40 алгоритмов, которые должен знать каждый программист на Python

Имран Ахмад

ПИТЕР®

Packt›

40 Algorithms Every Programmer Should Know

Hone your problem-solving skills by learning different algorithms and their implementation in Python

Imran Ahmad

Packt

BIRMINGHAM - MUMBAI

40 алгоритмов, которые должен знать каждый программист на Python

Имран Ахмад

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018.1
УДК 004.421+004.43
А95

Ахмад Имран

А95 40 алгоритмов, которые должен знать каждый программист на Python. — СПб.: Питер, 2023. — 368 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1908-0

Понимание работы алгоритмов и умение применять их для решения прикладных задач — must-have для любого программиста или разработчика. Эта книга поможет вам не только развить навыки использования алгоритмов, но и разобраться в принципах их функционирования, в их логике и математике.

Вы начнете с введения в алгоритмы, от поиска и сортировки перейдете к линейному программированию, ранжированию страниц и графам и даже поработаете с алгоритмами машинного обучения. Теории не бывает без практики, поэтому вы займетесь прогнозами погоды, кластеризацией твитов, механизмами рекомендаций фильмов. И, наконец, освоите параллельную обработку, что даст вам возможность решать задачи, требующие большого объема вычислений.

Дойдя до конца, вы превратитесь в эксперта по решению реальных вычислительных задач с применением широкого спектра разнообразных алгоритмов.

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

ББК 32.973.2-018.1
УДК 004.421+004.43

ISBN 978-1789801217 англ.

ISBN 978-5-4461-1908-0

Краткое содержание

ЧАСТЬ III
Расширенные возможности

Оглавление

ЧАСТЬ II
Алгоритмы машинного обучения

ЧАСТЬ III
Расширенные возможности

Моему отцу, Иньятулла Хану,
который продолжает мотивировать меня
на непрерывное изучение чего-то нового
и неизведанного

Об авторе

Имран Ахмад — сертифицированный инструктор Google с многолетним опытом. Преподает такие дисциплины, как программирование на языке Python, машинное обучение (МО), алгоритмы, большие данные (big data) и глубокое обучение. В своей диссертации он разработал новый алгоритм на основе линейного программирования под названием ASTRA. Этот алгоритм применяется для оптимального распределения ресурсов в облачных вычислениях. На протяжении последних четырех лет Имран работает над социально значимым проектом машинного обучения в аналитической лаборатории при Федеральном правительстве Канады. Проект связан с автоматизацией иммиграционных процессов. Имран разрабатывает алгоритмы оптимального использования GPU для обучения сложных моделей МО.

Предисловие

Алгоритмы всегда играли важную роль как в теории, так и в практическом применении вычислительной техники. Эта книга посвящена использованию алгоритмов для решения прикладных задач. Чтобы извлечь максимальную пользу из применения алгоритмов, необходимо глубокое понимание их логики и математики. Мы начнем с введения в алгоритмы и изучим различные методы их проектирования. Двигаясь дальше, мы узнаем о линейном программировании, ранжировании страниц и графах, а также поработаем с алгоритмами машинного обучения, пытаясь понять математику и логику, лежащие в их основе. Книга содержит практические примеры, такие как прогноз погоды, кластеризация твитов или рекомендательные механизмы для просмотра фильмов, которые покажут, как наилучшим образом применять алгоритмы. Прочтя эту книгу, вы станете увереннее использовать алгоритмы для решения реальных вычислительных задач.

ДЛЯ КОГО ЭТА КНИГА

Эта книга для серьезного программиста! Она подойдет вам, если вы опытный программист и хотите получить более глубокое представление о математических основах алгоритмов. Если вы имеете ограниченные знания в области программирования или обработки данных и хотите узнать больше о том, как пользоваться проверенными в деле алгоритмами для совершенствования методов проектирования и написания кода, то эта книга также будет вам полезна. Опыт программирования на Python вам точно понадобится, а вот знания в области анализа и обработки данных полезны, но не обязательны.

О ЧЕМ ЭТА КНИГА

В *главе 1 «Обзор алгоритмов»* кратко излагаются основы алгоритмов. Мы начнем с раздела, посвященного основным концепциям, необходимым для понимания работы алгоритмов. В нем кратко рассказывается о том, как люди начали использовать алгоритмы для математической формулировки определенных классов задач. Речь также пойдет об ограничениях алгоритмов. В следующем разделе будут объяснены различные способы построения логики алгоритма. Поскольку в этой книге для написания алгоритмов используется Python, мы узнаем, как настроить среду для запуска примеров кода. Затем мы обсудим способы количественной оценки производительности алгоритма и ее сравнения с другими алгоритмами. В конце главы нас ждут различные способы проверки работы алгоритма.

Глава 2 «Структуры данных, используемые в алгоритмах» посвящена необходимым структурам, которые содержат временные данные. Алгоритмы могут быть требовательными с точки зрения данных, вычислений или же и того и другого одновременно. Но для оптимальной реализации любого алгоритма ключевое значение имеет выбор подходящей структуры данных. Многие алгоритмы имеют рекурсивную и итеративную логику и требуют специализированных структур данных (которые сами являются итеративными). Поскольку для примеров мы используем Python, эта глава посвящена структурам данных Python, подходящим для реализации обсуждаемых в книге алгоритмов.

В *главе 3 «Алгоритмы сортировки и поиска»* представлены основные алгоритмы, используемые для сортировки и поиска. Впоследствии они станут основой для изучения более сложных алгоритмов. Мы познакомимся с различными типами алгоритмов сортировки и сравним эффективность разных подходов. Затем рассмотрим алгоритмы поиска, сравним их между собой и количественно оценим их производительность и сложность. Наконец, обсудим области применения этих алгоритмов.

В *главе 4 «Разработка алгоритмов»* раскрываются основные концепции проектирования алгоритмов. В ней мы рассмотрим различные типы алгоритмов и обсудим их сильные и слабые стороны. Понимание этих концепций важно, когда речь заходит о разработке сложных алгоритмов. Глава начинается с обсуждения различных типов алгоритмических конструкций. Затем представлено решение знаменитой задачи коммивояжера. Далее мы обсудим линейное программирование и его ограничения. Наконец, разберем практический пример, демонстрирующий использование линейного программирования для планирования производственных мощностей.

Глава 5 «Графовые алгоритмы» посвящена алгоритмам решения графовых задач, которые очень распространены в информатике. Существует множество вычислительных задач, которые лучше всего представить в виде графов. Мы познакомимся с методами такого представления и поиска (обхода) на графах. Обход графа означает систематическое прохождение по ребрам графа с целью посещения всех его вершин. Алгоритм поиска может многое рассказать о структуре графа. Многие алгоритмы начинают с обхода входного графа, чтобы получить информацию о его структуре. Несколько других графовых алгоритмов осуществляют более тщательный поиск. Методы поиска лежат в основе графовых алгоритмов. Мы обсудим два наиболее распространенных вычислительных представления графов: в виде списка смежности и в виде матрицы смежности. Далее мы рассмотрим алгоритмы поиска в ширину и поиска в глубину и выясним, в каком порядке они посещают вершины.

В *главе 6 «Алгоритмы машинного обучения без учителя»* мы узнаем об алгоритмах, которые пытаются изучить внутренние структуры, закономерности и взаимосвязи на основе предоставленных данных без какого-либо контроля. Сначала мы обсудим методы кластеризации. Это методы машинного обучения, которые ищут сходства и взаимосвязи между элементами данных в наборе, а затем объединяют эти элементы в различные группы (кластеры). Данные в каждом кластере имеют некоторое сходство, основанное на присущих им атрибутах или признаках. В следующем разделе нас ждут алгоритмы снижения размерности, которые используются, когда имеется целый ряд признаков. Далее следуют алгоритмы, имеющие дело с обнаружением выбросов (аномалий). Наконец, в главе представлен поиск ассоциативных правил. Это метод анализа, используемый для изучения больших наборов транзакционных данных с целью выявления определенных закономерностей и правил. Эти закономерности отражают интересные взаимосвязи и ассоциации между различными элементами в транзакциях.

Глава 7 «Традиционные алгоритмы обучения с учителем» посвящена алгоритмам, используемым для решения задач с помощью размеченного набора данных с входными признаками (и соответствующими выходными метками или классами). Эти данные затем используются для обучения модели. С ее помощью прогнозируются результаты для ранее неизвестных точек данных. Сначала мы познакомимся с понятием классификации. Затем изучим простейший алгоритм машинного обучения — линейную регрессию — и один из самых значимых алгоритмов — дерево решений. Мы обсудим его слабые и сильные стороны, после чего разберем два важных алгоритма, SVM и XGBoost.

Глава 8 «Алгоритмы нейронных сетей» познакомит нас с основными понятиями и компонентами нейронной сети, которая становится все более важным

методом машинного обучения. Мы изучим типы нейронных сетей и различные функции активации, используемые для их реализации. Мы подробно обсудим алгоритм обратного распространения ошибки. Он широко применяется для решения проблемы сходимости нейронной сети. Далее познакомимся с переносом обучения, необходимым для значительного упрощения и частичной автоматизации обучения моделей. Наконец, в качестве практического примера мы узнаем, как использовать глубокое обучение для выявления фальшивых документов.

В *главе 9 «Алгоритмы обработки естественного языка»* представлены алгоритмы NLP. Эта глава построена как переход от теории к практике. В ней рассматриваются главные принципы и математические основы. Мы обсудим широко используемую нейронную сеть для разработки и реализации важных практических задач в области текстовых данных. Также мы разберем ограничения NLP. Наконец, рассмотрим практический пример, в котором модель обучается распознавать эмоциональную окраску отзывов на кинофильмы.

Глава 10 «Рекомендательные системы» посвящена механизмам рекомендаций. Это способ моделирования доступной информации о пользовательских предпочтениях и использования этой информации для предоставления обоснованных рекомендаций. Основой рекомендательной системы всегда является записанное взаимодействие между пользователем и продуктом. В этой главе объясняется основная идея рекомендательных систем и представлены различные типы механизмов рекомендаций. Наконец, демонстрируется пример использования рекомендательной системы для предложения продуктов различным пользователям.

Глава 11 «Алгоритмы обработки данных» посвящена алгоритмам, ориентированным на данные. Глава начинается с краткого обзора таких алгоритмов. Затем рассматриваются критерии классификации данных и описывается применение алгоритмов к приложениям потоковой передачи. Наконец, вашему вниманию будет предложен практический пример извлечения закономерностей из данных Twitter.

Глава 12 «Криптография» познакомит с криптографическими алгоритмами. Глава начинается с истории возникновения криптографии. Затем обсуждаются алгоритмы симметричного шифрования, алгоритмы хеширования MD5 и SHA, а также ограничения и недостатки, связанные с реализацией симметричных алгоритмов. Далее представлены алгоритмы асимметричного шифрования и их использования для создания цифровых сертификатов. В конце нас ждет практический пример, обобщающий все эти методы.

Глава 13 «Крупномасштабные алгоритмы» объяснит, как алгоритмы обрабатывают данные, которые не могут поместиться в память одной ноды и требуют обработки несколькими процессорами. Мы обсудим типы алгоритмов, для которых необходимо параллельное выполнение, а также процесс распараллеливания алгоритмов. Далее будет представлена архитектура CUDA. Мы узнаем, как использовать один или несколько графических процессоров для ускорения алгоритма и как его изменить, чтобы эффективно использовать мощность GPU. Наконец, мы рассмотрим кластерные вычисления и выясним, как Apache Spark создает коллекции данных RDD для чрезвычайно быстрой параллельной реализации стандартных алгоритмов.

Глава 14 «Практические рекомендации» начинается с темы объяснимости алгоритма, которая становится все более важной теперь, когда мы понимаем логику автоматизированного принятия решений. Затем раскрываются проблема этики алгоритмов и возможность возникновения предвзятости при их реализации. Далее подробно обсуждаются методы решения NP-трудных задач. Наконец, кратко изложены способы реализации алгоритмов и связанные с этим сложности.

ЧТО ВАМ ПОТРЕБУЕТСЯ ПРИ ЧТЕНИИ ЭТОЙ КНИГИ

- Требуемое программное обеспечение: Python 3.7.2 или более поздней версии.
- Технические характеристики оборудования: минимум 4 Гбайта оперативной памяти, 8 Гбайт и более (рекомендуется).
- Операционная система: Windows/Linux/Mac.

Вы можете загрузить файлы с примерами кода из репозитория GitHub по адресу https://github.com/PacktPublishing/40-Algorithms-Every-Programmer-Should-Know.

УСЛОВНЫЕ ОБОЗНАЧЕНИЯ

В этой книге используется ряд условных обозначений.

`Код в тексте`

Указывает кодовые слова в тексте, имена таблиц базы данных, имена файлов, расширения файлов, пути, URL-адреса, вводимые пользователем, и аккаунты пользователя в Twitter. Вот пример: «Давайте посмотрим, как добавить новый элемент в стек с помощью `push` или удалить элемент из стека с помощью `pop`».

Блоки кода (листинги) выглядят следующим образом:

```
define swap(x, y)
    buffer = x
    x = y
    y = buffer
```

Когда мы хотим привлечь ваше внимание к определенной части листинга, соответствующие строки или элементы выделены полужирным шрифтом:

```
define swap(x, y)
    buffer = x
    x = y
    y = buffer
```

Любой ввод или вывод командной строки также выделяется полужирным шрифтом:

```
pip install a_package
```

Новые термины, важные понятия или слова выделены *курсивом*. Вот пример: «Один из способов уменьшить сложность алгоритма — это пойти на компромисс в отношении его точности, создав тип алгоритма, называемый *приближенным алгоритмом*».

Так оформлены предупреждения или важные примечания.

Так оформлены советы и рекомендации.

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.

Часть I

ОСНОВЫ И БАЗОВЫЕ АЛГОРИТМЫ

В этой части книги мы разберемся, что такое алгоритм, как его создать, какие структуры данных используются в алгоритмах. Кроме того, мы подробно изучим алгоритмы сортировки и поиска, а также применение алгоритмов для решения графических задач. Главы, из которых состоит часть I:

- Глава 1. Обзор алгоритмов.
- Глава 2. Структуры данных, используемые в алгоритмах.
- Глава 3. Алгоритмы сортировки и поиска.
- Глава 4. Разработка алгоритмов.
- Глава 5. Графовые алгоритмы.

1

Обзор алгоритмов

Эта книга содержит информацию, необходимую для понимания, классификации, выбора и реализации важных алгоритмов. Помимо объяснения логики алгоритмов, в ней рассматриваются структуры данных, среды разработки и производственные среды, подходящие для тех или иных классов алгоритмов. Мы сфокусируемся на современных алгоритмах машинного обучения, которые обретают все большее значение. Наряду с теорией мы рассмотрим и практические примеры использования алгоритмов для решения актуальных повседневных задач.

В первой главе дается представление об основах алгоритмов. Она начинается с раздела, посвященного основным концепциям, необходимым для понимания работы разных алгоритмов. В этом разделе кратко рассказывается о том, как люди начали использовать алгоритмы для математической формулировки определенного класса задач. Речь также пойдет об ограничениях различных алгоритмов. В следующем разделе будут объяснены способы построения логики алгоритма. Поскольку в этой книге для написания алгоритмов используется Python, мы объясним, как настроить среду для запуска примеров. Затем обсудим различные способы количественной оценки работы алгоритма и сравнения с другими алгоритмами. В конце главы познакомимся с различными способами проверки конкретной реализации алгоритма.

Если кратко, то в этой главе рассматриваются следующие основные моменты:

- Что такое алгоритм.
- Определение логики алгоритма.

- Введение в пакеты Python.
- Методы разработки алгоритмов.
- Анализ производительности.
- Проверка алгоритма.

ЧТО ТАКОЕ АЛГОРИТМ

Говоря простым языком, алгоритм — это набор правил для выполнения определенных вычислений, направленных на решение определенной задачи. Он предназначен для получения результатов при использовании любых допустимых входных данных в соответствии с точно прописанными командами. Словарь American Heritage дает такое определение:

> *«Алгоритм — это конечный набор однозначных инструкций, которые при заданном наборе начальных условий могут выполняться в заданной последовательности для достижения определенной цели и имеют определимый набор конечных условий».*

Разработка алгоритма — это создание набора математических правил для эффективного решения реальной практической задачи. На основе такого набора правил можно создать более общее математическое решение, многократно применимое к широкому спектру аналогичных задач.

Этапы алгоритма

Различные этапы разработки, развертывания и, наконец, использования алгоритма проиллюстрированы на следующей диаграмме (рис. 1.1).

Как показано на диаграмме, прежде всего нужно сформулировать задачу и подробно описать необходимый результат. Как только задача четко поставлена, мы подходим к этапу разработки.

Разработка состоит из двух этапов.

- *Проектирование*. На этом этапе разрабатываются и документируются архитектура, логика и детали реализации алгоритма. При разработке алгоритма мы учитываем как точность, так и производительность. Часто для решения одной и той же задачи можно использовать два или несколько разных алгоритмов. Этап проектирования — это итеративный процесс,

который включает в себя сравнение различных потенциально пригодных алгоритмов. Некоторые алгоритмы могут давать простые и быстрые решения, но делают это в ущерб точности. Другие могут быть очень точными, но их выполнение занимает значительное время из-за их сложности. Одни сложные алгоритмы эффективнее других. Прежде чем сделать выбор, следует тщательно изучить все компромиссы. Разработка наиболее эффективного алгоритма особенно важна при решении сложной задачи. Правильно разработанный алгоритм приведет к эффективному решению, способному одновременно обеспечить как надлежащую производительность, так и достаточную степень точности.

- *Кодирование*. Разработанный алгоритм превращается в компьютерную программу. Важно, чтобы программа учитывала всю логику и архитектуру, предусмотренную на этапе проектирования.

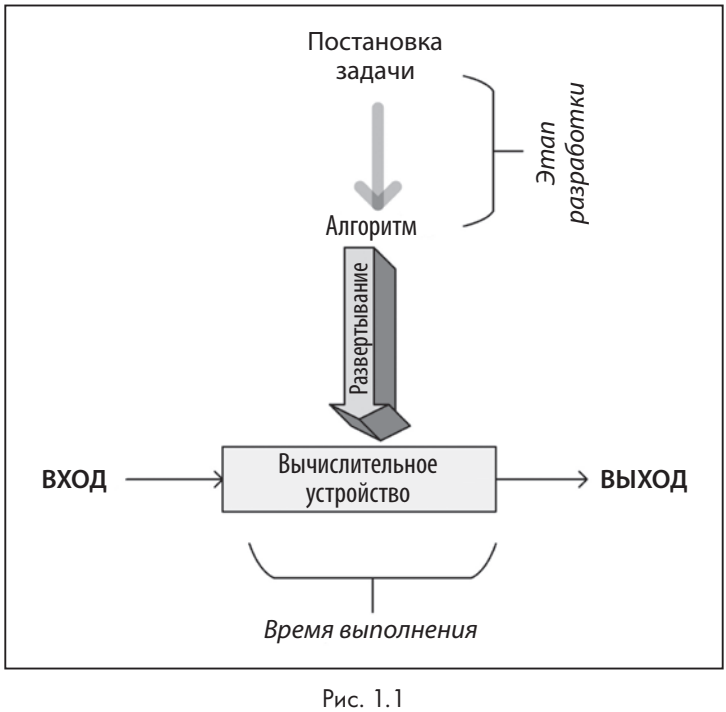


Рис. 1.1

Этапы проектирования и кодирования алгоритма носят итеративный характер. Разработка алгоритма, отвечающего как функциональным, так и нефункциональным требованиям, занимает массу времени и усилий. Функциональные

требования определяют то, какими должны быть правильные выходные данные для конкретного набора входных данных. Нефункциональные требования алгоритма в основном касаются производительности для заданного объема данных. Проверку алгоритма и анализ его производительности мы обсудим далее в этой главе. Проверка должна подтвердить, соответствует ли алгоритм его функциональным требованиям. Анализ производительности — подтвердить, соответствует ли алгоритм своему основному нефункциональному требованию: производительности.

После разработки и реализации на выбранном языке программирования код алгоритма готов к развертыванию. Развертывание алгоритма включает в себя разработку реальной производственной среды, в которой будет выполняться код. Производственная среда должна быть спроектирована в соответствии с потребностями алгоритма в данных и обработке. Например, для эффективной работы распараллеливаемых алгоритмов потребуется кластер с соответствующим количеством компьютерных узлов. Для алгоритмов, требующих больших объемов данных, возможно, потребуется разработать конвейер ввода данных и стратегию их кэширования и хранения. Более подробно проектирование производственной среды обсуждается в главах 13 и 14. После разработки и реализации производственной среды происходит развертывание алгоритма, который принимает входные данные, обрабатывает их и генерирует выходные данные в соответствии с требованиями.

ОПРЕДЕЛЕНИЕ ЛОГИКИ АЛГОРИТМА

При разработке алгоритма важно найти различные способы его детализировать. Нам необходимо разработать как его логику, так и архитектуру. Этот процесс можно сравнить с постройкой дома: нужно спроектировать структуру, прежде чем фактически начинать реализацию. Для эффективности итеративного процесса проектирования более сложных распределенных алгоритмов важно предварительно планировать, как их логика будет распределена по нодам кластера во время выполнения алгоритма. Этого можно добиться с помощью псевдокода и планов выполнения, о которых мы поговорим в следующем разделе.

Псевдокод

Самый простой способ задать логику алгоритма — составить высокоуровневое неформальное описание алгоритма, которое называется *псевдокодом*. Прежде

чем описывать логику в псевдокоде, полезно сформулировать основные шаги на простом человеческом языке. Затем это словесное описание структурируется и преобразуется в псевдокод, точно отражающий логику и последовательность алгоритма. Хорошо написанный псевдокод должен достаточно подробно описывать алгоритм, даже если такая детализация не обязательна для описания основного хода работы и структуры алгоритма. На рис. 1.2 показана последовательность шагов.

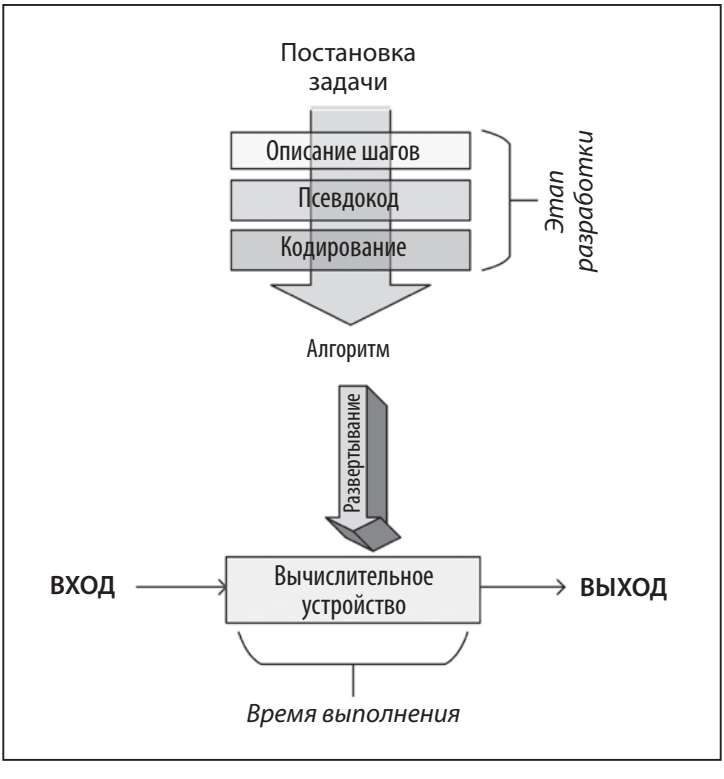


Рис. 1.2

Как только псевдокод готов (см. следующий раздел), мы можем написать код алгоритма, используя выбранный язык программирования.

Практический пример псевдокода

Ниже представлен псевдокод алгоритма распределения ресурсов, называемого *SRPMP*. В кластерных вычислениях возможно множество ситуаций, когда па-

раллельные задачи необходимо выполнить на наборе доступных ресурсов, в совокупности называемых *пулом ресурсов*. Данный алгоритм назначает задачи ресурсу и создает маппинг-сет (mapping set), называемый Ω (омега). Представленный псевдокод отражает логику и последовательность алгоритма, что более подробно объясняется в следующем разделе.

```
1: BEGIN Mapping_Phase
2: Ω = { }
3: k = 1
4: FOREACH Ti∈T
5:     ωi = A(Δk,Ti)
6:     add {ωi,Ti} to Ω
7:     state_changeTi [STATE 0: Idle/Unmapped] → [STATE 1: Idle/Mapped]
8:     k=k+1
9:     IF (k>q)
10:       k=1
11:    ENDIF
12: END FOREACH
13: END Mapping_Phase
```

Давайте построчно разберем этот алгоритм.

1. Мы начинаем маппинг с выполнения алгоритма. Маппинг-сет Ω пуст.
2. Первый раздел выбирается в качестве пула ресурсов для задачи T_1 (см. строку 3 кода). *Целевой телевизионный рейтинг* (Television Rating Point, TRPs) итеративно вызывает алгоритм *ревматоидного артрита* (Rheumatoid Arthritis, RA) для каждой задачи T_i с одним из разделов, выбранным в качестве пула ресурсов.
3. Алгоритм RA возвращает набор ресурсов, выбранных для задачи T_i, представленный посредством ω_i (см. строку 5 кода).
4. T_i и ω_i добавляются в маппинг-сет (см. строку 6 кода).
5. Состояние T_i меняется со `STATE 0:Idle/Mapping` на `STATE 1:Idle/Mapped` (см. строку 7 кода).
6. Обратите внимание, что для первой итерации выбран первый раздел и `k=1`. Для каждой последующей итерации значение `k` увеличивается до тех пор, пока не достигнуто `k>q`.
7. Если переменная `k` становится больше `q`, она сбрасывается в `1` (см. строки 9 и 10 кода).
8. Этот процесс повторяется до тех пор, пока каждой задаче не будет присвоен набор используемых ресурсов, что будет отражено в маппинг-сете, называемом Ω.

9. Как только задаче присваивается набор ресурсов на этапе маппинга, она запускается на выполнение.

Использование сниппетов

С ростом популярности такого простого, но мощного языка программирования, как Python, приобретает популярность и альтернативный подход, который заключается в представлении логики алгоритма непосредственно на языке программирования, но в несколько упрощенной версии. Как и псевдокод, такой фрагмент кода отражает основную логику и структуру предлагаемого алгоритма без указания подробностей. Иногда его называют *сниппетом*. В этой книге сниппеты используются вместо псевдокода везде, где это возможно, поскольку они экономят один дополнительный шаг. Например, давайте рассмотрим простой сниппет для функции Python, которая меняет местами значения двух переменных:

```
define swap(x, y)
    buffer = x
    x = y
    y = buffer
```

Обратите внимание, что сниппеты не всегда могут заменить псевдокод. В псевдокоде мы иногда резюмируем много строк кода в виде одной строки псевдокода, отражая логику алгоритма, при этом не отвлекаясь на излишние детали.

Создание плана выполнения

С помощью псевдокода или сниппетов не всегда возможно задать логику более сложных распределенных алгоритмов. Разработку таких алгоритмов можно разбить на этапы, выполняющиеся в определенном порядке. Верная стратегия разделения крупной задачи на оптимальное количество этапов с должной очередностью имеет решающее значение для эффективного исполнения алгоритма.

Нам необходимо и применить эту стратегию, и одновременно полностью отразить логику и структуру алгоритма. План выполнения — один из способов детализации того, как алгоритм будет разделен на множество задач. Задача может представлять собой сопоставления или преобразования, которые сгруппированы в блоки, называемые *этапами*. На следующей диаграмме (рис. 1.3) показан план, который создается средой выполнения Apache Spark перед исполнением

алгоритма. В нем подробно описываются задачи времени выполнения, на которые будет разделено задание, созданное для выполнения нашего алгоритма.

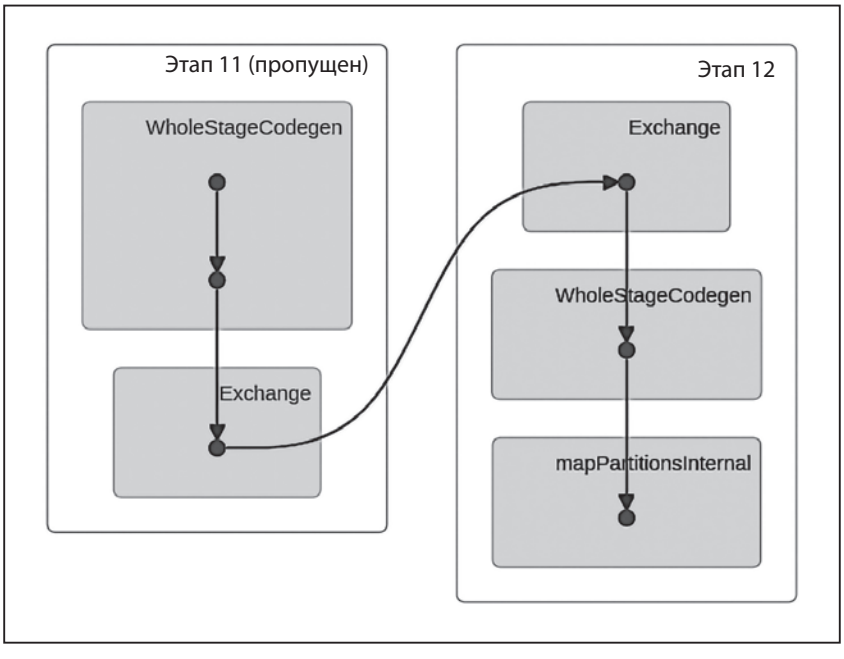


Рис. 1.3

Обратите внимание, что на данной диаграмме представлены пять задач, которые были разделены на два разных этапа: *этап 11* и *этап 12*.

ВВЕДЕНИЕ В БИБЛИОТЕКИ PYTHON

После разработки алгоритм должен быть реализован на языке программирования в соответствии с проектом. Для этой книги я выбрал Python, так как это гибкий язык программирования с открытым исходным кодом. Python также является базовым языком для инфраструктур облачных вычислений, приобретающих все более важное значение, таких как *Amazon Web Services* (AWS), *Microsoft Azure* и *Google Cloud Platform* (GCP).

Официальная домашняя страница Python доступна по адресу https://www.python.org/. Здесь вы найдете инструкции по установке и полезное руководство для начинающих.

Если вы раньше не использовали Python, рекомендуем ознакомиться с этим руководством в целях самообразования. Базовое представление о языке Python поможет лучше понять концепции, представленные в этой книге.

Установите последнюю версию Python 3. На момент написания это версия 3.7.3, которую мы и будем использовать для выполнения упражнений в книге.

Библиотеки Python

Python — это язык общего назначения. Он разработан таким образом, чтобы обеспечить минимальную функциональность. В зависимости от цели использования Python вам придется установить дополнительные библиотеки. Самый простой способ — программа установки pip. Следующая команда `pip` устанавливает дополнительную библиотеку:

```
pip install a_package
```

Установленные ранее библиотеки необходимо периодически обновлять, чтобы иметь возможность использовать новейшие функциональные возможности. Обновление запускается с помощью флага `upgrade`:

```
pip install a_package --upgrade
```

Другим дистрибутивом Python для научных вычислений является Anaconda. Его можно загрузить по адресу https://www.anaconda.com.

В дистрибутиве Anaconda применяется следующая команда для установки новых библиотек:

```
conda install a_package
```

Для обновления существующих библиотек дистрибутива Anaconda:

```
conda update a_package
```

Существует множество доступных библиотек Python. Некоторые важные библиотеки, имеющие отношение к алгоритмам, описаны далее.

Экосистема SciPy

Scientific Python (SciPy) — произносится как *сай пай* — это группа библиотек Python, созданных для научного сообщества. Она содержит множество функций, включая широкий спектр генераторов случайных чисел, программ линейной

алгебры и оптимизаторов. SciPy — это комплексная библиотека, и со временем специалисты разработали множество расширений для ее настройки и дополнения в соответствии со своими потребностями.

Ниже приведены основные библиотеки, которые являются частью этой экосистемы:

- *NumPy.* При работе с алгоритмами требуется создание многомерных структур данных, таких как массивы и матрицы. NumPy предлагает набор типов данных для массивов и матриц, которые необходимы для статистики и анализа данных. Подробную информацию о NumPy можно найти по адресу http://www.numpy.org/.
- *scikit-learn*. Это расширение для машинного обучения является одним из самых популярных расширений SciPy. Scikit-learn предоставляет широкий спектр важных алгоритмов машинного обучения, включая классификацию, регрессию, кластеризацию и проверку моделей. Вы можете найти более подробную информацию о scikit-learn по адресу http://scikit-learn.org/.
- *pandas.* Библиотека программного обеспечения с открытым исходным кодом. Она содержит сложную табличную структуру данных, которая широко используется для ввода, вывода и обработки табличных данных в различных алгоритмах. Библиотека pandas содержит множество полезных функций, а также обеспечивает высокую производительность с хорошим уровнем оптимизации. Более подробную информацию о pandas можно найти по адресу http://pandas.pydata.org/.
- *Matplotlib*. Содержит инструменты для создания впечатляющих визуализаций. Данные могут быть представлены в виде линейных графиков, точечных диаграмм, гистограмм, круговых диаграмм и т. д. Более подробную информацию можно найти по адресу https://matplotlib.org/.
- *Seaborn*. Считается аналогом популярной библиотеки ggplot2 в R. Она основана на Matplotlib и предлагает великолепный расширенный интерфейс для графического отображения статистики. Подробную информацию можно найти по адресу https://seaborn.pydata.org/.
- *IPython*. Усовершенствованная интерактивная консоль, предназначенная для того, чтобы упростить написание, тестирование и отладку кода Python.
- *Запуск Python в интерактивном режиме*. Этот режим программирования полезен для обучения и экспериментов с кодом. Программы на Python могут быть сохранены в текстовом файле с расширением `.py`, и этот файл можно запустить из консоли.

Реализация Python с помощью Jupyter Notebook

Еще один способ запуска программ на Python — через Jupyter Notebook. Jupyter Notebook представляет собой пользовательский интерфейс для разработки на основе браузера. Именно Jupyter Notebook используется для демонстрации примеров кода в этой книге. Возможность комментировать и описывать код с помощью текста и графики делает Jupyter Notebook идеальным инструментом для демонстрации и объяснения любого алгоритма, а также отличным инструментом для обучения.

Чтобы запустить программу, вам нужно запустить процесс `Jupyter-notebook`, а затем открыть свой любимый браузер и перейти по ссылке http://localhost:8888 (рис. 1.4).

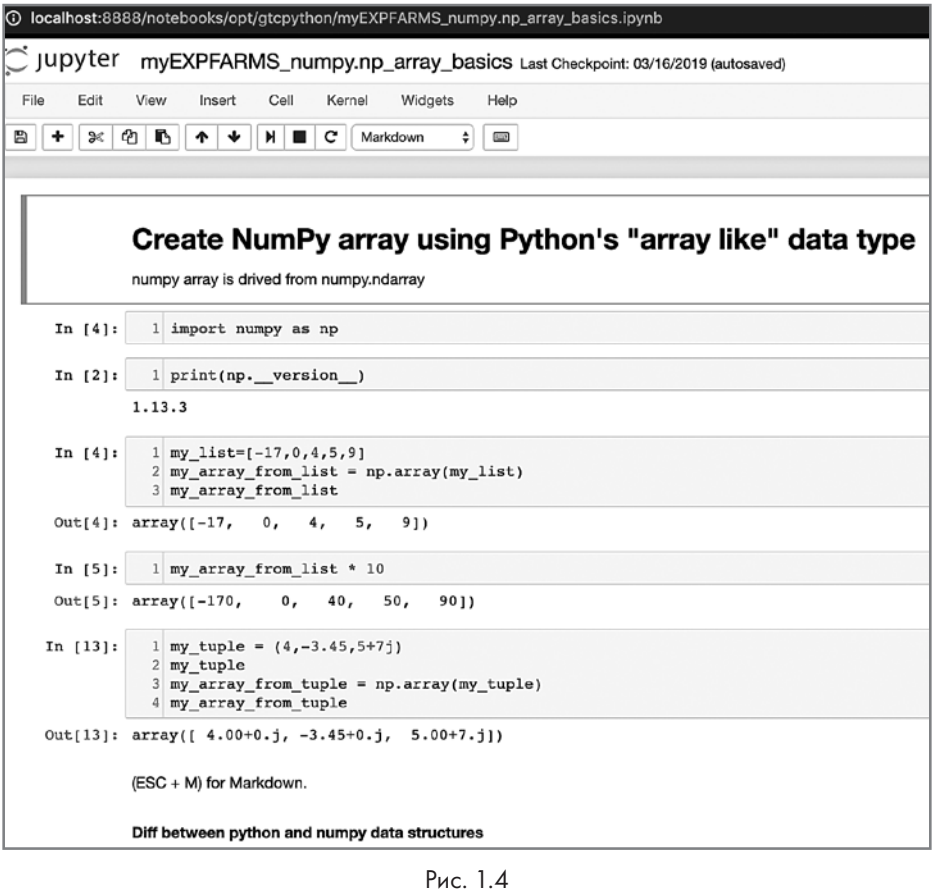


Рис. 1.4

Обратите внимание, что Jupyter Notebook состоит из ряда блоков, называемых *ячейками*.

МЕТОДЫ РАЗРАБОТКИ АЛГОРИТМОВ

Алгоритм — это математическое решение реальной проблемы. При разработке и настройке алгоритма мы должны задавать себе следующие вопросы:

- *Вопрос 1*. Даст ли алгоритм тот результат, который мы ожидаем?
- *Вопрос 2*. Является ли данный алгоритм оптимальным способом получения этого результата?
- *Вопрос 3*. Как алгоритм будет работать с большими наборами данных?

Важно оценить сложность задачи, прежде чем искать для нее решение. При разработке подходящего решения полезно охарактеризовать задачу с точки зрения ее требований и сложности. Как правило, алгоритмы можно разделить на следующие типы в зависимости от характеристик задачи:

- *Алгоритмы с интенсивным использованием данных*. Такие алгоритмы предъявляют относительно простые требования к обработке. Примером является алгоритм сжатия, применяемый к огромному файлу. В таких случаях размер данных обычно намного больше, чем объем памяти процессора (одной ноды или кластера), поэтому для эффективной обработки данных в соответствии с требованиями может потребоваться итеративный подход.
- *Вычислительноемкие алгоритмы*. Такие алгоритмы предъявляют значительные требования к обработке, но не задействуют больших объемов данных. Пример — алгоритм поиска очень большого простого числа. Чтобы добиться максимальной производительности, нужно найти способ разделить алгоритм на фазы так, чтобы хотя бы некоторые из них были распараллелены.
- *Вычислительноемкие алгоритмы с интенсивным использованием данных*. Хорошим примером здесь служат алгоритмы, используемые для анализа эмоций в видеотрансляциях. Такие алгоритмы являются наиболее ресурсоемкими алгоритмами и требуют тщательной разработки и разумного распределения доступных ресурсов.

Чтобы определить сложность и ресурсоемкость задачи, необходимо изучить параметры данных и вычислений, чем мы и займемся в следующем разделе.

Параметры данных

Чтобы классифицировать параметры данных задачи, мы рассмотрим ее *объем*, *скорость* и *разнообразие* (часто называемые «три V» — Volume, Velocity, Variety):

- *Объем* (Volume). Ожидаемый размер данных, которые будет обрабатывать алгоритм.
- *Скорость* (Velocity). Ожидаемая скорость генерации новых данных при использовании алгоритма. Она может быть равна нулю.
- *Разнообразие* (Variety). Количество различных типов данных, с которым, как ожидается, будет работать алгоритм.

На рис. 1.5 эти параметры показаны более подробно. В центре диаграммы расположены максимально простые данные с небольшим объемом, малым разнообразием и низкой скоростью. По мере удаления от центра сложность данных возрастает. Она может увеличиваться по одному или нескольким из трех параметров. К примеру, на векторе скорости располагается *пакетный* процесс как самый простой, за ним следует *периодический* процесс, а затем процесс, *близкий*

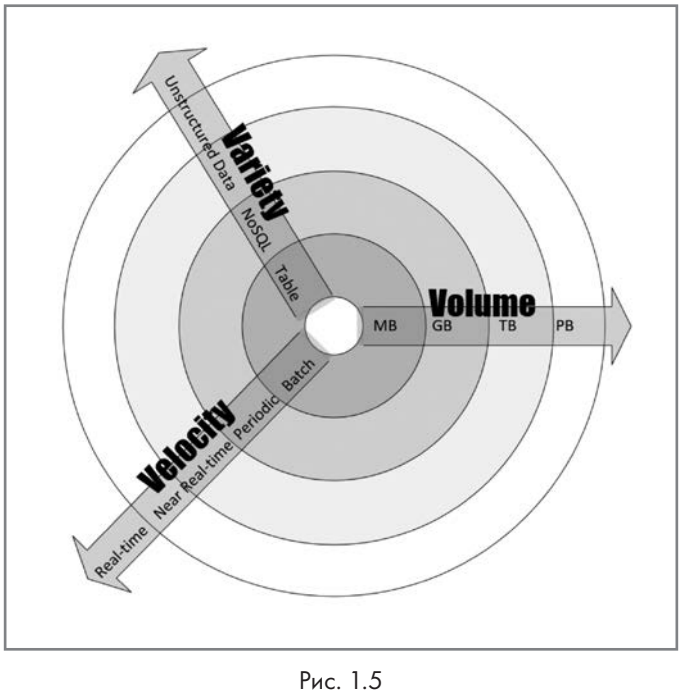


Рис. 1.5

к реальному времени. Наконец, мы видим процесс в *реальном времени*, который является наиболее сложным для обработки в контексте скорости передачи данных.

Например, если входные данные представляют собой простой `csv`-файл, то объем, скорость и разнообразие данных будут низкими. С другой стороны, если входные данные представляют собой прямую трансляцию с камеры видеонаблюдения, то объем, скорость и разнообразие данных будут довольно высокими, и эту проблему следует иметь в виду при разработке соответствующего алгоритма.

Параметры вычислений

Параметры вычислений касаются требований к обработке рассматриваемой задачи. От этих требований зависит, какой тип архитектуры лучше всего подойдет для алгоритма. Например, алгоритмы глубокого обучения, как правило, требуют большой вычислительной мощности. Это означает, что для таких алгоритмов важно иметь многоузловую параллельную архитектуру везде, где это возможно.

Практический пример

Предположим, что мы хотим провести анализ эмоциональной окраски в видеозаписи. Для этого мы должны отметить на видео человеческие эмоции: печаль, счастье, страх, радость, разочарование и восторг. Это трудоемкий процесс, требующий больших вычислительных мощностей. Как видно на рис. 1.6, для

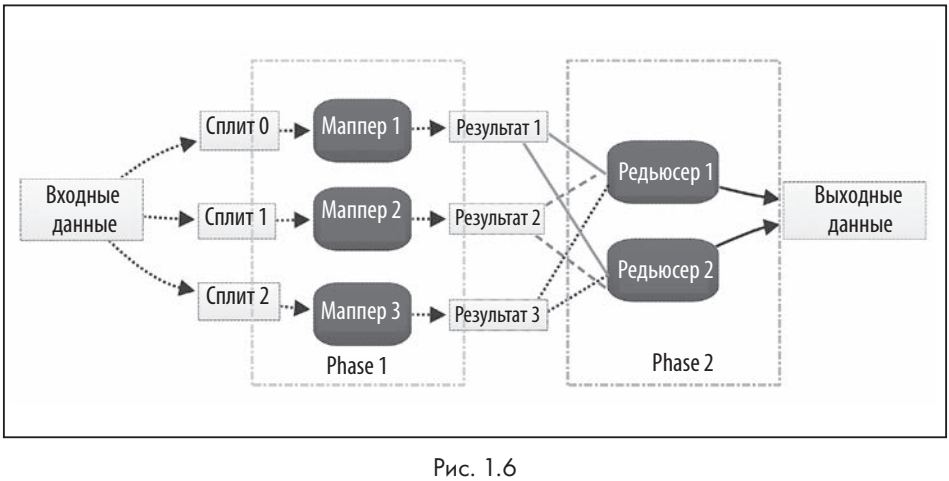


Рис. 1.6

измерения вычислений мы разделили обработку на пять задач, состоящих из двух этапов. Все преобразование и подготовка данных осуществляются в трех мапперах. Для этого мы делим видео на три части, которые называются *сплитами*. После выполнения маппинга обработанное видео попадает в два агрегатора, которые называются *редьюсерами*. Чтобы провести анализ эмоциональной окраски, редьюсеры группируют части видео в соответствии с эмоциями. Наконец, результаты объединяются в выводе данных.

Обратите внимание, что количество мапперов напрямую зависит от параллельности выполнения алгоритма. Оптимальное количество мапперов и редьюсеров зависит от характеристик данных, типа алгоритма, который необходимо использовать, и количества доступных ресурсов.

АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ

Анализ производительности алгоритма — важная часть его разработки, и одним из способов такой оценки выступает анализ сложности алгоритма.

Теория сложности — это изучение того, насколько сложны алгоритмы. Чтобы быть полезным, алгоритму необходимо обладать тремя ключевыми функциями:

- Он должен быть верным. От алгоритма мало пользы, если он не дает правильных ответов.
- Хороший алгоритм должен быть понятным для компьютера. Лучший алгоритм в мире окажется бесполезным, если его слишком сложно реализовать.
- Хороший алгоритм должен быть эффективным. Невозможно использовать алгоритм, который даст правильный результат, но при этом на его работу уйдет тысяча лет или потребуется 1 миллиард терабайт памяти.

Существуют два типа анализа для количественной оценки сложности алгоритма:

- *Анализ пространственной сложности* (space complexity analysis) — оценка требований к памяти во время выполнения алгоритма.
- *Анализ временной сложности* (time complexity analysis) — оценка времени, необходимого для выполнения алгоритма.

