятность проезда из города 0 в любой соседний город равна `1 / G.degree(0)`. Давайте ее вычислим (листинг 19.9).

Листинг 19.9. Вычисление вероятности путешествия в соседний город

```
num_neighbors = G.degree(0)
prob_travel = 1 / num_neighbors
print("The probability of traveling from Town 0 to one of its "
      f"{G.degree(0)} neighboring towns is {prob_travel}")
```

```
The probability of traveling from Town 0 to one of its
5 neighboring towns is 0.2
```

Если мы находимся в городе 0, а город *i* является его соседом, то есть 20%-ная вероятность того, что мы попадем из него именно туда. Естественно, если город *i* не соседствует с городом 0, она упадет до 0. Отслеживать вероятности для любого возможного `i` можно с помощью вектора `v`. Значение `v[i]` будет равно `0.2`, если `i` находится в `G[0]`, и `0` — в противном случае. Вектор `v` называется *вектором*

перехода, поскольку отслеживает вероятность перехода из города 0 в другие города. Вычислить его можно несколькими способами.

- Выполнить `np.array([0.2 if i in G[0] else 0 for i in G.nodes])`. Каждый *i*-й элемент будет равен 0,2, если `i` находится в `G[0]`, и 0 — в противном случае.
- Выполнить `np.array([1 if i in G[0] else 0 for i in G.nodes]) * 0.2`. Здесь мы просто умножаем 0,2 на двоичный вектор, отслеживающий присутствие или отсутствие ребер, ведущих к `G[0]`.
- Выполнить `M[:,0] * 0.2`, где `M` — матрица смежности. Каждый столбец в ней отслеживает двоичное присутствие или отсутствие ребер между узлами, значит, столбец `0` в `M` будет соответствовать массиву из предыдущего примера.

Третья операция вычисления самая простая. Естественно, 0,2 равно `1 / G.degree(0)`. Как говорилось в начале главы, эту степень можно вычислить также сложением значений в столбце матрицы смежности. Таким образом, вектор перехода можно вычислить выполнением `M[:,0]/M[:,0].sum()`. Код листинга 19.10 вычисляет этот вектор, используя все перечисленные методологии.

ПРИМЕЧАНИЕ

На данный момент матрица смежности M сохраняется с помощью переменной adjacency_matrix. Однако эта матрица не учитывает удаленное ребро между городами 3 и 9, поэтому мы вычисляем ее повторно, выполнив adjacency_matrix = nx.to_numpy_array(G).

Листинг 19.10. Вычисление вектора перехода

```
transition_vector = np.array([0.2 if i in G[0] else 0 for i in G.nodes])

adjacency_matrix = nx.to_numpy_array(G)
v2 = np.array([1 if i in G[0] else 0 for i in G.nodes]) * 0.2
v3 = adjacency_matrix[:,0] * 0.2
v4 = adjacency_matrix[:,0] / adjacency_matrix[:,0].sum()

for v in [v2, v3, v4]:
    assert np.array_equal(transition_vector, v)

print(transition_vector)
```

Повторно вычисляет матрицу смежности, чтобы учесть ранее удаленное ребро

Вычисляет вектор перехода прямо из столбца матрицы смежности

Все четыре полученные версии вектора перехода идентичны

```
[0. 0. 0. 0.2 0.2 0. 0.2 0.2 0. 0. 0. 0. 0. 0.2 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ]
```

Вычислить вектор перехода для любого города *i* можно выполнением `M[:,i] / M[:,i].sum()`, где `M` — матрица смежности. Более того, эти векторы можно вычислить все разом, выполнив `M / M.sum(axis=0)`. Эта операция делит каждый столбец матрицы на связанную с ним степень. В итоге мы получаем матрицу, столбцы которой соответствуют векторам перехода. Такая матрица (рис. 19.4) называется

матрицей переходов. Далее мы ее вычислим: исходя из наших ожиданий, выходной столбец 0 должен равняться `transition_vector` города 0 (листинг 19.11).

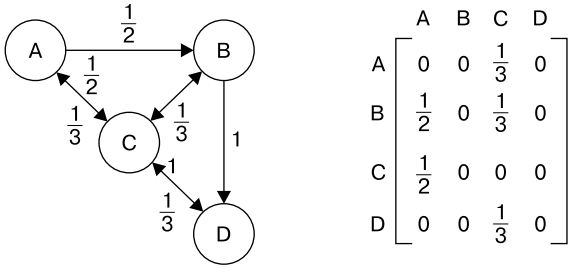


Рис. 19.4. Если M — матрица смежности, тогда M / M.sum(axis=0) соответствует матрице переходов, даже если смежности направленные. Здесь показан направленный граф. Его ребра обозначены вероятностями переходов, которые также представлены в матрице, соответствующей M / M.sum(axis=0). Каждый столбец в ней является вектором перехода, чьи вероятности суммируются в 1,0. Согласно этой матрице вероятность проезда из A в C равна 1/2, а из C в A — 1/3

Листинг 19.11. Вычисление матрицы переходов

```
transition_matrix = adjacency_matrix / adjacency_matrix.sum(axis=0)
assert np.array_equal(transition_vector, transition_matrix[:,0])
```

Наша матрица переходов обладает удивительным свойством — она позволяет вычислить вероятность поездки в каждый город всего в нескольких строках кода. Если мы хотим узнать вероятность того, что после десяти остановок окажемся в городе *i*, то достаточно проделать следующее.

1. Инициализировать вектор `v`, который равен `np.ones(31) / 31`.
2. Через десять итераций установить `v` равным `transition_matrix @ v`.
3. Вернуть `v[i]`.

Позднее мы выведем это прекрасное свойство с нуля, а сейчас докажем сделанное заявление, вычислив вероятности поездки в города 12 и 3 при помощи матричного умножения (листинг 19.12). Исходя из прежних наблюдений, мы ожидаем, что эти вероятности будут равны 0,051 и 0,047 соответственно.

Листинг 19.12. Вычисление вероятностей поездки при помощи матрицы переходов

```
v = np.ones(31) / 31
for _ in range(10):
    v = transition_matrix @ v

for i in [12, 3]:
    print(f"The probability of winding up in Town {i} is {v[i]:.3f}.")

The probability of winding up in Town 12 is 0.051.
The probability of winding up in Town 3 is 0.047.
```

Наши ожидания подтвердились.

Мы можем смоделировать поток трафика, используя серию операций матричного умножения. Они лежат в основе *центральности PageRank*, представляющей наиболее значимую меру важности узлов в истории. Центральность PageRank придумали основатели Google. Они применяли ее для ранжирования веб-страниц путем моделирования онлайн-путешествия пользователя в виде серии случайных щелчков по узлам графа Интернета. Эти щелчки по страницам аналогичны приезду машины в случайные города. Вероятность посещения более популярных сайтов повышенная. Это открытие позволило Google обнаруживать релевантные сайты автоматически. В результате компания смогла обойти всех конкурентов и достичь ежегодного оборота в триллионы долларов. Иногда наука о данных окупается сполна.

Центральность PageRank легко вычислить, но трудно вывести. Тем не менее с помощью базовой теории вероятностей можно продемонстрировать, почему повторяющееся умножение `transition_matrix` дает вероятности поездки в тот или иной город.

ПРИМЕЧАНИЕ

Если вас не интересует выведение центральности PageRank, можете сразу перейти к следующему разделу. Там будет описано использование PageRank в NetworkX.

19.2.1. Выведение центральности PageRank на основе теории вероятностей

Нам известно, что `transition_matrix[i][j]` соответствует вероятности поездки из города *j* сразу в город *i*, но это предполагает, что наша машина находится в *j*. А что, если ее нахождение не определено? Вдруг вероятность ее нахождения в городе *j* равна 50 %. В таком случае вероятность намеченной поездки составит `0.5 * transition_matrix[i][j]`. В целом если вероятность текущего местонахождения равна `p`, то вероятность путешествия из текущей точки `j` в новую точку `i` составляет `p * transition_matrix[i][j]`.

Предположим, машина начинает свой путь в случайном городе и едет в другой. Какова вероятность того, что она поедет из города 3 в город 0? По факту машина может начать путешествие в любом из 31 города, значит, вероятность того, что им окажется город 3, равна `1 / 31`. Таким образом, вероятность путешествия из города 3 в город 0 составляет `transition_ matrix[0][3] / 31` (листинг 19.13).

Листинг 19.13. Вычисление вероятности проезда в определенный город из случайного места старта

```
prob = transition_matrix[0][3] / 31
print("Probability of starting in Town 3 and driving to Town 0 is "
      f"{prob:.2}")

Probability of starting in Town 3 and driving to Town 0 is 0.004
```

Существует множество вариантов того, как доехать до города 0 прямиком из случайной стартовой локации. Давайте выведем все ненулевые экземпляры `transition_matrix[0][i] / 31` для каждого возможного города *i* (листинг 19.14).

Листинг 19.14. Вычисление вероятности того, что случайные маршруты приведут в город 0

```
for i in range(31):
    prob = transition_matrix[0][i] / 31
    if not prob:
        continue

    print(f"Probability of starting in Town {i} and driving to Town 0 is "
          f"{prob:.2}")

print("\nAll remaining transition probabilities are 0.0")
```

```
Probability of starting in Town 3 and driving to Town 0 is 0.004
Probability of starting in Town 4 and driving to Town 0 is 0.0054
Probability of starting in Town 6 and driving to Town 0 is 0.0065
Probability of starting in Town 7 and driving to Town 0 is 0.0046
Probability of starting in Town 13 and driving to Town 0 is 0.0054

All remaining transition probabilities are 0.0
```

В город 0 нас ведут пять разных путей. Каждый из них имеет свою вероятность, а сумма этих вероятностей равна вероятности начать движение в любом случайном городе и проехать сразу в город 0 (рис. 19.5).

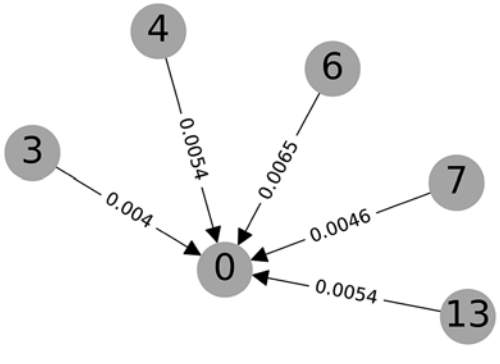


Рис. 19.5. Пять разных маршрутов ведут из случайного места старта в город 0. Каждому маршруту присвоена небольшая вероятность. Сложение этих значений дает вероятность начать в случайной локации и проехать напрямую в город 0

Далее мы вычислим эту вероятность. Более того, сравним полученный показатель с результатом случайных симуляций. Их мы прогоним, выполнив `random_drive(num_stops=1)` 50 000 раз (листинг 19.15). Это даст нам частоту, с которой

город 0 оказывается первой остановкой в нашем рандомизированном путешествии. Мы ожидаем, что эта частота будет аппроксимироваться к сумме вероятностей.

Листинг 19.15. Вычисление вероятности первой остановки в городе 0

```
np.random.seed(0)
prob = sum(transition_matrix[0][i] / 31 for i in range(31))
frequency = np.mean([random_drive(num_stops=1) == 0
                     for _ in range(50000)])

print(f"Probability of making our first stop in Town 0: {prob:.3f}")
print(f"Frequency with which our first stop is Town 0: {frequency:.3f}")

Probability of making our first stop in Town 0: 0.026
Frequency with which our first stop is Town 0: 0.026
```

Найденная вероятность согласуется с наблюдаемой частотой: город 0 окажется в нашем путешествии первой остановкой примерно в 2,6 % случаев. Стоит отметить, что ее можно вычислить более кратко с помощью операции скалярного произведения векторов, нужно лишь выполнить `transition_matrix[0] @ v`, где `v` — это 31-элементный вектор, все элементы которого равны `1 / 31`. Код листинга 19.16 выполняет это краткое вычисление.

Листинг 19.16. Определение вероятности путешествия с помощью скалярного произведения

```
v = np.ones(31) / 31
assert transition_matrix[0] @ v == prob
```

Выполнение `transition_matrix[i] @ v` ведет к возвращению вероятности того, что первая остановка случится в городе *i*. Эту вероятность можно вычислить для каждого города, выполнив `[transition_ matrix[i] @ v for i in range(31)`. Естественно, данная операция равноценна матричному умножению `transition_matrix` и `v`, поэтому `transition_matrix @ v` возвращает все вероятности первой остановки. Код листинга 19.17 вычисляет этот массив с помощью `stop_1_probabilities` и выводит вероятность первой остановки в городе 12. Ее значение должно аппроксимироваться к частоте, вычисленной путем случайных симуляций.

Листинг 19.17. Вычисление всех вероятностей первой остановки

```
np.random.seed(0)
stop_1_probabilities = transition_matrix @ v
prob = stop_1_probabilities[12]
frequency = np.mean([random_drive(num_stops=1) == 12
                     for _ in range(50000)])

print('First stop probabilities:')
print(np.round(stop_1_probabilities, 3))
print(f"\nProbability of making our first stop in Town 12: {prob:.3f}")
print(f"Frequency with which our first stop is Town 12: {frequency:.3f}")
```

```
First stop probabilities:
[0.026 0.033 0.045 0.046 0.033 0.019 0.025 0.038 0.033 0.031 0.019 0.041
 0.052 0.03  0.036 0.019 0.031 0.039 0.023 0.031 0.027 0.019 0.018 0.044
 0.038 0.046 0.015 0.045 0.04 0.035 0.023]

Probability of making our first stop in Town 12: 0.052
Frequency with which our first stop is Town 12: 0.052
```

Мы установили, что `transition_matrix @ v` возвращает вектор вероятностей первой остановки. Теперь нужно доказать, что итеративное повторение этой операции в итоге даст вектор вероятностей десятой остановки. Однако сначала мы ответим на простой вопрос: «Какова вероятность сделать вторую остановку в городе *i*?» Из прежних рассуждений нам известно следующее.

- Вероятность первой остановки в городе *j* равна `stop_1_probabili- ties[j]`.
- Если вероятность нахождения в определенном месте равна `p`, то вероятность путешествия из оттуда в локацию *i* равна `p * transition_matrix[i][j]`.
- В связи с этим вероятность сделать первую остановку в городе *j*, а затем поехать в город *i* равна `p * transition_matrix[i][j]`, где `p = stop_1_probabilities[j]`.
- Вероятность этой поездки можно вычислить для каждого возможного города *j*.
- Сумма этих вероятностей равна вероятности сделать первую остановку в случайном городе, а затем поехать прямо в город *i*. Эта сумма равна `sum(p * transition_ matrix[i][j] for j, p in enumerate(stop_1_probabilities))`.
- Данную вероятность можно вычислить более кратко с помощью операции скалярного умножения, которая будет выглядеть как `transition_matrix[i] @ stop_1_probabilities`.

Вероятность сделать вторую остановку в городе *i* равна `transition_matrix[i] @ stop_1_probabilities`. Ее можно вычислить для каждого города умножением матрицы на вектор. Таким образом, `transition_matrix @ stop_1_probabilities` возвращает все вероятности второй остановки. Однако `stop_1_probabilities` равно `transition_ matrix @ v`, значит, вероятности второй остановки также равны `transition_matrix @ transition_matrix @ v`.

Далее мы подтвердим свои расчеты, вычислив вероятности второй остановки. После этого выведем вероятность выполнения второй остановки в городе 12, которая должна аппроксимироваться к частоте, найденной путем случайных симуляций (листинг 19.18).

Листинг 19.18. Вычисление всех вероятностей второй остановки

```
np.random.seed(0)
stop_2_probabilities = transition_matrix @ transition_matrix @ v
prob = stop_2_probabilities[12]
```

```
frequency = np.mean([random_drive(num_stops=2) == 12
                                    for _ in range(50000)])

print('Second stop probabilities:')
print(np.round(stop_2_probabilities, 3))
print(f"\nProbability of making our second stop in Town 12: {prob:.3f}")
print(f"Frequency with which our second stop is Town 12: {frequency:.3f}")

Second stop probabilities:
[0.027 0.033 0.038 0.043 0.033 0.023 0.028 0.039 0.039 0.026 0.021 0.032
 0.048 0.034 0.039 0.023 0.032 0.041 0.023 0.029 0.025 0.024 0.023 0.04
 0.029 0.043 0.021 0.036 0.036 0.042 0.031]

Probability of making our second stop in Town 12: 0.048
Frequency with which our second stop is Town 12: 0.048
```

Мы смогли вывести вероятности второй остановки прямо из вероятностей первой. Аналогичным образом можно вывести вероятности и для третьей остановки. Если повторять этот процесс, то можно без проблем показать, что `stop_3_probabilities` равно `transition_matrix @ stop_2_probabilities`. Естественно, этот вектор также равняется `M @ M @ M @ v`, где `M` — матрица переходов.

Повторяя этот процесс, можно вычислить вероятности четвертой остановки, затем пятой и так до вероятностей *N*-й остановки. Чтобы получить вероятности *N*-й остановки, нужно лишь выполнить `M @ v` в течение *N* итераций. Давайте определим функцию, вычисляющую все вероятности *N*-й остановки прямо из матрицы переходов `M` (листинг 19.19).

ПРИМЕЧАНИЕ

Мы работаем со случайным процессом, состоящим из N отдельных шагов, в котором вероятности N-го шага можно вычислить непосредственно из шага N – 1. Подобные процессы называются цепями Маркова в честь математика Андрея Маркова, занимавшегося изучением случайных процессов.

Листинг 19.19. Вычисление вероятностей N-й остановки

```
def compute_stop_likelihoods(M, num_stops):
    v = np.ones(M.shape[0]) / M.shape[0]
    for _ in range(num_stops):
        v = M @ v

    return v

stop_10_probabilities = compute_stop_likelihoods(transition_matrix, 10)
prob = stop_10_probabilities[12]

print('Tenth stop probabilities:')
print(np.round(stop_10_probabilities, 3))
print(f"\nProbability of making our tenth stop in Town 12: {prob:.3f}")
```

```
Tenth stop probabilities:
[0.029 0.035 0.041 0.047 0.035 0.023 0.029 0.041 0.034 0.021 0.014 0.028
 0.051 0.038 0.044 0.025 0.037 0.045 0.02  0.026 0.02  0.02  0.019 0.039
 0.026 0.047 0.02 0.04 0.04 0.04 0.027]

Probability of making our tenth stop in Town 12: 0.051
```

Как уже говорилось, интерактивное матричное умножение формирует основу центральности PageRank. В следующем разделе мы сравним свои результаты с реализацией алгоритма PageRank из NetworkX, чтобы глубже понять его устройство.

19.2.2. Вычисление центральности PageRank с помощью NetworkX

Функция для вычисления центральности PageRank есть в библиотеке NetworkX. Вызов `nx.pagerank(G)` вернет словарь сопоставлений между ID узлов и значениями их центральности. Давайте выведем центральность PageRank для города 12 (листинг 19.20). Будет ли она равняться 0,051?

Листинг 19.20. Вычисление центральности PageRank с помощью NetworkX

```
centrality = nx.pagerank(G)[12]
print(f"The PageRank centrality of Town 12 is {centrality:.3f}.")

The PageRank centrality of Town 12 is 0.048.
```

Выведенное значение PageRank равно 0,048, то есть чуть ниже ожидаемого. Эта разница обусловлена небольшой корректировкой, которая гарантирует работу PageRank для всех возможных сетей. Напомню, что изначально этот алгоритм предназначался для моделирования случайных щелчков в графе веб-ссылок. В этом графе имеются направленные ребра, в связи с чем определенные веб-страницы могут исходящих ссылок не иметь. Таким образом, если пользователь при серфинге Интернета будет опираться на исходящие ссылки, то он может застрять на тупиковой странице (рис. 19.6). Чтобы исключить эту проблему, разработчики PageRank предположили, что пользователь в конечном итоге устанет щелкать на ссылках и начнет путешествие заново, перейдя на абсолютно случайную веб-страницу. Иными словами, он телепортируется в один из узлов `len(G.nodes)` графа Интернета. Создатели PageRank запрограммировали выполнение такой телепортации в 15 % переходов, гарантировав, что пользователь никогда не окажется в узле, лишенном исходящих ссылок.

В нашем примере с сетью дорог такая телепортация будет аналогична вызову вертолетной службы. Представьте, что в 15 % визитов в города мы начинаем скучать от местных достопримечательностей. Тогда мы вызываем вертолет, который уносит нас в совершенно случайный город. Когда мы оказываемся в воздухе, вероятность

полететь в любой город равняется 1/31. После приземления мы арендуем машину и продолжаем путешествие, используя существующую сеть дорог. Таким образом, в 15 % случаев мы летим из города *i* в город *j* с вероятностью 1/31.

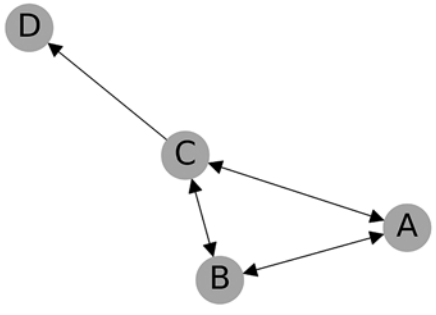


Рис. 19.6. Направленный граф из четырех узлов. Мы можем свободно путешествовать между взаимосвязанными узлами A, B и C, но узел D исходящих ребер не имеет. Рано или поздно путешествие приведет нас из C в D, где мы застрянем навсегда. Телепортация же исключает такую возможность. В 15 % переходов мы будем телепортироваться в случайно выбранный узел. Даже если мы отправимся в узел D, то все равно сможем телепортироваться обратно в узлы A, B и C

В оставшихся 85 % мы едем из города *i* в город *j* с вероятностью `transition_matrix[j][i]`. Следовательно, фактическая вероятность путешествия равна взвешенному среднему `transition_matrix[j][i]` и `1 / 31`. Соответствующими весами здесь являются 0,85 и 0,15. Как говорилось в главе 5, взвешенное среднее можно вычислить с помощью функции `np.average`. Еще его можно вычислить непосредственно выполнением `0.85 * transition_matrix[j][i] + 0.15 / 31`.

Получение взвешенного среднего всех элементов матрицы переходов даст совершенно новую матрицу. Далее мы передадим ее в функцию `compute_stop_likelihoods` и выведем вероятность путешествия в город 12. Ожидается, что эта вероятность упадет с 0,051 до 0,048 (листинг 19.21).

Листинг 19.21. Встраивание в модель рандомизированной телепортации

```
                Умножает transition_matrix на 0,85, после чего прибавляет к каждому
                элементу 0.15 / 31. Более подробно об арифметических операциях над
                                    2D-массивами NumPy читайте в главе 13

new_matrix = 0.85 * transition_matrix + 0.15 / 31
stop_10_probabilities = compute_stop_likelihoods(new_matrix, 10)

prob = stop_10_probabilities[12]
print(f"The probability of winding up in Town 12 is {prob:.3f}.")

The probability of winding up in Town 12 is 0.048.
```

Полученный вывод согласуется с результатом NetworkX. А сохранит ли он свою согласованность, если увеличить количество остановок с 10 до 1000? Сейчас выясним. Мы передадим 1000 остановок в `compute_stop_likelihoods` и проверим, является ли показатель PageRank для города 12 по-прежнему равным 0,048 (листинг 19.22).

Листинг 19.22. Вычисление вероятности после 1000 остановок

```
prob = compute_stop_likelihoods(new_matrix, 1000)[12]
print(f"The probability of winding up in Town 12 is {prob:.3f}.")

The probability of winding up in Town 12 is 0.048.
```

Центральность по-прежнему составляет 0,048. Десяти итераций было достаточно для схождения к стабильному значению. Почему так произошло? Вычисление PageRank — это не более чем повторяющееся умножение матрицы и вектора. Все элементы умножаемых векторов являются значениями между 0 и 1. Возможно, это покажется вам знакомым: вычисление нами PageRank практически идентично степенному методу, с которым мы познакомились в главе 14. Этот метод заключается в итеративном получении произведения матрицы и вектора. В конечном итоге оно сходится к собственному вектору матрицы. Напомню, что собственный вектор `v` матрицы `M` — это особый вектор, для которого `norm(v) == norm(M @ v)`. Обычно десяти итераций оказывается достаточно для такого схождения. Получается, что наши значения PageRank сходятся, так как мы выполняем степенной метод. Это доказывает, что наш вектор центральности — это собственный вектор матрицы переходов. Таким образом, центральность PageRank нераздельно связана с красотой математики, стоящей за уменьшением размерности.

Имея любой граф `G`, мы вычисляем его центральности PageRank следующим образом.

1. Получаем матрицу смежности `M` графа.
2. Преобразуем ее в матрицу переходов выполнением `M = M / M.sum(axis=0)`.
3. Обновляем `M`, чтобы включить случайную телепортацию. Делается это путем получения взвешенного среднего `M` и `1 / n`, где `n` равно числу узлов в графе. Веса обычно устанавливаются на 0,85 и 0,15, значит, взвешенное среднее равно `0.85 * M + 0.15 / n`.
4. Возвращаем самый большой (и единственный) собственный вектор матрицы `M`. Вычислить его можно выполнением `v = M @ v` в течение примерно десяти итераций. Изначально вектор `v` устанавливается равным `np.ones(n) / n`.

Матрицы переходов связывают теорию графов с теорией вероятностей и теорией матриц. Их можно использовать также для кластеризации данных сети при помощи *марковского алгоритма кластеризации*. В следующем разделе мы задействуем матрицы переходов для кластеризации сообществ в графах.

ТИПИЧНЫЕ ВЫЧИСЛЕНИЯ ЦЕНТРАЛЬНОСТИ В NETWORKX

- `G.in_degree(i)` — возвращает входящую степень узла `i` в направленном графе.
- `G.degree(i)` — возвращает степень узла `i` в ненаправленном графе.
- `nx.pagerank(G)` — возвращает словарь сопоставлений между ID узлов и их центральностями PageRank.

19.3. ОБНАРУЖЕНИЕ СООБЩЕСТВ С ПОМОЩЬЮ МАРКОВСКОЙ КЛАСТЕРИЗАЦИИ

Граф `G` представляет собой сеть городов, часть из которых попадает в локализованные округа. Сейчас ID округов нам известны, ну а если бы мы их не знали? Как определить округа? Этот вопрос мы разберем с помощью визуализации без окрашивания узлов в соответствии с округами (листинг 19.23; рис. 19.7).

Листинг 19.23. Построение G без окрашивания узлов по округам

```
np.random.seed(1)
nx.draw(G)
plt.show()
```

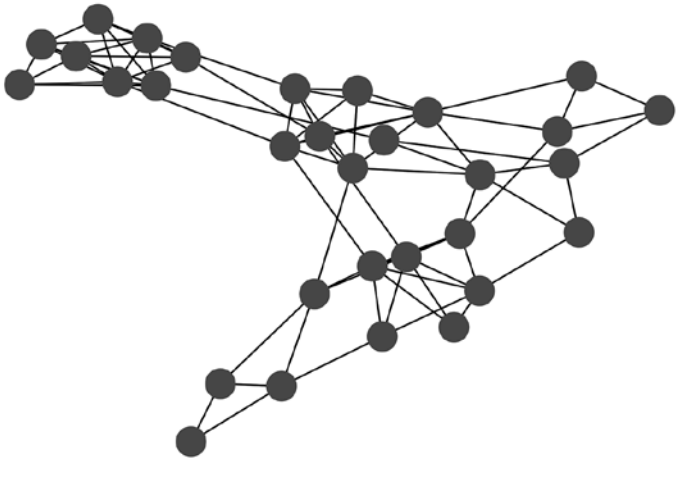


Рис. 19.7. Сеть дорог между городами. Эти города не окрашивались на основе принадлежности к округам, но отдельные округа мы все равно распознать в этой сети можем — они представлены в виде кластеризованных в пространстве групп

Наш граф не имеет ни цветов, ни меток, но это не мешает обнаружить потенциальные округа — они изображены в виде тесно связанных кластеров узлов. В теории графов подобные кластеры формально называются *сообществами*. Графы с явно видимыми сообществами имеют *структуру сообществ*. Она характерна для многих типов графов, включая графы городов и друзей в социальных сетях.

ПРИМЕЧАНИЕ

Некоторые распространенные графы структуры сообществ не имеют. К примеру, в Интернете нет тесно кластеризованных сообществ веб-страниц.

Процесс нахождения сообществ графа называется *обнаружением сообществ*, или *кластеризацией графа*. Существует множество алгоритмов кластеризации графов, часть из которых строится на симуляциях потока трафика.

Как можно использовать трафик для нахождения кластеров округов в нашей сети? Нам известно, что города одного округа с большей вероятностью соединены дорогой, чем города из разных округов. Поэтому если мы поедем в соседний город, то, скорее всего, останемся в том же округе. В структурированных по сообществам графах эта логика сохраняется, даже если мы едем через два города. Предположим, что мы отправились из города i в город j, а затем в город k. Исходя из структуры нашей сети города i и k наверняка окажутся в одном округе, и вскоре мы это утверждение подтвердим. Однако сначала нужно вычислить вероятность перехода из города i в город k через две остановки. Эта вероятность называется *случайным потоком*, или просто *потоком*. Поток тесно связан с вероятностью перехода, но, в отличие от нее, он охватывает города, которые не имеют прямой связи. Нам нужно вычислить этот поток между каждой парой городов и сохранить результат в *матрице потоков*. Позднее мы покажем, что средний поток выше в городах, принадлежащих к одному сообществу.

ПРИМЕЧАНИЕ

Как правило, в теории сетей поток — очень туманное понятие. Однако в марковском алгоритме кластеризации это определение ограничивается вероятностью конечного путешествия между узлами.

Как же вычислить матрицу значений потока? Один из подходов состоит в симуляции путешествия между случайными городами с двумя остановками. Затем симулированные частоты можно преобразовать в вероятности. Однако гораздо проще вычислить эти вероятности напрямую. Используя немного математики, можно показать, что матрица потоков равна `transition_matrix @ transition_matrix`.

ПРИМЕЧАНИЕ

Подтвердить это утверждение можно следующим образом. Ранее мы показали, что вероятности второй остановки равны transition_matrix @ transition_matrix @ v. Более того, transition_matrix @ transition_matrix дает новую матрицу M. Значит, вероятности второй остановки равны M @ v. По существу, M служит той же цели, что и transition_matrix, но отслеживает две остановки, а не одну. Получается, что M вполне вписывается в наше определение матрицы потоков.

По сути, случайная симуляция аппроксимирует произведение матрицы переходов с самой собой. Давайте это проверим (листинг 19.24).

Листинг 19.24. Сравнение вычисленного потока со случайными симуляциями

```
np.random.seed(0)
flow_matrix = transition_matrix @ transition_matrix

simulated_flow_matrix = np.zeros((31, 31))
num_simulations = 10000
for town_i in range(31):
    for _ in range(num_simulations):
        town_j = np.random.choice(G[town_i])
        town_k = np.random.choice(G[town_j])
        simulated_flow_matrix[town_k][town_i] += 1

simulated_flow_matrix /= num_simulations
assert np.allclose(flow_matrix, simulated_flow_matrix, atol=1e-2)
```

Отслеживает частоту, с которой мы попадаем из города i в город k после двух остановок

Убеждается в близком сходстве симулированных частот с непосредственно вычисленным потоком

Наша `flow_matrix` согласуется со случайными симуляциями. Далее проверим теорию о том, что поток между городами в одном округе выше, чем в прочих. Напомню, что каждому городу в `G.nodes` был присвоен ID округа. Мы считаем, что средний поток между городами *i* и *j* выше, если `G.nodes[i]['county_id']` равно `G.nodes[j]['county_ id']`. Убедиться в этом можно делением потоков на два списка: `county_flows` и `between_county_flows`. В них фиксируются потоки внутри и вне округов соответственно. Мы построим гистограмму для каждого из этих списков и сравним средние значения потоков в них (рис. 19.8). Если мы правы, то `np.mean(county_flows)` должно оказаться заметно выше, чем средний поток второго списка. Помимо этого, проверим, являются ли какие-либо потоки округов меньше `np.min(county_flows)`.

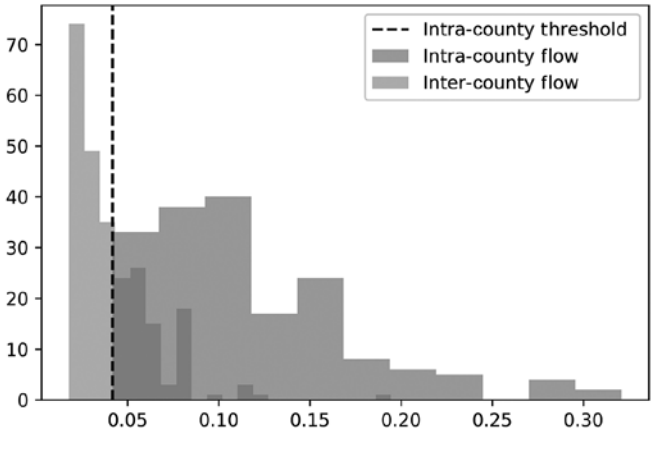


Рис. 19.8. Две гистограммы, представляющие все ненулевые потоки между округами и внутри них. Разделение между типами потоков очевидно. Потоки внутри округов сильно смещены влево: порога в районе 0,042 достаточно, чтобы отделить 132 потока между округами от распределения потоков внутри округов

Заметьте, что для чистоты сравнения нужно рассматривать только ненулевые потоки. Поэтому необходимо пропускать `flow_matrix[j][i]`, если ее значение равно нулю. Нулевое значение подразумевает невозможность проезда из города *i* в город *j* всего с двумя остановками (вероятность этого равна нулю). Необходимо не менее трех остановок, что указывает на значительную удаленность городов друг от друга. Это почти гарантирует, что они находятся в разных округах. Таким образом, включение нулевых потоков ложно сместило бы распределение значений потоков внутри округов в сторону нуля. Далее попробуем проанализировать потоки только между близко расположенными городами (листинг 19.25).

Листинг 19.25. Сравнение распределений потоков внутри округов и между ними

```
def compare_flow_distributions():
    county_flows = []
    between_county_flows = []
    for i in range(31):
        county = G.nodes[i]['county_id']
        nonzero_indices = np.nonzero(flow_matrix[:,i])[0]
        for j in nonzero_indices:
            flow = flow_matrix[j][i]

            if county == G.nodes[j]['county_id']:
                county_flows.append(flow)
            else:
                between_county_flows.append(flow)

    mean_intra_flow = np.mean(county_flows)
    mean_inter_flow = np.mean(between_county_flows)
    print(f"Mean flow within a county: {mean_intra_flow:.3f}")
    print(f"Mean flow between different counties: {mean_inter_flow:.3f}")

    threshold = min(county_flows)
    num_below = len([flow for flow in between_county_flows
                     if flow < threshold])
    print(f"The minimum intra-county flow is approximately {threshold:.3f}")
    print(f"{num_below} inter-county flows fall below that threshold.")

    plt.hist(county_flows, bins='auto', alpha=0.5,
             label='Intra-County Flow')
    plt.hist(between_county_flows, bins='auto', alpha=0.5,
             label='Inter-County Flow')
    plt.axvline(threshold, linestyle='--', color='k',
                label='Intra-County Threshold')
    plt.legend()
    plt.show()

compare_flow_distributions()
```

Отслеживает ненулевые потоки внутри округов

Отслеживает ненулевые потоки между округами

Перебирает в столбце i только ненулевые значения

Проверяет, находятся ли два города в одном округе

Отслеживает все потоки между округами, которые меньше минимальных потоков внутри округов

Построение гистограммы потоков внутри округов

Построение гистограммы потоков между округами

```
Mean flow within a county: 0.116
Mean flow between different counties: 0.042
The minimum intra-county flow is approximately 0.042
132 inter-county flows fall below that threshold.
```

Средний поток между округами в три раза меньше, чем средний поток между городами внутри разных округов. Это отличие явно видно в построенном распределении: потоки ниже порога 0,04 — это гарантированно потоки между округами. Таким образом, можно изолировать города из разных округов, используя явный порог. Естественно, мы сможем наблюдать этот порог только благодаря тому, что знаем ID округов. В реальном же сценарии их фактические ID были бы неизвестны, в связи с чем явно определить порог разделения не вышло бы. В таком случае пришлось бы предположить, что он имеет малое значение, например 0,01. Допустим, мы сделали такое предположение относительно наших данных. Сколько тогда ненулевых потоков между округами будут меньше 0,01? Сейчас выясним (листинг 19.26).

Листинг 19.26. Уменьшение порога разделения

```
num_below = np.count_nonzero((flow_matrix > 0.0) & (flow_matrix < 0.01))
print(f"{num_below} inter-county flows fall below a threshold of 0.01")

0 inter-county flows fall below a threshold of 0.01
```

Ни одно из значений потоков не оказывается ниже жесткого порога 0,01. Что же делать? Один из вариантов — манипулировать распределением потоков, чтобы подчеркнуть разницу между большими и малыми значениями. В идеале нужно сделать так, чтобы малые оказались ниже 0,01, а большие при этом не уменьшились. Это можно выполнить с помощью простого процесса, называемого *раздуванием*. Раздувание влияет на значения вектора, сохраняя его среднее постоянным. Значения ниже этого среднего уменьшаются, а остальные — возрастают. Продемонстрируем этот процесс на простом примере. Предположим, мы раздуваем некий вектор `v`, представленный `[0.7, 0.3]`. Среднее этого вектора равно 0,5. Нам нужно увеличить `v[0]`, попутно уменьшив `v[1]`. Частичным решением будет возвести в квадрат каждый элемент `v`, выполнив `v ** 2`. Таким образом мы уменьшим `v[1]` с 0,30 до 0,09. К сожалению, в результате и `v[0]` уменьшится с 0,70 до 0,49, то есть опустится ниже исходного среднего значения вектора. Компенсировать это падение можно делением возведенного в квадрат вектора на его сумму, чтобы получить раздутый вектор `v2`, сумма значений которого равна 1. В результате `v2.mean()` будет равно `v.mean()`. Более того, `v2[0]` больше `v[0]`, а `v2[1]` меньше `v[1]`. Проверим это (листинг 19.27).

Листинг 19.27. Подчеркивание разницы значений с помощью раздувания вектора

```
v = np.array([0.7, 0.3])
v2 = v ** 2
v2 /= v2.sum()
assert v.mean() == round(v2.mean(), 10)
assert v2[0] > v[0]
assert v2[1] < v[1]
```

Как и вектор `v`, столбцы нашей матрицы потоков представляют векторы, элементы которых суммируются в 1. Можно раздуть каждый столбец, возведя в квадрат его

элементы, а затем разделив их на сумму следующего столбца. Для этой цели мы определим функцию `inflate`. С ее помощью раздуем матрицу потоков и повторно выполним `compare_flow_distributions()`, чтобы проверить, уменьшился ли порог для потоков между округов (листинг 19.28; рис. 19.9).

Листинг 19.28. Подчеркивание разницы между потоками с помощью раздувания вектора

```
def inflate(matrix):
    matrix = matrix ** 2
    return matrix / matrix.sum(axis=0)

flow_matrix = inflate(flow_matrix)
compare_flow_distributions()

Mean flow within a county: 0.146
Mean flow between different counties: 0.020
The minimum intra-county flow is approximately 0.012
118 inter-county flows fall below that threshold.
```

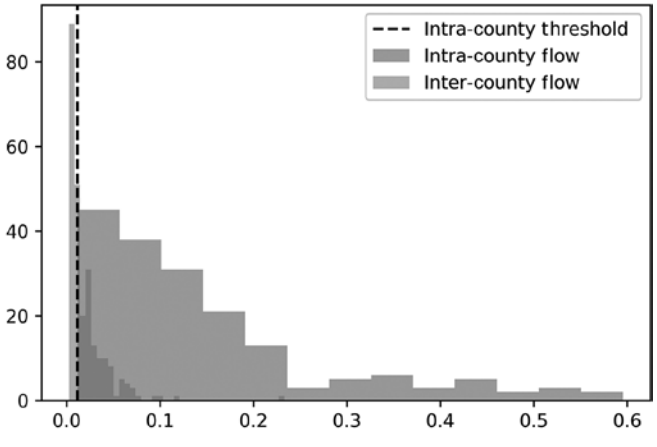


Рис. 19.9. Две гистограммы, представляющие ненулевые потоки между округами и внутри них после раздувания. Разделение между этими видами потоков стало очевидным — порог уменьшился с 0,042 до 0,012

После раздувания порог уменьшился с 0,042 до 0,012, но все еще остается выше 0,01. Как нам еще выделить разницу между ребрами внутри округов и между ними? Ответ на удивление прост, хоть его логика и не слишком очевидна, — нам нужно лишь получить произведение `flow_matrix` с самой собой и раздуть полученный результат. Иными словами, установка матрицы потоков равной `inflate (flow_matrix @ flow_matrix)` вызовет значительное уменьшение порога. Проверим это утверждение и потом разберем доводы в пользу этого уменьшения (листинг 19.29; рис. 19.10).

Листинг 19.29. Раздувание произведения flow_matrix с самой собой

```
flow_matrix = inflate(flow_matrix @ flow_matrix)
compare_flow_distributions()

Mean flow within a county: 0.159
Mean flow between different counties: 0.004
The minimum intra-county flow is approximately 0.001
541 inter-county flows fall below that threshold.
```

До этого шага flow_matrix равнялась inflate(transition_matrix @ transition_matrix). По сути, мы повторяем матричное умножение, дополняя его раздуванием

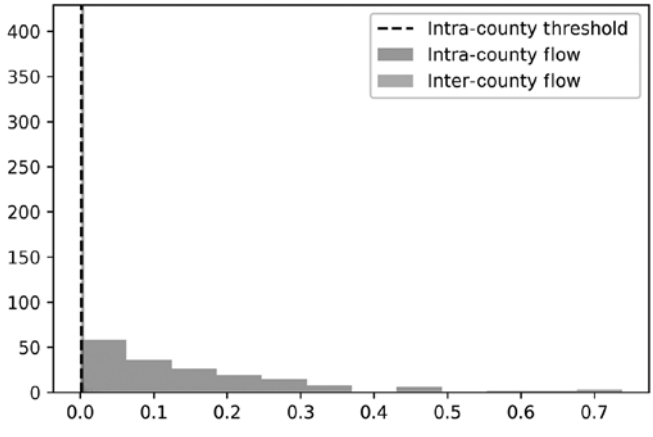


Рис. 19.10. Две гистограммы, представляющие все ненулевые потоки между округами после раздувания flow_matrix @ flow_matrix. Теперь большинство потоков между городами оказались ниже очень малого порога разделения, равного 0,001

Порог уменьшился до 0,001. Под него подпадает более 500 дорог между округами. Почему этот подход оказался успешным? Ответить на этот вопрос можно простой аналогией. Предположим, что мы можем прокладывать новые дороги между городами, но все они требуют ежегодного обслуживания. Недостаточно хорошее обслуживание приведет к появлению на дороге выбоин и трещин. Водители неохотно выбирают маршрут по поврежденной дороге, поэтому периодический ремонт очень важен. Однако в условиях нашей аналогии недостаточно средств, чтобы строить новые дороги и при этом ремонтировать все уже существующие в `G`. Перед местной транспортной службой поставлена сложная задача, требующая решить:

- где строить новые дороги;
- какие из существующих ремонтировать;
- какие из существующих игнорировать.

Специалисты этой службы делают следующее предположение: пары городов с высоким трафиком требуют более качественной транспортной инфраструктуры.

Таким образом, дорога между городами *i* и *j* будет обслуживаться только при высоком значении `flow_matrix[i][j]` или `flow_matrix[j][i]`. Если же значение `flow_matrix[i][j]` высоко, но дороги между *i* и *j* нет, тогда ресурсы будут выделены на соединение этих городов прямой дорогой.

ПРИМЕЧАНИЕ

Пара не соседних городов по-прежнему будет иметь высокий поток, если между ними существует несколько коротких объездов. Прокладывание дороги между этой парой городов имеет смысл, поскольку это уменьшит поток движения по объездным маршрутам.

К сожалению, не все существующие дороги будут обслуживаться. Менее популярные пути между округами имеют меньшие потоки, и дорожная служба их проигнорирует. Следовательно, эти дороги частично разрушатся и водители будут ездить между округами менее охотно, предпочитая более ухоженные маршруты внутри округов и новые дороги между городами.

ПРИМЕЧАНИЕ

Напомню, мы предполагаем, что водители путешествуют случайным образом, не имея конкретного места назначения. Направление их бесцельного перемещения определяется исключительно качеством дорог.

Возможность прокладывания, обслуживания и разрушения дорог неизбежно изменит нашу матрицу переходов. Вероятности переходов между разрушающимися дорогами с низкими потоками уменьшатся. При этом вероятности переходов между обслуженными дорогами с высокими потоками, напротив, увеличатся. Нам нужно смоделировать это изменение матрицы так, чтобы ее столбцы по-прежнему суммировались в 1. Как это сделать? Конечно же, с помощью раздувания. Функция `inflation` выделяет разницу между значениями матрицы, сохраняя сумму элементов столбцов равной 1. Значит, мы смоделируем последствия решений дорожной службы, установив матрицу переходов `M` равной `inflation(flow_matrix)`.

Но это еще не все. Изменяя матрицу переходов, мы также изменяем поток внутри графа. Поток равен `M @ M`, где `M` — матрица потоков после раздувания. Естественно, это изменение повлияет на выделение местных ресурсов — после очередного этапа прокладывания и разрушения дороги вероятности переходов станут равны `inflate(M @ M)`. Влияние таких повторяющихся дорожных работ можно смоделировать как `M = inflate(M @ M)`. Заметьте, что в текущей версии кода `M` установлена как `flow_matrix`. Таким образом, выполнение `flow_matrix = inflate(flow_matrix @ flow_matrix)` приведет к улучшению популярных дорог, несмотря на то что при этом менее популярные разрушатся (рис. 19.11).

Этот итеративный цикл обратной связи имеет неожиданные последствия. С каждым годом дороги между округами будут становиться все хуже и хуже. В результате все больше водителей будут оставаться в границах своих округов и все больше

ресурсов станет выделяться на дороги внутри округов в ущерб дорогам между ними. В итоге они просто разрушатся и путешествовать из округа в округ уже не получится. Каждый отдельный округ станет подобен изолированному острову, отрезанному от соседей. Изоляция характеризует ужасную дорожную политику, но зато сильно облегчает обнаружение сообществ. Изолированный кластер городов легко выявить, поскольку он не граничит с другими кластерами. Получается, что наша модель возведения и разрушения дорог выступает основой для алгоритма сетевой кластеризации — *марковского алгоритма кластеризации* (MCL), называемого также *марковской кластеризацией*.

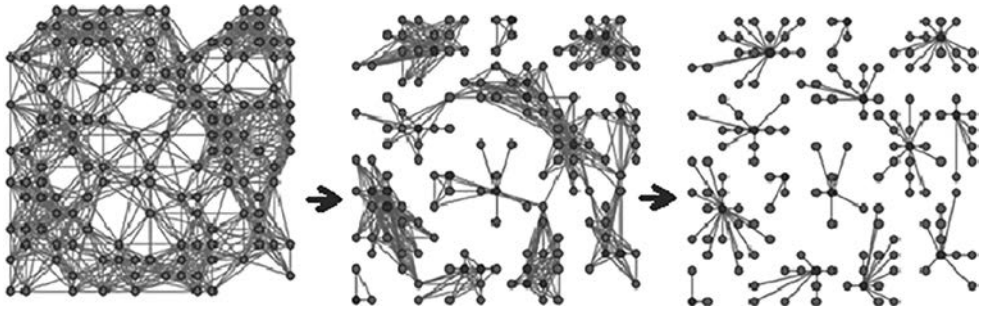


Рис. 19.11. Моделирование изменений в графе дорог при помощи раздувания. Дороги между тесно связанными городами улучшаются. Ресурсы для этого перенаправляются с менее популярных дорог, в результате чего те разрушаются. В итоге останутся лишь дороги внутри сообществ графа

MCL реализуется выполнением `inflate(flow_matrix @ flow_matrix)` в течение многих итераций. С каждой итерацией потоки между округами становятся все меньше и в конечном итоге падают до нуля. При этом потоки внутри округов свои положительные значения сохраняют. Эта двоичная разница позволяет нам определить тесно связанные кластеры округов. Код листинга 19.30 реализует MCL путем выполнения `flow_matrix = inflate (flow_matrix @ flow_matrix)` в течение 20 итераций.

Листинг 19.30. Повторяющееся раздувание произведения flow_matrix с самой собой

```
for _ in range(20):
    flow_matrix = inflate(flow_matrix @ flow_matrix)
```

В соответствии со сказанным теперь определенные ребра в графе `G` должны иметь нулевой поток. Мы ожидаем, что эти ребра соединяют разные округа. Давайте изолируем предполагаемые ребра между округами (листинг 19.31). Мы переберем каждое ребро `(i, j)`, вызвав метод `G.edges()`. Затем отследим каждое ребро `(i, j)`, для которого этого потока не существует, и упорядочим все полученные ребра в списке `suspected_inter_county`.

Листинг 19.31. Выбор предполагаемых ребер между округами

```
suspected_inter_county = [(i, j) for (i, j) in G.edges()
                          if not (flow_matrix[i][j] or flow_matrix[j][i])]
num_suspected = len(suspected_inter_county)
print(f"We suspect {num_suspected} edges of appearing between counties.")

We suspect 57 edges of appearing between counties.
```

Поток отсутствует в 57 ребрах. Мы предполагаем, что они соединяют города, расположенные в разных округах. Удаление этих предполагаемых ребер из графа должно разорвать все связи между округами, в результате чего последующая визуализация графа должна содержать только кластеризованные округа. Проверим это, удалив предполагаемые ребра из копии нашего графа (листинг 19.32; рис. 19.12). Мы используем NetworkX-метод `remove_edge_from` для удаления всех ребер в списке `suspected_inter_county`.

Листинг 19.32. Удаление предполагаемых ребер между округами

```
np.random.seed(1)
G_copy = G.copy()
G_copy.remove_edges_from(suspected_inter_county)
nx.draw(G_copy, with_labels=True, node_color=node_colors)
plt.show()
```

Выполнение G.copy() возвращает скопированную версию графа G. Мы можем удалить ребра в его копии, сохранив их в оригинале

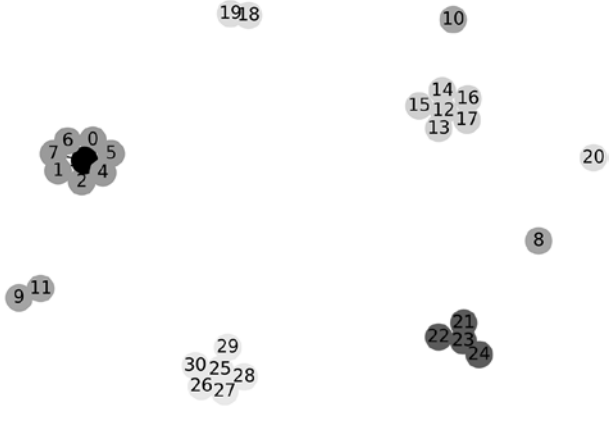


Рис. 19.12. Сеть городов после удаления всех предполагаемых ребер между округами. Все округа оказались изолированы друг от друга. Четыре из шести округов сохранились полностью, а два оказались лишены полноценной связи

Все ребра между округами были удалены. К сожалению, несколько ключевых ребер внутри округов также подпали под удаление. Города 8, 10 и 20 больше не связаны ни с одним прочим городом. Наш алгоритм сработал слишком агрессивно. Почему? Проблема кроется в небольшой ошибке модели: она предполагает, что

путешественники могут поехать в соседние города, но не позволяет им оставаться в текущей локации, что и вызывает неожиданные последствия.

Проиллюстрируем это на простой сети из двух узлов. Представьте, что города A и B соединяет одна дорога. В нашей модели водитель из города A не имеет иного выбора, кроме как поехать в город B. Оставаться в каком-либо городе ему не позволено, поэтому он вынужден из города B ехать обратно в A. Между этими городами не существует маршрута с двумя остановками, следовательно, поток между ними окажется равен нулю и связывающая их дорога будет удалена. Естественно, это нелепая ситуация — нам нужно предоставить водителю возможность оставаться в городе B. Как это сделать? Один из вариантов — добавить ребро, ведущее от города B к нему самому. Оно будет подобно кольцевой дороге, которая возвращает тебя в стартовое местоположение (рис. 19.13). Иными словами, это ребро является петлей. Добавление петель в граф ограничит неожиданное поведение модели. Код листинга 19.33 показывает влияние петель на простую матрицу смежности для двух узлов.

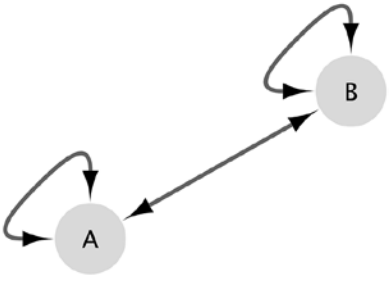


Рис. 19.13. Граф, показывающий возможные маршруты между городами A и B. Петли у каждого узла позволяют водителю оставаться в одном городе, а не ехать в соседний. Без них он будет вынужден безостановочно путешествовать из A в B и обратно. Если такое случится, поток между этими городами окажется равен нулю

Листинг 19.33. Стимуляция потока за счет добавления петель

```
def compute_flow(adjacency_matrix):
    transaction_matrix = adjacency_matrix / adjacency_matrix.sum(axis=0)
    return (transaction_matrix @ transaction_matrix)[1][0]

M1 = np.array([[0, 1], [1, 0]])
M2 = np.array([[1, 1], [1, 1]])

flow1, flow2 = [compute_flow(M) for M in [M1, M2]]
print(f"The flow from A to B without self-loops is {flow1}")
print(f"The flow from A to B with self-loops is {flow2}")

The flow from A to B without self-loops is 0.0
The flow from A to B with self-loops is 0.5
```

Добавление петель в граф `G` должно ограничить недопустимое удаление ребер. Добавить их можно выполнением `G.add_edge(i, i)` для каждого `i` в `G.nodes`. С учетом этого теперь определим функцию `run_mcl`, которая будет выполнять MCL для входного графа по такому алгоритму.

1. Добавлять петли в каждый узел графа.
2. Вычислять матрицы переходов графа путем деления матрицы смежности на суммы элементов в ее столбцах.
3. Вычислять матрицы потоков из `transition_matrix @ transition_matrix`.
4. Устанавливать `flow_matrix` равной `inflate(flow_matrix @ flow_matrix)` в течение 20 итераций.
5. Удалять из графа все ребра, не имеющие потока.

После определения `run_mcl` мы выполним эту функцию для копии графа `G` (листинг 19.34). Визуализированный результат должен сохранить все релевантные ребра внутри округов, удалив все ребра между ними (рис. 19.14).

Листинг 19.34. Определение функции MCL

```
def run_mcl(G):
    for i in G.nodes:
        G.add_edge(i, i)

    adjacency_matrix = nx.to_numpy_array(G)
    transition_matrix = adjacency_matrix / adjacency_matrix.sum(axis=0)
    flow_matrix = inflate(transition_matrix @ transition_matrix)

    for _ in range(20):
        flow_matrix = inflate(flow_matrix @ flow_matrix)

    G.remove_edges_from([(i, j) for i, j in G.edges()
                         if not (flow_matrix[i][j] or flow_matrix[j][i])])

G_copy = G.copy()
run_mcl(G_copy)
nx.draw(G_copy, with_labels=True, node_color=node_colors)
plt.show()
```

Добавляет петли в каждый узел графа

Наш граф идеально кластеризовался на шесть отдельных округов. Города, расположенные в каждом округе, доступны друг для друга, но от внешнего мира изолированы. В теории графов подобные изолированные графы называются *компонентами связности* — два узла находятся в одной компоненте связности, если между ними существует путь. В противном случае узлы существуют в разных компонентах, то есть в разных сообществах. Чтобы вычислить полную компоненту узла, достаточно выполнить для него `nx.shortest_path_length`. Алгоритм нахождения кратчайшего пути возвращает только те узлы, которые доступны внутри кластеризованного сообщества. Код листинга 19.35 задействует `nx.shortest_path_length` для вычисления

всех городов, которые остаются доступными из города 0, и подтверждает, что все они имеют один ID округа.

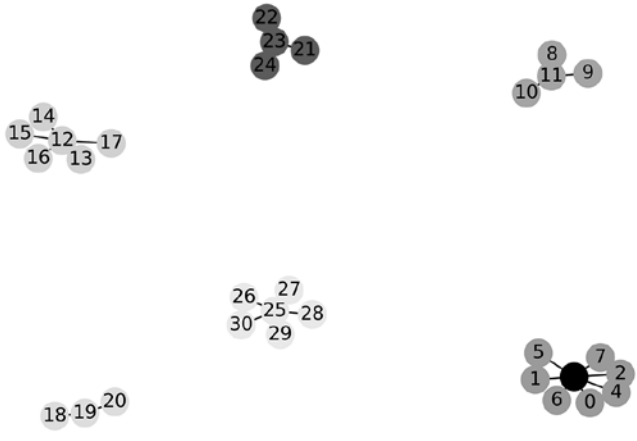


Рис. 19.14. Сеть городов после удаления ребер между округами с помощью MCL. Все округа оказались изолированы друг от друга. Внутренние же соединения во всех округах сохранились

Листинг 19.35. Использование длин путей для обнаружения кластера округа

```
component = nx.shortest_path_length(G_copy, source=0).keys()
county_id = G.nodes[0]['county_id']
for i in component:
    assert G.nodes[i]['county_id'] == county_id

print(f"The following towns are found in County {county_id}:")
print(sorted(component))
```

```
The following towns are found in County 0:
[0, 1, 2, 3, 4, 5, 6, 7]
```

Небольшим изменением алгоритма поиска кратчайшего пути можно извлечь компоненты связности графа. Здесь мы не станем разбирать эти изменения, но вам будет полезно поработать с ними самостоятельно. Измененный алгоритм поиска компонент содержится в NetworkX — вызов `nx.connected_components(G)` вернет результат итерации по всем компонентам связности в `G`. Каждая компонента связности сохраняется как набор ID узлов. В листинге 19.36 мы используем эту функцию для вывода всех кластеров округов.

Листинг 19.36. Извлечение всех кластеризованных компонент связности

```
for component in nx.connected_components(G_copy):
    county_id = G.nodes[list(component)[0]]['county_id']
    print(f"\nThe following towns are found in County {county_id}:")
    print(component)
```

```
The following towns are found in County 0:
{0, 1, 2, 3, 4, 5, 6, 7}

The following towns are found in County 1:
{8, 9, 10, 11}

The following towns are found in County 2:
{12, 13, 14, 15, 16, 17}

The following towns are found in County 3:
{18, 19, 20}

The following towns are found in County 4:
{24, 21, 22, 23}

The following towns are found in County 5:
{25, 26, 27, 28, 29, 30}
```

ТИПИЧНЫЕ ВЫЧИСЛЕНИЯ ДЛЯ МАТРИЦЫ СЕТИ

- `adjaceny_matrix = nx.to_numpy_array(G)` — возвращает матрицу смежности графа.
- `degrees = adjaceny_matrix.sum(axis=0)` — вычисляет вектор степеней, используя матрицу смежности.
- `transition_matrix = adjacency_matrix / degrees` — вычисляет матрицу переходов графа.
- `stop_1_probabilities = transition_matrix @ v` — вычисляет вероятности совершить первую остановку в каждом узле. Здесь предполагается, что `v` — это вектор равноценных стартовых вероятностей.
- `stop_2_probabilities = transition_matrix @ stop_1_probabilities` — вычисляет вероятности сделать вторую остановку в каждом узле.
- `transition_matrix @ stop_n_probabilities` — возвращает вероятности совершения $N + 1$ остановок в каждом узле.
- `flow_matrix = transition_matrix @ transition_matrix` — вычисляет матрицу вероятностей перехода между `i` и `j` в два шага.
- `(flow_matrix ** 2) / (flow_matrix ** 2).sum(axis=0)` — раздувает потоки в матрице потоков.

Мы успешно обнаружили сообщества в нашем графе с помощью небольшого объема кода. К сожалению, данная реализация MCL не будет масштабироваться на очень большие сети. Для этого потребуются дополнительные оптимизации. Они были интегрированы во внешнюю библиотеку марковской кластеризации. Далее

мы установим эту библиотеку и импортируем из модуля `markov_clustering` две функции: `get_clusters` и `run_mcl` (листинг 19.37).

ПРИМЕЧАНИЕ

Для установки библиотеки марковской кластеризации выполните из терминала pip install markov_clustering.

Листинг 19.37. Импорт из библиотеки марковской кластеризации

```
from markov_clustering import get_clusters, run_mcl
```

Имея матрицу смежности `M`, можно применить марковскую кластеризацию, выполнив `get_clusters(run_mcl(M))`. Этот вызов вложенной функции вернет список `clusters`. Каждый элемент в `clusters` представляет кортеж узлов, формирующих кластеризованное сообщество. Давайте выполним эту кластеризацию для исходного графа `G` (листинг 19.38). Полученные кластеры должны сохранять согласованность с компонентами связности в `G_copy`.

Листинг 19.38. Получение кластеров с помощью библиотеки марковской кластеризации

```
adjacency_matrix = nx.to_numpy_array(G)
clusters = get_clusters(run_mcl(adjacency_matrix))

for cluster in clusters:
    county_id = G.nodes[cluster[0]]['county_id']
    print(f"\nThe following towns are found in County {county_id}:")
    print(cluster)

The following towns are found in County 0:
(0, 1, 2, 3, 4, 5, 6, 7)

The following towns are found in County 1:
(8, 9, 10, 11)

The following towns are found in County 2:
(12, 13, 14, 15, 16, 17)

The following towns are found in County 3:
(18, 19, 20)

The following towns are found in County 4:
(21, 22, 23, 24)

The following towns are found in County 5:
(25, 26, 27, 28, 29, 30)
```

С помощью марковской кластеризации можно обнаружить сообщества в структурированных на сообщества графах. Эта возможность пригодится, когда мы будем искать группы друзей в социальных сетях.

19.4. ОБНАРУЖЕНИЕ ГРУПП ДРУЗЕЙ В СОЦИАЛЬНЫХ СЕТЯХ

В виде сетей можно представлять многие процессы, включая отношения между людьми. В этих *социальных сетях* узлы представляют отдельных людей. Если два человека каким-то образом социально взаимодействуют, между ними имеется ребро. К примеру, можно связать таким образом двух людей, если они дружат.

Существует множество типов социальных сетей. Некоторые из них являются цифровыми. Например, сервис FriendHook построен вокруг онлайн-связей. Однако социальные сети изучались не одно десятилетие, прежде чем возникли социальные медиаресурсы. Одна из наиболее изученных сетей под названием *«Клуб карате Закари»* возникла в 1970-е годы. Основывалась она на социальной структуре университетского клуба карате, задокументированной ученым Уэйном Закари. В течение трех лет он отслеживал дружбу между 34 членами этого клуба. Ребрами обозначались связи тех участников клуба, которые взаимодействовали за его пределами. Однако спустя три года произошло неожиданное событие: инструктор мистер Хи покинул клуб, чтобы открыть собственный, утянув за собой половину его членов. К большому удивлению Закари, большинство ушедших спортсменов можно было определить исключительно по структуре сети.

Мы же далее повторим эксперимент Закари. Для начала нужно будет скачать известную сеть клуба карате, доступную в NetworkX. Вызов `nx.karate_club_graph()` вернет этот граф. Код листинга 19.39 выводит узлы графа вместе с их атрибутами. Напомню, что выводить узлы с атрибутами можно вызовом `G.nodes(data=True)`.

Листинг 19.39. Загрузка графа клуба карате

```
G_karate = nx.karate_club_graph()
print(G_karate.nodes(data=True))
```

```
[(0, {'club': 'Mr. Hi'}), (1, {'club': 'Mr. Hi'}), (2, {'club': 'Mr. Hi'}),
(3, {'club': 'Mr. Hi'}), (4, {'club': 'Mr. Hi'}), (5, {'club': 'Mr. Hi'}),
(6, {'club': 'Mr. Hi'}), (7, {'club': 'Mr. Hi'}), (8, {'club': 'Mr. Hi'}),
(9, {'club': 'Officer'}), (10, {'club': 'Mr. Hi'}), (11, {'club':
'Mr. Hi'}), (12, {'club': 'Mr. Hi'}), (13, {'club': 'Mr. Hi'}), (14,
{'club': 'Officer'}), (15, {'club': 'Officer'}), (16, {'club': 'Mr. Hi'}),
(17, {'club': 'Mr. Hi'}), (18, {'club': 'Officer'}), (19, {'club':
'Mr. Hi'}), (20, {'club': 'Officer'}), (21, {'club': 'Mr. Hi'}), (22,
{'club': 'Officer'}), (23, {'club': 'Officer'}), (24, {'club':
'Officer'}), (25, {'club': 'Officer'}), (26, {'club': 'Officer'}), (27,
{'club': 'Officer'}), (28, {'club': 'Officer'}), (29, {'club': 'Officer'}),
(30, {'club': 'Officer'}), (31, {'club': 'Officer'}), (32, {'club':
'Officer'}), (33, {'club': 'Officer'})]
```

Нашим узлам соответствуют 34 человека. У каждого узла есть атрибут `club`, который устанавливается равным `Mr. Hi`, если человек перешел в открытый мистером Хи клуб, и равным `Officer` — в противном случае. Давайте визуализируем эту сеть, закрасив каждый узел на основе установленного атрибута `club` (листинг 19.40; рис. 19.15).

Листинг 19.40. Визуализация графа клуба карате

```
np.random.seed(2)
club_to_color = {'Mr. Hi': 'k', 'Officer': 'b'}

node_colors = [club_to_color[G_karate.nodes[i]['club']]
               for i in G_karate]

nx.draw(G_karate, node_color=node_colors)
plt.show()
```

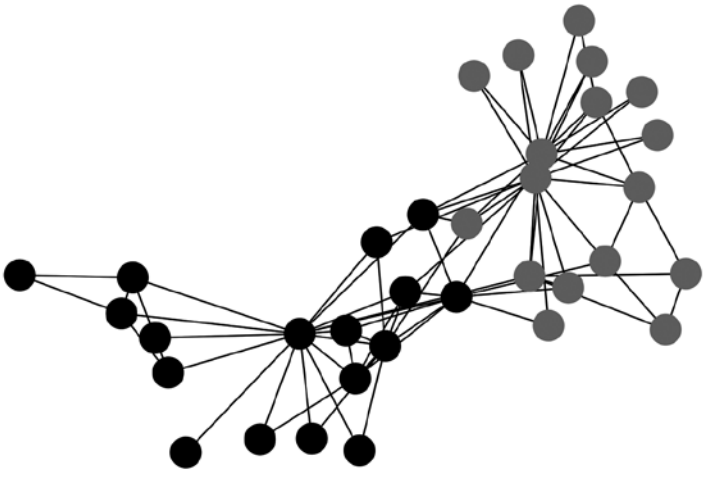


Рис. 19.15. Визуализированный граф клуба карате. Цвета узлов отражают разделение клуба и совпадают со структурой сообществ графа

Представленный граф имеет явно выраженную структуру сообществ, и это неудивительно — многие социальные сети содержат выраженные сообщества. В данном случае сообщества соответствуют разделению клуба: закрашенный черным кластер слева представляет членов клуба, ушедших с мистером Хи, а правый кластер — тех, кто решил остаться. Эти кластеры отражают группы друзей, которые сформировались в течение нескольких лет. Когда произошел раскол, большинство членов просто ушли вместе с близкой им группой друзей.

А получится ли извлечь кластеры друзей автоматически? Можно попробовать реализовать это с помощью MCL. Сначала мы выполним этот алгоритм для матрицы смежности графа и выведем все полученные кластеры (листинг 19.41).

Листинг 19.41. Кластеризация графа клуба карате

```
adjacency_matrix = nx.to_numpy_array(G_karate)
clusters = get_clusters(run_mcl(adjacency_matrix))
for i, cluster in enumerate(clusters):
    print(f"Cluster {i}:\n{cluster}\n")
```

```
Cluster 0:
(0, 1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 16, 17, 19, 21)

Cluster 1:
(2, 8, 9, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33)
```

Как и ожидалось, сформировались два кластера. Теперь мы еще раз построим граф, закрасив каждый узел на основе ID его кластера (листинг 19.42; рис. 19.16).

Листинг 19.42. Закрашивание узлов построенного графа на основе их принадлежности к кластерам

```
np.random.seed(2)
cluster_0, cluster_1 = clusters
node_colors = ['k' if i in cluster_0 else 'b'
               for i in G_karate.nodes]

nx.draw(G_karate, node_color=node_colors)
plt.show()
```

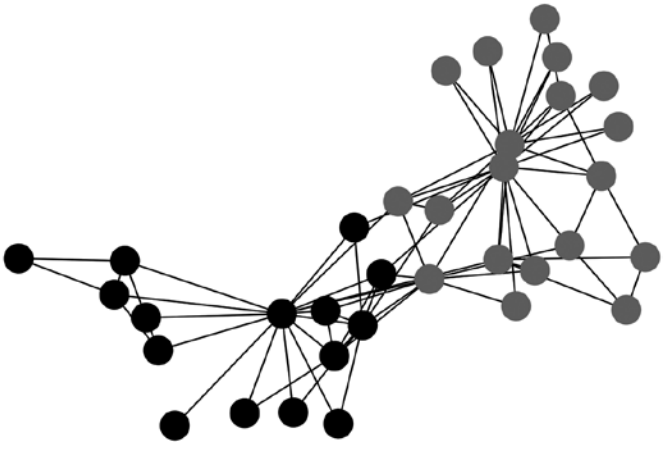


Рис. 19.16. Визуализированный граф клуба карате. Цвета узлов соответствуют кластерам сообщества и совпадают с итоговым разделением клуба

Полученные кластеры почти идентичны двум сформировавшимся клубам. MCL успешно извлек группы друзей в социальной сети, значит, этот алгоритм явно пригодится нам при решении практического задания, в котором требуется проанализировать цифровую социальную сеть. Извлечение существующих групп друзей может оказаться при этом незаменимым. Естественно, в крупных сетях количество групп будет больше двух — можно ожидать встретить десяток или даже несколько десятков кластеров друзей. Нам также наверняка понадобится визуализировать эти кластеры в графе. Ручное присваивание цветов десяткам кластеров окажется утомительной задачей, поэтому нужно научиться генерировать цвета кластеров автоматически. В NetworkX это можно реализовать так.

1. Сопоставить каждый узел и ID его кластер, добавив всем узлам атрибут `cluster_id`.
2. Установить каждый элемент `node_colors` равным ID кластера, а не цвету. Для этого нужно выполнить `[G.nodes[n]['cluster_id'] for n in G.nodes]`, где `G` представляет кластеризованный социальный граф.
3. Передать `cmap=plt.cm.tab20` в `nx.draw` вместе с численным списком `node_colors`. Параметр `cmap` присваивает ID каждого кластера цветовое сопоставление. `plt.cm.tab20` представляет цветовую палитру, используемую для генерации этого сопоставления. Ранее мы уже применяли сопоставление палитры для генерации тепловых карт (см. главу 8).

Далее проделаем эти шаги для автоматического раскрашивания кластеров (листинг 19.43; рис. 19.17).

Листинг 19.43. Автоматическое раскрашивание кластеров социального графа

```
np.random.seed(2)
for cluster_id, node_indices in enumerate(clusters):
    for i in node_indices:
        G_karate.nodes[i]['cluster_id'] = cluster_id

node_colors = [G_karate.nodes[n]['cluster_id'] for n in G_karate.nodes]
nx.draw(G_karate, node_color=node_colors, cmap=plt.cm.tab20)
plt.show()
```

Присваивает каждому узлу ID кластера

Сопоставляет цвета узлов с численными ID кластеров

Использует цветовую палитру plt.cm.tab20 для присваивания цвета в соответствии с ID каждого кластера

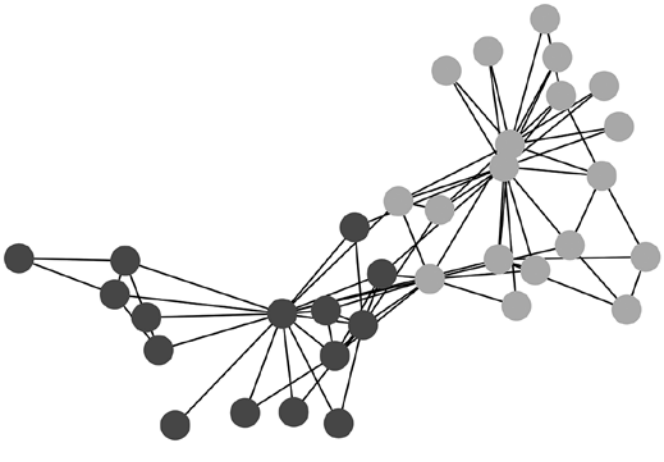


Рис. 19.17. Визуализированный граф клуба карате. Цвета узлов соответствуют кластерам сообществ и были сгенерированы автоматически

На этом мы закончили углубленное изучение теории графов. В следующей главе используем полученные знания для построения простого алгоритма прогнозирования на основе графов.

РЕЗЮМЕ

- Количество ребер узла в ненаправленном графе называется *степенью* узла. Вычислить степень каждого узла можно сложением элементов столбцов в матрице смежности графа.
- В теории графов любая мера важности узла обычно называется *центральностью узла*. Важность, ранжированная на основе степени узла, называется *степенью центральности*.
- Иногда степень центральности оказывается неподходящей мерой значимости узла. В таком случае более удачно вывести центральность позволит симуляция случайного трафика в сети. Этот трафик можно преобразовать в вероятность случайно оказаться в конкретном узле.
- Вероятность трафика можно вычислить непосредственно из *матрицы переходов* графа. Эта матрица отслеживает вероятность случайно переместиться из узла `i` в узел `j`. Итеративное умножение матрицы переходов на вектор вероятности дает вектор вероятностей оказаться в той или иной конечной точке. Более высокие вероятности соответствуют более центральным узлам. Эта мера центральности известна как *центральность PageRank*. В математическом смысле она соответствует собственному вектору матрицы переходов.
- Определенные графы при визуализации демонстрируют тесно связанные кластеры узлов, которые называются *сообществами*. Про графы с отчетливо видимыми сообществами говорят, что они содержат *структуру сообществ*. Процесс выявления сообществ в графах называется *обнаружением сообществ*.
- Обнаруживать сообщества можно с помощью *марковского алгоритма кластеризации* (MCL). Он требует вычисления *случайного потока*, представляющего вероятность перехода с несколькими остановками. Получение произведения матрицы переходов с самой собой дает матрицу потоков. Более низкие значения с большей вероятностью соответствуют ребрам между сообществами. Различия между низкими и высокими значениями потоков можно дополнительно усилить с помощью *раздувания*. Итеративное повторение матричного умножения и раздувания ведет к падению значений потоков между сообществами до нуля. Последующее удаление ребер с нулевым потоком полностью изолирует сообщества графа друг от друга. Такие изолированные компоненты можно определить с помощью одной из вариаций алгоритма нахождения кратчайшего пути.
- В *социальных сетях* ребра представляют связи между людьми. Социальные сети обычно содержат структуру сообществ, что говорит о возможности использовать MCL для обнаружения в этих сетях кластеров друзей.

20 Машинное обучение с учителем на основе сетей

В этой главе

- ✓ Использование классификаторов в машинном обучении с учителем.
- ✓ Простое прогнозирование на основе сходства.
- ✓ Параметры для оценки качества прогнозов.
- ✓ Распространенные методы обучения с учителем в scikit-learn.

Люди могут учиться, наблюдая реальный мир, и в некотором смысле машины тоже на это способны. Обучение компьютеров метафорически понимать мир через контролируемый опыт называется *машинным обучением с учителем*. В последние годы такой вид машинного обучения очень активно освещался в новостях — компьютеры научились прогнозировать цены на бирже, диагностировать заболевания и даже водить машины. Эти достижения по праву были признаны передовыми инновациями, хотя отчасти алгоритмы, стоящие за этими инновациями, не так уж новы. Разновидности существующих техник машинного обучения существуют уже несколько десятилетий, но из-за ограниченности вычислительных возможностей применить их эффективно не получалось. Только в последнее время вычислительная мощь компьютеров достигла нужного уровня, и идеи, выработанные много лет назад, наконец начинают давать плоды в виде значительных технологических достижений.

В данной главе мы изучим одну из самых давних и простых техник машинного обучения с учителем. Алгоритм, называемый *методом K ближайших соседей*, был

разработан управлением Военно-воздушными силами США в 1951 году. Он опирается на теорию сетей и корнями уходит в открытия средневекового ученого Альхазена. Несмотря на возраст алгоритма, его использование имеет много общего с более современными техниками. Разобравшись в нем, мы сможем перенести его на более обширное поле машинного обучения с учителем.

20.1. ОСНОВЫ МАШИННОГО ОБУЧЕНИЯ С УЧИТЕЛЕМ

Обучение с учителем используется для автоматизации определенных задач, которые в противном случае выполняет человек. Машина наблюдает, как он выполняет задачу, после чего учится это поведение повторять. Мы проиллюстрируем это с помощью набора данных о цветах, с которым познакомились в главе 14. Напомню, что он представляет три разных вида ириса, которые показаны на рис. 20.1. Визуально они похожи, и ботаники для их распознавания ориентируются на тонкие различия в длине и ширине листьев. Обладание таким видом экспертных знаний требует обучения — без него ни человек, ни машина эти виды цветов различить не смогут.

Рис. 20.1. Три вида ириса: щетинистый, разноцветный и виргинский. Все они похожи друг на друга. Для их распознавания можно использовать тонкие различия в размерах листьев, но для этого необходимо учиться

Предположим, что профессор ботаники проводит экологический анализ местного луга. На нем растут сотни растений ириса, и профессор хочет узнать распределение видов среди них. Однако ученый занят написанием грантов, и у него нет времени, чтобы оценить их все самостоятельно. Поэтому он нанимает ассистента, которому и поручает эту задачу. К сожалению, ассистент не ботаник и ему не хватает навыков для различения видов. Он решает тщательно измерить длину и ширину листьев каждого цветка. Можно ли использовать эти измерения для автоматического определения видов? Этот вопрос лежит в сердце обучения с учителем.

По сути, мы хотим построить модель, сопоставляющую входные измерения с одной из трех категорий видов. В машинном обучении эти входные измерения называются *признаками*, а выходные категории — *классами*. Цель обучения с учителем

заключается в том, чтобы построить модель, способную определять классы на основе признаков. Называется такая модель *классификатором*.

ПРИМЕЧАНИЕ

По определению классы являются дискретными категориальными переменными, такими как виды цветов или типы машины. В качестве альтернативы эти модели называют также регрессорами, которые прогнозируют численные переменные, например стоимость дома или скорость автомобиля.

В машинном обучении существует много видов классификаторов, разделению которых на категории посвящены целые книги. Но, несмотря на их разнообразие, построение большинства из них состоит из одних и тех же шагов. Для реализации классификатора нам потребуется проделать следующее.

1. Вычислить признаки для каждой точки данных. В нашем ботаническом примере все точки данных являются цветами, значит, нужно измерить длину листьев каждого из них.
2. Затем эксперт в этой предметной области должен присвоить метки подмножеству точек данных. У нашего ботаника нет выбора, кроме как вручную определить виды в подмножестве цветов. Без надзора профессора правильно построить классификатор не получится. Термин *«обучения с учителем»* отражает контролируемую фазу разметки данных. Для разметки подмножества цветов потребуется время, но потраченные усилия себя оправдают, как только классификатор сможет делать автоматические прогнозы.
3. Показать классификатору комбинацию признаков и размеченные вручную классы. После этого он попытается изучить связь между полученными признаками и классами. Эта фаза обучения в разных классификаторах реализуется по-разному.
4. Показать классификатору набор признаков, которые он до этого не встречал. Затем он попытается спрогнозировать ассоциированные с этими признаками классы на основе значений, полученных из размеченных данных.

Для построения классификатора ботанику нужен набор признаков коллекции опознанных цветов. Каждому цветку присваиваются четыре признака, которые мы рассматривали ранее в главе 14:

- длина цветного лепестка;
- ширина цветного лепестка;
- длина зеленого листа, поддерживающего лепесток;
- ширина зеленого листа, поддерживающего лепесток.

Эти признаки можно сохранить в матрице. Каждый ее столбец будет соответствовать одному из четырех признаков, а каждая строка — размеченному цветку. Метки класса сохраняются в массиве NumPy. Подобные массивы предназначены

для хранения чисел, а не текста. В машинном обучении метки классов выражаются в виде целых чисел, относящихся к диапазону от 0 до $N - 1$, где N — общее число классов. В примере с ирисом мы имеем дело с тремя видами, значит, метки классов представлены значениям от 0 до 2.

Как говорилось в главе 14, мы можем загружать известные признаки ириса и метки классов с помощью функции scikit-learn `load_iris`. Сейчас мы так и сделаем. В соответствии с соглашением scikit-learn матрица признаков обычно присваивается переменной `X`, а массив с метками классов — переменной `y`. Следуя этому соглашению, код листинга 20.1 загружает `X` и `y` ириса, передавая `return_X_y=True` в `load_iris`.

Листинг 20.1. Загрузка признаков ириса и меток классов

```
from sklearn.datasets import load_iris
X, y = load_iris(return_X_y=True)
num_classes = len(set(y))
print(f"We have {y.size} labeled examples across the following "
      f"{num_classes} classes:\n{set(y)}\n")
print(f"First four feature rows:\n{X[:4]}")
print(f"\nFirst four labels:\n{y[:4]}")

We have 150 labeled examples across the following 3 classes:
{0, 1, 2}

First four feature rows:
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]]

First four labels:
[0 0 0 0]
```

Все 150 измерений цветов были размечены как принадлежащие одному из трех видов цветов. Подобная разметка — нелегкий труд. Представьте, что у нашего ботаника есть время на разметку только четвертой части всех цветов. После этого профессор строит классификатор для прогнозирования классов оставшихся. Давайте симулируем этот сценарий, начав с выбора первой четверти точек данных в `X` и `y`. Этот срез данных называется `X_train` и `y_train`, поскольку используется для обучения. Подобные наборы данных называются *обучающими выборками*. После составления обучающей выборки мы изучим содержимое `y_train` (листинг 20.2).

Листинг 20.2. Создание обучающей выборки

```
sampling_size = int(y.size / 4)
X_train, y_train = X[:sampling_size], y[:sampling_size]
print(f"Training set labels:\n{y_train}")

Training set labels:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Наша обучающая выборка содержит только размеченные примеры для вида 0. Прочие виды цветов в ней не представлены. Чтобы расширить это представление, необходимо произвести случайный выбор значений из `X` и `y`. Такую случайную выборку можно составить с помощью функции scikit-learn `train_test_split`, которая получает на входе `X` и `y` и возвращает четыре случайно сгенерированных результата. Первыми двумя результатами являются `X_train` и `y_train`, соответствующие нашему обучающему набору. Оставшиеся два охватывают признаки и классы вне обучающего набора. Эти выводы можно использовать для тестирования классификатора после обучения, в связи с чем их данные обычно называются *тестовой выборкой*. Мы будем называть тестовые признаки и классы `X_test` и `y_test` соответственно. Чуть позже в этой главе мы применяем данную тестовую выборку для оценки обученной модели.

Код листинга 20.3 вызывает функцию `train_test_split` и передает ей опциональный параметр `train_size=0.25`. Параметр `train_size` гарантирует, что в обучающей выборке окажется 25 % всех данных. В завершение листинга мы выводим `y_train`, чтобы убедиться, что все метки представлены подобающим образом.

Листинг 20.3. Создание обучающего набора путем случайной выборки

```
from sklearn.model_selection import train_test_split
import numpy as np
np.random.seed(0)
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.25)
print(f"Training set labels:\n{y_train}")

Training set labels:
[0 2 1 2 1 0 2 0 2 0 0 2 0 2 1 1 1 2 2 1 1 0 1 2 2 0 1 1 1 1 0 0 0 2 1 2 0]
```

В обучающих данных имеются метки всех трех классов. Как теперь использовать `X_train` и `y_train` для прогнозирования классов цветов, оставшихся в тестовой выборке? Один из простых подходов основан на геометрической близости. Как мы видели в главе 14, признаки в наборе данных ирисов можно изобразить графически в многомерном пространстве. Эти визуализированные данные формируют пространственные кластеры: элементы в `X_test` с большей вероятностью будут принадлежать к тому же классу, что и точки `X_train`, находящиеся в смежном кластере.

Проиллюстрируем эту догадку, построив `X_train` и `X_test` в двухмерном пространстве (листинг 20.4; рис. 20.2). Мы применяем анализ главных компонент для уменьшения наших данных до двух измерений, после чего построим график уменьшенных признаков, попутно окрасив каждую точку в соответствии с ее классом. Помимо этого, отобразим на графике элементы тестовой выборки, используя треугольный маркер для обозначения отсутствия метки. Затем угадаем принадлежность неразмеченных точек на основе их близости к размеченным данным.

Листинг 20.4. Графическая визуализация обучающей и тестовой выборок

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

pca_model = PCA()
transformed_data_2D = pca_model.fit_transform(X_train)

unlabeled_data = pca_model.transform(X_test)
plt.scatter(unlabeled_data[:,0], unlabeled_data[:,1],
            color='khaki', marker='^', label='test')

for label in range(3):
    data_subset = transformed_data_2D[y_train == label]
    plt.scatter(data_subset[:,0], data_subset[:,1],
                color=['r', 'k', 'b'][label], label=f'train: {label}')

plt.legend()
plt.show()
```

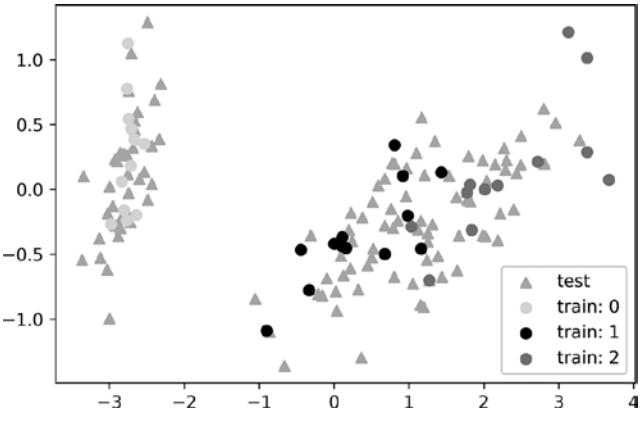


Рис. 20.2. Точки данных цветов, построенные в 2D. Каждый размеченный цветок окрашен на основе класса своего вида. На графике также есть неразмеченные цветы. Их принадлежность можно определить визуально на основе близости к размеченным точкам

В левой части нашего графика множество неразмеченных точек кластеризуются вокруг вида 0. Здесь двусмысленности нет — эти неразмеченные цветы определенно принадлежат к одному виду. В любой другой части графика конкретные неразмеченные цветы приближены и к виду 1, и к виду 2. Для каждой такой точки нужно количественно оценить, какие размеченные виды находятся ближе. Для этого потребуется проследить евклидово расстояние между каждым признаком в `X_test` и `X_train` (листинг 20.5). По сути, нам нужна матрица расстояний `M`, в которой `M[i][j]` равно евклидову расстоянию между `X_test[i]` и `X_test[j]`. Ее можно легко сгенерировать, используя функцию scikit-learn `euclidean_distances`. Нужно

лишь выполнить `euclidean_distances(X_test, X_train)`, чтобы вернуть матрицу расстояний.

Листинг 20.5. Вычисление евклидова расстояния между точками

```
from sklearn.metrics.pairwise import euclidean_distances
distance_matrix = euclidean_distances(X_test, X_train)

f_train, f_test = X_test[0], X[0]
distance = distance_matrix[0][0]
print(f"Our first test set feature is {f_train}")
print(f"Our first training set feature is {f_test}")
print(f"The Euclidean distance between the features is {distance:.2f}")

Our first test set feature is [5.8 2.8 5.1 2.4]
Our first training set feature is [5.1 3.5 1.4 0.2]
The Euclidean distance between the features is 4.18
```

Имея неразмеченные точки в `X_test`, можно присвоить им класс, используя следующую стратегию.

1. Упорядочить все точки данных в обучающей выборке на основе их расстояния до неразмеченных точек.
2. Выбрать старших K ближайших соседей точки. Мы пока что произвольно установим K равным 3.
3. Выбрать наиболее часто встречающиеся классы среди K соседних точек.

По сути, мы предполагаем, что каждая неразмеченная точка относится к классу, являющемуся общим для ее соседей. Эта стратегия формирует основу алгоритма K ближайших соседей (KNN). Далее мы применим ее к случайно выбранной точке (листинг 20.6).

Листинг 20.6. Разметка точки на основе ее ближайших соседей

```
from collections import Counter
np.random.seed(6)
random_index = np.random.randint(y_test.size)
labeled_distances = distance_matrix[random_index]
labeled_neighbors = np.argsort(labeled_distances)[:3]
labels = y_train[labeled_neighbors]

top_label, count = Counter(labels).most_common()[0]
print(f"The 3 nearest neighbors of Point {random_index} have the "
      f"following labels:\n{labels}")
print(f"\nThe most common class label is {top_label}. It occurs {count} "
       "times.")

The 3 nearest neighbors of Point 10 have the following labels: [2 1 2]

The most common class label is 2. It occurs 2 times.
```

Среди соседей точки 10 наиболее распространена метка класса 2. Как это соотносится с фактическим классом цветка (листинг 20.7)?

Листинг 20.7. Проверка истинности класса спрогнозированной метки

```
true_label = y_test[random_index]
print(f"The true class of Point {random_index} is {true_label}.")

The true class of Point 10 is 2.
```

KNN успешно определил класс цветка в точке 10. Для этого потребовалось лишь проверить размеченных соседей и подсчитать наиболее распространенные среди них метки. Интересно, что этот процесс можно переформулировать в виде задачи теории графов. Мы можем рассмотреть каждую точку как узел, а его метку — как атрибут узла, после чего выбрать неразмеченную точку и протянуть ребра до K ее ближайших размеченных соседей. Визуализация графа соседей позволит нам определить принадлежность рассматриваемой точки.

ПРИМЕЧАНИЕ

Этот тип структуры графа называется графом K ближайших соседей (k-NNG). Подобные графы используются в различных областях, включая планирование движения транспорта, сжатие изображений и роботостроение. Кроме того, их можно применять для улучшения алгоритма кластеризации DBSCAN.

Продемонстрируем описанную сетевую форму задачи, построив граф соседей точки 10 (листинг 20.8; рис. 20.3). Для этого задействуем NetworkX.

Листинг 20.8. Визуализация ближайших соседей с помощью NetworkX

```
import networkx as nx
np.random.seed(0)
def generate_neighbor_graph(unlabeled_index, labeled_neighbors):
    G = nx.Graph()
    nodes = [(i, {'label': y_train[i]}) for i in labeled_neighbors]
    nodes.append((unlabeled_index, {'label': 'U'}))
    G.add_nodes_from(nodes)
    G.add_edges_from([(i, unlabeled_index) for i in labeled_neighbors])
    labels = y_train[labeled_neighbors]
    label_colors = ['pink', 'khaki', 'cyan']
    colors = [label_colors[y_train[i]] for i in labeled_neighbors] + ['k']
    labels = {i: G.nodes[i]['label'] for i in G.nodes}
    nx.draw(G, node_color=colors, labels=labels, with_labels=True)
    plt.show()
    return G

G = generate_neighbor_graph(random_index, labeled_neighbors)
```

Строит и возвращает граф NetworkX, содержащий связи между неразмеченной точкой данных и ее ближайшими размеченными соседями

Получает метки ближайших соседей

Окрашивает размеченных соседей на основе их меток

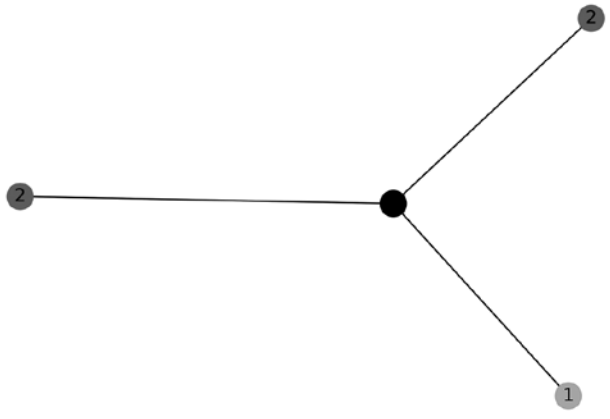


Рис. 20.3. Граф NetworkX, представляющий неразмеченную точку и трех ее ближайших размеченных соседей. Двое из них размечены как принадлежащие к классу 2. Таким образом, можно предположить, что неразмеченная точка также принадлежит к этому доминирующему классу

KNN работает, когда присутствует всего трое соседей. А что произойдет, если увеличить их число до четырех? Сейчас выясним (листинг 20.9; рис. 20.4).

Листинг 20.9. Увеличение числа ближайших соседей

```
np.random.seed(0)
labeled_neighbors = np.argsort(labeled_distances)[:4]
G = generate_neighbor_graph(random_index, labeled_neighbors)
```

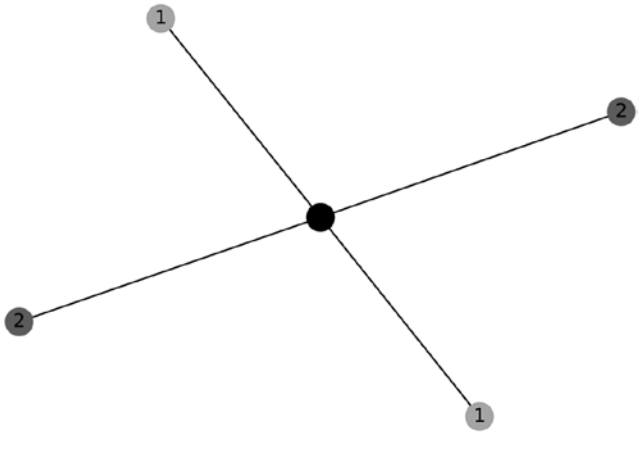


Рис. 20.4. Граф NetworkX, представляющий неразмеченную точку и четырех ее ближайших размеченных соседей. Двое из них относятся к классу 2, а остальные двое — к классу 1. Доминирующего класса нет, значит, определить принадлежность неразмеченной точки не получится

Здесь у нас равенство! Ни одна из меток не доминирует, и решение мы принять не можем. Что же делать? Один из вариантов — случайным образом нарушить это равенство. Но более удачным решением будет учитывать расстояния до размеченных точек. Те из них, что находятся ближе к точке 10, с большей вероятностью будут разделять с ней правильный общий класс. Значит, нужно придать более близким точкам больше веса. Но как?

В первичной реализации KNN голоса всех размеченных точек были равноценными, как при демократии. Теперь же мы хотим взвесить каждый голос на основе расстояния. Для этого есть простая схема, подразумевающая приписывание каждой размеченной точке `1 / distance` голосов. В результате точка, удаленная на одну единицу измерения, получит один голос, удаленная на 0,5 единицы получит два, а та, что удалена на две единицы, — всего половину голоса. Это нельзя назвать честной политикой, но такой подход улучшит результат выполнения алгоритма.

Код листинга 20.10 присваивает каждой размеченной точке количество голосов, равное ее обратному расстоянию от точки 10. Затем мы позволим размеченным точкам проголосовать и на основе подсчета голосов выберем класс для точки 10.

Листинг 20.10. Взвешивание голосов соседей на основе расстояния

```
from collections import defaultdict
class_to_votes = defaultdict(int)

for node in G.neighbors(random_index):
    label = G.nodes[node]['label']
    distance = distance_matrix[random_index][node]
    num_votes = 1 / distance
    print(f"A data point with a label of {label} is {distance:.2f} units "
          f"away. It receives {num_votes:.2f} votes.")
    class_to_votes[label] += num_votes

print()
for class_label, votes in class_to_votes.items():
    print(f"We counted {votes:.2f} votes for class {class_label}.")

top_class = max(class_to_votes.items(), key=lambda x: x[1])[0]
print(f"Class {top_class} has received the plurality of the votes.")
```

```
A data point with a label of 2 is 0.54 units away. It receives 1.86 votes.
A data point with a label of 1 is 0.74 units away. It receives 1.35 votes.
A data point with a label of 2 is 0.77 units away. It receives 1.29 votes.
A data point with a label of 1 is 0.98 units away. It receives 1.02 votes.

We counted 3.15 votes for class 2.
We counted 2.36 votes for class 1.
Class 2 has received the plurality of the votes.
```

И мы снова не ошиблись, выбрав в качестве класса точки 10 класс 2. Дополнительное взвешенное голосование потенциально может улучшить итоговые прогнозы.

Естественно, это никак не гарантируется. Иногда взвешенное голосование может только ухудшить результаты. В зависимости от установленного значения *K* оно может либо улучшить, либо ухудшить наши прогнозы. И мы не можем быть уверены в том или ином, пока не протестируем эффективность прогнозирования по ряду параметров. Такое тестирование потребует выработки надежного показателя для измерения точности прогнозов.

20.2. ИЗМЕРЕНИЕ ТОЧНОСТИ ПРОГНОЗИРОВАНИЯ МЕТОК

К этому моменту мы познакомились с прогнозированием класса одной случайно выбранной точки. Теперь нам нужно проанализировать прогнозы для всех точек в `X_test`. Для этого определим функцию `predict`, получающую индекс неразмеченной точки и значение *K*, которое мы установим равным 1.

ПРИМЕЧАНИЕ

Мы намеренно передаем низкое значение *K* для генерации множества ошибок, требующих доработки. Позже измерим эту погрешность среди нескольких значений *K*, чтобы оптимизировать эффективность.

Последним параметром является логический `weighted_voting`, который мы устанавливаем как `False`. Этот параметр определяет, нужно ли распределять голоса согласно расстоянию (листинг 20.11).

Листинг 20.11. Параметризация прогнозов KNN

```
def predict(index, K=1, weighted_voting=False):
    labeled_distances = distance_matrix[index]
    labeled_neighbors = np.argsort(labeled_distances)[:K]
    class_to_votes = defaultdict(int)
    for neighbor in labeled_neighbors:
        label = y_train[neighbor]
        distance = distance_matrix[index][neighbor]
        num_votes = 1 / max(distance, 1e-10) if weighted_voting else 1
        class_to_votes[label] += num_votes
    return max(class_to_votes, key=lambda x: class_to_votes[x])

assert predict(random_index, K=3) == 2
assert predict(random_index, K=4, weighted_voting=True) == 2
```

Прогнозирует метку точки, используя ее индекс строки в матрице расстояний на основе K ближайших соседей. Логический параметр weighted_voting определяет, нужно ли взвешивать голоса на основе расстояния до соседей

Получает K ближайших соседей

Возвращает метку класса с большинством голосов

Если weighted_voting равно False, взвешивает голоса поровну, в противном случае — на основе обратного расстояния. При вычислении обратного расстояния мы принимаем меры предосторожности, чтобы избежать деления на ноль

Теперь выполним `predict` для всех неразмеченных индексов (листинг 20.12). Следуя общепринятому соглашению об именовании, мы сохраняем спрогнозированные классы в массиве `y_pred`.

Листинг 20.12. Прогнозирование классов всех неразмеченных цветов

```
y_pred = np.array([predict(i) for i in range(y_test.size)])
```

Нам нужно сравнить спрогнозированные классы с фактическими классами в `y_test`. Начнем с вывода обоих массивов, `y_pred` и `y_test` (листинг 20.13).

Листинг 20.13. Сравнение спрогнозированных и фактических классов

```
print(f"Predicted Classes:\n{y_pred}")
print(f"\nActual Classes:\n{y_test}")

Predicted Classes:
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 2 0 2 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0
 2 1 1 2 0 2 0 0 1 2 2 1 2 1 2 1 1 1 1 1 1 1 2 1 0 2 1 1 1 2 2 0 0 2 1 0 0
 1 0 2 1 0 1 2 1 0 2 2 2 2 0 0 2 2 0 2 0 2 2 0 0 2 0 0 0 1 2 2 0 0 0 1 1 0
 0 1]

Actual Classes:
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0
 1 1 1 2 0 2 0 0 1 2 2 2 2 1 2 1 1 2 2 2 2 1 2 1 0 2 1 1 1 1 2 0 0 2 1 0 0
 1 0 2 1 0 1 2 1 0 2 2 2 2 0 0 2 2 0 2 0 2 2 0 0 2 0 0 0 1 2 2 0 0 0 1 1 0
 0 1]
```

Два этих массива сравнивать трудно, но для этого есть более простой способ, а именно агрегация массивов в единую матрицу `M`, представленную тремя столбцами и строками, соответствующими количеству классов. В строках будут отслеживаться спрогнозированные классы, а в столбцах — идентификаторы истинных классов. Каждый элемент `M[i][j]` отражает совпадения между спрогнозированным классом *i* и фактическим классом *j*, как показано на рис. 20.5. Такой тип матричного представления известен как *матрица несоответствий*, или *матрица ошибок*. Как мы вскоре увидим, она помогает количественно оценить ошибки прогнозирования.

		Фактические		
		Ирис щетинистый	Ирис разноцветный	Ирис виргинский
Спрогнозированные	Ирис щетинистый	**14**	1	1
	Ирис разноцветный	1	**11**	3
	Ирис виргинский	1	3	**10**

Рис. 20.5. Гипотетическое матричное представление спрогнозированных и фактических классов. Строки соответствуют спрогнозированным, а столбцы — фактическим классам. Каждый элемент M[i][j] отражает число совпадений между спрогнозированным классом *i* и фактическим классом *j*. Таким образом, диагональ матрицы демонстрирует все точные прогнозы

Сейчас мы вычислим эту матрицу, используя `y_pred` и `y_test`, и визуализируем ее в виде тепловой карты с помощью Seaborn (листинг 20.14; рис. 20.6).

Листинг 20.14. Вычисление матрицы несоответствий

```
import seaborn as sns
def compute_confusion_matrix(y_pred, y_test):
    num_classes = len(set(y_pred) | set(y_test))
    confusion_matrix = np.zeros((num_classes, num_classes))
    for prediction, actual in zip(y_pred, y_test):
        confusion_matrix[prediction][actual] += 1

    return confusion_matrix

M = compute_confusion_matrix(y_pred, y_test)
sns.heatmap(M, annot=True, cmap='YlGnBu',
            yticklabels=[f"Predict {i}" for i in range(3)],
            xticklabels = [f"Actual {i}" for i in range(3)])
plt.yticks(rotation=0)
plt.show()
```

Вычисляет матрицу несоответствий между y_pred и y_test

Проверяет общее число классов. Это значение определяет количество строк и столбцов матрицы

Каждый спрогнозированный класс Prediction соответствует фактическому классу Actual. Для каждой такой пары мы добавляем 1 в строку Prediction и столбец Actual нашей матрицы. Заметьте, если Prediction == Actual, значит, добавленное значение появляется на диагонали матрицы

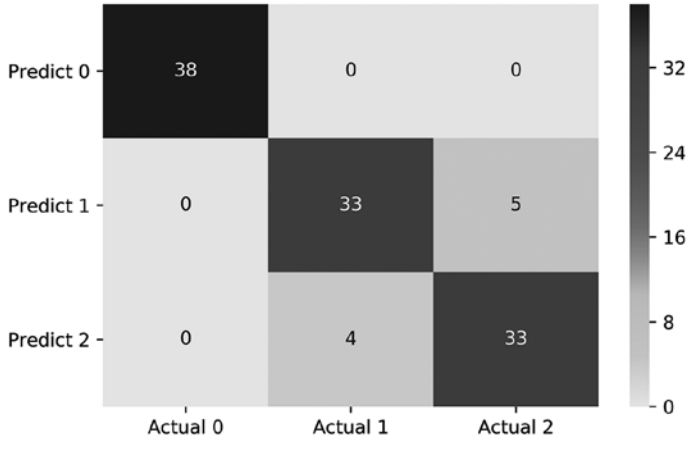


Рис. 20.6. Матрица несоответствий, сопоставляющая спрогнозированные и фактические результаты. Строки соответствуют спрогнозированным, а столбцы — фактическим классам. Элементы матрицы отражают все экземпляры среди спрогнозированных и фактических классов. Диагональ матрицы показывает все точные прогнозы. Большая часть подсчитанных результатов лежит вдоль этой диагонали, указывая на высокую точность нашей модели

Большинство значений матрицы лежат на ее диагонали. Каждый элемент диагонали в `M[i][i]` отслеживает количество точно спрогнозированных экземпляров класса *i*. Такие точные прогнозы обычно называются *истинно положительными*. Исходя из отображенных по диагонали значений, мы знаем, что количество истинно

положительных очень высоко. Теперь выведем их общее число, выполнив сложение по `M.diagonal()` (листинг 20.15).

Листинг 20.15. Подсчет количества точных прогнозов

```
num_accurate_predictions = M.diagonal().sum()
print(f"Our results contain {int(num_accurate_predictions)} accurate "
       "predictions.")

Our results contain 104 accurate predictions.
```

Результаты включают 104 точных прогноза — мы добились высокой точности. Естественно, не все прогнозы верны. Иногда наш классификатор путается и прогнозирует метку класса ошибочно — из 113 прогнозов девять в матрице лежат вне диагонали. Доля точных прогнозов называется показателем *общей точности* (accuracy). Общую точность можно вычислить делением суммы по диагонали на общую сумму элементов матрицы — в нашем случае деление 104 на 113 даст высокий показатель общей точности (листинг 20.16).

Листинг 20.16. Измерение общей точности

```
accuracy = M.diagonal().sum() / M.sum()
assert accuracy == 104 / (104 + 9)
print(f"Our predictions are {100 * accuracy:.0f}% accurate.")

Our predictions are 92% accurate.
```

Наши прогнозы довольно точны, но неидеальны. В выводе все же есть ошибки, и распределены они неравномерно. К примеру, при анализе матрицы мы видим, что прогнозы класса 0 всегда верны. Модель никогда не путает этот класс с другими — все 38 его прогнозов лежат на диагонали. Но для двух других классов это не так — модель периодически путает экземпляры классов 1 и 2.

Давайте попробуем оценить путаницу количественно. Рассмотрим элементы в строке 1 матрицы, которая отслеживает прогнозы класса 1. Сложение вдоль этой строки дает общее число элементов, спрогнозированных как принадлежащие к классу 1 (листинг 20.17).

Листинг 20.17. Подсчет спрогнозированных элементов класса 1

```
row1_sum = M[1].sum()
print(f"We've predicted that {int(row1_sum)} elements belong to Class 1.")

We've predicted that 38 elements belong to Class 1.
```

Мы спрогнозировали, что в классе 1 содержится 38 элементов. Сколько из этих прогнозов оказались верны? Из них 33 лежат вдоль диагонали `M[1][1]`, значит, мы корректно определили 33 истинно положительных класса 1. При этом оставшиеся пять прогнозов находятся в столбце 2. Эти пять *ложноположительных* представляют элементы класса 2, которые мы ошибочно определили как принадлежащие классу 1. Они делают прогнозы для класса 1 менее надежными. Одно то, что наша

модель возвращает метку класса 1, еще не означает верность данного прогноза. В действительности метка класса 1 верна только в 33 из 38 случаев. Соотношение 33/38 дает показатель, называемый *точностью* (precision), — так называется количество истинно положительных, разделенное на сумму истинно положительных и ложноположительных. Точность класса *i* также равна `M[i][i]`, разделенной на сумму вдоль строки *i*. Низкий показатель точности указывает, что спрогнозированная метка класса не особо надежна. Выведем точность класса 1 (листинг 20.18).

Листинг 20.18. Вычисление точности класса 1

```
precision = M[1][1] / M[1].sum()
assert precision == 33 / 38
print(f"Precision of Class 1 is {precision:.2f}")

Precision of Class 1 is 0.87
```

Точность класса 1 равна 0,87, значит, его метка надежна лишь в 87 % случаев. В оставшихся 13 % прогнозы оказываются ложноположительными. Они и являются причиной ошибки, но не только они — среди столбцов матрицы можно обнаружить и другие несоответствия. Рассмотрим, к примеру, столбец 1, отслеживающий все элементы в `y_test`, чья истинная метка равна классу 1. Сложение значений столбца 1 дает общее число элементов класса 1 (листинг 20.19).

Листинг 20.19. Подсчет общего числа элементов класса 1

```
col1_sum = M[:,1].sum()
assert col1_sum == y_test[y_test == 1].size
print(f"{int(col1_sum)} elements in our test set belong to Class 1.")

37 elements in our test set belong to Class 1.
```

Классу 1 принадлежат 37 элементов тестовой выборки. Из них 33 лежат вдоль диагонали `M[1][1]` — эти истинно положительные элементы были определены корректно. Оставшиеся же четыре элемента лежат в строке 2 — эти *ложноотрицательные* принадлежат классу 2. Значит, идентификация элементов класса 1 неполноценна. Из 37 возможных экземпляров классов только 33 были определены верно. Соотношение 33/37 дает показатель, называемый *полнотой*, — так называется количество истинно положительных, разделенное на сумму истинно положительных и ложноположительных. Полнота класса *i* также равна `M[i][i]`, разделенной на сумму вдоль столбца *i*. Низкая полнота говорит о том, что наш предиктор часто упускает подходящие экземпляры класса. Давайте выведем полноту класса 1 (листинг 20.2).

Листинг 20.20. Вычисление полноты класса 1

```
recall = M[1][1] / M[:,1].sum()
assert recall == 33 / 37
print(f"Recall of Class 1 is {recall:.2f}")

Recall of Class 1 is 0.89
```

Полнота класса 1 равна 0,89, значит, мы можем верно определить 89 % экземпляров этого класса. Оставшиеся же 11 % определяются ошибочно. Данное значение полноты измеряет долю правильно определенных цветов класса 1 на лугу. В противоположность этому точность измеряет вероятность того, что прогноз класса 1 окажется корректным.

Стоит отметить, что максимальной полноты 1 достичь легко. Достаточно просто разметить каждую входящую точку данных как принадлежащую классу 1. В таком случае мы обнаружим все верные экземпляры этого класса, но за такую высокую полноту придется заплатить точностью. А упадет она значительно, поскольку все экземпляры класса 0 и класса 2 будут ошибочно определены как принадлежащие классу 1. Этот низкий показатель точности равен `M[1][1] / M.sum()` (листинг 20.21).

Листинг 20.21. Проверка точности при полноте 1,0

```
low_precision = M[1][1] / M.sum()
print(f"Precision at a trivially maximized recall is {low_precision:.2f}")

Precision at a trivially maximized recall is 0.29
```

Аналогичным образом максимизированная точность окажется бесполезной при низкой полноте. Представьте, что точность класса 1 равна 1. В таком случае мы бы имели 100%-ную уверенность, что все прогнозы класса 1 верны. Однако если соответствующая полнота будет низкой, то большинство экземпляров класса 1 будут ошибочно определены как принадлежащие другому классу. Поэтому высокий уровень уверенности оказывается не особо полезным, если классификатор игнорирует большую часть истинных экземпляров.