Анализ пространственной сложности

При анализе пространственной сложности оценивают объем памяти, необходимый алгоритму для хранения структур временных данных в процессе работы. Способ разработки алгоритма влияет на количество, тип и размер этих структур данных. В эпоху распределенных вычислений и постоянно растущих объемов данных, которые необходимо обрабатывать, анализ пространственной сложности приобретает все большее значение. Размер, тип и количество структур данных определяют требования к памяти для соответствующего оборудования. Современные структуры данных, используемые в распределенных вычислениях, такие как *устойчивые распределенные наборы данных* (Resilient Distributed Datasets, RDDs), должны иметь эффективные механизмы распределения ресурсов, учитывающие требования к памяти на различных этапах выполнения алгоритма.

Анализ пространственной сложности необходим для эффективного проектирования алгоритмов. Если при разработке алгоритма не провести надлежащий анализ, то недостаток памяти для временных структур данных может привести к ненужным перегрузкам диска. Это способно значительно повлиять на производительность и эффективность алгоритма.

В этой главе мы детально рассмотрим временну́ю сложность. Пространственная сложность более подробно будет обсуждаться в главе 13, где мы будем иметь дело с крупномасштабными распределенными алгоритмами со сложными требованиями к памяти во время выполнения.

Анализ временной сложности

Анализ временной сложности позволяет узнать, сколько времени потребуется алгоритму для выполнения задачи, исходя из его структуры. В отличие от пространственной сложности, временная сложность не зависит от оборудования, на котором будет выполняться алгоритм. Она зависит исключительно от структуры алгоритма. Основная цель анализа временной сложности — ответить на ключевые вопросы: будет ли этот алгоритм масштабироваться? Насколько хорошо алгоритм будет обрабатывать большие наборы данных?

Для этого нужно определить влияние увеличения объема данных на производительность алгоритма и убедиться, что алгоритм точен и хорошо масштабируется. Производительность алгоритма становится все более важным показателем в современном мире «больших данных».

Часто мы можем разработать несколько алгоритмов для решения одной и той же задачи. В таком случае нужно проанализировать временную сложность, чтобы ответить на следующий вопрос:

> *«С учетом обозначенной проблемы какой из нескольких алгоритмов наиболее эффективен с точки зрения экономии времени?»*

Существуют два основных подхода к вычислению временной сложности алгоритма:

- *Профилирование после реализации.* При данном подходе реализуются различные алгоритмы-кандидаты и сравнивается их производительность.
- *Теоретический подход до реализации.* При этом подходе производительность каждого алгоритма математически аппроксимируется перед запуском алгоритма.

Преимущество теоретического подхода заключается в том, что он зависит лишь от структуры самого алгоритма. Он не зависит от оборудования, на котором будет выполняться выбранный алгоритм, или от языка программирования, используемого для реализации алгоритма.

Оценка эффективности

Производительность алгоритма обычно зависит от типа входных данных. Например, если данные уже отсортированы в соответствии с контекстом задачи, алгоритм может работать невероятно быстро. Если отсортированные входные данные используются для проверки конкретного алгоритма, то он даст неоправданно высокое значение производительности. Это не будет истинным отражением настоящей производительности алгоритма в большинстве сценариев. Чтобы решить проблему зависимости алгоритмов от входных данных, мы должны учитывать различные типы сценариев при проведении анализа производительности.

Наилучший сценарий

При наилучшем сценарии входные данные организованы таким образом, чтобы алгоритм обеспечивал наилучшую производительность. Анализ наилучшего сценария дает верхнюю границу производительности.

Наихудший сценарий

Второй способ оценить производительность алгоритма — попытаться найти максимально возможное время, необходимое для выполнения задачи при заданном наборе условий. Такой анализ алгоритма весьма полезен, поскольку мы при этом гарантируем, что независимо от условий производительность алгоритма всегда будет лучше результатов нашего анализа. Анализ наихудшего сценария лучше всего подходит для оценки производительности при решении сложных проблем с большими наборами данных. Анализ наихудшего сценария дает нижнюю границу производительности алгоритма.

Средний сценарий

Этот подход начинается с разделения всех возможных входных данных на группы. Затем проводится анализ производительности на основе ввода данных от каждой группы. Далее вычисляется среднее значение производительности для каждой из групп.

Анализ среднего сценария не всегда точен, так как он должен учитывать все возможные комбинации входных данных, что не всегда легко сделать.

Выбор алгоритма

Как узнать, какой алгоритм является наилучшим решением? Как узнать, какой алгоритм сработает быстрее? *Временная сложность (time complexity)* и *«О-большое»* (обсуждаемые позже в этой главе) — хорошие инструменты для получения ответов на такие вопросы.

Рассмотрим простую задачу сортировки списка чисел. Существует несколько доступных алгоритмов, которые способны выполнить эту работу. Вопрос в том, как выбрать правильный.

Прежде всего следует заметить, что если в списке не слишком много чисел, то не имеет значения, какой алгоритм мы выберем для сортировки. Например, если в списке всего 10 чисел ($n = 10$), то какой бы алгоритм мы ни выбрали, его выполнение вряд ли займет более нескольких микросекунд, даже при очень плохой разработке. Но как только размер списка достигнет одного миллиона, выбор правильного алгоритма станет важным шагом. Плохой алгоритм может выполняться несколько часов, в то время как хороший способен завершить

сортировку списка за пару секунд. Таким образом, в случае большого объема входных данных имеет смысл приложить усилия: выполнить анализ производительности и выбрать алгоритм, который будет эффективно решать требуемую задачу.

«О-большое»

«О-большое» используется для количественной оценки производительности алгоритмов по мере увеличения размера входных данных. Это одна из самых популярных методик, используемых для проведения анализа наихудшего сценария. В этом разделе мы обсудим различные типы «О-большого».

Константная временная сложность (O(1))

Если выполнение алгоритма занимает одинаковое количество времени независимо от размера входных данных, то про него говорят, что он выполняется постоянное время. Такая сложность обозначается как $O(1)$. В качестве примера рассмотрим доступ к n-му элементу массива. Независимо от размера массива для получения результата потребуется одно и то же время. Например, следующая функция вернет первый элемент массива (ее сложность $O(1)$):

```
def getFirst(myList):
    return myList[0]
```

Выходные данные показаны ниже (рис. 1.7):

```
In [2]:  1 getFirst([1,2,3])
Out[2]: 1

In [3]:  1 getFirst([1,2,3,4,5,6,7,8,9,10])
Out[3]: 1
```

Рис. 1.7

- Добавление нового элемента в стек с помощью `push` или удаление элемента из стека с помощью `pop`. Независимо от размера стека добавление или удаление элемента займет одно и то же время.

- Доступ к элементу хеш-таблицы.
- Блочная (иначе называемая корзинная или карманная) сортировка (Bucket sort).

Линейная временная сложность (O(n))

Считается, что алгоритм имеет линейную временную сложность, обозначаемую $O(n)$, если время выполнения прямо пропорционально размеру входных данных. Простой пример — добавление элементов в одномерную структуру данных:

```
def getSum(myList):
    sum = 0
    for item in myList:
        sum = sum + item
    return sum
```

Взгляните на основной цикл алгоритма. Число итераций в основном цикле линейно увеличивается с увеличением значения n, что приводит к сложности $O(n)$ на рис. 1.8.

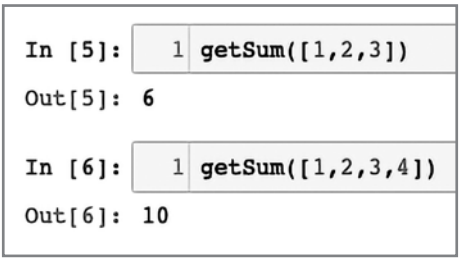


Рис. 1.8

Ниже приведены некоторые другие примеры операций с массивами:

- Поиск элемента.
- Нахождение минимального значения среди всех элементов массива.

Квадратичная временная сложность (O(n²))

Считается, что алгоритм выполняется за квадратичное время, если время выполнения алгоритма пропорционально квадрату размера входных данных. Например, простая функция, которая суммирует двумерный массив, выглядит следующим образом:

```
def getSum(myList):
    sum = 0
    for row in myList:
        for item in row:
            sum += item
    return sum
```

Обратите внимание на вложенный цикл внутри основного цикла. Этот вложенный цикл и придает коду сложность $O(n^2)$ (рис. 1.9).

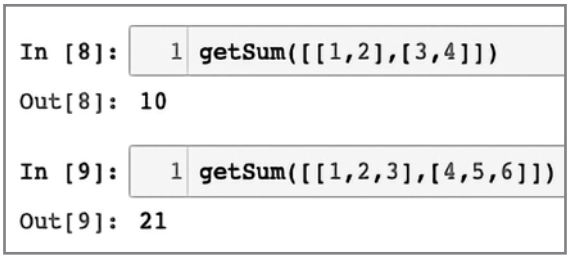


Рис. 1.9

Другим примером квадратичной временной сложности является алгоритм *сортировки пузырьком* (представленный в главе 3).

Логарифмическая временная сложность (O(logn))

Считается, что алгоритм выполняется за логарифмическое время, если время выполнения алгоритма пропорционально логарифму размера входных данных. С каждой итерацией размер входных данных уменьшается в несколько раз. Примером логарифмической временной сложности является бинарный поиск. Алгоритм бинарного поиска используется для поиска определенного элемента в одномерной структуре данных, такой как список в Python. Элементы в структуре данных должны быть отсортированы в порядке убывания. Алгоритм бинарного поиска реализован в функции с именем `searchBinary` следующим образом:

```
def searchBinary(myList,item):
    first = 0
    last = len(myList)-1
    foundFlag = False
    while( first<=last and not foundFlag):
        mid = (first + last)//2
```

```
        if myList[mid] == item :
            foundFlag = True
        else:
            if item < myList[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return foundFlag
```

Работа основного цикла подразумевает тот факт, что список упорядочен. На каждой итерации список делится пополам, пока не будет получен результат (рис. 1.10):

```
In [11]:  1 searchBinary([8,9,10,100,1000,2000,3000], 10)
          2
Out[11]: True

In [12]:  1 searchBinary([8,9,10,100,1000,2000,3000], 5)
Out[12]: False
```

Рис. 1.10

Определив функцию, мы переходим к поиску определенного элемента в строках 11 и 12. Алгоритм бинарного поиска более подробно обсуждается в главе 3.

Обратите внимание, что среди четырех последних представленных типов временной сложности $O(n^2)$ имеет худшую производительность, а $O(\log n)$ — лучшую. Фактически производительность $O(\log n)$ можно рассматривать как золотой стандарт производительности любого алгоритма (что, однако, не всегда достижимо). С другой стороны, $O(n^2)$ не так плох, как $O(n^3)$, но все же алгоритмы этого типа нельзя использовать для больших данных, поскольку временная сложность ограничивает объем данных, которые они способны обработать за разумное количество времени.

Чтобы понизить сложность алгоритма, мы можем пожертвовать точностью и использовать *приближенный алгоритм*.

Весь процесс оценки производительности алгоритмов носит итерационный характер, как показано на рис. 1.11.

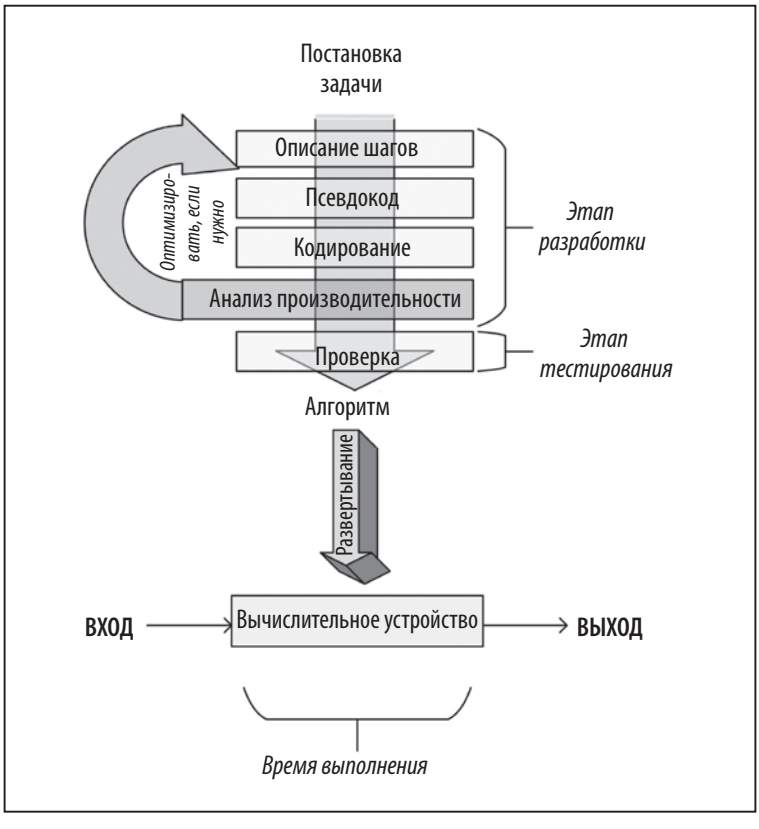


Рис. 1.11

ПРОВЕРКА АЛГОРИТМА

В процессе проверки алгоритма мы должны убедиться, что он действительно предоставляет математическое решение для нашей задачи. Необходимо проверить результаты для максимального числа возможных значений и типов входных данных.

Точные, приближенные и рандомизированные алгоритмы

Для различных типов алгоритмов используются разные методы тестирования. Сначала выявим различие между *детерминированными* и *рандомизированными* алгоритмами.

В случае детерминированных алгоритмов конкретные входные данные всегда порождают одинаковые выходные данные. Но для ряда классов алгоритмов в качестве входных данных используется последовательность случайных чисел, что делает выходные данные разными при каждом запуске алгоритма. Алгоритм *k*-средних, который подробно описан в главе 6, является хорошим примером (рис. 1.12).

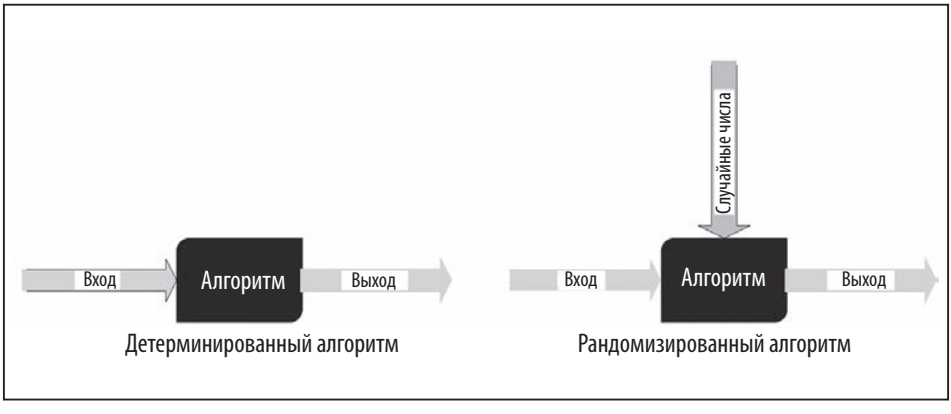


Рис. 1.12

Иногда, чтобы упростить логику и ускорить выполнение алгоритма, используются некоторые допущения. Таким образом, алгоритмы также делятся на два типа:

- *Точный алгоритм* (exact algorithm). Ожидается, что такой алгоритм даст точное решение без каких-либо допущений или приближений.
- *Приближенный алгоритм* (approximate algorithm). Когда сложность задачи слишком велика для имеющихся ресурсов, мы упрощаем нашу задачу, делая допущения. Алгоритмы, основанные на этих упрощениях или допущениях, называются приближенными алгоритмами и дают не совсем точное решение.

Чтобы понять разницу между точными и приближенными алгоритмами, рассмотрим знаменитую *задачу коммивояжера* (TSP — travelling salesman problem), впервые представленную в 1930 году. Задача состоит в том, чтобы найти кратчайший маршрут для торговца, при котором он посетит каждый город из списка, а затем вернется в исходную точку (поэтому она и называется задачей коммивояжера). Для ее решения нужно рассмотреть все возможные комбинации городов и выбрать наименее затратную. Сложность этого подхода составляет $O(n!)$,

где n — количество городов. Очевидно, что временная сложность при $n > 30$ слишком велика.

Если число городов превышает 30, одним из способов снижения сложности является введение ряда приближений и допущений.

При описании требований к приближенному алгоритму важно задать ожидаемую точность. Во время проверки такого алгоритма мы должны убедиться, что погрешность результатов остается в допустимом диапазоне.

Объяснимость алгоритма

Если алгоритм применяется в особо важной ситуации, мы должны иметь возможность объяснить причину того или иного результата. Необходимо убедиться, что вследствие использования алгоритма было принято справедливое решение.

Возможность четко определить признаки, которые прямо или косвенно повлияли на конкретное решение, называется *объяснимостью* алгоритма. Алгоритмы, используемые в жизненно важных ситуациях, оцениваются с точки зрения обоснованности результатов. Этический анализ стал стандартной частью процесса проверки алгоритмов, влияющих на принятие решений, которые касаются жизни людей.

В случае с алгоритмами глубокого обучения достичь объяснимости бывает трудно. Например, если алгоритм используется для отказа в заявлении на ипотеку, важны его прозрачность и возможность объяснить причину.

Алгоритмическая объяснимость — это область активных исследований. Один из эффективных методов *LIME* (Local Interpretable Model-Agnostic Explanations) был предложен в материалах Ассоциации вычислительной техники (ACM) на 22-й международной конференции по извлечению знаний из данных Special Interest Group on Knowledge Discovery (SIGKDD) в 2016 году. Согласно этому методу, во входные данные каждого экземпляра вносятся небольшие изменения, а затем предпринимается попытка отобразить локальную границу принятия решений для этого экземпляра. После этого можно количественно оценить влияние каждой переменной для этого экземпляра.

РЕЗЮМЕ

Эта глава была посвящена изучению основ алгоритмов. Мы узнали об этапах разработки и обсудили различные способы определения логики алгоритма. Далее разобрались, как разработать алгоритм. Наконец, рассмотрели способы анализа производительности и различные аспекты проверки алгоритма.

Прочитав эту главу, мы узнали, что такое псевдокод алгоритма. Нам стали понятны различные этапы разработки и развертывания алгоритма. Мы также узнали, как применять «О-большое» для оценки производительности алгоритма.

Следующая глава посвящена структурам данных, используемым в алгоритмах. А начнем мы с рассмотрения структур данных, представленных в Python. Затем научимся создавать более сложные структуры (стеки, очереди и деревья), необходимые для разработки сложных алгоритмов.

2

Структуры данных, используемые в алгоритмах

Во время работы алгоритму необходима структура данных, чтобы хранить временные данные в памяти. Выбор подходящей структуры имеет ключевое значение для эффективной реализации алгоритма. Определенные классы алгоритмов являются рекурсивными или итеративными по своей логике и нуждаются в специально разработанных структурах данных. Например, добиться хорошей производительности рекурсивного алгоритма проще, если использовать вложенные структуры. Поскольку в книге мы используем Python, эта глава будет посвящена структурам данных Python. Однако эти знания пригодятся и для работы с другими языками, такими как Java и C++.

Прочтя эту главу, вы узнаете, как именно Python обрабатывает сложные структуры данных и какие из них используются для определенного типа данных.

В этой главе нас ждут:

- Структуры данных в Python.
- Абстрактные типы данных.
- Стеки и очереди.
- Деревья.

СТРУКТУРЫ ДАННЫХ В PYTHON

В любом языке программирования структуры данных используются для хранения и управления сложными данными. В Python структуры данных — это контейнеры, позволяющие эффективно управлять данными, организовывать их и осуществлять поиск. Они организованы в *коллекции* — группы элементов данных, которые требуется хранить и обрабатывать совместно. В Python существуют пять различных структур данных для хранения коллекций:

- *Список (List)*. Упорядоченная изменяемая последовательность элементов.
- *Кортеж (Tuple)*. Упорядоченная неизменяемая последовательность элементов.
- *Множество (Set)*. Неупорядоченная последовательность элементов.
- *Словарь (Dictionary)*. Неупорядоченная последовательность пар «ключ — значение».
- *DataFrame*. Двумерная структура для хранения двумерных данных.

Давайте рассмотрим их более подробно.

Список

В Python *список* — это основная структура данных, используемая для хранения изменяемой последовательности элементов.

Элементы данных в списке могут быть разных типов.

Чтобы создать список, элементы нужно заключить в квадратные скобки [ ] и разделить запятыми. Ниже представлен пример кода, который создает список из четырех элементов данных разных типов:

```
>>> aList = ["John", 33,"Toronto", True]
>>> print(aList)
['John', 33, 'Toronto', True]Ex
```

Список в Python — это удобный способ создания одномерных изменяемых структур данных, необходимых главным образом на различных внутренних этапах алгоритма.

Использование списков

При работе со списками мы получаем полезные инструменты для управления данными.

Давайте посмотрим, что можно делать со списками.

- *Индексация списка*. Поскольку положение каждого элемента в списке детерминировано, к нему можно получить доступ с помощью индекса. Следующий код демонстрирует этот принцип:

```
>>> bin_colors=['Red','Green','Blue','Yellow']
>>> bin_colors[1]
'Green'
```

  Список из четырех элементов, созданный этим кодом, показан на рис. 2.1.

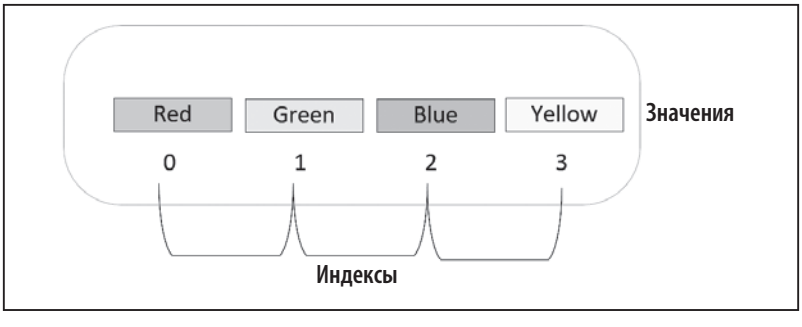


Рис. 2.1

  Обратите внимание, что индексация начинается с 0, поэтому второй элемент *Green* извлекается с помощью индекса *1*: `bin_color[1]`.

- *Срез списка*. Извлечение подмножества элементов списка путем указания диапазона индексов называется *срезом*. Пример кода среза:

```
>>> bin_colors=['Red','Green','Blue','Yellow']
>>> bin_colors[0:2]
['Red', 'Green']
```

Список — одна из самых популярных одномерных структур данных в Python.

При срезе списка диапазон указывается следующим образом: первое число (включительно) и второе число (не включительно). Например, bin_colors[0:2] будет включать bin_color[0] и bin_color[1], но не bin_color[2]. Следует учитывать это при использовании списков; некоторые пользователи Python жалуются, что это не слишком очевидно.

Рассмотрим следующий сниппет:

```
>>> bin_colors=['Red','Green','Blue','Yellow']
>>> bin_colors[2:]
['Blue', 'Yellow']
>>> bin_colors[:2]
['Red', 'Green']
```

Если в квадратных скобках не указан первый индекс, это означает начало списка, а если пропущен второй — конец списка. Выше мы видим пример такого кода.

- *Отрицательная индексация*. В Python имеются и отрицательные индексы, которые отсчитываются от конца списка. Это показано в следующем коде:

  ```
  >>> bin_colors=['Red','Green','Blue','Yellow']
  >>> bin_colors[:-1]
  ['Red', 'Green', 'Blue']
  >>> bin_colors[:-2]
  ['Red', 'Green']
  >>> bin_colors[-2:-1]
  ['Blue']
  ```

  Отрицательные индексы особенно полезны, когда в качестве точки отсчета мы хотим использовать последний элемент вместо первого.

- *Вложенность*. Элемент списка может относиться к простому или сложному типу данных. Это позволяет создавать вложенные списки и дает возможность использовать потенциал итеративных и рекурсивных алгоритмов.

  Рассмотрим пример списка внутри списка (вложенность):

  ```
  >>> a = [1,2,[100,200,300],6]
  >>> max(a[2])
  300
  >>> a[2][1]
  200
  ```

- *Итерация*. Python позволяет выполнять итерацию для каждого элемента в списке с помощью цикла `for`.

  Это показано в следующем примере:

  ```
  >>> bin_colors=['Red','Green','Blue','Yellow']
  >>> for aColor in bin_colors:
          print(aColor + " Square")
  Red Square
  Green Square
  Blue Square
  Yellow Square
  ```

Обратите внимание, что данный код выполняет итерацию по списку и отображает каждый элемент.

Лямбда-функции

Существует множество *лямбда-функций*, которые можно использовать в списках. Они особенно полезны в работе с алгоритмами и позволяют создавать функцию на лету. Иногда в литературе их также называют *анонимными функциями*. Рассмотрим их применение:

- *Фильтрация данных*. Для фильтрации данных мы должны определить предикат — функцию, которая тестирует каждый аргумент и возвращает логическое значение. Пример использования такой функции:

```
>>> list(filter(lambda x: x > 100, [-5, 200, 300, -10, 10, 1000]))
[200, 300, 1000]
```

  В данном коде мы фильтруем список с помощью лямбда-функции, которая задает критерии фильтрации. Функция `filter()` предназначена для отбора элементов из последовательности на основе определенного критерия и обычно используется вместе с лямбда-функцией. Ее также можно применять для фильтрации элементов кортежей или наборов. В нашем примере заданным критерием является `x > 100`. Код проверяет все элементы списка и отсеивает те из них, которые не соответствуют этому критерию.

- *Преобразование данных*. Функция `map()` используется для преобразования данных с помощью лямбда-функции. Пример:

```
>>> list(map(lambda x: x ** 2, [11, 22, 33, 44,55]))
[121, 484, 1089, 1936, 3025]
```

  Использование функции `map()` вместе с лямбда-функцией открывает большие возможности. При этом лямбда-функция задает преобразователь, изменяющий каждый элемент последовательности. В приведенном примере преобразователем выступает возведение во вторую степень. Таким образом, мы используем функцию `map()` для получения квадрата каждого элемента в списке.

- *Агрегирование данных*. Для агрегирования данных используется `reduce()`, которая рекурсивно применяет функцию к паре значений для каждого элемента списка:

```
from functools import reduce
def doSum(x1,x2):
    return x1+x2
x = reduce(doSum, [100, 122, 33, 4, 5, 6])
```

Обратите внимание, что для `reduce()` необходимо определить функцию агрегирования данных. В приведенном примере кода такой функцией является `functools`. Она определяет, как именно нужно агрегировать элементы данного списка. Агрегирование начинается с первых двух элементов, а его результат заменяет эти два элемента. Процесс сокращения будет происходить до тех пор, пока не останется одно агрегированное число. `x1` и `x2` в функции `doSum()` представляют собой пару чисел для каждой итерации; `doSum()` является критерием агрегирования.

В результате мы получаем единое значение (равное `270`).

Функция range

С помощью функции `range()` можно легко сгенерировать большой список чисел. Она используется для автоматического добавления последовательностей чисел в список.

Функцию `range()` очень просто использовать — нужно только указать количество элементов, которые мы хотим видеть в списке. По умолчанию последовательность начинается с нуля и далее увеличивается с шагом, равным единице:

```
>>> x = range(6)
>>> x
[0,1,2,3,4,5]
```

Мы также можем указать конечное число и шаг, например:

```
>>> oddNum = range(3,29,2)
>>> oddNum
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27]
```

В результате получим нечетные числа от `3` до `27`.

Временная сложность списков

Временную сложность различных функций списка можно обобщить, используя нотацию «О-большое» (табл. 2.1).

Чем больше список, тем больше времени требуется на выполнение операций, представленных в таблице. Это не касается только вставки элемента. По мере увеличения списка влияние на производительность становится все более выраженным.

Таблица 2.1

Операция	Временная сложность
Вставить элемент	$O(1)$
Удалить элемент	$O(n)$ (так как в худшем случае, возможно, придется перебрать весь список)
Срез списка	$O(n)$
Извлечение элемента	$O(n)$
Копирование	$O(n)$

Кортеж

Еще одна структура данных, которую можно использовать для хранения коллекции, — *кортеж*. В отличие от списков, кортежи являются неизменяемыми (доступными только для чтения) структурами данных. Кортежи состоят из нескольких элементов, заключенных в круглые скобки ( ).

Кортежи, как и списки, могут включать в себя элементы разных типов, в том числе сложные типы данных — внутри кортежа может быть еще один кортеж. Таким образом, мы можем создавать вложенные структуры. Это особенно полезно для работы с итеративными и рекурсивными алгоритмами.

В данном коде показано, как создавать кортежи:

```
>>> bin_colors=('Red','Green','Blue','Yellow')
>>> bin_colors[1]
'Green'
>>> bin_colors[2:]
('Blue', 'Yellow')
>>> bin_colors[:-1]
('Red', 'Green', 'Blue')
# Nested Tuple Data structure (вложенный кортеж)
>>> a = (1,2,(100,200,300),6)
>>> max(a[2])
300
>>> a[2][1]
200
```

Обратите внимание, что в представленном выше коде `a[2]` относится к третьему элементу, который является кортежем: `(100,200,300)`; `a[2][1]` относится ко второму элементу внутри этого кортежа, который является числом `200`.

По возможности старайтесь использовать неизменяемые структуры данных вместо изменяемых (например, кортежи вместо списков), так как это улучшит производительность. В особенности это касается обработки больших данных: неизменяемые структуры работают значительно быстрее, чем изменяемые. Мы платим определенную цену за возможность изменять элементы данных в списке. Нужно понять, действительно ли это необходимо или же можно использовать кортеж, что будет намного быстрее.

Временная сложность кортежей

Временную сложность различных функций кортежей можно обобщить с помощью «О-большого» (табл. 2.2).

Таблица 2.2

Функция	Временная сложность
`Append()`	$O(1)$

`Append()` — это функция, которая добавляет элемент в конец уже существующего кортежа. Ее сложность равна $O(1)$.

Словарь

Хранение данных в виде пар «ключ — значение» особенно полезно при работе с распределенными алгоритмами. В Python коллекция пар «ключ — значение» хранится в виде структуры данных, называемой *словарем*. Чтобы создать словарь, в качестве атрибута следует выбрать ключ, лучше всего подходящий для идентификации данных во время обработки. Значением ключа может быть элемент любого типа, например число или строка. В Python в качестве значений также используются сложные типы данных, например списки. Если использовать в качестве значения ключа словарь, можно создавать вложенные словари.

Чтобы создать простой словарь, который присваивает цвета различным переменным, пары «ключ — значение» должны быть заключены в фигурные скобки { }. Например, следующий код создает простой словарь, состоящий из трех пар «ключ — значение»:

```
>>> bin_colors ={
      "manual_color": "Yellow",
```

```
    "approved_color": "Green",
    "refused_color": "Red"
  }
>>> print(bin_colors)
{'manual_color': 'Yellow', 'approved_color': 'Green', 'refused_color':
'Red'}
```

Три пары «ключ — значение», созданные предыдущим фрагментом кода, также проиллюстрированы на рис. 2.2.

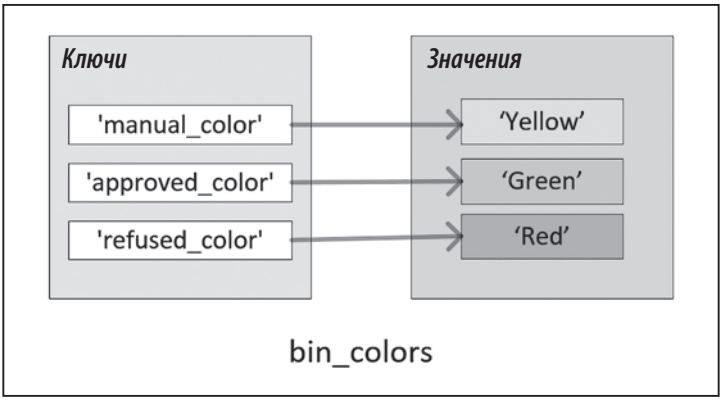


Рис. 2.2

Теперь давайте посмотрим, как получить и обновить значение, связанное с ключом.

1. Чтобы получить значение, связанное с ключом, можно использовать либо функцию `get()`, либо ключ в качестве индекса:

```
>>> bin_colors.get('approved_color')
'Green'
>>> bin_colors['approved_color']
'Green'
```

2. Чтобы обновить значение, связанное с ключом, используйте следующий код:

```
>>> bin_colors['approved_color']="Purple"
>>> print(bin_colors)
{'manual_color': 'Yellow', 'approved_color': 'Purple',
'refused_color': 'Red'}
```

   Данный код показывает, как обновить значение, связанное с определенным ключом в словаре.

Временная сложность словаря

В табл. 2.3 приведена временная сложность словаря с использованием «О-большого».

Таблица 2.3

Операция	**Временная сложность**
Получить значение или ключ	O(1)
Установить значение или ключ	O(1)
Скопировать словарь	O(n)

Из анализа сложности словаря следует важный вывод: время, необходимое для получения или установки значения ключа, никак не зависит от размера словаря. Это означает, что время, затраченное на добавление пары «ключ — значение» в словарь размером три (например), равно времени, затраченному на добавление пары «ключ — значение» в словарь размером один миллион.

Множество

Множество — это коллекция элементов одного или разных типов. Элементы заключены в фигурные скобки { }. Взгляните на следующий сниппет:

```
>>> green = {'grass', 'leaves'}
>>> print(green)
{'grass', 'leaves'}
```

Отличительной особенностью множества является то, что в нем хранится только уникальное значение каждого элемента. Если мы попытаемся добавить дубль, он будет проигнорирован:

```
>>> green = {'grass', 'leaves','leaves'}
>>> print(green)
{'grass', 'leaves'}
```

Рассмотрим операции над множествами. Для этого возьмем два множества:

- множество с именем yellow, в котором содержатся вещи желтого цвета;
- множество с именем red, в котором содержатся вещи красного цвета.

Обратите внимание, что некоторые вещи содержатся в обоих множествах. Эти два множества и их взаимосвязь можно представить с помощью диаграммы Венна (рис. 2.3).

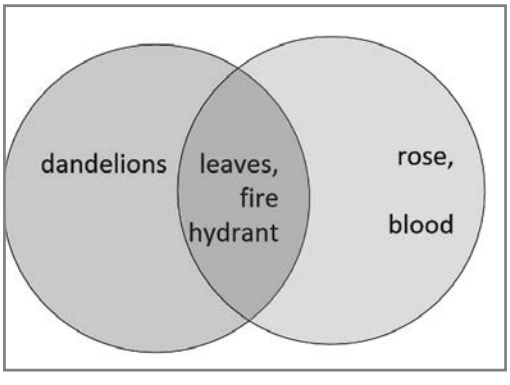


Рис. 2.3 (dandelions — одуванчики, leaves — листья, fire hydrant — пожарный кран, rose — роза, blood — кровь)

Реализация этих множеств в Python выглядит следующим образом:

```
>>> yellow = {'dandelions', 'fire hydrant', 'leaves'}
>>> red = {'fire hydrant', 'blood', 'rose', 'leaves'}
```

Теперь рассмотрим код, который демонстрирует операции на множествах с использованием Python:

```
>>> yellow|red
{'dandelions', 'fire hydrant', 'blood', 'rose', 'leaves'}
>>> yellow&red
{'fire hydrant'}
```

В данном примере продемонстрированы две операции: объединение и пересечение. Объединение совмещает все элементы обоих множеств, а пересечение дает набор общих элементов для двух множеств. Обратите внимание:

- `yellow|red` используется для объединения двух множеств;
- `yellow&red` используется для получения пересечения yellow и red.

Анализ временной сложности множеств

В табл. 2.4 приведен анализ временной сложности для множеств.

Таблица 2.4

Операция	Сложность
Добавление элемента	$O(1)$
Удаление элемента	$O(1)$
Копирование	$O(n)$

На основании анализа сложности можно сделать вывод, что время, затраченное на добавление элемента, не зависит от размера конкретного множества.

DataFrame

DataFrame — табличная структура данных, доступная в библиотеке Python `pandas`. Это одна из наиболее важных структур данных для алгоритмов. Она используется для обработки классических структурированных данных. Рассмотрим таблицу (табл. 2.5).

Таблица 2.5

id	name (имя)	age (возраст)	decision (решение)
1	Fares	32	True
2	Elena	23	False
3	Steven	40	True

Теперь представим эту таблицу с помощью DataFrame.

Простейший DataFrame может быть создан с помощью следующего кода:

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...             ['1', 'Fares', 32, True],
...             ['2', 'Elena', 23, False],
...             ['3', 'Steven', 40, True]])
>>> df.columns = ['id', 'name', 'age', 'decision']
>>> df
  id   name   age  decision
0  1  Fares   32      True
1  2  Elena   23     False
2  3  Steven  40      True
```

Обратите внимание, что в данном коде `df.column` — это список, в котором содержатся имена столбцов.

DataFrame используются и в других популярных языках и фреймворках для реализации табличной структуры данных. Примерами могут служить язык программирования R и платформа Apache Spark.

Терминология DataFrame

Ознакомимся с терминологией, необходимой для работы с DataFrame:

- *Ось*. В документации pandas один столбец или строка DataFrame называется осью (axis).
- *Метка*. DataFrame позволяет отмечать как столбцы, так и строки так называемой меткой (label).

Создание подмножества DataFrame

По сути, существуют два основных способа создания подмножества DataFrame (пусть это будет подмножество с именем `myDF`):

- выбор столбца;
- выбор строки.

Рассмотрим их по очереди.

Выбор столбца

При работе с алгоритмами машинного обучения важно использовать правильный набор признаков. Далеко не все доступные нам признаки могут понадобиться на разных этапах алгоритма. В Python отбор признаков происходит путем выбора столбцов.

Получить доступ к столбцу можно с помощью его имени (атрибута *name*), как показано ниже:

```
>>> df[['name','age']]
     name  age
0   Fares  32
1   Elena  23
2  Steven  40
```

Позиция столбца является детерминированной. Доступ к нему по его расположению можно получить следующим образом:

```
>>> df.iloc[:,3]
0 True
1 False
2 True
```

Обратите внимание, что в этом коде мы извлекаем первые три строки DataFrame.

Выбор строки

Каждая строка DataFrame соответствует точке данных в пространстве задачи. Чтобы создать подмножество из имеющихся элементов данных, необходимо выбрать строки. Существуют два метода создания подмножества:

- указать расположение строк;
- задать критерии фильтра.

Подмножество строк может быть получено по расположению следующим образом:

```
>>> df.iloc[1:3,:]
  id name age decision
1 2 Elena 23 False
2 3 Steven 40 True
```

Данный код вернет первые две строки и все столбцы.

Чтобы создать подмножество с помощью фильтра, мы должны указать критерии выбора в одном или нескольких столбцах. Это происходит следующим образом:

```
>>> df[df.age>30]
   id   name  age  decision
0  1   Fares   32     True
2  3  Steven   40     True

>>> df[(df.age<35)&(df.decision==True)]
   id   name  age  decision
0  1   Fares   32     True
```

Обратите внимание, что этот код создает подмножество строк, удовлетворяющее условию, указанному в фильтре.

Матрица

Матрица — это двумерная структура данных с фиксированным количеством столбцов и строк.

На каждый элемент матрицы можно ссылаться по его столбцу и строке.

В Python матрицу можно создать с помощью массива numpy:

```
>>> myMatrix = np.array([[11, 12, 13], [21, 22, 23], [31, 32, 33]])
>>> print(myMatrix)
[[11 12 13]
[21 22 23]
[31 32 33]]
>>> print(type(myMatrix))
<class 'numpy.ndarray'>
```

Этот код создает матрицу, содержащую три строки и три столбца.

Операции с матрицами

Существует множество операций, доступных для матричных данных. Давайте транспонируем матрицу из предыдущего примера. Для этого используем функцию `transpose()`, которая преобразует столбцы в строки, а строки в столбцы:

```
>>> myMatrix.transpose()
array([[11, 21, 31],
       [12, 22, 32],
       [13, 23, 33]])
```

Отметим, что матричные операции часто используются при обработке мультимедийных данных.

Теперь, когда мы узнали о структурах данных в Python, перейдем к абстрактным типам данных.

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

В широком смысле абстракция — принцип, используемый для определения сложных систем с точки зрения их общих базовых функций. Применение этой концепции при создании структур данных приводит к появлению *абстрактных типов данных* (АТД[1]). Используя АТД, мы получаем универсальную, независимую от реализации структуру данных. Это позволяет написать более простой и чистый код алгоритма, не углубляясь в детали разработки. АТД можно реализовать на любом языке программирования, например C++, Java и Scala. В этом разделе мы будем использовать АТД в Python. Начнем с вектора.

[1] Или ADT (Abstract Data Type). — *Примеч. ред.*

Вектор

Вектор — это одномерная структура для хранения данных, одна из самых популярных в Python. В Python имеются два способа создания векторов.

- Использование списка Python. Самый простой способ создания вектора — применить список Python следующим образом:

```
>>> myVector = [22,33,44,55]
>>> print(myVector)
[22 33 44 55]
>>> print(type(myVector))
<class 'list'>
```

  Этот код создает список из четырех элементов.

- Использование массива `numpy`. Еще один популярный способ создания вектора — применение массивов NumPy, как показано ниже:

```
>>> myVector = np.array([22,33,44,55])
>>> print(myVector)
[22 33 44 55]
>>> print(type(myVector))
<class 'numpy.ndarray'>
```

Обратите внимание, что мы создали `MyVector`, используя `np.array`.

В Python мы используем нижнее подчеркивание при написании целых чисел, разделяя их на разряды. Это делает их удобочитаемыми и уменьшает вероятность ошибки, что особенно важно при работе с большими числами. Например, один миллиард можно представить так: a=1_000_000_000.

Стек

Стек — это линейная структура данных для хранения одномерного списка. Элементы в стеке могут обрабатываться по принципу LIFO (Last-In, First-Out: *«последним пришел — первым ушел»*) либо по принципу FILO (First-In, Last-Out: *«первым пришел — последним ушел»*. Порядок добавления и удаления элементов определяет характер стека. Новые элементы могут добавляться и удаляться только с одного конца списка.

Ниже приведены операции со стеками:

- *isEmpty.* Возвращает true, если стек пуст;

- *push*. Добавляет новый элемент;
- *pop*. Возвращает элемент, добавленный последним, и удаляет его.

На рис. 2.4 показано, как операции `push()` и `pop()` можно использовать для добавления и удаления данных из стека.

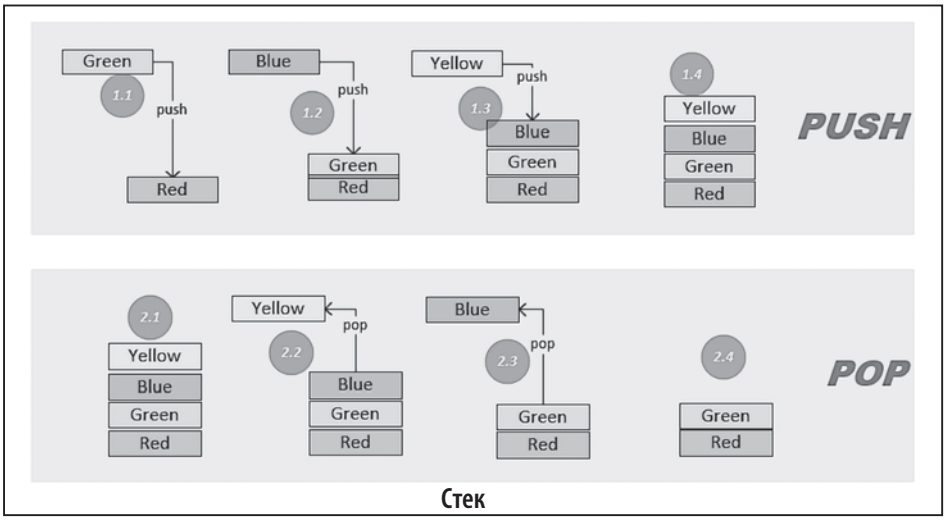


Рис. 2.4

В верхней части рис. 2.4 показано использование операции `push()` для добавления элементов в стек. На шагах *1.1*, *1.2* и *1.3* операция `push()` используется три раза для добавления трех элементов в стек. В нижней части рисунка демонстрируется извлечение сохраненных значений из стека. На шагах *2.2* и *2.3* операция `pop()` используется для извлечения двух элементов из стека в формате LIFO.

Давайте создадим класс с именем `Stack`, в котором опишем все операции, связанные с классом stack. Код этого класса будет выглядеть следующим образом:

```
class Stack:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
```

```
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
```

Чтобы поместить четыре элемента в стек, можно использовать следующий код (рис. 2.5).

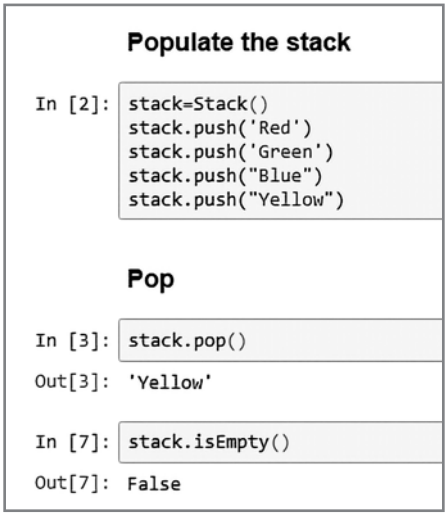


Рис. 2.5

Этот код создает стек с четырьмя элементами данных.

Временная сложность стеков

Рассмотрим временную сложность стеков, используя «О-большое» (табл. 2.6).

Таблица 2.6

Операция	Временная сложность
`push`	$O(1)$
`pop`	$O(1)$
`size`	$O(1)$
`peek`	$O(1)$

Видим, что производительность этих четырех операций не зависит от размера стека.

Практический пример

Стек часто применяется на практике в качестве структуры данных. Например, он используется для хранения истории веб-браузера. Другой пример — выполнение операции Undo при работе с текстом.

Очередь

Как и стек, *очередь* хранит *n* элементов в одномерной структуре. Элементы добавляются и удаляются по принципу FIFO (First-In, First-Out: *«первым пришел — первым ушел»*). Каждая очередь имеет *начало* и *конец*. Когда элементы удаляются из начала, операция называется *удалением из очереди* — dequeue. Когда элементы добавляются в конец, операция называется *постановкой в очередь* — enqueue.

На следующей диаграмме (рис. 2.6) в верхней части показана операция enqueue(). Шаги *1.1*, *1.2* и *1.3* добавляют три элемента в очередь; итоговая очередь показана на шаге *1.4*. Обратите внимание, что *Yellow — в конце*, а *Red — в начале*.

В нижней части диаграммы представлена операция dequeue(). Шаги *2.2*, *2.3* и *2.4* удаляют элементы из начала очереди один за другим.

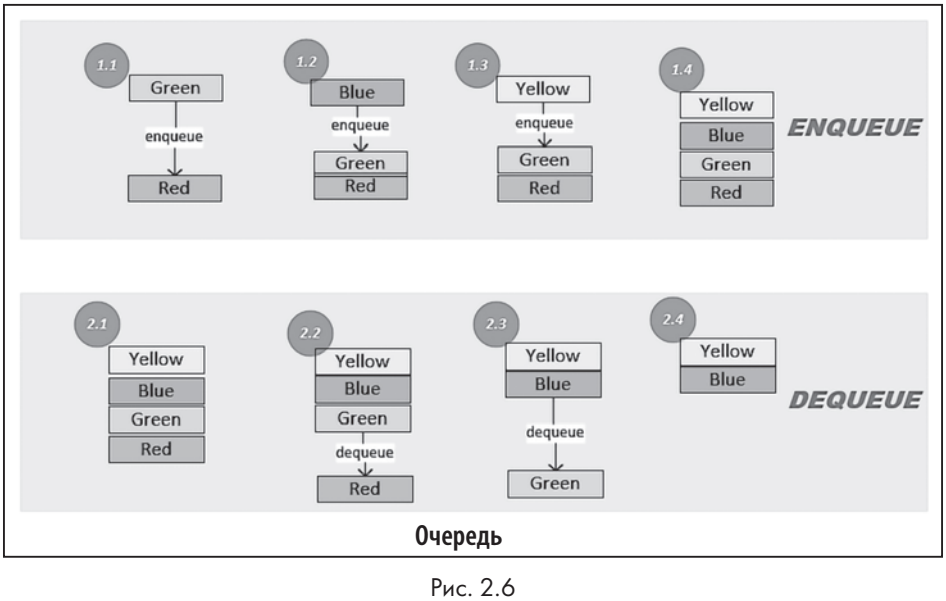


Рис. 2.6

Эта очередь может быть реализована с помощью следующего кода:

```
class Queue(object):
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def enqueue(self, item):
        self.items.insert(0,item)
    def dequeue(self):
        return self.items.pop()
    def size(self):
        return len(self.items)
```

С помощью скриншота на рис. 2.7 выполним постановку и удаление элементов из очереди, как это показано на диаграмме выше.

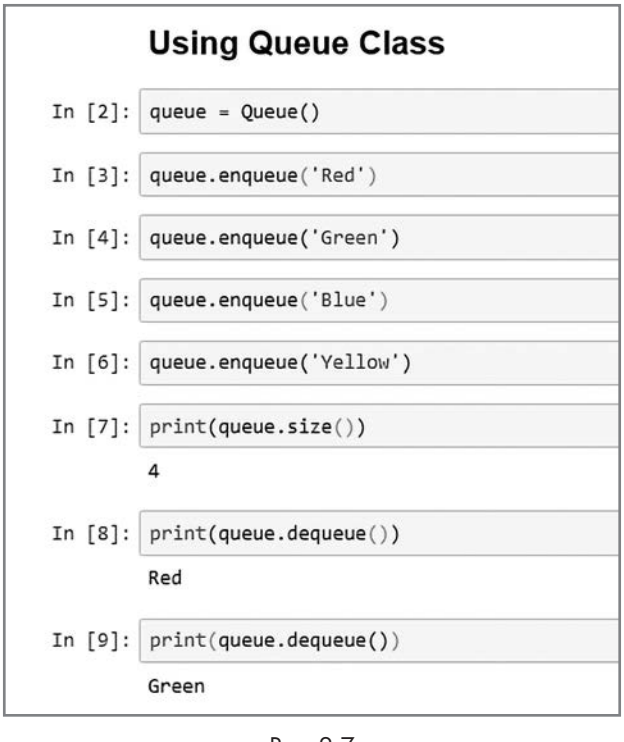


Рис. 2.7

Обратите внимание, что код сначала создает очередь, а затем помещает в нее четыре элемента.

Базовый принцип использования стеков и очередей

Рассмотрим базовый принцип использования стеков и очередей с помощью аналогии. Представьте, что мы получаем почтовую корреспонденцию и складываем ее на стол. Письма накапливаются в стопки, пока мы не находим время, чтобы открыть и просмотреть их одно за другим. Есть два способа сделать это:

- Мы складываем письма в стопку, и всякий раз, когда мы получаем новое письмо, мы кладем его наверх. Когда мы хотим прочитать письма, мы начинаем с того, которое лежит сверху. Стопка — это то, что мы называем *стеком*. Обратите внимание, что последнее поступившее письмо находится сверху и будет обработано первым. Взять письмо из верхней части стопки означает выполнить операцию *pop*. Положить новое письмо сверху — выполнить операцию *push*. Если в итоге у нас получится большая стопка, а письма продолжат приходить, то есть вероятность, что мы никогда не доберемся до очень важного письма в самом низу.
- Мы складываем письма в стопку, но сначала хотим открыть самое старое письмо: каждый раз, когда мы хотим просмотреть одно или несколько писем, мы начинаем с более старых. Это — *очередь*. Добавление письма в стопку — операция *enqueue (постановка в очередь)*. Удаление письма из стопки — операция *dequeue (удаление из очереди)*.

Дерево

Дерево — иерархическая структура данных, что делает ее особенно полезной при разработке алгоритмов. Мы используем деревья везде, где требуются иерархические отношения между элементами данных.

Давайте подробнее рассмотрим эту интересную и важную структуру.

Каждое дерево имеет конечный набор узлов, так что в нем есть начальный элемент данных, называемый *корнем* (root), и набор узлов, соединенных между собой *ветвями* (branches).

Терминология

Рассмотрим некоторые термины, связанные с древовидной структурой данных (табл. 2.7).

Таблица 2.7

Корневой узел (Root node)	Узел без родителя называется корневым узлом. Например, на следующей диаграмме (рис. 2.8) корневым узлом служит *A*. В алгоритмах корневой узел, как правило, содержит наиболее важное значение во всей древовидной структуре
Уровень узла (Level of a node)	Расстояние от корневого узла называется уровнем узла. На следующей диаграмме уровень узлов *D*, *E* и *F* равен двум
Узлы-братья (Siblings nodes)	Два узла в дереве называются *братьями*, если они расположены на одном уровне. Например, если мы взглянем на диаграмму, то увидим, что узлы *B* и *C* являются братьями
Дочерний и родительский узлы (Child and parent node)	Узел *F* является дочерним по отношению к узлу *C*, если они напрямую связаны и уровень узла *C* меньше уровня узла *F*. И наоборот, узел *C* является родительским для узла *F*. Узлы *C* и *F* на следующей диаграмме демонстрируют отношения родительского и дочернего узлов
Степень узла (Degree of a node)	Степень узла — это количество его дочерних элементов. Например, на рис. 2.8 узел *B* имеет степень 2
Степень дерева (Degree of a tree)	Степень дерева равна максимальной степени составляющих его узлов. Дерево, представленное на следующей диаграмме, имеет степень 2
Поддерево (Subtree)	Поддерево — это часть дерева с выбранным узлом в качестве корневого, а все его дочерние элементы — это узлы дерева. На диаграмме поддерево от узла *E* состоит из узла *E* в качестве корневого и узлов *G* и *H* в качестве дочерних
Концевой узел (Leaf node)	Узел в дереве без дочерних элементов называется *концевым*. Например, на рис. 2.8 *D*, *G*, *H* и *F* — это четыре концевых узла
Внутренний узел (Internal node)	Любой узел, который не является ни корневым, ни концевым, называется внутренним. У внутреннего узла имеются по крайней мере один родительский и один дочерний узлы

Обратите внимание, что деревья — это своего рода сети или графы, которые мы будем изучать в главе 5. При анализе графов и сетей вместо ветвей мы используем термин «ребро». Большая часть остальной терминологии остается неизменной.

Типы деревьев

Существует несколько типов деревьев:

- *Двоичное дерево* (binary tree). Если степень дерева равна двум, оно называется *двоичным*. Например, дерево, показанное на следующей диаграмме, является двоичным, поскольку имеет степень 2 (рис. 2.8).

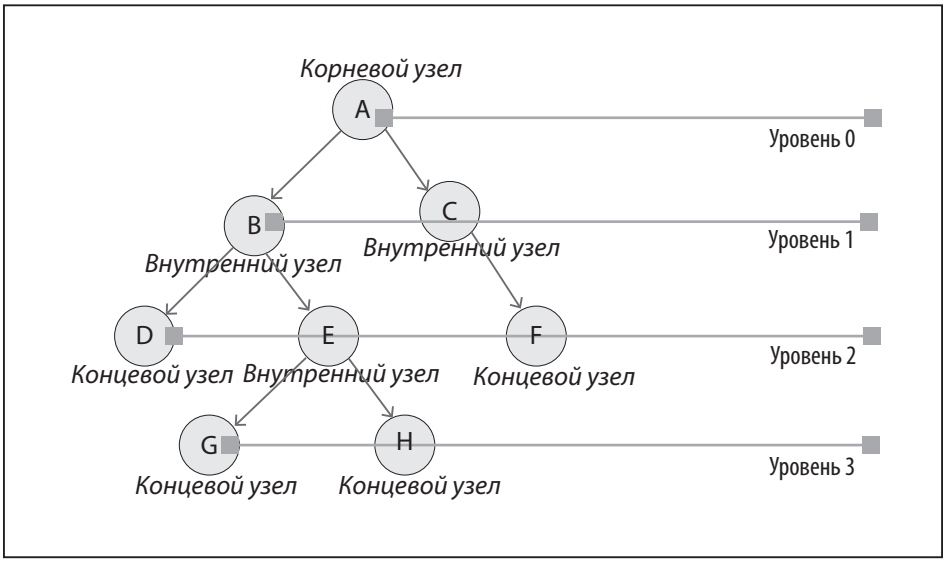


Рис. 2.8

Обратите внимание, что дерево на рис. 2.8 имеет четыре уровня и восемь узлов.

- *Полное дерево* (full tree). Это дерево, в котором все узлы имеют одинаковую степень, которая равна степени дерева. На диаграмме ниже представлены упомянутые типы деревьев (рис. 2.9).

  Обратите внимание, что двоичное дерево слева не является полным, так как узел *C* имеет степень 1, а все остальные узлы — степень 2. Деревья в центре и справа являются полными.

- *Идеальное дерево* (perfect tree). Это особый тип полного дерева, у которого все конечные узлы расположены на одном уровне. На рис. 2.9 двоичное дерево справа является идеальным полным деревом, поскольку все его конечные узлы находятся *на уровне 2*.

- *Упорядоченное дерево* (ordered tree). Если дочерние элементы узла организованы в определенном порядке согласно установленным критериям, дерево называется *упорядоченным*. Дерево, например, может быть упорядочено слева направо в порядке возрастания. Таким образом, значение узлов одного уровня будет увеличиваться при движении слева направо.

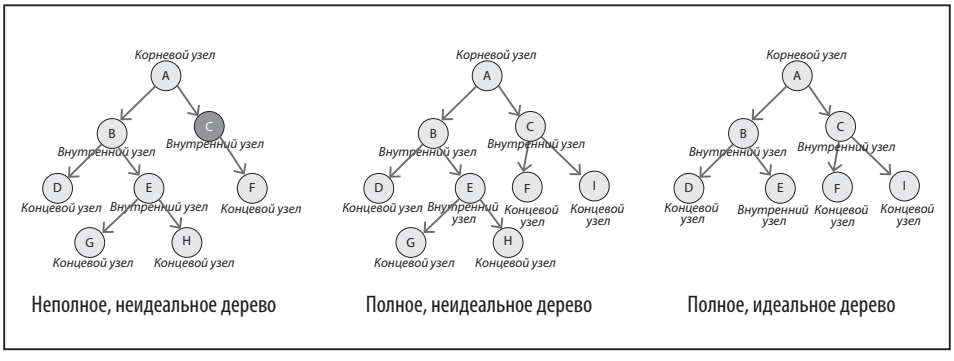


Рис. 2.9

Практические примеры

Дерево — одна из основных структур данных, используемых при разработке деревьев решений. Их мы обсудим в главе 7. Благодаря своей иерархической структуре деревья используются в алгоритмах сетевого анализа (см. главу 5), а также в алгоритмах поиска и сортировки, где применяются стратегии типа «разделяй и властвуй».

РЕЗЮМЕ

В этой главе мы обсудили структуры данных, используемые для реализации различных типов алгоритмов. Теперь вы сможете подобрать подходящую структуру данных для вашего алгоритма. Помните, что выбор той или иной структуры влияет на производительность.

Следующая глава посвящена алгоритмам сортировки и поиска. При работе с ними мы будем использовать изученные ранее структуры данных.

3

Алгоритмы сортировки и поиска

Алгоритмы сортировки и поиска — важный класс алгоритмов. Они могут использоваться как самостоятельно, так и в качестве основы для более сложных алгоритмов, о которых мы поговорим в следующих главах. Мы познакомимся с различными типами алгоритмов сортировки, сравним их производительность при различных подходах к разработке; затем подробно рассмотрим несколько алгоритмов поиска. Наконец, мы разберем изученные алгоритмы на практическом примере.

К концу главы вы научитесь оценивать сильные и слабые стороны различных алгоритмов сортировки и поиска. Такие алгоритмы — основа для большинства более сложных алгоритмов. Знакомство с ними поможет вам понять современные сложные алгоритмы.

Итак, в этой главе вас ждут:

- Алгоритмы сортировки.
- Алгоритмы поиска.
- Практическое применение.

Для начала рассмотрим некоторые алгоритмы сортировки.

АЛГОРИТМЫ СОРТИРОВКИ

В эпоху больших данных необходимы современные алгоритмы, чтобы эффективно сортировать и быстро находить элементы в сложных структурах. Выбор стратегии сортировки и поиска зависит от размера и типа данных. Хотя конечный результат будет одинаковым для различных алгоритмов, для эффективного решения реальной проблемы нужно подобрать наиболее подходящий вариант.

В данной главе представлены следующие алгоритмы сортировки:

- сортировка пузырьком (bubble sort);
- сортировка вставками (insertion sort);
- сортировка слиянием (merge sort);
- сортировка Шелла (Shell sort);
- сортировка выбором (selection sort).

Обмен значений переменных в Python

При реализации алгоритмов сортировки и поиска мы сталкиваемся с необходимостью обмена значений двух переменных между собой. В Python для этого есть простой способ:

```
var1 = 1
var2 = 2
var1,var2 = var2,var1
>>> print (var1,var2)
>>> 2 1
```

Давайте посмотрим, как это работает (рис. 3.1).

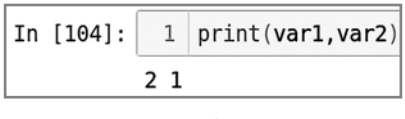


Рис. 3.1

Данный способ обмена значений используется во всех алгоритмах сортировки и поиска, о которых мы поговорим в этой главе.

Начнем с рассмотрения алгоритма сортировки пузырьком.

Сортировка пузырьком

Сортировка пузырьком (bubble sort) — это самый простой и медленный алгоритм сортировки. Он спроектирован так, что наибольшее значение перемещается вправо по списку на каждой итерации цикла. При наихудшем сценарии производительность этого алгоритма равна $O(n^2)$, поэтому его следует использовать только для небольших наборов данных.

Логика сортировки пузырьком

В основе сортировки пузырьком лежит ряд итераций, называемых *проходами* (passes). Для списка размера *N* нужно совершить *N* — 1 проходов. Рассмотрим подробно первую итерацию: проход 1.

Цель первого прохода — вывести наибольшее значение в конец списка. По мере выполнения алгоритма оно будет постепенно перемещаться вправо по списку.

В процессе сортировки значения соседних элементов сравниваются между собой попарно. Если в паре большее значение находится слева, происходит перестановка (обмен). Это продолжается до тех пор, пока мы не дойдем до конца списка. Работа алгоритма показана на следующей схеме (рис. 3.2).

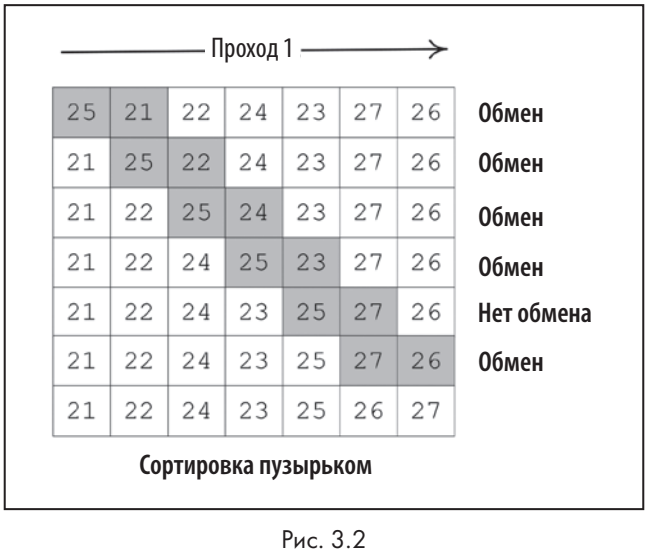


Рис. 3.2

Теперь посмотрим, как сортировка пузырьком реализуется в Python:

```
#Pass 1 of Bubble Sort
lastElementIndex = len(list)-1
print(0,list)
for idx in range(lastElementIndex):
                    if list[idx]>list[idx+1]:
list[idx],list[idx+1]=list[idx+1],list[idx]
print(idx+1,list)
```

Первый проход сортировки пузырьком на Python выглядит следующим образом (рис. 3.3).

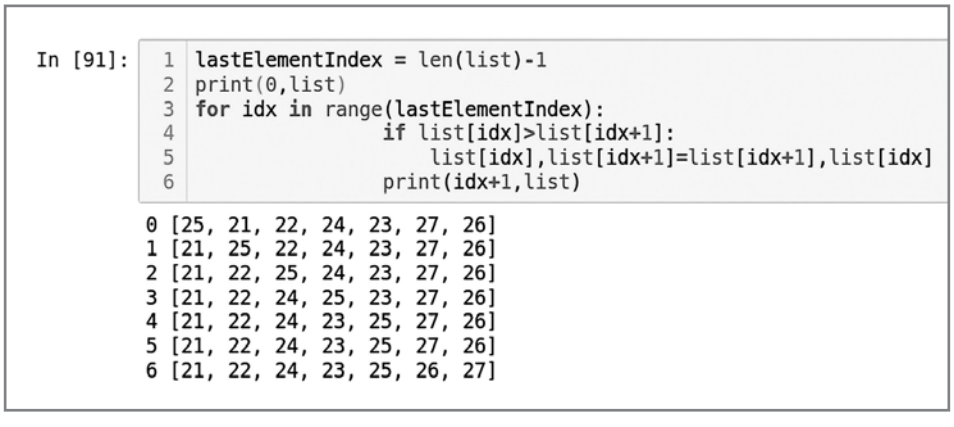


Рис. 3.3

После первой итерации алгоритма наибольшее значение оказывается в конце списка. Затем начинается следующий проход. Его цель — переместить второе по величине значение на предпоследнюю позицию в списке. Для этого алгоритм снова сравнивает значения соседних элементов и меняет их местами, если нужно. Последний элемент не затрагивается, поскольку уже был помещен в нужную позицию на первой итерации.

Проходы выполняются до тех пор, пока все элементы данных не будут расположены в порядке возрастания. Чтобы полностью отсортировать список, алгоритму потребуется $N - 1$ проходов для списка размером N. Полная реализация сортировки пузырьком на Python выглядит следующим образом (рис. 3.4).

Теперь рассмотрим производительность алгоритма `BubbleSort`.

```
In [5]: def BubbleSort(list):
        # Excahnge the elements to arrange in order
            lastElementIndex = len(list)-1
            for passNo in range(lastElementIndex,0,-1):
                for idx in range(passNo):
                    if list[idx]>list[idx+1]:
                        list[idx],list[idx+1]=list[idx+1],list[idx]
            return list
```

Рис. 3.4

Анализ производительности сортировки пузырьком

Легко увидеть, что сортировка пузырьком включает в себя два уровня циклов:

- *Внешний цикл*. Совокупность *проходов*. Например, первый проход — это первая итерация внешнего цикла.
- *Внутренний цикл*. Оставшиеся элементы в списке сортируются до тех пор, пока наибольшее значение не окажется справа. На первом проходе будет $N - 1$ сравнений, на втором — $N - 2$. На каждом последующем проходе количество сравнений будет уменьшаться на единицу.

Из-за двух уровней цикличности наихудшая сложность алгоритма равна $O(n^2)$.

Сортировка вставками

Основная идея сортировки вставками заключается в том, что на каждой итерации мы удаляем элемент из имеющейся у нас структуры данных, а затем вставляем его в нужную позицию. Именно поэтому алгоритм называется *сортировкой вставками* (insertion sort). На первой итерации мы сортируем два элемента данных. Затем мы расширяем выборку: берем третий элемент и находим для него позицию согласно его значению. Алгоритм выполняется до тех пор, пока все элементы не будут перемещены в правильное положение. Данный процесс показан на следующей диаграмме (рис. 3.5).

Алгоритм сортировки вставками на Python выглядит так:

```
def InsertionSort(list):
    for i in range(1, len(list)):
```

```
        j = i-1
        element_next = list[i]
        while (list[j] > element_next) and (j >= 0):
            list[j+1] = list[j]
            j=j-1
        list[j+1] = element_next
    return list
```

Рис. 3.5

Обратите внимание, что в основном цикле мы проходим по всему списку. В каждой итерации двумя соседними элементами являются `list[j]` (текущий элемент) и `list[i]` (следующий элемент).

В выражениях `list [j] > element_next>` и `j >= 0` мы сравниваем текущий элемент со следующим.

Давайте используем этот код для сортировки массива (рис. 3.6).

```
In [134]:  1 list = [25,26,22,24,27,23,21]

In [135]:  1 InsertionSort(list)
           2 print(list)

           [21, 22, 23, 24, 25, 26, 27]
```

Рис. 3.6

Рассмотрим производительность алгоритма сортировки вставками.

Из описания алгоритма очевидно, что если структура данных уже отсортирована, он выполняется очень быстро. Фактически в этом случае сортировка имеет линейное время выполнения, то есть $O(n)$. При наихудшем сценарии каждый внутренний цикл перемещает все элементы в списке. Если внутренний цикл мы обозначим i, наихудшая производительность алгоритма сортировки вставками определяется так:

$$w(N) = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2} = \frac{N^2 - N}{2};$$

$$w(N) \approx \frac{1}{2} N^2 = O(N^2).$$

Как правило, сортировка вставкой используется в работе с небольшими структурами данных. Для больших структур этот алгоритм не рекомендуется, поскольку обладает квадратичной средней производительностью.

Сортировка слиянием

Мы изучили два алгоритма сортировки: пузырьком и вставками. Производительность обоих будет лучше, если данные уже частично отсортированы. Третий алгоритм, с которым мы познакомимся, — *алгоритм сортировки слиянием* (merge sort), разработанный в 1940 году Джоном фон Нейманом. Отличительной чертой этого алгоритма является тот факт, что его производительность не зависит от упорядоченности входных данных. Подобно MapReduce и другим алгоритмам обработки больших данных, в его основе лежит стратегия «разделяй и властвуй». На этапе *разделения* алгоритм рекурсивно разбивает данные на две части до тех пор, пока размер данных не станет меньше определенного порогового значения. На этапе *слияния* алгоритм объединяет данные, пока мы не получим окончательный результат. Логика этого алгоритма объясняется на следующей диаграмме (рис. 3.7).

Давайте сначала рассмотрим псевдокод алгоритма сортировки слиянием:

```
mergeSort(list, start, end)
    if(start < end)
        midPoint = (end - start) / 2 + start
        mergeSort(list, start, midPoint)
        mergeSort(list, midPoint + 1, start)
        merge(list, start, midPoint, end)
```

Мы видим, что алгоритм состоит из трех шагов:

1. Разделение входного списка на две равные части.
2. Использование рекурсии для разделения до тех пор, пока длина каждого списка не будет равна 1.
3. Наконец, объединение отсортированных частей в список и вывод результата.

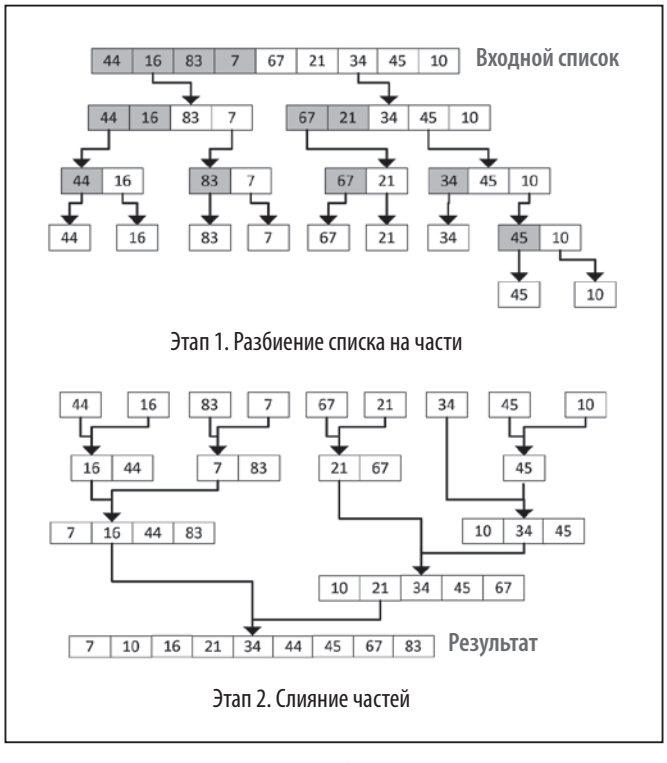


Рис. 3.7

Код реализации `MergeSort` показан здесь (рис. 3.8).

Когда этот код выполнится в среде Python, он сгенерирует следующий вывод (рис. 3.9).

Результатом работы кода является отсортированный список.

```
In [6]: def MergeSort(list):
            if len(list)>1:
                mid = len(list)//2 #splits list in half
                left = list[:mid]
                right = list[mid:]

                MergeSort(left) #repeats until length of each list is 1
                MergeSort(right)

                a = 0
                b = 0
                c = 0
                while a < len(left) and b < len(right):
                    if left[a] < right[b]:
                        list[c]=left[a]
                        a = a + 1
                    else:
                        list[c]=right[b]
                        b = b + 1
                    c = c + 1
                while a < len(left):
                    list[c]=left[a]
                    a = a + 1
                    c = c + 1

                while b < len(right):
                    list[c]=right[b]
                    b = b + 1
                    c = c + 1
            return list
```

Рис. 3.8

```
In [180]: 1 list = [44,16,83,7,67,21,34,45,10]
          2 MergeSort(list)
          3 print(list)
          4
          5

          [7, 10, 16, 21, 34, 44, 45, 67, 83]
```

Рис. 3.9

Сортировка Шелла

Алгоритм сортировки пузырьком сравнивает значения соседних элементов и меняет их местами, если они не стоят в нужном порядке. Если список частично отсортирован, мы получаем приемлемую производительность: сортировка завершается, как только в цикле прекращается обмен значениями.

В случае с полностью неотсортированным списком размера N алгоритм должен совершить $N - 1$ полных проходов.

Дональд Шелл предложил свой алгоритм сортировки, поставив под сомнение необходимость выбора соседних элементов для сравнения и обмена. Алгоритм получил название *сортировка Шелла* (Shell sort).

Попробуем разобраться в этой концепции.

На первом проходе мы используем элементы, расположенные с фиксированным промежутком (вместо ближайших соседей). В итоге получаем подсписок, состоящий из пар элементов данных. Процесс показан на диаграмме (рис. 3.10). На втором проходе алгоритм сортирует подсписки, содержащие по четыре элемента данных. В последующих проходах количество элементов в каждом подсписке увеличивается, а количество самих подсписков уменьшается. Когда остается только один подсписок, содержащий все элементы данных, сортировка завершена.

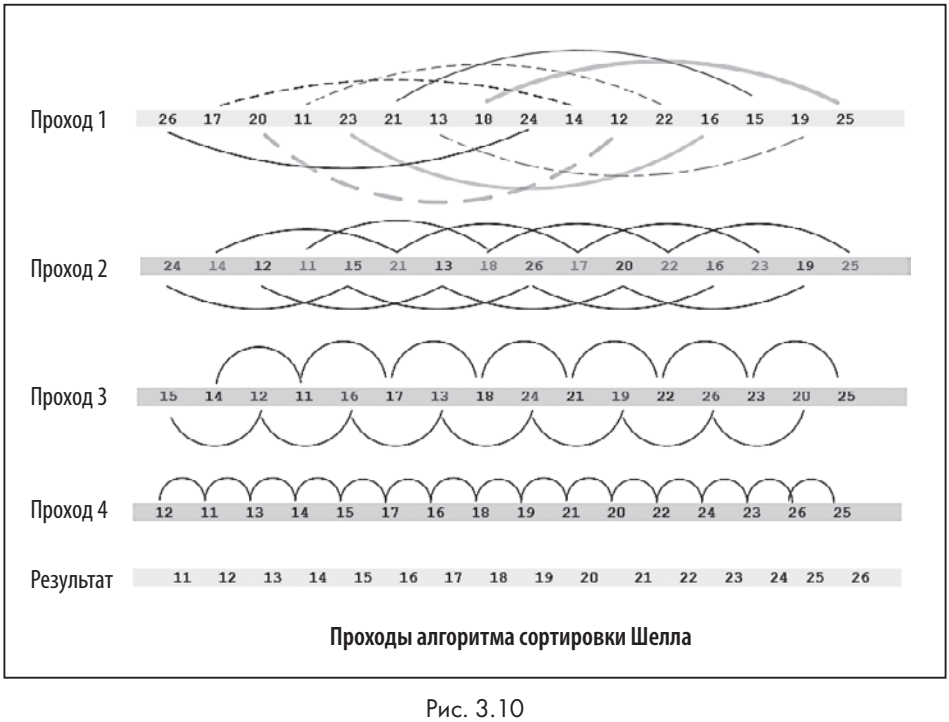


Рис. 3.10

На языке Python код для реализации алгоритма сортировки Шелла выглядит так:

```
def ShellSort(list):
    distance = len(list) // 2
    while distance > 0:
        for i in range(distance, len(list)):
            temp = input_list[i]
            j = i
# Сортировка подсписка для текущего значения дистанции
            while j >= distance and list[j - distance] > temp:
                list[j] = list[j - distance]
                j = j-distance
            list[j] = temp
# Уменьшаем расстояние до следующего элемента
        distance = distance//2
    return list
```

Представленный выше код можно использовать для сортировки списка следующим образом (рис. 3.11).

```
In [119]:  1 list = [26,17,20,11,23,21,13,18,24,14,12,22,16,15,19,25]

In [120]:  1 shellSort(list)
           2 print(list)

           [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]
```

Рис. 3.11

Обратите внимание, что вызов функции `ShellSort` привел к сортировке исходного входного массива.

Анализ производительности сортировки Шелла

Сортировка Шелла не предназначена для больших данных. Она используется для наборов данных среднего размера. Грубо говоря, алгоритм даст достаточно хорошую производительность при работе со списком, содержащим до 6000 элементов. Если данные частично упорядочены, производительность будет выше. В идеале список полностью отсортирован, тогда для проверки порядка потребуется только один проход через N элементов, что обеспечит наилучшую производительность $O(N)$.

Сортировка выбором

Как мы выяснили, сортировка пузырьком является одним из простейших алгоритмов сортировки. *Сортировка выбором* (selection sort) — это его улуч-

шенная версия. С ее помощью мы стараемся минимизировать общее количество обменов значений переменных. За каждый проход выполняется один обмен (сравните с $N - 1$ в случае сортировки пузырьком). Вместо того чтобы перемещать наибольшее значение маленькими шагами, мы ищем его на каждой итерации и ставим в конец списка. Это значит, что в результате первого прохода наибольшее значение окажется справа, а в результате второго прохода к нему переместится следующее по величине значение. По мере выполнения алгоритма последующие элементы будут перемещаться в нужное место согласно их значению. Последний элемент будет перемещен после $(N - 1)$-го прохода. Таким образом, сортировка выбором требует $N - 1$ проходов для сортировки N элементов (рис. 3.12).

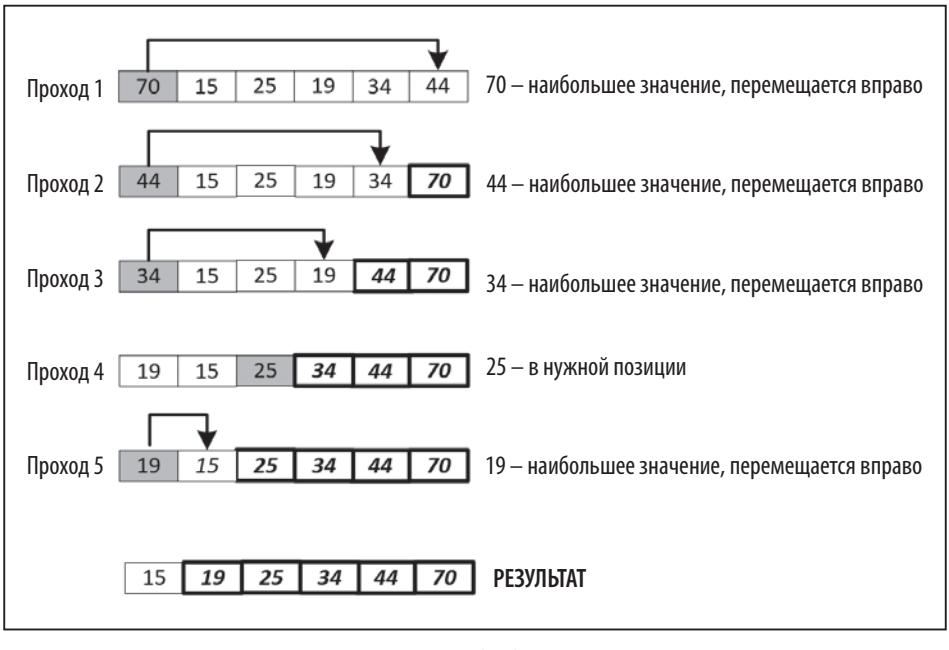


Рис. 3.12

Реализация сортировки выбором на Python:

```
def SelectionSort(list):
    for fill_slot in range(len(list) - 1, 0, -1):
        max_index = 0
        for location in range(1, fill_slot + 1):
            if list[location] > list[max_index]:
                max_index = location
        list[fill_slot],list[max_index] = list[max_index],list[fill_slot]
```

Когда алгоритм сортировки выбором выполнится, результат будет следующим (рис. 3.13).

```
In [202]:  1 list = [70,15,25,19,34,44]
           2 SelectionSort(list)
           3 print(list)

           [15, 19, 25, 34, 44, 70]
```

Рис. 3.13

На выходе мы получаем отсортированный список.

Анализ производительности сортировки выбором

Наихудшая производительность алгоритма сортировки выбором — $O(N^2)$, аналогично сортировке пузырьком. Поэтому его не следует использовать для обработки больших наборов данных. Тем не менее сортировка выбором — это более продуманный алгоритм, чем сортировка пузырьком, и его средняя производительность лучше из-за сокращения числа обменов значений.

Выбор алгоритма сортировки

Выбор правильного алгоритма сортировки зависит как от размера, так и от состояния имеющихся входных данных. Для небольших отсортированных списков использование продвинутого алгоритма приведет к ненужному усложнению кода при незначительном приросте производительности. Например, не следует использовать сортировку слиянием для небольших наборов данных. Сортировка пузырьком будет намного проще как для понимания, так для и реализации. Если данные частично отсортированы, мы можем воспользоваться этим преимуществом и применить сортировку вставкой. Для больших наборов данных лучше всего использовать алгоритм сортировки слиянием.

АЛГОРИТМЫ ПОИСКА

Качественный алгоритм поиска — это один из самых важных инструментов для работы со сложными структурами данных. Самый простой и не слишком эффективный подход заключается в поиске совпадений в каждой точке данных. Но по мере увеличения объема данных нам потребуются все более сложные алгоритмы поиска.

В этом разделе представлены следующие алгоритмы поиска:

- линейный поиск (linear search);
- бинарный поиск (binary search);
- интерполяционный поиск (interpolation search).

Давайте рассмотрим каждый из них более подробно.

Линейный поиск

Одна из простейших стратегий поиска данных состоит в том, чтобы просто перебрать все элементы в поисках цели. В каждой точке данных выполняется поиск совпадения. При обнаружении искомых данных алгоритм возвращает результат и выходит из цикла. В противном случае он продолжает поиск до тех пор, пока не достигнет конца структуры данных. Очевидным недостатком алгоритма является то, что он очень медленный, поскольку осуществляет исчерпывающий поиск. Преимущество же заключается в том, что данные не нужно сортировать, как того требуют другие алгоритмы, представленные в этой главе.

Давайте посмотрим на код для *линейного поиска*:

```
def LinearSearch(list, item):
    index = 0
    found = False
# Сравниваем значение с каждым элементом данных
    while index < len(list) and found is False:
        if list[index] == item:
            found = True
    else:
        index = index + 1
  return found
```

Давайте теперь посмотрим на вывод нашего кода (рис. 3.14).

```
1 list = [12, 33, 11, 99, 22, 55, 90]
2 print(LinearSearch(list, 12))
3 print(LinearSearch(list, 91))
```

```
True
False
```

Рис. 3.14

Обратите внимание, что запуск функции `linearSearch` возвращает значение `True`, если данные найдены.

Производительность линейного поиска

Как было сказано выше, перед нами простой алгоритм, который выполняет исчерпывающий поиск. Его наихудшая сложность — $O(N)$.

Бинарный поиск

Необходимым условием для работы *алгоритма бинарного поиска* является упорядоченность данных. Алгоритм итеративно делит список на две части и отслеживает самые низкие и самые высокие индексы, пока не найдет искомое значение:

```
def BinarySearch(list, item):
    first = 0
    last = len(list)-1
    found = False

while first<=last and not found:
    midpoint = (first + last)//2
    if list[midpoint] == item:
        found = True
    else:
        if item < list[midpoint]:
            last = midpoint-1
        else:
            first = midpoint+1
return found
```

Вывод выглядит следующим образом (рис. 3.15).

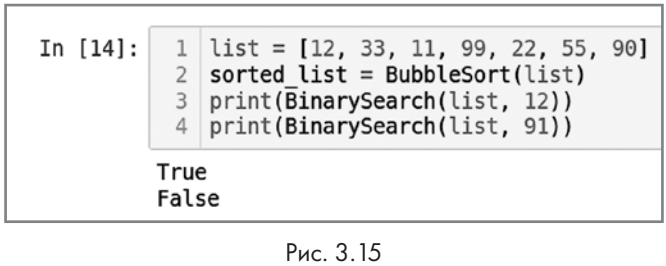


Рис. 3.15

Обратите внимание, что вызов функции `BinarySearch` вернет значение `True`, если значение найдено в списке ввода.

Производительность бинарного поиска

Бинарный (говорят также «двоичный») поиск назван так потому, что на каждой итерации алгоритм разделяет данные на две части. Если данные содержат N элементов, для итерации потребуется максимум $O(\log N)$ шагов. Это означает, что алгоритм имеет время выполнения $O(\log N)$.

Интерполяционный поиск

Бинарный поиск основан на логике, согласно которой он сосредотачивается на средней части данных. *Интерполяционный поиск* более сложен. Он использует целевое значение для оценки положения элемента в отсортированном массиве. Давайте попробуем понять это на примере. Предположим, мы хотим найти слово в словаре английского языка, например *river*. Мы будем использовать эту информацию для интерполяции и начнем поиск слов, начинающихся с *r*. В обобщенном виде интерполяционный поиск может быть реализован следующим образом:

```
def IntPolsearch(list,x ):
    idx0 = 0
    idxn = (len(list) - 1)
    found = False
    while idx0 <= idxn and x >= list[idx0] and x <= list[idxn]:
    # Ищем срединную точку
        mid = idx0 +int(((float(idxn - idx0)/( list[idxn] - list[idx0])) *
( x - list[idx0])))
# Сравниваем значение в средней точке со значением поиска
        if list[mid] == x:
           found = True
           return found
   if list[mid] < x:
           idx0 = mid + 1
return found
```

Вывод выглядит следующим образом (рис. 3.16).