Хорошая предиктивная модель должна давать и высокую точность, и высокую полноту. По этой причине нам нужно совместить два этих показателя в один. Но как объединить две различные меры оценки? Одно из очевидных решений подразумевает получение среднего через выполнение `(precision + recall)/2`. К сожалению, это решение имеет неожиданный недостаток. И точность, и полнота являются дробями — `M[1][1]/M[1].sum()` и `M[1][1]/M[:,1].sum()` соответственно. Они имеют одинаковое делимое, но разные делители. И это проблема. Дроби можно складывать только при равных делителях. Выходит, задуманное нами сложение, необходимое для получения среднего, не подходит. Что же делать? Можно получить обратное значение точности и полноты. В результате этой инверсии делитель и делимое поменяются местами и `1 / precision` с `1 / recall` будут иметь одинаковый делитель `M[1][1]`. Теперь эти инверсированные дроби можно сложить. Посмотрим, что происходит при получении среднего инвертированных показателей (листинг 20.22).

Листинг 20.22. Получение среднего инвертированных показателей

```
inverse_average = (1 / precision + 1 / recall) / 2
print(f"The average of the inverted metrics is {inverse_average:.2f}")

The average of the inverted metrics is 1.14
```

Среднее инвертированных показателей больше 1,0, но и точность и полнота имеют максимальный предел 1. Значит, операция их агрегирования должна давать значение меньше 1,0. Обеспечить это можно инвертированием полученного среднего (листинг 20.23).

Листинг 20.23. Инвертирование инвертированного среднего

```
result = 1 / inverse_average
print(f"The inverse of the average is {result:.2f}")

The inverse of the average is 0.88
```

Итоговый агрегированный показатель равен 0,88, то есть лежит между точностью 0,87 и полнотой 0,89. Получается, что это агрегирование представляет идеальный баланс точности и полноты. Данный обобщенный показатель называется *F-мерой*. Ее можно вычислить более прямым путем посредством выполнения `2 * precision * recall / (precision + recall)` (листинг 20.24).

ПРИМЕЧАНИЕ

Инверсия арифметического среднего инвертированных значений называется средним гармоническим. Среднее гармоническое предназначено для измерения центральной тенденции показателей, таких как скорости. В качестве примера представьте, что атлет бежит вокруг озера три круга, каждый длиной в милю. Первый круг он пробегает за 10 мин, следующий — за 16, а последний — за 20. Получается, что скорости атлета в милях в минуту будут равны 1/10 (0,1), 1/16 (0,0625) и 1/20 (0,05). Арифметическое среднее составляет (0,1 + 0,0625 + 0,05) / 3, то есть приблизительно 0,071. Однако это значение ошибочно, поскольку здесь суммируются разные делители. Вместо этого нужно вычислить среднее гармоническое, то есть 3 / (10 + 16 + 20), которое равняется примерно 0,065 мили/мин. По определению F-мера соответствует среднему гармоническому точности и полноты.

Листинг 20.24. Вычисление F-меры класса 1

```
f_measure = 2 * precision * recall / (precision + recall)
print(f"The f-measure of Class 1 is {f_measure:.2f}")

The f-measure of Class 1 is 0.88
```

Здесь нужно отметить, что, хотя в этом случае F-мера и равна среднему точности и полноты, так бывает не всегда. Рассмотрим прогноз, имеющий один истинно положительный результат, один ложноположительный и ноль ложноотрицательных. Каковы будут точность и полнота? Как их среднее значение сопоставляется с F-мерой? Сейчас выясним (листинг 20.25).

Листинг 20.25. Сравнение F-меры со средним

```
tp, fp, fn = 1, 1, 0
precision = tp / (tp + fp)
recall = tp / (tp + fn)
f_measure = 2 * precision * recall / (precision + recall)
average = (precision + recall) / 2
```

```
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"Average: {average}") print(f"F-measure: {f_measure:.2f}")

Precision: 0.5
Recall: 1.0
Average: 0.75
F-measure: 0.67
```

В этом теоретическом примере точность оказалась низкой — всего 50 %. При этом полнота составила все 100 %. Среднее этих показателей составляет допустимые 75 %. Тем не менее F-мера оказывается намного ниже этого среднего, поскольку высокую полноту невозможно оправдать исключительно низким значением точности.

F-мера предоставляет нам надежную оценку отдельного класса. С учетом этого далее вычислим данный показатель для каждого класса в наборе данных (листинг 20.26).

Листинг 20.26. Вычисление F-меры для каждого класса

```
def compute_f_measures(M):
    precisions = M.diagonal() / M.sum(axis=0)
    recalls = M.diagonal() / M.sum(axis=1)
    return 2 * precisions * recalls / (precisions + recalls)

f_measures = compute_f_measures(M)
for class_label, f_measure in enumerate(f_measures):
    print(f"The f-measure for Class {class_label} is {f_measure:.2f}")

The f-measure for Class 0 is 1.00
The f-measure for Class 1 is 0.88
The f-measure for Class 2 is 0.88
```

F-мера класса 0 равна 1 — этот выделяющийся класс можно определить с идеальной точностью и полнотой. При этом F-мера классов 1 и 2 составляет 0,88. Различие между этими классами неидеально, и один часто ошибочно принимается за другой. Эти ошибки снижают точность и полноту каждого класса. Тем не менее итоговый показатель 0,88 полностью приемлем.

ПРИМЕЧАНИЕ

Официального стандарта для приемлемой F-меры не существует. Ее подходящие значения от задачи к задаче могут варьироваться. Однако принято рассматривать F-меру как экзаменационную оценку — значение в диапазоне от 0,9 до 1,0 оценивается как 5 баллов и показывает, что модель справляется исключительно хорошо. Показатель F-меры от 0,80 до 0,89 расценивается как 4 балла и говорит о возможности доработки, хотя модель и так приемлема. Значение в диапазоне от 0,70 до 0,79 уже оценивается как 3 балла, то есть модель справляется адекватно, но качество ее результатов не впечатляет. F-мера от 0,60 до 0,69 соотносится с 2 баллами, являясь неприемлемым показателем для модели, но все же лучше, чем полная случайность. Значения ниже 0,6 обычно рассматриваются как совсем ненадежные.

Мы вычислили F-меру для трех разных классов. Эти показатели можно совместить в один, получив их среднее. Код листинга 20.27 выводит объединенный показатель F-меры.

ПРИМЕЧАНИЕ

Три наши F-меры представляют собой дроби с потенциально разными делителями. Как уже говорилось, дроби следует совмещать только при равных делителях. К сожалению, в отличие от точности и полноты, не существует метода для достижения равенства делителя полученных показателей F-мер. Так что для получения объединенного показателя у нас остается единственный вариант — вычислить их среднее.

Листинг 20.27. Вычисление объединенной F-меры для всех классов

```
avg_f = f_measures.mean()
print(f"Our unified f-measure equals {avg_f:.2f}")

Our unified f-measure equals 0.92
```

Показатель F-меры 0,92 идентичен показателю общей точности. И это неудивительно, поскольку и F-мера, и общая точность предназначены для измерения эффективности модели. Тем не менее нужно подчеркнуть, что F-мера и общая точность не обязательно будут одинаковыми. Разница между ними особенно заметна в случаях, когда классы *не сбалансированы*. В несбалансированном наборе данных существует намного больше экземпляров некоторого класса A, чем некоторого класса B. Далее мы рассмотрим пример, в котором есть 100 экземпляров класса A и всего один экземпляр класса B. Кроме того, мы предположим, что прогнозы относительно класса B имеют 100 % полноты и 50 % точности. Этот сценарий можно представить с помощью матрицы несоответствий размером 2×2, имеющей форму `[[99, 0], [1, 1]]`. Давайте сравним общую точность с объединенной F-мерой для этого несбалансированного результата (листинг 20.28).

Листинг 20.28. Сравнение показателей эффективности при анализе несбалансированных данных

```
M_imbalanced = np.array([[99, 0], [1, 1]])
accuracy_imb = M_imbalanced.diagonal().sum() / M_imbalanced.sum()
f_measure_imb = compute_f_measures(M_imbalanced).mean()
print(f"The accuracy for our imbalanced dataset is {accuracy_imb:.2f}")
print(f"The f-measure for our imbalanced dataset is {f_measure_imb:.2f}")

The accuracy for our imbalanced dataset is 0.99
The f-measure for our imbalanced dataset is 0.83
```

Общая точность близка к 100 %. Такое значение вводит в заблуждение — оно не отражает правдиво ту ужасную точность, с которой модель прогнозирует второй класс. При этом более низкое значение F-меры лучше демонстрирует соотношение между прогнозами для разных классов. Как правило, ввиду чувствительности к отсутствию баланса F-мера считается более значимым показателем. Забегая вперед, скажу, что при оценке классификаторов мы опираемся именно на F-меру.

20.2.1. Функции оценки прогнозов в scikit-learn

Все рассмотренные на данный момент показатели оценки прогнозов присутствуют в scikit-learn. Их можно импортировать из модуля `sklearn.metrics`. Функция каждого показателя получает на входе `y_pred` и `y_test` и возвращает выбранный нами критерий оценки. К примеру, можно вычислить матрицу несоответствий, импортировав и выполнив `confusion_matrix` (листинг 20.29).

Листинг 20.29. Вычисление матрицы несоответствий с помощью scikit-learn

```
from sklearn.metrics import confusion_matrix
new_M = confusion_matrix(y_pred, y_test)
assert np.array_equal(new_M, M)
print(new_M)
```

```
[[38  0 0]
 [ 0 33 5]
 [ 0 4 33]]
```

Аналогичным образом можно вычислить общую точность, импортировав и выполнив `accuracy_score` (листинг 20.30).

Листинг 20.30. Вычисление общей точности с помощью scikit-learn

```
from sklearn.metrics import accuracy_score
assert accuracy_score(y_pred, y_test) == accuracy
```

Кроме того, F-меру можно вычислить с помощью функции `f1_score` (листинг 20.31). При использовании этой функции есть дополнительные нюансы, поскольку F-меру можно вернуть в виде вектора объединенного среднего. Передача `average=None` в данную функцию приведет к возвращению вектора значений F-меры для каждого класса.

Листинг 20.31. Вычисление всех F-мер с помощью scikit-learn

```
from sklearn.metrics import f1_score
new_f_measures = f1_score(y_pred, y_test, average=None)
assert np.array_equal(new_f_measures, f_measures)
print(new_f_measures)
```

```
[1.   0.88 0.88]
```

При этом в случае передачи `average='macro'` возвращается единый общий показатель.

ПРИМЕЧАНИЕ

Передавая average='micro', мы вычисляем средние показатели точности и полноты для всех классов. Затем эти средние значения используются для вычисления единого показателя F-меры (листинг 20.32). Как правило, такой подход не особо влияет на итоговый показатель общей F-меры.

Листинг 20.32. Вычисление общей F-меры с помощью scikit-learn

```
new_f_measure = f1_score(y_pred, y_test, average='macro')
assert new_f_measure == new_f_measures.mean()
assert new_f_measure == avg_f
```

Используя функцию `f1_score`, можно легко оптимизировать наш классификатор KNN относительно его входных параметров.

ТИПИЧНЫЕ ФУНКЦИИ ОЦЕНКИ КЛАССИФИКАТОРА В SCIKIT-LEARN

- `M = confusion_matrix(y_pred, y_test)` — возвращает матрицу несоответствий `M` на основе спрогнозированных классов в `y_pred` и фактических классов в `y_test`. Каждый элемент матрицы `M[i][j]` отражает количество случаев, когда для любого возможного `index` часть `y_pred[index] == i`, а `y_test[index] == j`.
- `accuracy_score(y_pred, y_test)` — возвращает показатель общей точности на основе спрогнозированных классов в `y_pred` и фактических классов в `y_test`. При рассмотрении матрицы несоответствий `M` показатель общей точности равен `M.diagonal().sum() / M.sum()`.
- `f_measure_vector = f1_score(y_pred, y_test, average=None)` — возвращает вектор F-мер для всех возможных классов `f_measure_vector.size`. F-мера класса *i* равна `f_measure_vector[i]`, что соответствует среднему гармоническому точности и полноты класса *i*. И точность, и полноту можно вычислить на основе матрицы несоответствий `M`. Точность класса *i* равна `M[i][i] / M[i].sum()`, а его полнота — `M[i][i] / M[:,i].sum()`. Итоговое значение F-меры `f_measure_vector[i]` равно `2 * preci-sion * recall / (precision + recall)`.
- `f1_score(y_pred, y_test, average='macro')` — возвращает среднюю F-меру, равную `f_measure_vector.mean()`.

20.3. ОПТИМИЗАЦИЯ ЭФФЕКТИВНОСТИ KNN

Сейчас наша функция `predict` получает два входных параметра: *K* и `weighted_voting`. Они должны устанавливаться до обучения, влияя на эффективность классификатора. Ученые, занимающиеся данными, называют их *гиперпараметрами*. Все модели машинного обучения имеют некие гиперпараметры, которые можно подгонять перебором всех возможных комбинаций *K* и `weighted_voting`. Наши значения *K* охватывают диапазон от `1` до `y_train.size`, а логический параметр `weighted_voting` устанавливается как `True` либо `False`. При каждой комбинации гиперпараметров мы выполняем обучение на `y_train` и вычисляем `y_pred`

(листинг 20.33). После этого на основе своих прогнозов получаем F-меру. Все F-меры наносятся на график относительно входного *K*. Мы строим две кривые: одну для `weighted_voting = True`, а вторую для `weighted_voting = False` (рис. 20.7). В завершение находим на графике максимальную F-меру и возвращаем ее оптимизированные параметры.

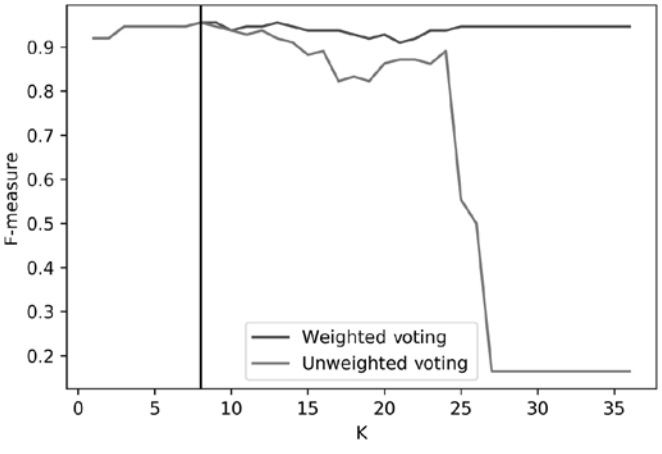


Рис. 20.7. График показателей эффективности распределения голосов взвешенного и невзвешенного KNN для диапазона входных значений K. F-мера максимизируется, когда K = 8. При низких значениях K нет большой разницы между взвешенным и невзвешенным голосованием. Однако при K > 10 эффективность невзвешенного распределения голосов начинает падать

Листинг 20.33. Оптимизация гиперпараметров KNN

```
k_values = range(1, y_train.size)
weighted_voting_bools = [True, False]
f_scores = [[], []]

params_to_f = {}
for k in k_values:
    for i, weighted_voting in enumerate(weighted_voting_bools):
        y_pred = np.array([predict(i, K=k,
                           weighted_voting=weighted_voting)
                   for i in range(y_test.size)])
        f_measure = f1_score(y_pred, y_test, average='macro')
        f_scores[i].append(f_measure)
        params_to_f[(k, weighted_voting)] = f_measure

(best_k, best_weighted), best_f = max(params_to_f.items(),
                                      key=lambda x: x[1])
plt.plot(k_values, f_scores[0], label='Weighted Voting')
```

Отслеживает сопоставление между каждой комбинацией параметров и F-мерой

Вычисляет прогнозы KNN для каждой комбинации параметров

Вычисляет F-меру для каждой комбинации параметров

Находит параметры, максимизирующие F-меру

```
plt.plot(k_values, f_scores[1], label='Unweighted Voting')
plt.axvline(best_k, c='k')
plt.xlabel('K')
plt.ylabel('F-measure')
plt.legend()
plt.show()

print(f"The maximum f-measure of {best_f:.2f} is achieved when K={best_k} "
      f"and weighted_voting={best_weighted}")

The maximum f-measure of 0.96 is achieved when K=8 and weighted_voting=True
```

Максимальная эффективность достигается при $K = 8$ и использовании взвешенного распределения голосов. Однако при таком его значении между взвешенным и невзвешенным выводами существенной разницы нет. Интересно, что по мере увеличения K невзвешенная F-мера начинает резко падать. При этом взвешенная F-мера продолжает расти и превышает 90 %. Таким образом, взвешенный KNN оказывается более стабилен, чем его невзвешенный вариант.

Такие результаты мы получили исчерпывающим перебором всех возможных входных параметров. Этот подход называется *вариацией параметров,* или *поиском по сетке*. Поиск по сетке — простой, но эффективный способ оптимизации гиперпараметров. Несмотря на свою вычислительную сложность при большом числе параметров, этот метод очень легко распараллелить. При достаточной вычислительной мощности поиск по сетке способен эффективно оптимизировать многие распространенные алгоритмы машинного обучения. Как правило, этот метод реализуется следующей серией шагов.

1. Выбираются интересующие гиперпараметры.
2. Каждому гиперпараметру присваивается диапазон значений.
3. Входные данные разделяются на обучающую и валидационную выборки. Последняя используется для оценки качества прогнозов. Такой подход называется *кросс-валидацией*. Заметьте, что можно дополнительно разделить данные на несколько обучающих и валидационных выборок. Это позволит усреднить набор результатов оценки прогнозирования до единого показателя.
4. Перебираются все возможные комбинации гиперпараметров.
5. В каждой итерации классификатор обучается на обучающих данных, используя заданные гиперпараметры.
6. С помощью валидационной выборки оценивается эффективность классификатора.
7. По завершении всех итераций возвращается комбинация гиперпараметров, обеспечившая максимальное значение показателя.

Scikit-learn позволяет выполнять поиск по сетке для всех встроенных алгоритмов машинного обучения. Далее мы задействуем эту библиотеку для поиска гиперпараметров в KNN.

20.4. ПОИСК ПО СЕТКЕ С ПОМОЩЬЮ SCIKIT-LEARN

Scikit-learn имеет встроенную логику для выполнения классификации KNN. Эту логику мы задействуем, импортировав класс `KNeighborsClassifier` (листинг 20.34).

Листинг 20.34. Импорт класса KNN из scikit-learn

```
from sklearn.neighbors import KNeighborsClassifier
```

Инициализация класса приводит к созданию объекта классификатора KNN. По общепринятому соглашению мы сохраняем этот объект в переменной `clf` (листинг 20.35).

ПРИМЕЧАНИЕ

Алгоритм KNN можно расширить за пределы простой классификации, подстроив для прогнозирования непрерывных значений. Представьте, что мы хотим спрогнозировать стоимость продажи дома. Это можно сделать усреднением известных стоимостей аналогичных домов по соседству. Аналогичным образом можно построить регрессор KNN, прогнозирующий непрерывное значение точки данных путем усреднения известных значений ее соседей. Scikit-learn включает класс KNeighborsRegressor, созданный конкретно для этой цели.

Листинг 20.35. Инициализация классификатора KNN из scikit-learn

```
clf = KNeighborsClassifier()
```

Инициализированный объект `clf` содержит настраиваемые параметры для K и взвешенного голосования. Значение K хранится в атрибуте `clf.n_neighbors`, а параметр взвешенного голосования — в атрибуте `clf.weights`. Давайте выведем и проанализируем оба этих атрибута (листинг 20.36).

Листинг 20.36. Вывод параметров настройки классификатора KNN

```
K = clf.n_neighbors
weighted_voting = clf.weights
print(f"K is set to {K}.")
print(f"Weighted voting is set to '{weighted_voting}'.")

K is set to 5.
Weighted voting is set to 'uniform'.
```

Наше K установлено равным 5, а взвешенное голосование — равным `uniform`, указывая на то, что все голоса будут взвешены как равные. Передача `weights='distance'` в функцию инициализации гарантирует, что голоса будут взвешены в соответствии с расстоянием. Кроме того, передача `n_neighbors=4` установит K равным 4 (листинг 20.37). Далее мы повторно инициализируем `clf` с этими параметрами.

Листинг 20.37. Установка параметров классификатора KNN из scikit-learn

```
clf = KNeighborsClassifier(n_neighbors=4, weights='distance')
assert clf.n_neighbors == 4
assert clf.weights == 'distance'
```

Теперь нам нужно обучить модель KNN. Любой классификатор `clf` из scikit-learn можно обучить с помощью метода `fit`. Для этого достаточно выполнить `clf.fit(X, y)`, где `X` представляет матрицу признаков, а `y` — массив меток классов. Обучим классификатор, используя обучающую выборку, определенную посредством `X_train` и `y_train` (листинг 20.38).

Листинг 20.38. Обучение классификатора KNN из scikit-learn

```
clf.fit(X_train, y_train)
```

После обучения `clf` может прогнозировать классы любой входной матрицы `X_test`, чьи размеры соответствуют `X_train`. Отвечает за прогнозирование метод `clf.predict` (листинг 20.39). Выполнение `clf.predict(X_test)` ведет к возвращению массива прогнозов `y_pred`. После этого `y_pred` вместе с `y_test` можно будет использовать для вычисления F-меры.

Листинг 20.39. Прогнозирование классов с помощью обученного классификатора KNN

```
y_pred = clf.predict(X_test)
f_measure = f1_score(y_pred, y_test, average='macro')
print(f"The predicted classes are:\n{y_pred}")
print(f"\nThe f-measure equals {f_measure:.2f}.")

The predicted classes are:
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0
 2 1 1 2 0 2 0 0 1 2 2 1 2 1 2 1 1 1 1 1 1 1 2 1 0 2 1 1 1 1 2 0 0 2 1 0 0
 1 0 2 1 0 1 2 1 0 2 2 2 2 0 0 2 2 0 2 0 2 2 2 0 0 2 0 0 0 1 2 2 0 0 0 1 1 0
 0 1]

The f-measure equals 0.95.
```

Классификатор `clf` позволяет также извлекать более тонкие результаты прогнозов. К примеру, можно сгенерировать долю голосов, полученных каждым классом для входной выборки в `X_test`. Для получения этого распределения голосов нужно выполнить `clf.predict_proba(X_test)`. Метод `predict_proba` возвращает матрицу, столбцы которой отражают проценты голосов. Код листинга 20.40 выводит первые четыре строки этой матрицы, соответствующие `X_test[:5]`.

Листинг 20.40. Вывод процентов голосов для каждого класса

```
vote_ratios = clf.predict_proba(X_test)
print(vote_ratios[:4])
```

```
array([[0.        , 0.21419074, 0.78580926],
       [0.        , 1.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [0.        , 0.        , 1.        ]])
```

Как видите, точка данных в `X_test[0]` получила 78,5 % голосов в пользу класса 2. Остальные голоса были отданы классу 1. При этом `X_test[4]` получила в пользу класса 2 все 100 % голосов. Несмотря на то что обеим точкам данных присвоена метка класса 2, вторая точка получила эту метку с большей степенью уверенности.

Стоит отметить, что все классификаторы scikit-learn содержат собственную версию `predict_proba`. Этот метод возвращает приблизительное распределение вероятностей принадлежности точек данных к некоторому классу. Индекс столбца с наивысшей вероятностью соответствует метке класса в `y_pred`.

АКТУАЛЬНЫЕ МЕТОДЫ КЛАССИФИКАЦИИ SCIKIT-LEARN

- `clf = KNeighborsClassifier()` — инициализация классификатора KNN, в котором $K = 5$, а голосование равновесно для пяти ближайших соседей.
- `clf = KNeighborsClassifier(n_neighbors=x)` — инициализирует классификатор KNN, в котором $K = x$, а голосование равновесно для x соседей.
- `clf = KNeighborsClassifier(n_neighbors=x, weights='distance')` — инициализирует классификатор KNN, в котором $K = x$, а голосование взвешивается в соответствии с расстоянием до каждого из x соседей.
- `clf.fit(X_train, y_train)` — подгоняет любой классификатор `clf` для прогнозирования классов y по признакам `X` на основе обучающих признаков `X_train` и обучающих размеченных классов `y_train`.
- `y = clf.predict(X)` — прогнозирует массив классов, связанных с матрицей признаков `X`. Каждый спрогнозированный класс `y[i]` сопоставляется со строкой признаков `X[i]` матрицы.
- `M = clf.predict_proba(X)` — возвращает матрицу `M` с распределениями вероятностей. Каждая строка `M[i]` представляет распределение вероятности того, что точка данных `i` принадлежит некоторому классу. Прогнозирование класса этой точки данных соответствует максимальному значению распределения. Если коротко, то `M[i].argmax() == clf.predict(X)[i]`.

Теперь переключимся на выполнение поиска по сетке в `KNeighborsClassifier`. Сначала нужно указать сопоставление в словаре между нашими гиперпараметрами и диапазонами их значений. Ключи словарей равны входным параметрам `n_neighbors` и `weights`. Значения словаря равны соответствующим перебираемым объектам `range(1, 40)` и `['uniform', 'distance']`. Давайте создадим словарь `hyperparams` (листинг 20.41).

Листинг 20.41. Определение словаря гиперпараметров

```
hyperparams = {'n_neighbors': range(1, 40),
               'weights': ['uniform', 'distance']}
```

В ручном поиске по сетке число соседей было от 1 до y_train.size при y_train.size, равном 37. Однако этот диапазон параметров можно ограничить любым значением. Здесь мы используем в качестве него красивое круглое число 40

Далее нужно импортировать scikit-learn-класс `GridSearchCV`, который мы применяем для поиска по сетке (листинг 20.42).

Листинг 20.42. Импорт класса поиска по сетке из scikit-learn

```
from sklearn.model_selection import GridSearchCV
```

Время инициализировать класс `GridSearchCV`. В метод инициализации мы передаем три входных параметра. Первый — `KNeighborsClassifier()`, инициализированный объект scikit-learn, гиперпараметры которого нужно оптимизировать, второй — словарь `hyperparams`, а третий — `scoring='f1_macro'`, устанавливающий в качестве параметра оценки усредненную F-меру.

Код листинга 20.43 выполняет `GridSearchCV(KNeighborsClassifier(), hyperparams, scoring='f1_macro')`. Этот инициализированный объект может выполнять классификацию, поэтому присваиваем его переменной `clf_grid`.

Листинг 20.43. Инициализация scikit-learn класса поиска по сетке

```
clf_grid = GridSearchCV(KNeighborsClassifier(), hyperparams,
                        scoring='f1_macro')
```

Теперь можно запускать поиск по сетке для полностью размеченного набора данных `X, y` (листинг 20.44). Перебор этих параметров реализуется с помощью выполнения `clf_grid.fit(X, y)`. Внутренние методы scikit-learn автоматически разделяют `X` и `y` во время процесса оценки.

Листинг 20.44. Поиск по сетке с помощью scikit-learn

```
clf_grid.fit(X, y)
```

Поиск по сетке завершен, оптимизированные параметры сохранены в атрибуте `clf_grid.best_params_`, а связанная с ними F-мера — в `clf_grid.best_score_`. Теперь выведем эти результаты (листинг 20.45).

Листинг 20.45. Проверка оптимизированных результатов поиска по сетке

```
best_f = clf_grid.best_score_
best_params = clf_grid.best_params_
print(f"A maximum f-measure of {best_f:.2f} is achieved with the "
      f"following hyperparameters:\n{best_params}")

A maximum f-measure of 0.99 is achieved with the following hyperparameters:
{'n_neighbors': 10, 'weights': 'distance'}
```

Поиск по сетке scikit-learn дал F-меру, равную 0,99. Это значение выше, чем результат нашего поиска, составивший 0,96. Почему так произошло? Дело в том, что scikit-learn выполнила более сложную версию кросс-валидации. Вместо разделения набора данных на две части она разбила данные на пять равных частей. Каждая часть выступала в качестве обучающей выборки, а внешние для нее данные использовались для тестирования. В итоге были вычислены и усреднены пять значений F-меры. Конечное значение 0,99 представляет собой более точную оценку эффективности классификатора.

ПРИМЕЧАНИЕ

Деление данных на пять частей для оценки называется пятипроходной кросс-валидацией. Как правило, данные можно разделить на K равных частей. В GridSearchCV это деление управляется параметром cv. При передаче cv = 2 данные делятся на две части и итоговая F-мера приближается к нашему изначальному значению 0,96.

Максимальная эффективность достигается при установке `n_neighbors` равным `10` и активации взвешенного голосования. Фактический классификатор KNN, содержащий эти параметры, хранится в атрибуте `clf_grid.best_estimator_` (листинг 20.46).

ПРИМЕЧАНИЕ

Получить F-меру, равную 0,99, позволяют несколько комбинаций гиперпараметров, и на разных машинах можно выбрать разные. В связи с этим ваш вывод параметров может быть несколько иным, несмотря на то что оптимизированная F-мера останется прежней.

Листинг 20.46. Обращение к оптимизированному классификатору

```
clf_best = clf_grid.best_estimator_
assert clf_best.n_neighbors == best_params['n_neighbors']
assert clf_best.weights == best_params['weights']
```

Прогноз относительно новых данных можно сделать, используя `clf_best`. В качестве альтернативы можно непосредственно задействовать оптимизированный объект `clf_grid`, выполнив `clf_grid.predict`. Оба этих объекта вернут идентичные результаты (листинг 20.47).

Листинг 20.47. Генерация прогнозов с помощью clf_grid

```
assert np.array_equal(clf_grid.predict(X), clf_best.predict(X))
```

РЕЛЕВАНТНЫЕ МЕТОДЫ И АТРИБУТЫ ПОИСКА ПО СЕТКЕ В SCIKIT-LEARN

- `clf_grid = GridSearchCV(ClassifierClass(), hyperparams, scoring = scoring_metric)` — создает объект поиска по сетке, оптимизирующий прогнозы классификатора по всем возможным гиперпараметрам на основе значения, указанного в `scoring`. Если `ClassifierClass()` равен `KNeighborsClassifier()`, тогда для оптимизации KNN используется `clf_grid`. Если `scoring_metric` равна `f1_macro`, для оптимизации применяется средняя F-мера.
- `clf_grid.fit(X, y)` — выполняет поиск по сетке для оптимизации эффективности классификатора по всем возможным комбинациям значений гиперпараметров.
- `clf_grid.best_score_` — возвращает оптимальный показатель эффективности классификатора после выполнения поиска по сетке.
- `clf_grid.best_params_` — возвращает комбинацию гиперпараметров, обеспечивающую оптимальную эффективность, на основе поиска по сетке.
- `clf_best = clf_grid.best_estimator_` — возвращает объект классификатора scikit-learn, дающий оптимальную эффективность, на основе поиска по сетке.
- `clf_grid.predict(X)` — сокращение для `clf_grid.best_estimator_.predict(X)`.

20.5. ОГРАНИЧЕНИЯ АЛГОРИТМА KNN

KNN — это простейший из всех алгоритмов обучения с учителем, и его простота вызывает определенные проблемы. В отличие от других алгоритмов, результаты KNN не интерпретируются, то есть можно спрогнозировать класс и входную точку данных, но нельзя понять, почему она принадлежит этому классу. Предположим, мы обучаем модель KNN, которая прогнозирует принадлежность учащегося старших классов к одной из десяти возможных социальных групп. Даже при точной модели мы все равно не сможем понять, почему этот студент классифицирован как спортсмен, а не как член певческого клуба. Чуть позже мы встретим алгоритмы, которые можно использовать, чтобы лучше понять связи признаков данных и класса.

К тому же KNN хорошо работает только при малом количестве признаков. По мере увеличения их числа в данные начинает проникать потенциально излишняя информация, в результате чего меры расстояния становятся менее надежными и качество прогнозов страдает. К счастью, излишество признаков можно частично нивелировать с помощью приемов уменьшения размерности, представленных

в главе 14. Но даже при верном их применении огромные наборы признаков все равно могут снизить точность прогнозов.

Наконец, самой большой проблемой KNN является скорость. При больших объемах обучающей выборки этот алгоритм может стать очень медленным. Предположим, мы создаем обучающую выборку с миллионом размеченных цветов. В наивной форме поиск ближайших соседей неразмеченного цветка потребует просканировать расстояние от него до каждого из миллиона других цветов, на что уйдет уйма времени. Естественно, можно выполнить оптимизацию под скорость, упорядочив обучающие данные более эффективно. Процесс этот аналогичен упорядочению слов в словаре. Представьте, что мы хотим найти слово *data* в словаре, не выстроенном по алфавиту. Поскольку слова хранятся в нем вразнобой, придется сканировать каждую страницу. На такой поиск в Оксфордском словаре, содержащем 6000 страниц, уйдет очень много времени. К счастью, все словари составляются по алфавиту, поэтому можно быстро найти искомое слово, выполнив серию несложных шагов. Сначала мы открываем словарь в середине, где на странице 3000 встречаем буквы *M* и *N*. Затем можно перелистнуть его до середины первой половины, оказавшись на странице 1500, где должны находиться слова, начинающиеся с *D*. Теперь мы уже гораздо ближе к цели. Повторение этого процесса несколько раз в итоге приведет нас к слову *data*.

Аналогичным образом можно быстро просканировать ближайших соседей, если сначала упорядочить обучающую выборку по расстоянию. В scikit-learn для сохранения расположенных рядом точек близко друг к другу применяется специальная структура данных, называемая *K-D-деревом*. Это позволяет ускорить сканирование и поиск соседей. Подробный разбор K-D-деревьев выходит за рамки нашего пособия, но я рекомендую почитать книгу по этой теме: Марчелло Ла Рокка (Marcello La Rocca), *Advanced Algorithms and Data Structures*[1] (Manning, 2021, www.manning.com/books/algorithms-and-data-structures-in-action). Несмотря на встроенную оптимизацию поиска, как уже говорилось, при больших размерах обучающей выборки KNN все равно будет работать медленно. Особой проблемой замедление становится во время оптимизации гиперпараметров. Чтобы продемонстрировать его наглядно, мы увеличим число элементов в обучающей выборке (`X, y`) в 2000 раз, после чего измерим время выполнения поиска по сетке для расширенных данных (листинг 20.48).

ВНИМАНИЕ

Код, представленный далее, будет выполняться долго.

Листинг 20.48. Оптимизация KNN при огромной обучающей выборке

```
import time
X_large = np.vstack([X for _ in range(2000)])
y_large = np.hstack([y for _ in range(2000)])
```

[1] *Ла Рокка М.* Продвинутые алгоритмы и структуры данных. — СПб.: Питер, 2024.

```
clf_grid = GridSearchCV(KNeighborsClassifier(), hyperparams,
                        scoring='f1_macro')
start_time = time.time()
clf_grid.fit(X_large, y_large)
running_time = (time.time() - start_time) / 60
print(f"The grid search took {running_time:.2f} minutes to run.")

The grid search took 16.23 minutes to run.
```

Поиск по сетке занял более 16 мин. Столь длительное выполнение недопустимо. Нам нужно альтернативное решение. В следующей главе мы познакомимся с новыми классификаторами, чье время выполнения прогнозов не зависит от размера обучающей выборки. Сначала мы сами разработаем эти классификаторы, ориентируясь на здравый смысл, а затем используем их реализации из scikit-learn.

РЕЗЮМЕ

- В *машинном обучении с учителем* наша цель — найти сопоставление между входными характеристиками данных, называемыми *признаками*, и выходными категориями — *классами*. Модель, идентифицирующая классы на основе признаков, называется *классификатором*.
- Для построения классификатора нам нужен набор данных с признаками и размеченными классами. Он называется *обучающей выборкой*.
- Одним из простейших классификаторов является метод *K ближайших соседей* (KNN). Он может классифицировать неразмеченные точки данных, выбирая среди множества классов *K* ближайших размеченных точек обучающей выборки. По сути, эти соседи голосуют в пользу присваивания точке того или иного класса. При желании голосование можно взвесить на основе расстояния между неразмеченной точкой данных и ее соседями.
- Эффективность классификатора можно оценить путем вычисления *матрицы несоответствий* `M`. Каждый элемент по диагонали `M[i][i]` отражает количество точно спрогнозированных экземпляров класса `i`. Точные прогнозы называются *истинно положительными* экземплярами класса. Доля от общего числа элементов вдоль диагонали матрицы `M` отражает *показатель общей точности* (accuracy).
- Элементы, спрогнозированные как принадлежащие классу A, но по факту принадлежащие классу B, называются *ложноположительными* класса A. Деление числа истинно положительных на сумму истинно положительных и ложноположительных дает параметр, называемый *точностью* (precision). Низкая точность говорит о том, что спрогнозированная метка класса не очень надежна.
- Фактические элементы класса A, спрогнозированные как принадлежащие классу B, называются *ложноотрицательными* класса A. Деление числа истинно

положительных на сумму истинно положительных и ложноотрицательных дает параметр, называемый *полнотой* (recall). Низкая полнота показывает, что наш предиктор часто ошибается в присваивании класса.

- Хороший классификатор должен давать высокие значения как точности, так и полноты. Их можно совместить в единый параметр, называемый *F-мерой*. При наличии точности `p` и полноты `r` F-мера вычисляется выполнением `2 * p * r / (p + r)`. Несколько показателей F-меры среди нескольких классов можно усреднить в один.
- Иногда F-мера может превосходить по значимости общую точность, особенно в случае несбалансированных данных, поэтому является предпочтительным параметром оценки.
- Для оптимизации эффективности KNN нужно выбрать оптимальное значение *K*. При этом также нужно решить, будет ли использоваться взвешенное голосование. Эти два параметризуемых входных значения называются *гиперпараметрами* и должны устанавливаться перед началом обучения. Все модели машинного обучения имеют гиперпараметры, которые можно подстраивать для повышения эффективности прогнозирования.
- Простейшей техникой оптимизации гиперпараметров является *поиск по сетке*, который выполняется перебором всех возможных комбинаций гиперпараметров. Перед началом перебора исходный набор данных разделяется на обучающую и валидационную выборки. Такое разделение называется *кросс-валидацией*. После ее выполнения мы перебираем все параметры и в завершение выбираем значения гиперпараметров, дающие наивысшее значение оценочного показателя.

21 Обучение линейных классификаторов с помощью логистической регрессии

В этой главе

- ✓ Разделение классов данных с помощью простых линейных срезов.
- ✓ Что такое логистическая регрессия.
- ✓ Обучение линейных классификаторов с помощью scikit-learn.
- ✓ Трактовка связи между прогнозом класса и параметрами обученного классификатора.

Классификацию данных, подобно кластеризации, можно рассматривать как геометрическую задачу, поскольку здесь размеченные классы аналогичным образом кластеризуются в абстрактном пространстве. Измеряя расстояние между точками данных, можно определять, какие из них принадлежат к одному кластеру, или классу. Однако, как вы узнали в предыдущей главе, вычисление этого расстояния может быть затратным. К счастью, связанные классы можно находить без измерения расстояния между всеми точками, и ранее мы это уже проделывали, когда в главе 14 анализировали клиентов магазина одежды. Каждый клиент был представлен двумя признаками: ростом и весом. Их графическая визуализация дала нам сигаровидный паттерн. Эту сигару мы повернули набок и разделили вертикальными линиями на три сегмента, представляющих три класса клиентов: малорослых, среднего роста и высоких.

Различные классы данных можно отделять отрезанием, как при использовании ножа. Выполняется оно с помощью простых линейных срезов. До этого мы

ограничивались вертикальными срезами сверху вниз. В текущей же главе научимся резать данные под углом для максимального разделения классов. Посредством таких направленных срезов можно классифицировать данные без вычисления расстояний. В ходе этого мы узнаем, как обучать и трактовать линейные классификаторы. Начнем же с возвращения к задаче деления клиентов по размеру одежды.

21.1. ЛИНЕЙНОЕ ДЕЛЕНИЕ КЛИЕНТОВ ПО РАЗМЕРУ ОДЕЖДЫ

В главе 14 мы симулировали рост клиентов (в дюймах) и вес (в фунтах). Клиенты большого роста/веса (дюймы/фунты) попадали в класс «Большой». Теперь выполним эту симуляцию еще раз (листинг 21.1). Рост и вес сохранены в матрице признаков `X`, а классы клиентов — в размеченном массиве `y`. В целях этой задачи мы сосредоточимся на двух классах: «Большой» и «Небольшой». Предположим, что клиенты, относящиеся к классу «Большой», имеют рост выше 72 дюймов и вес более 160 фунтов. После симуляции этих данных мы построим точечный график для `X`, в котором точки данных будут закрашены на основе их меток классов в `y` (рис. 21.1). Визуальное представление поможет обнаружить пространственное разделение между типами клиентов.

Листинг 21.1. Симуляция разделенных на категории размеров клиентов

```
import matplotlib.pyplot as plt
import numpy as
np np.random.seed(1)

def plot_customers(X, y, xlabel='Inches (in)', ylabel='Pounds (lb)'):
    colors = ['g', 'y']
    labels = ['Not Large', 'Large']
    for i, (color, label) in enumerate(zip(colors, labels)):
        plt.scatter(X[:,0][y == i], X[:,1][y == i], color=color, label=label)

    plt.xlabel(xlabel)
    plt.ylabel(ylabel)

inches = np.arange(60, 78, 0.1)
random_fluctuations = np.random.normal(scale=10, size=inches.size)
pounds = 4 * inches - 130 + random_fluctuations
X = np.array([inches, pounds]).T
y = ((X[:,0] > 72) & (X[:,1] > 160)).astype(int)

plot_customers(X, y)
plt.legend()
plt.show()
```

Строит график размеров клиентов, закрашивая их на основе принадлежности к тому или иному классу. Рост и вес рассматриваются как два разных признака в матрице признаков X. Тип класса клиентов хранится в размеченном массиве y

Клиенты подразделяются на два класса: «Большой» и «Небольшой»

Следует линейной формуле из главы 14, моделируя вес как функцию от роста

Клиент считается большим, если он выше 72 дюймов, а его вес больше 160 фунтов

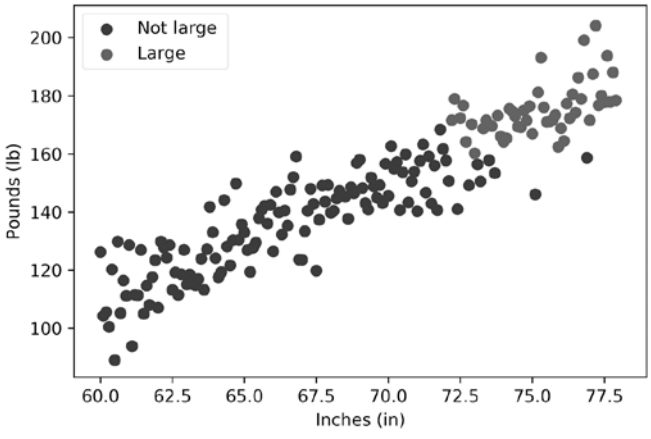


Рис. 21.1. График размеров клиентов: дюймы и фунты. «Большие» и «небольшие» клиенты закрашены по-разному на основе своего класса

Полученный график напоминает сигару с двумя концами разного цвета. Теперь можно представить нож, разрезающий ее в месте разделения по цвету (листинг 21.2). Полученный срез выступает как граница, разделяющая два класса клиентов. Представить ее можно с помощью линии, имеющей наклон –3,5 и пересекающей ось *Y* в точке 415. Формула данной линии — `lbs = -3.5 * inches + 415`. Добавим эту линейную границу на график (рис. 21.2).

ПРИМЕЧАНИЕ

В следующей главе вы научитесь вычислять линейную границу автоматически.

Листинг 21.2. Построение границы для разделения двух классов клиентов

```
def boundary(inches): return -3.5 * inches + 415
plt.plot(X[:,0], boundary(X[:,0]), color='k', label='Boundary')
plot_customers(X, y)
plt.legend()
plt.show()
```

Построенная линия называется *линейной решающей границей*, поскольку ее можно использовать для точного выбора класса клиента. Большинство клиентов в классе «Большой» расположены выше линии. Рассматривая клиента с размерами `(inches, lbs)`, мы прогнозируем его класс, проверяя на истинность неравенство `lbs > -3.5 * inches + 415` (листинг 21.3). В случае его истинности клиент относится к классу «Большой». Далее мы применим это неравенство для прогнозирования классов клиентов. Прогнозы сохраним в массиве `y_pred` и оценим их с помощью F-меры.

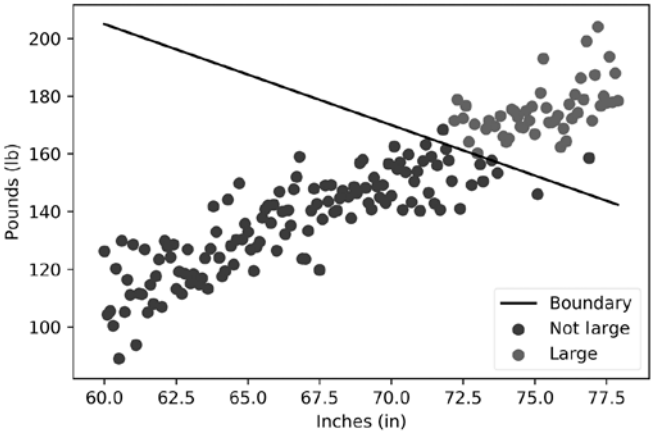


Рис. 21.2. График размеров клиентов: дюймы и фунты. Линейная граница разделяет «больших» и «небольших» клиентов

ПРИМЕЧАНИЕ

Как говорилось в главе 20, F-мера — это предпочтительный способ оценки эффективности прогнозирования классов. Напомню, что она равна среднему гармоническому точности и полноты классификатора.

Листинг 21.3. Прогнозирование классов с помощью линейной границы

```
from sklearn.metrics import f1_score
y_pred = []
for inches, lbs in X:
    prediction = int(lbs > -3.5 * inches + 415)
    y_pred.append(prediction)

f_measure = f1_score(y_pred, y)
print(f'The f-measure is {f_measure:.2f}')

The f-measure is 0.97
```

Если b является логическим значением Python, int(b) возвращает 1, когда оно True, и 0 — в противном случае. Таким образом, можно вернуть метку класса для размеров (inches, lbs), выполнив int(lbs > –3.5 * inches + 415)

Как и ожидалось, F-мера получилась высокой. При неравенстве `lbs > -3.5 * inches + 415` можно точно классифицировать наши данные. Более того, классификацию можно выполнить более сжато, используя скалярное произведение векторов. Рассмотрите такой алгоритм.

1. Наше неравенство перестраивается в `3.5 * inches + lbs - 415 > 0`.
2. Скалярное произведение векторов `[x, y, z]` и `[a, b, c]` равно `a * x + b * y + c * z`.
3. Если получить скалярное произведение векторов `[inches, lbs, 1]` и `[3.5, 1, -415]`, результат окажется равен `3.5 * inches + lbs - 415`.
4. Таким образом, неравенство сократилось до `w @ v > 0`, где `w` и `v` являются векторами, а `@` — оператором скалярного произведения, как показано на рис. 21.3.

Заметьте, что только один из векторов зависит от значений `lbs` и `inches`. На второй вектор, `[3.5, 1, -415]`, размеры клиентов уже не влияют. Аналитики данных называют этот инвариантный вектор *весовым вектором* или просто *весом*.

ПРИМЕЧАНИЕ

Это название не связано с весом наших клиентов в фунтах.

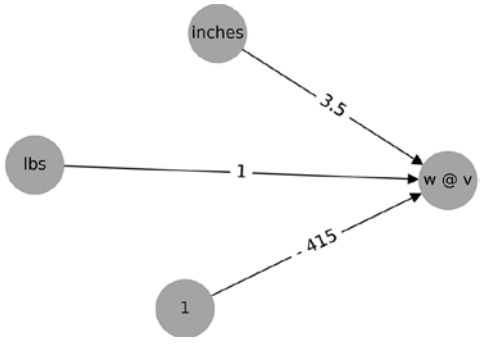


Рис. 21.3. Скалярное произведение между weights и [inches, lbs, 1] можно визуализировать в виде направленного графа. В этом графе крайние левые узлы представляют размеры [inches, lbs, 1], а веса ребер выражают веса [3,5, 1, –415]. Каждый узел мы умножаем на соответствующее ему ребро и суммируем полученные результаты. Эта сумма равна произведению между векторами v и w. Классификация клиента определяется верностью неравенства w @ v > 0

Используя скалярное произведение векторов, мы воссоздадим содержимое `y_pred` в двух строках кода (листинг 21.4).

1. Установим вектор `weights` равным `[3.5, 1, -415]`.
2. Классифицируем каждого клиента в `X` по его размерам `(inches, lbs)`, применяя скалярное произведение `weights` и `[inches, lbs, 1]`.

Листинг 21.4. Прогнозирование классов с помощью скалярного произведения векторов

```
weights = np.array([3.5, 1, -415])
predictions = [int(weights @ [inches, lbs, 1] > 0) for inches, lbs in X]
assert predictions == y_pred
```

Этот код можно еще больше упростить, если задействовать матричное произведение.

Рассмотрите такой вариант.

1. Сейчас нам необходимо перебирать каждый ряд `[inches, lbs]` в матрице `X` и добавлять 1 для получения вектора `[inches, lbs, 1]`.

2. Вместо этого можно внести в матрицу `X` столбец единиц, получив трехколоночную матрицу `M`, каждая строка которой будет представлена как `[inches, lbs, 1]`. Эту матрицу `M` мы назовем *дополненной матрицей признаков*.
3. Выполнение `[weights @ v for v in M]` ведет к возвращению скалярных произведений между `weights` и каждой строкой `M`. Естественно, эта операция равнозначна выполнению матричного умножения `M` на `weights`.
4. Это матричное произведение можно вычислить сжато, выполнив `M @ weights`.
5. Выполнение `M @ weights > 0` вернет массив логических значений, каждый элемент которого равен `true`, только если `3.5 * inches + lbs - 415 > 0` для соответствующих размеров клиента.

По сути, `M @ weights > 0` возвращает логический вектор, чье значение `i` равно `true`, если `y_pred[i] == 1`, и `false` — в противном случае. Эти логические значения мы преобразуем в численные метки NumPy методом `astype`, после чего можно будет сгенерировать прогнозы выполнением `(M @ weights > 0).astype(int)`. Проверим это (листинг 21.5).

Листинг 21.5. Прогнозирование классов с помощью матричного умножения

```
M = np.column_stack([X, np.ones(X.shape[0])])
print("First five rows of our padded feature matrix are:")
print(np.round(M[:5], 2))

predictions = (M @ weights > 0).astype(int)
assert predictions.tolist() == y_pred

First five rows of our padded feature matrix are:
[[ 60. 126.24 1. ]
 [ 60.1 104.28 1. ]
 [ 60.2 105.52 1. ]
 [ 60.3 100.47 1. ]
 [ 60.4 120.25 1. ]]
```

Добавляет в матрицу X столбец единиц для создания трехколоночной матрицы M

Проверяет, остаются ли наши прогнозы прежними. Заметьте, что матричное умножение возвращает массив NumPy, который для этого сравнения необходимо преобразовать в список

Мы сократили классификацию клиентов до простого умножения матрицы на вектор. Такой классификатор на основе матричного умножения называется *линейным классификатором*. Ему для отнесения входных признаков к определенным категориям нужен лишь вектор весов. Здесь мы определяем функцию `linear_classifier`, которая получает матрицу признаков `X` и вектор весов `weights`, возвращая массив прогнозов классов (листинг 21.6).

Листинг 21.6. Определение функции линейной классификации

```
def linear_classifier(X, weights):
    M = np.column_stack([X, np.ones(X.shape[0])])
    return (M @ weights > 0).astype(int)

predictions = linear_classifier(X, weights)
assert predictions.tolist() == y_pred
```

Линейные классификаторы проверяют, дают ли взвешенные признаки и константа в сумме значение больше нуля. Константа, которая хранится в `weights[-1]`, называется *смещением*, а оставшиеся веса — *коэффициентами*. Во время классификации каждый коэффициент умножается на соответствующий ему признак. В нашем случае коэффициент `inches` в `weights[0]` умножается на `inches`, а коэффициент `lbs` в `weights[1]` — на `lbs`. Таким образом, `weights` содержит два коэффициента и одно смещение, принимая форму `[inches_coef, lbs_coef, bias]`.

Вектор `weights` мы получили с помощью известной решающей границы, но его также можно вычислить непосредственно из нашей обучающей выборки `(X, y)`. В следующем разделе мы рассмотрим обучение линейного классификатора, состоящее из нахождения коэффициентов и смещения, которые линейно разделяют наши классы клиентов.

ПРИМЕЧАНИЕ

Результаты обучения не соответствуют weights, потому что неравенству M @ weights > 0 удовлетворяет бесчисленное множество векторов weights. Подтвердить это можно умножением обеих сторон на положительную константу k. Естественно, 0 * k равно 0. При этом weights * k дает новый вектор w2. Таким образом, M @ w2 больше 0, если M @ weights > 0, и наоборот. Существует бесконечное число констант k, а значит, и бесчисленное множество векторов w2, но все они указывают в одном направлении.

21.2. ОБУЧЕНИЕ ЛИНЕЙНОГО КЛАССИФИКАТОРА

Нам нужно найти вектор весов, оптимизирующий прогнозирование классов в `X`. Начнем с установки `weights` равными трем случайным значениям. После этого вычислим F-меру, связанную с этим случайным вектором (листинг 21.7). Ожидается, что ее значение будет очень низким.

Листинг 21.7. Классификация с помощью случайных весов

```
np.random.seed(0)
weights = np.random.normal(size=3)
y_pred = linear_classifier(X, weights)
f_measure = f1_score(y_pred, y)

print('We inputted the following random weights:')
print(np.round(weights, 2))
print(f'\nThe f-measure is {f_measure:.2f}')

We inputted the following random weights:
[1.76 0.4 0.98]

The f-measure is 0.43
```

Как и ожидалось, F-мера получилась ужасной. Чтобы понять, почему так произошло, выведем `y_pred` (листинг 21.8).

Листинг 21.8. Вывод спрогнозированных классов

```
print(y_pred)

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

Всем точкам данных были присвоены метки класса 1. Произведение весов и каждого вектора признаков всегда больше нуля, значит, наши веса слишком высоки. Их уменьшение приведет к большему числу прогнозов класса 0. К примеру, если установить веса равными `[0, 0, 0]`, все прогнозы классов окажутся равными 0 (листинг 21.9).

Листинг 21.9. Сдвиг прогнозов классов путем уменьшения весов

```
assert np.all(linear_classifier(X, [0, 0, 0]) == 0)
```

Уменьшение весов ведет к увеличению числа прогнозов класса 0, а их повышение — к увеличению прогнозов класса 1. Таким образом, можно сознательно поднимать и снижать веса, пока наши прогнозы не совпадут с фактическими метками классов. Далее мы разработаем стратегию корректировки весов, чтобы получить соответствие меткам, и начнем с подстройки смещения в `weights[-1]`.

ПРИМЕЧАНИЕ

Подстройка коэффициентов — более тонкий процесс, поэтому начнем со смещения.

Наша цель — минимизировать различия между прогнозами в `y_pred` и фактическими метками в `y`. Как этого добиться? В качестве простого решения достаточно выполнить сравнение внутри каждой пары меток *спрогнозированного/фактического* классов. На основе результатов этого сравнения смещение можно скорректировать так.

- Если прогноз совпадает с фактическим классом, значит, он верен, смещение не трогаем.
- Если прогноз равен 1, а фактический класс представлен 0, значит, вес слишком высок, уменьшаем смещение на единицу.
- Если прогноз равен 0, а фактический класс представлен 1, значит, вес слишком мал, увеличиваем смещение на единицу.

Далее мы определим функцию для вычисления этого сдвига смещения на основе спрогнозированных и фактических меток (листинг 21.10).

ПРИМЕЧАНИЕ

Помните, что, по существующим соглашениям, сдвиг смещения вычитается из весов. Значит, наша функция get_bias_shift возвращает положительное значение, когда веса должны уменьшаться.

Листинг 21.10. Вычисление сдвига смещения на основе качества прогноза

```
def get_bias_shift(predicted, actual):
    if predicted == actual:
        return 0
    if predicted > actual:
        return 1

    return -1
```

Математически можно показать, что функция `get_bias_shift` равнозначна `predicted – actual`. Код листинга 21.11 определенно подтверждает это для всех четырех комбинаций меток спрогнозированных и фактических классов.

Листинг 21.11. Вычисление сдвига смещения с помощью арифметики

```
for predicted, actual in [(0, 0), (1, 0), (0, 1), (1, 1)]:
    bias_shift = get_bias_shift(predicted, actual)
    assert bias_shift == predicted - actual
```

Стоит отметить, что величина сдвига смещения на единицу была выбрана произвольно. Вместо этого его можно сдвигать на 0,1 единицы, 10 единиц или даже 100. Величина сдвига регулируется параметром *«скорость обучения»* (learning rate). Для подстройки величины смещения скорость обучения умножается на `predicted – actual`. Значит, если мы хотим уменьшить смещение на 0,1, то можем просто выполнить `learning_rate * (predicted – actual)`, где `learning_rate` равна 0,1 (листинг 21.12). Эта корректировка может повлиять на качество обучения, в связи с чем мы переопределим функцию `get_bias_shift`, установив в ней параметр `learning_rate` равным 0,1.

Листинг 21.12. Вычисление сдвига смещения с использованием скорости обучения

```
def get_bias_shift(predicted, actual, learning_rate=0.1):
    return learning_rate * (predicted - actual)
```

Теперь можно корректировать смещение. Код листинга 21.13 перебирает каждый вектор `[inches, lbs, 1]` в `M`. Для каждого *i*-го вектора мы прогнозируем метку класса и сравниваем ее с фактическим классом в `y[i]`.

ПРИМЕЧАНИЕ

Напомню, что прогнозирование класса для каждого вектора v равно int(v @ weights > 0).

Используя каждый прогноз, мы вычисляем сдвиг смещения и вычитаем его из смещения, хранящегося в `weights[-1]`. По завершении всех итераций выводим скорректированное смещение и сравниваем его с исходным значением.

Листинг 21.13. Итеративный сдвиг смещения

```
def predict(v, weights): return int(v @ weights > 0)

starting_bias = weights[-1]
for i, actual in enumerate(y):
    predicted = predict(M[i], weights)
    bias_shift = get_bias_shift(predicted, actual)
    weights[-1] -= bias_shift

new_bias = weights[-1]
print(f"Our starting bias equaled {starting_bias:.2f}.")
print(f"The adjusted bias equals {new_bias:.2f}.")

Our starting bias equaled 0.98.
The adjusted bias equals -12.02
```

Прогнозирует метку класса для вектора v, связанного со строкой в матрице M

Смещение значительно уменьшилось, и это вполне логично, учитывая, что веса были слишком большими. Теперь проверим, улучшил ли этот сдвиг F-меру (листинг 21.14).

Листинг 21.14. Проверка эффективности модели после сдвига смещения

```
y_pred = linear_classifier(X, weights)
f_measure = f1_score(y_pred, y)
print(f'The f-measure is {f_measure:.2f}')

The f-measure is 0.43
```

F-мера осталась прежней. Простой подстройки смещения оказалось недостаточно. Необходимо также подстроить коэффициенты, но как? Первым на ум приходит наивный подход с вычитанием сдвига смещения из каждого коэффициента. Достаточно просто перебрать каждый обучающий образец и выполнить `weights-= bias_shift`. К сожалению, такой подход имеет серьезный недостаток. Он всегда корректирует коэффициенты, но делать это опасно, когда связанные с ними признаки равны нулю. Причину этого проиллюстрируем на простом примере.

Представьте, что пустая запись в нашем наборе данных клиентов ошибочно записана как `(0, 0)`. Модель рассматривает эту точку данных как клиента, чей вес и рост равны нулю. Естественно, такого клиента физически существовать не может, но это к делу не относится. Такой гипотетический клиент определенно будет «небольшим», а значит, верным классом для него окажется `0`. При классификации этого клиента наша линейная модель получает скалярное произведение `[0, 0, 1]` и `[inches_coef, lbs_coef, bias]`. Конечно же, эти коэффициенты умножаются на ноль и утрачиваются, в результате чего скалярное произведение оказывается равным одному `bias` (рис. 21.4). Если `bias > 0`, классификатор ошибочно присваивает клиенту метку класса `1`. Здесь нужно уменьшить смещение с помощью `bias_shift`. Будем ли мы также корректировать коэффициенты? Нет. Наши коэффициенты на полученный прогноз не влияли, и их качество мы оценить не можем. Известно лишь то, что они установлены на свои оптимальные значения. Если так, то вычитание из них сдвига смещения только ухудшит модель.

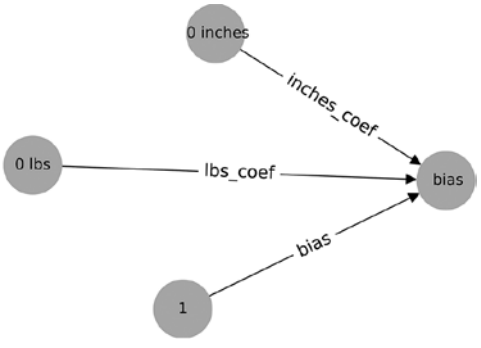


Рис. 21.4. Можно визуализировать скалярное произведение между weights и [0, 0, 1] в виде направленного графа. В нем узлы слева представляют признаки с нулевым значением, а веса ребер — коэффициенты и смещение. Мы умножаем каждый узел на вес соответствующего ему ребра и суммируем результаты. Полученная сумма равна bias. Классификация клиентов определяется неравенством bias > 0. Коэффициенты не влияют на прогноз, а значит, изменяться не должны

Никогда не следует сдвигать `lbs_coef`, если признак `lbs` равен нулю. Однако для ненулевых входных данных вычитание `bias_shift` из `lbs_coef` продолжает быть актуальным. Гарантировать это можно, установив сдвиг `lbs_coef` равным `bias_shift if lbs else 0`. В качестве альтернативы можно установить его равным `bias_shift * lbs`. Это произведение равно нулю, когда нулю равен `lbs`. В ином случае произведение сдвигает `lbs_coef` в направлении смещения. Аналогичным образом можно сдвинуть `inches_coef` на `bias_shift * inches` единиц. Иными словами, мы сдвинем каждый коэффициент на произведение его признака и `bias_shift`.

NumPy позволяет вычислять все сдвиги весов одновременно посредством выполнения `bias_shift * [inches, lbs, 1]`. Естественно, вектор `[inches, lbs, 1]` соответствует строке в дополненной матрице признаков `M`. Таким образом, мы можем скорректировать веса на основе каждого *i*-го прогноза, выполнив `weights -= bias_shift * M[i]`.

С учетом этого мы переберем каждую фактическую метку в `y` и скорректируем веса на основе спрогнозированных значений. После этого еще раз проверим, улучшился ли показатель F-меры (листинг 21.15).

Листинг 21.15. Вычисление всех сдвигов весов в одной строке кода

```
old_weights = weights.copy()
for i, actual in enumerate(y):
    predicted = predict(M[i], weights)
    bias_shift = get_bias_shift(predicted, actual)
    weights -= bias_shift * M[i]

y_pred = linear_classifier(X, weights)
f_measure = f1_score(y_pred, y)
```

```
print("The weights previously equaled:")
print(np.round(old_weights, 2))
print("\nThe updated weights now equal:")
print(np.round(weights, 2))
print(f'\nThe f-measure is {f_measure:.2f}')

The weights previously equaled:
[ 1.760.4  -12.02]

The updated weights now equal:
[ -4.642.22 -12.12]

The f-measure is 0.78
```

В ходе этой итерации `inches_coef` уменьшился на 6,39 единицы (с 1,76 до –4,63), а смещение уменьшилось на 0,1 единицы (с –12,02 до –12,12). Столь значительное расхождение вполне логично, поскольку сдвиг коэффициента пропорционален росту. Средний рост клиентов составляет 64 дюйма, значит, сдвиг коэффициента в 64 раза больше смещения. Как мы вскоре увидим, большие различия в сдвигах весов могут привести к проблемам. Позднее мы устраним их с помощью процесса, называемого *стандартизацией*, но сначала вернемся к F-мере.

Ее значение увеличилось с 0,43 до 0,78. Наша стратегия сдвига весов работает! А что произойдет, если эту итерацию повторить 1000 раз? Сейчас выясним. Код листинга 21.16 отслеживает изменения F-меры в течение 1000 итераций сдвига весов, после чего мы визуализируем каждую *i*-ю F-меру относительно *i*-й итерации (рис. 21.5). Полученный график используем для отслеживания улучшения эффективности классификатора во времени.

ПРИМЕЧАНИЕ

В этом упражнении мы устанавливаем веса на их исходные случайные значения. Это позволит отследить повышение эффективности относительно стартовой F-меры, равной 0,43.

Листинг 21.16. Корректировка весов в течение множества итераций

```
np.random.seed(0)
weights = np.random.normal(size=3)

f_measures = []
for _ in range(1000):
    y_pred = linear_classifier(X, weights)
    f_measures.append(f1_score(y_pred, y))
    for i, actual in enumerate(y):
        predicted = predict(M[i], weights)
        bias_shift = get_bias_shift(predicted,
        weights -= bias_shift * M[i]

print(f'The f-measure after 1000 iterations is {f_measures[-1]:.2f}') plt.
plot(range(len(f_measures)), f_measures)
plt.xlabel('Iteration')
```

Устанавливает стартовые веса на случайные значения

Повторяет логику сдвига весов в течение 1000 итераций

Отслеживает эффективность весов при каждой итерации

Сдвигает веса путем перебора всех пар меток спрогнозированных/фактических классов

```
plt.ylabel('F-measure')
plt.show()

The f-measure after 1000 iterations is 0.68
```

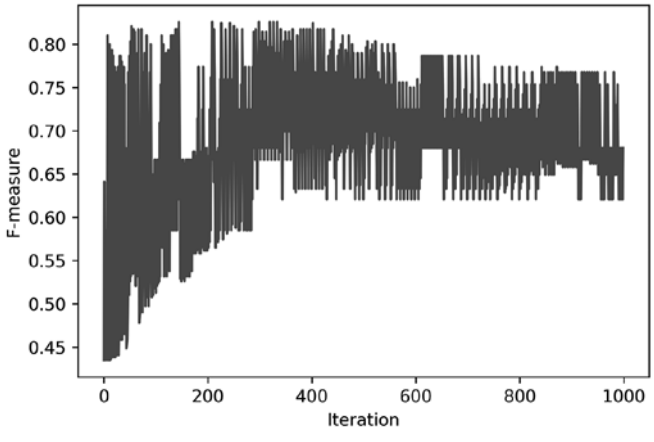


Рис. 21.5. График итераций относительно F-меры модели. На каждой итерации веса модели подстраиваются. F-мера широко колеблется между низкими и адекватными значениями. Эти колебания необходимо устранить

Итоговая F-мера оказалась равна 0,68. Наш классификатор очень плохо обучен. Что же произошло? Если верить графику, то эффективность классификатора в ходе итераций сильно колеблется. Иногда F-мера подскакивает до 0,8, а иногда падает примерно до 0,6. После примерно 400 итераций классификатор начинает непрерывно колебаться между этими двумя значениями. Столь резкие колебания вызваны сдвигом весов, которые постоянно слишком высоки. Это можно сравнить с самолетом, который летит слишком быстро. Представьте, что после взлета он перемещается со скоростью 600 миль/ч. Самолет сохраняет ее, что позволяет ему пролететь за 3 ч 1500 миль. Однако по мере приближения к точке назначения пилот отказывается сбросить скорость, и самолет пролетает мимо нужного аэропорта, в связи с чем вынужден возвращаться обратно. Если пилот на скорректирует скорость, его судно продолжит промахиваться мимо места посадки, что приведет к серии бесконечных U-образных разворотов, аналогичных колебаниям на нашем графике. Для пилота решение довольно простое — нужно просто уменьшить скорость в нужный момент полета.

Наше решение будет аналогичным — мы станем медленно уменьшать сдвиг весов с каждой дополнительной итерацией. А как это сделать? Один из вариантов подразумевает деление сдвига на `k` для каждой k-й итерации. Применим эту стратегию. Мы сбросим веса на случайные значения и переберем значения `k` в диапазоне от 1 до 1001. На каждой итерации будем устанавливать сдвиг весов равным `bias_shift * M[i] / k`, после чего еще раз сгенерируем график эффективности модели (листинг 21.17; рис. 21.6).

Листинг 21.17. Уменьшение сдвига весов в течение множества итераций

```
np.random.seed(0)
def train(X, y,
          predict=predict):
    M = np.column_stack([X, np.ones(X.shape[0])])
    weights = np.random.normal(size=X.shape[1] + 1)
    f_measures = []
    for k in range(1, 1000):
        y_pred = linear_classifier(X, weights)
        f_measures.append(f1_score(y_pred, y))

        for i, actual in enumerate(y):
            predicted = predict(M[i], weights)
            bias_shift = get_bias_shift(predicted, actual)
            weights -= bias_shift * M[i] / k
        return weights, f_measures

weights, f_measures = train(X, y)
print(f'The f-measure after 1000 iterations is {f_measures[-1]:.2f}')
plt.plot(range(len(f_measures)), f_measures)
plt.xlabel('Iteration')
plt.ylabel('F-measure')
plt.show()

The f-measure after 1000 iterations is 0.82
```

Обучает линейную модель на признаках X и метках y. Эта функция используется и в других частях главы

Функция predict управляет сдвигом весов, позволяя сравнивать спрогнозированные и фактические классы. Позже в этой главе мы изменим predict для внесения в сдвиг весов дополнительных нюансов

Модель с N признаками имеет всего N + 1 весов, представляющих N коэффициентов и одно смещение

На каждой k-й итерации мы ослабляем сдвиг весов путем деления на k, тем самым уменьшая связанные с этим колебания

Возвращает оптимизированные веса вместе с прослеженными показателями эффективности в течение 1000 итераций

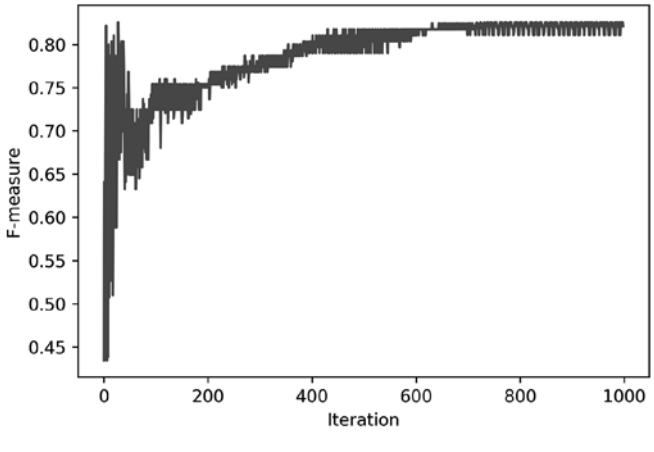


Рис. 21.6. График итераций относительно F-меры модели. Веса модели корректируются на каждой k-й итерации пропорционально 1/k. Деление сдвига весов на k ограничивает колебания, и F-мера сходится к разумному значению

Постепенное уменьшение сдвига весов оказалось успешным. F-мера сходится к устойчивому значению 0,82. Этого мы добились при помощи *алгоритма перцептрона*. *Перцептрон* — это простой линейный классификатор, придуманный в 1950-х годах. Обучать такие классификаторы очень легко. Для этого нужно лишь выполнить для обучающей выборки `(X, y)` следующие шаги.

1. Добавить в матрицу признаков `X` столбец единиц, создав дополненную матрицу `M`.
2. Создать вектор `weights`, содержащий `M.shape[1]` случайных значений.
3. Перебрать каждую *i*-ю строку в `M` и спрогнозировать *i*-й класс, выполнив `M[i] @ weights > 0`.
4. Сравнить *i*-й прогноз с фактической меткой класса в `y[i]`, после чего вычислить сдвиг смещения, выполнив `(predicted – actual) * lr`, где `lr` — скорость обучения.
5. Скорректировать веса, выполнив `weights -= bias_shift * M[i] / k`. Изначально константа `k` устанавливается равной `1`.
6. Повторить шаги 3–5 в течение множества итераций, каждый раз увеличивая `k` на 1 для ограничения колебаний.

В процессе повторения перцептрон в конечном итоге сходится к устойчивой F-мере. Однако она не обязательно оптимальна. К примеру, наш перцептрон сошелся к ее значению 0,82. Это приемлемый уровень качества модели, но он не соответствует изначальному показателю 0,97. Обученная нами решающая граница не отделяет данные так же хорошо, как изначальная решающая граница.

Как сравнить эти две границы визуально? Довольно просто (листинг 21.18). С помощью алгебраических манипуляций можно преобразовать вектор весов `[inches_coef, lbs_coef, bias]` в линейную решающую границу, равную `lbs = -(inches_coef * inches + bias)/ lbs_coef`. С учетом этого мы построим график старой и новой решающих границ вместе с данными о клиентах (рис. 21.7).

Листинг 21.18. Сравнение новой и старой решающих границ

```
inches_coef, lbs_coef, bias = weights
def new_boundary(inches):
    return -(inches_coef * inches + bias) / lbs_coef

plt.plot(X[:,0], new_boundary(X[:,0]), color='k', linestyle='--',
         label='Trained Boundary', linewidth=2)

plt.plot(X[:,0], boundary(X[:,0]), color='k', label='Initial Boundary')
plot_customers(X, y)
plt.legend()
plt.show()
```

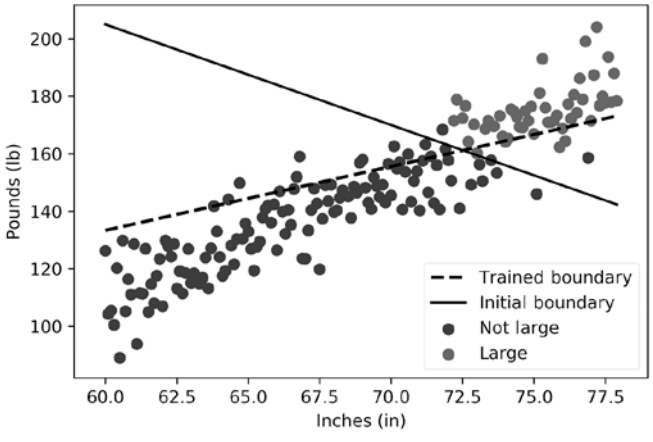


Рис. 21.7. График размеров клиентов — дюймы относительно фунтов. Две линейные границы разделяют «больших» и «небольших» клиентов. Обученное разделение оказывается хуже в сравнении с базовым

Обученная линейная граница уступает изначальной, но это не вина перцептрона. Напротив, обучению мешают большие колеблющиеся признаки в матрице `X`. В следующем разделе мы рассмотрим, почему большие значения `X` вредят эффективности модели, и ограничим их пагубное влияние с помощью процесса *стандартизации*, при которой `X` корректируется как равная `(X – X.mean(axis=0)) / X.std(axis=0)`.

21.2.1. Улучшение эффективности перцептрона с помощью стандартизации

Обучению перцептрона мешают большие значения признаков в `X`, и причина тому в расхождении между сдвигами коэффициентов и смещения. Как уже говорилось, сдвиг коэффициента пропорционален значению связанного с ним признака. Кроме того, эти значения могут быть довольно большими. К примеру, средний рост клиента превышает 60 дюймов — сдвиг `inches_coef` более чем в 60 раз больше сдвига смещения, поэтому никак не получится скорректировать его чуть-чуть, не изменив значительно коэффициенты. Таким образом, подстраивая смещение, мы вынуждены сильно сдвигать `inches_coef` в сторону неоптимального значения.

Нашему обучению недостает плавности, поскольку сдвиги коэффициентов излишне велики. Однако их можно сократить, уменьшив среднее столбцов в матрице `X`. Помимо этого, нам нужно уменьшить дисперсию в этой матрице. В противном случае необычайно большие размеры клиентов могут вызывать излишние сдвиги

коэффициентов. В связи с этим требуется уменьшить средние столбцов и стандартные отклонения. Для начала мы выведем текущие значения `X.mean(axis=0)` и `X.std(axis=0)` (листинг 21.19).

Листинг 21.19. Вывод средних значений признаков и стандартных отклонений

```
means = X.mean(axis=0)
stds = X.std(axis=0)
print(f"Mean values: {np.round(means, 2)}")
print(f"STD values: {np.round(stds, 2)}")

Mean values: [ 68.95 146.56]
STD values: [ 5.2 23.26]
```

Средние признаков и стандартные отклонения довольно высоки. Что же предпринять для их уменьшения? Как мы узнали в главе 14, сдвинуть среднее набора данных в сторону нуля нетрудно — достаточно вычесть `means` из `X`. А вот скорректировать стандартные отклонения уже сложнее, но математически можно показать, что `(X – means) / stds` возвращает матрицу, рассеяния всех столбцов в которой равны 1.

ПРИМЕЧАНИЕ

Вот доказательство. Выполнение X – means возвращает матрицу, среднее значение каждого столбца v который равно 0. Таким образом, дисперсия каждого v равна [e * e for e in v] / N, где N — количество элементов в столбце. Естественно, эту операцию можно выразить как простое скалярное произведение v @ v / N. Стандартное отклонение std равно квадратному корню из дисперсии, значит, std = sqrt(v @ v) / sqrt(N). Заметьте, что sqrt(v @ v) равно абсолютной величине v, которую можно выразить как norm(v). Таким образом, std = norm(v) / sqrt(N). Предположим, что делим v на std для генерации нового вектора v2. Поскольку v2 = v / std, ожидается, что абсолютная величина v2 будет равна norm(v) / std. Стандартное отклонение v2 равно norm(v2) / sqrt(N). Заменяя norm(v2), мы получаем norm(v) / (sqrt(N) * std). Однако norm(v) / sqrt(N) = std. Значит, стандартное отклонение v2 сокращается до std / std, что равно 1.

Этот простой процесс называется *стандартизацией*. Далее мы стандартизируем нашу матрицу признаков, выполнив `(X – means) / stds` (листинг 21.20). Полученная матрица будет содержать средние столбцов, равные 0, и стандартные отклонения столбцов, равные 1.

Листинг 21.20. Стандартизация матрицы признаков

```
def standardize(X):
    return (X - means) / stds
```

Стандартизирует размеры, полученные из распределения клиентов. Эта функция повторно используется в других частях главы

```
X_s = standardize(X)
assert np.allclose(X_s.mean(axis=0), 0)
assert np.allclose(X_s.std(axis=0), 1)
```

Теперь проверим, приведет ли обучение на стандартизированной матрице признаков к улучшению результатов (листинг 21.21). Мы также построим относительно этих стандартизированных данных обученную решающую границу (рис. 21.8).

Листинг 21.21. Обучение на стандартизированной матрице признаков

```
np.random.seed(0)
weights, f_measures = train(X_s, y)
print(f'After standardization, the f-measure is {f_measures[-1]:.2f}')

def plot_boundary(weights):
    a, b, c = weights
    new_boundary = lambda x: -(a * x + c) / b
    plt.plot(X_s[:,0], new_boundary(X_s[:,0]), color='k', linestyle='--',
        label='Trained Boundary', linewidth=2)
    plot_customers(X_s, y, xlabel='Standardized Inches',
                   ylabel='Standardized Pounds')
plt.legend()
plt.show()

plot_boundary(weights)

After standardization, the f-measure is 0.98
```

Строит график линейной решающей границы, полученной из весов, вместе со стандартизированными данными

Преобразует веса в линейную функцию

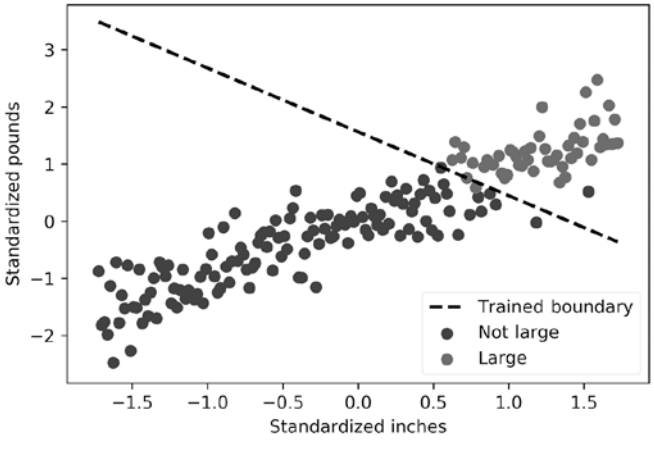


Рис. 21.8. График стандартизированных размеров клиентов. Обученная решающая граница разделяет «больших» и «небольших» клиентов. Это разделение обученной границей сопоставимо с результатом разделения базовой решающей границы на рис. 21.2

Сработало! Новая F-мера равна 0,98. Это ее значение выше базового, которое составляло 0,97. Более того, угол наклона новой решающей границы близок к углу

наклона базовой границы на рис. 21.2. Таким образом, мы добились улучшения эффективности модели путем стандартизации.

ПРИМЕЧАНИЕ

Стандартизация похожа на нормализацию. Обе эти техники уменьшают значения входных данных и устраняют различия в единицах измерения (например, между дюймами и сантиметрами). В некоторых задачах, таких как анализ главных компонент, они могут друг друга заменять. Однако при обучении линейных классификаторов стандартизация дает более качественные результаты.

Здесь нужно отметить, что обученный классификатор теперь требует предварительной стандартизации входных данных. Поэтому для классификации любых новых данных `d` нужно выполнять `linear_classifier(standardize(d), weights)` (листинг 21.22).

Листинг 21.22. Стандартизация новых входных данных для классификатора

```
new_data = np.array([[63, 110], [76, 199]])
predictions = linear_classifier(standardize(new_data), weights)
print(predictions)
```

```
[0 1]
```

Мы стандартизировали данные и достигли высокой эффективности классификации. К сожалению, обучающий алгоритм все равно не гарантирует получения оптимальной F-меры. Качество обучения перцептрона может колебаться, даже если этот алгоритм неоднократно выполняется для одной обучающей выборки. И дело здесь в случайных весах, присваиваемых на начальном этапе обучения: определенные стартовые веса сходятся к менее точной решающей границе. Далее мы продемонстрируем эту непоследовательность модели с помощью пятикратного обучения перцептрона (листинг 21.23). После каждого такого обучения будем проверять, стала ли полученная F-мера ниже базового показателя 0,97.

Листинг 21.23. Проверка последовательности качества обучения перцептрона

```
np.random.seed(0)
poor_train_count = sum([train(X_s, y)[1][-1] < 0.97 for _ in range(5)])
print("The f-measure fell below our baseline of 0.97 in "
      f"{poor_train_count} out of 5 training instances")
```

```
The f-measure fell below our baseline of 0.97 in 4 out of 5
training instances
```

В 80 % случаев эффективность обученной модели опустилась ниже базовой границы. Очевидно, что наша простая модель перцептрона имеет недостатки, о которых поговорим в следующем разделе. В ходе этого процесса мы выведем одну из самых популярных моделей в науке о данных — логистическую регрессию.

21.3. УЛУЧШЕНИЕ ЛИНЕЙНОЙ КЛАССИФИКАЦИИ С ПОМОЩЬЮ ЛОГИСТИЧЕСКОЙ РЕГРЕССИИ

Во время прогнозирования классов наша линейная граница принимает простое двоичное решение. Однако, как мы узнали в главе 20, не все прогнозы нужно рассматривать как равноценные. Иногда в одних прогнозах мы более уверены, чем в других. К примеру, если все соседи в модели KNN единогласно проголосуют за класс 1, в этом прогнозе мы будем уверены на 100 %. Если же за него проголосуют всего шесть соседей, то уверенность составит 66 %. Суть в том, что нашему перцептрону недостает этой меры уверенности. У модели есть всего два вывода, 0 и 1, которые определяются тем, лежат ли данные выше либо ниже решающей границы.

А что насчет точки данных, находящейся ровно на этой границе? Сейчас наша логика присвоит такой точке класс 0.

ПРИМЕЧАНИЕ

Если размеры в v лежат на решающей границе, тогда weights @ v == 0. В связи с этим int(weights @ v > 0) возвращает 0.

Однако это произвольное присваивание. Если точка находится не ниже и не выше решающей границы, определить ее класс нельзя. Таким образом, уверенность в ее принадлежности к тому или иному классу должна равняться 50 %. А что, если эту точку сместить на 0,0001 единицы выше границы? Это должно повысить уверенность в ее принадлежности классу 1, но ненамного. Можно предположить, что вероятность класса 1 увеличивается до 50,001 %, а вероятность класса 0 уменьшается до 49,999 %. Только в случае довольно большого удаления точки от границы уверенность возрастет значительно, как показано на рис. 21.9. К примеру, если точка находится в 100 единицах выше границы, то наша уверенность в классе 1 должна достичь 100 %, а уверенность в классе 0 упасть до 0 %.

Уверенность в принадлежности к тому или иному классу определяется расстоянием от границы и позицией относительно нее. Если точка лежит в 100 единицах ниже решающей границы, вероятности ее принадлежности классу 1 и 0 нужно поменять местами. Расстояние и позицию точки можно отразить с помощью *ориентированного расстояния*. В отличие от стандартного, ориентированное расстояние может быть отрицательным. Отрицательное расстояние присваивается точке, если она находится ниже решающей границы.

ПРИМЕЧАНИЕ

То есть, если точка находится в 100 единицах ниже границы, ее ориентированное расстояние до нее равно –100.

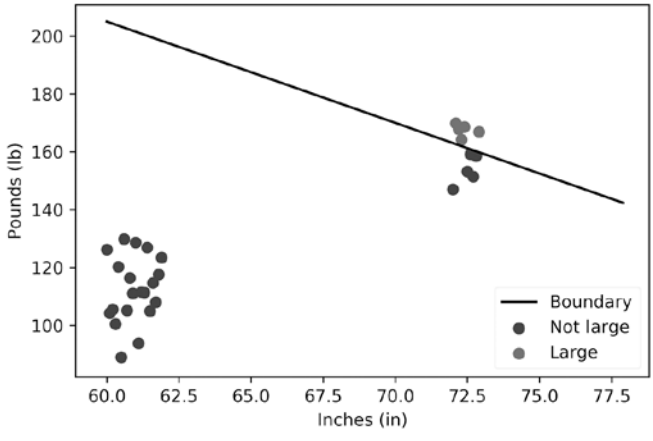


Рис. 21.9. График размеров клиентов: дюймы относительно фунтов. Линейная граница разделяет два класса клиентов. Показаны только удаленные и близкие к границе клиенты. Находящихся близко классифицировать сложнее. Мы намного увереннее в метке класса для тех клиентов, которые находятся далеко от решающей границы

Теперь выберем функцию для вычисления уверенности в классе 1 на основе ориентированного расстояния от границы. По мере его увеличения до бесконечности функция должна подниматься до 1. И наоборот, в случае уменьшения ориентированного расстояния до минус бесконечности функция должна опускаться до 0. Наконец, в случае нулевого ориентированного расстояния она должна равняться 0,5. В данной книге мы уже встречали функцию, которая в эти критерии вписывается. В главе 7 была представлена функция распределения нормальной кривой. Эта S-образная кривая равна вероятности случайного извлечения значения из нормального распределения, которое меньше некоторого `z` либо равно ему. Эта функция начинается в 0 и увеличивается до 1. Она также равна 0,5 при `z == 0`. Напомню, что функцию распределения можно вычислить выполнением `scipy.stats.norm.cdf(z)` (листинг 21.24). Здесь мы строим график такой функции для значений `z` от –10 до 10 (рис. 21.10).

Листинг 21.24. Измерение неопределенности с помощью stats.norm.cdf

```
from scipy import stats
z = np.arange(-10, 10, 0.1)
assert stats.norm.cdf(0.0) == 0.5
plt.plot(z, stats.norm.cdf(z))
plt.xlabel('Directed Distance')
plt.ylabel('Confidence in Class 1')
plt.show()
```

Подтверждает, что кривая имеет равную уверенность в обоих классах, когда z лежит непосредственно на пороге 0

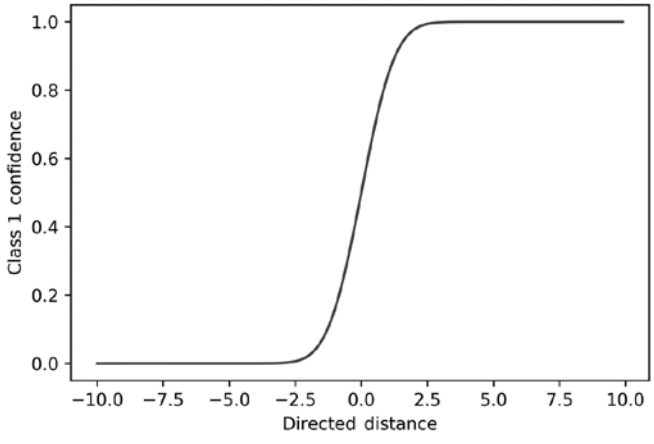


Рис. 21.10. Функция распределения нормального распределения. Эта S-образная кривая начинается в 0 и поднимается до 1. При входном значении 0 она равняется 0,5. Этот график вписывается в наши критерии для отражения неопределенности исходя из ориентированного расстояния от решающей границы

S-образная кривая функции распределения соответствует заявленным нами критериям уверенности. Она вполне годится для вычисления неопределенности классификатора. Правда, в последние десятилетия использование этой кривой стало не особо популярным, и на то есть несколько причин. Одна из наиболее весомых проблем связана с отсутствием точной формулы для вычисления `stats.norm.cdf`. Вместо этого площадь под нормальным распределением вычисляется путем аппроксимации, в связи с чем аналитики данных начали применять другую S-образную кривую, простую формулу которой легко запомнить. Речь идет о *логистической* кривой. Логистическая функция `z` равна `1 / (1 – e ** z)`, где `e` — константа, равная примерно 2,72. Во многом аналогично нормальному распределению логистическая функция располагается в промежутке от 0 до 1 и равна 0,5 при `z == 0`. Давайте построим график логистической кривой вместе с `stats.norm.cdf` (листинг 21.25; рис. 21.11).

Листинг 21.25. Измерение неопределенности с помощью логистической кривой

```
from math import e
plt.plot(z, stats.norm.cdf(z), label='CDF')
plt.plot(z, 1 / (1 + e ** -z), label='Logistic Curve', linestyle='--')
plt.xlabel('Directed Distance')
plt.ylabel('Confidence in Class 1')
plt.legend()
plt.show()
```

Две эти кривые не совпадают в точности, но они обе:

- равны примерно 1 при `z > 5`;
- равны примерно 0 при `-z > 5`;

- равны неопределенному значению между 0 и 1 при `-5 < z < 5`;
- равны 0,5 при `z == 0`.

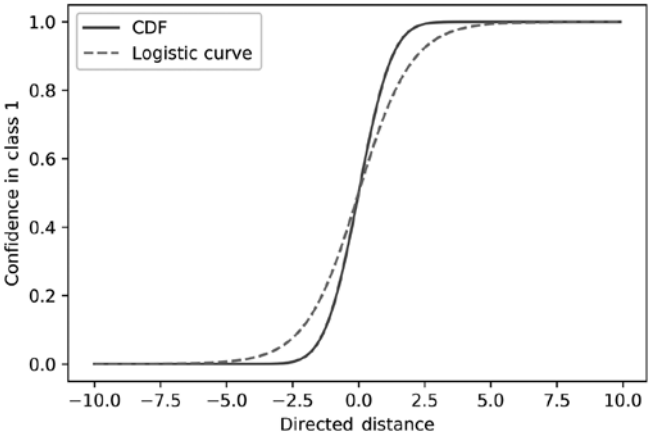


Рис. 21.11. Функция распределения нормального распределения, построенная вместе с логистической кривой. Обе эти S-образные кривые начинаются в 0 и восходят к 1, будучи равными 0,5 при входном значении 0. Обе кривые вписываются в наши критерии отражения неопределенности, исходя из ориентированного расстояния от решающей границы

Получается, что логистическую кривую можно задействовать в качестве меры неопределенности. Далее мы применим ее для присваивания вероятностей принадлежности к классу 1 всем нашим клиентам. Для этого потребуется вычислить ориентированное расстояние между размерами каждого клиента и границей. И вычислить эти вероятности будет на удивление легко — достаточно просто выполнить `M @ weights`, где `M` — дополненная матрица признаков. По сути, мы все время вычисляли эти расстояния, просто до этого момента не использовали их полноценно.

ПРИМЕЧАНИЕ

Давайте подтвердим, что M @ weights возвращает расстояния до решающей границы. Для большей ясности мы используем изначальные веса [3.5, 1, –415], представляющие решающую границу lbs = –3.5 * inches - 415. Таким образом получаем расстояние между размерами (inches, lbs) и точкой решающей границы (inches, –3.5 * inches + 415). Естественно, обе координаты x равны inches, значит, мы получаем расстояние вдоль оси Y, которое равно lbs - (–3.5* inches + 415). Эта формула перестраивается в 3.5 * inches + lbs – 415, равняясь скалярному произведению между [3.5, 1, –415] и [inches, lbs, 1]. Первый вектор соответствует weights, а второй представляет строку в M. Следовательно, M @ weights возвращает массив ориентированных расстояний.

Если `M @ weights` возвращает ориентированные расстояния, тогда `1 / (1 + e ** -(M @ weights))` возвращает вероятности принадлежности классу 1. Код листинга 21.26

строит график расстояния относительно вероятности. Мы также добавляем в этот график прогнозы двоичного перцептрона, которые соответствуют `M @ weights > 0` (рис. 21.12).

ПРИМЕЧАНИЕ

Напомню, что мы вычислили weights путем обучения на стандартизированных признаках в X_s. Значит, нужно добавить в X_s столбце единиц, чтобы дополнить матрицу признаков.

Листинг 21.26. Сравнение логистической неуверенности с прогнозами перцептрона

```
M = np.column_stack([X_s, np.ones(X_s.shape[0])])
distances = M @ weights
likelihoods = 1 / (1 + e ** -distances)
plt.scatter(distances, likelihoods, label='Class 1 Likelihood')
plt.scatter(distances, distances > 0,
            label='Perceptron Prediction', marker='x')

plt.xlabel('Directed Distance')
plt.legend()
plt.show()
```

Ориентированные расстояния до границы равны произведению дополненной матрицы признаков и весов

Прогнозы перцептрона определяются distances > 0. Заметьте, что Python автоматически преобразует True и False в 1 и 0 соответственно, значит, можно подставить distances > 0 прямо в plt.scatter без приведения к целым числам

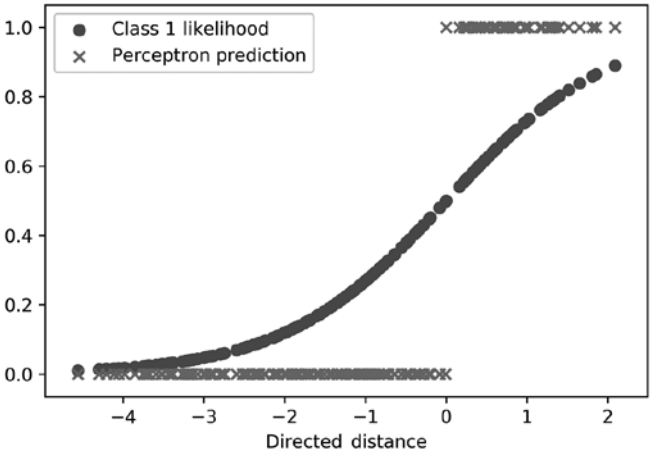


Рис. 21.12. Вероятности принадлежности классу 1, исходя из логистической кривой, построенные вместе с прогнозами перцептрона. Они отражают степень вероятности, притом что прогнозы перцептрона ограничиваются 0 и 1

График логистических вероятностей непрерывно возрастает вместе с ориентированным расстоянием. В противоположность этому прогнозы перцептрона совершенно примитивны — он уверен в метке класса 1 либо на 100, либо на 0 %. Интересно, что

и логистическая кривая, и перцептрон в случае сильно отрицательного ориентированного расстояния имеют уверенность 0 %. Однако по мере увеличения этого расстояния графики начинают расходиться. Логистический график более консервативен — его уверенность возрастает медленно и в основном остается ниже 85 %. Уверенность же перцептрона при `distances > 0` подскакивает до 100 %, причем этот скачок ничем не обоснован. Модель излишне уверена, подобно неопытному подростку, который неизбежно совершает ошибки. К счастью, можно обучить ее осторожности, внедрив критерий неопределенности, отражаемый логистической кривой.

Для внедрения этого критерия нужно изменить вычисление сдвига весов. Сейчас этот сдвиг пропорционален `predicted – actual`, где переменные представляют метки спрогнозированного и фактического классов. Вместо этого можно сделать сдвиг пропорциональным `confidence(predicted) – actual`, где `confidence(predicted)` отражает нашу уверенность в спрогнозированном классе. В модели перцептрона `confidence(predicted)` всегда равна 0 либо 1. В логистической же модели сдвиг весов учитывает более полноценный диапазон значений.

Рассмотрим, к примеру, точку данных, которая имеет метку класса 1 и лежит прямо на решающей границе. Перцептрон при наблюдении этих данных во время обучения вычисляет нулевой сдвиг весов, значит, подстраивать он их не будет. Из этого наблюдения он ничему не научится. В противоположность этому логистическая модель возвращает сдвиг весов, пропорциональный 0,5 – 1 = –0,5. Эта модель будет корректировать свою оценку неопределенности в метке класса и соответствующим образом подстраивать веса. В отличие от перцептрона логистическая модель гибка в обучении.

Теперь обновим код обучения нашей модели, добавив логистическую неопределенность (листинг 21.27). Для этого нужно просто заменить вывод функции `predict` с `int(weights @ v > 0)` на `1 / (1 + e** -(weights @ v))`. Здесь мы делаем это с помощью двух строк кода. После этого обучаем доработанную модель генерировать новый вектор весов и строим график новой решающей границы для оценки результатов (рис. 21.13).

Листинг 21.27. Внесение в обучение неопределенности

```
                                  Наша функция train получает опциональный предиктор класса строкового
                                уровня, называемый predict, настроенный на возвращение int(weights @ v > 0).
                                        Здесь мы заменяем его на более подробную функцию logistic_predict
np.random.seed(0)
def logistic_predict(v, weights): return 1 / (1 + e ** -(weights @ v))
def train_logistic(X, y): return train(X, y, predict=logistic_predict)
logistic_weights = train_logistic(X_s, y)[0]
plot_boundary(logistic_weights)
```

Обученная решающая граница практически идентична выводу перцептрона. Однако функция `train_logistic` несколько отличается — она дает более последовательные

результаты, нежели перцептрон. До этого мы показали, что обученный перцептрон в четырех из пяти обучающих циклов показывает себя хуже базовой границы. Актуально ли это для `train_logistic`? Сейчас выясним (листинг 21.28).

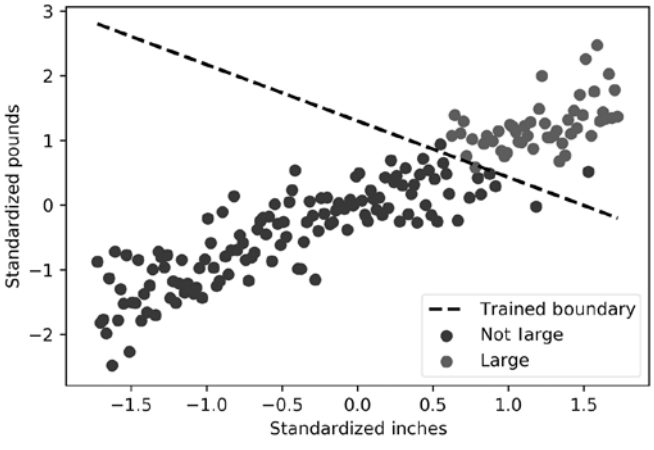


Рис. 21.13. График стандартизированных размеров клиентов. Логистически обученная граница отделяет «больших» клиентов от «небольших». Это разделение совпадает с разделением базовой решающей границы на рис. 21.2

Листинг 21.28. Проверка согласованности результатов при обучении логистической модели

```
np.random.seed(0)
poor_train_count = sum([train_logistic(X_s, y)[1][-1] < 0.97
                        for _ in range(5)])
print("The f-measure fell below our baseline of 0.97 in "
      f"{poor_train_count} out of 5 training instances")
```

```
The f-measure fell below our baseline of 0.97 in 0 out of 5 training instances
```

Результаты этой обученной модели не опускаются ниже базовой границы ни в одном из циклов, значит, перцептрон она по качеству превосходит. Называется эта превосходящая модель *классификатором на основе логистической регрессии*. Алгоритм обучения этой модели также обычно зовется *логистической регрессией*.

ПРИМЕЧАНИЕ

Возможно, это название не является семантически корректным. Классификаторы прогнозируют категориальные переменные, а регрессионная модель — численные значения. Технически классификатор на основе логистической регрессии использует логистическую регрессию для прогнозирования численной неопределенности, но это не регрессионная модель. Однако термин «логистическая регрессия» в среде машинного обучения стал применяться повсеместно наравне с термином «классификатор на основе логистической регрессии».

Классификатор на основе логистической регрессии обучается подобно перцептрону, но с одним небольшим отличием. Сдвиг весов в этом случае не пропорционален `int(distance – y[i] > 0)`, где `distance = M[i] @ weights`. Он пропорционален `1 / (1 + e ** -distance) – y[i]`. Это отличие дает более стабильные показатели в случайных обучающих прогонах.

ПРИМЕЧАНИЕ

А что произойдет, если сдвиг весов будет прямо пропорционален distance - y[i]? В таком случае обученная модель научится минимизировать расстояние между линией и значениями в y. В целях классификации пользы от этого немного, а вот для регрессии такой подход бесценен. К примеру, если установить y равным lbs, а X — inches, можно будет обучить линию прогнозировать вес клиентов на основе их роста. Используя всего две строки кода, мы задействуем train для реализации этого вида алгоритма линейной регрессии. Сможете додуматься как?

21.3.1. Выполнение логистической регрессии для более чем двух признаков

Мы обучили свою модель логистической регрессии на двух характеристиках клиентов — росте (`inches`) и весе (`lbs`). Однако функция `train_logistic` может обрабатывать любое число входных признаков, что мы и подтвердим, добавив третий — обхват талии клиента. В среднем обхват талии равен 45 % от роста человека. Этот факт мы используем для симуляции размеров обхвата талии клиентов, после чего передадим все три характеристики в `train_logistic` и оценим эффективность обученной модели (листинг 21.29).

Листинг 21.29. Обучение модели логистической регрессии на трех признаках

```
np.random.seed(0)
random_fluctuations = np.random.normal(size=X.shape[0], scale=0.1)

waist = 0.45 * X[:,0] + random_fluctuations
X_w_waist = np.column_stack([X_s, (waist - waist.mean()) / waist.std()])
weights, f_measures = train_logistic(X_w_waist, y)

print("Our trained model has the following weights:")
print(np.round(weights, 2))
print(f'\nThe f-measure is {f_measures[-1]:.2f}')
```

Каждый размер waist равен 45 % от роста клиента плюс случайная флуктуация

Значения обхвата талии необходимо стандартизировать, прежде чем добавлять их массив к остальным стандартизированным размерам клиентов

```
Our trained model has the following weights:
[ 1.65 2.91 1.26 -4.08]

The f-measure is 0.97
```

Эта обученная на трех признаках модель продолжает справляться исключительно хорошо, демонстрируя F-меру 0,97. Основное ее отличие в том, что теперь она содержит четверо весов. Первые трое представлены коэффициентами,

соответствующими трем размерам клиентов, а четвертые являются смещением. Геометрически эти веса представляют собой линейную границу повышенной размерности, которая принимает форму трехмерной линии, называемой *плоскостью*. Она делит наши два класса клиентов в 3D-пространстве, как показано на рис. 21.14.

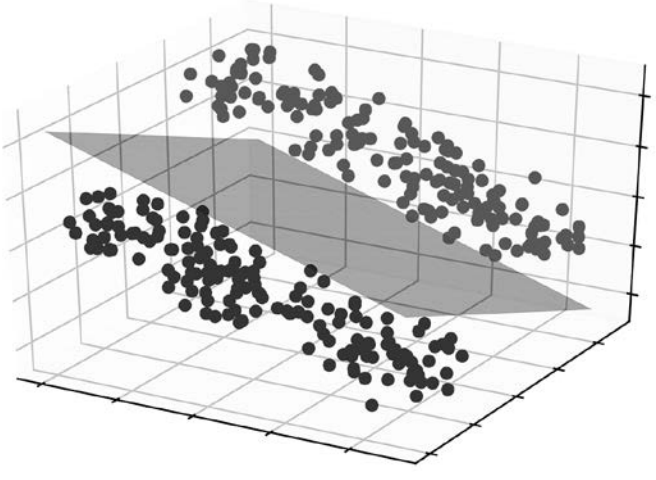


Рис. 21.14. Линейная классификация в 3D-пространстве. Линейная плоскость рассекает данные, разделяя их на два отдельных класса

Аналогичным образом можно выполнить оптимизацию линейного деления в любом произвольном числе измерений. Полученные веса представляют многомерную линейную решающую границу. Вскоре мы выполним логистическую регрессию для набора данных с 13 признаками, в чем нам поможет реализация логистической регрессии из scikit-learn.

21.4. ОБУЧЕНИЕ ЛИНЕЙНЫХ КЛАССИФИКАТОРОВ С ПОМОЩЬЮ SCIKIT-LEARN

В scikit-learn есть класс для классификации с помощью логистической регрессии `logisticRegression`, и начнем мы с его импорта (листинг 21.30).

ПРИМЕЧАНИЕ

В scikit-learn есть также класс perceptron, который можно импортировать из sklearn.linear_model.

Листинг 21.30. Импорт класса LogisticRegression из scikit-learn

```
from sklearn.linear_model import LogisticRegression
```

Далее мы инициализируем объект классификатора `clf` (листинг 21.31).

Листинг 21.31. Инициализация классификатора scikit-learn LogisticRegression

```
clf = LogisticRegression()
```

Как говорилось в главе 20, любой `clf` можно обучить, выполнив `clf.fit(X, y)`. Давайте обучим наш логистический классификатор, используя стандартизированную матрицу `X_s`, включающую два признака (листинг 21.32).

Листинг 21.32. Обучение классификатора scikit-learn LogisticRegression

```
clf.fit(X_s, y)
```

Классификатор изучил вектор весов `[inches_coef, lbs_coef, bias]`. Коэффициенты этого вектора хранятся в атрибуте `clf.coef_`. При этом к смещению нужно обращаться отдельно с помощью атрибута `clf.intercept_` (листинг 21.33). Объединение этих атрибутов дает нам полноценный вектор, который можно визуализировать в виде решающей границы (рис. 21.15).

Листинг 21.33. Обращение к обученной решающей границе

```
coefficients = clf.coef_
bias = clf.intercept_
print(f"The coefficients equal {np.round(coefficients, 2)}")
print(f"The bias equals {np.round(bias, 2)}")
plot_boundary(np.hstack([clf.coef_[0], clf.intercept_]))

The coefficients equal [[2.22 3.22]]
The bias equals [-3.96]
```

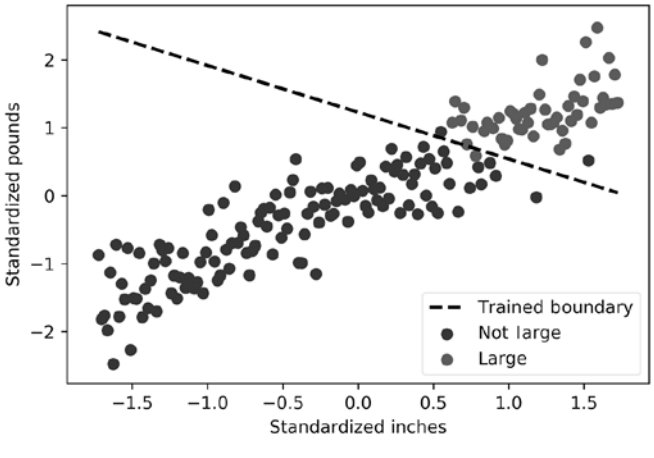


Рис. 21.15. График стандартизированных характеристик клиентов. Логистически обученная решающая граница, полученная с помощью scikit-learn, разделяет «больших» и «небольших» клиентов

Для выполнения прогнозов относительно новых данных мы выполняем `clf.predict` (листинг 21.34). Напомню, что для получения адекватных прогнозов входные данные необходимо стандартизировать.

Листинг 21.34. Прогнозирование классов с помощью линейного классификатора

```
new_data = np.array([[63, 110], [76, 199]])
predictions = clf.predict(standardize(new_data))
print(predictions)

[0 1]
```

Помимо этого, можно вывести вероятности меток классов, выполнив `clf.predict_proba` (листинг 21.35). Эти вероятности представляют неопределенности меток, сгенерированные логистической кривой.

Листинг 21.35. Вывод неопределенности, связанной с каждым классом

```
probabilities = clf.predict_proba(standardize(new_data))
print(probabilities)

[[9.99990471e-01 9.52928118e-06]
 [1.80480919e-03 9.98195191e-01]]
```

В двух последних листингах мы стандартизировали входные данные с помощью собственной функции `standardize`, но в scikit-learn есть и собственный класс стандартизации под названием `StandardScaler`. Далее мы этот класс импортируем и инициализируем (листинг 21.36).

Листинг 21.36. Инициализация класса стандартизации из scikit-learn

```
from sklearn.preprocessing import StandardScaler
standard_scaler = StandardScaler()
```

Выполнение `standard_scaler.fit_transform(X)` приведет к возвращению стандартизированной матрицы (листинг 21.37). Средние столбцов этой матрицы равны 0, а стандартные отклонения — 1. Естественно, эта матрица идентична имеющейся у нас матрице `X_s`.

Листинг 21.37. Стандартизация обучающих данных с помощью scikit-learn

```
X_transformed = standard_scaler.fit_transform(X)
assert np.allclose(X_transformed.mean(axis=0), 0)

assert np.allclose(X_transformed.std(axis=0), 1)
assert np.allclose(X_transformed, X_s)
```

Объект `standard_scaler` выучил средние значения и стандартные отклонения, связанные с нашей матрицей признаков, значит, теперь он может стандартизировать данные на основе этих статистик. Код листинга 21.38 стандартизирует матрицу

new_data, выполняя standard_scaler.transform(new_data). Эти стандартизированные данные мы передаем классификатору. Спрогнозированный результат должен соответствовать ранее полученному массиву predictions.

Листинг 21.38. Стандартизация новых данных с помощью scikit-learn

```
data_transformed = standard_scaler.transform(new_data)
assert np.array_equal(clf.predict(data_transformed), predictions)
```

Путем совмещения классов LogisticRegression и StandardScaler можно обучать логистические модели на сложных входных данных. В следующем разделе мы обучим модель, способную обрабатывать более двух признаков и прогнозировать более двух меток классов.

РЕЛЕВАНТНЫЕ МЕТОДЫ ЛИНЕЙНОГО КЛАССИФИКАТОРА В SCIKIT-LEARN

- clf = LogisticRegression() — инициализирует классификатор на основе логистической регрессии.
- scaler = StandardScaler() — инициализирует стандартный масштабатор.
- clf.fit(scalar.fit_transform(X)) — обучает классификатор на стандартизированных данных.
- clf.predict(scalar.transform(new_data)) — прогнозирует классы на основе стандартизированных данных.
- clf.predict_proba(scalar.transform(new_data)) — прогнозирует вероятности классов на основе стандартизированных данных.

21.4.1. Обучение мультиклассовых линейных моделей

Мы продемонстрировали, как линейные классификаторы могут находить решающие границы, разделяющие два класса данных. Но многие задачи требуют проводить различие между бóльшим числом классов. Рассмотрим, к примеру, идущую из глубин веков практику дегустации вина. Как известно, некоторые эксперты способны почувствовать разницу между множеством классов вин, используя исключительно органы чувств. Предположим, мы хотим создать машину для дегустации вина. С помощью датчиков она будет обнаруживать химические паттерны в стакане напитка, передавая полученные измерения линейному классификатору в виде признаков. Затем тот будет определять сорт вина (впечатляя нас точностью и сообразительностью). Для обучения такого линейного классификатора нужна обучающая выборка. К счастью, подходящий набор данных можно получить через scikit-learn. Давайте загрузим его (листинг 21.39), импортировав

и выполнив функцию `load_wine`, после чего выведем из этих данных названия признаков и метки классов.

Листинг 21.39. Импорт набора данных вин из scikit-learn

```
from sklearn.datasets import load_wine
data = load_wine()

num_classes = len(data.target_names)
num_features = len(data.feature_names)
print(f"The wine dataset contains {num_classes} classes of wine:")
print(data.target_names)
print(f"\nIt contains the {num_features} features:")
print(data.feature_names)

The wine dataset contains 3 classes of wine:
['class_0' 'class_1' 'class_2']

It contains the 13 features:
['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash', 'magnesium',
'total_phenols', 'flavanoids', 'nonflavanoid_phenols', 'proanthocyanins',
'color_intensity', 'hue', 'od280/od315_of_diluted_wines', 'proline']
```

В наборе данных есть параметр «флавоноиды»

Набор данных содержит 13 измеренных признаков, включая концентрацию алкоголя (признак 0), уровень магния (признак 4) и оттенок (признак 10). При этом всего в нем содержатся три класса вин.

ПРИМЕЧАНИЕ

За давностью лет сейчас уже неизвестно, каким именно винам принадлежат эти характеристики, но есть вероятность, что речь идет о красных сортах вроде каберне, мерло и пино-нуар.

Как обучить модель логистической регрессии различать три этих вида? Для этого можно сначала обучить простой двоичный классификатор проверять, принадлежит ли вино к классу 0. Либо можно обучить другой классификатор, который будет прогнозировать принадлежность вина к классу 1. Наконец, третий классификатор мог бы определять, относится ли вино к классу 2. По сути, это и есть встроенная в scikit-learn логика линейной классификации по нескольким классам. Оперируя тремя категориями, scikit-learn обучает три решающие границы, по одной для каждого класса. Затем модель вычисляет три разных прогноза для входных данных и выбирает тот, что имеет наивысший уровень уверенности.

ПРИМЕЧАНИЕ

Это еще одна причина, по которой вычисление уверенности критически важно для выполнения линейной классификации.

Если обучить конвейер логистической регрессии на трех классах данных о винах, то мы получим три решающие границы, соответствующие классам 0, 1 и 2. Каждая из них будет иметь собственный вектор весов. У каждого вектора весов окажется свое смещение, значит, у обученной модели смещений будет три. Эти три смещения будут храниться в трехэлементном массиве `clf.intercept_`. Обращение к `clf.intercept_[i]` предоставит нам смещение для класса *i*. Далее мы обучим эту модель и выведем три полученных смещения (листинг 21.40).

Листинг 21.40. Обучение мультиклассового предиктора сорта вина

```
X, y = load_wine(return_X_y=True)
clf.fit(standard_scaler.fit_transform(X), y)
biases = clf.intercept_

print(f"We trained {biases.size} decision boundaries, corresponding to "
      f"the {num_classes} classes of wine.\n")

for i, bias in enumerate(biases):
    label = data.target_names[i]
    print(f"The {label} decision boundary has a bias of {bias:0.2f}")
```

```
We trained 3 decision boundaries, corresponding to the 3 classes of wine.