```
In [16]:  1 list = [12, 33, 11, 99, 22, 55, 90]
          2 sorted_list = BubbleSort(list)
          3 print(IntPolsearch(list, 12))
          4 print(IntPolsearch(list,91))

          True
          False
```

Рис. 3.16

Обратите внимание, что перед использованием `IntPolsearch` массив необходимо упорядочить с помощью алгоритма сортировки.

Производительность интерполяционного поиска

Если данные распределены неравномерно, производительность алгоритма интерполяционного поиска будет низкой. Наихудшая производительность этого алгоритма — $O(N)$, наилучшая — $O(\log(\log N))$, если данные достаточно однородны.

ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ

Инструменты эффективного и точного поиска в структуре данных на практике имеют решающее значение. В зависимости от выбранного алгоритма поиска вам может потребоваться предварительная сортировка данных. Выбор подходящих алгоритмов сортировки и поиска зависит от типа и размера данных, а также от характера задачи, которую вы пытаетесь решить.

Используем изученные алгоритмы для решения следующей задачи: сопоставление обращения в иммиграционный департамент некой страны с архивными записями. Когда кто-то подает заявление на визу для въезда в страну, система пытается сопоставить его с уже имеющейся информацией. Если найдено хотя бы одно совпадение, то система вычисляет, сколько раз в прошлом этот человек получал одобрение или отказ на въезд. Если же совпадение не найдено, система классифицирует заявителя как нового и выдает ему новый идентификатор. Система должна осуществить поиск в архиве, обнаружить и идентифицировать кандидата. Эта информация крайне важна, потому что если человек ранее подавал заявку и она была отклонена, это может негативно повлиять на решение по новому обращению. Аналогично, если заявка была одобрена в прошлом, это увеличивает шансы на получение визы. Как правило, в архивной базе данных миллионы строк, и нам потребуется хорошо продуманное решение для достижения нужного результата.

Предположим, что архивная таблица в базе данных выглядит следующим образом (табл. 3.1).

В данной таблице первый столбец — `Персональный идентификатор`. Он присваивается каждому заявителю в архивной базе данных. Если в ней содержится 30 миллионов уникальных заявителей, то мы получим 30 миллионов индивидуальных персональных идентификаторов.

Таблица 3.1

Персональный идентификатор	Идентификатор заявления	Имя	Фамилия	Дата рождения	Решение	Дата принятия решения
45583	677862	Джон	Доу	2000-09-19	Одобрено	2018-08-07
54543	877653	Иксмен	Икс	1970-03-10	Отказано	2018-06-07
34332	344565	Агро	Вака	1973-02-15	Отказано	2018-05-05
45583	677864	Джон	Доу	2000-09-19	Одобрено	2018-03-02
22331	344553	Кэл	Сортс	1975-01-02	Одобрено	2018-04-15

Второй столбец — это `Идентификатор заявления`. Каждый такой идентификатор связан с уникальным заявлением в системе. В прошлом человек мог подавать заявки несколько раз. Это означает, что в базе данных у нас будет больше уникальных идентификаторов заявлений, чем персональных идентификаторов. Например, как видно из табл. 3.1, у Джона Доу только один персональный идентификатор, но два идентификатора заявлений.

В таблице представлен лишь образец архивного набора данных. Предположим, что наша структура данных содержит около одного миллиона строк с записями о заявителях за последние 10 лет. Новые кандидаты непрерывно добавляются со средней скоростью около двух претендентов в минуту. Для каждого кандидата нам необходимо выполнить следующие действия:

- Выдать заявителю новый идентификатор заявления.
- Проверить архивную базу данных на наличие совпадений.
- Если совпадение найдено, использовать уже имеющийся личный идентификатор заявителя. Нам также необходимо определить, сколько раз заявка была одобрена или отклонена.
- Если совпадений не найдено, мы выдаем этому человеку новый личный идентификатор.

Предположим, что к нам пришел новый заявитель со следующими учетными данными:

- Имя: Джон
- Фамилия: Доу
- Дата рождения: 2000-09-19

Как же нам разработать приложение, способное выполнять эффективный и экономичный поиск?

Одна из стратегий поиска нового заявления в базе данных выглядит следующим образом:

- Отсортировать базу данных по Дате рождения.
- Каждый раз, когда прибывает кандидат, присваивать ему новый Идентификатор заявления.
- Отобрать все записи, соответствующие данной Дате рождения. Это первичный поиск.
- В найденных записях выполнить вторичный поиск, используя Имя и Фамилию.
- Если совпадение найдено, использовать Персональный идентификатор в качестве ссылки на заявителя. Подсчитать количество одобрений и отказов.
- Если совпадений не найдено, выдать заявителю новый Персональный идентификатор.

Давайте попробуем подобрать правильный алгоритм сортировки архивной базы данных. Мы можем смело исключить сортировку пузырьком, так как объем данных огромен. Сортировка методом Шелла будет работать лучше, но только в том случае, если наши списки частично упорядочены. Лучше всего для этой задачи подходит сортировка слиянием.

Когда прибывает новый человек, нам необходимо отыскать его в базе данных. Поскольку данные уже отсортированы, можно использовать либо интерполяционный, либо бинарный поиск. Скорее всего, записи о кандидатах будут упорядочены согласно Дате рождения, поэтому мы можем без опаски использовать бинарный поиск.

Сначала мы осуществим поиск на основе Даты рождения, что даст нам соответствующий набор заявителей. Далее нам потребуется найти нужного человека в небольшой группе людей, у которых одна и та же дата рождения. Поскольку мы успешно сократили данные до небольшого подмножества, мы можем ис-

пользовать любой алгоритм поиска, включая сортировку пузырьком. Обратите внимание, что в данном случае мы немного упростили проблему вторичного поиска. Затем нам нужно подсчитать общее количество одобрений и отказов путем агрегирования результатов поиска, если найдено более одного совпадения.

На практике мы должны дополнительно использовать алгоритм нечеткого поиска, так как имя и фамилия могут быть записаны немного по-разному. Для этого нам может потребоваться какой-либо алгоритм определения сходства (когда точки данных, сходство которых превышает определенный порог, считаются одинаковыми).

РЕЗЮМЕ

В этой главе мы рассмотрели сильные и слабые стороны различных алгоритмов сортировки и поиска, количественно оценили их производительность и узнали, когда следует использовать каждый из них.

В следующей главе мы изучим динамические алгоритмы и разберем практический пример разработки алгоритма. Также мы подробно рассмотрим алгоритм ссылочного ранжирования PageRank и алгоритм линейного программирования.

4

Разработка алгоритмов

В этой главе представлены основные концепции проектирования алгоритмов. Мы изучим сильные и слабые стороны различных методов разработки и научимся создавать эффективные алгоритмы.

Мы обсудим разнообразие решений, доступных при разработке алгоритмов, а также важность определения характеристик конкретной задачи. Затем опробуем различные методы проектирования на примере знаменитой *задачи коммивояжера*. Наконец, обсудим области применения линейного программирования и узнаем, как его можно использовать для решения реальных задач.

К концу главы вы сможете понять основные концепции разработки эффективного алгоритма.

Итак, мы рассмотрим следующие темы:

- Различные подходы к разработке алгоритма.
- Компромиссы при выборе архитектуры алгоритма.
- Лучшие методы постановки реальной задачи.
- Решение прикладной задачи оптимизации.

Сначала рассмотрим основные концепции разработки алгоритма.

ЗНАКОМСТВО С ОСНОВНЫМИ КОНЦЕПЦИЯМИ РАЗРАБОТКИ АЛГОРИТМА

Как уже говорилось в главе 1, словарь American Heritage Dictionary определяет алгоритм следующим образом:

> *«Конечный набор однозначных инструкций, которые при заданном наборе начальных условий могут выполняться в заданной последовательности для достижения определенной цели и имеют определимый набор конечных условий».*

Разработка алгоритма заключается в том, чтобы придумать этот *«конечный набор однозначных инструкций»*, позволяющий наиболее эффективным способом добиться *«достижения определенной цели»*. В случае сложной прикладной задачи разработка алгоритма — дело весьма утомительное. Чтобы осуществить качественную реализацию, мы должны всесторонне изучить поставленную задачу. Прежде чем придумать, *как* это будет сделано (то есть разработать алгоритм), необходимо выяснить, *что* нужно сделать (то есть понять требования). Понимание задачи включает в себя как функциональные, так и нефункциональные требования. Давайте посмотрим, что это такое:

- Функциональные требования формально определяют интерфейсы ввода и вывода, а также связанные с ними функции. Они помогают понять, как данные будут обрабатываться и какие вычисления необходимо реализовать, чтобы получить результат.
- Нефункциональные требования определяют ожидания в отношении производительности и безопасности алгоритма.

Разработка алгоритма подразумевает удовлетворение как функциональных, так и нефункциональных требований наилучшим возможным способом, учитывая обстоятельства и доступные ресурсы.

Чтобы получить результат, соответствующий функциональным и нефункциональным требованиям, надо ответить на три вопроса, сформулированные в главе 1:

- *Вопрос 1*. Даст ли разработанный алгоритм ожидаемый результат?
- *Вопрос 2*. Является ли данный алгоритм оптимальным способом получения этого результата?
- *Вопрос 3*. Как алгоритм будет работать с большими наборами данных?

Далее мы подробно рассмотрим эти вопросы.

Вопрос 1. Даст ли разработанный алгоритм ожидаемый результат?

Алгоритм — это математическое решение реальной задачи. Чтобы быть полезным, он должен давать точные результаты. О проверке правильности алгоритма нужно думать заранее, и она должна быть заложена в его архитектуру. Прежде чем разрабатывать стратегию проверки алгоритма, нам нужно рассмотреть следующие два аспекта:

- *Определение истины* (truth). Для проверки алгоритма нам нужны некоторые известные правильные результаты для заданного набора входных данных. В контексте поставленной задачи такие результаты называются *истинами*. Пытаясь найти лучшее решение, мы последовательно совершенствуем наш алгоритм и используем *истину* в качестве ориентира.
- *Выбор метрик*. Кроме того, нужно решить, как именно мы собираемся количественно оценивать отклонение от определенной истины. Выбор правильных показателей (метрик) поможет точно оценить качество нашего алгоритма.

Например, для алгоритмов машинного обучения в качестве истины можно использовать существующие размеченные данные. Для количественной оценки отклонения от истины можно выбрать одну или несколько метрик, таких как *доля правильных ответов* (accuracy), *полнота* (recall) или *точность* (precision). Важно отметить, что в некоторых случаях результат не ограничен одним значением, а представляет собой диапазон для заданного набора входных данных. Во время разработки нашей целью будет итеративное улучшение алгоритма до тех пор, пока результат не окажется в пределах диапазона, указанного в требованиях.

Вопрос 2. Является ли данный алгоритм оптимальным способом получения результата?

Более развернуто этот вопрос звучит так:

> *Является ли алгоритм оптимальным решением и можем ли мы убедиться, что для этой задачи не существует другого решения, которое было бы лучше нашего?*

На первый взгляд, на этот вопрос ответить довольно просто. Однако в случае с некоторыми классами алгоритмов исследователи потратили десятилетия на безуспешные попытки подтвердить оптимальность конкретного решения.

Таким образом, сначала важно понять задачу и ее требования и оценить ресурсы, доступные для запуска алгоритма. Необходимо признать следующее утверждение:

> *Нужно ли стремиться найти оптимальное решение этой проблемы? Поиск и проверка оптимального решения — очень сложный и длительный процесс. Поэтому лучше всего подойдет работоспособное (приемлемое) решение, основанное на эвристике.*

Понимание задачи и ее сложности — важно, и это помогает оценить требования к ресурсам.

Прежде чем мы углубимся в детали, давайте определим два термина.

- *Полиномиальный алгоритм* (polynomial algorithm). Если алгоритм имеет временную сложность $O(n^k)$, мы называем его полиномиальным, где k — константа.
- *Сертификат* (certificate). Предлагаемый вариант решения, полученный по окончании итерации, называется *сертификатом*. По мере итеративного продвижения к решению конкретной задачи мы обычно генерируем серию сертификатов. Если решение стремится к сходимости, каждый сгенерированный сертификат будет лучше предыдущего. В какой-то момент, когда сертификат будет соответствовать требованиям, мы выберем его в качестве окончательного решения.

В главе 1 мы ввели понятие нотации «О-большое», которое можно использовать для анализа временной сложности алгоритма. В контексте этого анализа мы рассматриваем следующие временные интервалы:

- время, необходимое алгоритму для получения предлагаемого решения, называемого сертификатом: t_r;
- время, необходимое для проверки предлагаемого решения (сертификата): t_s.

Определение сложности задачи

На протяжении многих лет исследовательское сообщество делило задачи на различные категории в зависимости от их сложности. Прежде чем разрабатывать решение, имеет смысл охарактеризовать задачу. Как правило, задачи делятся на три типа:

- Тип 1. Задачи, для которых доказано существование полиномиального алгоритма.

- Тип 2. Задачи, для которых доказано, что они не могут быть решены с помощью полиномиального алгоритма.
- Тип 3. Задачи, для которых не найден полиномиальный алгоритм, но не доказано, что его не существует.

Рассмотрим различные классы задач:

- *Недетерминированные полиномиальные, NP* (non-deterministic polynomial). Чтобы задача была NP-задачей, она должна удовлетворять следующему условию:
  - гарантированно существует полиномиальный алгоритм, который может быть использован для проверки оптимальности варианта решения (сертификата).
- *Полиномиальные, P* (polynominal). Это типы задач, которые можно рассматривать как подмножество NP. В дополнение к выполнению условия задачи NP задачи P должны удовлетворять еще одному условию:
  - гарантированно существует по крайней мере один полиномиальный алгоритм, который может быть использован для их решения.

Взаимосвязь между классами P и NP показана на следующей диаграмме (рис. 4.1).

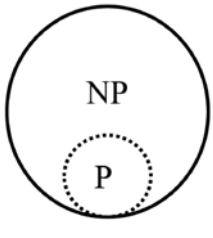


Рис. 4.1

Если задача принадлежит к классу NP, то принадлежит ли она также и к классу P? Это одна из величайших проблем в информатике, которая до сих пор остается нерешенной. Математический институт Клэя включил ее в число Задач тысячелетия. За решение этой проблемы предлагается 1 миллион долларов, так как оно оказало бы значительное влияние на такие области, как искусственный интеллект, криптография и теоретические компьютерные науки (рис. 4.2).

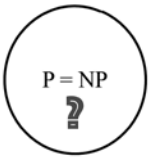


Рис. 4.2

Вернемся к списку классов задач.

- *NP-полные* (NP-complete). Данная категория содержит самые сложные задачи из всех NP. NP-полная задача удовлетворяет следующим двум условиям:
  - не существует известных полиномиальных алгоритмов для генерации сертификата;
  - существуют известные полиномиальные алгоритмы для проверки того, что предлагаемый сертификат является оптимальным.
- *NP-трудные* (NP-hard). Эта категория содержит задачи, которые по крайней мере так же сложны, как и любая задача категории NP; при этом они необязательно принадлежат категории NP.

Теперь попробуем нарисовать диаграмму, чтобы проиллюстрировать различные классы задач (рис. 4.3).

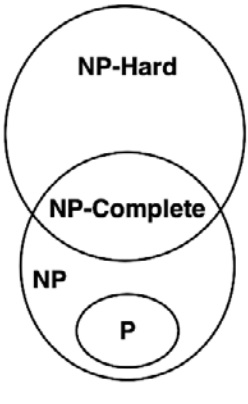


Рис. 4.3

Исследовательскому сообществу еще предстоит доказать, является ли P = NP. И хотя это еще не доказано, весьма вероятно, что P ≠ NP. В таком случае для

NP-полных задач не существует полиномиального решения. Предыдущая диаграмма основана на этом предположении.

Вопрос 3. Как алгоритм будет работать с большими наборами данных?

Для получения результата алгоритм обрабатывает данные определенным образом. Как правило, по мере увеличения объема данных требуется все больше времени для их обработки и вычисления результатов. Существуют наборы данных, представляющие особую сложность для инфраструктур и алгоритмов из-за их объема, разнообразия и скорости. Иногда их обозначают термином «*большие данные*» (big data). Хорошо разработанный алгоритм должен быть масштабируемым: по возможности эффективно работать, использовать доступные ресурсы и генерировать правильные результаты в разумные сроки. Архитектура алгоритма становится еще более важной при работе с большими данными. Чтобы количественно оценить масштабируемость алгоритма, необходимо учитывать следующие два аспекта:

- *Увеличение потребностей в ресурсах по мере роста объема входных данных*. Оценка такого требования называется анализом пространственной сложности.
- *Увеличение времени, затрачиваемого на выполнение, по мере роста объема входных данных*. Оценка данного параметра называется анализом временной сложности.

Мы живем в эпоху лавинообразного роста объемов данных. Термин «*большие данные*» стал общепринятым, поскольку он отражает размер и сложность данных, которые обычно требуется обрабатывать современным алгоритмам.

На этапе разработки и тестирования многие алгоритмы используют лишь небольшую выборку данных, поэтому важно учитывать аспект масштабируемости. В частности, необходимо тщательно проанализировать (протестировать или спрогнозировать) то, как на производительность алгоритма повлияет увеличение объема данных.

ПОНИМАНИЕ АЛГОРИТМИЧЕСКИХ СТРАТЕГИЙ

Хороший алгоритм максимально оптимизирует использование доступных ресурсов, по возможности разбивая задачу на более мелкие подзадачи. Существуют различные алгоритмические стратегии для разработки алгоритмов.

В этом разделе мы рассмотрим:

- стратегию «разделяй и властвуй»;
- стратегию динамического программирования;
- стратегию жадного алгоритма.

Стратегия «разделяй и властвуй»

Одна из стратегий состоит в том, чтобы разделить большую задачу на более мелкие, которые можно решать независимо друг от друга. Промежуточные решения, полученные при работе с этими подзадачами, затем объединяются в итоговое решение. Это и называется стратегией *«разделяй и властвуй»*.

Математически, если мы решаем задачу (P) с n входными данными, в рамках которой необходимо обработать набор данных d, мы разделяем ее на k подзадач, от P_1 до P_k. Каждая из подзадач будет обрабатывать свой раздел набора данных, d. Как правило, у нас будут подзадачи от P_1 до P_k, обрабатывающие данные от d_1 до d_k.

Обратимся к практическому примеру.

Практический пример — «разделяй и властвуй» применительно к Apache Spark

Apache Spark — это платформа с открытым исходным кодом, которую используют для решения сложных распределенных задач. В ней реализована стратегия «разделяй и властвуй». Задача делится на подзадачи, которые обрабатываются независимо. Мы продемонстрируем это на простом примере подсчета слов из списка.

Давайте предположим, что у нас имеется следующий список слов:

```
wordsList = [python, java, ottawa, news, java, ottawa]
```

Мы хотим подсчитать, сколько раз встречается каждое слово в этом списке, и для эффективного решения задачи применим стратегию «разделяй и властвуй».

Реализация принципа «разделяй и властвуй» показана на рис. 4.4.

На рисунке показаны следующие этапы, на которые делится задача:

1. *Разбиение (Splitting)*. Входные данные делятся на части (*сплиты*), которые могут обрабатываться независимо. Это называется *разбиением*. На рисунке мы видим три *сплита*.

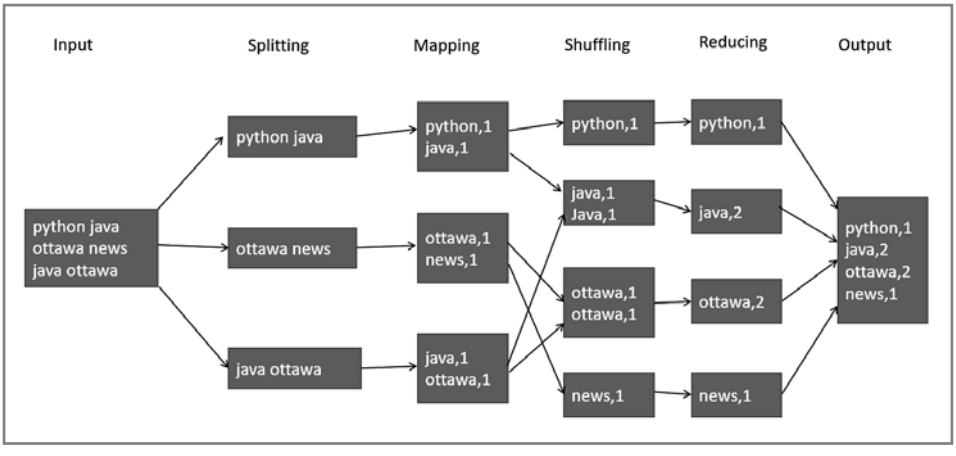


Рис. 4.4

2. *Маппинг (Mapping)*. Это любая операция, которую можно независимо применить к каждому сплиту. Как видно на диаграмме, во время маппинга каждое слово в сплите отображается в виде пары «ключ — значение». В нашем примере имеются три маппера, которые выполняются параллельно, в соответствии с тремя сплитами.
3. *Перетасовка (Shuffling)*. Это процесс объединения одинаковых ключей. Как только аналогичные ключи объединены, к их значениям можно применять функции агрегирования. Обратите внимание, что *перетасовка* — это ресурсоемкая операция, поскольку необходимо объединить аналогичные ключи, которые изначально могут быть разбросаны по сети.
4. *Сокращение (Reducing)*. Выполнение функции агрегирования значений одинаковых ключей называется *сокращением*. На рисунке нам нужно подсчитать количество одинаковых слов.

Давайте посмотрим, как мы можем реализовать решение в коде. Чтобы продемонстрировать стратегию «разделяй и властвуй», нам понадобится платформа распределенных вычислений. Для этого запустим Python на Apache Spark.

1. Чтобы использовать Apache Spark, создадим контекст выполнения Apache Spark:

```
import findspark
findspark.init()
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()
sc = spark.sparkContext
```

2. Теперь создадим демонстрационный список, содержащий несколько слов. Мы преобразуем этот список во встроенную распределенную структуру данных Spark, называемую *устойчивым распределенным набором данных* (RDD, Resilient Distributed Dataset):

```
wordsList = ['python', 'java', 'ottawa', 'ottawa', 'java','news']
wordsRDD = sc.parallelize(wordsList, 4)
# Напечатать тип wordsRDD
print (wordsRDD.collect())
```

3. Далее используем функцию `map()`, чтобы преобразовать слова в пары «ключ — значение» (рис. 4.5).

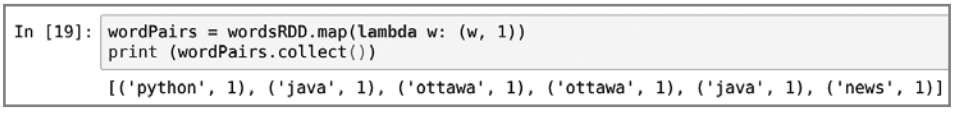


Рис. 4.5

4. Наконец, используем функцию `reduce()` для агрегирования и получения конечного результата (рис. 4.6).

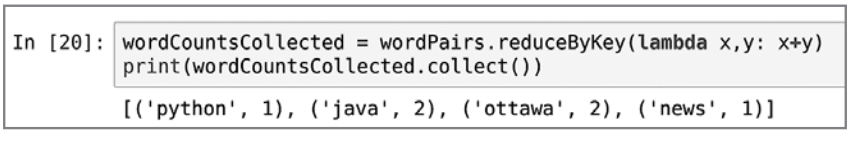


Рис. 4.6

Данный код демонстрирует, как мы можем использовать стратегию «разделяй и властвуй» для подсчета количества одинаковых слов.

Современные инфраструктуры облачных вычислений, такие как Microsoft Azure, Amazon Web Services и Google Cloud, обеспечивают масштабируемость за счет прямой или косвенной реализации стратегии «разделяй и властвуй».

Стратегия динамического программирования

Динамическое программирование — это стратегия, предложенная в 1950-х годах Ричардом Беллманом для оптимизации некоторых классов алгоритмов. В ее основе лежит интеллектуальный механизм кэширования, который пытается

повторно использовать результаты тяжелых вычислений. Этот интеллектуальный механизм кэширования называется *мемоизацией* (memorization).

Динамическое программирование увеличивает производительность, когда задачу можно разделить на подзадачи. В подзадачах могут выполняться одни и те же вычисления. Идея состоит в том, чтобы выполнить какое-то вычисление единожды (что является трудоемким шагом), а затем повторно использовать его для других подзадач. Это возможно при помощи мемоизации, что особенно полезно при решении рекурсивных задач, которые могут многократно получать одни и те же входные данные.

Жадные алгоритмы

Прежде чем мы погрузимся в эту тему, дадим определения двум понятиям:

- *Алгоритмические затраты* (algorithmic overheads). Всякий раз, когда мы пытаемся найти оптимальное решение какой-то задачи, это занимает время. По мере того как задачи становятся все более и более сложными, время поиска оптимального решения также возрастает. Обозначим алгоритмические затраты как Ω_i.
- *Дельта от оптимального* (delta from optimal). Для некоторой задачи оптимизации существует оптимальное решение. Как правило, мы итеративно оптимизируем решение, используя выбранный алгоритм. Для любой задачи всегда существует идеальное решение, называемое *оптимальным решением*. Как уже говорилось, если классифицировать задачу, может оказаться, что оптимальное решение неизвестно или что для его расчета и проверки потребуется неоправданно много времени. Предполагая, что оптимальное решение известно, разницу между ним и текущим решением на i-й итерации мы называем *дельтой от оптимального* и представляем как Δ_i.

Для решения комплексных задач нам доступны две возможные стратегии:

- Стратегия 1. Потратить больше времени на поиск решения, наиболее близкого к оптимальному, чтобы Δ_i оказалось как можно меньше.
- Стратегия 2. Минимизировать алгоритмические затраты Ω_i и на скорую руку выбрать работоспособное решение.

Жадные алгоритмы основаны на второй стратегии, в которой мы не прилагаем усилий для поиска наилучшего решения и вместо этого минимизируем затраты алгоритма.

Жадный алгоритм — это быстрая и простая стратегия поиска глобального оптимального значения для многоступенчатых задач. Мы выбираем локальные оптимальные значения, но не проверяем, являются ли они при этом глобально оптимальными. Обычно жадный алгоритм не приводит к значению, которое можно считать глобально оптимальным (если только нам не повезет). Однако поиск глобального оптимального значения — непростая задача. Следовательно, жадный алгоритм работает быстрее по сравнению с алгоритмами «разделяй и властвуй» и алгоритмами динамического программирования.

Как правило, жадный алгоритм состоит из следующих шагов:

1. Предположим, что у нас есть набор данных *D*, из которого мы выберем элемент *k*.
2. Предположим, что вариантом решения (или сертификатом) выступает *S*. Рассмотрим возможность включения *k* в решение *S*. Если включение возможно, то решением будет *Union* (*S, e*).
3. Повторяем процесс до тех пор, пока *S* не заполнится или *D* не окажется исчерпанным.

ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ — РЕШЕНИЕ ЗАДАЧИ КОММИВОЯЖЕРА

Давайте рассмотрим упоминаемую ранее *задачу коммивояжера* (TSP — travelling salesman problem). Это хорошо известная задача, придуманная в качестве своего рода вызова в 1930-х годах. Она относится к классу NP-трудных задач. Для начала мы можем случайным образом сгенерировать маршрут, отвечающий условию посещения всех городов, не заботясь об оптимальном решении. Затем будем работать над улучшением решения на каждой итерации. Каждый маршрут, сгенерированный в итерации, называется вариантом решения (или сертификатом). Доказательство того, что сертификат является оптимальным, требует экспоненциально возрастающего количества времени. Вместо этого будем использовать различные решения на основе эвристики, которые генерируют маршруты, близкие к оптимальным, но не оптимальные.

Чтобы выполнить работу, коммивояжеру необходимо посетить определенный список городов (табл. 4.1).

Таблица 4.1

ВВОД	Список из n городов (обозначается как V) и расстояний между каждой парой городов, $d\ ij$ ($1 \le i, j \le n$)
ВЫВОД	Кратчайший маршрут, который включает в себя каждый город ровно один раз и завершается в исходном городе

Обратите внимание:

- Расстояния между городами в списке известны.
- Каждый город в данном списке необходимо посетить *только* один раз.

Получится ли у нас составить план поездки? Каким будет оптимальное решение, позволяющее свести к минимуму общее расстояние, пройденное коммивояжером?

В табл. 4.2 приведены расстояния между пятью канадскими городами, которые мы можем использовать для TSP.

Таблица 4.2

	Оттава	**Монреаль**	**Кингстон**	**Торонто**	**Садбери**
Оттава	—	199	196	450	484
Монреаль	199	—	287	542	680
Кингстон	196	287	—	263	634
Торонто	450	542	263	—	400
Садбери	484	680	634	400	—

Обратите внимание, что цель состоит в разработке маршрута, который начинается и заканчивается в исходном городе. Например, типичным маршрутом может быть Оттава — Садбери — Монреаль — Кингстон — Торонто — Оттава с общей длиной пути *484 + 680 + 287 + 263 + 450 = 2164*. Является ли он маршрутом, при котором продавец преодолеет минимальное расстояние? Каким будет оптимальное решение, позволяющее свести к минимуму общее расстояние, пройденное коммивояжером? Я предлагаю вам подумать над этим и подсчитать.

Использование стратегии полного перебора

Первое решение, которое приходит на ум для задачи TSP, — это использовать *полный перебор* (brute force, иначе называемый «метод грубой силы») и попытаться найти кратчайший путь, при котором коммивояжер посетит каждый город ровно один раз и вернется в исходный город. Итак, стратегия полного перебора работает следующим образом:

1. Рассчитать все возможные маршруты.
2. Выбрать среди них кратчайший маршрут.

Проблема в том, что для n числа городов существует $(n - 1)!$ возможных маршрутов. Это означает, что пять городов дадут $4! = 24$ маршрута и мы выберем самый короткий из них. Очевидно, что такой метод сработает лишь потому, что у нас не так много городов. По мере увеличения числа городов полный перебор превращается в неустойчивую стратегию из-за большого количества перестановок, генерируемых этим методом.

Давайте посмотрим, как можно реализовать стратегию полного перебора в Python.

Прежде всего обратим внимание, что маршрут {1, 2, 3} представляет собой маршрут из города 1 в город 2 и город 3. Общее расстояние — это все расстояние, пройденное за маршрут. Мы будем считать, что расстояние между городами является кратчайшим, то есть евклидовым.

Определим три служебные функции:

- `distance_points`. Вычисляет абсолютное расстояние между двумя точками;
- `distance_tour`. Вычисляет общее расстояние, которое коммивояжер должен преодолеть на данном маршруте;
- `generate_cities`. Случайным образом генерирует набор из n городов, расположенных в прямоугольнике шириной `500` и высотой `300`.

Давайте рассмотрим следующий код:

```
import random
from itertools import permutations
alltours = permutations

def distance_tour(aTour):
    return sum(distance_points(aTour[i - 1], aTour[i])
               for i in range(len(aTour)))
```

```
aCity = complex

def distance_points(first, second): return abs(first - second)

def generate_cities (number_of_cities):
    seed=111;width=500;height=300
    random.seed((number_of_cities, seed))
    return frozenset(aCity(random.randint(1, width), random.randint(1,
height))
                     for c in range(number_of_cities))
```

В предыдущем коде мы применили `alltours` из функции `permutations` пакета `itertools`. Мы также представили расстояние комплексным числом. Это означает следующее:

- вычисление расстояния между двумя городами *a* и *b* сводится к `distance(a,b)`;
- мы можем создать *n*-число городов, просто вызвав `generate_cities(n)`.

Теперь давайте определим функцию `brute_force`, которая генерирует все возможные маршруты через эти города.

Как только она их сгенерирует, будет выбран маршрут с наименьшим расстоянием:

```
def brute_force(cities):
    "Создать все возможные маршруты по городам и выбрать самый короткий."
    return shortest_tour(alltours(cities))

def shortest_tour(tours): return min(tours, key=distance_tour)
```

Далее определим служебные функции для визуализации:

- `visualize_tour`. Отображает все города и связи на конкретном маршруте. Она также показывает город, с которого начался маршрут.
- `visualize_segment`. Используется функцией `visualize_tour` для отображения городов и связей в сегменте.

Взгляните на код:

```
%matplotlib inline
import matplotlib.pyplot as plt
def visualize_tour(tour, style='bo-'):
    if len(tour) > 1000: plt.figure(figsize=(15, 10))
    start = tour[0:1]
    visualize_segment(tour + start, style)
```

```
    visualize_segment(start, 'rD')
def visualize_segment (segment, style='bo-'):
    plt.plot([X(c) for c in segment], [Y(c) for c in segment], style,
clip_on=False)
    plt.axis('scaled')
    plt.axis('off')
def X(city): "X axis"; return city.real
def Y(city): "Y axis"; return city.imag
```

Реализуем функцию `tsp()`, которая выполняет следующие действия:

1. Генерирует маршрут на основе алгоритма и количества запрошенных городов.
2. Вычисляет время, необходимое для выполнения алгоритма.
3. Строит график.

Как только `tsp()` определена, мы можем использовать ее для создания маршрута (рис. 4.7).

```
[ ] from time import clock
    from collections import Counter
    def tsp(algorithm, cities):
        t0   = clock()
        tour = algorithm(cities)
        t1   = clock()
        assert Counter(tour) == Counter(cities) # Every city appears exactly once in tour
        visualize_tour(tour)
        print("{}: {} cities ⇒ tour length {:.0f} (in {:.3f} sec)".format(
              name(algorithm), len(tour), distance_tour(tour), t1 - t0))

    def name(algorithm): return algorithm.__name__.replace('_tsp', '')
```

```
[ ] tsp(brute_force, generate_cities(10))

    brute_force: 10 cities ⇒ tour length 1218 (in 10.962 sec)
```

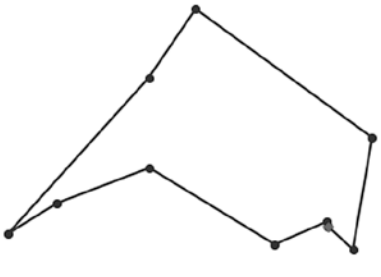


Рис. 4.7

Обратите внимание, что мы используем функцию `tsp()` для создания маршрута по 10 городам. Поскольку $n = 10$, то мы получим (10-1)! = 362 880 возможных перестановок. Если n увеличивается, количество перестановок резко возрастает и стратегию полного перебора использовать уже нельзя.

Использование жадного алгоритма

Если для решения TSP мы используем жадный алгоритм, то на каждом шаге можем выбрать город, который выглядит приемлемым вариантом, вместо того чтобы искать идеальное решение. Поэтому всякий раз, когда нам нужно выбрать город, мы просто выбираем ближайший, не утруждая себя проверкой того, является ли этот выбор глобально оптимальным.

Стратегия жадного алгоритма проста:

1. Начать с любого города.
2. На каждом этапе продолжать строить маршрут, перемещаясь в следующий ближайший непосещенный город.
3. Повторить шаг 2.

Определим функцию с именем `greedy_algorithm`, которая реализует эту логику:

```
def greedy_algorithm(cities, start=None):
    C = start or first(cities)
    tour = [C]
    unvisited = set(cities - {C})
    while unvisited:
        C = nearest_neighbor(C, unvisited)
        tour.append(C)
        unvisited.remove(C)
    return tour

def first(collection): return next(iter(collection))

def nearest_neighbor(A, cities):
    return min(cities, key=lambda C: distance_points(C, A))
```

Теперь используем `greedy_algorithm` для создания маршрута по 2000 городам (рис. 4.8).

Обратите внимание, что для создания маршрута по 2000 городам потребовалось всего 0,514 секунды. Если бы мы использовали стратегию полного перебора, мы получили бы $(2000 - 1)!$ перестановок, то есть почти бесконечное количество.

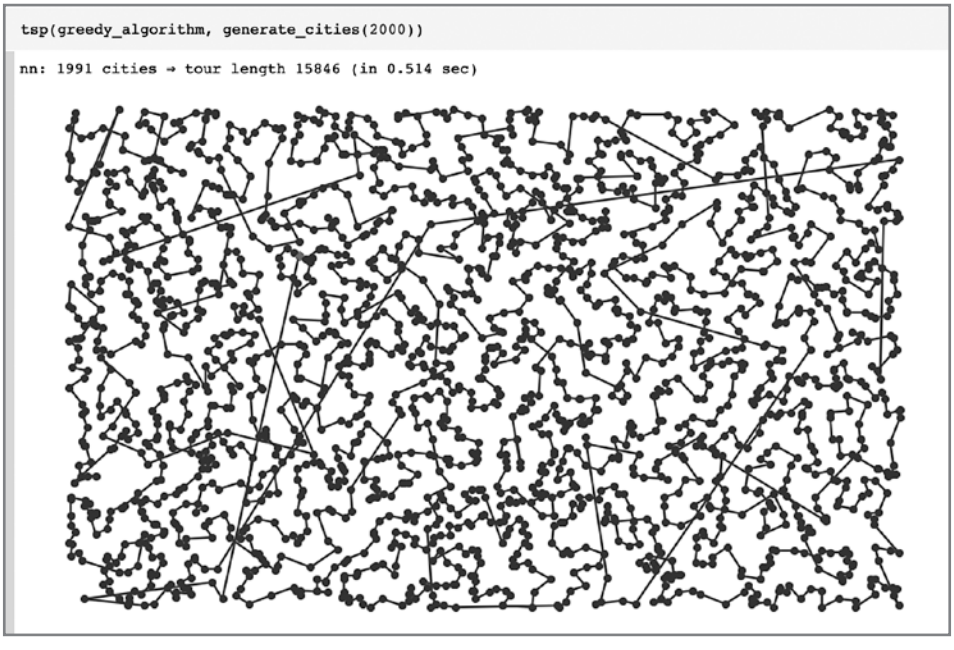


Рис. 4.8

Жадный алгоритм основан на эвристике, и нет никаких доказательств того, что решение будет оптимальным.

Теперь ознакомимся с архитектурой алгоритма PageRank.

АЛГОРИТМ PAGERANK

В качестве практического примера рассмотрим алгоритм *PageRank*, который изначально использовался Google для ранжирования результатов поиска по пользовательскому запросу. Он генерирует число, которое количественно определяет релевантность результатов поиска по запросу пользователя. Алгоритм был разработан в Стэнфорде в конце 1990-х годов двумя аспирантами, Ларри Пейджем и Сергеем Брином, которые позже основали Google.

Алгоритм PageRank был назван в честь Ларри Пейджа (Page), который создал его вместе с Сергеем Брином во время учебы в Стэнфордском университете.

Прежде всего необходимо сформулировать задачу, для которой PageRank был изначально разработан.

Постановка задачи

Всякий раз, когда мы вводим запрос в поисковую систему в интернете, мы получаем большое количество результатов. Чтобы сделать результаты полезными для конечного пользователя, необходимо отсортировать веб-страницы по ряду критериев. Отображаемые результаты ранжируются согласно критериям, заданным базовым алгоритмом.

Реализация алгоритма PageRank

Главная составляющая алгоритма PageRank — поиск наилучшего способа измерения релевантности каждой страницы, возвращаемой в результатах запроса. Для вычисления числа от `0` до `1`, которое количественно отражает релевантность конкретной страницы, алгоритм учитывает два компонента информации:

- *Информация, непосредственно относящаяся к пользовательскому запросу*. Этот компонент оценивает в контексте запроса, насколько релевантно содержимое веб-страницы. Содержание страницы напрямую зависит от автора страницы.
- *Информация, которая не относится непосредственно к пользовательскому запросу*. Данный компонент пытается количественно оценить релевантность каждой веб-страницы в контексте ее ссылок, просмотров и соседних страниц. Этот компонент трудно рассчитать, так как веб-страницы неоднородны и трудно разработать общие для всей сети критерии.

Чтобы реализовать алгоритм PageRank на Python, сначала импортируем необходимые библиотеки:

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline
```

В демонстрационных целях мы будем анализировать только пять веб-страниц в сети. Назовем этот набор страниц `myPages`, и пусть они находятся в сети с именем `myWeb`:

```
myWeb = nx.DiGraph()
myPages = range(1,5)
```

Соединим их случайным образом, чтобы смоделировать реальную сеть:

```
connections = [(1,3),(2,1),(2,3),(3,1),(3,2),(3,4),(4,5),(5,1),(5,4)]
myWeb.add_nodes_from(myPages)
myWeb.add_edges_from(connections)
```

Теперь давайте построим следующий график:

```
pos=nx.shell_layout(myWeb)
nx.draw(myWeb, pos, arrows=True, with_labels=True)
plt.show()
```

Это создаст визуальное представление нашей сети (рис. 4.9).

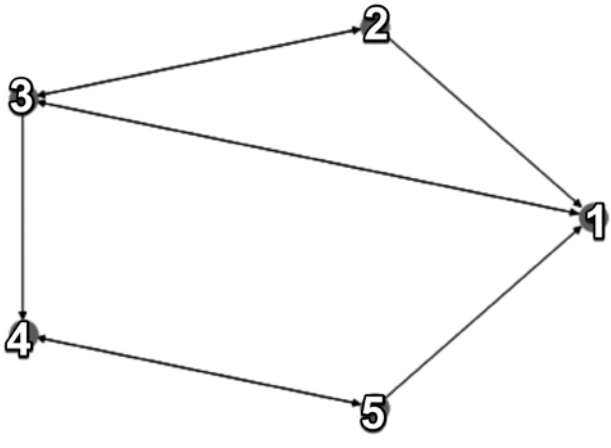


Рис. 4.9

В алгоритме PageRank шаблоны веб-страницы содержатся в матрице, называемой *матрицей переходов* (transition matrix). Существуют алгоритмы, которые постоянно обновляют матрицу переходов, фиксируя непрерывно меняющееся состояние сети. Размер матрицы переходов равен $n \times n$, где n — количество вершин. Числа в матрице обозначают вероятность того, что далее посетитель перейдет на эту страницу по предыдущей ссылке.

На графике выше показана наша статическая сеть. Зададим функцию для создания матрицы переходов (рис. 4.10).

```
def createPageRank(aGraph):
  nodes_set = len(aGraph)
  M = nx.to_numpy_matrix(aGraph)
  outwards = np.squeeze(np.asarray(np.sum(M, axis=1)))
  prob_outwards = np.array(
   [1.0/count
    if count>0 else 0.0 for count in outwards])
  G = np.asarray(np.multiply(M.T, prob_outwards))
  p = np.ones(nodes_set) / float(nodes_set)
  if np.min(np.sum(G,axis=0)) < 1.0:
    print ('WARN: G is substochastic')
  return G, p
```

Рис. 4.10

Обратите внимание, что эта функция вернет *G*, которая и представляет матрицу переходов для нашего графика.

Давайте сгенерируем матрицу переходов (рис. 4.11).

```
[6] G, p = createPageRank(myWeb)
    print (G)
```

1	2	3	4	5	
[0.	0.5	0.33333333	0.	0.5	]
[0.	0.	0.33333333	0.	0.	]
[1.	0.5	0.	0.	0.	]
[0.	0.	0.33333333	0.	0.5	]
[0.	0.	0.	1.	0.	]]

Рис. 4.11

Обратите внимание, что матрица переходов имеет размерность *5 × 5*. Каждый столбец соответствует вершине графа. Например, столбец *2* описывает вторую вершину. Существует вероятность 0.5, что посетитель перейдет с вершины *2* на вершину *1* или вершину *3*. Обратите внимание, что диагональ матрицы переходов содержит `0`, так как на нашем графе нет ссылки от вершины к самой себе. В реальной сети такая ссылка вполне может быть.

Обратите внимание, что матрица переходов является разреженной матрицей. По мере увеличения числа вершин большинство ее значений будет равно `0`.

ЗНАКОМСТВО С ЛИНЕЙНЫМ ПРОГРАММИРОВАНИЕМ

Базовый алгоритм линейного программирования был разработан Джорджем Данцигом в Калифорнийском университете в Беркли в начале 1940-х годов. Данциг использовал эту концепцию для экспериментов с логистическим планированием материально-технического снабжения войск во время работы в ВВС США. В конце Второй мировой войны Данциг начал работать на Пентагон и превратил свой алгоритм в метод линейного программирования. Он использовался для планирования боевых действий.

Сегодня линейное программирование применяется в решении важных практических задач, связанных с минимизацией или максимизацией переменной на основе определенных ограничений. Вот некоторые примеры таких задач:

- минимизация времени на ремонт автомобиля в автосервисе на основе имеющихся ресурсов;
- распределение доступных ресурсов в вычислительной среде для минимизации времени отклика;
- максимизация прибыли компании на основе оптимального распределения ресурсов внутри компании.

Формулировка задачи линейного программирования

Условия для применения линейного программирования следующие:

- Задачу можно сформулировать с помощью набора уравнений.
- Переменные, используемые в уравнении, должны быть линейными.

Целевая функция

Обратите внимание, что цель приведенных в примере задач заключается в минимизации или максимизации переменной. Эта цель математически сформулирована как линейная функция нескольких переменных и называется *целевой функцией*. Цель задачи линейного программирования состоит в том, чтобы минимизировать или максимизировать целевую функцию, оставаясь в пределах заданных ограничений.

Определение ограничений

При попытке минимизировать или максимизировать что-либо на практике необходимо принимать во внимание определенные ограничения. Например, если мы хотим затратить минимум времени на ремонт автомобиля, мы должны учитывать ограниченный штат доступных механиков. Указание каждого ограничения с помощью линейного уравнения очень важно при формулировании задачи линейного программирования.

ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ — ПЛАНИРОВАНИЕ ПРОИЗВОДСТВА С ПОМОЩЬЮ ЛИНЕЙНОГО ПРОГРАММИРОВАНИЯ

Рассмотрим практический пример использования линейного программирования для решения реальной задачи. Предположим, что нам требуется максимизировать прибыль современной фабрики, которая производит два типа роботов:

- *Улучшенная модель (A)*. Робот содержит полный набор функций. Производство каждой единицы улучшенной модели приносит прибыль в размере 4200 долларов.
- *Базовая модель (B)*. Она обеспечивает только базовую функциональность. Производство каждой единицы базовой модели приводит к прибыли в размере 2800 долларов.

Для изготовления каждого робота необходимы три типа специалистов и определенное количество дней. Эти данные представлены в следующей таблице (табл. 4.3).

Таблица 4.3

Тип робота	**Техник**	**Специалист по ИИ**	**Инженер**
Робот A: улучшенная модель	3 дня	4 дня	4 дня
Робот B: базовая модель	2 дня	3 дня	3 дня

Фабрика работает по 30-дневным циклам. Один специалист по ИИ доступен в течение 30 дней в цикле. Каждый из двух инженеров возьмет по 8 выходных

дней в течение 30 дней. Таким образом, инженер доступен только в течение 22 дней в цикле. Имеется один техник, доступный в течение 20 дней в 30-дневном цикле.

В табл. 4.4 показано количество сотрудников, работающих на нашей фабрике.

Таблица 4.4

	Техник	**Специалист по ИИ**	**Инженер**
Количество человек	1	1	2
Общее количество дней в цикле	1 × 20 = 20 дней	1 × 30 = 30 дней	2 × 22 = 44 дня

Смоделировать это можно следующим образом:

- Максимальная прибыль = 4200A + 2800B.
- Результат зависит от следующих условий:
  - A ≥ 0: Количество произведенных улучшенных роботов может быть `0` или более.
  - B ≥ 0: Количество произведенных базовых роботов может быть `0` или более.
  - 3A + 2B ≤ 20: Ограничения, связанные с доступностью техника.
  - 4A+3B ≤ 30: Ограничения, связанные с доступностью специалиста по ИИ.
  - 4A+ 3B ≤ 44: Ограничения, связанные с доступностью инженеров.

В первую очередь импортируем библиотеку Python под названием `pulp`, предназначенную для реализации линейного программирования:

```
import pulp
```

Затем для создания экземпляра класса для решения задачи вызовем функцию этой библиотеки — `LpProblem`. Назовем экземпляр `Profit maximizing problem` (задачей максимизации прибыли):

```
# Создание экземпляра класса для решения задчи
model = pulp.LpProblem("Profit maximising problem", pulp.LpMaximize)
```

Далее определим две линейные переменные, A и B. Переменная A представляет собой количество произведенных улучшенных роботов, а переменная B — количество произведенных базовых роботов:

```
A = pulp.LpVariable('A', lowBound=0, cat='Integer')
B = pulp.LpVariable('B', lowBound=0, cat='Integer')
```

Определим целевую функцию и ограничения следующим образом:

```
# Целевая функция
model += 5000 * A + 2500 * B, "Profit"

# Ограничения
model += 3 * A + 2 * B <= 20
model += 4 * A + 3 * B <= 30
model += 4 * A + 3 * B <= 44
```

Для генерации решения используем функцию solve:

```
# Решение задачи
model.solve()
pulp.LpStatus[model.status]
```

Затем выведем значения A и B и значение целевой функции (рис. 4.12).

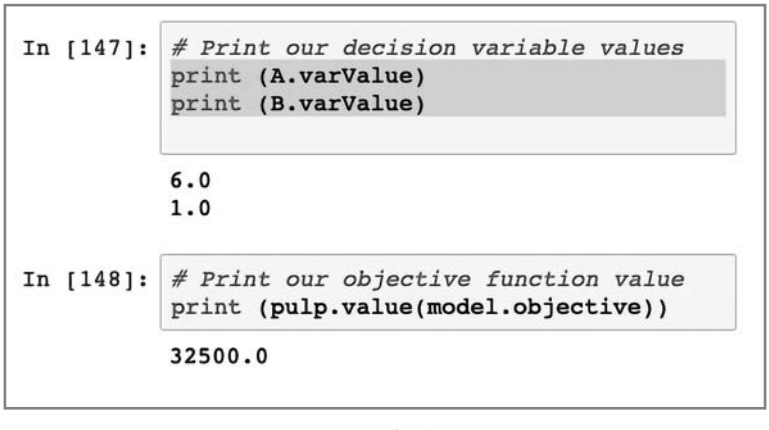


Рис. 4.12

Линейное программирование широко применяется в обрабатывающей промышленности при подсчете объема продукции, необходимой для оптимизации использования доступных ресурсов.

Вот мы и добрались до конца главы! Давайте подведем итоги.

РЕЗЮМЕ

В этой главе мы обсудили различные подходы к разработке алгоритма. Мы рассмотрели компромиссы, связанные с выбором правильной архитектуры алгоритма, и лучшие методы постановки реальной задачи. Мы также узнали, как решить реальную задачу оптимизации. Эти знания могут быть использованы для реализации хорошо проработанных алгоритмов.

В следующей главе мы сосредоточимся на графовых алгоритмах. Мы изучим различные способы представления графов и методы установления отношений соседства между элементами данных для выполнения исследования. Наконец, мы изучим оптимальные способы поиска информации на графах.

5

Графовые алгоритмы

Существует класс вычислительных задач, решить которые проще всего, представив их в виде графов и применив алгоритмы, называемые *графовыми* (graph algorithms). Например, графовые алгоритмы используются для эффективного поиска значения в графическом представлении данных. Прежде всего алгоритм определяет структуру графа, затем ищет правильную стратегию прохода по его *ребрам* с целью получить данные, хранящиеся в *вершинах*. Поскольку графовым алгоритмам для работы необходимо искать значения, в основе их разработки лежат эффективные стратегии поиска. Использование графовых алгоритмов — один из наилучших способов поиска информации в сложных структурах данных, связанных между собой значимыми отношениями. Сегодня, в эпоху больших данных, социальных сетей и распределенных данных, такие методы становятся все более важными и нужными.