The class_0 decision boundary has a bias of 0.41
The class_1 decision boundary has a bias of 0.70
The class_2 decision boundary has a bias of -1.12
```

Вдобавок к смещению каждая решающая граница должна иметь коэффициенты. Они используются для взвешивания входных признаков во время классификации, значит, между ними и признаками существует соответствие «один к одному». Наш набор данных содержит 13 признаков, представляющих различные свойства вина, выходит, каждая решающая граница должна иметь 13 соответствующих коэффициентов. Коэффициенты для трех разных границ можно сохранить в матрице размером 3×13. В scikit-learn она находится в `clf.coef_`. Каждая *i*-я строка в ней соответствует границе класса *i*, а каждый *i*-й столбец — коэффициенту *i* признака. К примеру, нам известно, что признак 0 представляет содержание алкоголя в вине, значит, `clf_coeff_[2][0]` соответствует коэффициенту содержания алкоголя для границы класса 2.

Давайте визуализируем матрицу коэффициентов в виде тепловой карты (листинг 21.41; рис. 21.16). Это позволит отобразить названия признаков и метки классов, соответствующие строкам и столбцам. Заметьте, что длинные имена признаков проще читать, если вывести транспонированную версию матрицы. В связи с этим мы передаем `clf.coeff_.T` в `sns.heatmap`.

Листинг 21.41. Визуализация транспонированной версии матрицы коэффициентов

```
import seaborn as sns
plt.figure(figsize = (20, 10))
coefficients = clf.coef_

sns.heatmap(coefficients.T, cmap='YlGnBu', annot=True,
            xticklabels=[f"Class {i} Boundary" for i in range(3)],
            yticklabels=data.feature_names)
plt.yticks(rotation=0)
sns.set(font_scale=2)
plt.show()
```

Устанавливает ширину и высоту тепловой карты на 20 и 10 дюймов соответственно

Транспонирует матрицу коэффициентов для простоты отображения их имен

Корректирует шрифт меток для лучшей удобочитаемости

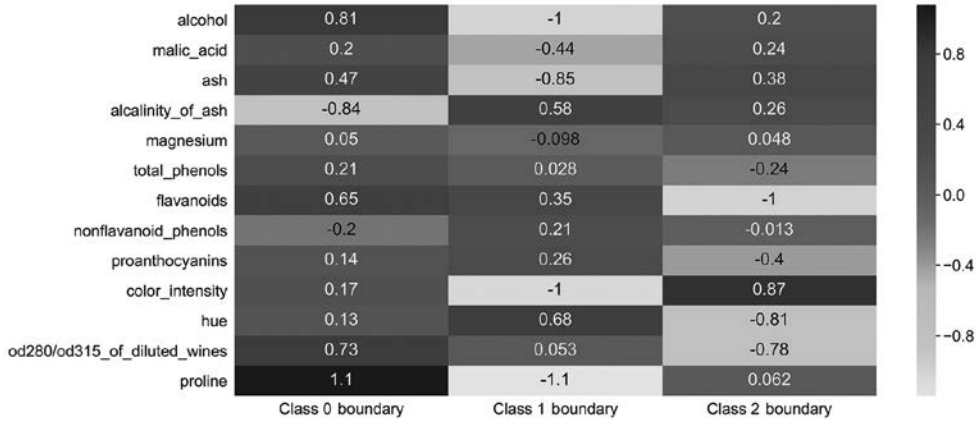


Рис. 21.16. Тепловая карта коэффициентов 13 признаков среди трех решающих границ

В этой тепловой карте коэффициенты для разных границ различаются. Например, коэффициент содержания алкоголя равен –0,81, –1 и 0,2 для границ классов 0, 1 и 2 соответственно. Подобные различия в коэффициентах могут оказаться очень полезными, так как позволяют лучше понять, как входные признаки влияют на прогнозы.

РЕЛЕВАНТНЫЕ АТРИБУТЫ ЛИНЕЙНЫХ КЛАССИФИКАТОРОВ SCIKIT-LEARN

- `clf.coef_` — обращается к матрице коэффициентов обученного линейного классификатора.
- `clf.intercept_` — обращается ко всем значениям смещений в обученном линейном классификаторе.

21.5. ИЗМЕРЕНИЕ ВАЖНОСТИ ПРИЗНАКОВ С ПОМОЩЬЮ КОЭФФИЦИЕНТОВ

В главе 20 мы говорили о том, что результаты классификатора KNN невозможно трактовать. Используя KNN, мы можем прогнозировать класс, связанный с входными признаками, но не можем понять, почему эти признаки принадлежат данному классу. К счастью, вывод логистической регрессии интерпретировать проще. Понять, почему признаки модели дают тот или иной прогноз, можно путем анализа соответствующих им коэффициентов.

Линейная классификация управляется взвешенной суммой признаков и коэффициентов. Значит, если модель получает три признака, A, B и C, опираясь при этом на три коэффициента [1, 0, 0,25], тогда ее прогноз частично определяется значением A + 0,25 × C. Заметьте, что в этом примере признак B обнуляется. Умножение нулевого коэффициента на признак всегда дает ноль, поэтому такой признак на прогноз модели не влияет.

Теперь рассмотрим признак, коэффициент которого очень близок к нулю. Он хотя и влияет на прогноз, но незначительно. И напротив, если коэффициент далек от нуля, связанный с ним признак будет влиять на прогноз модели намного сильнее. По сути, коэффициенты с высокими абсолютными значениями оказывают на модель большее влияние, поэтому при ее анализе связанные с ними признаки оказываются более важными. К примеру, в нашем случае признак A самый весомый, поскольку его коэффициент дальше всех от нуля (рис. 21.17).

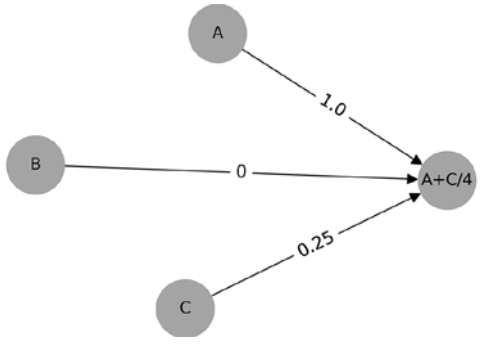


Рис. 21.17. Взвешенную сумму признаков [A, B, C] и коэффициентов [1, 0, 0,25] можно визуализировать в виде направленного графа. В нем левые узлы представляют признаки, а веса ребер — коэффициенты. Мы умножаем каждый узел на вес соответствующего ему ребра и суммируем результаты. Эта сумма равна A + C/4, значит, A в четыре раза значительнее C. При этом B обнуляется и на результаты не влияет

При анализе *значимости признаков* их можно оценивать по соответствующим коэффициентам. Показатель значимости ранжирует полезность признаков во время классификации. Разные модели классификаторов дают разные показатели значимости признаков. В линейных моделях для грубой оценки этого показателя используются абсолютные значения коэффициентов.

ПРИМЕЧАНИЕ

Модели, представленные в главе 22, имеют более детальные показатели значимости признаков.

Какой признак наиболее полезен для верного определения вина класса 0? Проверить это можно, упорядочив признаки на основе абсолютных значений коэффициентов класса 0 в `clf.coef_[0]` (листинг 21.42).

Листинг 21.42. Ранжирование признаков класса 0 по значимости

```
def rank_features(class_label):
    absolute_values = np.abs(clf.coef_[class_label])
    for i in np.argsort(absolute_values)[::-1]:
        name = data.feature_names[i]
        coef = clf.coef_[class_label][i]
        print(f"{name}: {coef:.2f}")

rank_features(0)
```

Ранжирует признаки на основе абсолютного значения коэффициентов в clf.coef_[class_label]

Вычисляет абсолютные значения

Упорядочивает индексы признаков по абсолютным значениям в порядке убывания

```
proline: 1.08
alcalinity_of_ash: -0.84
alcohol: 0.81
od280/od315_of_diluted_wines: 0.73
flavanoids: 0.65
ash: 0.47
total_phenols: 0.21
malic_acid: 0.20
nonflavanoid_phenols: -0.20
color_intensity: 0.17
proanthocyanins: 0.14
hue: 0.13
magnesium: 0.05
```

Верхнюю строчку ранжированного списка занимает пролин. Это типичный химический компонент вина, концентрация которого зависит от вида использованного в приготовлении винограда. Концентрация пролина — самый важный признак для определения вин класса 0. Теперь проверим, какой признак доминирует в классификации вин класса 1 (листинг 21.43).

Листинг 21.43. Ранжирование признаков класса 1 по значимости

```
rank_features(1)

proline: -1.14
color_intensity: -1.04
alcohol: -1.01
ash: -0.85
hue: 0.68
alcalinity_of_ash: 0.58
malic_acid: -0.44
flavanoids: 0.35
proanthocyanins: 0.26
nonflavanoid_phenols: 0.21
magnesium: -0.10
od280/od315_of_diluted_wines: 0.05
total_phenols: 0.03
```

И в классе 0, и в классе 1 самым важным признаком оказалась концентрация пролина. Тем не менее этот признак влияет на прогнозирование этих двух классов по-разному: коэффициент пролина в классе 0 положителен (1,08), а в классе 1 отрицателен (–1,14). Знак коэффициента очень важен. Положительные коэффициенты увеличивают взвешенную сумму линейных значений, а отрицательные — уменьшают. Следовательно, при классификации класса 1 пролин уменьшает взвешенную сумму. Это дает отрицательно направленное расстояние от решающей границы, в связи с чем вероятность класса 1 падает до нуля. При этом положительный коэффициент класса 0 имеет противоположный эффект. Таким образом, высокая концентрация пролина обусловливает следующее:

- вино с меньшей вероятностью будет относиться к классу 1;
- вино с большей вероятностью будет относиться к классу 0.

Для проверки данной гипотезы построим гистограммы концентрации этих двух классов вина (листинг 21.44; рис. 21.18).

Листинг 21.44. Построение гистограмм пролина для вин классов 0 и 1

```
index = data.feature_names.index('proline')
plt.hist(X[y == 0][:, index], label='Class 0')
plt.hist(X[y == 1][:, index], label='Class 1', color='y')
plt.xlabel('Proline concentration')
plt.legend()
plt.show()
```

В среднем концентрация пролина в классе 0 оказывается выше, чем в классе 1. Данное отличие служит сигналом для различения двух видов вин, и наш классификатор успешно этот сигнал изучил. Анализируя коэффициенты классификатора, мы также кое-что узнали о химическом составе разных вин.

В отличие от моделей KNN, логистическая регрессия допускает интерпретацию результатов. Такие классификаторы легко обучаются и быстро выполняются, значит, их можно считать превосходящими по своим возможностям модели KNN. К сожалению, линейные классификаторы все равно имеют ряд серьезных недочетов, которые ограничивают их практическое применение в определенных обстоятельствах.

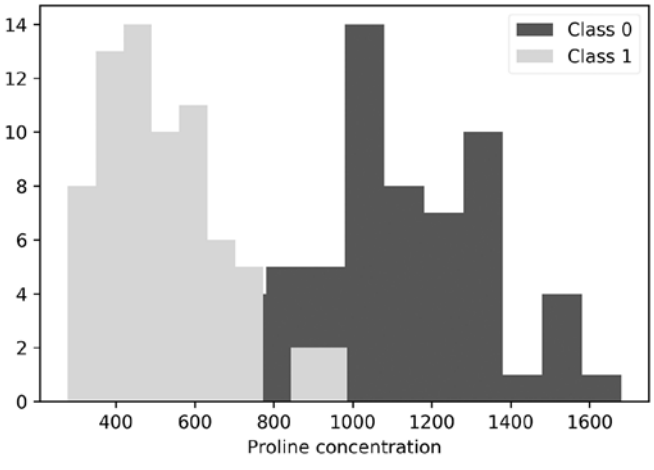


Рис. 21.18. Гистограмма концентрации пролина в винах классов 0 и 1. Концентрация в винах класса 0 заметно выше, чем в винах класса 1. Наш классификатор выделил этот сигнал, сделав пролин самым важным коэффициентом как для класса 0, так и для класса 1

21.6. ОГРАНИЧЕНИЯ ЛИНЕЙНЫХ КЛАССИФИКАТОРОВ

Линейные классификаторы плохо справляются с неподготовленными данными. Как мы видели, для достижения наилучших результатов необходима стандартизация. Они также не могут обрабатывать без препроцессинга данных категориальные признаки. Предположим, мы создаем модель, которая должна прогнозировать, будет ли взят из приюта домашний питомец. Наша модель может определять три категории животных: кошек, собак и кроликов. Проще всего представить эти категории с помощью чисел: 0 для кошек, 1 для собак и 2 для кроликов. Однако такое представление приведет к провалу данной модели. Она будет вдвое больше внимания уделять кроликам, чем собакам, и при этом полностью игнорировать кошек. Для того чтобы модель рассматривала всех животных как равноценных, их категории необходимо преобразовать в трехэлементный двоичный вектор v.

Если питомец принадлежит к категории `i`, `v[i]` должен устанавливаться равным `1`. В противном случае `v[i]` равняется `0`. Таким образом, кошку мы представляем как `v = [1, 0, 0]`, собаку — как `v = [0, 1, 0]`, а кролика — как `v = [0, 0, 1]`. Такая векторизация аналогична векторизации из главы 13, и выполнить ее можно с помощью scikit-learn. Но подобные преобразования все равно могут быть трудоемкими. В следующей главе мы рассмотрим модели, способные анализировать необработанные данные без дополнительного препроцессинга.

ПРИМЕЧАНИЕ

Векторизация категориальных переменных часто называется прямым унитарным кодированием. В scikit-learn есть преобразователь OneHotEncoder, который можно импортировать из sklearn.preprocessing. Класс OneHotEncoder может автоматически обнаруживать и векторизовать все категориальные признаки в обучающей выборке.

Самое серьезное ограничение линейных классификаторов отражено в их названии — линейные классификаторы изучают *линейные* решающие границы. Говоря точнее, для разделения классов данных им требуется линия (или плоскость в более высоких измерениях). Однако существует бесчисленное множество задач классификации, в которых такое линейное деление невозможно. Рассмотрим, к примеру, классификацию городских и пригородных домовладений. В рамках этой задачи предположим, что наш прогноз определяется расстоянием от центра города. Все домовладения, находящиеся в пределах двух единиц от центра, считаются городскими, а все остальные — пригородными. Код листинга 21.45 симулирует эти домашние владения с двухмерным нормальным распределением. Мы также обучим классификатор на основе линейной регрессии проводить различие между двумя классами владений. Наконец, визуализируем линейную границу модели и фактические домовладения в двухмерном пространстве (рис. 21.19).

Листинг 21.45. Симуляция нелинейного сценария с возможностью разделения

```
np.random.seed(0)
X = np.array([[np.random.normal(), np.random.normal()]
              for _ in range(200)])
y = (np.linalg.norm(X, axis=1) < 2).astype(int)

clf = LogisticRegression()
clf.fit(X, y)
weights = np.hstack([clf.coef_[0], clf.intercept_])

a, b, c = weights
boundary = lambda x: -(a * x + c) / b
```

Координаты x и y каждого домовладения взяты из двух стандартных нормальных распределений

Центр города расположен в точке (0, 0). В связи с этим пространственное расстояние от центра до дома равняется его норме. Строения, находящиеся в пределах двух единиц от центра, размечаются как городские

Наши данные были взяты из распределения со средним 0 и стандартным отклонением 1. Значит, для обучения линейной модели требуется стандартизация

```
plt.plot(range(-4, 5), boundary(range(-4, 5)), color='k', linestyle='--',
                                linewidth=2, label='Decision Boundary')
for i in [0, 1]:
    plt.scatter(X[y == i][:, 0], X[y == i][:, 1],
                label= ['Suburban', 'Urban'][i],
                color=['b', 'y'][i])

plt.legend()
plt.show()
```

Строит график обученной решающей границы вместе с координатами домовладений

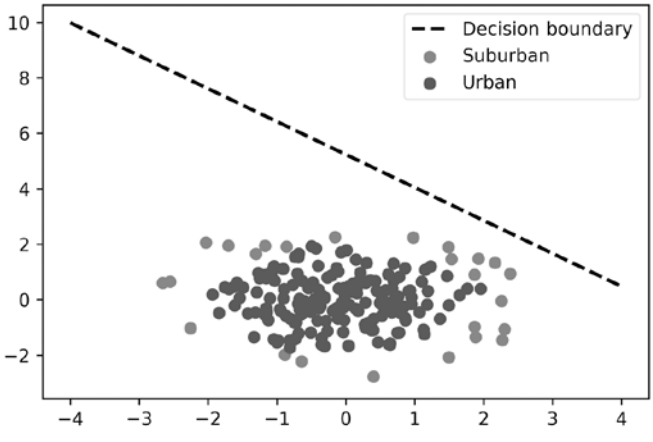


Рис. 21.19. График симулированных домовладений относительно центра города, находящегося в (0, 0). Дома, расположенные ближе к центру, считаются городскими. Между городскими и пригородными владениями линейного разделения не существует, поэтому линейная граница не способна их различить

Линейная граница не может разделить эти два класса, поскольку это не позволяет сделать сама геометрия набора данных. На языке науки о данных мы говорим, что такие данные *линейно неделимы*. В связи с этим линейный классификатор невозможно адекватно обучить. Нам нужен нелинейный подход. В следующей главе мы познакомимся с техникой деревьев решений, которая способна преодолеть это ограничение.

РЕЗЮМЕ

- В определенных случаях мы можем разделить классы данных с помощью *линейных решающих границ*. Все точки данных ниже такой границы классифицируются как принадлежащие к классу 0, а все точки выше нее — как принадлежащие к классу 1. По сути, эта линейная граница проверяет, дают ли взвешенные признаки и константа в сумме больше нуля. Значение константы при этом называется *смещением*, а оставшиеся веса — *коэффициентами*.

- С помощью алгебраических действий можно преобразовать линейную классификацию в неравенство матричного произведения, определяемое как `M @ weights > 0`. Такая классификация на базе умножения определяет *линейный классификатор*. Матрица `M` является *дополненной матрицей признаков* с добавленным в нее столбцом единиц, `weights` — вектором, а последний элемент этого вектора — смещением. Остальные же веса представляют коэффициенты.
- Для получения хорошей решающей границы мы начинаем с инициализации случайных `weights`. После этого итеративно корректируем эти веса на основе различия между спрогнозированными и фактическими классами. В простейшем линейном классификаторе этот сдвиг весов пропорционален разнице между спрогнозированными и фактическими классами. Таким образом, сдвиг пропорционален одному из трех значений: –1, 0 либо 1.
- Никогда не нужно корректировать коэффициент, если связанный с ним признак равен нулю. Гарантировать соблюдение этого ограничения можно умножением сдвига весов на значение соответствующего признака в матрице `M`.
- Итеративная корректировка весов может привести к колебаниям классификатора между высоким и низким качеством прогнозов. Для нивелирования этих колебаний необходимо уменьшать сдвиг весов на каждой последующей итерации. Это можно сделать делением сдвига весов на `k` на каждой k-й итерации.
- Итеративная корректировка весов может сходиться к хорошей решающей границе, но не обязательно к оптимальной. Улучшить эту границу можно уменьшением стандартного отклонения и среднего данных. Подобную *стандартизацию* можно реализовать через вычитание средних значений с последующим делением на стандартные отклонения. Среднее полученного набора данных будет равно 0, а стандартное отклонение — 1.
- Простейший линейный классификатор называется *перцептроном*. Перцептроны очень хороши, но их результаты могут быть непоследовательными. Этот недочет частично объясняется недостатком детальности. Классификация точек, расположенных ближе к решающей границе, менее однозначна. Отразить эту неопределенность можно с помощью S-образной кривой, расположенной в промежутке от 0 до 1. В качестве хорошей меры неопределенности выступает функция распределения, но более простая *логистическая кривая* вычисляется легче. Эта кривая равна `1 / (1 + e ** -z)`.
- Внести критерий неопределенности в обучение модели можно с помощью установки сдвига весов пропорциональным `actual – 1 / (1 + e ** -distance)`. Здесь `distance` представляет ориентированное расстояние до решающей границы. Вычислить все ориентированные расстояния одновременно можно посредством выполнения `M @ weights`.
- Модель, обученная с помощью логистической неопределенности, называется *классификатором на основе логистической регрессии*. Такой классификатор дает более последовательные результаты в сравнении с простым перцептроном.

- Линейные классификаторы можно расширить до определения N классов, обучив N разных линейных решающих границ.
- Коэффициенты в линейном классификаторе выступают в качестве меры *значимости признаков*. Коэффициенты с наибольшими абсолютными значениями соотносятся с признаками, значительно влияющими на прогнозы модели. Знак коэффициента определяет то, указывает ли наличие признака на присутствие либо отсутствие класса.
- Модели линейной классификации не справляются, когда данные оказываются *линейно неделимыми*, и хорошей линейной решающей границы не существует.

22 Обучение нелинейных классификаторов с помощью деревьев решений

В этой главе

- ✓ Классификация линейно неделимых наборов данных.
- ✓ Автоматическая генерация логических правил `if/else` из обучающих данных.
- ✓ Что такое дерево решений.
- ✓ Что такое случайный лес.
- ✓ Обучение моделей на основе деревьев с помощью scikit-learn.

До сих пор мы изучали техники обучения с учителем, опирающиеся на геометрию данных. Но связь между обучением и геометрией не соответствует нашему повседневному опыту. На когнитивном уровне люди не обучаются через абстрактный пространственный анализ. Их обучение — итог логических выводов, которые они делают об окружающем мире. Сформировав такие выводы, ими можно делиться с другими. Маленькие дети узнают, что демонстрацией ложной истерики иногда можно получить лишнее печенье. Родитель, в свою очередь, осознает, что потакание капризам ребенка косвенно ведет к еще более плохому поведению. Студент понимает, что если будет усердно учиться, то с большей вероятностью сдаст экзамен на отлично. Такое осознание ничего нового для нас не представляет, являясь частью нашей коллективной социальной мудрости. Сделав полезный логический вывод, им можно поделиться с другими, чтобы использовать его шире. Подобный

обмен лежит в основе современной науки. Ученый узнает, что определенные белки вируса являются удачными мишенями для лекарства. Он публикует свое открытие в журнале, и это знание распространяется среди всего научного сообщества. В итоге на основе обнаруженных этим ученым данных разрабатывается новое лекарственное средство.

В текущей главе мы научимся алгоритмически получать логические выводы из обучающих данных. Эти простые логические правила позволят нам обучать модели, не ограниченные геометрией данных.

22.1. АВТОМАТИЧЕСКОЕ ИЗУЧЕНИЕ ЛОГИЧЕСКИХ ПРАВИЛ

Проанализируем, казалось бы, тривиальную задачу. Предположим, что над лестницей висит лампочка, которая подсоединена к двум выключателям, один из которых находится у основания лестницы, а второй — наверху. Когда оба выключателя отключены, лампочка — тоже. При включении любого из них лампочка загорается, но при включении обоих снова тухнет. Такая схема позволяет зажечь свет, находясь внизу лестницы, и выключить после подъема по ней.

Представить состояния «включено/отключено» выключателей и лампочки можно с помощью двоичных цифр 0 и 1. Имея две переменные выключателей, `switch0` и `switch1`, можно легко показать, что лампочка включена всегда, когда `switch0 + switch1 == 1`. Можем ли мы, оперируя материалом из двух последних глав, обучить классификатор этой простой связи? Чтобы это выяснить, сохраним все возможные комбинации выключателей в двухстолбцовой матрице признаков `X`, после чего построим график ее строк в двухмерном пространстве, разметив каждую точку на основе соответствующего состояния лампочки «включено/отключено» (листинг 22.1; рис. 22.1). Полученный график позволит понять, как с этой задачей справятся KNN и линейные классификаторы.

Листинг 22.1. Построение графика задачи с двумя выключателями в 2D-пространстве

```
import numpy as np
import matplotlib.pyplot as plt
X = np.array([[0, 0], [1, 0], [0, 1], [1, 1]])
y = (X[:,0] + X[:,1] == 1).astype(int)

for i in [0, 1]:
    plt.scatter(X[y == i][:,0], X[y == i][:,1],
                marker=['o', 'x'][i], color=['b', 'k'][i],
                s=1000)
plt.xlabel('Switch 0')
plt.ylabel('Switch 1')
plt.show()
```

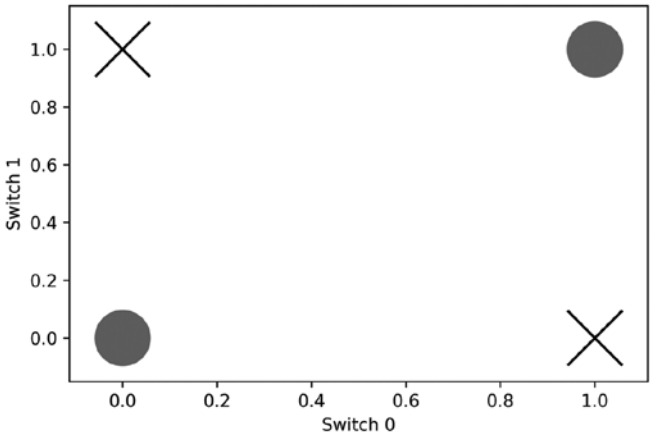


Рис. 22.1. График всех состояний системы освещения. Включенная лампочка представлена как ×, а отключенная — как ●. Ближайшими соседями каждого маркера ● являются маркеры × (и наоборот). Выходит, что KNN для классификации здесь использовать нельзя. Кроме того, между маркерами × и ● нет линейного разделения, значит, применить линейную классификацию тоже не получится

Четыре точки графика лежат в четырех углах квадрата. Пары точек, относящихся к одному классу, располагаются по его диагоналям, а все смежные точки принадлежат к разным классам. Два ближайших соседа каждой комбинации выключателей «включено» являются членами класса «отключено», и наоборот. Следовательно, KNN эти данные правильно классифицировать не сможет. Между размеченными классами отсутствует также линейное разделение, значит, нельзя провести линейную границу без пересечения диагонали, соединяющей две точки одного класса, и обучение линейного классификатора тоже отпадает. Что же делать? Один из вариантов — определить в качестве предиктивной модели две вложенные инструкции `if/else`. Давайте напишем и протестируем такой классификатор `if/else` (листинг 22.2).

Листинг 22.2. Классификация данных при помощи вложенных инструкций if/else

```
def classify(features):
    switch0, switch1 = features

    if switch0 == 0:
        if switch1 == 0:
            prediction = 0
        else:
            prediction = 1
    else:
        if switch1 == 0:
            prediction = 1
        else:
            prediction = 0
```

```
    return prediction

for i in range(X.shape[0]):
    assert classify(X[i]) == y[i]
```

Наш классификатор `if/else` точен на 100 %, но мы его не обучали. Мы запрограммировали этот классификатор сами. Ручное построение модели не относится к машинному обучению с учителем, значит, нужно найти способ автоматически получать точные инструкции `if/else` из обучающих данных. Разберемся, как это сделать.

Начнем с простого обучающего примера. Наша обучающая выборка представляет собой серию записанных наблюдений зависимости между одним выключателем и одной лампочкой. Если выключатель включен, лампочка тоже, и наоборот. Мы случайно включаем/выключаем лампочку и записываем наблюдения. Состояние лампочки фиксируется в массиве `y_simple`. Единственный признак, соответствующий выключателю, записан в одностолбцовой матрице `X_simple`. Естественно, `X_simple[i][0]` всегда будет равняться `y[i]`. Сгенерируем эту простую обучающую выборку (листинг 22.3).

Листинг 22.3. Генерация обучающей выборки с одним выключателем

```
np.random.seed(0)
y_simple = np.random.binomial(1, 0.5, size=10)
X_simple = np.array([[e] for e in y_simple])
print(f"features: {X_simple}")
print(f"\nlabels: {y_simple}")

features: [[1]
  [1]
  [1]
  [1]
  [0]
  [1]
  [0]
  [1]
  [1]
  [0]]

labels: [1 1 1 1 0 1 0 1 1 0]
```

Состояние лампочки симулируется путем случайного подбрасывания монеты

Состояние выключателя всегда равно состоянию лампочки

Далее подсчитаем все исходы, в которых и выключатель, и лампочка отключены (листинг 22.4).

Листинг 22.4. Подсчет совпадений состояний «отключено»

```
count = (X_simple[:,0][y_simple == 0] == 0).sum()
print(f"In {count} instances, both the switch and the light are off")

In 3 instances, both the switch and the light are off
```

Теперь подсчитаем случаи, в которых выключатель и лампочка включены (листинг 22.5).

Листинг 22.5. Подсчет совпадений состояния «включено»

```
count = (X_simple[:,0][y_simple == 1] == 1).sum()
print(f"In {count} instances, both the switch and the light are on")
```

```
In 7 instances, both the switch and the light are on
```

Эти совпадения пригодятся при обучении классификатора. Давайте более систематизированно отразим эти подсчеты в матрице совпадений `M`. Ее строки отражают состояние «включено/отключено» выключателя, а столбцы — состояние лампочки. Каждый элемент `M[i][j]` показывает количество случаев, когда выключатель находится в состоянии `i`, а лампочка — в состоянии `j`. Таким образом, `M[0][0]` должно равняться 7, а `M[1][1]` — 3.

Теперь для вычисления матрицы совпадений определим функцию `get_co_occurrence`, которая будет получать обучающую выборку `(X, y)`, а также индекс столбца `col`, возвращая совпадения между всеми классами в `y` и всеми состояниями признаков в `X[:,col]` (листинг 22.6).

Листинг 22.6. Вычисление матрицы совпадений

```
def get_co_occurrence(X, y, col=0):
    co_occurrence = []
    for i in [0, 1]:
        counts = [(X[:,col][y == i] == j).sum()
                  for j in [0, 1]]
        co_occurrence.append(counts)

    return np.array(co_occurrence)

M = get_co_occurrence(X_simple, y_simple)
assert M[0][0] == 3
assert M[1][1] == 7
print(M)
```

```
[[3 0]
 [0 7]]
```

Используя `get_co_occurrence`, мы вычислили матрицу `M`. Все совпадения лежат вдоль ее диагонали. Лампочка никогда не бывает включена, если выключатель находится в состоянии «отключено», и наоборот. Но давайте предположим, что выключатель неисправен. Мы его выключаем, но лампочка продолжает гореть. Добавим это аномальное наблюдение в наши данные и еще раз вычислим матрицу `M` (листинг 22.7).

Листинг 22.7. Добавление в данные несовпадения из-за поломки

```
X_simple = np.vstack([X_simple, [1]])
y_simple = np.hstack([y_simple, [0]])
M = get_co_occurrence(X_simple, y_simple)
print(M)

[[3 1]
 [0 7]]
```

При отключении выключателя лампочка в большинстве случаев тоже гаснет, но не всегда. Как нам точно спрогнозировать ее состояние, если известно, что выключатель отключен? Чтобы это выяснить, необходимо разделить `M[0]` на `M[0].sum()`. В результате мы получим распределение вероятностей возможных состояний лампочки при состоянии выключателя, равном `0` (листинг 22.8).

Листинг 22.8. Вычисление вероятностей состояний лампочки при выключенном выключателе

```
bulb_probs = M[0] / M[0].sum()
print("When the switch is set to 0, the bulb state probabilities are:")
print(bulb_probs)

prob_on, prob_off = bulb_probs
print(f"\nThere is a {100 * prob_on:.0f}% chance that the bulb is off.")
print(f"There is a {100 * prob_off:.0f}% chance that the bulb is on.")

When the switch is set to 0, the bulb state probabilities are:
[0.75 0.25]

There is a 75% chance that the bulb is off.
There is a 25% chance that the bulb is on.
```

Когда выключатель отключен, следует предположить, что и лампочка тоже. Такая догадка будет верна в 75 % случаев. Эта доля верности вписывается в наше определение общей точности из главы 20, значит, при отключенном выключателе состояние лампочки можно спрогнозировать с 75%-ной точностью.

Теперь оптимизируем эту точность для сценария, в котором выключатель включен (листинг 22.9). Начнем с вычисления `bulb_probs` для `M[1]`. Затем выберем состояние выключателя, соответствующее максимальной вероятности. По сути, мы делаем вывод, что состояние лампочки равно `bulb_probs.argmax()` с показателем точности `bulb_probs.max()`.

Листинг 22.9. Прогнозирование состояния лампочки при включенном выключателе

```
bulb_probs = M[1] / M[1].sum()
print("When the switch is set to 1, the bulb state probabilities are:")
print(bulb_probs)

prediction = ['off', 'on'][bulb_probs.argmax()]
accuracy = bulb_probs.max()
```

```
print(f"\nWe assume the bulb is {prediction} with "
      f"{100 * accuracy:.0f}% accuracy")

When the switch is set to 1, the bulb state probabilities are:
[0. 1.]

We assume the bulb is on with 100% accuracy
```

Когда выключатель отключен, мы с 75%-ной точностью предполагаем, что лампочка тоже отключена. Если же он включен, мы со 100%-ной точностью предполагаем, что и лампочка горит. Как совместить эти значения в единый показатель точности? В качестве наивного решения можно просто усреднить 0,75 и 1,00, но такой подход ошибочен. Эти две точности не должны взвешиваться поровну, так как выключатель находится в состоянии «включено» почти вдвое чаще, чем в состоянии «отключено». Подтвердить это можно сложением содержимого столбцов матрицы совпадений `M`. Выполнив `M.sum(axis=1)`, мы вернем количество состояний `off` и `on` выключателя (листинг 22.10).

Листинг 22.10. Подсчет состояний on и off выключателя

```
for i, count in enumerate(M.sum(axis=1)):
    state = ['off', 'on'][i]
    print(f"The switch is {state} in {count} observations.")

The switch is off in 4 observations.
The switch is on in 7 observations.
```

Выключатель чаще включен, чем выключен. В связи с этим для нахождения адекватного показателя точности нужно получить взвешенное среднее от 0,75 и 1,00. Веса должны соответствовать количествам состояний «включено/отключено» выключателя, полученным из `M` (листинг 22.11).

Листинг 22.11. Вычисление точности

```
accuracies = [0.75, 1.0]
total_accuracy = np.average(accuracies, weights=M.sum(axis=1))
print(f"Our total accuracy is {100 * total_accuracy:.0f}%")

Our total accuracy is 91%
```

Если выключатель в состоянии «отключено», мы прогнозируем, что и лампочка тоже. В противном случае прогноз гласит, что лампочка горит. Эта модель точна на 91 %. Более того, в Python ее можно представить как простую инструкцию `if/else`. Самое же главное, что эту модель можно обучить с нуля, следуя такому алгоритму.

1. Выбрать признак в матрице признаков `X`.
2. Подсчитать совпадения между двумя возможными состояниями признака и двумя типами классов. Сохранить количество этих совпадений в матрице `M` размером 2×2.

3. Для строки `i` в `M` вычислить распределение вероятностей классов, когда признак находится в состоянии `i`. Это распределение вероятностей равно `M[i] / M[i].sum()`. В `M` всего две строки, значит, можно сохранить распределения в двух переменных, `probs0` и `probs1`.
4. Определить часть `if` нашей условной модели. Если признак равен 0, мы возвращаем метку `probs0.argmax()`. Это максимизирует общую точность инструкции `if`, равную `probs0.max()`.
5. Определить часть `else` условной модели. Когда признак не равен 0, мы возвращаем метку `probs1.argmax()`. Это максимизирует точность инструкции `else`, равную `probs1.max()`.
6. Совместить инструкции `if` и `else` в единую инструкцию `if/else`. Иногда `probs0.argmax()` будет равняться `probs1.argmax()`. В таких случаях использование инструкции `if/else` излишне. Вместо этого можно вернуть обычное правило `f"prediction = {probs0.argmax()}"`.
7. Общая точность объединенной инструкции `if/else` равна взвешенному среднему `probs0.max()` и `probs1.max()`. Веса соответствуют количеству состояний признаков, полученных сложением содержимого столбцов `M`.

Теперь для выполнения этих семи шагов мы определим функцию `train_if_else`, которая будет возвращать обученную инструкцию `if/else` вместе с соответствующей точностью (листинг 22.12).

Листинг 22.12. Обучение простой модели if/else

```
def train_if_else(X, y, feature_col=0, feature_name='feature'):
    M = get_co_occurrence(X, y, col=feature_col)
    probs0, probs1 = [M[i] / M[i].sum() for i in [0, 1]]

    if_else = f"""if {feature_name} == 0:
    prediction = {probs0.argmax()}
else:
    prediction = {probs1.argmax()}
    """.strip()

    if probs0.argmax() == probs1.argmax():
        if_else = f"prediction = {probs0.argmax()}"

    accuracies = [probs0.max(), probs1.max()]
    total_accuracy = np.average(accuracies, weights=M.sum(axis=1))
    return if_else, total_accuracy

if_else, accuracy = train_if_else(X_simple, y_simple, feature_name='switch')
print(if_else)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")
```

Обучает инструкцию if/else на выборке (X, y) и возвращает письменную инструкцию вместе с соответствующей точностью. Эта инструкция обучается на признаке в X[:,feature_col]. Соответствующее имя признака хранится в feature_name

Создает письменную инструкцию if/else

Если обе части условной инструкции возвращают одинаковый прогноз, упрощаем эту инструкцию до этого прогноза

```
if switch == 0:
    prediction = 0
else:
    prediction = 1

This statement is 91% accurate.
```

Мы можем обучить простую модель `if/else` с помощью одного признака. Далее разберемся, как обучать вложенную модель `if/else`, используя два признака. Впоследствии мы расширим эту логику на большее число признаков.

22.1.1. Обучение вложенной модели if/else на двух признаках

Вернемся к нашей системе с двумя выключателями, соединенными с одной лампочкой над лестницей. Напомню, что все состояния этой системы представлены набором данных `(X, y)`, сгенерированным в листинге 22.1. Два признака, `switch0` и `switch1`, соответствуют столбцам `0` и `1` матрицы `X`. Однако функция `train_if_else` может обучаться только на одном столбце одновременно. Значит, мы обучим две отдельные модели: одну на `switch0`, а другую на `switch1` (листинг 22.13). Насколько хорошо справится каждая из них? Это мы выясним через вывод показателей их точности.

Листинг 22.13. Обучение моделей на системе с двумя выключателями

```
feature_names = [f"switch{i}" for i in range(2)]
for i, name in enumerate(feature_names):
    _, accuracy = train_if_else(X, y, feature_col=i, feature_name=name)
    print(f"The model trained on {name} is {100 * accuracy:.0f}% "
          "accurate.")

The model trained on switch0 is 50% accurate.
The model trained on switch1 is 50% accurate.
```

Обе модели ужасны! Одной инструкции `if/else` недостаточно, чтобы охватить всю сложность задачи. Что же делать? Как вариант можно разбить задачу на части, обучив две отдельные модели: модель A будет рассматривать только сценарии, где `switch0` отключен, а модель B — все оставшиеся случаи, в которых `switch0` включен. Затем совместим эти модели в единый согласованный классификатор.

Начнем с первого сценария, в котором `switch0` выключен. Когда он выключен, `X[:,0] == 0`. Значит, мы начинаем с изолирования обучающей подвыборки, удовлетворяющей данному логическому условию, сохраняя ее в переменных `X_switch0_off` и `y_switch0_off` (листинг 22.14).

Листинг 22.14. Изолирование обучающей подвыборки, в которой switch0 отключен

```
is_off = X[:,0] == 0
X_switch0_off = X[is_off]
y_switch0_off = y[is_off]
print(f"Feature matrix when switch0 is off:\n{X_switch0_off}")
print(f"\nClass labels when switch0 is off:\n{y_switch0_off}")

Feature matrix when switch0 is off:
[[0 0]
 [0 1]]

Class labels when switch0 is off:
[0 1]
```

Все элементы столбца 0 теперь равны 0

В этой подвыборке `switch0` всегда выключен, значит, `X_switch0_off[:,0]` всегда равно нулю. Этот нулевой столбец теперь излишен, и его можно удалить с помощью функции NumPy `np.delete` (листинг 22.15).

Листинг 22.15. Удаление лишнего столбца признаков

```
X_switch0_off = np.delete(X_switch0_off, 0, axis=1)
print(X_switch0_off)

[[0]
 [1]]
```

Столбец 0 был удален

Выполнение np.delete(X, r) возвращает копию X, где строка r удалена, а выполнение np.delete(X, c, axis=1) возвращает копию X, где удален столбец c. Здесь мы удаляем лишний столбец 0

Далее мы обучим модель `if/else` на изолированной подвыборке (листинг 22.16). Эта модель прогнозирует включение лампочки, исходя из состояния `switch1`. Прогнозы действительны, только если `switch0` выключен. Модель мы сохраняем в переменной `switch0_off_model`, а показатель ее точности — в соответствующей переменной `switch0_off_accuracy`.

Листинг 22.16. Обучение модели при выключенном switch0

```
results = train_if_else(X_switch0_off, y_switch0_off,
                        feature_name='switch1')
switch0_off_model, off_accuracy = results
print("If switch 0 is off, then the following if/else model is "
      f"{100 * off_accuracy:.0f}% accurate.\n\n{switch0_off_model}")

If switch 0 is off, then the following if/else model is 100% accurate.

if switch1 == 0:
    prediction = 0
else:
    prediction = 1
```

При выключенном `switch0` обученная модель `if/else` может спрогнозировать состояние лампочки со 100%-ной точностью. Теперь мы обучим соответствующую модель охватывать все случаи, в которых `switch0` включен. Начнем с фильтрации обучающих данных на основе условия `X[:,0] == 1` (листинг 22.17).

Листинг 22.17. Изолирование обучающей подвыборки, где switch0 включен

```
def filter_X_y(X, y, feature_col=0, condition=0):
    inclusion_criteria = X[:,feature_col] == condition
    y_filtered = y[inclusion_criteria]
    X_filtered = X[inclusion_criteria]
    X_filtered = np.delete(X_filtered, feature_col, axis=1)
    return X_filtered, y_filtered

X_switch0_on, y_switch0_on = filter_X_y(X, y, condition=1)
```

Фильтрует обучающие данные на основе признака в столбце feature_col матрицы X. Возвращает подвыборку обучающих данных, где этот признак соответствует установленному значению условия

Логический массив, в котором i элемент True, если X[i][feature_col] равно условию

Столбец feature_col становится лишним, поскольку все отфильтрованные значения равны условию. В связи с этим данный столбец из обучающих данных исключается

Далее мы обучим `switch0_on_model` на отфильтрованной обучающей выборке (листинг 22.18).

Листинг 22.18. Обучение модели при включенном switch0

```
results = train_if_else(X_switch0_on, y_switch0_on,
                        feature_name='switch1')
switch0_on_model, on_accuracy = results
print("If switch 0 is on, then the following if/else model is "
      f"{100 * on_accuracy:.0f}% accurate.\n\n{switch0_on_model}")

If switch 0 is on, then the following if/else model is 100% accurate.

if switch1 == 0:
    prediction = 1
else:
    prediction = 0
```

Если `switch == 0`, тогда `switch0_off_model` справляется со 100%-ной точностью. Во всех остальных случаях такую же 100%-ную точность демонстрирует `switch1_on_model`. Эти модели можно легко объединить в одну вложенную инструкцию `if/else`. Далее мы определим функцию `combine_if_else`, выполняющую слияние двух отдельных инструкций `if/else`, после чего применим ее к нашим двум моделям (листинг 22.19).

Листинг 22.19. Совмещение отдельных моделей if/else

```
def combine_if_else(if_else_a, if_else_b, feature_name='feature'):
    return f"""
if {feature_name} == 0:
{add_indent(if_else_a)}
else:
{add_indent(if_else_b)}
""".strip()
```

Совмещает две инструкции if/else, if_else_a и if_else_b в единую вложенную инструкцию

Каждой инструкции во время вложения добавляется стандартный отступ Python из четырех пробелов

```
def add_indent(if_else):
    return '\n'.join([4 * ' ' + line for line in if_else.split('\n')])

nested_model = combine_if_else(switch0_off_model, switch0_on_model,
                               feature_name='switch0')
print(nested_model)

if switch0 == 0:
    if switch1 == 0:
        prediction = 0
    else:
        prediction = 1
else:
    if switch1 == 0:
        prediction = 1
    else:
        prediction = 0
```

Эта дополнительная функция помогает отделить все инструкции во время вложения

Мы воссоздали вложенную модель `if/else` из листинга 22.2. Эта модель на 100 % точна. Убедиться в этом можно, получив взвешенное среднее `off_accuracy` и `on_accuracy`. Эти точности соответствуют состояниям `off/on` выключателя `switch0`, значит, их веса соответствуют количествам `off/on`, связанным со `switch0`. Эти количества равны длинам массивов `y_switch0_off` и `y_switch0_on`. Далее мы получим взвешенное среднее и подтвердим, что общая точность равна 1 (листинг 22.20).

Листинг 22.20. Вычисление общей вложенной точности

```
accuracies = [off_accuracy, on_accuracy]
weights = [y_switch0_off.size, y_switch0_on.size]
total_accuracy = np.average(accuracies, weights=weights)
print(f"Our total accuracy is {100 * total_accuracy:.0f}%")

Our total accuracy is 100%
```

Мы можем генерировать вложенную модель, обученную на двух признаках, автоматически. Наша стратегия строится на создании отдельных обучающих выборок. Это разделение определяется состояниями `on/off` признаков. Такой тип деления называется *двоичным разбиением*. Как правило, мы делим обучающую выборку `(X, y)`, используя два параметра:

- признак `i`, соответствующий столбцу `i` в `X`. Например, `switch0` в столбце `0` матрицы `X`;
- условие `c`, где `X[:,i] == c` является `True` для некоторых точек данных, но не всех. Например, условие `0`, соответствующее состоянию `off`.

Деление по признаку `i` и условию `c` можно выполнить так.

1. Получить обучающую подвыборку `(X_a, y_a)`, в которой `X_a[:,i] == c`.
2. Получить обучающую подвыборку `(X_b, y_b)`, в которой `X_b[:,i] != c`.

3. Удалить столбец `i` из `X_a` и `X_b`.
4. Вернуть отделенные подвыборки `(X_a, y_a)` и `(X_b, y_b)`.

ПРИМЕЧАНИЕ

Эти шаги не предполагают выполнения для непрерывного признака. Позже в этой главе мы рассмотрим, как преобразовывать такие признаки в двоичные переменные для деления.

В листинге 22.21 определяется функция `split` для выполнения перечисленных ранее шагов. Далее мы внедрим ее в конвейер систематического обучения.

Листинг 22.21. Определение функции двоичного разбиения

Выполняет двоичное разбиение для признака в столбце feature_col матрицы признаков X

В результате деления получаются две обучающие подвыборки, (X_a, y_a) и (X_b, y_b). В первой признак X_a[:,feature_col] всегда равен условию

```
def split(X, y, feature_col=0, condition=0):
    has_condition = X[:,feature_col] == condition
    X_a, y_a = [e[has_condition] for e in [X, y]]
    X_b, y_b = [e[~has_condition] for e in [X, y]
    X_a, X_b = [np.delete(e, feature_col, axis=1) for e in [X_a, X_b]]
    return [X_a, X_b, y_a, y_b]

X_a, X_b, y_a, y_b = split(X, y)
assert np.array_equal(X_a, X_switch0_off)
assert np.array_equal(X_b, X_switch0_on)
```

Во второй обучающей выборке X_a[:,feature_col] никогда не равен условию

Выполнив разделение по `switch0`, мы смогли обучить вложенную модель. До этого деления мы сначала пробовали обучить простые модели `if/else`. Справлялись они ужасно, и у нас не оставалось выбора, кроме как разделить обучающие данные. Однако обученные вложенные модели все равно нужно сравнивать с простыми моделями, возвращаемыми `train_if_else`. Если более простая модель покажет сопоставимые результаты, то возвращать необходимо ее.

ПРИМЕЧАНИЕ

Вложенная модель на двух признаках никогда не будет работать хуже простой модели, основанной на одном. Однако есть вероятность, что две эти модели будут справляться одинаково. В таком случае стоит следовать правилу бритвы Оккама: когда две соперничающие теории дают одинаковые прогнозы, лучшей будет более простая.

Теперь оформим процесс обучения вложенных моделей на двух признаках. Имея обучающую выборку `(X, y)`, мы проделываем следующие шаги.

1. Выбираем признак `i`, по которому нужно выполнить деление. Изначально он указывается с помощью параметра. Позднее мы научимся выбирать его автоматически.
2. Пробуем обучить простую модель на одном признаке `i`. Если эта модель справится со 100%-ной точностью, возвращаем в выводе ее.

Теоретически можно обучать модели с одним признаком на столбцах `0` и `1`, используя `train_if_else`. Это позволит систематически сравнивать все модели на одном признаке. Однако такой подход не будет масштабироваться при увеличении числа признаков с двух до *N*.

3. Выполняем деление по признаку `i`, применяя `split`. Эта функция вернет две обучающие выборки, `(X_a, y_a)` и `(X_b, y_b)`.
4. Обучаем две простые модели, `if_else_a` и `if_else_b`, используя обучающие выборки, возвращенные `split`. Соответствующие точности будут представлены как `accuracy_a` и `accuracy_b`.
5. Совмещаем `if_else_a` и `if_else_b` во вложенную условную модель `if/else`.
6. Вычисляем точность вложенной модели, применяя взвешенное среднее `accuracy_a` и `accuracy_b`. Веса при этом равны `y_a.size` и `y_b.size`.
7. Если вложенная модель превосходит простую модель из шага 2, возвращаем ее. В противном случае возвращаем простую.

Далее определим функцию `train_nested_if_else` для выполнения всех перечисленных шагов. Она возвращает обученную модель и ее показатель точности (листинг 22.22).

Листинг 22.22. Обучение вложенной модели if/else

```
def train_nested_if_else(X, y, split_col=0,
                         feature_names=['feature1', 'feature1']):
    split_name = feature_names[split_col]
    simple_model, simple_accuracy = train_if_else(X, y, split_col,
                                                  split_name)
    if simple_accuracy == 1.0:
        return (simple_model, simple_accuracy)

    X_a, X_b, y_a, y_b = split(X, y, feature_col=split_col)
    in_name = feature_names[1 - split_col]
    if_else_a, accuracy_a = train_if_else(X_a, y_a, feature_name=in_name)
    if_else_b, accuracy_b = train_if_else(X_b, y_b, feature_name=in_name)
    nested_model = combine_if_else(if_else_a, if_else_b, split_name)
    accuracies = [accuracy_a, accuracy_b]
    nested_accuracy = np.average(accuracies, weights=[y_a.size, y_b.size])
    if nested_accuracy > simple_accuracy:
        return (nested_model, nested_accuracy)

    return (simple_model, simple_accuracy)

feature_names = ['switch0', 'switch1']
model, accuracy = train_nested_if_else(X, y, feature_names=feature_names)
print(model)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")
```

Обучает вложенную инструкцию if/else на выборке (X, y) и возвращает письменную инструкцию вместе с соответствующей точностью. Эта инструкция обучается путем разделения по признаку в X[:,split_col]. Имена признаков в ней сохранены в массиве feature_names

Имена признаков в инструкции хранятся в массиве feature_names

Имя признака, расположенного во внутренней части вложенной инструкции

Совмещает простые модели

Обучает две простые модели

```
if switch0 == 0:
    if switch1 == 0:
        prediction = 0
    else:
        prediction = 1
else:
    if switch1 == 0:
        prediction = 1
    else:
        prediction = 0

This statement is 100% accurate.
```

Наша функция обучила модель, имеющую 100%-ную точность. Относительно текущего набора данных эта точность должна сохраняться, даже если разделить его по `switch1`, а не `switch0`. Проверим (листинг 22.23).

Листинг 22.23. Деление по switch1 вместо switch0

```
model, accuracy = train_nested_if_else(X, y, split_col=1,
                                       feature_names=feature_names)
print(model)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")

if switch1 == 0:
    if switch0 == 0:
        prediction = 0
    else:
        prediction = 1
else:
    if switch0 == 0:
        prediction = 1
    else:
        prediction = 0

This statement is 100% accurate.
```

Деление по любому из признаков дает одинаковый результат. Это верно для нашей системы с двумя выключателями, но не сработает во многих реальных обучающих выборках. Нередко бывает, что один вариант деления превосходит другой. В следующем разделе мы узнаем, как расставлять приоритеты признаков во время деления.

22.1.2. Выбор предпочтительного признака для деления

Предположим, нам нужно обучить модель `if/else`, прогнозирующую, идет ли на улице дождь. Если дождь идет, она будет возвращать `1`, в противном случае — `0`. При этом модель будет опираться на следующие два признака.

- На дворе сейчас осень? Да или нет?

  Мы предположим, что осень является местным сезоном дождей, в связи с чем данный признак прогнозирует их выпадение в 60 % случаев.

- На улице повышенная влажность? Да или нет?

  Обычно дождь идет при повышенной влажности. Иногда же ее повышение обусловливается оросительными системами, работающими в солнечный день. При этом во время моросящего дождя в лесу влажность может казаться низкой, если листва деревьев препятствует падению его капель на землю. Предположим, что этот признак прогнозирует дождь в 95 % случаев.

Далее мы смоделируем эти признаки и метки классов с помощью случайной выборки, которую сделаем из 100 наблюдений погодных условий, сохранив результаты в обучающем наборе (`X_rain, y_rain`) (листинг 22.24).

Листинг 22.24. Моделирование обучающей выборки для дождливого дня

```
np.random.seed(1)
y_rain = np.random.binomial(1, 0.6, size=100)
is_wet = [e if np.random.binomial(1, 0.95) else 1 - e for e in y_rain]
is_fall = [e if np.random.binomial(1, 0.6) else 1 - e for e in y_rain]
X_rain = np.array([is_fall, is_wet]).T
```

Дождь идет в 60 % случаев

В 95 % случаев уровень повышенной влажности означает дождь

В 60 % случаев то, что сейчас осенний сезон, означает дождь

Теперь обучим модель, проведя разделение по признаку осени (листинг 22.25).

Листинг 22.25. Обучение модели путем деления по признаку осени

```
feature_names = ['is_autumn', 'is_wet']
model, accuracy = train_nested_if_else(X_rain, y_rain,
                                       feature_names=feature_names)
print(model)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")

if is_autumn == 0:
    if is_wet == 0:
        prediction = 0
    else:
        prediction = 1
else:
    if is_wet == 0:
        prediction = 0
    else:
        prediction = 1

This statement is 95% accurate.
```

Мы обучили вложенную модель, имеющую точность 95 %. А что, если вместо этого произвести деление по признаку повышенной влажности (листинг 22.26)?

Листинг 22.26. Обучение модели путем деления по признаку повышенной влажности

```
model, accuracy = train_nested_if_else(X_rain, y_rain, split_col=1,
                                       feature_names=feature_names)
print(model)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")
```

```
if is_wet == 0:
    prediction = 0
else:
    prediction = 1

This statement is 95% accurate.
```

Разделение на основе повышенной влажности дает более простую (а значит, лучшую) модель с сохранением той же точности. Не все разделения равнозначны: разбивка по некоторым признакам дает предпочтительные результаты. А как выбирать наиболее подходящий для разделения признак? Простейшим решением будет перебор всех признаков в `X`, обучение моделей путем разделения по каждому из них и возвращение простейшей модели с максимальной точностью. Такой подход с перебором работает, когда `X.size[1] == 2`, но при увеличении числа признаков масштабироваться не сможет. Наша же цель — выработать технику, которая способна работать при тысячах признаков, значит, нужно найти альтернативное решение.

Один из вариантов требует проверки распределения классов в обучающей выборке. Сейчас наш массив `y_rain` содержит два двоичных класса, `0` и `1`. Метка `1` соответствует дождливой погоде. Следовательно, сумма этого массива равна количеству наблюдений дождливой погоды. При этом размер данного массива равен общему числу наблюдений, значит, `y_rain.sum() / y_rain.size` соответствует общей вероятности дождя. Выведем ее (листинг 22.27).

Листинг 22.27. Вычисление вероятности дождя

```
prob_rain = y_rain.sum() / y_rain.size
print(f"It rains in {100 * prob_rain:.0f}% of our observations.")

It rains in 61% of our observations
```

Дождь наблюдается в 61 % общего числа наблюдений. Как эта вероятность меняется при разделении по признаку осени? Такое разделение возвращает две обучающие выборки с двумя массивами меток классов. Эти массивы мы назовем `y_fall_a` и `y_fall_b`. Деление `y_fall_b.sum()` на размер массива вернет вероятность дождя в осенний день. Выведем ее и вероятность дождя в другие времена года (листинг 22.28).

Листинг 22.28. Вычисление вероятности выпадения дождя в разные сезоны

```
y_fall_a, y_fall_b = split(X_rain, y_rain, feature_col=0)[-2:]
for i, y_fall in enumerate([y_fall_a, y_fall_b]):
    prob_rain = y_fall.sum() / y_fall.size
    state = ['not autumn', 'autumn'][i]
    print(f"It rains {100 * prob_rain:.0f}% of the time when it is "
          f"{state}")

It rains 55% of the time when it is not autumn
It rains 66% of the time when it is autumn
```

Как и ожидалось, дождь с большей вероятностью выпадает в осенний период, но разница в вероятностях здесь не очень значительна. Осенью дождь идет в 66 % случаев, а в другие времена года — в 55 %. Важно отметить, что эти два значения вероятности близки к общей вероятности наблюдения дождя, равной 61 %. Если нам известно, что на дворе осень, то уверенность в дождливой погоде несколько возрастает. И все же в отношении исходной обучающей выборки возрастает она несущественно, значит, разделение по признаку осени оказывается не особо информативным. А что, если произвести разделение по признаку повышенной влажности? Проверим, приведет ли это к изменению вероятностей (листинг 22.29).

Листинг 22.29. Вычисление вероятности дождя на основе повышенной влажности

```
y_wet_a, y_wet_b = split(X_rain, y_rain, feature_col=1)[-2:]
for i, y_wet in enumerate([y_wet_a, y_wet_b]):
    prob_rain = y_wet.sum() / y_wet.size
    state = ['not wet', 'wet'][i]
    print(f"It rains {100 * prob_rain:.0f}% of the time when it is "
          f"{state}")

It rains 10% of the time when it is not wet
It rains 98% of the time when it is wet
```

Если нам известно, что на дворе высокая влажность, то мы практически уверены в том, что идет дождь. Если же на дворе сухо, то вероятность дождя составляет 10 %. Это низкий процент, но для нашего классификатора он весьма значителен. Нам известно, что низкая влажность прогнозирует отсутствие дождя с точностью 90 %. Чисто интуитивно влажность оказывается более информативным признаком, нежели осень, но как оценить эту разницу количественно? Разделение по признаку влажности дает два массива с метками классов, отражающих либо низкую, либо очень высокую вероятность дождя. Эти крайние значения вероятностей указывают на дисбаланс между классами. Как мы узнали в главе 20, несбалансированный набор данных содержит существенно больше экземпляров класса A, чем класса B. В результате модели становится проще выделять в этих данных класс A. Для сравнения отмечу, что операция разделения по признаку осени возвращает два массива, вероятности которых находятся в умеренном диапазоне между 55 и 66 %.

Классы в `y_fall_a` и `y_fall_b` лучше сбалансированы, в связи с чем сложно различить классы «дождливо»/«сухо».

При выборе между двумя вариантами разделения следует склоняться к тому, которое дает менее сбалансированные метки классов. Далее мы выясним, как оценить дисбаланс классов количественно. Как правило, дисбаланс связан с формой распределения вероятностей классов. Это распределение можно рассматривать как вектор `v`, где `v[i]` соответствует вероятности наблюдения класса *i*. Более высокое значение `v.max()` указывает на больший дисбаланс классов. В нашем наборе данных с двумя классами можно вычислить `v` как `[1 - prob_rain, prob_rain]`,

где `prob_rain` отражает вероятность дождя. Этот двухэлементный вектор можно визуализировать в виде отрезка в двухмерном пространстве, как описывалось в главе 12 (рис. 22.2).

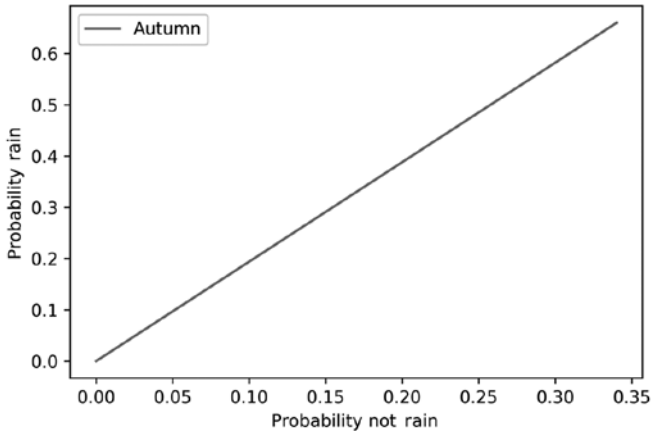


Рис. 22.2. Распределение вероятностей по меткам классов в y_fall_a (на дворе осень), визуализированное в виде двухмерного отрезка. Ось Y представляет вероятность дождя (0,66), а ось X — вероятность его отсутствия (1 – 0,66 = 0,36)

Подобные визуализации могут быть очень информативными, поэтому далее мы проделаем следующее.

1. Вычислим векторы распределения классов для разделения по признаку осени, используя массивы `y_fall_a` и `y_fall_b`.
2. Вычислим векторы распределения классов для разделения по признаку повышенной влажности с помощью массивов `y_wet_a` и `y_wet_b`.
3. Визуализируем все четыре массива в виде отрезков в двухмерном пространстве.

Эта визуализация покажет, как можно эффективно измерять дисбаланс классов (листинг 22.30; рис. 22.3).

Листинг 22.30. Построение векторов распределения классов

Возвращает распределение вероятностей по меткам классов в двоичной системе из двух классов. Это распределение можно рассматривать как двухмерный вектор

```
def get_class_distribution(y):
    prob_rain = y.sum() / y.size
    return np.array([1 - prob_rain, prob_rain])

def plot_vector(v, label, linestyle='-', color='b'):
    plt.plot([0, v[0]], [0, v[1]], label=label,
             linestyle=linestyle, c=color)
```

Строит двухмерный вектор в виде отрезка от начала координат до v

```
classes = [y_fall_a, y_fall_b, y_wet_a, y_wet_b]
distributions = [get_class_distribution(y) for y in classes]
labels = ['Not Autumn', 'Autumn', 'Not Wet', 'Wet']
colors = ['y', 'g', 'k', 'b']
linestyles = ['-.', ':', '-', '--']
for tup in zip(distributions, labels, colors, linestyles):
    vector, label, color, linestyle = tup
    plot_vector(vector, label, linestyle=linestyle, color=color)

plt.legend()
plt.xlabel('Probability Not Rain')
plt.ylabel('Probability Rain')
plt.axis('equal')
plt.show()
```

Перебирает четыре уникальных вектора распределения, получающихся в результате каждого возможного разделения, и отрисовывает их на графике

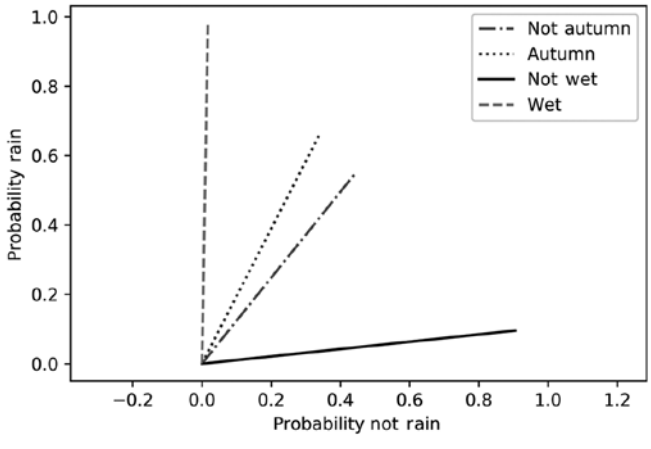


Рис. 22.3. График распределений четырех векторов разделения по каждому признаку. Векторы повышенной влажности намного менее сбалансированы, в связи с чем расположены ближе к осям. Самое важное — то, что эти векторы длиннее векторов разделения по признаку осени

В нашем графике два несбалансированных вектора влажности сильно наклонены в сторону осей *X* и *Y*. При этом два сбалансированных вектора осени примерно равноудалены от обеих осей. Однако выделяется здесь не направленность векторов, а их длина: сбалансированные векторы осени гораздо короче векторов, связанных с признаком влажности. И это не совпадение. Доказано, что несбалансированные распределения имеют более высокие значения абсолютной величины векторов. Кроме того, как показано в главе 13, абсолютная величина равна квадратному корню из `v @ v`. Таким образом, скалярное произведение вектора распределения с самим собой тем больше, чем это вектор менее сбалансирован.

Продемонстрируем это свойство для каждого двухмерного вектора `v = [1 - p, p]`, где `p` — вероятность дождя. Код листинга 22.31 графически отображает абсолютную величину `v`, отражающую вероятности дождя, располагающиеся на промежутке от 0 до 1. Он также отображает на графике квадрат абсолютной величины, равный `v @ v`. Отраженные на графике значения должны максимизироваться при очень низком или очень высоком `p` и минимизироваться, когда `v` идеально сбалансирован при `p = 0.5` (рис. 22.4).

Листинг 22.31. Построение графика абсолютных величин векторов распределений

```
prob_rain = np.arange(0, 1.001, 0.01)
vectors = [np.array([1 - p, p]) for p in prob_rain]
magnitudes = [np.linalg.norm(v) for v in vectors]
square_magnitudes = [v @ v for v in vectors]
plt.plot(prob_rain, magnitudes, label='Magnitude')
plt.plot(prob_rain, square_magnitudes, label='Squared Magnitude',
         linestyle='--')
plt.xlabel('Probability of Rain')
plt.axvline(0.5, color='k', label='Perfect Balance', linestyle=':')
plt.legend()
plt.show()
```

Вероятность дождя располагается в промежутке от 0 до 1 включительно

Векторы представляют все возможные распределения двух классов, где классами выступают дождливая и сухая погода

Абсолютные величины векторов, вычисленные с помощью NumPy

Вычисляет квадраты абсолютных величин векторов в виде простого скалярного произведения

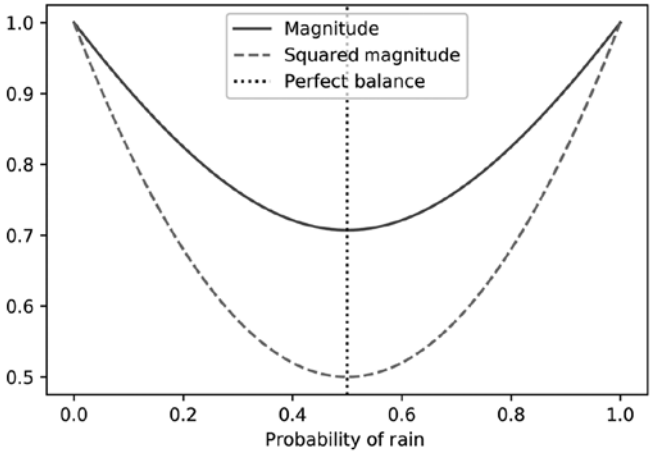


Рис. 22.4. График абсолютных величин векторов распределений и их возведенных в квадрат значений для каждого вектора распределения [1 – p, p]. Отображенные на графике значения минимизируются, когда вектор идеально сбалансирован при p = 0,5

Возведенная в квадрат абсолютная величина максимальна при 1, когда `v` находится в полном дисбалансе при `p = 0.0` и `p = 1.0`. Минимизируется же она, когда `p = 0.5`,

то есть v сбалансирован. Таким образом, v @ v служит превосходным показателем для оценки дисбаланса меток классов, но аналитики данных предпочитают иной, а именно 1 - v @ v. Этот показатель, называемый *коэффициентом Джини*, по сути, поворачивает кривую графика — она минимизируется при значении 0.0 и максимизируется при 0.5. Убедимся в этом, построив график коэффициента Джини для всех значений p (листинг 22.32; рис. 22.5).

ПРИМЕЧАНИЕ

В теории вероятностей коэффициент Джини имеет конкретную интерпретацию. Предположим, что любой точке данных случайно присваивается класс i с вероятностью v[i], где v — векторизованное распределение. Вероятность выбора точки, принадлежащей к классу i, также равна v[i]. Таким образом, вероятность выбора точки, принадлежащей к классу i, и верной разметки этой точки равна v[i] * v[i]. Из этого следует, что вероятность верной разметки любой точки равна sum(v[i] * v[i] for i in range(len(v)). Это выражение упрощается до v @ v. Таким образом, 1 - v @ v равно вероятности ошибочной разметки данных. Коэффициент Джини равен вероятности ошибки, которая уменьшается по мере роста дисбаланса в данных.

Листинг 22.32. Построение графика коэффициентов Джини

```
gini_impurities = [1 - (v @ v) for v in vectors]
plt.plot(prob_rain, gini_impurities)
plt.xlabel('Probability of Rain')
plt.ylabel('Gini Impurity')
plt.show()
```

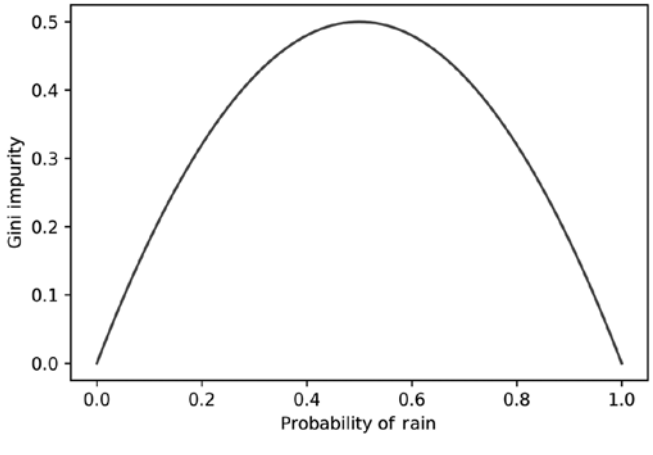


Рис. 22.5. График коэффициентов Джини для каждого вектора распределения [1 - p, p]. Коэффициент Джини максимален, когда вектор идеально сбалансирован при p = 0.5

Коэффициент Джини — это стандартная мера дисбаланса классов. Плохо сбалансированные наборы данных считаются более «чистыми», поскольку их метки могут сильно склоняться в сторону одного класса. При обучении вложенной модели следует делать разделение на основе признака, который минимизирует общий коэффициент Джини. Для любого разделения с метками классов `y_a` и `y_b` это его общее значение можно найти так.

1. Вычислить коэффициент Джини для `y_a`. Он равен `1 - v_a @ v_a`, где `v_a` — это распределение классов в `y_a`.
2. Вычислить коэффициент Джини для `y_b`. Он равен `1 - v_b @ v_b`, где `v_b` — распределение классов в `y_b`.
3. Наконец, получить взвешенное среднее этих двух коэффициентов. Их веса будут равны `y_a.size` и `y_b.size`, как и при вычислении общей точности.

Вычислим коэффициенты Джини, связанные с признаками осени и повышенной влажности (листинг 22.33).

Листинг 22.33. Вычисление коэффициента Джини для каждого признака

```
def compute_impurity(y_a, y_b):
    v_a = get_class_distribution(y_a)
    v_b = get_class_distribution(y_b)
    impurities = [1 - v @ v for v in [v_a, v_b]]
    weights = [y.size, y_b.size]
    return np.average(impurities, weights=weights)

fall_impurity = compute_impurity(y_fall_a, y_fall_b)
wet_impurity = compute_impurity(y_wet_a, y_wet_b)
print(f"When we split on Autumn, the Impurity is {fall_impurity:0.2f}.")
print(f"When we split on Wetness, the Impurity is {wet_impurity:0.2f}.")
```

Возвращает взвешенный коэффициент Джини, связанный с метками классов, хранящимися в массивах y_a и y_b

Вектор распределения классов в y_a

Вектор распределения классов в y_b

Коэффициенты Джини для двух распределений классов

Возвращает взвешенное среднее двух коэффициентов Джини

```
When we split on Autumn, the Impurity is 0.45.
When we split on Wetness, the Impurity is 0.04.
```

Как и ожидалось, коэффициент Джини минимизируется при разделении по признаку влажности. Это разделение ведет к менее сбалансированным обучающим данным, что упрощает обучение классификатора. Далее мы будем выполнять разделение на основе признаков, имеющих минимальный коэффициент Джини. С учетом этого определим функцию `sort_feature_indices`, которая будет получать обучающую выборку `(X, y)` и возвращать список индексов признаков, упорядоченных на основе коэффициента Джини, связанного с разделением по каждому из этих признаков (листинг 22.34).

Листинг 22.34. Упорядочение признаков на основе коэффициента Джини

```
def sort_feature_indices(X, y):
    feature_indices = range(X.shape[1])
    impurities = []

    for i in feature_indices:
        y_a, y_b = split(X, y, feature_col=i)[-2:]
        impurities.append(compute_impurity(y_a, y_b))

    return sorted(feature_indices, key=lambda i: impurities[i])

indices = sort_feature_indices(X_rain, y_rain)
top_feature = feature_names[indices[0]]
print(f"The feature with the minimal impurity is: '{top_feature}'")

The feature with the minimal impurity is: 'is_wet'
```

Упорядочивает индексы признаков в X по связанным с ними коэффициентам Джини в порядке возрастания

Выполняет разделение по признакам в столбце i и вычисляет коэффициент Джини полученного разделения

Возвращает упорядоченные индексы столбцов в X. Первый столбец соответствует минимальному коэффициенту Джини

Функция `sort_feature_indices` окажется бесценной при обучении вложенных моделей `if/else` с использованием более двух признаков.

22.1.3. Обучение моделей if/else с помощью более чем двух признаков

Обучение модели прогнозированию текущей погоды — довольно простая задача. Далее же мы обучим более сложный ее вариант, предсказывающий вероятность дождя на завтра. Эта модель будет опираться на следующие три признака.

- Шел ли дождь сегодня? Да или нет?

  Если дождь сегодня шел, значит, завтра его вероятность очень высока.

- Облачный ли сегодня день? Да или нет?

  В облачный день вероятность дождя выше, что увеличивает вероятность его выпадения завтра.

- Сегодня осенний день? Да или нет?

  Мы предположим, что осенью чаще идут дожди и наблюдается облачность.

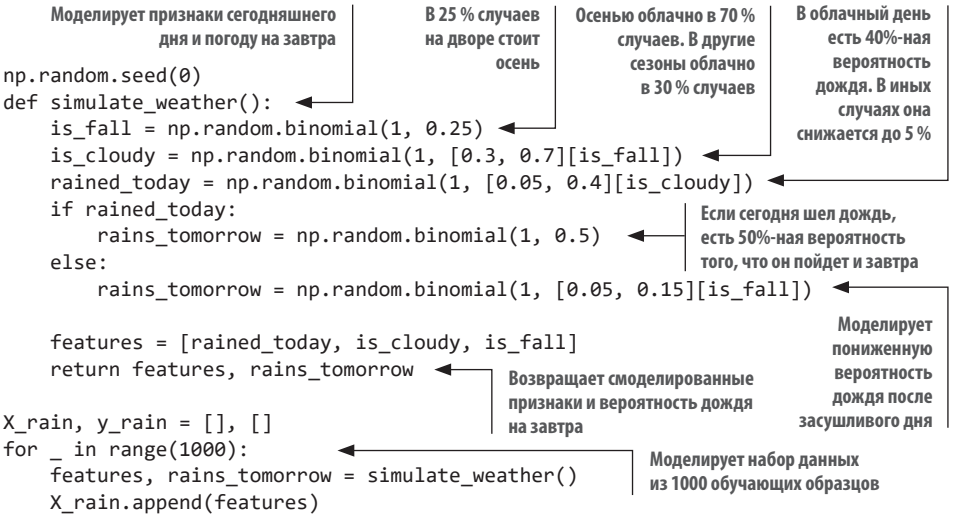
А чтобы сделать задачу поинтереснее, предположим также наличие между этими тремя признаками сложной, но реалистичной связи.

- Вероятность того, что сегодня осенний день, составляет 25 %.
- Осенью в 70 % случаев стоит облачная погода. В другие сезоны облачность наблюдается в 30 % случаев.

- Если сегодня облачно, то есть 40%-ная вероятность того, что в итоге пойдет дождь. В противном случае вероятность дождя равна 5 %.
- Если сегодня идет дождь, есть 50%-ная вероятность того, что он пойдет и завтра.
- Если сегодня стоит сухой и солнечный осенний день, то есть 15%-ная вероятность того, что завтра пойдет дождь. В сухие солнечные дни весны, лета и зимы вероятность дождя на завтра составляет 5 %.

Код листинга 22.35 моделирует обучающую выборку (`X_rain, y_rain`) на основе вероятностной связи между указанными признаками.

Листинг 22.35. Моделирование обучающей выборки с тремя признаками




```
np.random.seed(0)
def simulate_weather():
    is_fall = np.random.binomial(1, 0.25)
    is_cloudy = np.random.binomial(1, [0.3, 0.7][is_fall])
    rained_today = np.random.binomial(1, [0.05, 0.4][is_cloudy])
    if rained_today:
        rains_tomorrow = np.random.binomial(1, 0.5)
    else:
        rains_tomorrow = np.random.binomial(1, [0.05, 0.15][is_fall])

    features = [rained_today, is_cloudy, is_fall]
    return features, rains_tomorrow

X_rain, y_rain = [], []
for _ in range(1000):
    features, rains_tomorrow = simulate_weather()
    X_rain.append(features)
    y_rain.append(rains_tomorrow)

X_rain, y_rain = np.array(X_rain), np.array(y_rain)
```

Столбцы в `X_rain` соответствуют признакам `'is_fall'`, `'is_cloudy'` и `'rained_today'`. Мы можем упорядочить эти признаки по коэффициенту Джини, чтобы оценить, насколько хорошо они разделяют данные (листинг 22.36).

Листинг 22.36. Упорядочение трех признаков по коэффициенту Джини

```
feature_names = ['rained_today', 'is_cloudy', 'is_fall']
indices = sort_feature_indices(X_rain, y_rain)
print(f"Features sorted by Gini Impurity:")
print([feature_names[i] for i in indices])

Features sorted by Gini Impurity:
['is_fall', 'is_cloudy', 'rained_today']
```

Разделение по признаку осени дает минимальный коэффициент Джини. Вторым идет признак облачности. Признак дождя имеет наивысший коэффициент Джини — он дает наиболее сбалансированные наборы данных, а значит, это неудачный вариант для разделения.

ПРИМЕЧАНИЕ

Высокий коэффициент Джини признака дождя может удивить. Казалось бы, если сегодня шел дождь, то мы с гораздо большей вероятностью предположим, что дождь пойдет и завтра. Таким образом, когда X_rain[:,0] == 1, коэффициент Джини оказывается мал. Однако в солнечный день нам ничто не указывает на вероятность той или иной погоды завтра. Поэтому, когда X_rain[:,0] == 0, коэффициент Джини оказывается высоким. В течение года гораздо больше солнечных дней, чем дождливых, поэтому средний коэффициент Джини высок. В противоположность этому признак осени является гораздо более информативным. Он дает нам представление о завтрашней погоде как осенью, так и в другое время года.

Как же правильно обучить модель, располагая этим списком ранжированных признаков? Все же `trained_nested_if_else` должна обрабатывать два признака, а не три. Интуитивным решением здесь будет обучение модели на двух самых подходящих признаках. Они дадут больший дисбаланс в обучающей выборке, упростив тем самым проведение различий между метками классов дождливых и сухих дней.

Код листинга 22.37 обучает модель на признаках осеннего сезона и облачности. Здесь мы также устанавливаем разделяющий столбец на осень, поскольку этот признак имеет минимальный коэффициент Джини.

Листинг 22.37. Обучение модели на двух наилучших признаках

```
skip_index = indices[-1]
X_subset = np.delete(X_rain, skip_index, axis=1)
name_subset = np.delete(feature_names, skip_index)
split_col = indices[0] if indices[0] < skip_index else indices[0] - 1
model, accuracy = train_nested_if_else(X_subset, y_rain,
                                       split_col=split_col,
                                       feature_names=name_subset)
print(model)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")

prediction = 0

This statement is 74% accurate.
```

Игнорирует итоговый признак, имеющий худший коэффициент Джини

Подмножество из двух лучших признаков

Корректирует столбец признака с лучшим разделением относительно индекса отброшенного признака с худшим коэффициентом Джини

Обучает вложенную модель на двух лучших признаках

Наша обученная модель чересчур проста, так как прогнозирует отсутствие дождя в любом случае. Ее точность составляет лишь 74 % — этот показатель не ужасен, но можно определенно добиться большего. Игнорирование признака дождя ограничило

нашу способность к точному прогнозированию. Значит, для подъема показателя точности необходимо задействовать все три признака. Это можно сделать так.

1. Выполнить разделение по признаку с минимальным коэффициентом Джини. Им, естественно, является осень.
2. Обучить две вложенные модели, используя функцию `train_nested_if_else`. Модель A будет анализировать только те сценарии, в которых временем года выступает не осень. При этом модель B будет обрабатывать все оставшиеся случаи, когда на дворе стоит осенняя пора.
3. Объединить модель A и модель B в один согласованный классификатор.

ПРИМЕЧАНИЕ

Эти этапы практически идентичны логике функции nested_if_else. Основное отличие в том, что теперь мы расширяем эту логику на третий признак.

Начнем с разделения данных по признаку осени, чей индекс сохранен в `indices[0]` (листинг 22.38).

Листинг 22.38. Разделение по признаку с минимальным коэффициентом Джини

```
X_a, X_b, y_a, y_b = split(X_rain, y_rain, feature_col=indices[0])
```

Далее обучим вложенную модель на `(X_a, y_a)` (листинг 22.39). Эта обучающая выборка содержит все наблюдения, сделанные вне осеннего периода.

Листинг 22.39. Обучение модели на неосенних данных

```
name_subset = np.delete(feature_names, indices[0])
split_col = sort_feature_indices(X_a, y_a)[0]
model_a, accuracy_a = train_nested_if_else(X_a, y_a,
                                           split_col=split_col,
                                           feature_names=name_subset)
print("If it is not autumn, then the following nested model is "
      f"{100 * accuracy_a:.0f}% accurate.\n\n{model_a}")

If it is not autumn, then the following nested model is 88% accurate.

if is_cloudy == 0:
    prediction = 0
else:
    if rained_today == 0:
        prediction = 0
    else:
        prediction = 1
```

Выполняет разделение по признаку в X_a, дающему лучший (минимальный) коэффициент Джини

Обучает вложенную модель на двух признаках из (X_a, y_a)

Модель `model_a` получилась очень точной. Теперь обучим модель `model_b` на основе осенних наблюдений, хранящихся в `(X_b, y_b)` (листинг 22.40).

Листинг 22.40. Обучение модели на осенних данных

```
split_col = sort_feature_indices(X_b, y_b)[0]
model_b, accuracy_b = train_nested_if_else(X_b, y_b,
                                           split_col=split_col,
                                           feature_names=name_subset)
print("If it is autumn, then the following nested model is "
      f"{100 * accuracy_b:.0f}% accurate.\n\n{model_b}")
```

Выполняет разделение по признаку в X_b, дающему лучший (минимальный) коэффициент Джини

Обучает вложенную модель на признаках из (X_b, y_b)

```
If it is autumn, then the following nested model is 79% accurate.

if is_cloudy == 0:
    prediction = 0
else:
    if rained_today == 0:
        prediction = 0
    else:
        prediction = 1
```

Когда на дворе осень, `model_b` справляется с точностью 79 %. В других случаях `model_a` дает точность 88 %. Теперь объединим эти модели в одно вложенное выражение (листинг 22.41). Здесь нам поможет функция `combine_if_else`, которую мы создали ранее как раз для этой цели. Также вычислим общую точность, равную взвешенному среднему `accuracy_a` и `accuracy_b`.

Листинг 22.41. Совмещение моделей в единое вложенное выражение

```
nested_model = combine_if_else(model_a, model_b,
                               feature_names[indices[0]])
print(nested_model)
accuracies = [accuracy_a, accuracy_b]
accuracy = np.average(accuracies, weights=[y_a.size, y_b.size])
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")

if is_fall == 0:
    if is_cloudy == 0:
        prediction = 0
    else:
        if rained_today == 0:
            prediction = 0
        else:
            prediction = 1
else:
    if is_cloudy == 0:
        prediction = 0
    else:
        if rained_today == 0:
            prediction = 0
        else:
            prediction = 1

This statement is 85% accurate.
```

Мы смогли сгенерировать вложенную модель на основе трех признаков. Этот процесс очень похож на обучение вложенной модели с использованием двух признаков. Аналогичным образом можно расширить логику на обучение модели с помощью четырех, десяти и даже ста признаков. В действительности нашу логику можно обобщить на обучение любой вложенной модели с помощью N признаков. Предположим, что нам предоставлена обучающая выборка `(X, y)`, в которой `X` содержит N столбцов. У нас должна быть возможность обучить модель, выполнив следующие шаги.

1. Если $N = 1$, вернуть простой невложенный результат `train_if_else(X, y)`. В противном случае перейти к следующему шагу.
2. Упорядочить N признаков на основе коэффициента Джини в порядке возрастания.
3. Обучить упрощенную модель с помощью $N - 1$ признаков (используя старшие признаки из шага 2). Если эта модель будет работать со 100%-ной точностью, вернуть ее в качестве результата. В противном случае перейти к следующему шагу.
4. Выполнить разделение по признаку с минимальным коэффициентом Джини. Эта операция вернет две обучающие выборки, `(X_a, y_a)` и `(X_b, y_b)`, каждая из которых будет содержать $N - 1$ признаков.
5. Обучить две модели на $N - 1$ признаков — `model_a` и `model_b`, — применяя обучающие выборки из предыдущего шага. Соответствующие показатели точности будут представлены как `accuracy_a` и `accuracy_b`.
6. Объединить `model_a` и `model_b` во вложенную условную модель `if/else`.
7. Вычислить точность вложенной модели, используя взвешенное среднее показателей `accuracy_a` и `accuracy_b`. Соответствующие веса будут равны `y_a.size` и `y_b.size`.
8. Если полученная модель превосходит по точности более простую модель из шага 3, вернуть ее. В противном случае вернуть простую.

В коде листинга 22.42 определяется рекурсивная функция `train`, выполняющая все описанные шаги.

Листинг 22.42. Обучение вложенной модели с помощью N признаков

```
def train(X, y, feature_names):
    if X.shape[1] == 1:
        return train_if_else(X, y, feature_name=feature_names[0])

    indices = sort_feature_indices(X, y)
    X_subset = np.delete(X, indices[-1], axis=1)
```

Обучает вложенное выражение if/else на выборке (X, y), включающей N признаков, и возвращает это выражение вместе с полученной точностью. Названия признаков хранятся в массиве feature_names

Упорядочивает индексы признаков по коэффициенту Джини

```
    name_subset = np.delete(feature_names, indices[-1])
    simple_model, simple_accuracy = train(X_subset, y, name_subset)
    if simple_accuracy == 1.0:
        return (simple_model, simple_accuracy)

    split_col = indices[0]
    name_subset = np.delete(feature_names, split_col)
    X_a, X_b, y_a, y_b = split(X, y, feature_col=split_col)
    model_a, accuracy_a = train(X_a, y_a, name_subset)
    model_b, accuracy_b = train(X_b, y_b, name_subset)
    accuracies = [accuracy_a, accuracy_b]
    total_accuracy = np.average(accuracies, weights=[y_a.size, y_b.size])
    nested_model = combine_if_else(model_a, model_b, feature_names[split_col])
    if total_accuracy > simple_accuracy:
        return (nested_model, total_accuracy)
    return (simple_model, simple_accuracy)
```

Обучает упрощенную модель с использованием N – 1 признаков, чтобы оценить, обеспечит ли она 100%-ную точность

Выполняет разделение по признаку с минимальным коэффициентом Джини

Обучает с помощью N – 1 признаков две упрощенные модели на двух выборках, возвращенных после разделения

Совмещает упрощенные модели

```
model, accuracy = train(X_rain, y_rain, feature_names)
print(model)
print(f"\nThis statement is {100 * accuracy:.0f}% accurate.")

if is_fall == 0:
    if is_cloudy == 0:
        prediction = 0
    else:
        if rained_today == 0:
            prediction = 0
        else:
            prediction = 1
else:
    if is_cloudy == 0:
        prediction = 0
    else:
        if rained_today == 0:
            prediction = 0
        else:
            prediction = 1

This statement is 85% accurate.
```

Ветвление инструкций `if/else` в нашей обученной модели напоминает ветви дерева. Эту схожесть можно сделать более выраженной, визуализировав результат в виде *дерева решений*. Деревья решений — это особые сетевые структуры, с помощью которых отображают решения `if/else`. В такой сети признаки выступают узлами, а условия — ребрами. Ветви `if` отходят от узлов вправо, а ветви `else` — влево. На рис. 22.6 представлена модель прогнозирования дождя в виде дерева решений.

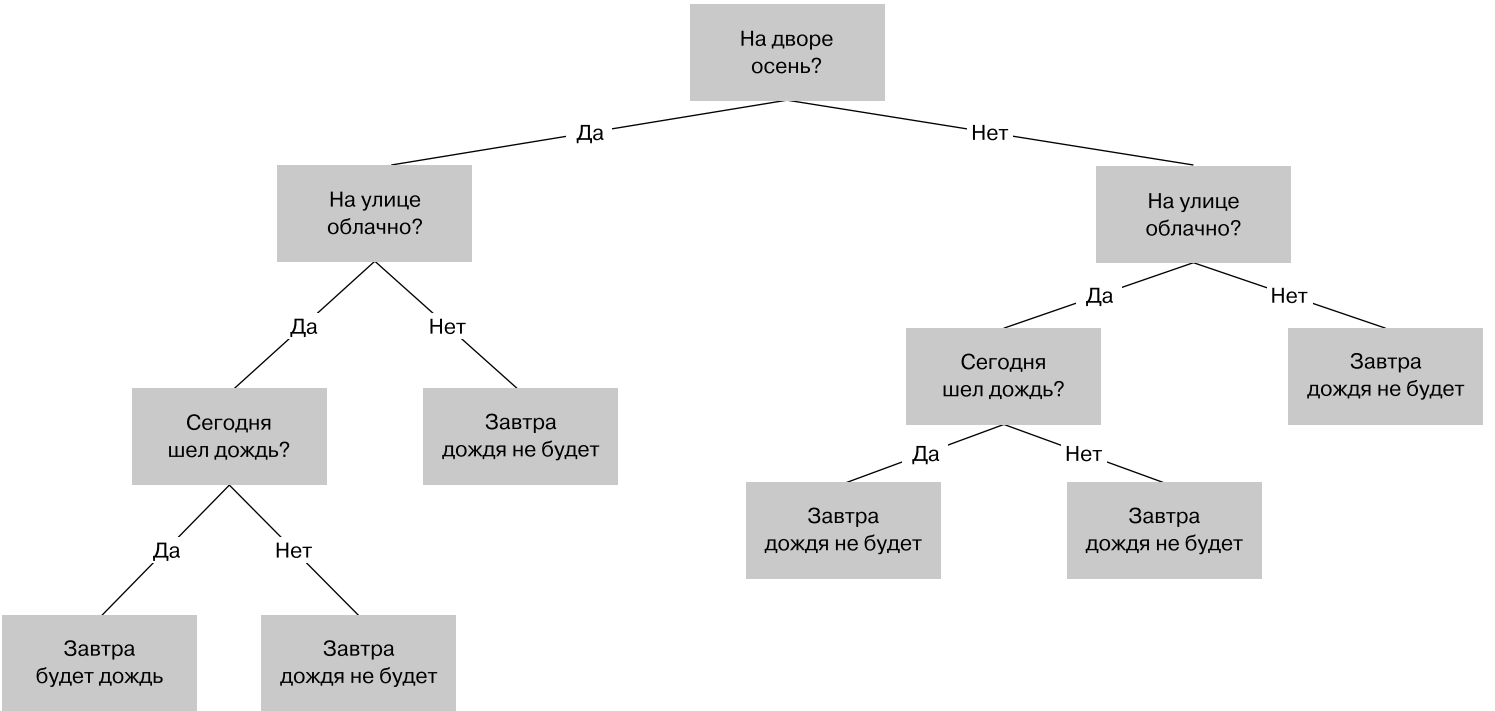


Рис. 22.6. Визуализация прогнозирующей дождь модели в виде дерева решений. Узлы этой сети представляют признаки модели, такие как «На дворе осень?». Ребра же представляют условные инструкции if/else. К примеру, если сейчас осень, то ребро дерева «Да» ответвляется влево. В противном случае ребро «Нет» ответвляется вправо

Любую вложенную инструкцию `if/else` можно визуализировать в виде подобного дерева, в связи с чем обученные условные классификаторы `if/else` так и называют — *деревья решений*. Обученные деревья решений повсеместно используются еще с 1980-х годов. Для их эффективного обучения существует множество стратегий, которые имеют ряд общих черт.

- Задача по обучению с использованием N признаков упрощается до нескольких подзадач с применением $N - 1$ признаков через разделение данных по одному из признаков.
- Выполняется это разделение выбором признака, обеспечивающего максимальный дисбаланс классов. Как правило, это реализуется с помощью коэффициента Джини, хотя существуют и альтернативные показатели.
- Следует избегать использования бесконечно сложных инструкций `if/else`, если более простые работают не хуже. Этот процесс называется *отсечением*, поскольку излишние ветки `if/else` отсекаются.

В scikit-learn есть хорошо оптимизированная реализация деревьев решений, с которой мы познакомимся в следующем разделе.

22.2. ОБУЧЕНИЕ ДЕРЕВЬЕВ РЕШЕНИЙ С ПОМОЩЬЮ SCIKIT-LEARN

В scikit-learn классификация посредством деревьев решений выполняется классом `DecisionTreeClassifier`. Импортируем его из `sklearn.tree` (листинг 22.43).

Листинг 22.43. Импортирование класса DecisionTreeClassifier

```
from sklearn.tree import DecisionTreeClassifier
```

Далее мы инициализируем этот класс как `clf`, после чего обучим `clf` на системе двух лестничных выключателей, которую разбирали в начале главы (листинг 22.44). Ее обучающая выборка хранится в параметрах `(X, y)`.

Листинг 22.44. Инициализация и обучение дерева решений

```
clf = DecisionTreeClassifier()
clf.fit(X, y)
```

Обученный классификатор можно визуализировать с помощью дерева решений. В scikit-learn есть функция `plot_tree`, которая выполняет эту визуализацию с помощью Matplotlib. Вызов `plot_tree(clf)` приведет к построению обученного дерева решений. Названия признаков и классов на этом графике можно изменять с помощью параметров `feature_names` и `class_names` (листинг 22.45).

Листинг 22.45. Отображение обученного дерева решений

```
from sklearn.tree import plot_tree
feature_names = ['Switch0', 'Switch1']
class_names = ['Off', 'On']
plot_tree(clf, feature_names=feature_names, class_names=class_names)
plt.show()
```

Далее мы импортируем `plot_tree` из `sklearn.tree` и визуализируем `clf` (рис. 22.7). На этом графике названия признаков представлены как `Switch0` и `Switch1`, а метки классов — как два состояния лампочки, `Off` и `On`.

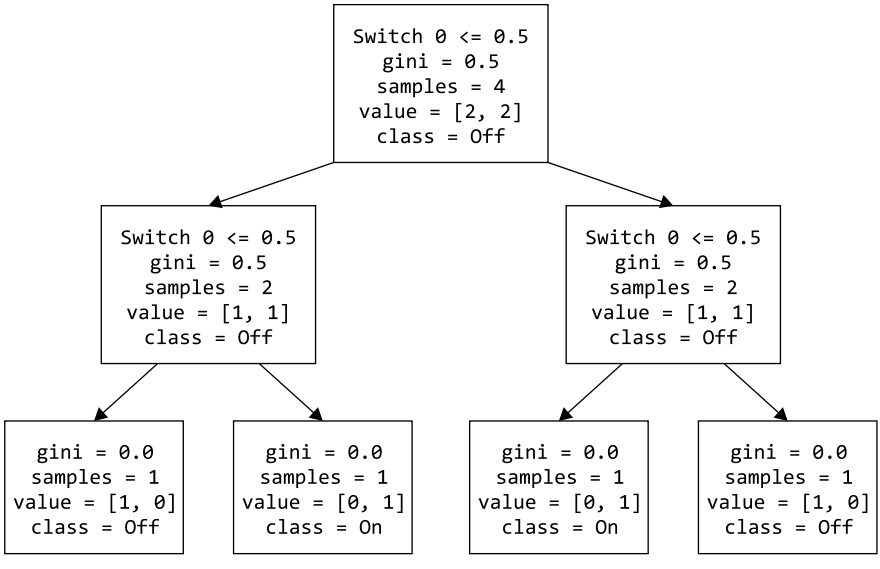


Рис. 22.7. Дерево решений системы с двумя выключателями. Каждый верхний узел содержит название признака вместе с дополнительной статистикой, такой как коэффициент Джини и доминирующий класс. Нижние узлы содержат итоговые спрогнозированные классы состояния лампочки

Визуализированная диаграмма отражает распределение классов в каждой условной позиции дерева. Помимо этого, в ней отражается соответствующий коэффициент Джини, а также доминирующий класс. Подобные визуализации имеют определенную пользу, но при большом числе признаков эти деревья могут становиться излишне громоздкими. По этой причине в scikit-learn реализована альтернативная функция визуализации — `export_text`, позволяющая отображать дерево с помощью упрощенной текстовой диаграммы. Вызов `export_text(clf)` приведет к возвращению строки, вывод которой представит дерево, состоящее из символов | и -. Названия признаков в нем можно устанавливать с помощью

параметра `feature_names`. Однако ввиду ограниченности вывода названия классов здесь отобразить не получится. Далее мы импортируем `export_text` из `sklearn.tree` и визуализируем наше дерево в виде простой строки (листинг 22.46).

Листинг 22.46. Отображение дерева решений в виде строки

```
from sklearn.tree import export_text
text_tree = export_text(clf, feature_names=feature_names)
print(text_tree)
```

```
|--- Switch0 <= 0.50
|    |--- Switch1 <= 0.50
|    |    |--- class: 0
|    |--- Switch1 > 0.50
|    |    |--- class: 1
|--- Switch0 > 0.50
|    |--- Switch1 <= 0.50
|    |    |--- class: 1
|    |--- Switch1 > 0.50
|    |    |--- class: 0
```

В этом тексте ясно просматривается логика ветвления. Изначально данные разделяются на основе `Switch0`. Выбор ветви зависит от того, выполняется ли условие `Switch0 <= 0.50`. Естественно, поскольку `Switch0` может быть равен 0 либо 1, эта логика равнозначна выражению `Switch0 == 0`. Почему в дереве используется неравенство, когда должно быть достаточно простого выражения `Switch0 == 0`? Ответ связан с тем, как `DecisionTreeClassifier` обрабатывает непрерывные признаки. До этого момента все наши признаки были логическими. Однако в большинстве реальных задач они численные. К счастью, любой численный признак можно трансформировать в логический. Для этого достаточно выполнить `feature >= thresh`, где `thresh` представляет некий численный порог. В scikit-learn деревья решений обнаруживают этот порог автоматически.

А как выбрать оптимальный порог для разделения численного признака? Легко. Нужно лишь выбрать такой, который минимизирует коэффициент Джини. Предположим, что мы проверяем набор данных, в основе которого лежит один численный признак. В этих данных класс всегда равен 0, когда признак менее 0,7, и 1 — в противном случае. Таким образом, `y = (v>= 0.7).astype(int)`, где `v` — вектор признаков. Применяя порог 0,7, мы можем идеально разделить наши метки классов. Это приведет к коэффициенту Джини 0, значит, можно выделить этот порог, вычислив коэффициент для всех его возможных значений. Затем останется выбрать такое значение, при котором коэффициент оказывается минимальным. Код листинга 22.47 отбирает вектор `feature` из нормального распределения, устанавливает `y` равным `(feature >= 0.7).astype(int)`, вычисляет коэффициент Джини для ряда значений порога и графически отображает результат (рис. 22.8). Минимальный коэффициент наблюдается при пороге 0,7.

Листинг 22.47. Выбор порога на основе минимального коэффициента Джини

```
np.random.seed(1)
feature = np.random.normal(size=1000)
y = (feature >= 0.7).astype(int)
thresholds = np.arange(0.0, 1, 0.001)
gini_impurities = []
for thresh in thresholds:
    y_a = y[feature <= thresh]
    y_b = y[feature >= thresh]
    impurity = compute_impurity(y_a, y_b)
    gini_impurities.append(impurity)

best_thresh = thresholds[np.argmin(gini_impurities)]
print(f"impurity is minimized at a threshold of {best_thresh:.02f}")
plt.plot(thresholds, gini_impurities)
plt.axvline(best_thresh, c='k', linestyle='--')
plt.xlabel('Threshold')
plt.ylabel('impurity')
plt.show()
```

Случайно отбирает численный признак из нормального распределения

Метки классов соответствуют 0, когда признак оказывается ниже порога 0,7, и 1 — в противном случае

Перебирает пороги в диапазоне от 0 до 1,0

При каждом пороге мы выполняем разделение и вычисляем итоговый коэффициент Джини

Выбирает порог, при котором коэффициент Джини минимизируется. Он должен равняться 0,7

```
Impurity is minimized at a threshold of 0.70
```

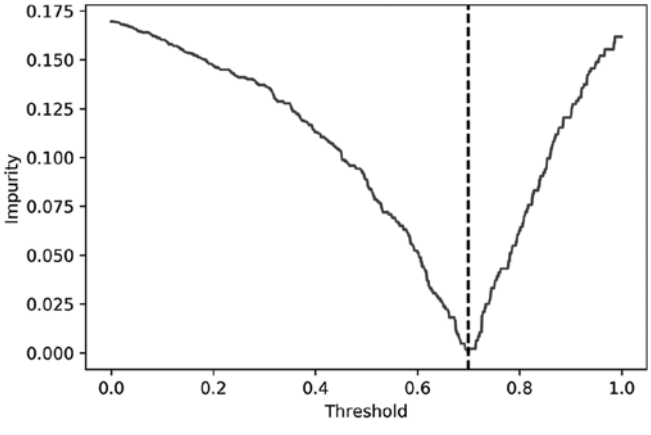


Рис. 22.8. График коэффициентов Джини для каждого возможного порога признака. Этот коэффициент минимизируется при значении порога 0,7, значит, численный признак f можно преобразовать в двоичный признак f >= 0.7

Таким образом scikit-learn получает пороги неравенств для всех признаков, используемых в обучении `DecisionTreeClassifier`, чтобы классификатор мог вывести условную логику из численных данных. Теперь мы обучим `clf` на численных данных о винах, представленных в предыдущей главе (листинг 22.48), после чего визуализируем соответствующее дерево.

ПРИМЕЧАНИЕ

Напомню, что набор данных для вин содержит три их класса. До сих пор мы обучали деревья решений только на системах из двух классов. Однако нашу логику ветвления if/else можно легко расширить для прогнозирования и большего числа классов. Рассмотрим, к примеру, выражение 0 if x == 0 else 1 if y == 0 else 2. Оно возвращает 0 при x == 0. В иных случаях возвращается 1, если y == 0, и 2, если y != 0. Внедрить эту дополнительную условную логику в классификатор будет совсем не сложно.

Листинг 22.48. Обучение дерева решений на численных данных

```
np.random.seed(0)
from sklearn.datasets import load_wine
X, y = load_wine(return_X_y=True)
clf.fit(X, y)
feature_names = load_wine().feature_names
text_tree = export_text(clf, feature_names=feature_names)
print(text_tree)
```

```
|--- proline <= 755.00
|    |--- od280/od315_of_diluted_wines <= 2.11
|    |    |--- hue <= 0.94
|    |    |    |--- flavanoids <= 1.58
|    |    |    |    |--- class: 2
|    |    |    |--- flavanoids > 1.58
|    |    |    |    |--- class: 1
|    |    |--- hue > 0.94
|    |    |    |--- color_intensity <= 5.82
|    |    |    |    |--- class: 1
|    |    |    |--- color_intensity > 5.82
|    |    |    |    |--- class: 2
|    |--- od280/od315_of_diluted_wines > 2.11
|    |    |--- flavanoids <= 0.80
|    |    |    |--- class: 2
|    |    |--- flavanoids > 0.80
|    |    |    |--- alcohol <= 13.17
|    |    |    |    |--- class: 1
|    |    |    |--- alcohol > 13.17
|    |    |    |    |--- color_intensity <= 4.06
|    |    |    |    |    |--- class: 1
|    |    |    |    |--- color_intensity > 4.06
|    |    |    |    |    |--- class: 0
|--- proline > 755.00
|    |--- flavanoids <= 2.17
|    |    |--- malic_acid <= 2.08
|    |    |    |--- class: 1
|    |    |--- malic_acid > 2.08
|    |    |    |--- class: 2
|    |--- flavanoids > 2.17
|    |    |--- magnesium <= 135.50
|    |    |    |--- class: 0
|    |    |--- magnesium > 135.50
|    |    |    |--- class: 1
```

Полученное дерево больше, чем любое другое дерево или условное выражение, которые мы до этого видели. Столь значительный размер объясняется его глубиной. В машинном обучении глубина равняется количеству вложенных инструкций `if/else`, необходимых для описания логики в рамках дерева. К примеру, в случае с одним выключателем требовалась одна инструкция `if/else`, следовательно, глубина дерева составляла 1. Система с двумя выключателями имела уже глубину 2. По той же логике глубина системы прогноза погоды равна 3. Предиктор вин оказывается еще глубже, что несколько усложняет следование за его логикой. В scikit-learn можно ограничить глубину обученного дерева с помощью гиперпараметра `max_depth`. Например, выполнение `DecisionTreeClassifier(max_depth=2)` приведет к созданию классификатора, глубина которого не может превысить две вложенные инструкции. Продемонстрирую использование этой возможности, обучив классификатор на данных о винах (листинг 22.49).

Листинг 22.49. Обучение дерева ограниченной глубины

```
clf = DecisionTreeClassifier(max_depth=2)
clf.fit(X, y)
text_tree = tree.export_text(clf, feature_names=feature_names)
print(text_tree)
```

```
|--- proline <= 755.00
|    |--- od280/od315_of_diluted_wines <= 2.11
|    |    |--- class: 2
|    |--- od280/od315_of_diluted_wines > 2.11
|    |    |--- class: 1
|--- proline > 755.00
|    |--- flavanoids <= 2.17
|    |    |--- class: 2
|    |--- flavanoids > 2.17
|    |    |--- class: 0
```

Глубина полученного дерева равна двум инструкциям `if/else`. Внешняя инструкция определяется концентрацией пролина — если она выше 755, значит, для распознания вина используются флавоноиды.

ПРИМЕЧАНИЕ

В наборе данных flavonoids ошибочно указаны как flavanoids.

В иных случаях для определения класса применяется признак `OD280 / OD315` разбавленных вин. На основе полученного вывода мы можем полноценно разобраться в рабочей логике модели. Более того, можем сопоставить важность признаков, определяющих прогнозирование классов.

- Пролин — самый важный признак. Он представлен на макушке дерева, а значит, имеет минимальный коэффициент Джини. Разделение по этому признаку должно вести к получению максимально несбалансированных данных. В несбалансированном наборе данных гораздо проще отделить один класс от

другого, выходит, знание концентрации пролина упростит различение классов вина. Этот довод согласуется с линейной моделью, обученной в главе 21, где коэффициенты пролина давали самый заметный сигнал.

- Флавоноиды и `OD280 / OD315` также являются важными определяющими прогноза (хотя и в меньшей степени, чем пролин).
- Остальные десять признаков менее релевантны.

Глубина, на которой признак находится в дереве, показывает его относительную важность. Эта глубина определяется величиной коэффициента Джини. Таким образом, данный коэффициент можно использовать для вычисления показателя важности. Показатели важности для всех признаков сохраняются в атрибуте `feature_importances_` классификатора `clf`. Код листинга 22.50 выводит `clf.feature_importances_`.

ПРИМЕЧАНИЕ

Говоря точнее, scikit-learn вычисляет важность признаков вычитанием коэффициента Джини разделения по текущему признаку из коэффициента Джини предыдущего разделения. Например, в дереве вин коэффициент Джини флавоноидов на глубине 2 вычитается из коэффициента Джини пролина на уровне 1. После этого важность взвешивается на основании доли обучающих образцов, представленных при разделении.

Листинг 22.50. Вывод важности признаков

```
print(clf.feature_importances_)
```

```
[0.         0.         0.         0.         0.         0.
 0.117799   0.         0.         0.         0.         0.39637021
 0.48583079]
```

В этом массиве важность признака `i` равна `feature_importances_[i]`. Большинство признаков получают балл 0, поскольку в обученном дереве отсутствуют. Давайте ранжируем оставшиеся признаки на основе их показателя важности (листинг 22.51).

Листинг 22.51. Ранжирование релевантных признаков по важности

```
for i in np.argsort(clf.feature_importances_)[::-1]:
    feature = feature_names[i]
    importance = clf.feature_importances_[i]
    if importance == 0:
        break

    print(f"'{feature}' has an importance score of {importance:0.2f}")
```

```
'proline' has an importance score of 0.49
'od280/od315_of_diluted_wines' has an importance score of 0.40
'flavanoids' has an importance score of 0.12
```

Среди всех признаков пролин ранжируется как самый важный. После него идут `OD280 / OD315` и флавоноиды.

Ранжирование признаков на основе дерева помогает делать важные выводы из имеющихся данных. Далее мы подчеркнем эту особенность, разобрав серьезную задачу по постановке диагноза «рак».

22.2.1. Изучение раковых клеток на основе важности признаков

Обнаруженная опухоль может оказаться раковой, и чтобы определить, злокачественная (раковая) она или же доброкачественная (не раковая), ее необходимо изучить под микроскопом. Этот прибор позволяет разглядеть отдельные клетки, каждая из которых обладает множеством измеримых признаков, включая:

- площадь;
- периметр;
- компактность (отношение возведенного в квадрат периметра к площади);
- радиус (клетка не идеально круглая, поэтому ее радиус вычисляется как среднее расстояние от центра до внешнего края);
- гладкость (вариация расстояния от середины клетки до ее краев);
- углубления (количество вогнутостей по периметру);
- вогнутость (средний внутренний угол углублений);
- симметрия (совпадение одной стороны клетки с другой);
- текстура (стандартное отклонение оттенков цвета на изображении клетки);
- фрактальная размерность (извилистость внешнего края, основанная на количестве отдельных линейных отрезков, необходимых для замера изогнутой границы).

Представьте себе технологию, позволяющую вычислить эти признаки для каждой отдельной клетки. Однако при биопсии опухоли под микроскопом мы увидим десятки клеток (рис. 22.9), значит, отдельные признаки необходимо объединить. Проще всего это сделать, вычислив их среднее значение и стандартное отклонение. Также можно сохранить самое экстремальное значение некоего признака по всем клеткам, например записать самую большую вогнутость, обнаруженную у клеток. Неформально эту статистику мы будем называть *худшей вогнутостью*.

ПРИМЕЧАНИЕ

Как правило, признаки вычисляются не для самой клетки, а для ее ядра. Оно расположено в центре клетки и представляет собой замкнутую округлую структуру, легко различимую под микроскопом.

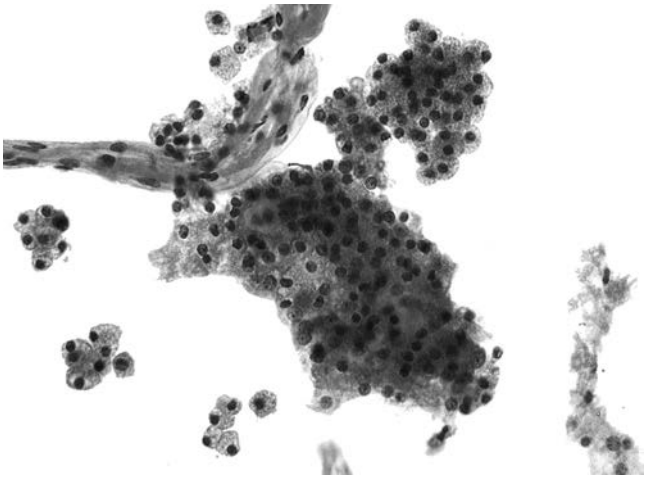


Рис. 22.9. Десятки опухолевых клеток под микроскопом. У каждой из них есть десять измеримых признаков. Их можно объединить для всех клеток с помощью трех разных статистик, получив 30 признаков для определения злокачественности опухоли

Три скомпонованных набора по десять признаков в общей сложности дают 30 признаков. Какие из них наиболее важны для определения злокачественности клетки? Это можно выяснить. В scikit-learn есть набор данных раковых клеток. Мы импортируем его из `sklearn.datasets`, после чего выведем названия признаков и имена классов (листинг 22.52).

Листинг 22.52. Импорт набора данных раковых клеток из scikit-learn

```
from sklearn.datasets import load_breast_cancer

data = load_breast_cancer()
feature_names = data.feature_names
num_features = len(feature_names)
num_classes = len(data.target_names)
print(f"The cancer dataset contains the following {num_classes} classes:")
print(data.target_names)
print(f"\nIt contains these {num_features} features:")
print(feature_names)

The cancer dataset contains the following 2 classes:
['malignant' 'benign']

It contains these 30 features:
['mean radius' 'mean texture' 'mean perimeter' 'mean area' 'mean smoothness'
 'mean compactness' 'mean concavity' 'mean concave points' 'mean symmetry'
 'mean fractal dimension' 'radius error' 'texture error' 'perimeter error'
```

```
 'area error' 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error' 'worst radius'
 'worst texture' 'worst perimeter' 'worst area' 'worst smoothness' 'worst compactness'
 'worst concavity' 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

В этом наборе данных содержатся 30 признаков. Их мы ранжируем по важности и выведем вместе с ее показателями (листинг 22.53). Признаки, показатель которых будет близок к нулю, проигнорируем.