Эту главу мы начнем с базовых концепций графовых алгоритмов. Затем познакомимся с основами сетевого анализа, а также с различными методами, которые можно использовать для обхода графов. Наконец, рассмотрим пример использования графовых алгоритмов для обнаружения мошенничества.

Итак, мы обсудим:

- Различные способы представления графов.
- Введение в теорию сетевого анализа.
- Понятие обхода графа.
- Практический пример — анализ мошенничества.
- Методы создания окрестностей в пространстве задачи.

К концу главы вы будете хорошо понимать, что такое графы и как с ними работать. Графовые алгоритмы позволят вам отображать взаимосвязанные структуры данных, извлекать информацию из элементов, связанных прямыми или косвенными отношениями, а также решать ряд сложных реальных задач.

ПРЕДСТАВЛЕНИЕ ГРАФОВ

Граф — это структура, которая отображает данные в виде *вершин* и *ребер*. Граф может быть представлен в виде `aGraph` = $(\mathcal{V}, \mathcal{E})$, где $\mathcal{V}$ — набор вершин, а $\mathcal{E}$ — набор ребер. Обратите внимание, что `aGraph` имеет $|\mathcal{V}|$ вершин и $|\mathcal{E}|$ ребер.

Вершина, $v \in \mathcal{V}$, представляет собой объект реального мира, например человека, компьютер, или деятельность.

Ребро, $v \in \mathcal{E}$, соединяет две вершины в сеть:

$$e(v_1, v_2) \mid e \in \mathcal{E} \,\&\, v_i \in \mathcal{V}.$$

Предыдущее уравнение указывает, что в графе все ребра принадлежат множеству $\mathcal{E}$ и все вершины принадлежат множеству $\mathcal{V}$.

Ребро соединяет две вершины и таким образом отображает связь между ними. Вот несколько примеров взаимосвязей, которые можно представить в виде графа:

- Дружба между людьми.
- Человек, связанный с другом в LinkedIn.
- Физическое соединение двух узлов в кластере.
- Человек, присутствующий на научной конференции.

В этой главе для представления графов мы будем использовать библиотеку Python `networkx`. Давайте попробуем создать простой граф на Python, используя `networtx`. Для начала создадим пустой граф, `aGraph`, без вершин или узлов:

```
import networkx as nx
G = nx.Graph()
```

Теперь добавим одну вершину:

```
G.add_node("Mike")
```

Мы можем добавить группу вершин, используя список:

```
G.add_nodes_from(["Amine", "Wassim", "Nick"])
```

Также можно добавить одно ребро между существующими вершинами:

```
G.add_edge("Mike", "Amine")
```

Давайте теперь выведем на экран ребра и вершины (рис. 5.1).

```
In [5]: 1 list(G.nodes)
Out[5]: ['Mike', 'Amine', 'Wassim', 'Nick']

In [6]: 1 list(G.edges)
Out[6]: [('Mike', 'Amine')]
```

Рис. 5.1

Обратите внимание, что если мы добавляем ребро, это также приводит и к добавлению связанных вершин (если их еще нет), как показано в данном случае:

```
G.add_edge("Amine","Imran")
```

Если мы выведем список узлов, то получим следующий результат (рис. 5.2).

```
In [9]: 1 list(G.edges)
Out[9]: [('Mike', 'Amine'), ('Amine', 'Imran')]
```

Рис. 5.2

Следует заметить, что запрос на добавление уже существующей вершины игнорируется без каких-либо сообщений. Запрос игнорируется или принимается в зависимости от типа созданного нами графа.

Типы графов

Графы можно разделить на четыре типа:

- Неориентированные графы (undirected graphs).
- Ориентированные графы (directed graphs).

- Неориентированные мультиграфы (undirected multigraphs).
- Ориентированные мультиграфы (directed multigraphs).

Давайте подробно рассмотрим каждый из них.

Неориентированные графы

В большинстве случаев отношения, представленные вершинами графа, можно рассматривать как неориентированные. В таких отношениях отсутствует иерархия. При этом ребра называются *неориентированными ребрами* (undirected edges), а полученный граф — *неориентированным графом* (undirected graph) (рис. 5.3).

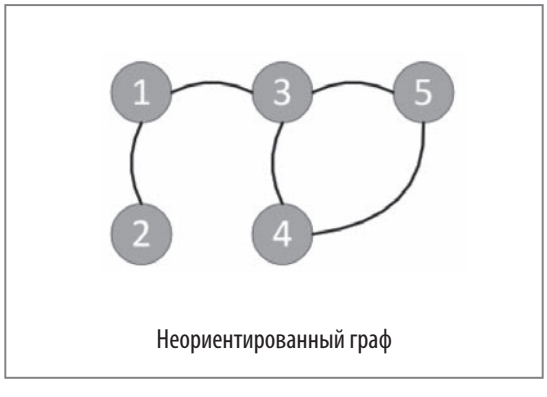


Рис. 5.3

Ниже приведены примеры неориентированных отношений:

- Майк и Амина (Майк и Амина знают друг друга).
- Вершина *A* и вершина *B* соединены (одноранговая связь).

Ориентированные графы

Граф, в котором связь между вершинами имеет некоторое направление, называется *ориентированным графом* (directed graph) (рис. 5.4).

Ниже приведены примеры ориентированных отношений:

- Майк и его дом (Майк живет в доме, но дом не живет в Майке).
- Джон руководит Полом (Джон — менеджер Пола).

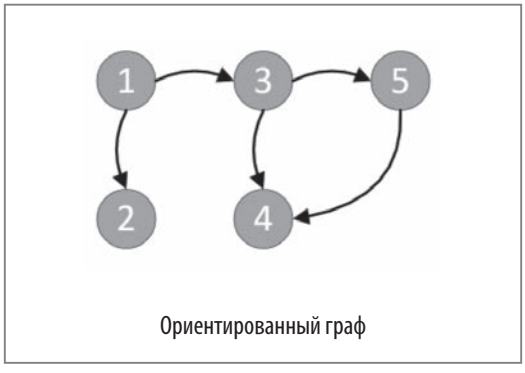


Рис. 5.4

Неориентированные мультиграфы

Иногда вершины имеют между собой более одного типа отношений. В таком случае может существовать более одного ребра, соединяющего одни и те же две вершины. Графы, в которых допускаются несколько параллельных ребер между двумя вершинами, называются *мультиграфами*. Мы должны четко указывать, является ли конкретный граф мультиграфом. Параллельные ребра представляют различные типы связей между вершинами.

Мультиграф представлен на рис. 5.5.

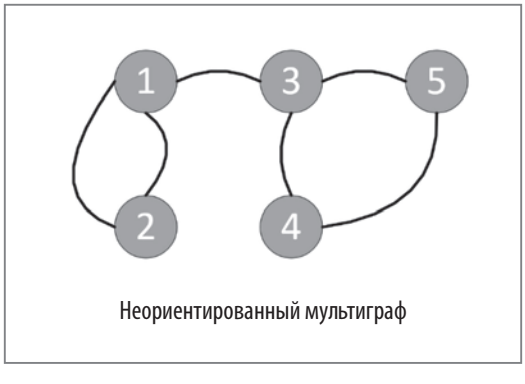


Рис. 5.5

Пример мультиграфа: Майк и Джон являются не только одноклассниками, но и коллегами.

Ориентированные мультиграфы

Если в мультиграфе существует направленная связь между узлами, мы называем его *ориентированным мультиграфом* (directed multigraphs) (рис. 5.6).

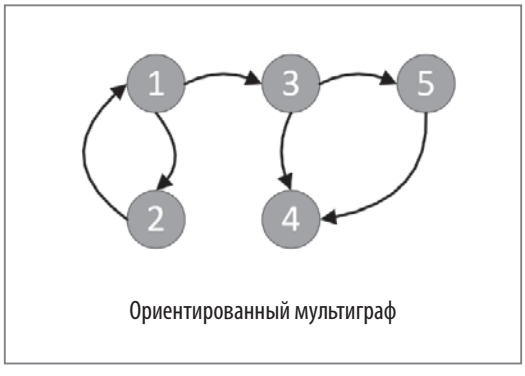


Рис. 5.6

Пример ориентированного мультиграфа: Майк отчитывается перед Джоном в офисе, а Джон учит Майка языку программирования Python.

Особые типы ребер

Ребра соединяют разные вершины графа вместе и отображают их взаимосвязь между собой. В дополнение к обычным ребрам допустимы следующие специальные типы:

- *Петля* (self-edge). Иногда вершина имеет отношение к самой себе. Например, Джон переводит деньги со своего рабочего счета на личный. Подобные особые отношения могут быть представлены ребром, ориентированным само на себя.
- *Гиперребро* (hyperedge). Иногда ребро объединяет более одной вершины. Ребро, соединяющее более одной вершины для представления соответствующих отношений, называется гиперребром. Например, предположим, что все трое — Майк, Джон и Сара — работают над одним проектом.

Граф, имеющий одно или несколько гиперребер, называется *гиперграфом* (hypergraph).

На рис. 5.7 представлены графы с петлей и гиперребром.

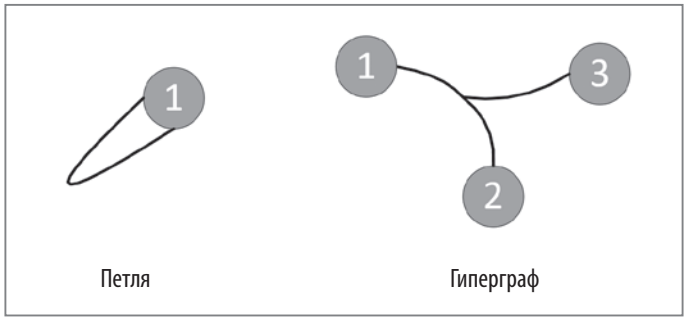


Рис. 5.7

Обратите внимание, что граф может содержать как петли, так и гиперребра одновременно.

Эгоцентрические сети

Непосредственная окрестность вершины, *m*, может содержать важную информацию, достаточную для исчерпывающего анализа. Эго-сеть основана на этой идее. Эго-сеть определенной вершины *m* состоит из всех вершин, напрямую соединенных с *m*, и самой вершины *m*. Вершина *m* называется *эго*, а вершины, с которыми она соединена, называются *альтерами*.

Эго-сеть конкретной вершины, 3, показана на рис 5.8.

Обратите внимание, что данная эго-сеть представляет собой окрестность первой степени. Эта концепция может быть распространена на окрестности *n*-степени, включающие в себя все вершины, удаленные от конкретной вершины на *n* шагов.

Анализ социальных сетей

Анализ социальных сетей (social network analysis, SNA) — одно из важных применений теории графов. Анализ сетевого графа считается анализом социальной сети, если применимо следующее:

- Вершины графа представляют людей.
- Ребра между ними представляют социальные отношения, такие как дружба, общее хобби, родство, сексуальные отношения, антипатии и т. д.

- Бизнес-вопрос, на который мы пытаемся ответить с помощью анализа графов, имеет некоторый выраженный социальный аспект.

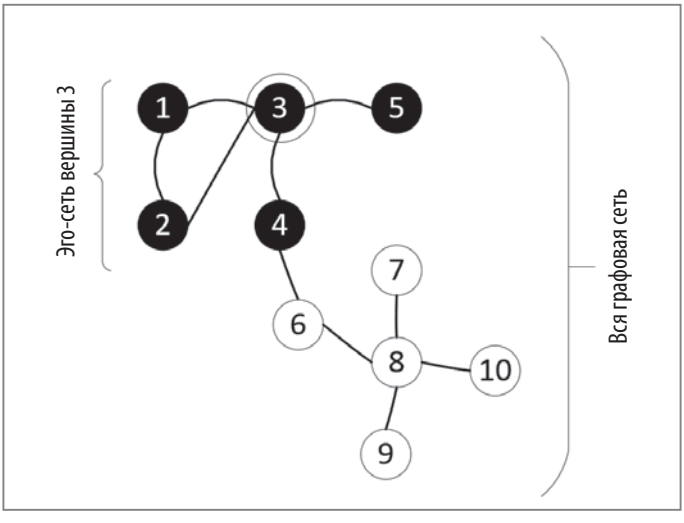


Рис. 5.8

Поведение человека отражено в SNA и всегда должно учитываться при анализе. Представляя социальные отношения в виде графа, SNA помогает разобраться, как люди взаимодействуют, что, в свою очередь, позволяет понять их мотивацию.

Анализируя деятельность человека с учетом его отношений с окружающими людьми, можно прийти к интересным, а иногда и удивительным открытиям. Альтернативные подходы, основанные на анализе отдельно взятого человека с учетом только его должностных функций, могут дать лишь ограниченное представление о нем.

SNA можно использовать для следующих целей:

- понимание действий пользователей на платформах социальных сетей, таких как Facebook, Twitter или LinkedIn;
- изучение природы мошенничества;
- исследование преступного поведения в обществе.

Компания LinkedIn внесла большой вклад в исследования и разработку новых методов, связанных с SNA. Фактически LinkedIn можно считать пионером многих алгоритмов в этой области.

SNA — благодаря распределенной и взаимосвязанной архитектуре социальных сетей — является одним из наиболее мощных примеров использования теории графов. Другой способ работы с графом — рассматривать его как сеть и применять алгоритмы, разработанные для сетей. Это целая область, называемая *теорией сетевого анализа*, и ее мы обсудим далее.

ВВЕДЕНИЕ В ТЕОРИЮ СЕТЕВОГО АНАЛИЗА

Мы знаем, что взаимосвязанные данные могут быть представлены в виде сети. В теории сетевого анализа подробно изучают методологии, разработанные для исследования и анализа таких данных. В этом разделе мы рассмотрим некоторые важные аспекты теории сетевого анализа.

Обратите внимание, что вершина — базовая единица сети. Сеть — это совокупность взаимосвязанных вершин (различных исследуемых объектов), а каждая линия соединения отображает отношения между ними. Чтобы решить задачу, нужно количественно оценить полезность и важность вершины в сети. Для этого существует несколько методов.

Рассмотрим ряд важных концепций, используемых в теории сетевого анализа.

Кратчайший путь

Путь представляет собой маршрут между выбранной начальной и конечной вершинами; это набор вершин, *p*, соединяющих начальную вершину с конечной. Ни одна вершина в *p* не повторяется.

Длина пути равна количеству входящих в его состав ребер. Из всех вариантов путь с наименьшей длиной называется *кратчайшим*. Расчет такого пути широко используется в алгоритмах теории графов, но не всегда прост для вычисления. Существуют различные алгоритмы поиска кратчайшего пути между начальным и конечным узлами. Один из самых популярных — *алгоритм Дейкстры*, опубликованный в конце 1950-х годов. Его можно использовать в *системе глобального позиционирования, GPS* (Global Positioning System), для расчета минимального расстояния между источником и пунктом назначения. Алгоритм Дейкстры также применяется в алгоритмах сетевой маршрутизации.

Между Google и Apple не прекращается битва за разработку наилучшего алгоритма поиска кратчайшего расстояния в картах Google и Apple Maps. Задача состоит в том, чтобы сделать алгоритм настолько быстрым, чтобы кратчайший путь вычислялся за считанные секунды.

Позже в этой главе мы обсудим *алгоритм поиска в ширину, BFS* (breadth-first search algorithm), который может быть преобразован в алгоритм Дейкстры. BFS предполагает одинаковые затраты на каждый вариант обхода графа. Для алгоритма Дейкстры затраты могут быть разными, и это необходимо учесть, чтобы превратить BFS в алгоритм Дейкстры.

Алгоритм Дейкстры вычисляет кратчайший путь от одной из вершин. Если мы хотим найти все пары кратчайших путей, нужно использовать *алгоритм Флойда — Уоршелла* (Floyd-Warshall algorithm).

Создание окрестностей

Поиск стратегий для создания окрестностей вокруг выбранных вершин имеет решающее значение для графовых алгоритмов. Методологии создания окрестностей основаны на выборе прямых связей с интересующей вершиной. Одной из таких методологий является стратегия k-го порядка, которая выбирает вершины, находящиеся на расстоянии k переходов от интересующей вершины.

Давайте рассмотрим различные критерии создания окрестностей.

Треугольники (triangles)

В теории графов поиск вершин, которые тесно связаны друг с другом, крайне важен для анализа. Один из методов состоит в том, чтобы попытаться идентифицировать в сети *треугольники*, представляющие собой подграф, состоящий из трех непосредственно соединенных между собой вершин.

Давайте обсудим практический пример: выявление мошенничества (он будет подробнее рассмотрен в конце этой главы). Если эго-сеть узла m состоит из трех вершин, включая вершину m, то эта эго-сеть является треугольником. Вершина m будет эго, а две связанные вершины будут альтерами, скажем, вершина A и вершина B. Если оба альтера являются известными мошенниками, то мы можем с уверенностью объявить вершину m мошенником. Если только один из альтеров замешан в мошенничестве, мы не сможем представить убедительные доказательства, но нам следует продолжить поиски.

Плотность (density)

Начнем с определения полносвязной сети. *Полносвязной сетью* (fully connected network) мы называем граф, где каждая вершина напрямую связана с каждой другой вершиной.

Если у нас имеется полносвязная сеть N, то количество ребер в сети может быть представлено следующим образом:

$$Edges_{total}\binom{N}{2}=\frac{N(N-1)}{2}.$$

Здесь в дело вступает *плотность* (density). Плотность измеряет отношение наблюдаемых ребер к максимальному количеству ребер, если $Edges_{Observed}$ — это количество ребер, которые мы хотим рассмотреть. Ее можно сформулировать так:

$$density=\frac{Edges_{observed}}{Edges_{total}}.$$

Обратите внимание, что для треугольника плотность сети равна `1` и она представляет собой максимально возможную связную сеть.

Показатели центральности

Существуют различные *показатели центральности* (centrality measures) конкретной вершины в графе или подграфе. С их помощью можно, например, количественно оценивать важность человека в социальной сети или важность здания для города.

В анализе графов широко используются следующие показатели центральности:

- степень (degree);
- посредничество (betweenness);
- близость (closeness);
- влиятельность (Eigenvector).

Давайте разберем их подробно.

Степень

Количество ребер, соединенных с определенной вершиной, называется ее *степенью* (degree). Степень может указывать на то, насколько хорошо подключена конкретная вершина и какова ее способность быстро распространять сообщение по сети.

Давайте рассмотрим aGraph = ($\mathcal{V}$, $\mathcal{E}$), где $\mathcal{V}$ представляет собой набор вершин, а $\mathcal{E}$ — набор ребер. Напомним, что aGraph содержит $|\mathcal{V}|$ вершин и $|\mathcal{E}|$ ребер. Если мы разделим степень узла на ($|\mathcal{V}|-1$), то получим *степень центральности* (degree centrality):

$$C_{DC_a} = \frac{deg(a)}{|\mathcal{V}|-1}.$$

Разберем конкретный пример. Взгляните на следующий граф (рис. 5.9).

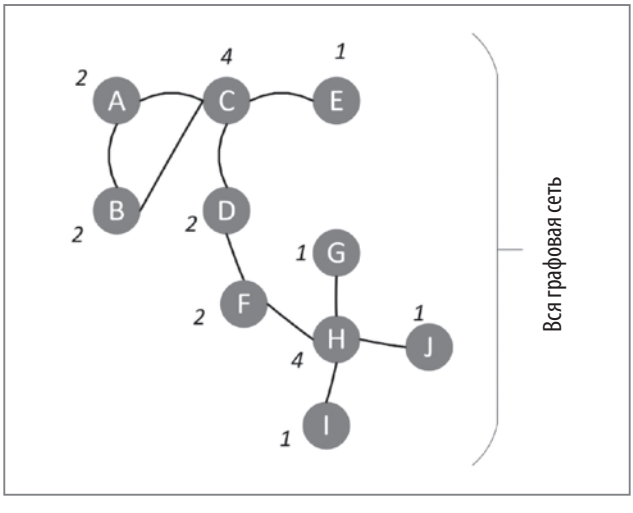


Рис. 5.9

На этом графе вершина *C* имеет степень *4*. Ее степень центральности может быть рассчитана следующим образом:

$$C_{DC_c} = \frac{deg(c)}{|\mathcal{V}|-1} = \frac{4}{10-1} = 0.44.$$

Посредничество

Посредничество, или *промежуточность* (betweenness),— это показатель центральности в графе. В контексте социальных сетей она позволяет количественно оценить вероятность того, что человек является частью взаимодействия в подгруппе. Для компьютерной сети посредничество количественно определяет негативное влияние отказа ребра на связь между вершинами графа.

Чтобы вычислить посредничество вершины a в определенном aGraph = ($\mathcal{V}$, $\mathcal{E}$), выполните следующие действия:

1. Вычислите кратчайший путь между каждой парой вершин в aGraph. Давайте представим это с помощью $n_{shortest_{Total}}$.
2. От $n_{shortest_{Total}}$ подсчитайте количество кратчайших путей, проходящих через вершину a. Давайте представим это с помощью $n_{shortest_a}$.
3. Вычислите посредничество $C_{betweenness_a} = \frac{n_{shortest_a}}{n_{shortest_{Total}}}$.

Удаленность и близость

Представим себе граф g. *Удаленность* (fairness) вершины a в графе g определяется как сумма расстояний от вершины a до других вершин. Обратите внимание, что центральность конкретной вершины количественно определяет ее общее расстояние от всех остальных вершин.

Противоположностью удаленности выступает *близость* (closeness).

Влиятельность

Влиятельность, или центральность собственного вектора (Eigenvector centrality), оценивает все вершины графа, определяя их важность в сети. Эта оценка — показатель связи конкретной вершины с другими важными вершинами во всей сети. Когда в Google создавали *алгоритм PageRank*, который присваивает оценку каждой веб-странице в интернете (чтобы отразить ее важность), идею они взяли из показателя влиятельности.

Вычисление показателей центральности с помощью Python

Давайте создадим сеть, после чего попытаемся рассчитать ее показатели центральности. Процесс проиллюстрирован в следующем блоке кода:

```
import networkx as nx
import matplotlib.pyplot as plt
vertices = range(1,10)
edges = [(7,2), (2,3), (7,4), (4,5), (7,3), (7,5), (1,6),(1,7),(2,8),(2,9)]
          G = nx.Graph()
G.add_nodes_from(vertices)
G.add_edges_from(edges)
nx.draw(G, with_labels=True,node_color='y',node_size=800)
```

Граф, созданный этим кодом, выглядит следующим образом (рис. 5.10).

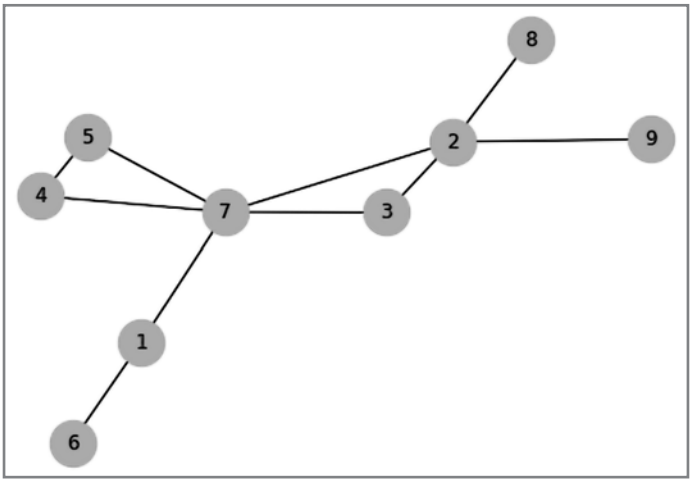


Рис. 5.10

На данный момент мы разобрали несколько показателей центральности. Давайте рассчитаем их для примера выше (рис. 5.11).

Обратите внимание, что показатели центральности должны дать оценку центральности конкретной вершины в графе или подграфе. Глядя на граф, кажется, что вершина *7* имеет наиболее центральное расположение (рис. 5.10). Вершина *7* имеет самые высокие значения по всем четырем показателям центральности, что отражает ее важность в данном контексте.

Теперь давайте разберемся, как мы можем извлекать из графов информацию. Графы — это сложные структуры данных, содержащие большое количество информации, хранящейся как в вершинах, так и в ребрах. Рассмотрим ряд стратегий, которые можно использовать для эффективного передвижения по графам, чтобы получать из них информацию, отвечающую запросам.

ПОНЯТИЕ ОБХОДА ГРАФА

Чтобы использовать графы, необходимо уметь извлекать из них информацию. *Обход графа* (graph traversal) — стратегия, обеспечивающая упорядоченное посещение каждой вершины и ребра. Необходимо добиться того, чтобы каждая

```
In [8]:  1 nx.degree_centrality(G)

Out[8]: {1: 0.25,
         2: 0.5,
         3: 0.25,
         4: 0.25,
         5: 0.25,
         6: 0.125,
         7: 0.625,
         8: 0.125,
         9: 0.125}

In [9]:  1 nx.betweenness_centrality(G)

Out[9]: {1: 0.25,
         2: 0.46428571428571425,
         3: 0.0,
         4: 0.0,
         5: 0.0,
         6: 0.0,
         7: 0.7142857142857142,
         8: 0.0,
         9: 0.0}

In [10]:  1 nx.closeness_centrality(G)

Out[10]: {1: 0.5,
          2: 0.6153846153846154,
          3: 0.5333333333333333,
          4: 0.47058823529411764,
          5: 0.47058823529411764,
          6: 0.34782608695652173,
          7: 0.7272727272727273,
          8: 0.4,
          9: 0.4}

In [11]:  1 centrality = nx.eigenvector_centrality(G)
          2 sorted((v, '{:0.2f}'.format(c)) for v, c in centrality.items())

Out[11]: [(1, '0.24'),
          (2, '0.45'),
          (3, '0.36'),
          (4, '0.32'),
          (5, '0.32'),
          (6, '0.08'),
          (7, '0.59'),
          (8, '0.16'),
          (9, '0.16')]
```

Рис. 5.11

вершина и ребро посещались ровно один раз, не больше и не меньше. В широком смысле имеются два различных способа перемещения по графу с целью поиска в нем данных. Если сначала учитывается ширина, то это называется *поиском в ширину* (breadth-first search, *BFS*), а если глубина, то *поиском в глубину* (depth-first search, *DFS*). Рассмотрим их по очереди.

BFS — поиск в ширину

BFS работает лучше всего, когда в `aGraph`, с которым мы имеем дело, поддерживается концепция слоев или уровней окрестностей. Например, когда отношения человека в LinkedIn отражены в виде графа, в нем будут присутствовать связи первого уровня, затем второго уровня и т. д, которые легко преобразуются в слои.

Алгоритм BFS начинает с корневой вершины и исследует вершины в окрестности. Затем он переходит на следующий уровень окрестности и повторяет процесс.

Давайте взглянем на алгоритм BFS поближе. Для этого сначала обратимся к следующему неориентированному графу (рис. 5.12).

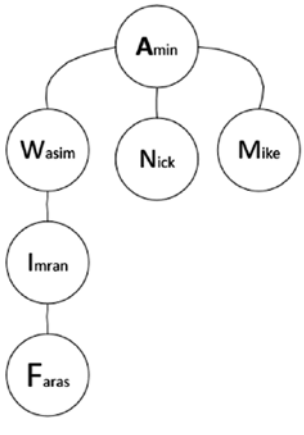


Рис. 5.12

Начнем с вычисления непосредственной окрестности каждой вершины и сохраним ее в списке, называемом *списком смежных вершин* (adjacency list). В Python для его хранения мы можем использовать структуру данных словаря:

```
graph={ 'Amin'   : {'Wasim', 'Nick', 'Mike'},
        'Wasim' : {'Imran', 'Amin'},
        'Imran' : {'Wasim','Faras'},
        'Faras' : {'Imran'},
        'Mike'  : {'Amin'},
        'Nick' :  {'Amin'}}
```

Подробно рассмотрим реализацию алгоритма BFS на Python.

Сначала мы объясним инициализацию, а затем основной цикл.

Инициализация

Мы будем использовать две структуры данных:

- `visited`. Содержит все вершины, которые были посещены. Изначально она будет пустой;
- `queue`. Здесь содержатся все вершины, которые мы хотим посетить на следующих итерациях.

Основной цикл

Реализуем основной цикл. Он будет выполняться до тех пор, пока в очереди не останется ни одного элемента. Для каждой вершины в очереди происходит следующее: если она уже была посещена, то посещается ее сосед.

Мы можем реализовать основной цикл в Python следующим образом:

1. Сначала извлечем первую вершину из очереди и выберем ее в качестве текущей вершины данной итерации.

   ```
   node = queue.pop(0)
   ```

2. Затем проверим, находится ли вершина в списке `visited`. Если это не так, мы добавляем ее в список посещенных вершин и используем `neighbours` для отображения непосредственно связанных с ней вершин.

   ```
   visited.append(node)
   neighbours = graph[node]
   ```

3. Теперь добавим соседние вершины в очередь.

   ```
   for neighbour in neighbours:
       queue.append(neighbour)
   ```

4. Как только основной цикл завершен, возвращается структура данных `visited`, которая содержит все пройденные вершины.
5. Полный код с инициализацией и основным циклом будет выглядеть следующим образом (рис. 5.13).

Давайте рассмотрим маршрут обхода исчерпывающего поиска для графа, который мы описали с помощью BFS. Схема обхода, при которой будут посещены все вершины, показана на рис. 5.14. Можно заметить, что во время выполнения она всегда работает с двумя структурами данных:

- `visited`. Содержит все вершины, которые были посещены;
- `queue`. Содержит вершины, которые еще предстоит посетить.

```
def bfs(graph, start):
    visited = []
    queue = [start]

    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.append(node)
            neighbours = graph[node]
            for neighbour in neighbours:
                queue.append(neighbour)
    return visited
```

Рис. 5.13

Вот как работает алгоритм:

1. Он начинает с первой вершины, *Amin*, которая является единственной вершиной на первом уровне.
2. Затем переходит на второй уровень и по очереди посещает все три вершины, *Wasim*, *Nick* и *Mike*.
3. После этого переходит на третий и четвертый уровни, которые содержат по одной вершине каждый, — *Imran* и *Faras*.

Как только все вершины были посещены, они добавляются в структуру `visited`, после чего итерации прекращаются (рис. 5.14).

Теперь попробуем с помощью BFS найти в этом графе конкретного человека. Давайте уточним данные для поиска и посмотрим на результаты (рис. 5.15).

Перейдем к алгоритму поиска в глубину.

DFS — поиск в глубину

DFS — это альтернатива BFS, используемая для поиска данных в графе. DFS отличается от BFS тем, что после запуска из корневой вершины алгоритм проходит как можно дальше по каждому из уникальных путей, перебирая их один за другим. Как только он успешно достигает конечной глубины каждого пути, он помечает флагом все вершины на этом пути как посещенные. После завершения пути алгоритм возвращается назад. Если он может найти еще один уникальный путь от корневого узла, процесс повторяется. Алгоритм продолжает двигаться по новым ветвям до тех пор, пока все ветви не будут посещены.

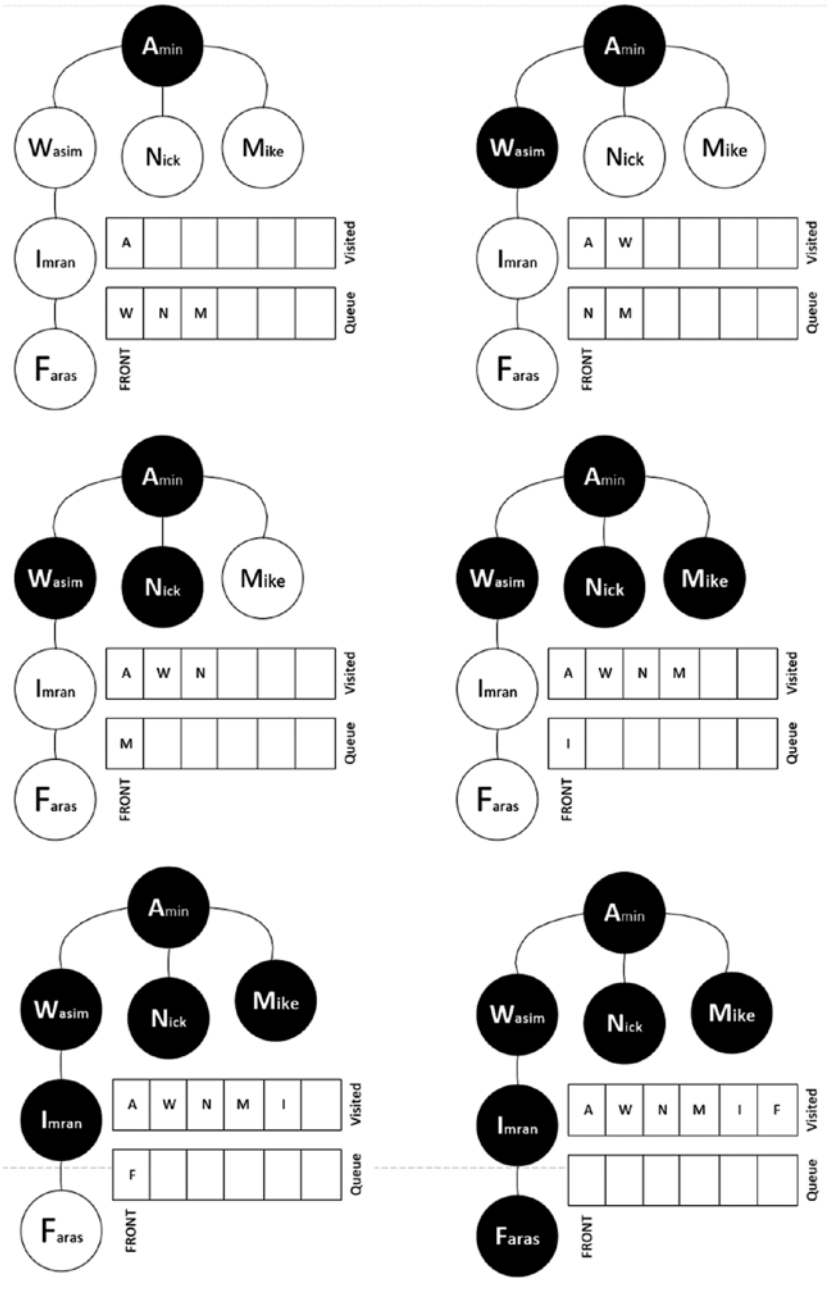

Amin
Wasim
Nick
Mike
Imran
Faras
A
W
N
M
I
F
Visited
Queue
FRONT

Рис. 5.14

```
In [97]: bfs(graph,'Amin')
Out[97]: ['Amin', 'Wasim', 'Nick', 'Mike', 'Imran', 'Faras']
```

Рис. 5.15

Обратите внимание, что граф может содержать цикл. Как уже упоминалось, во избежание зацикливания мы используем логический флаг для пометки обработанных вершин.

Для реализации DFS мы будем использовать структуру данных *стек* (см. главу 2). Как мы помним, стек основан на принципе *«последним пришел — первым ушел» (LIFO)*. Это противопоставляет его очереди, которая работает по принципу *«первым пришел — первым ушел» (FIFO)* и использовалась для BFS.

Код для DFS выглядит таким образом:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)
    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited
```

Вновь используем следующий код для тестирования функции `dfs`, определенной ранее:

```
graph={ 'Amin' : {'Wasim', 'Nick', 'Mike'},
         'Wasim' : {'Imran', 'Amin'},
         'Imran' : {'Wasim','Faras'},
         'Faras' : {'Imran'},
         'Mike'  :{'Amin'},
         'Nick'  :{'Amin'}}
```

Если мы запустим этот алгоритм, результат будет выглядеть так (рис. 5.16).

```
Out[94]: {'Amin', 'Faras', 'Imran', 'Mike', 'Nick', 'Wasim'}
```

Рис. 5.16

Рассмотрим полный маршрут обхода графа с использованием методологии DFS.

1. Итерация начинается с верхней вершины, *Amin*.
2. Затем алгоритм переходит на второй уровень, *Wasim*. Оттуда он движется к нижним уровням, пока не достигнет конца — вершин *Imran* и *Fares*.
3. После завершения первой полной ветви он возвращается назад, после чего переходит на второй уровень, чтобы перейти к вершинам *Nick* и *Mike*.

Схема обхода показана на рис. 5.17.

Обратите внимание, что DFS также можно использовать для работы с деревьями.

Далее мы разберем на примере, как изученные в этой главе концепции могут использоваться для решения реальной задачи.

ПРАКТИЧЕСКИЙ ПРИМЕР — ВЫЯВЛЕНИЕ МОШЕННИЧЕСТВА

Давайте посмотрим, как мы можем использовать *SNA* для выявления мошенничества. Поскольку люди являются социальными животными, считается, что на человеческое поведение влияют окружающие. Термин *гомофилия* был предложен для обозначения влияния на человека его личной сети социальных связей. Расширяет эту концепцию понятие *гомофильная сеть* — группа людей, которые, вероятно, связаны друг с другом благодаря какому-то общему фактору. Например, они могут иметь одинаковое происхождение или хобби, быть членами одной банды, учиться в одном университете, или их может объединять некоторая комбинация других факторов.

Пытаясь выявить мошенничество в гомофильной сети, мы можем основываться на отношениях между исследуемым лицом и другими людьми (вероятность их причастности к мошенничеству уже тщательно просчитана). Пометка человека как мошенника исходя из информации о его окружении иногда называется *виной по ассоциации*.

Чтобы понять данный процесс, сначала обратимся к простому случаю. Представим сеть из девяти человек: девять вершин и восемь ребер. Известно, что четверо членов сети — мошенники, то есть четыре вершины классифицируются как *мошеннические* (fraud, F). Остальные пятеро участников потенциально чисты перед законом — эти вершины классифицируются как *не мошеннические* (non-fraud, NF).

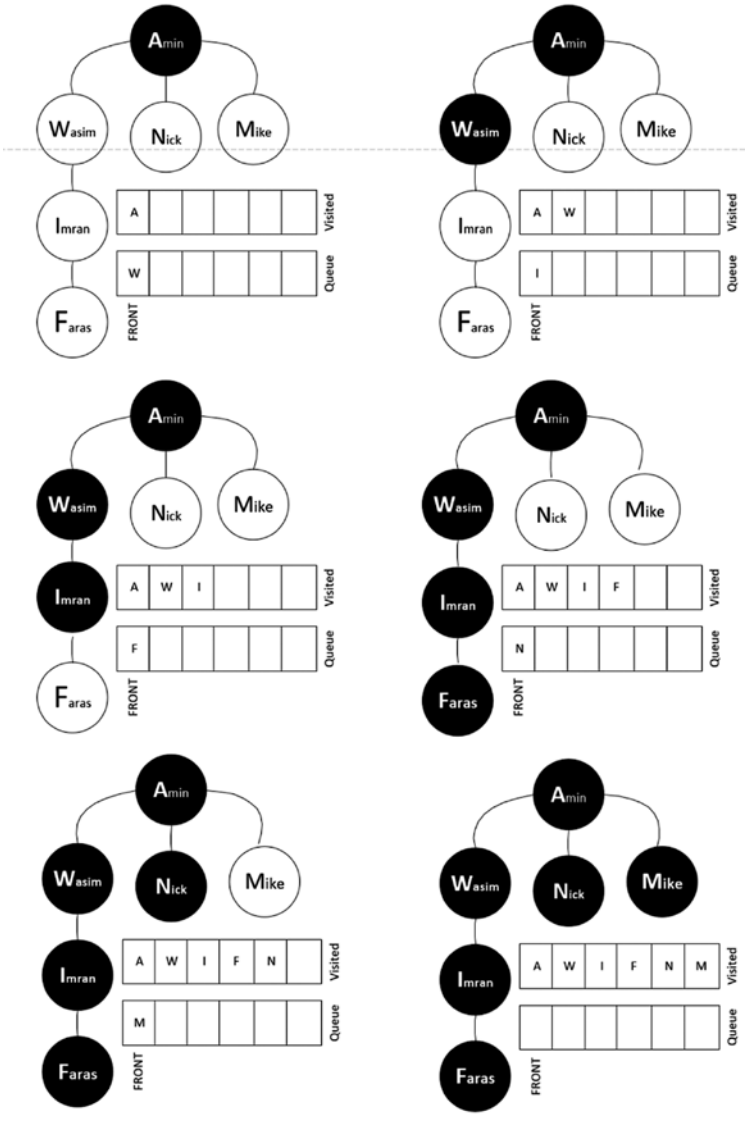


Рис. 5.17

Для создания такого графа напишем код, состоящий из следующих шагов:

1. Прежде всего импортируем нужные библиотеки:

```
import networkx as nx
import matplotlib.pyplot as plt
```

2. Определим структуры данных `vertices` и `edges`:

```
vertices = range(1,10)
edges= [(7,2), (2,3), (7,4), (4,5), (7,3), (7,5), (1,6),(1,7),(2,8),(2,9)]
```

3. Создадим экземпляр графа:

```
G = nx.Graph()
```

4. Нарисуем сам граф:

```
G.add_nodes_from(vertices)
G.add_edges_from(edges)
pos=nx.spring_layout(G)
```

5. Далее определим NF-вершины:

```
nx.draw_networkx_nodes( G,pos,
                        nodelist=[1,4,3,8,9],
                        with_labels=True,
                        node_color='g',
                        node_size=1300)
```

6. Теперь создадим вершины, которые, как нам заранее известно, причастны к мошенничеству:

```
nx.draw_networkx_nodes(G,pos,
                        nodelist=[2,5,6,7],
                        with_labels=True,
                        node_color='r',
                        _size=1300)
```

7. Создадим метки для вершин:

```
nx.draw_networkx_edges(G,pos,edges,width=3,alpha=0.5,edge_color='b'
) labels={} labels[1]=r'1 NF' labels[2]=r'2 F' labels[3]=r'3 NF'
labels[4]=r'4 NF' labels[5]=r'5 F' labels[6]=r'6 F' labels[7]=r'7
F' labels[8]=r'8 NF' labels[9]=r'9 NF'
nx.draw_networkx_labels(G,pos,labels,font_size=16)
```

Если запустить код выше, то мы увидим граф, подобный этому (рис. 5.18).

Обратите внимание, что мы уже провели тщательный анализ с целью классификации каждой вершины как *F* или *NF*. Предположим, что мы добавляем в сеть еще одну вершину с именем *q*, как показано на следующем рисунке. У нас нет никакой предварительной информации об этом человеке и о том, причастен ли он к мошенничеству. Мы хотим классифицировать его как *NF* или *F* на основе его связей с существующими членами социальной сети (рис. 5.19).

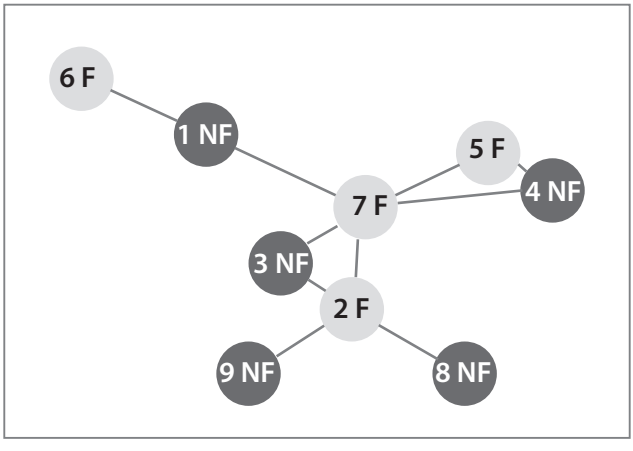


Рис. 5.18

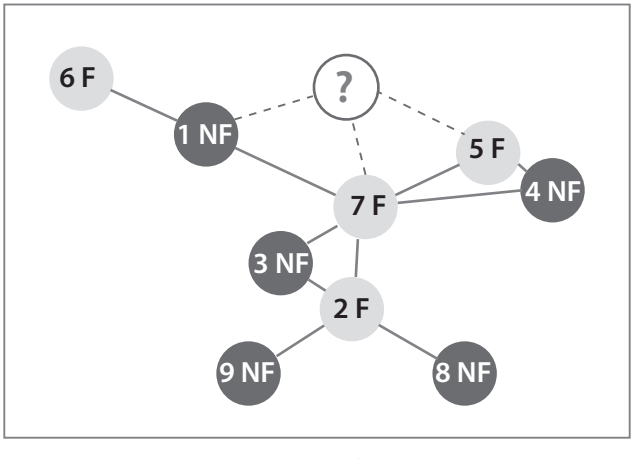


Рис. 5.19

Мы разработали два способа классифицировать нового участника *q* как *F* или *NF*:

- Простой метод, в котором не используются показатели центральности и дополнительная информация о типе мошенничества.
- Методология сторожевой башни — продвинутый метод, в котором используются показатели центральности существующих вершин, а также дополнительная информация о типе мошенничества.

Мы подробно обсудим каждый метод.

Простой анализ мошенничества

Простая техника анализа мошенничества основана на предположении, что на поведение человека влияют люди, с которыми он связан. Две вершины в сети с большей вероятностью будут вести себя одинаково, если они связаны друг с другом.

Основываясь на этом предположении, разработаем простую методику. Нам нужно определить вероятность того, что узел *a* является *F*. Представим эту вероятность как *P(F/q)* и вычислим ее следующим образом:

$$P(F \mid q) = \frac{1}{degree_q} \sum_{n_j \in Neighborhood_n \mid class(n_j)=F} w(n, n_j) DOS_{normalized_j}.$$

Применим это к предыдущему рисунку, где $Neighborhood_n$ представляет собой окрестность вершины *n*, а *w (n, nj)* — вес связи между *n* и *nj*.

Кроме того, $degree_q$ — это степень узла *q*. Далее вероятность рассчитывается следующим образом:

$$P(F \mid q) = \frac{1+1}{3} = \frac{2}{3} = 0.67.$$

Как мы видим, вероятность того, что этот человек замешан в мошенничестве, составляет 67 %. Нам нужно задать порог. Если пороговое значение составляет 30 %, то мы можем с уверенностью пометить данную вершину как *F*.

Обратите внимание, что этот процесс необходимо повторять для каждой новой вершины в сети.

Теперь рассмотрим продвинутый способ анализа мошенничества.

Анализ мошенничества методом сторожевой башни

Простой метод анализа мошенничества, представленный ранее, имеет два ограничения:

- Он не оценивает важности каждой вершины в социальной сети. Отношения с человеком, находящимся в центре сети, означают более высокую степень вовлеченности в мошенничество, нежели отношения с человеком на периферии сети.

- Когда мы помечаем кого-то как вовлеченного в известный случай мошенничества в существующей сети, мы не учитываем тяжесть преступления.

Методология анализа мошенничества по принципу сторожевой башни устраняет эти два ограничения. Прежде всего рассмотрим две концепции.

Оценка негативного воздействия

Если человек причастен к мошенничеству, считается, что с этим человеком связано *негативное воздействие* (negative outcome). Не каждое негативное воздействие одинаково серьезно или опасно. Человек, который выдает себя за другого, оказывает более серьезное негативное воздействие по сравнению с тем, кто просто пытается использовать просроченную подарочную карту на 20 долларов.

Мы будем оценивать негативное воздействие баллами от 1 до 10 следующим образом (табл. 5.1).

Таблица 5.1

Негативное воздействие	**Оценка негативного воздействия**
Выдача себя за другого	10
Причастность к краже кредитных карт	8
Отправка поддельного чека	7
Судимость	6
Отсутствие судимости	0

Наша оценка основывается на ретроспективном анализе случаев мошенничества и их последствий.

Степень подозрения (DOS)

Степень подозрения (degree of suspicion, DOS) количественно определяет вероятность причастности человека к мошенничеству. Нулевое значение DOS означает низкую вероятность совершения преступления человеком, а значение DOS, равное 9, — высокую вероятность.

Анализ исторических данных показывает, что профессиональные мошенники занимают важное положение в собственных социальных сетях. Чтобы учесть этот факт, мы вычислим все четыре показателя центральности для каждой вер-

шины в нашей сети. Затем мы возьмем среднее значение этих вершин. Это и будет показывать важность конкретного человека в сети.

Если человек, связанный с вершиной, замешан в мошенничестве, мы проиллюстрируем негативное воздействие, оценив его с помощью заранее определенных значений из таблицы выше (табл. 5.1). Это делается для того, чтобы тяжесть преступления отражалась в значении каждого отдельного DOS.

Наконец, чтобы получить значение DOS, перемножим среднее значение показателей центральности и оценку негативного воздействия. Нормализуем DOS, разделив его на максимальное значение DOS в сети.

Давайте рассчитаем DOS для каждой из девяти вершин указанной сети (табл. 5.2).

Таблица 5.2

	Вершина 1	**Вершина 2**	**Вершина 3**	**Вершина 4**	**Вершина 5**	**Вершина 6**	**Вершина 7**	**Вершина 8**	**Вершина 9**
Степень центральности	0.25	0.5	0.25	0.25	0.25	0.13	0.63	0.13	0.13
Посредничество	0.25	0.47	0	0	0	0	0.71	0	0
Близость	0.5	0.61	0.53	0.47	0.47	0.34	0.72	0.4	0.4
Влиятельность	0.24	0.45	0.36	0.32	0.32	0.08	0.59	0.16	0.16
Среднее значение центральности	0.31	0.51	0.29	0.26	0.26	0.14	0.66	0.17	0.17
Оценка негативного воздействия	0	6	0	0	7	8	10	0	0
DOS	0	3	0	0	1.82	1.1	6.625	0	0
Нормализованный DOS	0	0.47	0	0	0.27	0.17	1	0	0

Каждая из вершин и ее нормализованный DOS показаны на рис. 5.20.

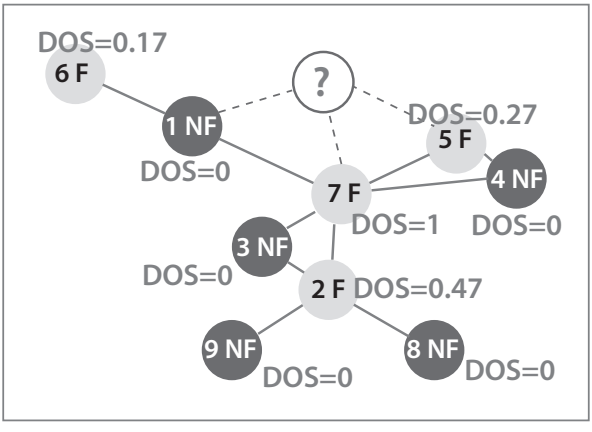


Рис. 5.20

Для расчета DOS новой добавленной вершины будем использовать следующую формулу:

$$DOS_k = \frac{1}{degree_k} \sum_{n_j \in Neighborhood_n} w(n, n_j) DOS_{normalized_j}.$$

Используя соответствующие значения, рассчитаем DOS:

$$DOS_k = \frac{(0+1+0.27)}{3} = 0.42.$$

Полученное значение указывает на риск мошенничества, связанный с новой вершиной, добавленной в систему. Это означает, что по шкале от 0 до 1 данный человек имеет значение DOS 0.42. Мы можем создать несколько подборок DOS, как показано в табл. 5.3.

Таблица 5.3

Значение DOS	**Классификация риска**
DOS = 0	Никакого риска
0 < DOS ≤ 0.10	Низкий риск
0.10 < DOS ≤ 0.3	Умеренный риск
DOS > 0.3	Высокий риск

Основываясь на этих критериях, можно сделать вывод, что новый человек относится к группе высокого риска и должен быть помечен.

Обычно показатели времени не учитываются при проведении такого анализа. Но сегодня существуют продвинутые методы, которые рассматривают рост графа с течением времени. Это позволяет исследователям взглянуть на взаимосвязь между вершинами по мере развития сети. Анализ временных рядов на графах во много раз усложняет задачу. Однако с его помощью можно получить дополнительные данные для доказательства мошенничества, которые в противном случае было бы невозможно обнаружить.

РЕЗЮМЕ

В этой главе мы узнали об алгоритмах, основанных на графах. Теперь мы сможем использовать различные методы представления графов и графовые алгоритмы поиска и обработки данных. Мы научились вычислять кратчайшее расстояние между двумя вершинами и строить окрестности в пространстве задачи. Эти знания помогают применить теорию графов для решения таких задач, как выявление мошенничества.

В следующей главе мы сосредоточимся на алгоритмах машинного обучения без учителя. Многие рассмотренные в этой главе практические методы дополняют алгоритмы обучения без учителя, с которыми мы познакомимся далее. Поиск доказательств мошенничества в наборе данных является примером такого метода.

Часть II

АЛГОРИТМЫ МАШИННОГО ОБУЧЕНИЯ

В этой части мы узнаем о различных видах алгоритмов машинного обучения: с учителем и без учителя. Кроме того, познакомимся с алгоритмами обработки естественного языка и с механизмами рекомендаций. В данную часть включены следующие главы:

- Глава 6. Алгоритмы машинного обучения без учителя.
- Глава 7. Традиционные алгоритмы обучения с учителем.
- Глава 8. Алгоритмы нейронных сетей.
- Глава 9. Алгоритмы обработки естественного языка.
- Глава 10. Рекомендательные системы.

6

Алгоритмы машинного обучения без учителя

Эта глава посвящена алгоритмам машинного обучения без учителя. Мы начнем с введения в методы обучения без учителя, затем узнаем о двух алгоритмах кластеризации: методе *k*-средних и алгоритмах иерархической кластеризации. Далее мы изучим алгоритм уменьшения размерности; он эффективен при наличии большого количества входных переменных. Мы узнаем, как обучение без учителя используется для обнаружения аномалий. Наконец, рассмотрим один из самых мощных методов обучения без учителя — поиск ассоциативных правил. Шаблоны, обнаруженные в результате такого поиска, отражают интересные взаимосвязи между различными элементами данных в транзакциях. Это помогает в принятии решений на основе данных.

К концу этой главы вы будете знать основные алгоритмы и методологии обучения без учителя и сможете использовать их для решения ряда реальных задач.

В главе представлены следующие темы:

- Обучение без учителя (unsupervised learning).
- Алгоритмы кластеризации (clustering algorithms).
- Снижение размерности (dimensionality reduction).
- Поиск ассоциативных правил (association rules mining).

- Алгоритмы обнаружения выбросов, или аномалий (anomaly detection algorithms).

ОБУЧЕНИЕ БЕЗ УЧИТЕЛЯ

Обучение без учителя (unsupervised learning) — это процесс придания структуры неструктурированным данным путем обнаружения и использования присущих им общих признаков. Если набор данных не создан каким-либо случайным процессом, то в данных в многомерном пространстве задачи существуют некоторые шаблоны. Алгоритмы обучения без учителя обнаруживают их и используют для структурирования данных. Эта концепция показана на рис. 6.1.

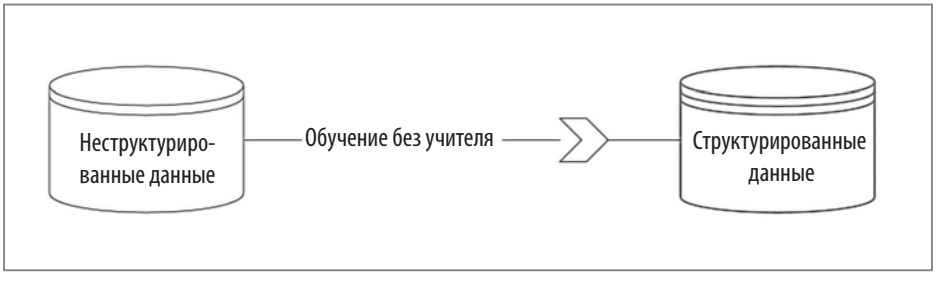


Рис. 6.1

Обучение без учителя структурирует данные, обнаруживая новые признаки в существующих шаблонах.

Обучение без учителя в жизненном цикле майнинга данных

Чтобы понять роль обучения без учителя, в первую очередь необходимо взглянуть на жизненный цикл *майнинга данных* (data mining) в целом. Существуют различные методологии, разделяющие его на независимые этапы, называемые *фазами*. В настоящее время есть два популярных способа построения жизненного цикла майнинга данных:

- *CRISP-DM (Cross-Industry Standard Process for Data Mining)*;
- *SEMMA (Sample, Explore, Modify, Model, Access)*.

CRISP-DM был разработан консорциумом датамайнеров, представляющих такие компании, как Chrysler и SPSS (Statistical Package for Social Science). SEMMA был предложен компанией SAS (Statistical Analysis System). В книге мы рассмотрим CRISP-DM и попытаемся понять место обучения без учителя в жизненном цикле майнинга данных. Отметим, что SEMMA состоит из схожих этапов.

Жизненный цикл CRISP-DM состоит из шести фаз, которые показаны на рис. 6.2.

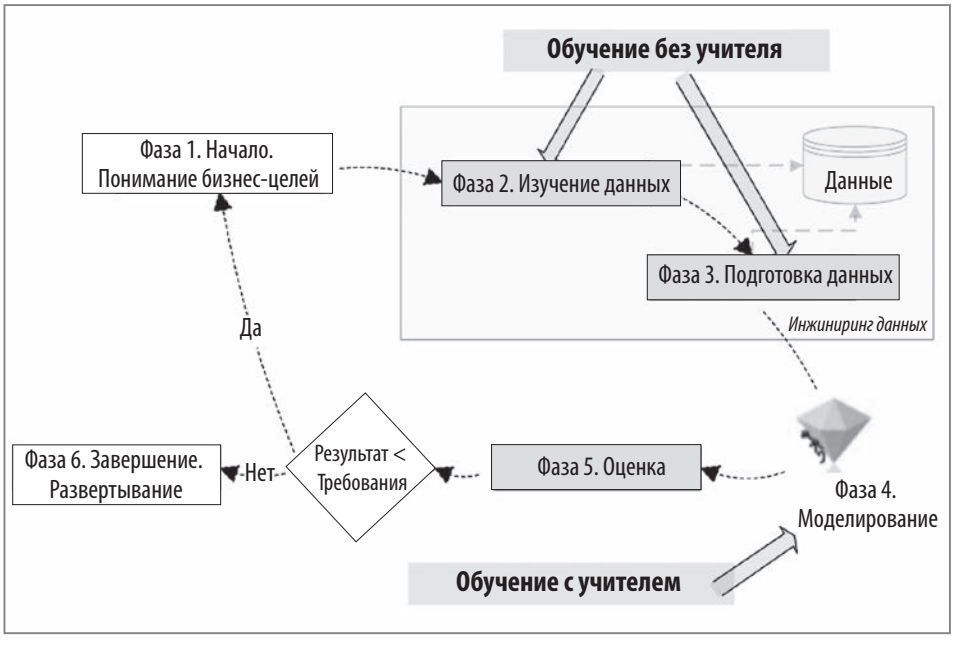


Рис. 6.2

Давайте по очереди разберем каждую фазу.

- *Фаза 1. Понимание бизнес-целей.* На данном этапе происходит понимание задачи с позиции бизнеса и определяются ее требования. Оценка масштаба задачи и правильное ее описание в терминах *машинного обучения (МО)* — важная часть этапа. Например, для задачи бинарной классификации иногда полезно сформулировать требования в терминах гипотезы, которая может быть доказана или отклонена. Эта фаза также посвящена документированию ожиданий от модели МО (она будет обучена на этапе моделирования). Так, для задачи классификации нам необходимо задать минимально приемлемую точность модели, которая может быть развернута в производственной среде.

Важно отметить, что первый этап CRISP-DM связан с пониманием с позиции бизнеса. Здесь мы фокусируемся на том, что нужно сделать, а не на том, как именно это будет сделано.

- *Фаза 2. Изучение данных.* На этом этапе мы исследуем данные и ищем подходящие для решения задачи датасеты (наборы данных). Необходимо оценить качество данных и их структуру, а также обнаружить полезные закономерности. Затем мы должны выявить подходящий признак, который можно использовать в качестве метки (или целевой переменной) в соответствии с требованиями фазы 1. Алгоритмы обучения без учителя способны сыграть ключевую роль в достижении следующих целей:
  - обнаружить закономерности в датасете;
  - изучить структуру датасета, проанализировав обнаруженные закономерности;
  - определить или вывести целевую переменную.
- *Фаза 3. Подготовка данных.* Речь идет о подготовке данных для модели МО, которую мы будем обучать. Размеченные данные разделяются на две неравные части. Большая часть называется *обучающими данными* (training data) и используется для дальнейшего обучения модели. Меньшая часть называется *проверочными данными* (testing data) и применяется для оценки модели (фаза 5). В качестве инструмента подготовки данных используются алгоритмы МО без учителя. С их помощью мы преобразовываем неструктурированные данные в структурированные и создаем дополнительные параметры, которые могут быть полезны при обучении модели.
- *Фаза 4. Моделирование.* На данном этапе мы используем обучение с учителем, чтобы сформулировать обнаруженные закономерности. Необходимо произвести подготовку данных в соответствии с требованиями выбранного алгоритма обучения с учителем и определить конкретный признак, который будет использоваться в качестве метки. На этапе подготовки мы разделили данные на проверочные и обучающие. Теперь нужно составить математические формулировки для описания взаимосвязей в интересующих нас закономерностях. Это делается путем обучения модели с помощью набора обучающих данных, созданного на этапе 3. Полученное математическое описание зависит от выбора алгоритма.
- *Фаза 5. Оценка.* Этот этап посвящен тестированию недавно обученной модели с использованием проверочных данных. Если результат не соответствует требованиям, установленным на этапе понимания бизнес-целей, мы повторяем цикл, начиная с фазы 1. Это проиллюстрировано на рис. 6.2.

- *Фаза 6. Внедрение*. Если результат совпадает с ожиданиями или превосходит их, то обученная модель развертывается в рабочей среде и начинает генерировать решение задачи, сформулированной на фазе 1.

Вторая и третья фазы жизненного цикла CRISP-DM связаны с пониманием данных и их подготовкой для обучения модели, что включает в себя обработку данных. На этом этапе некоторые организации привлекают к работе дата-инженеров.

Очевидно, что процесс поиска решения задачи целиком зависит от данных. Сочетание машинного обучения с учителем и без учителя используется для выработки приемлемого решения. В этой главе основное внимание уделяется роли обучения без учителя.

Инжиниринг данных включает в себя фазы 2 и 3 и является наиболее трудоемкой частью машинного обучения. Две эти фазы способны занять до 70 % времени и ресурсов типового проекта МО. Важную роль в инжиниринге данных играют алгоритмы обучения без учителя.

В следующих разделах содержится более подробная информация об алгоритмах без учителя.

Современные тенденции исследований в области обучения без учителя

В течение многих лет исследования алгоритмов МО были в большей степени сосредоточены на методах обучения с учителем. Поскольку эти методы применяются непосредственно для формирования логических выводов, их преимущества с точки зрения времени, затрат и точности относительно легко поддаются измерению. Эффективность алгоритмов МО без учителя была признана не так давно. Поскольку обучение без учителя не управляется человеком, оно в меньшей степени зависит от предположений и потенциально может обеспечить сходимость решения по всем параметрам. Хотя масштаб и требования к обработке таких алгоритмов сложнее контролировать, они обладают большим потенциалом для выявления скрытых закономерностей. Специалисты работают над объединением методов МО с учителем и без учителя, чтобы получить новые мощные алгоритмы.

Практические примеры

В настоящее время обучение без учителя используется для лучшего понимания данных и придания им большей структурированности — например, при сегментации рынка, выявлении мошенничества или анализе рыночной корзины (что обсуждается далее в этой главе). Давайте рассмотрим два примера.

Распознавание речи

Обучение без учителя применяется для определения отдельных голосов на записи. Алгоритм использует тот факт, что речь каждого человека имеет свои особенности. Он создает потенциально различимые аудиопаттерны, которые применяются для распознавания речи. Например, Google использует эту технику на своих устройствах Google Home, чтобы научить их различать голоса разных людей. Пройдя обучение, приложение Google Home способно персонализировать ответ для каждого пользователя.

Предположим, что у нас есть записанный разговор трех человек, общающихся друг с другом в течение получаса. Используя алгоритмы обучения без учителя, мы можем идентифицировать в этом наборе данных голоса разных людей. Благодаря обучению без учителя мы структурируем заданный набор неструктурированных данных. Создание структуры привносит дополнительные параметры, полезные для извлечения информации и подготовки данных для выбранного нами алгоритма МО. На следующей диаграмме показано, как обучение без учителя используется для распознавания голоса (рис. 6.3).

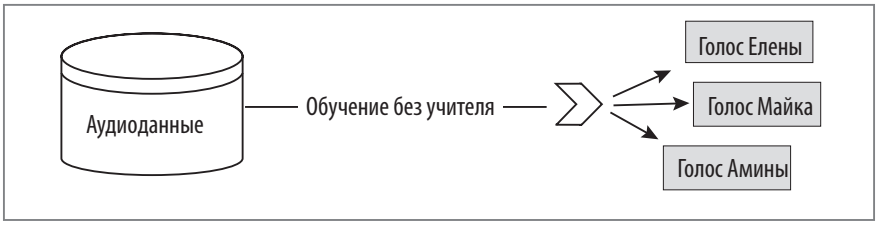


Рис. 6.3

Обратите внимание, что в этом случае обучение без учителя предполагает, что мы добавляем новый признак с тремя различными уровнями.

Классификация документов

Алгоритмы обучения без учителя также применяются в сфере хранения неструктурированных текстовых данных. Например, если у нас есть набор

PDF-документов, то обучение без учителя можно применить следующим образом:

- выделить различные темы в наборе данных;
- связать каждый PDF-документ с определенной темой.

Такое использование обучения без учителя показано на рис. 6.4. Это еще один пример структурирования изначально неструктурированных данных.

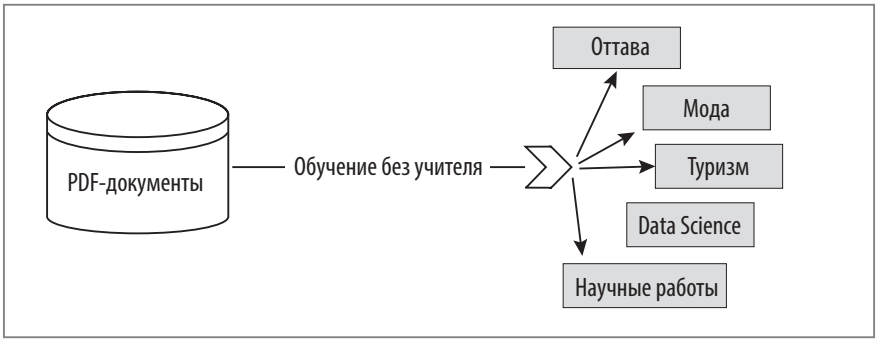


Рис. 6.4

Обратите внимание, что в этом случае обучение без учителя предполагает, что мы добавляем новый признак с пятью различными уровнями.

АЛГОРИТМЫ КЛАСТЕРИЗАЦИИ

Рассмотрим один из самых простых и мощных методов обучения без учителя. Он заключается в объединении схожих шаблонов в группы с помощью алгоритмов кластеризации. Этот метод используется для изучения конкретного аспекта данных, связанного с нашей задачей. Алгоритмы кластеризации формируют естественные группы элементов данных. Это происходит независимо от каких-либо целей или предположений, поэтому данный процесс можно классифицировать как метод обучения без учителя.

С помощью кластерных алгоритмов элементы данных в пространстве задачи распределяются по группам на основе их сходства между собой. Выбор наилучшего способа кластеризации зависит от характера задачи. Рассмотрим методы, которые можно использовать для вычисления сходства между различными элементами данных.

Количественная оценка сходства

Надежность группы, созданной с помощью алгоритмов кластеризации, основана на предположении, что мы можем точно оценить сходство (близость) между различными точками данных в пространстве задачи. Для этого мы используем различные *меры расстояния* (distance measures). Ниже приведены три наиболее популярные меры расстояния, используемые для количественной оценки сходства:

- *Евклидово расстояние* (Euclidean distance).
- *Манхэттенское расстояние* (Manhattan distance), или *расстояние городских кварталов*.
- *Косинусное расстояние* (Cosine distance).

Давайте рассмотрим эти меры расстояния более подробно.

Евклидово расстояние

Расстояние между точками помогает определить меру сходства между двумя точками данных и широко используется в таких методах машинного обучения без учителя, как кластеризация. *Евклидово расстояние* (Euclidean distance) — наиболее распространенная и понятная мера расстояния. Это кратчайшее расстояние между двумя точками данных в многомерном пространстве. Рассмотрим две точки — $A(1,1)$ и $B(4,4)$ — в двумерном пространстве, как это показано на рис. 6.5.

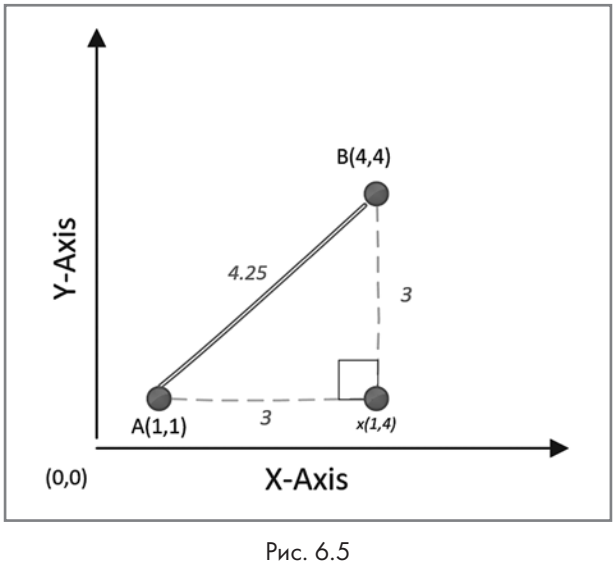


Рис. 6.5

Чтобы рассчитать расстояние между A и B, то есть $d(A, B)$, мы можем использовать следующую формулу Пифагора:

$$d(A,\ B) = \sqrt{(a_2 - b_2)^2 + (a_1 - b_1)^2} = \sqrt{(4-1)^2 + (4-1)^2} = \sqrt{9+9} = 4.25.$$

Обратите внимание, что данный расчет предназначен для двумерного пространства задач. Для n-мерного пространства задач мы можем вычислить расстояние между двумя точками A и B следующим образом:

$$d(A,\ B) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}.$$

Манхэттенское расстояние

Часто измерение кратчайшей дистанции между двумя точками с помощью евклидова расстояния не отражает сходства (или близости) между двумя точками. Например, если это две точки на карте, то фактическое расстояние от точки A до точки B при использовании наземного транспорта (автомобиль или такси) будет больше, чем евклидово расстояние. Для подобных ситуаций мы используем *манхэттенское расстояние* (Manhattan distance), также называемое *расстоянием городских кварталов*. Оно обозначает самый длинный маршрут между двумя точками. Эта мера расстояния лучше отражает близость двух элементов в качестве точек отправления и назначения, до которых можно добраться в оживленном городе. Сравнение показателей манхэттенского и евклидова расстояний показано на рис. 6.6.

Обратите внимание, что манхэттенское расстояние всегда будет равно или больше соответствующего евклидова расстояния.

Косинусное расстояние

Рассмотренные выше меры расстояния не предназначены для работы с многомерным пространством. В многомерном пространстве задачи более точно отражает близость между двумя точками данных *косинусное расстояние* (Cosine distance). Чтобы рассчитать косинусное расстояние, нужно измерить угол, образованный двумя точками, соединенными с исходной точкой. Если точки данных расположены близко, то угол будет острым, независимо от их координат. Если же они находятся далеко, то угол будет тупым (рис. 6.7).

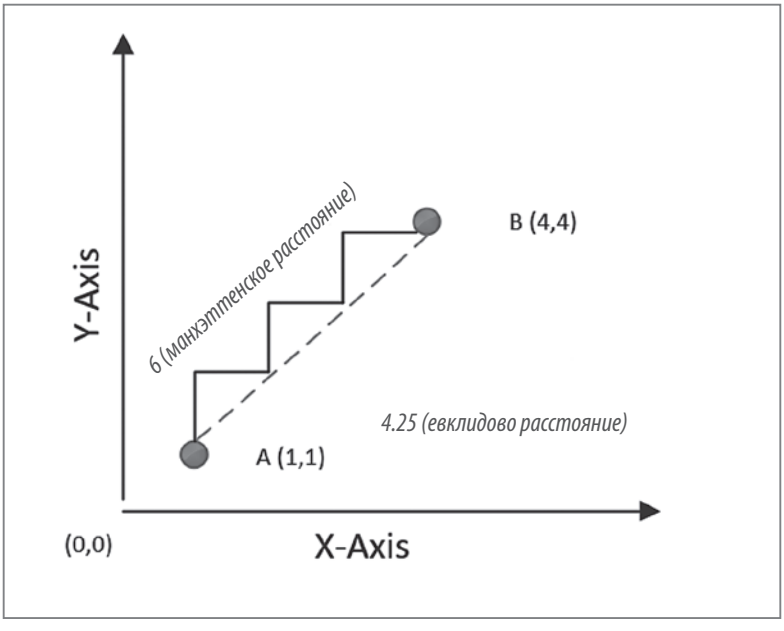

Y-Axis
B (4,4)
6 (манхэттенское расстояние)
4.25 (евклидово расстояние)
A (1,1)
(0,0)
X-Axis

Рис. 6.6

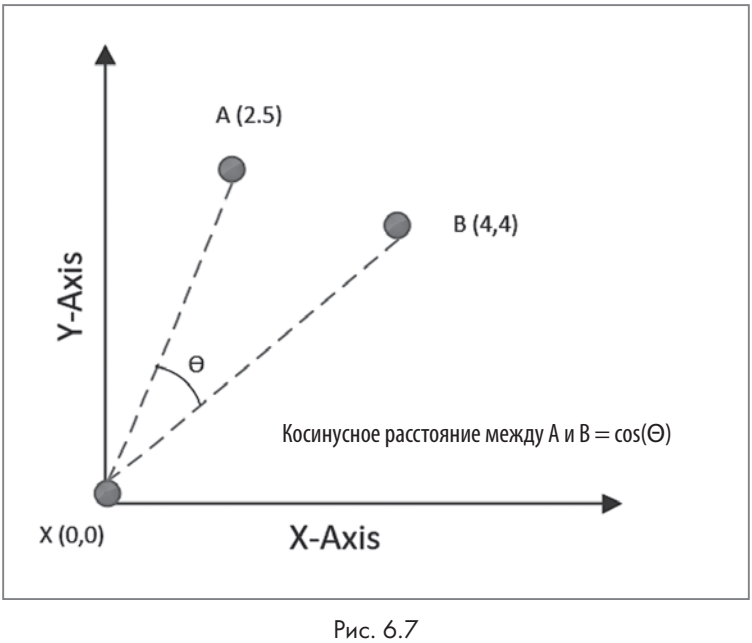

A (2.5)
B (4,4)
Y-Axis
Θ
Косинусное расстояние между A и B = cos(Θ)
X (0,0)
X-Axis

Рис. 6.7

Текстовые данные, по сути, можно считать пространством с высокой размерностью. Поскольку мера косинусного расстояния очень хорошо работает с *n*-мерными пространствами, это удачный выбор в случае с текстовыми данными.

Обратите внимание, что на предыдущем рисунке косинус угла между $A(2,5)$ и $B(4,4)$ является косинусным расстоянием. Исходной точкой служит начало координат, то есть $X(0,0)$. Но на самом деле любая точка в пространстве задач может быть исходной; это не обязательно должно быть начало координат.

Алгоритм кластеризации методом *k*-средних

Название алгоритма кластеризации *метод k-средних* (*k*-means) происходит от принципа его работы: он создает несколько *кластеров, k*, определяя их центры, чтобы вычислить близость между точками данных. В данном алгоритме применяется относительно простой (но тем не менее популярный из-за своей масштабируемости и скорости) подход к кластеризации. Метод *k*-средних использует итеративную логику. Центр кластера перемещается до тех пор, пока он не окажется в оптимальной точке.

Важно отметить, что в алгоритмах методом *k*-средних отсутствует одна из основных функций, необходимых для кластеризации. Алгоритм *k*-средних не может определить наиболее подходящее количество кластеров. Оптимальное количество кластеров, *k*, зависит от количества естественных групп в наборе данных. Эта особенность упрощает алгоритм и максимально повышает его производительность. Он не содержит ничего лишнего и эффективно справляется с большими наборами данных. Предполагается, что для вычисления *k* будет использоваться какой-то внешний механизм, выбор которого зависит от характера задачи. Иногда *k* напрямую определяется контекстом задачи кластеризации. Например, нам нужно разделить класс студентов на два кластера. Один кластер состоит из студентов, обладающих навыками в области анализа данных, а другой — навыками программирования. Таким образом, *k* будет равно двум. Для ряда других задач значение *k* может оказаться неочевидным. В таких случаях для оценки наиболее подходящего количества кластеров необходимо использовать итеративную процедуру проб и ошибок или алгоритм, основанный на эвристике.

Логика кластеризации методом *k*-средних

В этом разделе описывается логика кластеризации методом *k*-средних. Подробно рассмотрим ход алгоритма.

Инициализация

Чтобы определить сходство (близость) точек данных и сгруппировать их, алгоритм *k*-средних использует меру расстояния; ее необходимо выбрать перед началом работы. По умолчанию будет использоваться евклидово расстояние. Кроме того, если в наборе данных есть *выбросы* (outliers), то необходимо выработать критерии, согласно которым происходит их удаление.

Этапы алгоритма *k*-средних

Шаги, составляющие алгоритм кластеризации *k*-средних, представлены в табл. 6.1.

Таблица 6.1

Шаг 1	Определяем количество кластеров, *k*
Шаг 2	Среди точек данных случайным образом выбираем *k* точек в качестве центров кластеров
Шаг 3	На основе выбранной меры расстояния итеративно вычисляем расстояние от каждой точки в пространстве задачи до каждого из *k* центров кластеров. В зависимости от объема данных это может быть довольно затратный шаг: например, если в кластере 10 000 точек и $k = 3$, это означает, что необходимо рассчитать 30 000 расстояний
Шаг 4	Привязываем каждую точку данных к ближайшему центру кластера
Шаг 5	Теперь каждый элемент данных в пространстве задачи привязан к центру кластера. Поскольку центры были выбраны в случайном порядке, нужно удостовериться, что они являются оптимальными. Пересчитываем центры кластеров, вычисляя средние значения точек данных в каждом кластере. Этот шаг объясняет, почему алгоритм называется методом *k*-средних
Шаг 6	Если на шаге 5 центры кластеров сместились, то нужно пересчитать привязку каждой точки данных к кластеру. Для этого повторяем шаг 3. Если центры кластеров не сместились или если предустановленное условие остановки (например, максимальное количество итераций) выполнено, то процесс завершается

На рис. 6.8 показан результат выполнения алгоритма *k*-средних в двумерном пространстве задачи.

Обратите внимание, что два результирующих кластера, созданных после выполнения *k*-средних, в данном случае хорошо различимы.

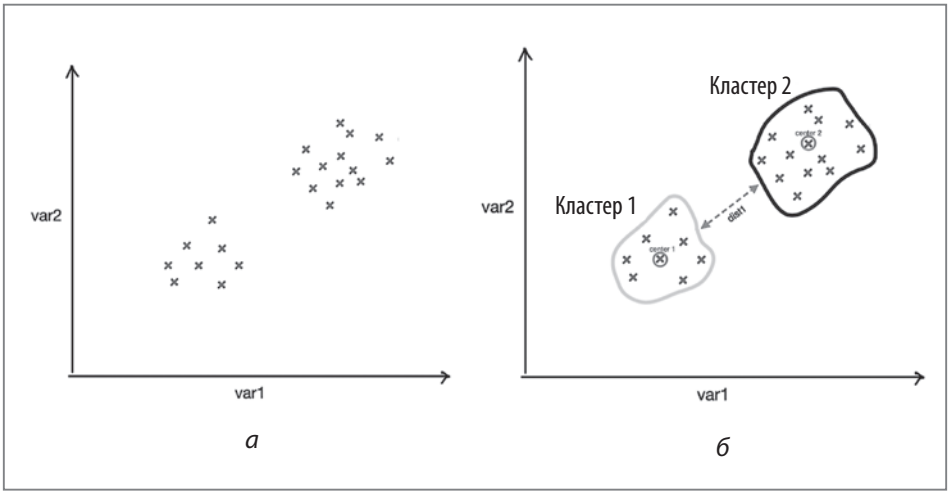


Рис. 6.8. *а* — точки данных до кластеризации; *б* — результирующие кластеры после выполнения алгоритма кластеризации методом *k*-средних

Условие остановки

Для алгоритма *k*-средних условием остановки по умолчанию является то, что на шаге 5 больше не происходит смещения центров кластеров. Но, как и многим другим алгоритмам, алгоритму *k*-средних может потребоваться много времени для достижения сходимости, особенно при обработке больших наборов данных в многомерном пространстве задачи. Вместо того чтобы ждать, пока алгоритм сойдется, мы можем самостоятельно задать условие остановки следующими способами.

- Указав максимальное время выполнения:
  - *Условие остановки*: $t > t_{max}$, где t — текущее время выполнения, а t_{max} — максимальное время выполнения, которое мы установили для алгоритма.
- Указав максимальное количество итераций:
  - *Условие остановки*: $m > m_{max}$, где m — текущая итерация, а m_{max} — максимальное количество итераций, которое мы установили для алгоритма.

Кодирование алгоритма *k*-средних

Перейдем к реализации алгоритма *k*-средних на Python.

1. Прежде всего надо импортировать необходимые библиотеки Python. Для кластеризации методом *k*-средних импортируем библиотеку `sklearn`:

```
from sklearn import cluster
import pandas as pd
import numpy as np
```

2. Чтобы применить алгоритм k-средних, создадим 20 точек данных в двумерном пространстве задачи, которые и будем использовать для кластеризации:

```
dataset = pd.DataFrame({
    'x': [11, 21, 28, 17, 29, 33, 24, 45, 45, 52, 51, 52, 55, 53,
55, 61, 62, 70, 72, 10],
    'y': [39, 36, 30, 52, 53, 46, 55, 59, 63, 70, 66, 63, 58, 23,
14, 8, 18, 7, 24, 10]
})
```

3. Сформируем два кластера ($k = 2$), а затем создадим кластер, вызвав функции `fit`:

```
myKmeans = cluster.KMeans(n_clusters=2)
myKmeans.fit(dataset)
```

4. Введем переменную с именем `centroid`, которая представляет собой массив, содержащий местоположение центра сформированных кластеров. В нашем случае, при $k = 2$, массив будет иметь размер 2. Затем создадим переменную с именем `label`, которая обозначает привязку каждой точки данных к одному из двух кластеров. Поскольку имеется 20 точек данных, то и массив будет иметь размер 20:

```
centroids = myKmeans.cluster_centers_
labels = myKmeans.labels_
```

5. Теперь выведем эти два массива, `centroids` и `labels` (рис. 6.9).

```
In [3]: print(labels)
        [1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1]

In [6]: print(centroids)
        [[57.09090909 15.09090909]
         [16.77777778 48.88888889]]
```

Рис. 6.9

Обратите внимание, что первый массив показывает соответствие кластера каждой точке данных, а второй представляет собой два центра кластера.

6. Визуализируем кластеры с помощью `matplotlib` (рис. 6.10).

```
In [16]: import matplotlib.pyplot as plt
         plt.scatter(dataset['x'],dataset['y'], s=10)
         plt.scatter(centers[0],centers[1],s=100)
         plt.show()
```

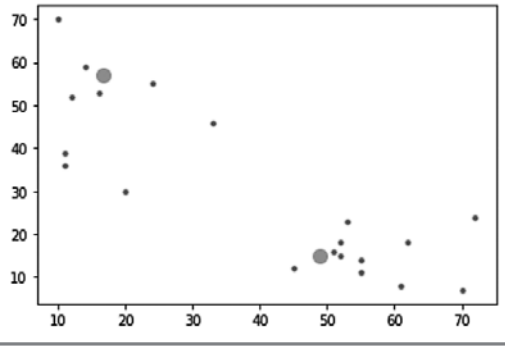


Рис. 6.10

Обратите внимание, что большие точки на графике являются центроидами, которые определил алгоритм *k*-средних.

Ограничения кластеризации методом *k*-средних

Алгоритм *k*-средних разработан как простой и быстрый алгоритм. Но из-за преднамеренной простоты в архитектуре он имеет ряд ограничений:

- Самое большое ограничение: количество кластеров должно быть заранее определено.
- Первоначальное назначение центров кластеров является случайным. Таким образом, при каждом запуске алгоритм может давать несколько разные кластеры.
- Каждая точка данных привязывается только к одному кластеру.
- Кластеризация методом *k*-средних чувствительна к выбросам.

Иерархическая кластеризация

Кластеризация методом *k*-средних реализует подход «сверху вниз», так как мы начинаем алгоритм с наиболее важных точек данных, которые служат центрами кластеров. Существует альтернативный подход к кластеризации, при котором вместо того, чтобы начинать сверху, мы запускаем алгоритм снизу. «Внизу» в этом контексте находится каждая отдельная точка данных в пространстве задачи.

Метод заключается в том, чтобы группировать похожие точки данных вместе по мере продвижения к центрам кластера. Подход «снизу вверх» используется алгоритмами иерархической кластеризации. Давайте рассмотрим его подробнее.

Этапы иерархической кластеризации

Иерархическая кластеризация состоит из следующих шагов:

1. Создаем отдельный кластер для каждой точки данных. Если наше пространство задачи состоит из 100 точек данных, то алгоритм начнет со 100 кластеров.
2. Группируем только ближайшие друг к другу точки.
3. Проверяем условие остановки; если оно еще не выполнено, то повторяем шаг 2.

Полученная кластерная структура называется *дендрограммой*.

В дендрограмме высота вертикальных линий показывает, насколько близко расположены элементы (рис. 6.11).

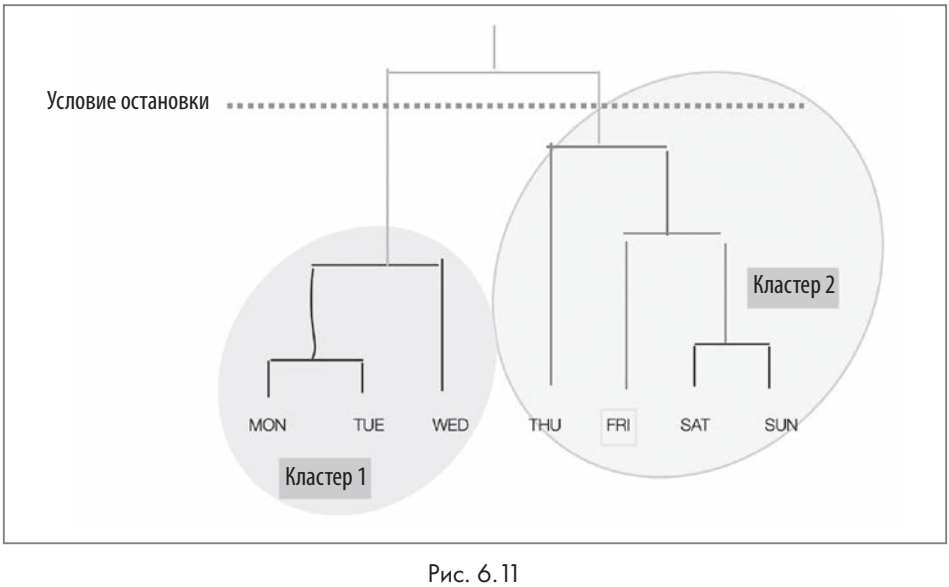


Рис. 6.11

Обратите внимание, что условие остановки показано пунктирной линией на рисунке выше.

Кодирование алгоритма иерархической кластеризации

Теперь давайте реализуем иерархический алгоритм на Python:

1. Сначала импортируем `AgglomerativeClustering` из библиотеки `sklearn.cluster`, а также пакеты `pandas` и `numpy`:

```
from sklearn.cluster import AgglomerativeClustering
import pandas as pd
import numpy as np
```

2. Создадим 20 точек данных в двумерном пространстве задачи:

```
dataset = pd.DataFrame({
    'x': [11, 21, 28, 17, 29, 33, 24, 45, 45, 52, 51, 52, 55, 53,
55, 61, 62, 70, 72, 10],
    'y': [39, 36, 30, 52, 53, 46, 55, 59, 63, 70, 66, 63, 58, 23,
14, 8, 18, 7, 24, 10]
})
```

3. Создадим иерархический кластер, указав гиперпараметры. Используем функцию `fit_predict` для фактической работы алгоритма:

```
cluster = AgglomerativeClustering(n_clusters=2,
affinity='euclidean', linkage='ward')
cluster.fit_predict(dataset)
```

4. Теперь рассмотрим связь каждой точки данных с двумя созданными кластерами (рис. 6.12).

```
In [3]:  1 print(cluster.labels_)

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

Рис. 6.12

Вы можете увидеть, что кластеры используются похожим образом как в иерархических алгоритмах, так и в алгоритмах *k*-средних.

Оценка кластеров

Цель качественной кластеризации состоит в том, чтобы точки данных, принадлежащие отдельным кластерам, были дифференцируемыми. Это подразумевает следующее:

- Точки данных, принадлежащие к одному и тому же кластеру, должны быть как можно более похожими.

- Точки данных, принадлежащие к отдельным кластерам, должны быть как можно более отличающимися.

Иногда пользователь может самостоятельно оценить результаты кластеризации благодаря визуализации кластеров. Однако для количественной оценки качества кластеров существуют математические методы, например *анализ силуэта*. С его помощью сравниваются плотность и разделение в кластерах, созданных с помощью алгоритма *k*-средних. График силуэта отображает близость каждой точки в определенном кластере по отношению к другим точкам в соседних кластерах. Каждому кластеру присваивается коэффициент в диапазоне [–0, 1]. В табл. 6.2 показано, что означают числа в этом диапазоне.

Таблица 6.2

Диапазон	**Значение**	**Описание**
0.71–1.0	Отлично	Кластеризация методом *k*-средних привела к созданию групп, которые достаточно отличимы друг от друга
0.51–0.70	Приемлемо	Кластеризация методом *k*-средних привела к созданию групп, которые более или менее отличимы друг от друга
0.26–0.50	Слабо	Кластеризация методом *k*-средних привела к группировке, но на качество результата полагаться не стоит
< 0.25	Кластеризации не обнаружено	С имеющимися данными и при выбранных параметрах не удалось провести кластеризацию методом *k*-средних

Обратите внимание, что каждый кластер в пространстве задачи получает отдельную оценку.

Применение кластеризации

Кластеризация используется везде, где нужно обнаружить базовые закономерности в наборах данных.

Кластеризация может использоваться, например, в следующих областях государственного управления:

- Анализ очагов преступности.
- Социально-демографический анализ.

В исследованиях рынка кластеризация применяется для следующих целей:

- Сегментация рынка.
- Таргетированная реклама.
- Категоризация клиентов.

Для общего исследования данных и удаления шума из данных реального времени также применяется *метод главных компонент*. Пример использования — торговля на фондовом рынке.

СНИЖЕНИЕ РАЗМЕРНОСТИ

Каждый признак данных соответствует размерности в пространстве задачи. Минимизация количества признаков с целью упрощения пространства задачи называется *снижением размерности* (dimensionality reduction). Его можно произвести двумя способами:

- *Отбор признаков* (feature selection). Определение набора признаков, значимых в контексте решаемой задачи.
- *Агрегация признаков* (feature aggregation). Объединение двух или более признаков для снижения размерности с использованием одного из следующих алгоритмов:
  - *метод главных компонент, PCA* (principal component analysis). Линейный алгоритм неконтролируемого машинного обучения;
  - *линейный дискриминантный анализ, LDA* (linear discriminant analysis). Линейный алгоритм машинного обучения с учителем;
  - *ядерный метод главных компонент, KPCA* (kernel principal component analysis). Нелинейный алгоритм.

Давайте более подробно рассмотрим один из популярных алгоритмов снижения размерности, а именно PCA.

Метод главных компонент (PCA)

PCA — это метод машинного обучения без учителя, который может быть использован для снижения размерности с помощью линейного преобразования. На рис. 6.13 мы видим две главные компоненты, PC1 и PC2, которые показывают форму распространения точек данных. PC1 и PC2 можно использовать для суммирования точек данных с соответствующими коэффициентами.