Листинг 22.53. Ранжирование раковых признаков по важности

```
X, y = load_breast_cancer(return_X_y=True)
clf = DecisionTreeClassifier()
clf.fit(X, y)
for i in np.argsort(clf.feature_importances_)[::-1]:
    feature = feature_names[i]
    importance = clf.feature_importances_[i]
    if round(importance, 2) == 0:
        break
    print(f"'{feature}' has an importance score of {importance:0.2f}")
```

```
'worst radius' has an importance score of 0.70
'worst concave points' has an importance score of 0.14
'worst texture' has an importance score of 0.08
'worst smoothness' has an importance score of 0.01
'worst concavity' has an importance score of 0.01
'mean texture' has an importance score of 0.01
'worst area' has an importance score of 0.01
'mean concave points' has an importance score of 0.01
'worst fractal dimension' has an importance score of 0.01
'radius error' has an importance score of 0.01
'smoothness error' has an importance score of 0.01
'worst compactness' has an importance score of 0.01
```

Тремя лидирующими признаками оказались *худший радиус (worst radius)*, *худшие углубления (worst concave points)* и *худшая текстура (worst texture)*. Злокачественность опухоли не определяется ни средним значением, ни стандартным отклонением. Вместо этого при постановке диагноза рака определяющим фактором выступает наличие нескольких экстремальных выбросов. Даже одна или две клетки нестандартной формы могут указывать на наличие заболевания. Среди ведущих признаков особенно выделяется худший радиус, демонстрирующий показатель важности `0.7`. Следующим по старшинству показателем является `0.14`. Эта разница предполагает, что радиус крупнейшей клетки выступает крайне важным индикатором рака. Для проверки этого предположения построим гистограммы измерений худшего радиуса по двум классам (листинг 22.54; рис. 22.10).

Листинг 22.54. Построение двух гистограмм показателей худшего радиуса

```
index = clf.feature_importances_.argmax()
plt.hist(X[y == 0][:, index], label='Malignant', bins='auto')
```

```
plt.hist(X[y == 1][:, index], label='Benign', color='y', bins='auto',
         alpha=0.5)

plt.xlabel('Worst Radius')
plt.legend()
plt.show()
```

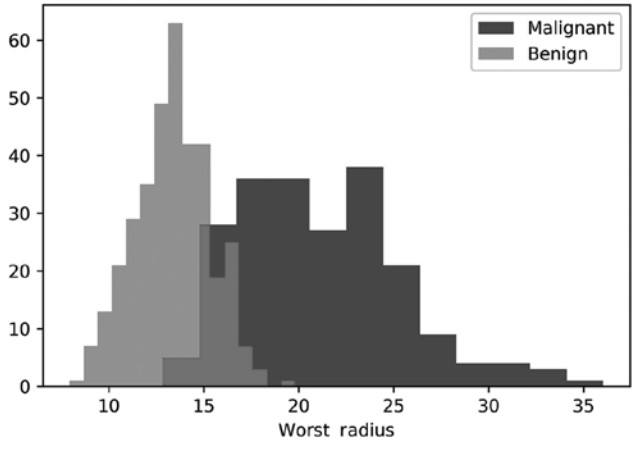


Рис. 22.10. Гистограмма измерений худшего радиуса клетки среди раковых и нераковых опухолей. У злокачественных образований этот радиус значительно больше

Эта гистограмма демонстрирует огромную разницу между измерениями худшего радиуса клетки у злокачественных и доброкачественных образований. Радиус клетки более 20 единиц — достоверный признак ее злокачественности.

Путем обучения дерева решений мы получили полезные данные из области медицины и биологии. Обычно деревья решений оказываются очень эффективными инструментами для выявления сигналов в сложных наборах данных. Эти деревья очень легко трактовать. Их обученные логические инструкции несложно проанализировать с помощью методов науки о данных. Более того, деревья решений обеспечивают дополнительные преимущества.

- Такие классификаторы очень быстро обучаются и оказываются на порядок быстрее классификаторов KNN. Кроме того, в отличие от линейных классификаторов, они не зависят от повторяющихся итераций обучения.
- Деревья решений не зависят от манипуляций, предшествующих обучению. Логистическая регрессия требует предварительной стандартизации данных, которая деревьям решений не нужна. К тому же линейные классификаторы не способны обрабатывать категориальные признаки без предварительной трансформации, а деревья могут делать это напрямую.

- Деревья решений не ограничиваются геометрической формой обучающих данных. В противоположность этому, как показано на рис. 22.1, KNN и линейные классификаторы не могут обрабатывать определенные геометрические конфигурации.

Однако за все эти преимущества приходится платить — обученные деревья решений иногда плохо работают с реальными данными.

22.3. ОГРАНИЧЕНИЯ ДЕРЕВЬЕВ РЕШЕНИЙ

Деревья решений хорошо усваивают обучающие данные — иногда даже слишком хорошо. Определенные деревья просто запоминают данные, не выводя полезных закономерностей. Подобное зазубривание данных — это серьезный недостаток. Представьте студента, готовящегося к выпускному экзамену по физике. Материалы экзамена прошлого года доступны онлайн и содержат ответы на все вопросы. Студент заучивает все данные прошлогоднего экзамена и теперь, встречая вопросы из него, может легко повторять ответы на них. Он уверен в себе, но в день экзамена случается провал! Вопросы на нем оказываются немного другими. В прошлом году на экзамене спрашивалось об ускорении теннисного мяча, брошенного с высоты 20 футов, а сейчас аналогичный вопрос поставлен уже для бильярдного шара, падающего с высоты 50 футов. Студент в растерянности. Он заучил ответы, но не закономерности, их обусловливающие, в результате чего не может справиться с экзаменом, так как его знания не обобщаются.

Чрезмерное заучивание уменьшает полезность итоговых моделей. В машинном обучении с учителем это явление называется *переобучением*. Переобученная модель максимально соответствует обучающим данным, в результате чего может давать неточные прогнозы относительно новых наблюдений. Деревья решений особенно уязвимы к переобучению, так как способны заучить обучающие данные. К примеру, наш детектор рака `clf` идеально запомнил обучающую выборку `(X, y)`. Убедиться в этом можно, выводя точность, с которой `clf.predict(X)` соответствует `y` (листинг 22.55).

Листинг 22.55. Проверка точности модели обнаружения рака

```
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(clf.predict(X), y)
print("Our classifier has memorized the training data with "
      f"{100 * accuracy:.0f}% accuracy.")

Our classifier has memorized the training data with 100% accuracy.
```

Этот классификатор может опознать любой обучающий образец со 100%-ной точностью, но это не значит, что он так же хорошо способен работать с реальными данными. Истинную точность классификатора можно более эффективно

определить с помощью кросс-валидации. Код листинга 22.56 разделяет (`X, y`) на обучающую выборку (`X_train, y_train`) и валидационную (`X_test, y_test`). Мы обучаем `clf` так, чтобы он в совершенстве запомнил (`X_train, y_train`), после чего проверяем, насколько хорошо он может работать с данными, с которыми ранее не встречался. Для этого вычисляем точность модели относительно валидационной выборки.

Листинг 22.56. Проверка точности модели с помощью кросс-валидации

```
np.random.seed(0)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, )
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)
accuracy = accuracy_score(clf.predict(X_test), y_test)
print(f"The classifier performs with {100 * accuracy:.0f}% accuracy on "
       "the validation set.")
```

```
The classifier performs with 90% accuracy on the validation set.
```

Истинная точность классификатора составила 90 %. Это хороший показатель, но можно добиться и лучшего. Нам нужен способ повысить эффективность за счет уменьшения переобучения в дереве. Это можно сделать, одновременно обучая несколько деревьев решений с помощью техники, называемой *случайным лесом*.

РЕЛЕВАНТНЫЕ МЕТОДЫ ДЕРЕВЬЕВ РЕШЕНИЙ В SCIKIT-LEARN

- `clf = DecisionTreeClassifier()` — инициализирует дерево решений.
- `clf = DecisionTreeClassifier(max_depth=x)` — инициализирует дерево решений с максимальной глубиной `x`.
- `clf.feature_importances_` — получает показатели важности признаков обученного дерева решений.
- `plot_tree(clf)` — строит график дерева решений `clf`.
- `plot_tree(clf, feature_names=x, class_names=y)` — строит график дерева решений с установленными названиями признаков и метками классов.
- `export_text(clf)` — представляет график дерева решений в виде простой строки.
- `export_text(clf, feature_names=x)` — представляет график дерева решений в виде простой строки с установленными названиями признаков.

22.4. ПОВЫШЕНИЕ ЭФФЕКТИВНОСТИ С ПОМОЩЬЮ СЛУЧАЙНЫХ ЛЕСОВ

Иногда совокупная точка зрения массы людей оказывается точнее отдельных прогнозов. В 1906 году на сельской ярмарке в Плимуте собралась толпа, чтобы угадать вес быка. Реальная масса животного составляла 1198 фунтов. Каждый присутствующий написал свою цифру, и на основе всех записей был выведен средний показатель, который оказался равен 1207 фунтам, отклонившись от действительной массы в пределах 1 %. Этот триумф коллективного разума называли *мудростью толпы*.

Современные демократические институты построены на принципе мудрости толпы. В демократических государствах люди собираются и голосуют за будущее своей страны. Обычно голосующие имеют сильно различающиеся политические взгляды, мнения и жизненный опыт. Однако каким-то образом их совокупный выбор усредняется к решению, способному принести пользу стране в долгосрочной перспективе. Демократический процесс отчасти зависит от разнообразия мнений граждан. Если все будут иметь одинаковое мнение, то рискуют допустить одни и те же ошибки, а разнообразие убеждений помогает уменьшить их вероятность. Толпа чаще принимает оптимальные решения, даже когда люди, из которых она состоит, мыслят по-разному.

Мудрость толпы — естественное явление. Она наблюдается не только у людей, но и у животных. Летучие мыши, рыбы, птицы и даже мухи могут оптимизировать свое поведение, находясь в окружении других представителей своего вида. Это явление можно наблюдать и в сфере машинного обучения. Толпа деревьев решений иногда способна превзойти по качеству прогнозов одно дерево. Однако, чтобы это произошло, входные данные для каждого дерева должны быть различными.

Далее вы наглядно познакомитесь с принципом мудрости толпы, инициализировав 100 деревьев решений (листинг 22.57). Большая коллекция деревьев, что неудивительно, называется *лесом*, поэтому мы сохраняем деревья в списке `forest`.

Листинг 22.57. Инициализация леса из 100 деревьев

```
forest = [DecisionTreeClassifier() for _ in range(100)]
```

А как правильно обучать деревья в таком лесу? В рамках наивного подхода можно обучить каждое отдельное дерево на обучающей выборке данных о раке (`X_train, y_train`). Однако тогда мы получим 100 деревьев, запомнивших в точности одинаковые данные. В результате такого отсутствия разнообразия прогнозы этих деревьев тоже будут одинаковыми. Без разнообразия мудрость толпы не работает. Как же поступить?

Одним из решений будет рандомизация обучающих данных. В главе 7 мы изучили технику бутстрэппинга с восполнением. В ней содержимое *N*-элементного массива итеративно сэмплируется. При этом элементы отбираются с восполнением, что делает возможным появление их дубликатов. Посредством такой выборки можно сгенерировать новый *N*-элементный набор данных, содержимое которого будет отличаться от исходных данных. Давайте выполним бутстрэппинг наших обучающих данных (листинг 22.58), чтобы случайным образом сгенерировать новую обучающую выборку (`X_train_new, y_train_new`).

Листинг 22.58. Случайный сэмплинг новой обучающей выборки

```
np.random.seed(1)
def bootstrap(X, y):
    num_rows = X.shape[0]
    indices = np.random.choice(range(num_rows), size=num_rows,
                               replace=True)
    X_new, y_new = X[indices], y[indices]
    return X_new, y_new

X_train_new, y_train_new = bootstrap(X_train, y_train)
assert X_train.shape == X_train_new.shape
assert y_train.size == y_train_new.size
assert not np.array_equal(X_train, X_train_new)
assert not np.array_equal(y_train, y_train_new)
```

Применяет бутстрэппинг к обучающей выборке (X, y), генерируя новую обучающую выборку

Отбирает случайные индексы точек данных в (X, y). Отбор выполняется с восполнением, чтобы некоторые индексы могли быть отобраны дважды

Возвращает случайную обучающую выборку на основе отобранных индексов

Новая обучающая выборка имеет тот же размер, что и исходная

Полученная через бутстрэппинг обучающая выборка не идентична исходной

А теперь выполним функцию `bootstrap` 100 раз, чтобы сгенерировать 100 разных обучающих выборок (листинг 22.59).

Листинг 22.59. Случайный сэмплинг 100 новых обучающих выборок

```
np.random.seed(1)
features_train, classes_train = [], []
for _ in range(100):
    X_train_new, y_train_new = bootstrap(X_train, y_train)
    features_train.append(X_train_new)
    classes_train.append(y_train_new)
```

По всем 100 полученным обучающим выборкам данные будут различаться, но при этом все признаки останутся неизменными. Однако можно повысить общее разнообразие путем рандомизации признаков в `features_train`. Как правило, мудрость толпы лучше всего работает, когда отдельные ее участники уделяют внимание расходящимся признакам. К примеру, в демократических выборах главные приоритеты городских жителей могут расходиться с приоритетами сельских. Горожан могут беспокоить жилищный вопрос и преступность, а сельчан — цены на урожай и имущественные налоги. Эти расходящиеся приоритеты способны

привести к консенсусу, который в долгосрочной перспективе окажется выгоден как сельским, так и городским жителям.

Говоря конкретнее, в машинном обучении с учителем разнообразие признаков может препятствовать переобучению. Возьмем для примера наш раковый набор данных. Как мы видели, «худший радиус» — чрезвычайно значимый признак, в связи с чем все обученные модели, в которых он присутствует, будут опираться на него как на основной. Однако в редких случаях опухоль может оказаться злокачественной даже при малом радиусе клеток. И если консервативные модели будут опираться на одни и те же признаки, то они станут распознавать эту опухоль ошибочно. Но если представить, что мы обучаем ряд моделей, исключив признак радиуса из набора, то они будут вынуждены искать альтернативные, более тонкие паттерны злокачественности клеток и окажутся устойчивее к колеблющимся реальным данным. По отдельности каждая модель может справляться не очень хорошо, поскольку ее набор признаков ограничен. Коллективно же они должны демонстрировать более высокую эффективность.

Нашей целью будет обучить деревья в лесу на случайных выборках признаков из `features_train`. Сейчас каждая матрица признаков содержит 30 связанных с раком показателей. Нам нужно уменьшить их число в каждой выборке. Каким же будет подходящий размер набора? Как мы видели, обычно удачным размером выборки оказывается результат вычисления квадратного корня из общего числа признаков. Для 30 это значение будет равно примерно 5, значит, установим в качестве размера нашей выборки 5.

Теперь переберем `features_train` и отфильтруем каждую матрицу признаков, оставив в них по пять случайных столбцов (листинг 22.60). Мы также проследим индексы этих случайно отобранных признаков для дальнейшего использования при валидации.

Листинг 22.60. Случайная выборка обучающих признаков

```
np.random.seed(1)
sample_size = int(X.shape[1] ** 0.5)
assert sample_size == 5
feature_indices = [np.random.choice(range(30), 5, replace=False)
                   for _ in range(100)]

for i, index_subset in enumerate(feature_indices):
    features_train[i] = features_train[i][:, index_subset]

for index in [0, 99]:
    index_subset = feature_indices[index]
    names = feature_names[index_subset]
    print(f"\nRandom features utilized by Tree {index}:")
    print(names)
```

Размер выборки признаков приблизительно равен квадратному корню из их общего числа

При наличии 30 признаков ожидается, что размер их выборки будет равен 5

Мы случайно отбираем 5 из 30 столбцов признаков для каждого дерева. Эта выборка выполняется без восполнения, так как повторяющиеся признаки не дадут новых сигналов во время обучения

Выводит случайно отобранные названия признаков в их первом и последнем подмножествах

```
Random features utilized by Tree 0:
['concave points error' 'worst texture' 'radius error'
 'fractal dimension error' 'smoothness error']

Random features utilized by Tree 99:
['mean smoothness' 'worst radius' 'fractal dimension error'
 'worst concave points' 'mean concavity']
```

Пять случайно отобранных признаков не включают худший радиус

Пять случайно отобранных признаков включают важный показатель — худший радиус

Мы рандомизировали каждую из 100 матриц признаков по строкам (точка данных) и столбцам (признак). Эти обучающие данные для каждого дерева очень разнообразны. Далее обучим каждое *i*-е дерево в `forest` на выборке `(features_train[i], classes_train[i])` (листинг 22.61).

Листинг 22.61. Обучение деревьев в лесу

```
for i, clf_tree in enumerate(forest):
    clf_tree.fit(features_train[i], classes_train[i])
```

Вот мы и обучили каждое дерево в лесу. Теперь проведем среди них голосование. Какая метка класса у точки данных в `X_test[0]`? Это можно проверить, используя принцип мудрости толпы. Здесь мы перебираем каждое обученное `clf_tree` в лесу. В каждой *i*-й итерации проделываем следующее.

1. Используем дерево для прогнозирования метки класса в `X_test[0]`. Напомню, что каждое *i*-е дерево в `forest` зависит от случайной подвыборки признаков. Индексы отобранных признаков хранятся в `feature_indices[i]`. Таким образом, прежде чем делать прогнозы, нам нужно отфильтровать `X_test[0]` по выбранным индексам.
2. Фиксируем прогноз как *голос* дерева относительно индекса `i`.

Когда все деревья проголосовали, мы подсчитываем распределение 100 их голосов и выбираем метку класса, получившую большинство (листинг 22.62).

ПРИМЕЧАНИЕ

Этот процесс очень похож на множественное голосование KNN, которое мы применяли в главе 20.

Листинг 22.62. Использование голосования деревьев для классификации точки данных

```
from collections import Counter
feature_vector = X_test[0]
votes = []
for i, clf_tree in enumerate(forest):
    index_subset = feature_indices[i]
    vector_subset = feature_vector[index_subset]
    prediction = clf_tree.predict([vector_subset])[0]
    votes.append(prediction)

class_to_votes = Counter(votes)
for class_label, votes in class_to_votes.items():
    print(f"We counted {votes} votes for class {class_label}.")
```

Перебирает 100 обученных деревьев

Корректирует столбцы в feature_vector, чтобы они соответствовали пяти индексам случайно отобранных признаков, связанных с каждым деревом

Каждое дерево голосует, возвращая прогноз метки класса

Подсчитывает все голоса

```
top_class = max(class_to_votes.items(), key=lambda x: x[1])[0]
print(f"\nClass {top_class} has received the plurality of the votes.")

We counted 93 votes for class 0.
We counted 7 votes for class 1.

Class 0 has received the plurality of the votes.
```

Девяносто три процента деревьев проголосовали за класс 0. Проверим, право ли это большинство (листинг 22.63).

Листинг 22.63. Проверка истинности спрогнозированной метки класса

```
true_label = y_test[0]
print(f"The true class of the data-point is {true_label}.")

The true class of the data-point is 0.
```

Лес успешно определил точку в `X_test[0]`. Теперь используем голосование для определения всех точек в валидационной выборке `X_test` и оценим точность прогноза с помощью `y_test` (листинг 22.64).

Листинг 22.64. Измерение точности леса деревьев

```
predictions = []
for i, clf_tree in enumerate(forest):
    index_subset = feature_indices[i]
    prediction = clf_tree.predict(X_test[:,index_subset])
    predictions.append(prediction)

predictions = np.array(predictions)
y_pred = [Counter(predictions[:,i]).most_common()[0][0]
          for i in range(y_test.size)]
accuracy = accuracy_score(y_pred, y_test)
print("The forest has predicted the validation outputs with "
      f"{100 * accuracy:.0f}% accuracy")

The forest has predicted the validation outputs with 96% accuracy
```

Наш случайно сгенерированный лес спрогнозировал проверочные результаты с 96%-ной точностью. Он превзошел единичное дерево решений, чья точность не превышала 90 %. Используя мудрость толпы, мы смогли повысить эффективность модели. В ходе этого мы также обучили *случайный лес* — коллекцию деревьев, чьи обучающие входные данные были рандомизированы для увеличения разнообразия. Случайные леса обучаются по следующему алгоритму.

1. Инициализация N деревьев решений. Их количество является гиперпараметром. Как правило, большее число деревьев обеспечивают повышенную точность, однако слишком большое количество увеличивает время классификации.
2. Генерация N случайных обучающих выборок путем сэмплинга с восполнением.

3. Случайный отбор `N ** 0.5` столбцов признаков для каждой из *N* обучающих выборок.
4. Обучение всех деревьев решений на *N* случайных обучающих выборок.

После обучения каждое дерево в лесу голосует за присвоение входным данным той или иной метки. Эти голоса суммируются, и классификатор выводит метку, получившую максимум.

Случайные леса очень гибки и не склонны к переобучению. Естественно, в scikit-learn есть соответствующая реализация.

22.5. ОБУЧЕНИЕ СЛУЧАЙНЫХ ЛЕСОВ С ПОМОЩЬЮ SCIKIT-LEARN

В scikit-learn классификация случайных лесов выполняется с помощью класса `RandomForestClassifier`. Давайте импортируем его из `sklearn.ensemble`, инициализируем и обучим, используя `(X_train, y_train)` (листинг 22.65). Наконец, проверим эффективность полученного классификатора на валидационной выборке `(X_test, y_test)`.

Листинг 22.65. Обучение случайного леса

```
np.random.seed(1)
from sklearn.ensemble import RandomForestClassifier
clf_forest = RandomForestClassifier()
clf_forest.fit(X_train, y_train)
y_pred = clf_forest.predict(X_test)
accuracy = accuracy_score(y_pred, y_test)
print("The forest has predicted the validation outputs with "
      f"{100 * accuracy:.0f}% accuracy")
```

Этот результат несколько выше прежних 96 % из-за случайных флуктуаций, а также дополнительных оптимизаций scikit-learn

```
The forest has predicted the validation outputs with 97% accuracy
```

По умолчанию случайный лес scikit-learn задействует 100 деревьев решений, но их число можно увеличить либо уменьшить с помощью параметра `n_estimators`. Код листинга 22.66 уменьшает число деревьев до десяти, выполняя `RandomForestClassifier(n_estimators=10)`, после чего заново вычисляет точность.

Листинг 22.66. Обучение случайного леса из десяти деревьев

```
np.random.seed(1)
clf_forest = RandomForestClassifier(n_estimators=10)
clf_forest.fit(X_train, y_train)
y_pred = clf_forest.predict(X_test)
accuracy = accuracy_score(y_pred, y_test)
print("The 10-tree forest has predicted the validation outputs with "
      f"{100 * accuracy:.0f}% accuracy")

The 10-tree forest has predicted the validation outputs with 97% accuracy
```

Даже при небольшом числе деревьев общая точность остается очень высокой. Иногда десяти деревьев вполне достаточно для обучения высокоточного классификатора.

Каждому из десяти деревьев в `clf_forest` присваивается случайное подмножество из пяти признаков. Каждый признак в этом подмножестве содержит собственный показатель важности. Scikit-learn может усреднить эти показатели по всем деревьям, после чего к агрегированным значениям можно будет обратиться через вызов `clf_forest.feature_importances_`. Далее с помощью атрибута `feature_importances_` мы выведем три ведущих признака в лесу (листинг 22.67).

Листинг 22.67. Ранжирование признаков случайного леса

```
for i in np.argsort(clf_forest.feature_importances_)[::-1][:3]:
    feature = feature_names[i]
    importance = clf_forest.feature_importances_[i]
    print(f"'{feature}' has an importance score of {importance:0.2f}")

'worst perimeter' has an importance score of 0.20
'worst radius' has an importance score of 0.16
'worst area' has an importance score of 0.16
```

Худший радиус по-прежнему значимый показатель, но теперь он сопоставим с худшей площадью и худшим периметром. В отличие от нашего дерева решений случайный лес не придает слишком большого значения отдельному входному признаку и более гибко обрабатывая изменчивые сигналы новых данных. Благодаря такой гибкости этот вид классификаторов часто выбирают для обучения на наборах данных среднего размера.

ПРИМЕЧАНИЕ

Случайные леса отлично работают с наборами данных, содержащими множество признаков и сотни или даже тысячи точек данных. Однако на наборы данных, содержащие миллион и более образцов, этот алгоритм масштабироваться не может. При обработке очень крупных наборов данных требуются более мощные техники глубокого обучения, но в нашей книге не предполагалось их рассматривать.

РЕЛЕВАНТНЫЕ МЕТОДЫ СЛУЧАЙНЫХ ЛЕСОВ В SCIKIT-LEARN

- `clf = RandomForestClassifier()` — инициализирует случайный лес.
- `clf = RandomForestClassifier(n_estimators=x)` — инициализирует случайный лес с количеством деревьев `x`.
- `clf.feature_importances_` — выводит показатели важности признаков обученного случайного леса.

РЕЗЮМЕ

- Определенные задачи классификации можно обработать с помощью вложенных инструкций `if/else`, но не посредством KNN или логистической регрессии.
- Можно обучить модель `if/else` с одним признаком, максимизируя точность по числу совпадений между каждым состоянием признака и каждой меткой класса.
- Можно обучить вложенную модель `if/else` с двумя признаками, выполняя *двоичное разбиение* по одному из них. Разделение по признаку приведет к возвращению двух обучающих выборок, каждая из которых будет связана с уникальным состоянием отдельного признака. Выборки можно использовать для вычисления двух моделей, включающих по одному признаку, после чего объединить эти модели во вложенной инструкции `if/else`. Точность вложенной инструкции равна взвешенному среднему точностей более простых моделей.
- Выбор признака для двоичного разбиения может повлиять на качество модели. Как правило, это разбиение дает лучшие результаты, если генерирует несбалансированные обучающие данные. Дисбаланс обучающей выборки можно отразить с помощью распределения ее меток классов. Чем более разбалансирован набор данных, тем бóльшую абсолютную величину будет иметь вектор распределения. Таким образом, разбалансированные наборы данных дают более высокое значение `v @ v`, где `v` — вектор распределения. Кроме того, значение `1 - v @ v` называется *коэффициентом Джини*. Уменьшение этого коэффициента минимизирует дисбаланс обучающей выборки, а значит, разделение всегда следует выполнять по признаку, дающему минимальный коэффициент Джини.
- Модель, опирающуюся на два признака, можно расширить для обработки N признаков. Модель, включающая N признаков, обучается путем разделения по признаку с минимальным коэффициентом Джини. После этого обучаются две простые модели, каждая из которых обрабатывает $N - 1$ признаков. Далее эти две упрощенные модели объединяются в более сложную вложенную модель, чья точность равна взвешенному среднему их точностей.
- Инструкция `if/else` в обученных условных моделях напоминает ветвление дерева. Это сходство можно сделать более наглядным, визуализировав вывод модели в виде *графика дерева решений*. Деревья решений — это особые сетевые структуры, используемые для выражения решений `if/else`. Любую вложенную инструкцию `if/else` можно визуализировать в виде дерева решений, в связи с чем обученные классификаторы `if/else` называются *деревьями решений*.
- *Глубина* дерева решений равна количеству вложенных инструкций `if/else`, необходимых для отражения логики этого дерева. За счет ограничения глубины можно получать более понятные графики.

- Глубина, на которой находится признак в дереве, является показателем его относительной важности. Определяется глубина коэффициентом Джини, значит, его можно использовать для вычисления степени важности.
- Чрезмерное запоминание уменьшает полезность обученных моделей. В машинном обучении с учителем это явление называется *переобучением*. Переобученная модель максимально соответствует обучающим данным, в связи с чем может давать плохие прогнозы относительно новых наблюдений. Деревья решений особенно склонны к переобучению, так как могут зазубривать обучающие данные.
- Переобучение можно предотвратить, если параллельно обучать множество деревьев решений. Такая коллекция деревьев называется *лесом*. Коллективная мудрость леса способна превзойти то, что понимает одно отдельное дерево, но для этого в лес необходимо внести разнообразие. Это выполняется генерацией случайных обучающих выборок со случайно выбранными признаками. Затем каждое дерево леса обучается на своей обучающей выборке. После этого все деревья леса голосуют, какую метку присвоить входным данным. Коллективная модель, созданная на основе голосования, называется *случайным лесом*.

23 Решение практического задания 5

В этой главе

- ✓ Очистка данных.
- ✓ Изучение сетей.
- ✓ Инжиниринг признаков.
- ✓ Оптимизация моделей машинного обучения.

FriendHook — это популярное приложение социальной сети, ориентированное на кампусы колледжей и университетов. Студенты могут устанавливать в этой сети дружеские связи. Для этого рекомендательный движок еженедельно рассылает пользователям письма с предложением новых друзей, сформированным на основе уже имеющихся у пользователей связей. Студенты могут игнорировать эти рекомендации либо отправлять рекомендованным людям запросы на дружбу. Нам был предоставлен недельный набор данных, включающих предложенные движком рекомендации и соответствующие реакции студентов. Эти данные хранятся в файле `friendhook/Observations.csv`. Помимо этого, у нас есть два дополнительных файла: `friendhook/Profiles.csv` и `friendhook/Friendships.csv`, в которых содержится информация о профилях пользователей и граф дружеских связей соответственно. Для защиты личной информации пользователей их профили были зашифрованы. Наша цель — создать модель, прогнозирующую

поведение пользователей в ответ на рекомендации друзей. Реализуем это следующим образом.

1. Загрузим три набора данных, содержащих зафиксированные реакции пользователей на рекомендации, их профили и графы дружеских связей.
2. Обучим и оценим модель, прогнозирующую поведение на основе признаков сети и профилей. При желании эту задачу можно разбить на две: обучение модели с применением признаков сети и последующее добавление признаков профилей, сопровождаемое оценкой изменения ее эффективности.
3. Убедимся, хорошо ли обобщается модель на другие учебные заведения.
4. Изучим внутреннее устройство модели, чтобы лучше понять поведение студентов.

ВНИМАНИЕ

Спойлер! Далее приводится решение практического задания 5. Я настоятельно рекомендую попытаться решить его самостоятельно, прежде чем читать решение. Исходные условия практического задания можно найти в самом его начале.

23.1. ИЗУЧЕНИЕ ДАННЫХ

Первым делом мы по отдельности разберем таблицы `Profiles`, `Observations` и `Friendships`, попутно очистив и скорректировав содержащиеся в них данные.

23.1.1. Анализ профилей

Начнем с загрузки таблицы `Profiles` в Pandas и обобщения ее содержимого (листинг 23.1).

Листинг 23.1. Загрузка таблицы Profiles

```
import pandas as pd

def summarize_table(df):
    n_rows, n_columns = df.shape
    summary = df.describe()
    print(f"The table contains {n_rows} rows and {n_columns} columns.")
    print("Table Summary:\n")
    print(summary.to_string())

df_profile = pd.read_csv('friendhook/Profiles.csv')
summarize_table(df_profile)
```

Эту функцию мы используем и для двух других таблиц нашего набора данных

```
The table contains 4039 rows and 6 columns.
Table Summary:
```

	Profile_ID	Sex	Relationship_ Status	Dorm	Major	Year
count	4039	4039	3631	4039	4039	4039
unique	4039	2	3	15	30	4
top	b90a1222d2b2	e807eb960650	ac0b88e46e20	a8e6e404d1b3	141d4cdd5aaf	c1a648750a4b
freq	1	2020	1963	2739	1366	1796

В этой таблице содержится 4039 профилей, относящихся к двум полам. Наиболее распространенный пол указан в 2020 профилях из 4039, что говорит о примерно равном распределении мужчин и женщин. Кроме того, эти профили отражают распределение студентов по 30 специальностям и 15 общежитиям. Подозрительно, что в самом часто упоминаемом общежитии проживают больше 2700 студентов. Это число кажется большим, но поиск в Google показывает, что крупные студенческие комплексы общежитий весьма распространены. К примеру, 17-этажное общежитие Sandburg Residence Hall при Университете Висконсин — Милуоки способно вместить до 2700 учащихся. Эти числа также могут представлять студентов из категории `Off-Campus Housing` (проживание за пределами кампуса). Такое количество можно объяснить многими причинами, но в дальнейшем нам необходимо учитывать определяющие факторы, стоящие за наблюдаемыми числами. Вместо того чтобы слепо манипулировать цифрами, важно помнить, что данные получены на основе реального поведения студентов и их физических ограничений.

В сводке по таблице, а именно в столбце `Relationship status`, имеется одна аномалия. Pandas обнаружила три категории `Relationship Status` только среди 3631 строки таблицы из 4039. Оставшиеся строки пустые, то есть не содержат какого-либо семейного статуса. Подсчитаем количество пустых строк (листинг 23.2).

Листинг 23.2. Подсчет профилей с пустой графой Relationship Status

```
is_null = df_profile.Relationship_Status.isnull()
num_null = df_profile[is_null].shape[0]
print(f"{num_null} profiles are missing the Relationship Status field.")
```

```
408 profiles are missing the Relationship Status field.
```

У 408 профилей значение в поле `Relationship_Status` отсутствует. И это вполне объяснимо — как говорилось в условии задачи, поле `Relationship_Status` необязательное. Получается, что 1/10 часть студентов отказались заполнять это поле. Но мы не можем продолжить анализ, когда в данных есть пустые значения. Нам нужно либо удалить строки с пустыми полями, либо заполнить эти поля неким значением. Удаление строк — неудачное решение, поскольку так мы отбросим потенциально ценную информацию, содержащуюся в других столбцах. Вместо этого

можно трактовать отсутствие статуса как четвертую категорию семейного статуса под названием `unspecified`. Для этого нужно присвоить соответствующим строкам ID категории. А какое значение ID выбрать? Чтобы ответить на этот вопрос, проанализируем все уникальные ID в столбце `Relationship Status` (листинг 23.3).

Листинг 23.3. Проверка уникальных значений в столбце Relationship Status

```
unique_ids = set(df_profile.Relationship_Status.values)
print(unique_ids)

{'9cea719429e9', nan, '188f9a32c360', 'ac0b88e46e20'}
```

Как и ожидалось, значения `Relationship Status` состоят из трех хеш-кодов и пустого `nan`. Хеш-коды — это зашифрованные версии трех возможных категорий статуса: `Single`, `In relationship` и `It's complicated`. Естественно, мы не можем знать, какая категория кроется за каждым из кодов. Наша цель — в конечном итоге использовать эту информацию в обученной модели. Однако scikit-learn не умеет обрабатывать хеш-коды или нулевые значения. Она может обрабатывать только числа, поэтому необходимо перевести категории в численные значения. Простейшим способом будет присвоить каждой категории число между 0 и 4. Выполним присваивание, начав с генерации словаря сопоставлений между каждой категорией и числом (листинг 23.4).

Листинг 23.4. Сопоставление значений поля Relationship Status с числами

```
import numpy as np
category_map = {'9cea719429e9': 0, np.nan: 1, '188f9a32c360': 2,
                'ac0b88e46e20': 3}

{'9cea719429e9': 0, nan: 1, '188f9a32c360': 2, 'ac0b88e46e20': 3}
```

Обычно мы генерировали это сопоставление автоматически, выполнив category_map = {id_: i for i, id_ в enumerate(unique_ids)}, но порядок численных присваиваний может различаться в зависимости от версии Python. В связи с этим устанавливаем эти сопоставления вручную, чтобы у всех читателей получился согласованный вывод

Далее заменим содержимое столбца `Relationship Status` подходящими численными значениями (листинг 23.5).

Листинг 23.5. Обновление столбца Relationship Status

```
nums = [category_map[hash_code]
        for hash_code in df_profile.Relationship_Status.values]
df_profile['Relationship_Status'] = nums
print(df_profile.Relationship_Status)
0       0
1       3
2       3
3       3
4       0
```

```
      ..
4034    3
4035    0
4036    3
4037    3
4038    0
Name: Relationship_Status, Length: 4039, dtype: int64
```

Мы преобразовали `Relationship status` в численную переменную, но оставшиеся пять столбцов таблицы все еще содержат хеш-коды. Нужно ли заменять и эти коды числами? Да, и вот почему.

- Как уже говорилось, scikit-learn не умеет обрабатывать строки или хеш-коды. На входе она может получать только числа.
- Людям читать хеш-коды сложнее, чем числа. Значит, замена многосимвольных кодов более короткими числами упростит анализ данных.

С учетом этого далее мы создадим в каждом столбце сопоставление категорий между хеш-кодами и числами. Эти сопоставления отразим с помощью словаря `col_to_mapping`. Используя сопоставления, мы также заменим все хеш-коды в `df_profile` числами (листинг 23.6).

Листинг 23.6. Замена всех хеш-кодов в Profile численными значениями

```
col_to_mapping = {'Relationship_Status': category_map}

for column in df_profile.columns:
    if column in col_to_mapping:
        continue

    unique_ids = sorted(set(df_profile[column].values))
    category_map = {id_: i for i, id_ in enumerate(unique_ids)}
    col_to_mapping[column] = category_map
    nums = [category_map[hash_code]
            for hash_code in df_profile[column].values]
    df_profile[column] = nums

head = df_profile.head()
print(head.to_string(index=False))
```

Упорядочение ID помогает добиться согласованного вывода у всех читателей независимо от их версии Python. Заметьте, что ID хеш-кодов можно упорядочить, только если среди них нет значений nan. В противном случае сортировка окажется неудачной

```
Profile_ID  Sex  Relationship_Status  Dorm  Major  Year
      2899    0                    0     5     13     2
      1125    0                    3    12      6     1
      3799    0                    3    12     29     2
      3338    0                    3     4     25     0
      2007    1                    0    12      2     0
```

На этом корректировка `df_profile` закончена, и мы переключимся на таблицу экспериментальных наблюдений.

23.1.2. Анализ экспериментальных наблюдений

Начнем с загрузки таблицы `Observations` в Pandas и обобщения ее содержимого (листинг 23.7).

Листинг 23.7. Загрузка таблицы Observations

```
df_obs = pd.read_csv('friendhook/Observations.csv')
summarize_table(df_obs)

The table contains 4039 rows and 5 columns.
Table Summary:
```

	Profile_ID	Selected_ Friend	Selected_Friend_ of_Friend	Friend_Request_ Sent	Friend_Request_ Accepted
count	4039	4039	4039	4039	4039
unique	4039	2219	2327	2	2
top	b90a1222d2b2	89581f99fa1e	6caa597f13cc	True	True
freq	1	77	27	2519	2460

Во всех пяти столбцах присутствует 4039 заполненных строк, то есть пустых значений в данной таблице нет. И это хорошо, но вот имена столбцов читать сложно. Они очень описательны, а еще очень длинны. Нам стоит подумать о сокращении некоторых имен для упрощения восприятия. Давайте коротко рассмотрим эти столбцы и подумаем, для каких изменение имени окажется актуальным.

- `Profile_ID` — ID пользователя, получившего рекомендацию дружбы. Это имя коротко и понятно. Оно также соответствует столбцу `Profile_ID` в `df_profile`, так что его стоит оставить как есть.
- `Selected_Friend` — существующий друг пользователя в столбце `Profile_ID`. Этот столбец можно упростить до `Friend`.
- `Selected_Friend_of_Friend` — случайно выбранный друг `Selected_Friend`, который еще не является другом `Profile_ID`. В нашем анализе этот случайный друг друга был предложен пользователю в качестве *рекомендованного друга*. Столбец можно переименовать, например, в `FoF`, короткий и легко запоминающийся акроним.
- `Friend_Requent_Sent` — этот логический столбец содержит `True`, если пользователь отправил запрос на дружбу предложенному другу друга, и `False` — в противном случае. Укоротим его имя до `Sent`.
- `Friend_Request_Accepted` — этот логический столбец содержит `True`, только если пользователь отправил запрос на дружбу и тот был принят. Его имя можно сократить до `Accepted`.

В итоге получается, что нам нужно переименовать четыре из пяти столбцов. Давайте изменим их имена и заново сгенерируем сводку (листинг 23.8).

Листинг 23.8. Переименование столбцов наблюдений

```
new_names = {'Selected_Friend': 'Friend',
             'Selected_Friend_of_Friend': 'FoF',
             'Friend_Request_Sent': 'Sent',
             'Friend_Request_Accepted': 'Accepted'}
df_obs = df_obs.rename(columns=new_names)
summarize_table(df_obs)

The table contains 4039 rows and 5 columns.
Table Summary:
```

	Profile_ID	Friend	FoF	Sent	Accepted
count	4039	4039	4039	4039	4039
unique	4039	2219	2327	2	2
top	b90a1222d2b2	89581f99fa1e	6caa597f13cc	True	True
freq	1	77	27	2519	2460

В обновленной таблице статистика стала более понятной. В общей сложности среди всех 4039 наблюдений мы видим 2219 ID `Friend` и 2327 уникальных ID `FoF`. Это означает, что в среднем каждый ID `Friend` и `FoF` используется примерно дважды. Ни один отдельный ID профиля не доминирует в данных, и это хороший признак. Это позволит нам спроектировать надежную предиктивную модель в противоположность той, что управляется сигналом одного профиля, а значит, более склонна к переобучению.

Дальнейший анализ показывает, что примерно 62 % (2519) рекомендаций привели к отправке запросов на дружбу. Это очень обнадеживающий результат. Предложения друзей друзей очень эффективны. Более того, около 60 % (2460) из отобранных рекомендаций привели к принятию запроса на дружбу. Игнорируются или отклоняются такие запросы лишь в 2 % (2519 – 2460 = 59) случаев. Естественно, эти числа предполагают отсутствие наблюдений, в которых `Sent` является `False` при `Accepted` — `True`. Такой сценарий невозможен, поскольку запрос на дружбу нельзя принять, если он еще не был отправлен. И все же в качестве дополнительной проверки мы протестируем целостность данных и убедимся, что такого у нас не случается (листинг 23.9).

Листинг 23.9. Проверка, является ли Sent = True для всех принятых запросов

```
condition = (df_obs.Sent == False) & (df_obs.Accepted == True)
assert not df_obs[condition].shape[0]
```

Исходя из наших наблюдений, пользовательское поведение отражает три возможных сценария.

- Пользователь отклоняет либо игнорирует рекомендацию дружбы, представленную в столбце `FoF`. Это происходит в 38 % случаев.
- Пользователь в ответ на рекомендацию отправляет запрос на дружбу, и он принимается. Это происходит в 62 % случаев.
- Пользователь в ответ на рекомендацию отправляет запрос на дружбу, и он отклоняется либо игнорируется. Это редкий исход, наблюдаемый всего в 1,2 % случаев.

Каждый из этих трех сценариев представляет три категории пользовательского поведения. Таким образом, можно закодировать это категориальное поведение, присвоив его паттернам *a*, *b* и *c* числа 0, 1 и 2. Последующий код выполняет это категориальное присваивание и сохраняет результат в столбце `Behavior` (листинг 23.10).

Листинг 23.10. Присваивание классов поведения пользовательским действиям

```
behaviors = []
for sent, accepted in df_obs[['Sent', 'Accepted']].values:
    behavior = 2 if (sent and not accepted) else int(sent) * int(accepted)
    behaviors.append(behavior)
df_obs['Behavior'] = behaviors
```

Python трактует логические значения True и False как 1 и 0 соответственно. Значит, эта арифметическая операция на основе наших определений поведения возвращает 0, 1 или 2

Кроме того, необходимо преобразовать ID профилей в первых трех столбцах из хеш-кодов в численные ID, согласующиеся с `df_profile.Profile_ID`. Код листинга 23.11 задействует для этого сопоставления, сохраненные в `col_to_mapping['Profile_ID']`.

Листинг 23.11. Замена всех хеш-кодов Observations численными значениями

```
for col in ['Profile_ID', 'Friend', 'FoF']:
    nums = [col_to_mapping['Profile_ID'][hash_code]
            for hash_code in df_obs[col]]
    df_obs[col] = nums

head = df_obs.head()
print(head.to_string(index=False))

Profile_ID  Friend    FoF  Sent  Accepted  Behavior
       2485    2899  2847  False     False         0
       2690    2899  3528  False     False         0
       3904    2899  3528  False     False         0
        709    2899  3403  False     False         0
        502    2899  345    True      True         1
```

Теперь `df_obs` согласуется с `df_profile`. Осталось проанализировать всего одну таблицу данных, `Friendships`, чем мы далее и займемся.

23.1.3. Анализ таблицы дружеских связей Friendships

Начнем с загрузки таблицы `Friendships` в Pandas и обобщения ее содержимого (листинг 23.12).

Листинг 23.12. Загрузка таблицы Friendships

```
df_friends = pd.read_csv('friendhook/Friendships.csv')
summarize_table(df_friends)

The table contains 88234 rows and 2 columns.
Table Summary:
```

	Friend_A	Friend_B
count	88234	88234
unique	3646	4037
top	89581f99fa1e	97ba93d9b169
freq	1043	251

В нашей социальной сети существует 88 000 дружеских связей. Они формируют довольно плотную сеть, в которой на каждый профиль FriendHook приходится приблизительно 22 друга. При этом у одного особо коммуникабельного пользователя (`89581f99fa1e`) числится более 1000 друзей. Однако точное количество дружеских связей выяснить нельзя, так как два столбца в сети несимметричны. По факту мы даже не можем проверить, правильно ли представлены все 4039 профилей в таблице.

Для более подробного анализа необходимо загрузить данные о дружбе в граф NetworkX. Код листинга 23.13 вычисляет этот социальный граф. ID узлов мы представляем с помощью численных значений, отображенных из хеш-кодов в столбцах. После вычисления графа подсчитываем количество узлов в `G.nodes`.

Листинг 23.13. Загрузка социального графа в NetworkX

```
import networkx as nx
G = nx.Graph()
for id1, id2 in df_friends.values:
    node1 = col_to_mapping['Profile_ID'][id1]
    node2 = col_to_mapping['Profile_ID'][id2]
    G.add_edge(node1, node2)

nodes = list(G.nodes)
num_nodes = len(nodes)
print(f"The social graph contains {num_nodes} nodes.")

The social graph contains 4039 nodes.
```

Попробуем почерпнуть из структуры графа больше информации, визуализировав его с помощью `nx.draw` (листинг 23.14; рис. 23.1). Заметьте, что граф этот довольно большой, поэтому его визуализация может занять от 10 до 30 с.

Листинг 23.14. Визуализация социального графа

```
import matplotlib.pyplot as plt
np.random.seed(0)
nx.draw(G, node_size=5)
plt.show()
```

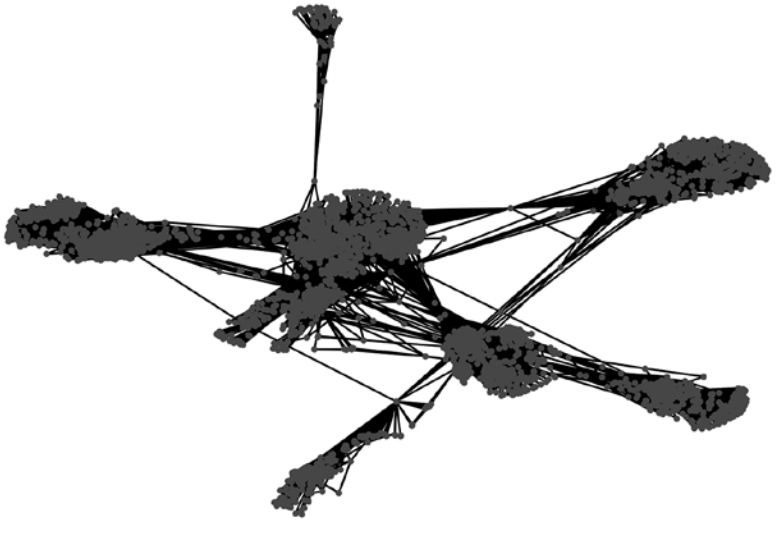


Рис. 23.1. Визуализированный социальный граф университета. Здесь отчетливо видны плотные кластеры социальных групп, которые можно извлечь с помощью марковского алгоритма кластеризации

В сети отчетливо вырисовываются густо кластеризованные социальные группы. Давайте извлечем их с помощью марковского алгоритма кластеризации и подсчитаем число кластеров (листинг 23.15).

Листинг 23.15. Поиск социальных групп с помощью алгоритма Маркова

```
import markov_clustering as mc
matrix = nx.toSciPysparse_matrix(G)
result = mc.run_mcl(matrix)
clusters = mc.get_clusters(result)
num_clusters = len(clusters)
print(f"{num_clusters} clusters were found in the social graph.")
```

```
10 clusters were found in the social graph.
```

В социальном графе обнаружено десять кластеров. Далее мы их визуализируем, закрасив каждый узел на основе ID его кластера. Для начала необходимо перебрать `clusters` и присвоить каждому узлу атрибут `cluster_id` (листинг 23.16).

Листинг 23.16. Присваивание атрибутов кластеров узлам

```
for cluster_id, node_indices in enumerate(clusters):
    for i in node_indices:
        node = nodes[i]
        G.nodes[node]['cluster_id'] = cluster_id
```

Далее мы закрасим узлы на основе присвоенных атрибутов (листинг 23.17; рис. 23.2).

Листинг 23.17. Закрашивание узлов согласно принадлежности кластерам

```
np.random.seed(0)
colors = [G.nodes[n]['cluster_id'] for n in G.nodes]
nx.draw(G, node_size=5, node_color=colors, cmap=plt.cm.tab20)
plt.show()
```

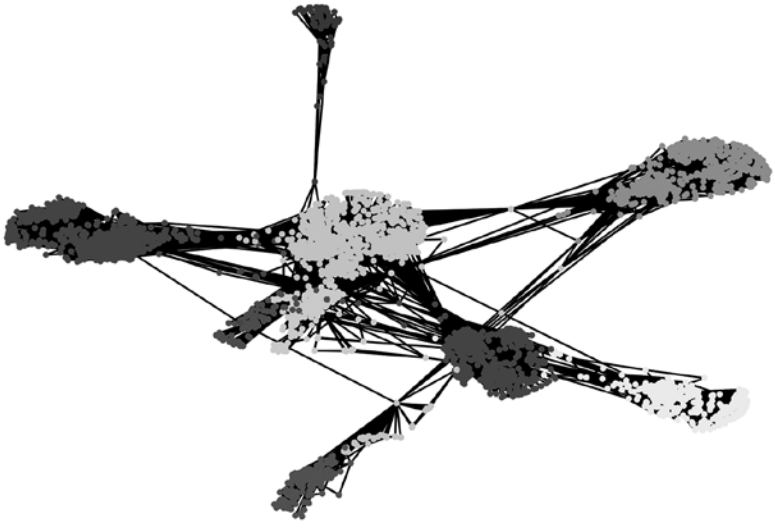


Рис. 23.2. Визуализированный социальный граф университета. С помощью кластеризации Маркова в нем найдены плотные социальные группы. Узлы в графе закрашены на основе ID их кластеров

Цвета кластеров явно соответствуют плотным социальным группам. Кластеризация оказалась эффективной, значит, присвоенные атрибуты `cluster_id` пригодятся при построении модели. Также будет полезным сохранить все пять признаков профилей в виде атрибутов узлов студентов (листинг 23.18). Переберем все строки в `df_profile` и сохраним значение каждого столбца в соответствующем ему узле.

Листинг 23.18. Присваивание атрибутов профилей узлам

```
attribute_names = df_profile.columns
for attributes in df_profile.values:
    profile_id = attributes[0]
    for name, att in zip(attribute_names[1:], attributes[1:]):
        G.nodes[profile_id][name] = att

first_node = nodes[0]
print(f"Attributes of node {first_node}:")
print(G.nodes[first_node])

Attributes of node 2899:
{'cluster_id': 0, 'Sex': 0, 'Relationship_Status': 0, 'Dorm': 5,
 'Major': 13, 'Year': 2}
```

Вот мы и закончили анализ входных данных. Далее обучим модель, прогнозирующую поведение пользователя. Для начала построим ее простой вариант, применяющий только признаки сети.

23.2. ОБУЧЕНИЕ ПРЕДИКТИВНОЙ МОДЕЛИ С ПОМОЩЬЮ ПРИЗНАКОВ СЕТИ

Наша цель — обучить на представленном наборе данных модель машинного обучения с учителем прогнозировать пользовательское поведение. Сейчас все возможные классы поведения хранятся в столбцах `Behavior` таблицы `df_obs`. Тремя метками этих классов поведения являются 0, 1 и 2. Напомню, что метка класса 2 встречается всего в 50 из 4039 отобранных экземпляров — класс 2 сильно разбалансирован по сравнению с другими метками классов. Есть основания для удаления этих 50 размеченных образцов из обучающих данных, но пока мы их оставим и посмотрим, к чему это приведет. Позднее при необходимости мы их удалим. Сейчас же мы определим массив обучающих меток классов как соответствующий столбцу `df_obs.Behavior` (листинг 23.19).

Листинг 23.19. Присваивание массива меток класса y

```
y = df_obs.Behavior.values
print(y)

[0 0 0 ... 1 1 1]
```

Теперь, когда у нас есть метки классов, нужно создать матрицу признаков `X`. Наша задача — заполнить эту матрицу признаками, взятыми из структуры социального графа. Позднее добавим в нее дополнительные признаки из профилей студентов, так что собирать всю матрицу за раз не нужно. Мы будем строить ее постепенно, добавляя новые признаки группами, чтобы лучше понять, как они влияют на

эффективность модели. С учетом сказанного далее мы создадим первую версию `X` и заполним ее базовыми признаками. Простейший вопрос, который можно задать о любом пользователе FriendHook, звучит так: «Сколько у этого пользователя друзей?» Это значение равно количеству ребер, связанных с узлом этого человека в социальном графе. Иными словами, количество друзей пользователя `n` равно `G.degree(n)`. Установим это количество в качестве первого признака в матрице. Мы переберем все строки в `df_obs` и присвоим количество ребер каждому профилю, указанному в каждой строке. Напомню, что каждая строка содержит три профиля: `Profile_ID`, `Friend` и `FoF`. Мы вычислим количество друзей каждого профиля, создав признаки `Profile_ID_Edge_Count`, `Friend_Edge_Count` и `FoF_Edge_Count`.

ПРИМЕЧАНИЕ

Не всегда легко придумать для признака удачное и согласованное имя. Вместо FoF_Edge_Count можно было выбрать в качестве имени FoF_Friend_Count. Однако сохранение согласованности вынудило бы нас также включить признак Friend_Friend_Count, что привело бы к формированию очень странного имени. В качестве альтернативы можно назвать три наших признака: Profile_Degree, Friend_Degree и FoF_Degree. Такие имена были бы короткими и информативными, но стоит помнить, что один из признаков профиля относится к специализации колледжа. В контексте колледжа ученые степени (degree) и основная специализация (major) имеют практически идентичное определение, поэтому именование на основе степеней в дальнейшем может вызвать путаницу. Именно поэтому мы используем фрагмент Edge_Count.

Далее мы сгенерируем матрицу признаков количества ребер размером 3×4039. Нам нужен способ отслеживать эти признаки вместе с соответствующими им именами. Требуется также удобный способ обновлять признаки и их имена, дополняя их входными данными. Простым решением будет сохранить признаки в таблице Pandas `df_features`. Она позволит обращаться к матрице признаков через `df_features.values`. Давайте вычислим `df_features`, чтобы создать начальную версию матрицы признаков (листинг 23.20).

Листинг 23.20. Создание матрицы признаков из количеств ребер

```
cols = ['Profile_ID', 'Friend', 'FoF']
features = {f'{col}_Edge_Count': [] for col in cols}
for node_ids in df_obs[cols].values:
    for node, feature_name in zip(node_ids, features.keys()):
        degree = G.degree(node)
        features[feature_name].append(degree)

df_features = pd.DataFrame(features)
X = df_features.values
```

Напомню, что степень узла равна количеству его ребер. Таким образом, G.degree(n) возвращает количество друзей, связанных с пользователем n

Начальная обучающая выборка готова. Теперь проверим качество сигнала в ней, обучив и протестировав простую модель. У нас на выбор есть несколько моделей. Одним из разумных вариантов является дерево решений. Деревья решений

способны обрабатывать нелинейные границы решений и легко поддаются трактовке. В то же время они уязвимы для переобучения, поэтому для подобающей оценки эффективности такой модели потребуется кросс-валидация. Код листинга 23.21 обучает дерево решений на подвыборке `(X, y)` и оценивает результаты на оставшихся данных. Во время оценки нужно помнить, что метки класса 2 сильно разбалансированы. Это значит, что более адекватным показателем оценки эффективности будет F-мера, а не простая точность.

ПРИМЕЧАНИЕ

В оставшейся части этой главы мы будем многократно обучать и тестировать модели классификации. Код листинга 23.21 определяет для этой цели функцию evaluate, получающую обучающую выборку (X, y) и тип модели, установленный как DecisionTreeClassifier. Затем эта функция разделяет X, y на обучающую и контрольную выборки, обучает классификатор и вычисляет F-меру, используя контрольную выборку. Наконец, она возвращает F-меру и классификатор для оценки.

Листинг 23.21. Обучение и оценка дерева решений

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score

def evaluate(X, y, model_type=DecisionTreeClassifier, **kwargs):
    np.random.seed(0)
    X_train, X_test, y_train, y_test = train_test_split(X, y)
    clf = model_type(**kwargs)
    clf.fit(X_train, y_train)
    pred = clf.predict(X_test)
    f_measure = f1_score(pred, y_test, average='macro')
    return f_measure, clf

f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:0.2f}")

The f-measure is 0.37
```

В оставшейся части раздела мы используем эту функцию неоднократно. Она обучает классификатор на подвыборке данных в (X, y). Тип классификатора указывается с помощью параметра model_type. Здесь этот параметр установлен на дерево решений. Дополнительные гиперпараметры классификатора можно установить с помощью **kwargs. После обучения эффективность классификатора оценивается на основе сохраненной подвыборки данных

Вычисляет F-меру

Разделяет (X, y) на обучающую и контрольную выборки

Параметр average='macro' нужен в связи с присутствием в обучающих данных трех меток классов

Это случайное стартовое значение гарантирует, что (X, y) от выполнения к выполнению будет разделяться согласованно

Полученная F-мера ужасна! Очевидно, что одного только количества ребер недостаточно для прогнозирования поведения пользователя. Возможно, здесь потребуется более сложная мера центральности узла. Ранее мы узнали, что мера центральности PageRank может оказаться более информативной, чем количество ребер. Приведет ли добавление значений PageRank в обучающую выборку к улучшению эффективности модели? Сейчас выясним (листинг 23.22).

Листинг 23.22. Добавление признаков PageRank

```
node_to_pagerank = nx.pagerank(G)
features = {f'{col}_PageRank': [] for col in cols}
for node_ids in df_obs[cols].values:
    for node, feature_name in zip(node_ids, features.keys()):
        pagerank = node_to_pagerank[node]
        features[feature_name].append(pagerank)

def update_features(new_features):
    for feature_name, values in new_features.items():
        df_features[feature_name] = values
    return df_features.values

X = update_features(features)
f_measure, clf = evaluate(X, y)

print(f"The f-measure is {f_measure:0.2f}")

The f-measure is 0.38
```

Эта функция используется повторно. Она обновляет таблицу df_features новыми признаками из словаря new_features

Возвращает измененную матрицу признаков

F-мера остается примерно такой же. Простых измерений центральности тут недостаточно. Нам нужно расширить `X`, включив в нее социальные группы, обнаруженные алгоритмом Маркова. В конце концов два человека, относящиеся к одной социальной группе, с большей вероятностью окажутся друзьями. А как нам внедрить эти социальные группы в матрицу признаков? В качестве наивного решения можно присвоить атрибут `cluster_id` каждого связанного узла в виде признака социальной группы. Однако такой подход имеет серьезный недостаток: существующие ID кластеров актуальны только для конкретного социального графа в `G` и никак не связаны с сетью любого другого колледжа. Иными словами, модель, обученную на ID кластеров в `G`, не получится применить к графу другого колледжа из `G_other`. Это не сработает. Одна из наших целей состоит в построении модели, способной обобщаться на другие колледжи. Значит, нужно более тонкое решение.

В качестве альтернативного подхода можно рассмотреть следующий двоичный вопрос: «Входят ли два человека в одну социальную группу?» Если да, то они с большей вероятностью в конечном итоге станут друзьями во FriendHook. Это двоичное сравнение между каждой парой ID профилей можно выполнить в одной строке наблюдений. Говоря точнее, мы можем спросить следующее.

- Относится ли пользователь из столбца `Profile_ID` к той же социальной группе, что и друг из столбца `Friend`? Назовем этот признак `Shared_Cluster_id_f`.
- Относится ли пользователь из столбца `Profile_ID` к той же социальной группе, что и друг друга из столбца `FoF`? Назовем этот признак `Shared_Cluster_id_fof`.
- Относится ли друг из столбца `Friend` к той же социальной группе, что и друг друга из столбца `FoF`? Назовем этот признак `Shared_Cluster_f_fof`.

Ответим на эти вопросы, добавив три дополнительных признака, и проверим, повысится ли в результате эффективность модели (листинг 23.23).

Листинг 23.23. Добавление признаков социальных групп

```
features = {f'Shared_Cluster_{e}': []
            for e in ['id_f', 'id_fof', 'f_fof']}

i = 0
for node_ids in df_obs[cols].values:
    c_id, c_f, c_fof = [G.nodes[n]['cluster_id']
                        for n in node_ids]
    features['Shared_Cluster_id_f'].append(int(c_id == c_f))
    features['Shared_Cluster_id_fof'].append(int(c_id == c_fof))
    features['Shared_Cluster_f_fof'].append(int(c_f == c_fof))

X = update_features(features)
f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:0.2f}")

The f-measure is 0.43
```

F-мера возросла с 0,38 до 0,43. Эффективность по-прежнему мала, но включение социальных групп несколько улучшило нашу модель. Насколько же значимы признаки социальных групп по сравнению с текущим качеством модели? Это можно проверить с помощью атрибута `feature_importance_` обученного классификатора (листинг 23.24).

Листинг 23.24. Ранжирование признаков по показателю их важности

```
def view_top_features(clf, feature_names):
    for i in np.argsort(clf.feature_importances_)[::-1]:
        feature_name = feature_names[i]
        importance = clf.feature_importances_[i]
        if not round(importance, 2):
            break

        print(f"{feature_name}: {importance:0.2f}")
feature_names = df_features.columns
view_top_features(clf, feature_names)

Shared_Cluster_id_fof: 0.18
FoF_PageRank: 0.17
Profile_ID_PageRank: 0.17
Friend_PageRank: 0.15
FoF_Edge_Count: 0.12
Profile_ID_Edge_Count: 0.11
Friend_Edge_Count: 0.10
```

Выводит топ признаков вместе с показателями их важности в классификаторе на основе степени значимости

Сортирует признаки на основе показателя их важности

Признаки с показателем важности менее 0,01 не отображаются

Самым значимым признаком в модели оказался `Shared_Cluster_id_fof`. Иными словами, совпадение социальных групп, к которым относятся пользователи и друзья их друзей, — наиболее значимый предиктор будущей онлайн-дружбы. Однако признаки PageRank также находятся в верхней части списка, указывая на то, что и центральность социального графа играет роль в определении дружбы. Естественно, эффективность нашей модели все еще низкая, поэтому следует быть осторожными с выводами о влиянии признаков на прогнозы. Вместо этого следует сосредоточиться на повышении качества модели. Какие еще признаки графа можно задействовать? Возможно, на прогнозы способен повлиять размер кластеров сети. Это можно выяснить, но здесь важно учитывать, что модель должна остаться обобщаемой. Размер кластеров может по неясной причине занять место ID кластеров, сделав модель специфичной для конкретного колледжа. Разберем, как такое может произойти.

Предположим, что в нашем наборе данных есть два социальных кластера, A и B. Они содержат 110 и 115 студентов соответственно. Таким образом, их размер практически идентичен и на прогноз влиять не должен. А теперь мы дополнительно предположим, что студенты из кластера A более склонны становиться друзьями во FriendHook, чем студенты из кластера B. Наша модель обнаружит это в процессе обучения и свяжет размер 110 со склонностью к установлению дружбы. По сути, она будет рассматривать этот размер как ID кластера. Это вызовет проблемы в дальнейшем, если модель вдруг встретит новый кластер размером 110.

Стоит ли совсем игнорировать размер кластеров? Не обязательно. Будучи учеными, мы хотим полноценно изучить влияние размера кластеров на прогнозы модели. Но при этом следует сохранять осторожность — если размер кластеров не оказывает значительного влияния на качество модели, то его необходимо убрать из списка признаков. А вот если он существенно улучшит ее эффективность, мы внимательно переоценим нашу конфигурацию. Давайте посмотрим, что произойдет при добавлении к признакам размера кластеров (листинг 23.25).

Листинг 23.25. Добавление к признакам размера кластеров

```
cluster_sizes = [len(cluster) for cluster in clusters]
features = {f'{col}_Cluster_Size': [] for col in cols}
for node_ids in df_obs[cols].values:
    for node, feature_name in zip(node_ids, features.keys()):
        c_id = G.nodes[node]['cluster_id']
        features[feature_name].append(cluster_sizes[c_id])

X = update_features(features)
f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:0.2f}")

The f-measure is 0.43
```

Добавление размера кластеров не привело к улучшению модели, так что в качестве предосторожности мы удалим его из набора признаков (листинг 23.26).

Листинг 23.26. Удаление размера кластеров из признаков

```
import re
def delete_features(df_features, regex=r'Cluster_Size'):

    df_features.drop(columns=[name for name in df_features.columns
                              if re.search(regex, name)], inplace=True)
    return df_features.values

X = delete_features(df_features)
```

Удаляет все названия признаков в df_features, которые соответствуют регулярному выражению regex. Эта функция повторно используется в других частях главы

Возвращает измененную матрицу признаков

F-мера остается равной 0,43. Что еще можно предпринять? Возможно, следует поразмышлять нестандартно. Каким образом социальные связи могут определять реальное поведение пользователей? Есть ли какие-то дополнительные, специфичные для данной задачи сигналы, которые можно задействовать? Да! Рассмотрим следующий сценарий. Предположим, что мы анализируем студента по имени Алекс, ID узла которого в сети `G` равен `n`. У Алекса во FriendHook есть 50 друзей, которые доступны через `G[n]`. Мы случайным образом отбираем двух из его друзей в `G[n]`. ID их узлов будут представлены как `a` и `b`. После этого мы проверяем, являются ли `a` и `b` друзьями. Являются. Похоже, что `a` находится в `list(G[n])`. Повторяем эту процедуру 100 раз. В 95 % отобранных экземпляров `a` является другом `b`. Получается, есть 95%-ная вероятность того, что любые двое друзей Алекса также являются друзьями. Эту вероятность мы назовем *вероятностью общей дружбы*. Теперь представим, что во FriendHook регистрируется Мэри и добавляет в качестве друга Алекса. Велик шанс, что она также подружится с друзьями Алекса, но гарантировать этого, конечно, нельзя. Однако вероятность общей дружбы, равная 0,95, дает нам бóльшую уверенность, чем вероятность 0,1.

Теперь попробуем внедрить эту вероятность в наши признаки, начав с ее вычисления для каждого узла в `G` (листинг 23.27). Полученные сопоставления «узел — вероятность» сохраним в словаре `friend_sharing_likelihood`.

Листинг 23.27. Вычисление вероятности общей дружбы

```
friend_sharing_likelihood = {}
for node in nodes:
    neighbors = list(G[node])
    friendship_count = 0
    total_possible = 0
    for i, node1 in enumerate(neighbors[:-1]):
        for node2 in neighbors[i + 1:]:
            if node1 in G[node2]:
                friendship_count += 1

            total_possible += 1

prob = friendship_count / total_possible if total_possible else 0
friend_sharing_likelihood[node] = prob
```

Отслеживает число общих дружеских связей среди соседей

Отслеживает число возможных общих дружеских связей. Заметьте, с помощью теории графов можно доказать, что это значение всегда равно len(neighbors) * (len(neighbors - 1))

Проверяет, являются ли двое соседей друзьями

Далее мы сгенерируем признак вероятности общей дружбы для каждого из трех ID профилей (листинг 23.28). После добавления этих признаков еще раз оценим эффективность модели.

Листинг 23.28. Добавление вероятности общей дружбы в качестве признака

```
features = {f'{col}_Friend_Sharing_Likelihood': [] for col in cols}
for node_ids in df_obs[cols].values:
    for node, feature_name in zip(node_ids, features.keys()):
        sharing_likelihood = friend_sharing_likelihood[node]
        features[feature_name].append(sharing_likelihood)

X = update_features(features)
f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:0.2f}")

The f-measure is 0.49
```

Эффективность возросла с 0,43 до 0,49. Она все еще невысока, но постепенно улучшается. А как вероятность общей дружбы соотносится с остальными признаками в модели? Сейчас выясним (листинг 23.29).

Листинг 23.29. Ранжирование признаков по показателю важности

```
feature_names = df_features.columns
view_top_features(clf, feature_names)

Shared_Cluster_id_fof: 0.18
Friend_Friend_Sharing_Likelihood: 0.13
FoF_PageRank: 0.11
Profile_ID_PageRank: 0.11
Profile_ID_Friend_Sharing_Likelihood: 0.10
FoF_Friend_Sharing_Likelihood: 0.10
FoF_Edge_Count: 0.08
Friend_PageRank: 0.07
Profile_ID_Edge_Count: 0.07
Friend_Edge_Count: 0.06
```

Один из наших признаков общей дружбы занял в списке высокую строчку, оказавшись на втором месте между `Shared_Cluster_id_fof` и `FoF_PageRank`. Нестандартное мышление позволило улучшить модель. Однако она по-прежнему не завершена. Показатель F-меры 0,49 неприемлем — его необходимо улучшить. Для этого потребуется внедрить в модель признаки из профилей, сохраненных в `df_profiles`.

23.3. ДОБАВЛЕНИЕ В МОДЕЛЬ ПРИЗНАКОВ ПРОФИЛЕЙ

Наша цель — внедрить в матрицу признаков атрибуты профилей `Sex`, `Relationship_Status`, `Major`, `Dorm` и `Year`. Основываясь на опыте работы с данными сети, это можно реализовать тремя способами.

- *Извлечение точного значения* — можно сохранить точное значение признака профиля, связанное с каждым из трех столбцов ID профиля в `df_obs`. Эта операция аналогична тому, как мы использовали точные значения количества ребер и результаты PageRank из сети.

  Пример признака: семейное положение друга друга в `df_obs`.

- *Определение равнозначности* — имея атрибут профиля, можно попарно сравнить его со всеми тремя столбцами ID профиля в `df_obs`. Для каждого сравнения мы будем возвращать логический признак, обозначающий, совпадает ли этот атрибут в двух столбцах. Этот процесс аналогичен выполненной ранее проверке принадлежности пары профилей к одной социальной группе.

  Пример признака: проживают ли конкретный пользователь и друг его друга в одном общежитии? Да или нет?

- *Размер* — получив атрибут профиля, можно вернуть количество профилей, в которых он тоже есть. Это будет аналогично попытке включить в модель размеры социальных групп.

  Пример признака: число студентов, проживающих в одном общежитии.

Воспользуемся извлечением точного значения для расширения матрицы признаков. Какие из пяти атрибутов окажутся удачными кандидатами для применения этого приема? Что ж, категориальные значения `Sex`, `Relationship_Status` и `Year` к колледжу не относятся и должны оставаться согласованными для всех учебных заведений. Однако это не относится к `Dorm` — названиям общежитий в сетях других колледжей. Наша цель — обучить модель, которую можно применять к другим социальным графам, поэтому атрибут `Dorm` окажется неподходящим признаком для извлечения точного значения.

А что насчет атрибута `Major`? Здесь ситуация сложнее. Определенные специализации вроде биологии и экономики имеются в большинстве колледжей и университетов. Другие же, например строительное дело, могут встречаться в более технических заведениях, но не в программе гуманитарных колледжей. При этом некоторым редким специализациям вроде игры на волынке и астробиологии обучают лишь в нескольких нишевых вузах. Таким образом, можно ожидать лишь некоторой согласованности среди специализаций. В связи с этим модель, задействующая точные значения специализаций, будет переиспользуемой частично. Потенциально этот частичный сигнал может усилить предиктивную эффективность в некоторых учебных заведениях, но это произойдет ценой ее падения в других. Стоит ли компромисс того? Возможно, но ответ неочевиден. Пока что мы посмотрим, насколько удачную модель нам удастся обучить без использования костыля в виде значений `Major`. Если же к улучшению модели это не приведет, еще раз пересмотрим свое решение.

Теперь применим извлечение точного значения к `Sex`, `Relationship_Status` и `Year`, после чего проверим, улучшилась ли модель (листинг 23.30).

Листинг 23.30. Добавление в качестве признаков точных значений профилей

```
attributes = ['Sex', 'Relationship_Status', 'Year']
for attribute in attributes:
    features = {f'{col}_{attribute}_Value': [] for col in cols}
    for node_ids in df_obs[cols].values:
        for node, feature_name in zip(node_ids, features.keys()):
            att_value = G.nodes[node][attribute]
            features[feature_name].append(att_value)

    X = update_features(features)

f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:0.2f}")

The f-measure is 0.74
```

Вау! F-мера значительно возросла, поднявшись с 0,49 до 0,74. Признаки профилей обеспечили очень полезный сигнал, но мы все еще можем добиться лучшего. Нам нужно внедрить информацию из атрибутов `Major` и `Dorm`. Для этого отлично подойдет метод определения равнозначности. Вопрос о том, обучаются два студента по одной специализации или проживают они в одном общежитии, не зависит от их принадлежности к тому или иному вузу. Далее мы применим определение равнозначности к атрибутам `Major` и `Dorm`, после чего заново вычислим F-меру (листинг 23.31).

Листинг 23.31. Добавление признаков с помощью определения равнозначности

```
attributes = ['Major', 'Dorm']
for attribute in attributes:
    features = {f'Shared_{attribute}_{e}': []
            for e in ['id_f', 'id_fof', 'f_fof']}

    for node_ids in df_obs[cols].values:
        att_id, att_f, att_fof = [G.nodes[n][attribute]
                                  for n in node_ids]
        features[f'Shared_{attribute}_id_f'].append(int(att_id == att_f))
        features[f'Shared_{attribute}_id_fof'].append(int(att_id == att_fof))
        features[f'Shared_{attribute}_f_fof'].append(int(att_f == att_fof))

    X = update_features(features)

f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:0.2f}")

The f-measure is 0.82
```

F-мера возросла до 0,82. Внедрение атрибутов `Major` и `Dorm` положительно сказалось на эффективности модели. Теперь мы рассмотрим добавление размеров `Major` и `Dorm`. Для этого подсчитаем число студентов, связанных с каждой специализацией и общежитием, включив полученные значения в качестве признаков. Однако

здесь нужна осторожность. Как уже говорилось, наша обученная модель может схалтурить, использовав размер вместо ID категории. Например, как мы видели, в крупнейшем общежитии проживают 2700 студентов. Таким образом, можно с легкостью опознать это общежитие только по его размеру. Нам нужно продвигаться осторожно. Давайте посмотрим, что произойдет при внедрении в качестве признаков размеров `Major` и `Dorm`. Если это особо не скажется на эффективности модели, то мы удалим эти признаки из модели. В противном случае снова переоценим конфигурацию (листинг 23.32).

Листинг 23.32. Добавление признаков профилей, связанных с размером

```
from collections import Counter

for attribute in ['Major', 'Dorm']:
    counter = Counter(df_profile[attribute].values)
    att_to_size = {k: v
                   for k, v in counter.items()}
    features = {f'{col}_{attribute}_Size': [] for col in cols}
    for node_ids in df_obs[cols].values:
        for node, feature_name in zip(node_ids, features.keys()):
            size = att_to_size[G.nodes[node][attribute]]
            features[feature_name].append(size)

    X = update_features(features)

f_measure, clf = evaluate(X, y)
print(f"The f-measure is {f_measure:0.2f}")

The f-measure is 0.85
```

Отслеживает, сколько раз каждый атрибут встречается в наборе данных

Эффективность возросла с 0,82 до 0,85. Добавление размеров действительно привело к улучшению модели. Давайте рассмотрим это влияние более подробно, начав с вывода показателей важности признаков (листинг 23.33).

Листинг 23.33. Ранжирование признаков по показателю их важности

```
feature_names = df_features.columns.values
view_top_features(clf, feature_names)

FoF_Dorm_Size: 0.25
Shared_Cluster_id_fof: 0.16
Shared_Dorm_id_fof: 0.05
FoF_PageRank: 0.04
Profile_ID_Major_Size: 0.04
FoF_Major_Size: 0.04
FoF_Edge_Count: 0.04
Profile_ID_PageRank: 0.03
Profile_ID_Friend_Sharing_Likelihood: 0.03
Friend_Friend_Sharing_Likelihood: 0.03
Friend_Edge_Count: 0.03
Shared_Major_id_fof: 0.03
```

```
FoF_Friend_Sharing_Likelihood: 0.02
Friend_PageRank: 0.02
Profile_ID_Dorm_Size: 0.02
Profile_ID_Edge_Count: 0.02
Profile_ID_Sex_Value: 0.02
Friend_Major_Size: 0.02
Profile_ID_Relationship_Status_Value: 0.02
FoF_Sex_Value: 0.01
Friend_Dorm_Size: 0.01
Profile_ID_Year_Value: 0.01
Friend_Sex_Value: 0.01
Shared_Major_id_f: 0.01
Friend_Relationship_Status_Value: 0.01
Friend_Year_Value: 0.01
```

По важности здесь доминируют два признака: `FoF_Dorm_Size` и `Shared_Cluster_id_fof`. Их показатели составляют 0,25 и 0,16 соответственно. Значения остальных признаков ниже 0,01.

Присутствие `FoF_Dorm_Size` несколько тревожит. Как говорилось, в данных сети доминирует одно общежитие. Возможно, наша модель просто заучила его, исходя из размера? Чтобы ответить на этот вопрос, визуализируем обученное дерево решений (листинг 23.34). Для упрощения уменьшим его глубину до 2, чтобы ограничить вывод решениями двух наиболее влиятельных признаков.

Листинг 23.34. Отображение ведущих ветвей дерева

```
from sklearn.tree import export_text

clf_depth2 = DecisionTreeClassifier(max_depth=2)
clf_depth2.fit(X, y)
text_tree = export_text(clf_depth2, feature_names=list(feature_names))
print(text_tree)

|--- FoF_Dorm_Size <= 278.50
|    |--- Shared_Cluster_id_fof <=
|    |    |--- class: 0
|    |--- Shared_Cluster_id_fof >
|    |    |--- class: 0
|--- FoF_Dorm_Size > 278.50
|    |--- Shared_Cluster_id_fof <= 0.50
|    |    |--- class: 0
|    |--- Shared_Cluster_id_fof > 0.50
|    |    |--- class: 1
```

Для функции export_text проблематично получать на входе массивы NumPy, поэтому мы преобразуем feature_names в список

Размер общежития, где живет друг друга, меньше 279. В таком случае наиболее вероятной меткой класса будет 0 (игнорирование рекомендации друга)

Размер общежития, где живет друг друга, меньше или равен 279

Пользователь и друг его друга не относятся к одной социальной группе. Наиболее вероятной меткой класса будет 0

Пользователь и друг его друга относятся к одной социальной группе. Наиболее вероятной меткой класса будет 1 (связь во FriendHook установлена)

Согласно дереву, наиболее важный сигнал представлен тем, окажется ли `FoF_Dorm_Size` меньше чем 279. Если в общежитии, где живет друг друга, числится меньше 279 студентов, значит, `FoF` и пользователь вряд ли станут друзьями во FriendHook. В противном случае они наверняка подружатся, если относятся к одной социальной

группе (`Shared_Cluster_id_fof > 0.50`). Здесь напрашивается вопрос: «Во скольких общежитиях числится не менее 279 студентов?» Давайте проверим (листинг 23.35).

Листинг 23.35. Поиск общежитий, где проживает не менее 279 студентов

```
counter = Counter(df_profile.Dorm.values)
for dorm, count in counter.items():
    if count < 279:
        continue

    print(f"Dorm {dorm} holds {count} students.")
```

```
Dorm 12 holds 2739 students.
Dorm 1 holds 413 students.
```

Всего в 15 общежитиях проживают более чем по 279 зарегистрированных во FriendHook студентов. По сути, наша модель при принятии решений опирается на два наиболее крупных общежития. Это ставит нас в затруднительное положение: с одной стороны, наблюдаемые сигналы очень интересны. Связи FriendHook с большей вероятностью образуются в определенных общежитиях, так как их размер играет в этом значительную роль. Данный вывод позволит разработчикам FriendHook лучше понять пользовательское поведение, что, в свою очередь, может привести к вовлечению в сеть дополнительных пользователей. Так что это знание нам определенно полезно. Однако у этой модели есть серьезный недостаток.

Модель сосредоточена преимущественно на двух крупнейших общежитиях, что может препятствовать ее обобщению на кампусы других колледжей. Возьмем, к примеру, кампус, общежитие которого может вместить не более 200 студентов. В этом случае наша модель никак не сможет спрогнозировать поведение пользователей.

ПРИМЕЧАНИЕ

Теоретически этой ситуации можно избежать, разделив размер общежития на общее количество студентов. Это действие гарантирует, что признак размера общежития всегда будет находиться между 0 и 1.

Больше же волнует реальная вероятность того, что наша модель просто опиралась на поведение, характерное лишь для жильцов двух конкретных общежитий. Именно этого сценария просит избежать условие задачи. Что же делать?

К сожалению, здесь нет очевидно правильного ответа. Иногда те, кто занимается данными, вынуждены принимать трудные решения, каждое из которых несет свои риски и компромиссы. Можно оставить наш список признаков неизменным, чтобы сохранить высокую эффективность модели, но так мы столкнемся с риском невозможности ее обобщения на другие вузы. В качестве альтернативы можно удалить связанные с размером признаки и сохранить модель обобщаемой ценой падения ее общей точности.

Вероятно, есть и третий вариант. Можно попробовать удалить связанные с размером признаки, параллельно изменив выбор классификатора. При этом возникнет некоторая вероятность того, что мы достигнем сопоставимой эффективности без использования размера общежитий. Это маловероятно, но попытаться стоит. Давайте присвоим копию текущей матрицы признаков переменной `X_with_sizes` (на случай, если она понадобится в дальнейшем) и затем удалим из матрицы `X` все признаки, связанные с размером (листинг 23.36). После этого поищем другие способы добиться эффективности выше 0,82.

Листинг 23.36. Удаление всех признаков, связанных с размером

```
X_with_sizes = X.copy()
X = delete_features(df_features, regex=r'_Size')
```

23.4. ОПТИМИЗАЦИЯ ЭФФЕКТИВНОСТИ ПРИ КОНКРЕТНОМ НАБОРЕ ПРИЗНАКОВ

В главе 22 мы узнали, что модели случайных лесов зачастую превосходят деревья решений по эффективности. Поможет ли изменение типа модели с дерева решений на случайный лес улучшить результаты в нашем случае? Сейчас выясним (листинг 23.37).

Листинг 23.37. Обучение и оценка случайного леса

```
from sklearn.ensemble import RandomForestClassifier
f_measure, clf = evaluate(X, y, model_type=RandomForestClassifier)
print(f"The f-measure is {f_measure:0.2f}")

The f-measure is 0.75
```

О нет! Эффективность модели, наоборот, упала. Как такое могло произойти? Что ж, нам достоверно известно, что случайные леса обычно превосходят деревья решений, но это не гарантирует, что так будет всегда. В определенных сценариях обучения деревья решений справляются лучше. Похоже, что у нас именно такой сценарий. При нашем наборе данных не получится улучшить качество прогнозирования за счет изменения модели на случайный лес.

ПРИМЕЧАНИЕ

В машинном обучении с учителем есть доказанная теорема под названием No Free Lunch («Бесплатных обедов не бывает»). Говоря простым языком, она утверждает, что конкретный алгоритм обучения не может всегда превосходить все остальные алгоритмы. Иными словами, мы не можем опираться на один и тот же алгоритм в любом виде задачи по обучению. Несмотря на то что некоторые из них будут отлично справляться в большинстве случаев, так будет не всегда. Случайные леса хорошо показывают себя в большинстве задач, но не во всех. В частности, они малоэффективны, когда прогноз зависит всего от одного или двух входных признаков. Случайная выборка признаков может разбавить этот сигнал и ухудшить качество прогнозов.

Смена типа модели не помогла. Возможно, удастся повысить ее эффективность путем оптимизации гиперпараметров. В данной книге мы сосредоточились на одном гиперпараметре дерева решений — максимальной глубине. Сейчас его значение — `None`, то есть глубина дерева не ограничена. Позволит ли ее ограничение улучшить качество прогнозов? Это можно проверить, используя простой поиск по сетке. Мы просканируем значения параметра `max_depth` в диапазоне от 1 до 100, выбрав ту глубину, при которой достигается максимальная точность (листинг 23.38).

Листинг 23.38. Оптимизация максимальной глубины с помощью поиска по сетке

```
from sklearn.model_selection import GridSearchCV
np.random.seed(0)

hyperparams = {'max_depth': list(range(1, 100)) + [None]}
clf_grid = GridSearchCV(DecisionTreeClassifier(), hyperparams,
                        scoring='f1_macro', cv=2)
clf_grid.fit(X, y)
best_f = clf_grid.best_score_
best_depth = clf_grid.best_params_['max_depth']
print(f"A maximized f-measure of {best_f:.2f} is achieved when "
      f"max_depth equals {best_depth}")

A maximized f-measure of 0.84 is achieved when max_depth equals 5
```

Передавая cv=2, мы выполняем двукратную кросс-валидацию для достижения лучшей согласованности с текущим случайным разделением (X, y) на обучающую и контрольную выборки. Заметьте, что поиск по сетке может разделять данные несколько иначе, что приведет к колебаниям значения F-меры

Установка `max_depth` равным `5` ведет к повышению F-меры с 0,82 до 0,84. Этот уровень точности сопоставим с нашей моделью, использовавшей размер общежитий. Таким образом, мы достигли равнозначной эффективности без применения этого признака. Естественно, это еще не все: мы не можем выполнить честное сравнение без предварительного поиска по сетке для матрицы признаков `X_with_sizes`, включающей размеры. Даст ли оптимизация `X_with_sizes` еще более точный классификатор? Давайте выясним это (листинг 23.39).

ПРИМЕЧАНИЕ

Любопытные читатели могут поинтересоваться, можно ли добиться улучшения результата случайного леса, выполняя поиск по сетке для некоторого количества деревьев. Конкретно в данном случае ответ будет «нет». Изменение числа деревьев со 100 на некоторое другое значение не окажет значительного влияния на точность модели.

Листинг 23.39. Применение поиска по сетке к обучающим данным, включающим размеры

```
np.random.seed(0)
clf_grid.fit(X_with_sizes, y)
best_f = clf_grid.best_score_
best_depth = clf_grid.best_params_['max_depth']
print(f"A maximized f-measure of {best_f:.2f} is achieved when "
      f"max_depth equals {best_depth}")

A maximized f-measure of 0.85 is achieved when max_depth equals 6
```

Поиск по сетке не привел к улучшению эффективности модели при обучении на `X_with_sizes`. Таким образом, можно заключить, что при выборе правильной максимальной глубины модель с учетом размеров дает примерно тот же результат, что и модель без их учета. То есть можно обучить обобщаемую, не зависящую от размера модель без ущерба для ее качества. Это отличная новость! Давайте обучим дерево решений на `X`, используя `max_depth`, равную `5`, после чего проанализируем нашу модель (листинг 23.40).

Листинг 23.40. Обучение дерева решений при max_depth, равной 5

```
clf = DecisionTreeClassifier(max_depth=5)
clf.fit(X, y)
```

23.5. ИНТЕРПРЕТАЦИЯ ОБУЧЕННОЙ МОДЕЛИ

Теперь мы выведем показатели важности признаков нашей модели (листинг 23.41).

Листинг 23.41. Ранжирование признаков по показателю их важности

```
feature_names = df_features.columns
view_top_features(clf, feature_names)

Shared_Dorm_id_fof: 0.42
Shared_Cluster_id_fof: 0.29
Shared_Major_id_fof: 0.10
Shared_Dorm_f_fof: 0.06
Profile_ID_Relationship_Status_Value: 0.04
Profile_ID_Sex_Value: 0.04
Friend_Edge_Count: 0.02
Friend_PageRank: 0.01
Shared_Dorm_id_f: 0.01
```

Осталось всего девять значительных признаков. Ведущие четыре относятся к общежитиям, социальным группам и специализациям. За ними следуют признаки, обозначающие категорию `Sex` и `Relationship Status` пользователя. Простые признаки сети вроде количества ребер и PageRank оказались в самом низу списка. Интересно, что вероятность общей дружбы вообще в список ведущих признаков не попала. А ведь для его реализации потребовалось немало воображения и усилий. Было приятно видеть, как после его добавления F-мера поднялась на 0,06 единицы. Однако в итоге все эти усилия оказались бессмысленными. При наличии достаточного числа дополнительных признаков вероятность наличия общей дружбы оказалась излишней. Подобный исход иногда расстраивает. К сожалению, выбор признаков — все же скорее творческий процесс, чем научный. Сложно заранее предположить, какие признаки стоит использовать, а каких избегать. Невозможно предугадать, как в полной мере признак интегрируется в модель, пока мы ее не обучим. Это не значит, что мы не должны проявлять креативность, поскольку она зачастую окупает затраченные усилия. Как ученые, мы должны экспериментировать!

Следует пытаться задействовать каждый доступный сигнал, пока не будет достигнута адекватная эффективность модели.

Перейдем к ведущим признакам. Только у трех из них показатель важности превышает 0,1: `Shared_Dorm_id_fof`, `Shared_Cluster_id_fof` и `Shared_Major_id_fof`. Таким образом, модель управляется в первую очередь следующими тремя вопросами.

- Проживают ли пользователь и друг его друга в одном общежитии? Да или нет?
- Относятся ли пользователь и друг его друга к одной социальной группе? Да или нет?
- Обучаются ли пользователь и друг его друга на одной специальности? Да или нет?

Логично предположить, что в случае положительного ответа на все эти вопросы пользователь и друг его друга наверняка установят связь во FriendHook. Давайте протестируем эту догадку с помощью вывода дерева (листинг 23.42). Ограничим его глубину до 3, чтобы упростить вывод и обеспечить должное представление трех ведущих признаков.

Листинг 23.42. Отображение ведущих ветвей дерева

```
clf_depth3 = DecisionTreeClassifier(max_depth=3)
clf_depth3.fit(X, y)
text_tree = export_text(clf_depth3,
                        feature_names=list(feature_names))
print(text_tree)
```

```
|--- Shared_Dorm_id_fof <= 0.50
|    |--- Shared_Cluster_id_fof <= 0.50
|    |    |--- Shared_Major_id_fof <= 0.50
|    |    |    |--- class: 0
|    |    |--- Shared_Major_id_fof > 0.50
|    |    |    |--- class: 0
|    |--- Shared_Cluster_id_fof > 0.50
|    |    |--- Shared_Major_id_fof <= 0.50
|    |    |    |--- class: 0
|    |    |--- Shared_Major_id_fof > 0.50
|    |    |    |--- class: 1
|--- Shared_Dorm_id_fof > 0.50
|    |--- Shared_Cluster_id_fof <= 0.50
|    |    |--- Profile_ID_Sex_Value <= 0.50
|    |    |    |--- class: 0
|    |    |--- Profile_ID_Sex_Value > 0.50
|    |    |    |--- class: 2
|    |--- Shared_Cluster_id_fof > 0.50
|    |    |--- Shared_Dorm_f_fof <= 0.50
|    |    |    |--- class: 1
|    |    |--- Shared_Dorm_f_fof > 0.50
|    |    |    |--- class: 1
```

Пользователь и друг его друга не живут в одном общежитии

Пользователь и друг его друга не относятся к одной социальной группе. В этом случае рекомендация дружбы игнорируется (доминирует класс 0)

Пользователь и друг его друга проживают в одном общежитии

В этой ветви доминирует класс 2, и его прогнозирование определяется признаком Sex. Чуть позже мы этот неожиданный результат обсудим

Пользователь и друг его друга относятся к одной социальной группе. В этом случае связь между ними во FriendHook наверняка образуется (доминирует класс 1)

Как и ожидалось, прогнозы модели в первую очередь определяются принадлежностью к одному общежитию и социальной группе. Если пользователь и друг его

друга проживают в одном общежитии и относятся к одной социальной группе, то они наверняка подружатся. Если же их не объединяет ни общежитие, ни социальная группа, то вероятность возникновения между ними связи мала. Кроме того, отдельные люди могут подружиться, если относятся к одной социальной группе и обучаются по одной специализации, даже не проживая в одном общежитии.

ПРИМЕЧАНИЕ

Этому текстовому представлению дерева недостает отображения точного количества меток классов на каждой ветви. Как говорилось в главе 22, эти количества можно вывести через вызов plot_tree(clf_depth3, feature_names=list(feature_names)). Для краткости я не буду генерировать этот график дерева, но вам это сделать рекомендую. В визуализированной статистике дерева вы увидите, что пользователь и FoF в 1635 случаях относятся к одному кластеру и общежитию; 93 % этих случаев представляют метки класса 1. Кроме того, вы увидите, что в 356 случаях пользователь и FoF не относятся ни к одному общежитию, ни к одному кластеру; 97 % этих случаев соответствуют метке класса 0. Выходит, что принадлежность к одной социальной группе и общежитию является сильным предиктором пользовательского поведения.

Мы почти готовы представить нашему заказчику модель, основанную на социальных группах, общежитиях и специализациях. Логика ее очень проста. Пользователи, которые принадлежат к одним социальным группам и общежитиям или осваивают одну учебную программу, с большей вероятностью установят связь. В этом нет ничего удивительного. Удивляет же нас тот факт, что признак `Sex` определяет прогнозирование метки класса 2. Напомню, что класс 2 соответствует отклонению запроса дружбы во FriendHook. Согласно нашему дереву, отклонение наиболее вероятно, если:

- пользователи проживают в одном общежитии, но не относятся к одной социальной группе;
- отправитель запроса является представителем определенного пола.

Естественно, нам известно, что метки класса 2 в наших данных довольно редки и встречаются всего в 1,2 % случаев. Возможно, такие прогнозы модели обусловлены случайным шумом, вызванным столь редкой выборкой. Это можно выяснить. Давайте проверим, насколько хорошо наша модель прогнозирует отклонение запросов. Для этого выполним `evaluate` для `(X, y_reject)`, где `y_reject[i]` равно `2`, если `y[i]` равно `2`, и является `0` в противном случае (листинг 23.43). Иными словами, оценим модель, прогнозирующую только отклонение запроса. Если ее F-мера окажется низкой, значит, наши прогнозы в основном определяются случайным шумом.

Листинг 23.43. Оценка классификатора отклонений заявок

```
y_reject = y *(y == 2)
f_measure, clf_reject = evaluate(X, y_reject, max_depth=5)
print(f"The f-measure is {f_measure:0.2f}")

The f-measure is 0.97
```

Вот это да! F-мера оказалась очень высока. То есть, несмотря на разреженность данных, мы можем прогнозировать отклонение очень хорошо. Какие же признаки определяют отклонение заявок? Это мы выясним с помощью вывода новых показателей важности признаков (листинг 23.44).

Листинг 23.44. Ранжирование признаков по показателю их важности

```
view_top_features(clf_reject, feature_names)

Profile_ID_Sex_Value: 0.40
Profile_ID_Relationship_Status_Value: 0.24
Shared_Major_id_fof: 0.21
Shared_Cluster_id_fof: 0.10
Shared_Dorm_id_fof: 0.05
```

Очень интересно! Отклонение заявок в основном обусловливается атрибутами `Sex` и `Relationship_Status` пользователя. Давайте визуализируем обученное дерево, чтобы получше в этом разобраться (листинг 23.45).

Листинг 23.45. Отображение дерева, прогнозирующего отклонение заявок

```
text_tree = export_text(clf_reject,
                        feature_names=list(feature_names))
print(text_tree)

|--- Shared_Cluster_id_fof <= 0.50
|    |--- Shared_Major_id_fof <= 0.50
|    |    |--- Shared_Dorm_id_fof <= 0.50
|    |    |    |--- class: 0
|    |    |--- Shared_Dorm_id_fof > 0.50
|    |    |    |--- Profile_ID_Relationship_Status_Value <= 2.50
|    |    |    |    |--- class: 0
|    |    |    |--- Profile_ID_Relationship_Status_Value > 2.50
|    |    |    |    |--- Profile_ID_Sex_Value <= 0.50
|    |    |    |    |--- class: 0
|    |    |    |    |--- Profile_ID_Sex_Value > 0.50
|    |    |    |    |    |--- class: 2
|    |--- Shared_Major_id_fof > 0.50
|    |    |--- Profile_ID_Sex_Value <= 0.50
|    |    |    |--- class: 0
|    |    |--- Profile_ID_Sex_Value > 0.50
|    |    |    |--- Profile_ID_Relationship_Status_Value <= 2.50
|    |    |    |    |--- class: 0
|    |    |    |--- Profile_ID_Relationship_Status_Value > 2.50
|    |    |    |    |--- class: 2
|--- Shared_Cluster_id_fof > 0.50
|    |--- class: 0
```

Пользователь и друг его друга не относятся к одной социальной группе

Семейное положение пользователя соответствует 3. Если пол пользователя соответствует 1, высока вероятность отклонения заявки (класс 2). Значит, отклонение зависит от пола пользователя и его семейного статуса

Пользователь и друг его друга относятся к одной социальной группе. В этом случае отклонение маловероятно

Согласно дереву, отказ дружить обычно вызван следующими обстоятельствами:

- пользователи не относятся к одной социальной группе;
- пользователи либо проживают в одном общежитии, либо учатся по одной специализации;

- пол отправителя заявки на дружбу относится к категории 1;
- семейное положение отправителя относится к категории 3. Дерево показывает, что категория семейного положения должна быть больше 2,5, тем не менее максимальное значение `df_Profile.Relationship_Status` равно `3`.

По сути, люди, чья категория `Sex` соответствует 1, а категория `Relationship Status` — 3, отправляют запросы на дружбу пользователям, находящимся вне их социальной группы. Эти запросы, скорее всего, будут отклонены. Естественно, мы не можем точно определить категории, обусловливающие отказ, но, как ученые, можем на эту тему порассуждать. Учитывая наше знание человеческой природы, будет неудивительно, если этот паттерн поведения определяется одинокими мужчинами. Возможно, они стараются связаться с женщинами вне своей социальной группы, чтобы назначать свидания. Если это так, то их запросы наверняка будут отклонены. Все это лишь догадки, но данная гипотеза достойна обсуждения с продакт-менеджерами FriendHook. Если она окажется верна, то в социальную сеть следует внести определенные коррективы для предотвращения нежелательных запросов дружбы с целью назначения свидания. В качестве альтернативы в социальную сеть можно внести и другие изменения, которые упростят для одиноких людей поиск спутников.

23.5.1. Почему столь важна обобщаемость модели

В данном практическом задании мы переживали за сохранение обобщаемости нашей модели. Модель, которая не может работать вне обучающей выборки, бесполезна, даже если она демонстрирует высокую точность. К сожалению, сложно понять, будет ли модель хорошо обобщаться, если ее не протестировать на внешних данных. Однако можно попробовать сохранять бдительность в отношении неявных смещений в данных, которые не позволят модели успешно обобщаться на другие наборы данных. Если этого не будет, можно столкнуться с серьезными последствиями. Приведу для примера такую историю.

В течение многих лет исследователи в сфере машинного обучения пытались автоматизировать область радиологии. В ее рамках опытные доктора оценивают медицинские снимки (например, рентгеновские) для постановки диагноза заболевания. Это можно рассматривать как задачу обучения с учителем, в которой снимки являются признаками, а диагнозы — метками классов. К 2016 году в научной литературе было опубликовано множество моделей для этой сферы. Каждая такая модель согласно ее внутренней оценке должна была демонстрировать высокую точность. В тот год ведущие исследователи в области машинного обучения публично заявили: «Нам следует прекратить обучать радиологов» и «Радиологам пора обеспокоиться сменой работы». Через четыре года такие прогнозы привели к недостатку специалистов по всему миру: студенты медицинских вузов не хотели посвящать свою жизнь области, которую ждала полная автоматизация. Однако к 2020 году обещанной

автоматизации не произошло. Большинство опубликованных моделей очень плохо справлялись с новыми данными. Почему? Как оказалось, результаты съемки от больницы к больнице различаются. Разные медицинские учреждения используют разные условия освещения и разные настройки оборудования. В результате модель, обученная в больнице A, не могла работать с данными из больницы B. Несмотря на, казалось бы, высокий показатель точности, модели не подошли для всеобщего применения. Исследователи оказались слишком оптимистичны, не учтя характерную для их данных вариативность. Эта оплошность непреднамеренно привела к кризису в медицинском сообществе. Более продуманная оценка способности моделей к обобщению могла бы предотвратить такой результат.

РЕЗЮМЕ

- Алгоритмы машинного обучения, превосходящие все прочие, не обязательно будут хорошо работать в любой ситуации. Наша модель дерева решений превзошла модель случайного леса, хотя в литературе случайные леса преподносятся как более эффективные. Никогда не нужно слепо предполагать, что модель непременно хорошо справится в каждом возможном сценарии. Напротив, следует разумно корректировать ее выбор, исходя из специфики поставленной задачи.
- Удачный подбор признаков является скорее творческим процессом, нежели научным. Мы не можем всегда заранее знать, какие признаки повысят эффективность модели. Однако разумное внедрение в модель разнообразных и интересных признаков в конечном итоге должно улучшить качество ее прогнозов.
- Необходимо внимательно относиться к признакам, внедряемым в модель. В противном случае она может не обобщаться должным образом на другие наборы данных.
- Подобающая оптимизация гиперпараметров иногда способна значительно повысить эффективность модели.
- Временами кажется, что ни один подход не работает и наших данных просто недостаточно. Тем не менее благодаря упорству и стараниям нам все же удается выявить значительные ресурсы. Помните: хороший ученый, работающий с данными, никогда не сдается, пока не исчерпает все возможности для анализа.

Леонард Апельцин

Data Science в действии

Перевел с английского Д. Брайт

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*Л. Киселева*
Литературный редактор	*Н. Рощина*
Художественный редактор	*В. Мостипан*
Корректоры	*Т. Никифорова, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 23.06.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 59,340. Тираж 700. Заказ 0000.

Роман Зыков

РОМАН С DATA SCIENCE. КАК МОНЕТИЗИРОВАТЬ БОЛЬШИЕ ДАННЫЕ

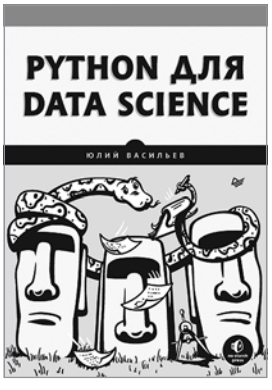
Как выжать все из своих данных? Как принимать решения на основе данных? Как организовать анализ данных (data science) внутри компании? Кого нанять аналитиком? Как довести проекты машинного обучения (machine learning) и искусственного интеллекта до топового уровня? На эти и многие другие вопросы Роман Зыков знает ответ, потому что занимается анализом данных почти двадцать лет. В послужном списке Романа — создание с нуля собственной компании с офисами в Европе и Южной Америке, ставшей лидером по применению искусственного интеллекта (AI) на российском рынке. Кроме того, автор книги создал с нуля аналитику в Ozon.ru.

Эта книга предназначена для думающих читателей, которые хотят попробовать свои силы в области анализа данных и создавать сервисы на их основе. Она будет вам полезна, если вы менеджер, который хочет ставить задачи аналитике и управлять ею. Если вы инвестор, с ней вам будет легче понять потенциал стартапа. Те, кто «пилит» свой стартап, найдут здесь рекомендации, как выбрать подходящие технологии и набрать команду. А начинающим специалистам книга поможет расширить кругозор и начать применять практики, о которых они раньше не задумывались, и это выделит их среди профессионалов в такой непростой и изменчивой области.

КУПИТЬ

Юлий Васильев

PYTHON ДЛЯ DATA SCIENCE



Python — идеальный выбор для манипулирования и извлечения информации из данных всех видов. «Python для data science» познакомит программистов с питоническим миром анализа данных. Вы научитесь писать код на Python, применяя самые современные методы, для получения, преобразования и анализа данных в управлении бизнесом, маркетинге и поддержки принятия решений.

Познакомьтесь с богатым набором встроенных структур данных Python для выполнения основных операций, а также для надежной экосистемы библиотек с открытым исходным кодом для data science, включая NumPy, pandas, scikit-learn, matplotlib и другие. Научитесь загружать данные в различных форматах, упорядочивать, группировать и агрегировать датасеты, а также создавать графики, карты и другие визуализации. На подробных примерах стройте реальные приложения, в том числе службу такси, использующую геолокацию, анализ корзины для определения товаров, которые обычно покупаются вместе, а также модель машинного обучения для прогнозирования цен на акции.