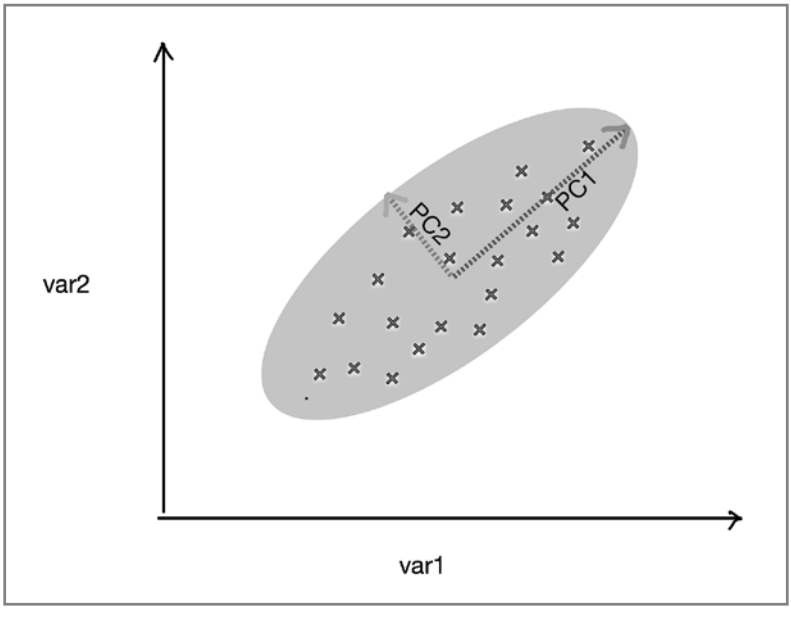


Рис. 6.13

Давайте рассмотрим следующий код:

```
from sklearn.decomposition import PCA
iris = pd.read_csv('iris.csv')
X = iris.drop('Species', axis=1)
pca = PCA(n_components=4)
pca.fit(X)
```

Теперь выведем коэффициенты нашей модели PCA (рис. 6.14).

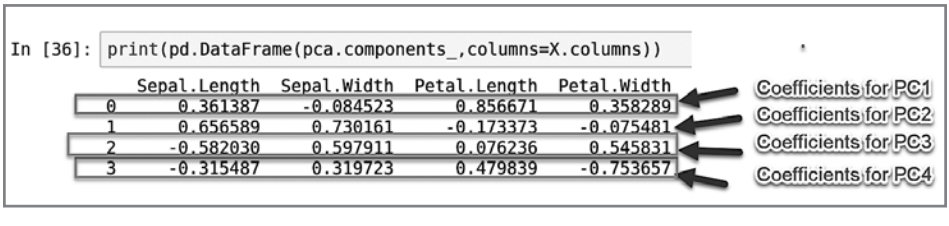


Рис. 6.14

Обратите внимание, что исходный DataFrame имеет четыре признака: `Sepal.Length`, `Sepal.Width`, `Petal.Length` и `Petal.Width`. На рис. 6.14 указаны коэффициенты четырех главных компонент, PC1, PC2, PC3 и PC4. Например, в первой

строке указаны коэффициенты PC1, которые можно использовать для замены исходных четырех переменных.

Основываясь на этих коэффициентах, мы можем рассчитать компоненты PCA для входного DataFrame X:

```
pca_df=(pd.DataFrame(pca.components_,columns=X.columns))

# Рассчитаем PC1, используя генерируемые коэффициенты
X['PC1'] = X['Sepal.Length']* pca_df['Sepal.Length'][0] + X['Sepal.Width']*
pca_df['Sepal.Width'][0]+ X['Petal.Length']*
pca_df['Petal.Length'][0]+X['Petal.Width']* pca_df['Petal.Width'][0]

# Вычислим PC2
X['PC2'] = X['Sepal.Length']* pca_df['Sepal.Length'][1] + X['Sepal.Width']*
pca_df['Sepal.Width'][1]+ X['Petal.Length']*
pca_df['Petal.Length'][1]+X['Petal.Width']* pca_df['Petal.Width'][1]

# Вычислим PC3
X['PC3'] = X['Sepal.Length']* pca_df['Sepal.Length'][2] + X['Sepal.Width']*
pca_df['Sepal.Width'][2]+ X['Petal.Length']*
pca_df['Petal.Length'][2]+X['Petal.Width']* pca_df['Petal.Width'][2]

# Вычислим PC4
X['PC4'] = X['Sepal.Length']* pca_df['Sepal.Length'][3] + X['Sepal.Width']*
pca_df['Sepal.Width'][3]+ X['Petal.Length']*
pca_df['Petal.Length'][3]+X['Petal.Width']* pca_df['Petal.Width'][3]
```

Далее выведем X после расчета компонент PCA (рис. 6.15).

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	PC1	PC2	PC3	PC4
0	5.1	3.5	1.4	0.2	2.818240	5.646350	-0.659768	0.031089
1	4.9	3.0	1.4	0.2	2.788223	5.149951	-0.842317	-0.065675
2	4.7	3.2	1.3	0.2	2.613375	5.182003	-0.613952	0.013383
3	4.6	3.1	1.5	0.2	2.757022	5.008654	-0.600293	0.108928
4	5.0	3.6	1.4	0.2	2.773649	5.653707	-0.541773	0.094610
...	...	...	...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	7.446475	5.514485	-0.454028	-0.392844
146	6.3	2.5	5.0	1.9	7.029532	4.951636	-0.753751	-0.221016
147	6.5	3.0	5.2	2.0	7.266711	5.405811	-0.501371	-0.103650
148	6.2	3.4	5.4	2.3	7.403307	5.443581	0.091399	-0.011244
149	5.9	3.0	5.1	1.8	6.892554	5.044292	-0.268943	0.188390

Рис. 6.15

Теперь выведем долю объясненной дисперсии и попытаемся оценить результаты использования PCA (рис. 6.16).

```
In [37]: print(pca.explained_variance_ratio_)
         [0.92461872 0.05306648 0.01710261 0.00521218]
```

Рис. 6.16

Доля объясненной дисперсии указывает на следующее:

- Если мы заменим четыре исходных признака на PC1, то сможем объяснить около 92,3 % дисперсии исходных переменных. Не объяснив 100 % дисперсии, мы вводим некоторые приближения.
- Если мы заменим четыре исходных признака на PC1 и PC2, то объясним дополнительные 5,3 % дисперсии исходных переменных.
- Если мы заменим четыре исходных признака на PC1, PC2 и PC3, мы объясним еще 0,017 % дисперсии исходных переменных.
- Если мы заменим четыре исходных признака четырьмя основными компонентами, то мы объясним 100 % дисперсии исходных переменных (92.4 + 0.053 + 0.017 + 0.005). Однако такая замена бессмысленна, поскольку этим мы не снизим размерность и ничего не добьемся.

Ограничения PCA

Ниже приведены ограничения PCA:

- PCA может использоваться только для непрерывных переменных и не подойдет для категориальных.
- При агрегировании PCA аппроксимирует переменные компонентов; это снижает размерность за счет точности. Этот компромисс следует тщательно изучить перед использованием PCA.

ПОИСК АССОЦИАТИВНЫХ ПРАВИЛ

Закономерности в наборе данных — это клад, который необходимо найти, исследовать и извлечь из него информацию. Для этого существует ключевой набор алгоритмов. Один из наиболее популярных алгоритмов в этом классе — *поиск*

ассоциативных правил (association rules mining). Он предоставляет нам следующие возможности:

- Измерение частоты закономерности.
- Установление *причинно-следственных* связей между закономерностями.
- Количественная оценка полезности закономерностей (сравнение их точности со случайным угадыванием).

Примеры использования

Поиск ассоциативных правил используется, когда мы пытаемся исследовать причинно-следственные связи между различными переменными датасета. Ниже приведены примеры вопросов, на которые он способен дать ответ:

- Какие значения влажности, облачности и температуры могут привести к завтрашнему дождю?
- Какой вид страхового возмещения может свидетельствовать о мошенничестве?
- Какие комбинации лекарств могут привести к осложнениям для пациентов?

Анализ рыночной корзины

Механизмы рекомендаций будут подробно исследованы в главе 10. *Анализ рыночной корзины* (market basket analysis) — это наиболее простой способ выработки рекомендаций. Для анализа корзины нужна только информация о том, какие товары были куплены вместе. Никакая информация о покупателе или о том, понравился ли ему конкретный товар, не требуется. Обратите внимание, что получить эти данные гораздо проще, чем данные о рейтингах.

Например, такого рода данные генерируются, когда мы совершаем покупки в супермаркете, и для их получения не нужно никакой специальной методики. Такие данные, собранные в течение определенного периода времени, называются *транзакционными данными*. Анализ рыночной корзины осуществляется путем применения поиска ассоциативных правил к транзакционным данным об одновременных покупках в круглосуточных магазинах, супермаркетах и сетях быстрого питания. Он измеряет условную вероятность совместной покупки набора товаров, что помогает ответить на следующие вопросы:

- Каково оптимальное размещение товаров на полках?
- Как стоит располагать товары в рекламном каталоге?

- Что следует порекомендовать потребителю, исходя из его покупательского поведения?

Поскольку анализ рыночной корзины позволяет оценить, как товары связаны друг с другом, он часто используется для розничной торговли — в супермаркетах, круглосуточных магазинах, аптеках и сетях быстрого питания. Преимущество этого анализа заключается в том, что результаты почти не требуют пояснений, а это значит, что они понятны бизнес-пользователям.

Давайте взглянем на типичный супермаркет. Все уникальные товары, доступные в магазине, могут быть представлены набором, $\pi = \{item_1, item_2, ..., item_m\}$. Итак, если в этом супермаркете продается 500 различных товаров, то это π будет набор размером в 500 элементов.

Люди покупают товары в магазине. Каждый раз, когда происходит оплата, товар добавляется к списку предметов в определенной транзакции, называемому *предметным набором* (itemset). За определенный период времени транзакции группируются в набор, представленный Δ, где $\Delta = \{t_1, t_2, ..., t_n\}$.

Приведем пример простого набора транзакционных данных, состоящего всего из четырех транзакций (табл. 6.3).

Таблица 6.3

t1	Wickets, pads
t2	Bat, wickets, pads, helmet
t3	Helmet, ball
t4	Bat, pads, helmet

Разберем этот пример более подробно:

π = {bat, wickets, pads, helmet, ball}[1] — набор всех уникальных предметов, доступных в магазине.

Рассмотрим одну из транзакций в Δ : t3. Обратите внимание, что предметы, купленные в t3, можно представить как $itemset_{t3}$ = {helmet, ball}, что указывает на то, что клиент приобрел оба предмета. Поскольку в этом предметном наборе два элемента, размер $itemset_{t3}$ считается равным двум.

[1] Bat — бита, wickets — ворота, pads — наколенники, helmet — шлем, ball — мяч. — *Примеч. пер.*

Ассоциативные правила

Ассоциативное правило математически описывает элементы взаимосвязи, участвующие в различных транзакциях. В правиле исследуется взаимосвязь между двумя предметными наборами в виде $X \Rightarrow Y$, где $X \subset \pi$, $Y \subset \pi$. Кроме того, X и Y являются непересекающимися предметными наборами; это означает, что $X \cap Y = \varnothing$.

Ассоциативное правило может быть описано так:

$$\{helmet,balls\} \Rightarrow \{bike\}.$$

Здесь *{helmet,ball}* — это X, а *{bike}* — Y.

Типы правил

Запуск алгоритмов ассоциативного анализа обычно приводит к возникновению большого количества правил из набора данных транзакций. Большинство из них бесполезно. Чтобы выбрать правила, которые могут дать нам полезную информацию, мы можем разбить их на три типа:

- тривиальные (trivial);
- необъяснимые (inexplicable);
- полезные (actionable).

Давайте подробно рассмотрим каждый из типов.

Тривиальные правила

Среди большого количества правил многие оказываются бесполезными, поскольку они описывают общеизвестные факты о бизнесе. Такие правила называются *тривиальными* (trivial rules). Даже если достоверность таких правил высока, они все равно не могут быть использованы для принятия решений, основанных на данных. Мы можем с чистой совестью игнорировать все тривиальные правила.

Ниже приведены примеры тривиальных правил:

- любой, кто прыгнет с высотного здания, скорее всего, погибнет;
- усердная учеба приводит к лучшим результатам на экзаменах;
- продажи домашних нагревателей увеличиваются по мере понижения температуры;

- вождение автомобиля с превышением скорости на шоссе приводит к более высокой вероятности аварии.

Необъяснимые правила

Среди правил, которые генерируются после запуска алгоритма ассоциативных правил, наиболее сложными в использовании являются те, которые не имеют очевидного объяснения. Обратите внимание, что правило считается полезным, только если оно помогает нам обнаружить и понять новую закономерность с четкой причинно-следственной связью. Если это не так и мы не можем объяснить, почему событие *X* привело к событию *Y*, то это *необъяснимое правило* (inexplicable rule). Оно представляет собой математическую формулу, которая отражает бессмысленную связь между двумя не зависящими друг от друга событиями.

Примеры необъяснимых правил:

- люди, которые носят красные рубашки, как правило, лучше сдают экзамены;
- зеленые велосипеды воруют с большей частотой;
- люди, которые покупают соленые огурцы, в итоге покупают также и подгузники.

Полезные правила

Полезные правила (actionable rule) — это и есть клад, за которым мы охотимся. Они понятны бизнесу и ведут к новым идеям. Это отличный инструмент для выявления причин тех или иных явлений в бизнесе. Например, полезные правила, касающиеся покупательских привычек, помогут выбрать лучшее расположение товара в магазине. Они подскажут, какие товары стоит разместить на одной полке, чтобы увеличить шансы на продажу, поскольку их обычно покупают вместе.

Ниже приведены примеры полезных правил и соответствующих решений:

- *Правило 1*. Показ рекламы в аккаунтах пользователей в социальных сетях повышает вероятность продаж.

  Полезное применение. Внедрение альтернативных способов рекламы продукта.

- *Правило 2*. Создание большего количества позиций в прайс-листе увеличивает вероятность продаж.

  Полезное применение. Один товар может участвовать в распродаже, в то время как цена на другой товар повышается.

Оценка качества правила

Ассоциативные правила оцениваются по трем показателям:

- поддержка (частота) объектов (Support (frequency) of items);
- достоверность (confidence);
- лифт (lift).

Давайте рассмотрим их более подробно.

Поддержка

Поддержка (support) — это количественная оценка того, насколько часто в нашем наборе данных встречается искомая закономерность. Она рассчитывается путем подсчета повторений интересующего нас шаблона, после чего результат делится на общее количество всех транзакций.

Давайте рассмотрим следующую формулу для конкретного предметного набора $itemset_a$:

$$numItemset_a = Количество\ транзакций,\ содержащих\ itemset_a$$

$$num_{total} = Общее\ количество\ транзакций$$

$$support(itemset_a) = \frac{numItemset_a}{nums_{total}}.$$

Просто взглянув на поддержку, мы можем получить представление о том, насколько часто встречается тот или иной шаблон. Низкая поддержка означает, что мы ищем редкое явление.

Например, если $itemset_a$ = *{helmet,ball}* появляется в двух транзакциях из шести, то $support(itemset_a) = 2/6 = 0.33$.

Достоверность

Достоверность (confidence) — это число, которое количественно определяет, насколько сильно мы можем связать левую часть правила (X) с правой частью (Y), вычисляя условную вероятность. Это вероятность того, что событие X приведет к событию Y, с учетом того, что событие X произошло.

Представим с точки зрения математики правило $X \Rightarrow Y$.

Достоверность этого правила представлена как $confidence(X \Rightarrow Y)$ и измеряется следующим образом:

$$confidence\,(X \Rightarrow Y) = \frac{support(X \cup Y)}{support(X)}.$$

Например, рассмотрим следующее правило:

$$\{helmet, ball\} \Rightarrow \{wickets\}.$$

Достоверность этого правила рассчитывается по следующей формуле:

$$confidence(helmet, ball \Rightarrow wickets) = \frac{support(helmet, ball \cup wickets)}{support(helmet, ball)} = \frac{\frac{1}{6}}{\frac{2}{6}} = 0.5.$$

Это означает, что если у кого-то в корзине есть шлем и мячи *{helmet, balls}*, то существует 50-процентная вероятность (или 0.5) того, что в ней также окажутся и ворота для крикета *{wickets}*.

Лифт

Другой способ оценить качество правила — это рассчитать его *лифт* (lift). Лифт показывает, насколько качественно правило прогнозирует результат по сравнению с простым предположением о результате в правой части уравнения. Если предметные наборы X и Y независимы, то лифт рассчитывается следующим образом:

$$Lift\,(X \Rightarrow Y) = \frac{support(X \cup Y)}{support(X) \times support(Y)}.$$

Алгоритмы анализа ассоциаций

В этом разделе мы рассмотрим два алгоритма, которые можно использовать для анализа ассоциаций:

- *априорный алгоритм* (apriori algorithm). Предложен Р. Агравалом и Р. Шрикантом в 1994 году;

- *алгоритм FP-роста* (FP-growth algorithm). Усовершенствование, предложенное Дж. Ханом и другими в 2001 году.

Давайте рассмотрим каждый из этих алгоритмов.

Априорный алгоритм

Априорный алгоритм — это итеративный и многофазный алгоритм, используемый для генерации правил ассоциации. Он основан на подходе, включающем в себя генерацию и тестирование.

Перед выполнением априорного алгоритма нам необходимо определить две переменные: $support_{threshold}$ и $confidence_{threshold}$ (порог поддержки и порог достоверности).

Алгоритм состоит из двух этапов:

- *Этап генерации кандидатов*. На данном этапе генерируются множества наборов предметов (кандидатов), куда входят наборы со значением поддержки, превышающим порог ($support_{threshold}$).
- *Этап фильтрации*. Алгоритм отфильтровывает все правила ниже ожидаемого значения порога достоверности ($confidence_{threshold}$).

После фильтрации результирующие правила предлагаются в качестве решения.

Ограничения априорного алгоритма

Главным узким местом априорного алгоритма является генерация правил-кандидатов на этапе 1 — например, $\pi = \{item_1, item_2, ..., item_m\}$ может создать 2^m возможных предметных наборов. Из-за своей многофазной структуры алгоритм сначала генерирует наборы предметов, а затем ищет среди них те, которые встречаются часто. Это ограничивает производительность и делает априорный алгоритм непригодным для больших объемов данных.

Алгоритм FP-роста

Алгоритм FP-роста (FP-growth algorithm; FP — frequent pattern — частый шаблон) — это усовершенствование априорного алгоритма. Он представляет частые транзакции в виде упорядоченного FP-дерева. Алгоритм состоит из двух этапов:

- создание FP-дерева;
- поиск частых шаблонов.

Разберем эти шаги подробно.

Создание FP-дерева

Давайте рассмотрим данные транзакций, отраженные в табл. 6.4. Сначала представим их в виде разреженной матрицы.

Таблица 6.4

ID	Bat	Wickets	Pads	Helmet	Ball
1	0	1	1	0	0
2	1	1	1	1	0
3	0	0	0	1	1
4	1	0	1	1	0

Вычислим частоту предметов и отсортируем их в порядке убывания (табл. 6.5).

Таблица 6.5

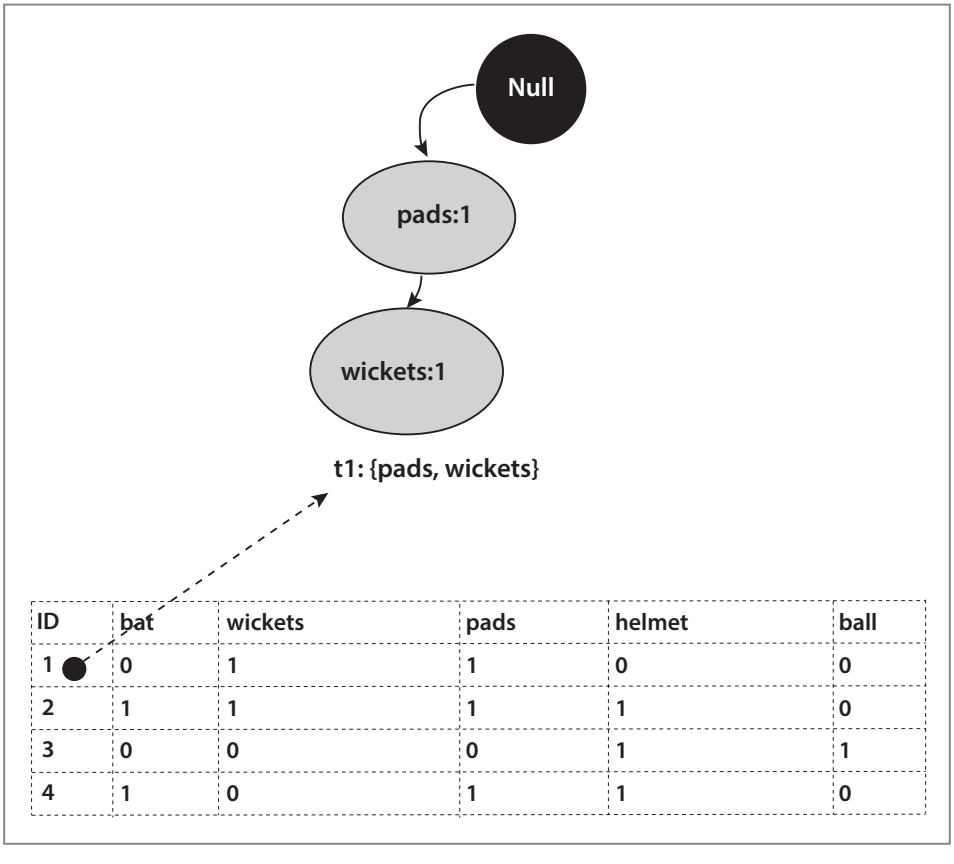
Предмет	Частота
pads	3
helmet	3
bat	2
wicket	2
ball	1

Теперь изменим порядок данных о транзакциях в зависимости от частоты (табл. 6.6).

Таблица 6.6

Транзакция	Исходные предметы	Упорядоченные предметы
t1	Wickets, pads	Pads, wickets
t2	Bat, wickets, pads, helmet	Helmet, pads, wickets, bat
t3	Helmet, ball	Helmet, ball
t4	Bat, pads, helmet	Helmet, pads, bat

Чтобы построить FP-дерево, начнем с его первой ветви. FP-дерево начинается с *Null* в качестве корня. Для начала представим каждый предмет в виде узла (на рис. 6.17, отображена первая транзакция t1). Обратите внимание, что метка каждого узла — это название предмета, а его частота добавляется после двоеточия. Например, предмет *pads* имеет частоту 1.



ID	bat	wickets	pads	helmet	ball
1	0	1	1	0	0
2	1	1	1	1	0
3	0	0	0	1	1
4	1	0	1	1	0

Рис. 6.17

Используя тот же принцип, отобразим все четыре транзакции. В результате у нас получится полное FP-дерево. FP-дерево имеет четыре концевых узла, каждый из которых представляет предметный набор, связанный с одной из четырех транзакций. Мы должны вычислить частоту каждого элемента и увеличить ее при многократном использовании — например, при добавлении t2 в FP-дерево

частота *helmet* увеличивается до двух. Аналогично, при добавлении t4 она будет еще раз увеличена — до трех. Полученное дерево показано на следующей схеме (рис. 6.18).

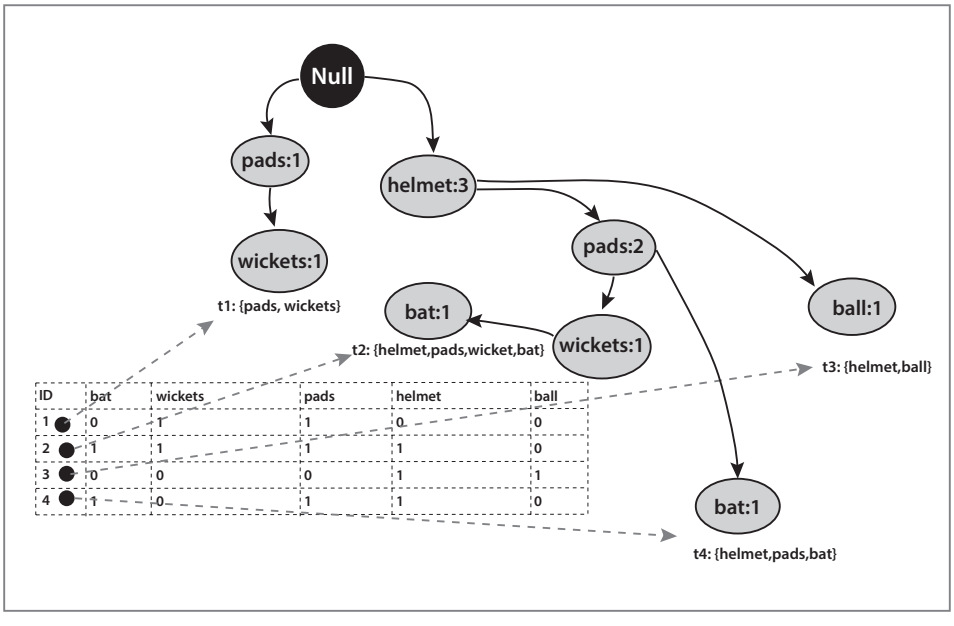


Рис. 6.18

Обратите внимание, что FP-дерево, сгенерированное на схеме, является упорядоченным.

Поиск частых шаблонов

Второй этап алгоритма FP-роста заключается в извлечении частых шаблонов из FP-дерева. Создавая упорядоченное дерево, мы стремимся к эффективной структуре данных, по которой можно легко перемещаться в поисках частых шаблонов.

Мы начинаем с концевого узла и движемся вверх — например, начнем с одного из таких узлов, *bat*. Затем нужно найти *условный базовый шаблон* (conditional pattern base) для *bat*. Он рассчитывается путем указания всех путей от концевого узла до вершины. Условный базовый шаблон для *bat* будет выглядеть следующим образом (табл. 6.7).

Таблица 6.7

Wicket: 1	Pads: 1	Helmet: 1
Pads: 1	Helmet: 1	

Частый шаблон для *bat* будет следующий:

{wicket, pads, helmet} : bat

{pads,helmet} : bat

Реализация алгоритма FP-роста на Python

В этом разделе мы научимся генерировать ассоциативные правила, используя алгоритм FP-роста на Python. Для этого нам понадобится библиотека `pyfpgrowth`. Если вы ранее не использовали `pyfpgrowth`, установите ее:

```
!pip install pyfpgrowth
```

Затем импортируем библиотеки, которые потребуются для реализации алгоритма:

```
import pandas as pd
import numpy as np
import pyfpgrowth as fp
```

Далее введем входные данные в `transactionSet`:

```
dict1 = {
  'id':[0,1,2,3],
  'items':[["wickets","pads"],
  ["bat","wickets","pads","helmet"],
  ["helmet","pad"],
  ["bat","pads","helmet"]]

}
transactionSet = pd.DataFrame(dict1)
```

Как только входные данные введены, сгенерируем шаблоны, которые основаны на параметрах, переданных в `find_frequent_patterns()`. Обратите внимание, что вторым параметром для функции служит *минимальная поддержка*, которая в данном случае равна 1:

```
patterns = fp.find_frequent_patterns(transactionSet['items'],1)
```

Шаблоны сгенерированы. Теперь выведем их. В шаблонах перечислены комбинации предметов и указана их поддержка (рис. 6.19).

```
In [39]: patterns

Out[39]: {('pad',): 1,
          ('helmet', 'pad'): 1,
          ('wickets',): 2,
          ('pads', 'wickets'): 2,
          ('bat', 'wickets'): 1,
          ('helmet', 'wickets'): 1,
          ('bat', 'pads', 'wickets'): 1,
          ('helmet', 'pads', 'wickets'): 1,
          ('bat', 'helmet', 'wickets'): 1,
          ('bat', 'helmet', 'pads', 'wickets'): 1,
          ('bat',): 2,
          ('bat', 'helmet'): 2,
          ('bat', 'pads'): 2,
          ('bat', 'helmet', 'pads'): 2,
          ('pads',): 3,
          ('helmet',): 3,
          ('helmet', 'pads'): 2}
```

Рис. 6.19

Далее сгенерируем правила (рис. 6.20).

```
In [22]: rules = fp.generate_association_rules(patterns,0.3)
         rules

Out[22]: {('helmet',): (('pads',), 0.6666666666666666),
          ('pad',): (('helmet',), 1.0),
          ('pads',): (('helmet',), 0.6666666666666666),
          ('wickets',): (('bat', 'helmet', 'pads'), 0.5),
          ('bat',): (('helmet', 'pads'), 1.0),
          ('bat', 'pads'): (('helmet',), 1.0),
          ('bat', 'wickets'): (('helmet', 'pads'), 1.0),
          ('pads', 'wickets'): (('bat', 'helmet'), 0.5),
          ('helmet', 'pads'): (('bat',), 1.0),
          ('helmet', 'wickets'): (('bat', 'pads'), 1.0),
          ('bat', 'helmet'): (('pads',), 1.0),
          ('bat', 'helmet', 'pads'): (('wickets',), 0.5),
          ('bat', 'helmet', 'wickets'): (('pads',), 1.0),
          ('bat', 'pads', 'wickets'): (('helmet',), 1.0),
          ('helmet', 'pads', 'wickets'): (('bat',), 1.0)}
```

Рис. 6.20

Каждое правило имеет левую и правую стороны, разделенные двоеточием (:). Здесь также указаны значения поддержки для каждого правила.

ПРАКТИЧЕСКИЙ ПРИМЕР — ОБЪЕДИНЕНИЕ ПОХОЖИХ ТВИТОВ В КЛАСТЕРЫ

Алгоритмы машинного обучения без учителя могут применяться в режиме реального времени для объединения похожих твитов. Для этого выполняются следующие операции:

- Шаг 1. *Тематическое моделирование*. Выделить различные темы в заданном наборе твитов.
- Шаг 2. *Кластеризация*. Связать каждый твит с одной из обнаруженных тем.

Такое использование обучения без учителя показано на следующей схеме (рис. 6.21).

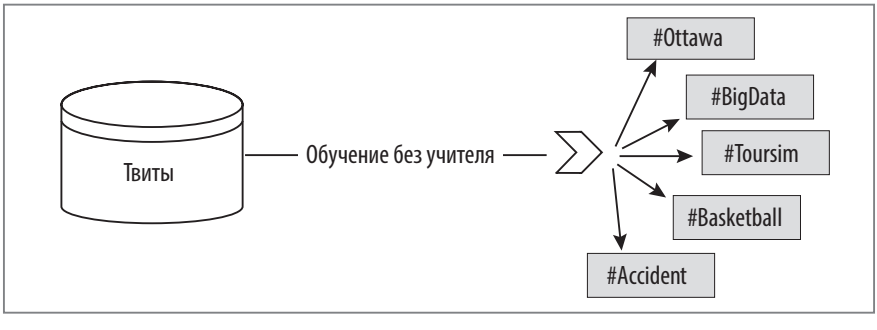


Рис. 6.21

Обратите внимание, что в этом примере требуется обработка входных данных в режиме реального времени.

Рассмотрим обозначенные шаги один за другим.

Тематическое моделирование

Тематическое моделирование — это процесс обнаружения концепций в наборе документов, которые могут быть использованы для их дифференциации. В случае с твитами речь идет о поиске наиболее подходящих тем, на которые можно разделить набор твитов. Латентное размещение Дирихле (latent Dirichlet allocation) — популярный алгоритм, который используется для выделения тем. Поскольку каждый твит представляет собой короткий текст из 144 символов, обычно посвященный определенной теме, для наших нужд мы можем написать более простой алгоритм. Алгоритм состоит из следующих операций.

1. Разбить твиты на лексемы.
2. Произвести предварительную обработку данных. Удалить стоп-слова, цифры, символы и выполнить стемминг (поиск основы слова).
3. Создать *терм-документальную матрицу* (Term-Document-Matrix, TDM) для твитов. Отобрать 200 слов, которые чаще всего появляются в уникальных твитах.
4. Отобрать 10 слов, которые прямо или косвенно обозначают концепцию или тему. Например, «мода», «Нью-Йорк», «программирование», «происшествие». Эти 10 слов теперь являются темами, которые мы успешно обнаружили и которые в дальнейшем станут центрами кластеров для твитов.

Теперь перейдем к следующему шагу — кластеризации.

Кластеризация

Как только мы определили темы, назначим их центрами кластеров. Теперь мы можем запустить алгоритм кластеризации методом k-средних, который привяжет каждый твит к одному из центров.

Так выглядит практический пример того, как набор твитов может быть сгруппирован по темам.

АЛГОРИТМЫ ОБНАРУЖЕНИЯ ВЫБРОСОВ (АНОМАЛИЙ)

Согласно словарному определению, *аномалия* — это нечто отличающееся, ненормальное, странное или с трудом классифицируемое. Это отклонение от общепринятого правила. В науке о данных аномалия — это точка данных, которая сильно отклоняется от ожидаемой модели. Методы поиска таких точек данных называются методами обнаружения аномалий, или выбросов.

Рассмотрим некоторые сферы применения алгоритмов обнаружения аномалий:

- мошенничество с кредитными картами;
- обнаружение злокачественной опухоли при магнитно-резонансной томографии (МРТ);
- предотвращение сбоев в кластерах;
- выдача себя за другого на экзаменах;
- несчастные случаи на шоссе.

В следующих разделах мы рассмотрим различные методы обнаружения аномалий.

Использование кластеризации

Алгоритмы кластеризации (например, методом *k*-средних) могут использоваться для группирования похожих точек данных. Если задать определенный порог, то любая точка за его пределами может быть классифицирована как аномалия. Проблема данного подхода заключается в том, что в группу, созданную с помощью кластеризации *k*-средних, могут также попасть аномальные точки. Это влияет на объективность, полезность и точность подхода.

Обнаружение аномалий на основе плотности

При подходе, основанном на *плотности* (density), мы пытаемся найти плотные окрестности. Для этой цели можно использовать алгоритм *k-ближайших соседей* (k-nearest neighbors, KNN). Отклонения, которые находятся далеко от обнаруженных плотных окрестностей, помечаются как аномалии.

Метод опорных векторов

Для изучения границ точек данных используется *метод опорных векторов* (SVM, support vector machine). Любые точки за пределами обнаруженных границ идентифицируются как аномалии.

РЕЗЮМЕ

В этой главе мы изучили методы машинного обучения без учителя. Мы узнали способы снижения размерности задачи и рассмотрели ситуации, в которых это требуется. Кроме того, мы познакомились с практическими примерами применения МО без учителя (анализ рыночной корзины и обнаружение аномалий).

В следующей главе мы рассмотрим различные методы обучения с учителем. Начнем с линейной регрессии, а затем перейдем к более сложным методам МО с учителем, таким как алгоритмы на основе дерева решений, SVM и XGBoast. Мы также изучим наивный алгоритм Байеса, который лучше всего подходит для неструктурированных текстовых данных.

7

Традиционные алгоритмы обучения с учителем

Приступим к изучению алгоритмов МО с учителем. Это один из наиболее важных типов современных алгоритмов. Отличительная особенность таких алгоритмов — использование размеченных данных для обучения модели. Теме обучения с учителем посвящены две главы нашей книги. В главе 7 мы коснемся всех традиционных алгоритмов обучения с учителем, за исключением нейронных сетей. Нейронные сети — обширная и непрерывно развивающаяся область, поэтому она заслуживает отдельной главы.

В первую очередь мы познакомимся с основными принципами машинного обучения с учителем и рассмотрим два типа моделей МО — *классификаторы* и *регрессоры*. Чтобы продемонстрировать возможности классификаторов, используем шесть алгоритмов классификации для решения одной практической задачи. Затем попробуем решить задачу с помощью трех алгоритмов регрессии. В итоге сравним результаты и обобщим полученные знания.

Наша цель — разобрать различные методы МО с учителем и выяснить, какие из них лучше всего подходят для определенных классов задач.

Итак, в этой главе обсудим следующие темы:

- Машинное обучение с учителем.
- Алгоритмы классификации.

- Методы оценки производительности классификаторов.
- Алгоритмы регрессии.
- Методы оценки производительности регрессионных алгоритмов.

Начнем с основных концепций МО с учителем.

МАШИННОЕ ОБУЧЕНИЕ С УЧИТЕЛЕМ

Машинное обучение опирается на анализ данных и создает автономные системы, которые помогают нам принимать решения (под контролем человека или же без него). Для этого используется ряд алгоритмов и методологий обнаружения и формулировки повторяющихся закономерностей в данных. Одной из самых популярных и мощных методологий, используемых в МО, является *машинное обучение с учителем* (supervised machine learning). Алгоритму предоставляются набор входных данных, называемых *признаками*, и соответствующие им выходные данные, называемые *целевыми переменными*. Затем алгоритм использует этот набор данных для обучения модели, которая отражает сложную взаимосвязь между признаками и целевыми переменными, представленную математической формулой. Обученная модель служит основным средством предсказания.

Предсказания делаются путем генерации целевой переменной для незнакомого набора признаков с помощью обученной модели.

Способность алгоритма обучения с учителем учиться на основе имеющихся данных аналогична способности человеческого мозга учиться на собственном опыте. Это свойство открывает возможности развития искусственного интеллекта и делегирования машинам принятия решений.

Обратимся к примеру. Применим метод МО с учителем для обучения модели, способной сортировать набор электронных писем на нужные (называемые *legit*) и нежелательные (называемые *spam*). Для начала нам понадобятся примеры из прошлого, чтобы машина могла узнать, какое содержимое электронных писем следует классифицировать как спам. Обучение на основе контента для текстовых данных — сложный процесс и требует использования алгоритма МО с учителем. Некоторые алгоритмы, которые можно применить для обучения модели в нашем примере, включают деревья решений и наивные байесовские классификаторы (их мы обсудим позже).

Терминология машинного обучения с учителем

Прежде чем углубиться в детали, определим некоторые основные термины МО с учителем (табл. 7.1).

Таблица 7.1

Термин	Объяснение
Целевая переменная (target variable)	Переменная, которую мы прогнозируем с помощью нашей модели. В модели обучения с учителем может быть только одна целевая переменная
Метка (label)	Если прогнозируемая целевая переменная является категориальной, она называется меткой
Признаки (features)	Набор входных переменных, используемых для предсказания метки
Конструирование признаков (feature engineering)	Преобразование признаков с целью подготовить их к работе с выбранным алгоритмом МО с учителем
Вектор признаков (feature vector)	Прежде чем предоставить алгоритму обучения с учителем необходимые данные, все признаки объединяются в структуру данных, называемую вектором признаков
Исторические данные (historical data)	Данные за прошлый период, используемые для формулирования взаимосвязи между целевой переменной и признаками. Исторические данные включают в себя примеры для обучения
Обучающие/контрольные данные (training/testing data)	Исторические данные с примерами для обучения делятся на две части: обучающие данные (большая часть) и контрольные (меньшая)
Модель (model)	Математическая формулировка закономерностей, которые наилучшим образом отражают взаимосвязь между целевой переменной и признаками
Обучение (training)	Создание модели с использованием обучающих данных
Тестирование (testing)	Оценка качества обученной модели с использованием контрольных данных
Предсказание (prediction)	Использование модели для предсказания целевой переменной

Модель, обученная с помощью МО с учителем, способна делать предсказания, оценивая целевую переменную на основе признаков.

Введем обозначения, которые мы будем использовать в обсуждении методов машинного обучения (табл. 7.2).

Таблица 7.2

Переменная	Термин
y	Фактическая метка
y'	Предсказываемая метка
d	Общее количество примеров
b	Количество обучающих примеров
c	Количество контрольных примеров

Теперь разберем применение этих понятий на практике.

Как уже упоминалось, *вектор признаков* — это структура данных, в которой хранятся все признаки.

Допустим, количество признаков равно n, количество обучающих примеров — b, а `X_train` представляет собой обучающий вектор признаков. Каждый пример — это строка в векторе признаков.

Набору обучающих данных соответствует вектор признаков `X_train`. Если в наборе данных представлено b примеров, то в `X_train` будет b строк. Если в наборе данных есть n переменных, то в нем будет и n столбцов. Таким образом, набор обучающих данных имеет размерность $n \times b$ (рис. 7.1).

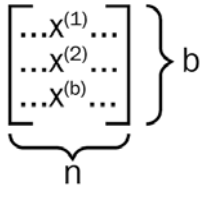


Рис. 7.1

Теперь предположим, что у нас имеется b примеров для обучения и c контрольных примеров. Конкретный пример обучения представлен как (X, y).

Используем верхний индекс для указания места обучающего примера в наборе.

Итак, размеченный набор данных представлен как $D = \{X^{(1)}, y^{(1)}), (X^{(2)}, y^{(2)}), ..., (X^{(d)}, y^{(d)})\}$.

Разделим его на две части — D_{train} и D_{test}.

Таким образом, обучающий набор можно представить как $D_{train} = \{X^{(1)}, y^{(1)}), (X^{(2)}, y^{(2)}), ..., (X^{(b)}, y^{(b)})\}$.

Цель обучения модели состоит в том, чтобы для любого *i*-го примера в обучающем наборе предсказываемое значение целевой переменной стало как можно ближе к фактическому значению в примерах.

Другими словами, $y'(i) \approx y(i)$; при $1 \leq i \leq b$.

Итак, контрольный набор можно представить как $D_{test} = \{X^{(1)}, y^{(1)}), (X^{(2)}, y^{(2)}), ..., (X^{(c)}, y^{(c)})\}$.

Значения целевой переменной представлены вектором *Y*:

$$Y = \{y^{(1)}, y^{(2)}, ..., y^{(m)}\}.$$

Благоприятные условия

Обучение с учителем основано на способности алгоритма обучать модель на примерах. Для этого необходимы следующие благоприятные условия.

- *Достаточное количество примеров*. Алгоритму требуется достаточное количество примеров для обучения модели.
- *Закономерности в исторических данных*. Примеры, используемые для обучения модели, должны содержать закономерности. Вероятность наступления интересующего нас события должна зависеть от сочетания закономерностей, тенденций и событий. Без них мы имеем дело со случайными данными, которые нельзя использовать для обучения модели.
- *Допустимые предположения*. При обучении модели ожидается, что предположения, касающиеся обучающих примеров, будут действительны и в будущем. Рассмотрим реальную задачу. Представим, что правительство поручило нам подготовить модель МО, которая может предсказать вероятность выдачи визы студенту. Подразумевается, что законы и процедуры не изменятся к тому моменту, когда модель будет использоваться. Если же они по-

меняются после обучения модели, ее нужно будет переподготовить, чтобы добавить в нее новую информацию.

Различие между классификаторами и регрессорами

В модели МО целевой переменной может быть *категориальная переменная* (category variable) или же *непрерывная переменная* (continuous variable). Тип переменной определяет тип модели. Существуют два типа моделей МО с учителем.

- *Классификаторы*. Если целевая переменная является категориальной переменной, модель МО называется классификатором. Классификаторы могут быть использованы для ответа, например, на следующие вопросы:
  - Является ли аномальный рост ткани злокачественной опухолью?
  - Будет ли завтра дождь, если исходить из текущих погодных условий?
  - Основываясь на профиле конкретного заявителя, следует ли одобрить его заявку на ипотеку?
- *Регрессоры*. Если целевая переменная является непрерывной, модель МО является регрессором.

  Регрессоры могут быть использованы для ответа, например, на следующие вопросы:
  - Исходя из текущих погодных условий, какое количество осадков ожидать завтра?
  - Сколько будет стоить конкретный дом с определенными характеристиками?

Давайте рассмотрим как классификаторы, так и регрессоры более подробно.

АЛГОРИТМЫ КЛАССИФИКАЦИИ

Если целевая переменная в машинном обучении с учителем является категориальной, то модель называется классификатором. При этом:

- целевая переменная называется *меткой*;
- исторические данные называются *размеченными данными*;
- данные, для которых необходимо предсказать метку, называются *неразмеченными данными*.

Способность точно размечать данные с помощью обученной модели — это и есть главная сила алгоритмов классификации. Классификаторы предсказывают метки для неразмеченных данных, отвечая тем самым на конкретный бизнес-вопрос.

Прежде чем углубиться в детали работы алгоритмов классификации, рассмотрим бизнес-задачу, на которой мы продемонстрируем работу классификаторов. Мы используем шесть различных алгоритмов для решения одной и той же задачи, что поможет нам сравнить их методологию, подход и производительность.

Задача классификации

Общую задачу для тестирования алгоритмов классификации назовем задачей классификации. Применение нескольких классификаторов для решения одной и той же задачи эффективно по двум причинам.

- Все входные переменные должны быть обработаны и собраны в виде сложной структуры данных — вектора признаков. Использование одного вектора для всех шести алгоритмов позволит избежать повторной подготовки данных.
- Мы сможем сравнить производительность различных алгоритмов, поскольку на входе будем использовать один и тот же вектор признаков.

Задача классификации состоит в том, чтобы предсказать вероятность того, что человек совершит покупку. В розничной торговле залогом увеличения продаж является понимание поведения покупателей. Для этого необходимо проанализировать закономерности в исторических данных. В первую очередь сформулируем задачу.

Постановка задачи

Задача: имея некоторые исторические данные, обучить бинарный классификатор прогнозировать покупку товара на основе профиля клиента.

Прежде всего рассмотрим размеченный набор данных, доступный для решения этой задачи:

$$x \in \Re^b, y \in \{0,1\}.$$

Для случая, когда $y = 1$, назовем это положительным классом, а когда $y = 0$, то отрицательным.

Хотя положительный и отрицательный классы можно выбирать произвольно, рекомендуется определять положительный класс как событие, представляющее интерес. Если в интересах банка мы пытаемся выявить мошенничество, то положительным классом ($y = 1$) должна стать мошенническая транзакция, а не наоборот.

Используем метки:

- фактическая метка, обозначаемая y;
- предсказанная метка, обозначаемая y'.

Обратите внимание, что для нашей задачи фактическое значение метки, найденной в примерах, представлено y. Если в нашем примере кто-то купил товар, мы говорим, что $y = 1$. Предсказанные значения представлены символом y'. Входной вектор признаков x имеет размерность 4. Нам необходимо определить, какова вероятность того, что пользователь совершит покупку, с учетом конкретных входных данных.

Итак, нам требуется рассчитать вероятность того, что $y = 1$ при определенном значении вектора признаков x. Математически это можно представить так:

$$y' = P(y = 1|x), \text{ где } x \in \Re^{n_x}.$$

Теперь научимся обрабатывать и преобразовывать различные входные переменные в вектор признаков x. В следующем разделе подробно обсуждается методология сборки различных частей x с использованием конвейера обработки.

Конструирование признаков с использованием пайплайна обработки данных

Подготовка данных для выбранного алгоритма называется *конструированием признаков* (feature engineering) и является важной частью жизненного цикла машинного обучения. Процесс конструирования разбит на разные этапы, или фазы. Многоступенчатый код, используемый для обработки данных, в совокупности называется *пайплайном данных* (data pipeline). Создание пайплайна данных с применением стандартных этапов обработки, где это возможно, позволяет использовать его многократно и сокращает затраты ресурсов на обучение моделей. Использование хорошо протестированных программных модулей также повышает качество кода.

Создадим многоразовый пайплайн обработки для нашей задачи. Как уже упоминалось, мы подготовим данные один раз, а затем используем их для всех классификаторов.

Импорт данных

Исторические данные для этой задачи хранятся в файле под названием `dataset` в формате `.csv`. Мы будем использовать функцию `pd.read_csv` из pandas для импорта данных в виде фрейма данных:

```
dataset = pd.read_csv('Social_Network_Ads.csv')
```

Отбор признаков

Процесс отбора признаков, соответствующих контексту нашей задачи, так и называется — *отбор признаков* (feature selection.). Это неотъемлемая часть конструирования признаков.

Как только файл импортирован, удаляем столбец `User ID`, который используется для идентификации человека и должен быть исключен при обучении модели:

```
dataset = dataset.drop(columns=['User ID'])
```

Теперь рассмотрим имеющийся набор данных:

```
dataset.head(5)
```

Он выглядит следующим образом (рис. 7.2).

	Gender	Age	Estimated Salary	Purchased
0	Male	19	19,000	0
1	Male	35	20,000	0
2	Female	26	43,000	0
3	Female	27	57,000	0
4	Male	19	76,000	0

Рис. 7.2

Приступим к обработке входного набора данных.

Унитарное кодирование

Многие алгоритмы машинного обучения требуют, чтобы все признаки были непрерывными переменными. Это означает, что если некоторые функции являются категориальными переменными, нужен способ преобразования их в непрерывные. Унитарное кодирование (one-hot encoding) — это один из наиболее эффективных способов выполнения такого преобразования. В нашей задаче единственная категориальная переменная — это `Gender`. Преобразуем ее в непрерывную, используя унитарное кодирование:

```
enc = sklearn.preprocessing.OneHotEncoder()
enc.fit(dataset.iloc[:,[0]])
onehotlabels = enc.transform(dataset.iloc[:,[0]]).toarray()
genders = pd.DataFrame({'Female': onehotlabels[:, 0], 'Male':
onehotlabels[:, 1]})
result = pd.concat([genders,dataset.iloc[:,1:]], axis=1, sort=False)
result.head(5)
```

Как только переменная преобразована, взглянем еще раз на набор данных (рис. 7.3).

	Female	Male	Age	Estimated Salary	Purchased
0	0.0	1.0	19	19,000	0
1	0.0	1.0	35	20,000	0
2	1.0	0.0	26	43,000	0
3	1.0	0.0	27	57,000	0
4	0.0	1.0	19	76,000	0

Рис. 7.3

Обратите внимание, что для преобразования переменной из категориальной в непрерывную с помощью унитарного кодирования мы конвертировали `Gender` в два отдельных столбца — `Male` и `Female`.

Определение признаков и метки

Давайте определим признаки и метки. В этой книге мы будем использовать `y` для обозначения метки, а `X` — для обозначения набора признаков:

```
y=result['Purchased']
X=result.drop(columns=['Purchased'])
```

`X` представляет вектор признаков и содержит все входные переменные, необходимые для обучения модели.

Разделение набора данных на контрольную и обучающую части

Теперь разделим набор данных на контрольные (25 %) и обучающие данные (75 %) с помощью `sklearn.model_selection` `import` `train_test_split`:

```
#from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25,
random_state = 0)
```

Это приведет к созданию следующих четырех структур данных:

- `X_train`: структура данных, содержащая признаки обучающих данных;
- `X_test`: структура данных, содержащая признаки контрольных данных;
- `y_train`: вектор, содержащий значения метки в наборе обучающих данных;
- `y_test`: вектор, содержащий значения метки в наборе контрольных данных.

Масштабирование признаков

Для многих алгоритмов машинного обучения рекомендуется масштабировать переменные от `0` до `1`. Это также называется *нормализацией признаков*. Применим масштабирование:

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Теперь данные готовы к использованию в качестве входных для классификаторов, о которых мы поговорим в следующих разделах.

Оценка классификаторов

Как только модель обучена, нужно оценить ее производительность. Для этого сделаем следующее:

1. Разделим размеченный набор данных на две части — для обучения и для контроля. Контрольная часть используется для оценки обученной модели.
2. С помощью признаков контрольной части создадим метку для каждой строки. Получим набор предсказанных меток.
3. Чтобы оценить модель, сравним предсказанные метки с фактическими.

Если задача не слишком примитивна, при оценке модели будут допущены некоторые ошибки классификации. Интерпретация этих ошибок зависит от выбора метрик производительности.

Получив наборы как фактических, так и предсказанных меток, можно переходить к оценке модели с помощью метрик производительности. Выбор наилучшей метрики для количественной оценки модели зависит от требований поставленной бизнес-задачи, а также от характеристик набора обучающих данных.

Матрица ошибок

Матрица ошибок (confusion matrix) используется для обобщения результатов оценки классификатора. Матрица ошибок для бинарного классификатора выглядит следующим образом (рис. 7.4).

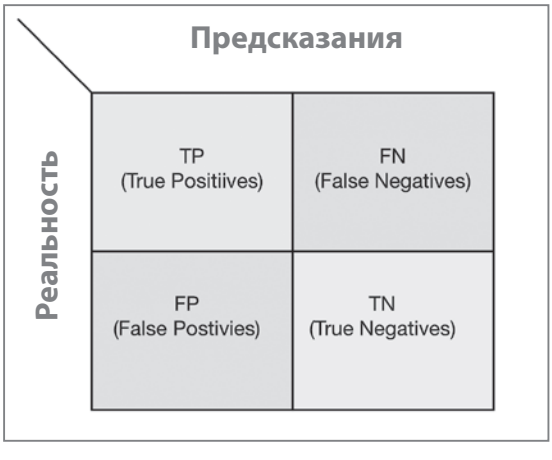


Рис. 7.4

Если метка классификатора, который мы обучаем, имеет два уровня, он называется *двоичным, или бинарным, классификатором*. Первое критически важное использование машинного обучения с учителем относится к Первой мировой войне: двоичный классификатор был применен для различения самолетов и летящих птиц.

Результаты классификации можно разделить на четыре категории:

- *Истинно положительные* (true positives, TP). Верно классифицированный положительный пример.

- *Истинно отрицательные* (true negative, TN). Верно классифицированный отрицательный пример.
- *Ложноположительные* (false positive, FP). Отрицательный пример, классифицированный как положительный.
- *Ложноотрицательные* (false negative, FN). Положительный пример, классифицированный как отрицательный.

В следующем разделе мы научимся использовать эти четыре категории для определения различных показателей производительности.

Метрики производительности

Метрики производительности используются для количественной оценки качества работы обученных моделей. Исходя из этого, определим следующие четыре показателя (табл. 7.3).

Таблица 7.3

Метрика	**Формула**
Доля правильных ответов (accuracy)	$\frac{TP+TN}{TP+TN+FP+FN}$
Полнота (recall)	$\frac{TP}{TP+FN}$
Точность (precision)	$\frac{TP}{TP+FP}$
F-мера (F1 score)	$2\left(\frac{Precision * Recall}{Precision + Recall}\right)$

Доля правильных ответов — это процент корректных классификаций среди всех прогнозов. При расчете этой метрики разница между *TP* и *TN* не учитывается. Оценка модели с точки зрения доли правильных ответов проста, но в определенных случаях она не сработает.

Иногда для оценки производительности модели требуется нечто большее, чем доля правильных ответов. Одна из таких ситуаций — использование модели для прогнозирования редкого события, например:

- предсказание мошеннических транзакций в банковской базе данных;
- предсказание вероятности механического отказа части двигателя воздушного судна.

В обоих случаях мы пытаемся предсказать редкое событие. Для этого нам понадобятся дополнительные метрики — *полнота* и *точность*. Рассмотрим их подробнее.

- *Полнота*. Отражает частоту попаданий. В первом примере это доля обнаруженных и помеченных мошеннических транзакций среди всех таких транзакций. Допустим, в нашем контрольном наборе данных был 1 миллион транзакций, из которых 100 были заведомо мошенническими, и модель смогла идентифицировать 78 из них. В этом случае значение полноты будет равно 78/100.
- *Точность*. Измеряет, сколько транзакций, помеченных моделью, на самом деле являются мошенническими. Вместо того чтобы сосредоточиться на мошеннических транзакциях, которые модель не смогла выявить, мы хотим определить, насколько модель точна в своем выборе.

Обратите внимание, что *F*-мера объединяет полноту и точность: если эти показатели идеальны, то и *F*-мера будет идеальной. Высокий балл *F*-меры означает, что мы подготовили качественную модель, которая обладает оптимальными полнотой и точностью.

Переобучение

Если модель МО дает отличный результат в среде разработки, но гораздо менее эффективна в производственной среде, это значит, что модель *переобучена*. Такая модель точно соответствует набору обучающих данных. Иными словами, правила, созданные моделью, чересчур детализированы. Лучше всего эту идею отражает *дилемма смещения — дисперсии* (bias — variance trade-off). Рассмотрим подробнее эти концепции.

Смещение

Любая модель МО обучается на основе определенных предположений. По сути, такие предположения — это аппроксимация явлений реального мира. Они упрощают фактические взаимосвязи между признаками и их характеристиками, что облегчает обучение модели. Большее количество предположений означает большее *смещение* (bias). При обучении модели сильные допущения дадут высокое смещение, а незначительные — низкое.

В линейной регрессии нелинейность признаков игнорируется, и они рассматриваются как линейные переменные. Таким образом, модели линейной регрессии по своей сути склонны демонстрировать высокое смещение.

Дисперсия

Дисперсия (variance) количественно определяет точность, с которой модель оценивает целевую переменную, если для обучения модели использовался другой набор данных. Она показывает, насколько хорошо математическая формулировка модели обобщает лежащие в ее основе закономерности.

Слишком точные правила, основанные на конкретных ситуациях, дают высокую дисперсию, а общие правила, применимые к различным ситуациям, — низкую.

Наша цель — обучить модели, которые демонстрируют низкое смещение и низкую дисперсию. Это непростая задача, которая лишает сна многих дата-сайентистов.

Дилемма смещения — дисперсии

При обучении модели МО довольно сложно определить уровень обобщения правил, лежащих в ее основе. Это отражено в дилемме смещения — дисперсии.

Обратите внимание, что упрощенные допущения = большее обобщение = низкая дисперсия = высокое смещение.

Соотношение между смещением и дисперсией определяется выбором алгоритма, характеристиками данных и различными гиперпараметрами. Важно достичь оптимального компромисса, опираясь на требования конкретной задачи.

Этапы классификации

Как только размеченные данные подготовлены, мы приступаем к разработке классификаторов. Она включает в себя обучение, оценку и внедрение. Три этапа реализации классификатора представлены в жизненном цикле *CRISP-DM* на следующей диаграмме (рис. 7.5). *CRISP-DM* подробно описан в главе 6.

На первых двух этапах реализации классификатора — контроля и обучения — используются размеченные данные. Они разделены на две части: большую часть — обучающие данные и меньшую часть — контрольные данные. Для разделения данных используется метод случайной выборки. Это позволяет гарантировать, что обе части содержат единые закономерности. Сначала выполняет-

ся обучение модели с помощью обучающих данных, затем — ее проверка посредством контрольных данных. Для оценки работы обученной модели используются различные матрицы производительности. После этого происходит развертывание модели. Обученная модель используется для извлечения информации и решения реальной задачи путем присваивания меток неразмеченным данным (см. рис. 7.5).

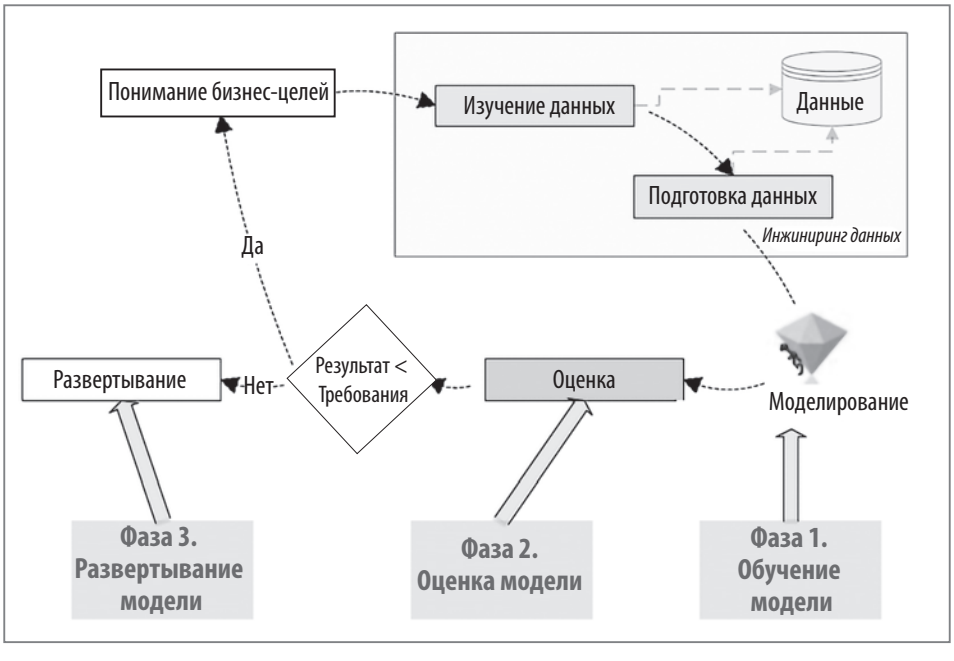


Рис. 7.5

Рассмотрим следующие алгоритмы классификации:

- алгоритм дерева решений (decision tree algorithm);
- алгоритм XGBoost (XGBoost algorithm);
- алгоритм случайного леса (random forest algorithm);
- алгоритм логистической регрессии (logistic regression algorithm);
- метод опорных векторов (SVM, support vector machine);
- наивный байесовский алгоритм (naive Bayes algorithm).

Начнем с алгоритма дерева решений.

Алгоритм дерева решений

Дерево решений основано на рекурсивном методе разделения («разделяй и властвуй»). Алгоритм генерирует набор правил, которые можно использовать для предсказания метки. Он начинается с корневого узла и расходится на несколько ветвей. Внутренние узлы представляют собой проверку по определенному атрибуту, а результат проверки является ответвлением на следующий уровень. Дерево решений заканчивается концевыми узлами, в которых содержатся решения. Процесс останавливается, если разделение больше не улучшает результат.

Этапы алгоритма дерева решений

Отличительной особенностью дерева решений является создание понятной для человека иерархии правил, используемых для предсказания метки. По своей природе это рекурсивный алгоритм. Создание иерархии правил состоит из следующих шагов.

1. *Поиск наиболее важного признака*. Алгоритм определяет признак, который наилучшим образом дифференцирует элементы в обучающем наборе данных относительно метки. Расчет основан на таких критериях, как *прирост информации* (information gain) или *критерий (примесь) Джини* (Gini impurity).
2. *Разделение*. С помощью этого признака алгоритм создает критерий, который используется для разделения обучающего набора данных на две ветви:
   - элементы данных, которые соответствуют критерию;
   - элементы данных, которые не соответствуют критерию.
3. *Проверка на наличие концевых узлов*. Если полученная ветвь в основном содержит метки одного класса, она становится окончательной и завершается концевым узлом.
4. *Проверка условий остановки и повтор*. Если предусмотренные условия остановки не выполнены, алгоритм возвращается к шагу 1 для следующей итерации. В противном случае модель считается обученной, и каждый узел результирующего дерева решений на самом низком уровне помечается как концевой узел. В качестве условия остановки можно просто задать количество итераций. Также можно установить условие, при котором алгоритм завершается, как только достигает определенного уровня однородности для каждого концевого узла.

Алгоритм дерева решений проиллюстрирован на следующей диаграмме (рис. 7.6).

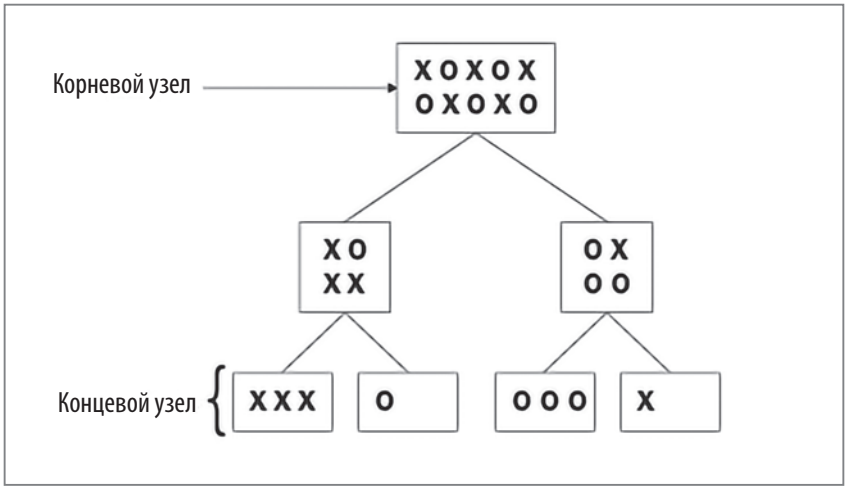


Рис. 7.6

На диаграмме корневой узел содержит множество крестиков и ноликов. Алгоритм создает критерий, по которому пытается отделить нолики от крестиков. На каждом уровне дерево решений создает разделы данных, которые становятся все более и более однородными с уровня 1 и выше. Идеальный классификатор имеет концевые узлы, которые содержат только нолики или крестики. Обучение идеальных классификаторов обычно затруднено из-за присущей набору обучающих данных неупорядоченности.

Использование алгоритма дерева решений для задачи классификации

Теперь используем алгоритм дерева решений на практике, чтобы решить поставленную задачу: нам необходимо предсказать, купит ли клиент некий продукт.

1. Прежде всего создадим экземпляр алгоритма дерева решений и обучим модель, используя обучающую часть данных, подготовленную для классификаторов:

```
classifier = sklearn.tree.DecisionTreeClassifier(criterion =
'entropy', random_state = 100, max_depth=2)
classifier.fit(X_train, y_train)
```

2. С помощью обученной модели предскажем метки для контрольной части размеченных данных. Создадим матрицу ошибок, чтобы обобщить производительность модели:

```
import sklearn.metrics as metrics
y_pred = classifier.predict(X_test)
cm = metrics.confusion_matrix(y_test, y_pred)
cm
```

Получим следующий результат (рис. 7.7).

```
Out[22]: array([[64,  4],
                [ 2, 30]])
```

Рис. 7.7

3. Теперь рассчитаем значения `accuracy`, `recall` и `precision` для созданного классификатора с помощью алгоритма дерева решений:

```
accuracy= metrics.accuracy_score(y_test,y_pred)
recall = metrics.recall_score(y_test,y_pred)
precision = metrics.precision_score(y_test,y_pred)
print(accuracy,recall,precision)
```

4. Выполнение этого кода приведет к следующему результату (рис. 7.8).

```
0.94 0.9375 0.8823529411764706
```

Рис. 7.8

Метрики производительности позволяют сравнить различные методы обучения моделей между собой.

Сильные и слабые стороны классификаторов дерева решений

В этом разделе мы рассмотрим достоинства и недостатки использования алгоритма дерева решений для классификации.

Сильные стороны

- Правила моделей, созданных с помощью алгоритма дерева решений, могут быть интерпретированы людьми. Такие модели называются *«белыми ящиками»* (whitebox models). «Белый ящик» требуется, если нужна прозрачность для отслеживания подробностей и причин решения, принимаемого моделью. Прозрачность имеет огромное значение в тех случаях, где необходимо избежать предвзятости и защитить уязвимые группы населения. Как правило, модель «белый ящик» используется для вынесения важнейших решений в государственных и страховых отраслях.

- Классификаторы на основе дерева решений предназначены для извлечения информации из дискретного пространства задачи. Это означает, что по большей части признаки являются категориальными переменными, поэтому использование дерева решений для обучения модели — хороший выбор.

Слабые стороны

- Если дерево, сгенерированное классификатором, слишком разрастается, то правила становятся слишком детализированными и это приводит к переобучению модели. Этот алгоритм склонен к переобучению, поэтому мы должны «подстригать» дерево по мере необходимости.
- Классификаторы на основе дерева решений не способны фиксировать нелинейные взаимосвязи в создаваемых ими правилах.

Примеры использования

Рассмотрим примеры практического применения алгоритма дерева решений.

Классификация записей

Классификаторы деревьев решений могут использоваться для классификации точек данных, например, в следующих случаях.

- *Заявки на ипотеку*. Бинарный классификатор определяет вероятность некредитоспособности заявителя.
- *Сегментация клиентов*. Классификатор распределяет клиентов по принципу их платежеспособности (высокой, средней и низкой), чтобы можно было настроить маркетинговые стратегии для каждой категории.
- *Медицинский диагноз*. Классификатор различает доброкачественные и злокачественные новообразования.
- *Анализ эффективности лечения*. Классификатор отбирает пациентов, положительно отреагировавших на конкретное лечение.

Отбор признаков

Алгоритм классификации на основе дерева решений выбирает небольшое подмножество признаков, чтобы создать для них правила. Его можно использовать для отбора признаков (если их много) в работе с другим алгоритмом машинного обучения.

Ансамблевые методы

Ансамбль в машинном обучении — это метод создания нескольких слегка отличающихся моделей с использованием разных параметров, а затем объединения их в агрегированную модель. Чтобы создать эффективный ансамбль, необходимо найти обобщающий критерий для итоговой модели. Рассмотрим некоторые ансамблевые алгоритмы.

Реализация градиентного бустинга с помощью алгоритма XGBoost

XGBoost создан в 2014 году и основан на принципах *градиентного бустинга* (gradient boosting). Он стал одним из самых популярных алгоритмов для ансамблей классификации. XGBoost генерирует множество взаимосвязанных деревьев и использует *градиентный спуск*, чтобы минимизировать остаточную ошибку. Это делает его идеально подходящим для распределенных инфраструктур, таких как Apache Spark, или для облачных систем, например Google Cloud или Amazon Web Services (AWS).

Реализуем градиентный бустинг с помощью алгоритма XGBoost:

1. Создадим экземпляр классификатора XGBClassfier и обучим модель, используя обучающую часть данных (рис. 7.9).

```
In [20]: from xgboost import XGBClassifier
         classifier = XGBClassifier()
         classifier.fit(X_train, y_train)

Out[20]: XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
                       colsample_bynode=1, colsample_bytree=1, gamma=0,
                       learning_rate=0.1, max_delta_step=0, max_depth=3,
                       min_child_weight=1, missing=None, n_estimators=100, n_jobs=1,
                       nthread=None, objective='binary:logistic', random_state=0,
                       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
                       silent=None, subsample=1, verbosity=1)
```

Рис. 7.9

2. Сгенерируем предсказания на основе обученной модели:

```
y_pred = classifier.predict(X_test)
cm = metrics.confusion_matrix(y_test, y_pred)
cm
```

Результат будет таким (рис. 7.10).

```
Out[21]: array([[64,  4],
                [ 3, 29]])
```

Рис. 7.10

3. Теперь количественно оценим производительность модели:

```
accuracy= metrics.accuracy_score(y_test,y_pred)
recall = metrics.recall_score(y_test,y_pred)
precision = metrics.precision_score(y_test,y_pred)
print(accuracy,recall,precision)
```

Это дает следующий результат (рис. 7.11).

```
0.93 0.90625 0.8787878787878788
```

Рис. 7.11

Далее рассмотрим алгоритм случайного леса.

Алгоритм случайного леса

Случайный лес — это ансамблевый метод, который работает путем объединения нескольких деревьев решений, чтобы снизить как смещение, так и дисперсию.

Обучение модели

В процессе обучения алгоритм случайного леса берет *N* выборок из обучающих данных и создает *m* подмножеств из общих данных. Подмножества создаются путем случайного выбора некоторых строк и столбцов входных данных. Алгоритм строит *m* независимых деревьев решений. Эти деревья классификации обозначены от C_1 до C_m.

Использование случайного леса для предсказания

Как только модель обучена, ее можно использовать для маркировки новых данных. Каждое из отдельных деревьев создает метку. Окончательное предсказание определяется путем голосования, как показано на рис. 7.12.

Обратите внимание, что на предыдущей диаграмме обучено *m* деревьев, которые обозначены от C_1 до C_m. Иными словами, Trees = $\{C_1, ..., C_m\}$.

Каждое дерево генерирует предсказание, которое представлено набором:

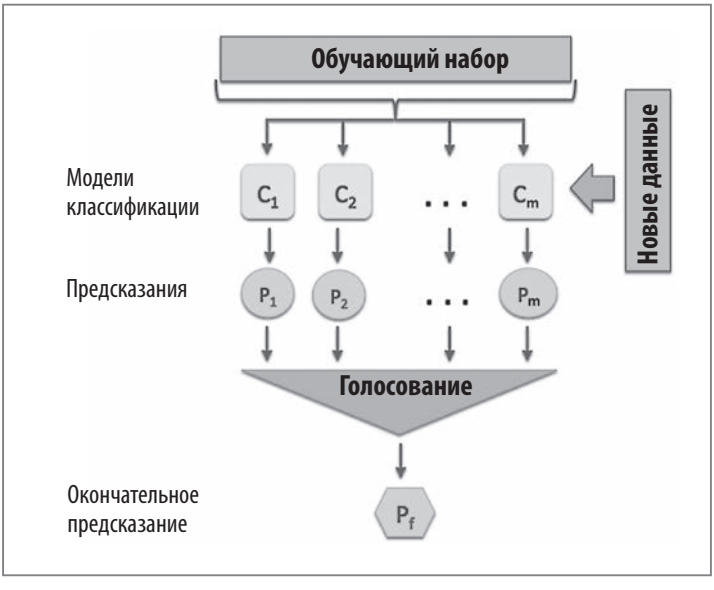


Рис. 7.12

Индивидуальные предсказания = P = {P_1, ..., P_m}.

Окончательное предсказание представлено P_f. Оно определяется большинством индивидуальных предсказаний. Для поиска решения большинства может использоваться функция `mode` (она возвращает число, которое повторяется чаще всего, — моду). Индивидуальное и окончательное предсказания связаны следующим образом:

$$P_f = \text{mode}\ (P).$$

Отличие алгоритма случайного леса от бустинга

Деревья, сгенерированные алгоритмом случайного леса, полностью независимы друг от друга. В них не содержится никакой информации о других деревьях ансамбля. Это отличает алгоритм случайного леса от других методов, таких как бустинг.

Использование алгоритма случайного леса в задаче классификации

Давайте создадим экземпляр алгоритма случайного леса и применим его для обучения модели с помощью обучающих данных.

Здесь мы рассмотрим два ключевых гиперпараметра:

- `n_estimators`
- `max_depth`

Гиперпараметр `n_estimators` регулирует количество построенных отдельных деревьев решений, а `max_depth` — глубину каждого из этих деревьев.

Другими словами, дерево решений может продолжать разделяться до тех пор, пока у него не появится узел, представляющий все примеры из обучающего набора. Задав значение `max_depth`, мы ограничиваем количество уровней разделения. Это контролирует сложность модели и определяет, насколько точно она соответствует обучающим данным. Гиперпараметр `n_estimators` управляет шириной модели случайного леса, а `max_depth` — глубиной модели (рис. 7.13).

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10, max_depth = 4,criterion = 'entropy', random_state = 0)
classifier.fit(X_train, y_train)
```

```
Out[9]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
                               max_depth=4, max_features='auto', max_leaf_nodes=None,
                               min_impurity_decrease=0.0, min_impurity_split=None,
                               min_samples_leaf=1, min_samples_split=2,
                               min_weight_fraction_leaf=0.0, n_estimators=10,
                               n_jobs=None, oob_score=False, random_state=0, verbose=0,
                               warm_start=False)
```

Рис. 7.13

Как только модель случайного леса будет обучена, используем ее для предсказания:

```
y_pred = classifier.predict(X_test)
cm = metrics.confusion_matrix(y_test, y_pred)
cm
```

Получаем результат (рис. 7.14).

```
Out[10]: array([[64,  4],
                [ 3, 29]])
```

Рис. 7.14

Теперь оценим, насколько хороша наша модель:

```
accuracy= metrics.accuracy_score(y_test,y_pred)
recall = metrics.recall_score(y_test,y_pred)
precision = metrics.precision_score(y_test,y_pred)
print(accuracy,recall,precision)
```

Наблюдаем следующий результат (рис. 7.15).

```
0.93 0.90625 0.8787878787878788
```

Рис. 7.15

Далее рассмотрим логистическую регрессию.

Логистическая регрессия

Логистическая регрессия — это алгоритм, используемый для бинарной классификации. Логистическая функция применяется для описания взаимодействия между входными признаками и целевой переменной. Это один из простейших методов классификации, который используется для моделирования бинарной зависимой переменной.

Допущения

Логистическая регрессия предполагает следующее:

- Набор обучающих данных не содержит пропущенных значений.
- Метка представляет собой бинарную категориальную переменную.
- Метка является *порядковой* — другими словами, категориальной переменной с упорядоченными значениями.
- Все признаки или входные переменные независимы друг от друга.

Установление отношений

Для логистической регрессии прогнозируемое значение рассчитывается следующим образом:

$$y' = \sigma(wX + j).$$

Давайте предположим, что $z = wX + j$.

Так что теперь:

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

Предыдущее соотношение может быть графически представлено следующим образом (рис. 7.16).

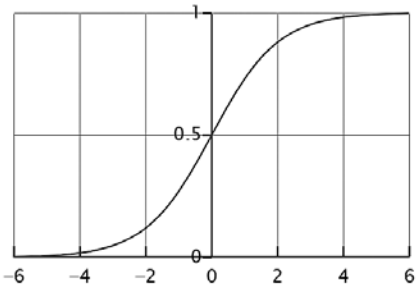


Рис. 7.16

Обратите внимание, что если z велико, $\sigma(z)$ будет равно `1`. Если z очень мало или является большим отрицательным числом, $\sigma(z)$ будет равно `0`. Итак, цель логистической регрессии — найти верные значения для w и j.

Логистическая регрессия названа в честь функции, которая используется для ее описания и называется *логистической* или *сигмоидной функцией*.

Функции потерь и стоимости

Функция потерь (loss function) используется для измерения ошибки в конкретном примере в обучающих данных. *Функция стоимости* (cost function) вычисляет ошибку (в целях ее минимизации) во всем наборе обучающих данных. Функция `loss` используется только для одного примера в наборе. Функция `cost` обозначает общую стоимость: с ее помощью высчитывается суммарное отклонение фактических значений от прогнозируемых. Оно зависит от выбора w и h.

Функция `loss`, используемая в логистической регрессии, выглядит так:

$$Loss\,(y'^{(i)}, y^{(i)}) = -(y^{(i)} \log y'^{(i)} + (1 - y^{(i)}) \log (1 - y'^{(i)}).$$

Обратите внимание, что, когда $y^{(i)} = 1$, $Loss(y'^{(i)}, y^{(i)}) = -\log y'^{(i)}$. Минимизация потерь приведет к большому значению $y'^{(i)}$. Поскольку это сигмоидная функция, максимальное значение будет равно `1`.

Если $y^{(i)} = 0$, $Loss\,(y'^{(i)}, y^{(i)}) = -\log (1 - y'^{(i)})$. Минимизация потерь приведет к тому, что $y'^{(i)}$ будет как можно меньше, то есть `0`.

Функция `cost` в логистической регрессии выглядит следующим образом:

$$Cost(w,\ b) = \frac{1}{b}\sum Loss(y'^{(i)},\ y^{(i)}).$$

Когда применять логистическую регрессию

Логистическая регрессия отлично подходит для бинарных классификаторов. Она не так эффективна, если объем данных огромен, а их качество не самое лучшее. Логистическая регрессия способна фиксировать более простые отношения. Хотя она, как правило, не обеспечивает максимальной производительности, но зато устанавливает очень хороший начальный ориентир.

Использование логистической регрессии в задаче классификации

В этом разделе мы увидим применение алгоритмов логистической регрессии в задаче классификации:

1. Создадим модель логистической регрессии и обучим ее, используя обучающие данные:

```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state = 0)
classifier.fit(X_train, y_train)
```

2. Теперь спрогнозируем значения контрольных данных и создадим матрицу ошибок:

```
y_pred = classifier.predict(X_test)
cm = metrics.confusion_matrix(y_test, y_pred)
cm
```

   После выполнения кода результат будет таким (рис. 7.17).

```
Out[11]: array([[65,  3],
                [ 6, 26]])
```

Рис. 7.17

3. Оценим показатели производительности:

```
accuracy= metrics.accuracy_score(y_test,y_pred)
recall = metrics.recall_score(y_test,y_pred)
precision = metrics.precision_score(y_test,y_pred)
print(accuracy,recall,precision)
```

4. Получим следующий результат (рис. 7.18).

```
0.91 0.8125 0.896551724137931
```

Рис. 7.18

Далее мы рассмотрим *метод опорных векторов (алгоритм SVM)*.

Метод опорных векторов (SVM)

Метод опорных векторов (SVM, support vector machine) — это классификатор, который находит оптимальную гиперплоскость, максимально увеличивающую зазор между двумя классами. Максимизировать зазор — это и есть цель оптимизации с помощью SVM. Зазор определяется как расстояние между разделяющей гиперплоскостью (границей принятия решения) и обучающими выборками, ближайшими к этой гиперплоскости, которые называются *опорными векторами*. Начнем с очень простого примера, где есть лишь два измерения, X_1 и X_2. Нам нужна линия, отделяющая нолики от крестиков. Работа алгоритма показана на следующей схеме (рис. 7.19).

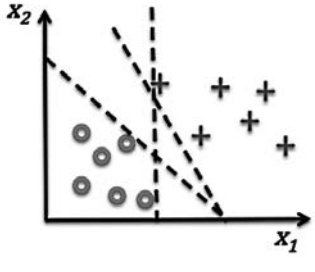


Рис. 7.19

Мы нарисовали две линии, и обе они идеально отделяют крестики от ноликов. Однако нам необходимо определить одну оптимальную линию (или границу принятия решения), которая даст наилучшие шансы правильно классифицировать большинство дополнительных примеров. Разумным выбором может быть линия, расположенная между этими двумя классами и равноудаленная от них, что обеспечит небольшой буфер для каждого класса, как показано на рис. 7.20.

Подготовим классификатор с помощью SVM.

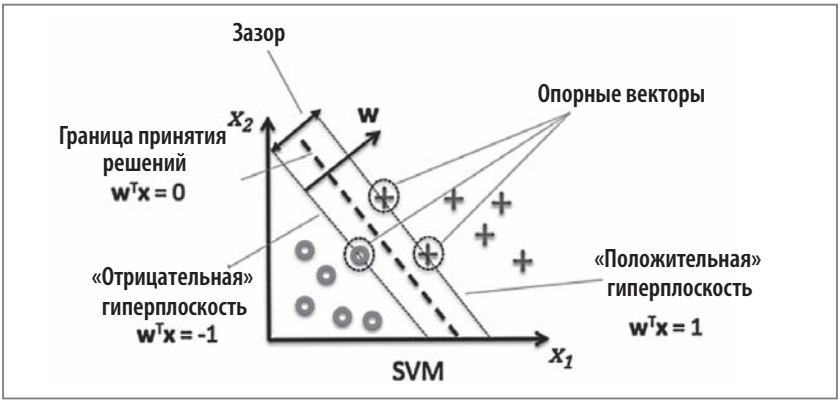


Рис. 7.20

Использование алгоритма SVM для задачи классификации

1. Прежде всего создадим экземпляр классификатора SVM, после чего используем обучающую часть размеченных данных для его обучения. Гиперпараметр `kernel` определяет тип преобразования, применяемого к входным данным, чтобы сделать их линейно разделимыми:

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear', random_state = 0)
classifier.fit(X_train, y_train)
```

2. После обучения сгенерируем несколько предсказаний и посмотрим на матрицу ошибок:

```
y_pred = classifier.predict(X_test)
cm = metrics.confusion_matrix(y_test, y_pred)
cm
```

3. Результат будем следующим (рис. 7.21).

```
Out[9]: array([[66,  2],
               [ 9, 23]])
```

Рис. 7.21

4. Теперь рассмотрим метрики производительности:

```
accuracy= metrics.accuracy_score(y_test,y_pred)
recall = metrics.recall_score(y_test,y_pred)
precision = metrics.precision_score(y_test,y_pred)
print(accuracy,recall,precision)
```

После выполнения кода получим следующие значения в качестве выходных данных (рис. 7.22).

```
0.89 0.71875 0.92
```

Рис. 7.22

Наивный байесовский алгоритм

Основанный на теории вероятностей *наивный байесовский алгоритм* — один из простейших алгоритмов классификации. При правильном использовании он способен давать точные предсказания. Наивный байесовский алгоритм назван так по двум причинам:

- он базируется на наивном предположении, что существует независимость между признаками и входной переменной;
- он основан на теореме Байеса.

Этот алгоритм пытается классифицировать экземпляры на основе вероятностей предыдущих атрибутов/экземпляров, предполагая полную независимость атрибутов.

Существуют три типа событий.

- *Независимые* события не влияют на вероятность возникновения другого события (например, получение электронного письма с предложением бесплатного участия в техническом мероприятии *и* реорганизация компании).
- *Зависимые* события влияют на вероятность наступления других событий, то есть каким-то образом связаны (например, на вероятность того, что вы попадете на конференцию вовремя, может повлиять забастовка персонала авиакомпании или задержка рейса).
- *Взаимоисключающие* события не могут происходить одновременно (например, вероятность выпадения тройки и шестерки при одном броске кости равна 0 — эти два результата являются взаимоисключающими).

Теорема Байеса

Теорема Байеса используется для вычисления условной вероятности между двумя независимыми событиями, *A* и *B*. Вероятность того, что события *A* и *B*

произойдут, представлена $P(A)$ и $P(B)$. Условная вероятность представлена $P(B|A)$ и означает, что событие B произойдет, если событие A уже произошло:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}.$$

Вычисление вероятностей

Наивный байесовский алгоритм основан на фундаментальных принципах теории вероятностей. Вероятность возникновения одного события (наблюдаемая вероятность) рассчитывается так: количество раз, когда событие произошло, делится на общее количество процессов, которые могли привести к этому событию. Например, колл-центр получает в исследуемый день более 100 звонков в службу поддержки, а весь предыдущий месяц получал 50 звонков в день. Необходимо подсчитать вероятность того, что на вызов ответят менее чем за 3 минуты, основываясь на предыдущих случаях, когда вызов был принят. Если колл-центру удастся уложиться в это время в 27 случаях, то наблюдаемая вероятность того, что на 100 звонков ответят менее чем за 3 минуты, будет следующей:

$$P(\text{100 звонков до 3 мин}) = (27/50) = 0.54\ (54\ \%).$$

На 100 звонков можно ответить менее чем за 3 минуты примерно в половине случаев, основываясь на записях о 50 случаях, когда это происходило в прошлом.

Правила умножения для независимых (AND) событий

Чтобы рассчитать вероятность одновременного возникновения двух или более событий, подумайте, являются ли события зависимыми. Если они независимы, используется простое правило умножения:

*P(событие 1 AND событие 2) = P(событие 1) * P(событие 2).*

Например, для расчета вероятности получения электронного письма с бесплатным входом на техническое мероприятие *и* одновременной реорганизации, происходящей на вашем рабочем месте, будет использоваться это простое правило умножения. Эти два события независимы, так как возникновение одного не влияет на вероятность возникновения другого.

Если вероятность получения электронного письма о техническом событии составляет 31 %, а вероятность реорганизации персонала — 82 %, то вероятность того и другого рассчитывается следующим образом:

$$P(email\ AND\ реорганизация) = P(email) * P(реорганизация) = \\ = (0.31) * (0.82) = 0.2542\ (25\ \%).$$

Общее правило умножения

Если два или более событий зависят друг от друга, используется общее правило умножения. В действительности эта формула верна как в случае независимых, так и в случае зависимых событий:

$$P(outcome\ 1\ AND\ outcome\ 2) = P(outcome\ 1) * P(outcome\ 2 \mid outcome\ 1).$$

Обратите внимание, что $P(событие\ 2 \mid событие\ 1)$ относится к условной вероятности наступления *события 2*, если *событие 1* уже наступило. Формула включает в себя зависимость между событиями. Если события независимы, то условная вероятность не имеет значения, поскольку один исход не влияет на вероятность наступления другого, и $P(событие\ 2 \mid событие\ 1)$ — это просто $P(событие\ 2)$. В таком случае формула становится простым правилом умножения.

Правила сложения для взаимоисключающих (OR) событий

При расчете вероятности наступления одного из двух взаимоисключающих событий используется следующее простое правило сложения:

$$P(событие\ 1\ OR\ событие\ 2) = P(событие\ 1) + P(событие\ 2).$$

Например, какова вероятность выпадения на игральной кости 6 или 3? Чтобы ответить на этот вопрос, обратите внимание, что эти события не могут произойти одновременно. Вероятность выпадения 6 равна (1/6), и то же самое можно сказать о выпадении 3:

$$P(6\ OR\ 3) = (1/6) + (1/6) = 0.33\ (33\ \%).$$

Если события не являются взаимоисключающими и могут происходить одновременно, используйте общую формулу сложения, представленную ниже. Она всегда действительна и для взаимоисключающих, и для взаимонеисключающих событий:

$$P(событие\ 1\ OR\ событие\ 2) = P(событие\ 1) + \\ + P(событие\ 2)\ P(событие\ 1\ AND\ событие\ 2).$$

Использование наивного байесовского алгоритма для задачи классификации

Теперь давайте используем наивный байесовский алгоритм для решения задачи классификации:

1. Для начала импортируем функцию `GaussianNB()` и используем ее для обучения модели:

```
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(X_train, y_train)
```

2. Теперь используем обученную модель для предсказания результатов. Спрогнозируем метки для контрольного набора `X_test`:

```
Предсказание результатов контрольного набора
y_pred = classifier.predict(X_test)
cm = metrics.confusion_matrix(y_test, y_pred)
cm
```

3. Выведем матрицу ошибок (рис. 7.23).

```
Out[10]: array([[66,  2],
                [ 6, 26]])
```

Рис. 7.23

4. Теперь выведем метрики производительности, чтобы оценить качество обученной модели:

```
accuracy= metrics.accuracy_score(y_test,y_pred)
recall = metrics.recall_score(y_test,y_pred)
precision = metrics.precision_score(y_test,y_pred)
print(accuracy,recall,precision)
```

Мы получаем результат в виде рис. 7.24.

```
0.92 0.8125 0.9285714285714286
```

Рис. 7.24

Среди алгоритмов классификации победителем становится...

Сравним производительность алгоритмов, которые мы протестировали. Метрики приведены в табл. 7.4.

Таблица 7.4

Алгоритм	Доля правильных ответов	Полнота	Точность
Дерево решений	0.94	0.93	0.88
XGBoost	0.93	0.90	0.87
Случайный лес	0.93	0.90	0.87
Логистическая регрессия	0.91	0.81	0.89
SVM	0.89	0.71	0.92
Наивный байесовский	0.92	0.81	0.92

Можно заметить, что классификатор дерева решений работает лучше всего с точки зрения доли правильных ответов и полноты. Если мы стремимся к точности, то между SVM и наивным байесовским классификатором ничья, так что любой из них нам подойдет.

АЛГОРИТМЫ РЕГРЕССИИ

Модель машинного обучения с учителем использует один из алгоритмов регрессии, если целевая переменная является непрерывной. В этом случае модель МО называется регрессором.

В данном разделе рассмотрены различные алгоритмы, которые можно использовать для обучения регрессионных моделей МО с учителем — или просто регрессоров. Прежде чем мы перейдем к деталям алгоритмов, давайте сформулируем задачу, которая позволит проверить их производительность, возможности и эффективность.

Задача регрессии

Аналогично подходу, который мы использовали с алгоритмами классификации, сначала сформулируем задачу, общую для всех алгоритмов регрессии. Назовем ее задачей регрессии. Для ее решения мы применим три алгоритма регрессии. Использование общей задачи в качестве проверки различных алгоритмов регрессии имеет два преимущества.

- Мы можем подготовить данные один раз и использовать подготовленные данные во всех трех алгоритмах регрессии.

- Мы можем осуществить сравнение производительности трех алгоритмов регрессии.

Перейдем к постановке задачи.

Постановка задачи регрессии

В наши дни важно уметь прогнозировать пробег различных транспортных средств (ТС). Эффективное транспортное средство не только не вредит окружающей среде, но и выгодно экономически. Пробег можно оценить исходя из мощности двигателя и характеристик ТС. Итак, наша задача состоит в том, чтобы обучить модель, способную на основе характеристик ТС предсказывать количество пройденных им *миль на галлон (MPG)*.

Рассмотрим исторический набор данных, который мы будем использовать для обучения регрессоров.

Изучение набора исторических данных

В табл. 7.5 приведены признаки из имеющегося набора исторических данных.

Таблица 7.5

Название	Тип	Описание
`NAME` *(имя)*	Категориальный	Идентифицирует конкретное транспортное средство
`CYLINDERS` *(цилиндры)*	Непрерывный	Количество цилиндров (от 4 до 8)
`DISPLACEMENT` *(литраж)*	Непрерывный	Объем двигателя в кубических дюймах
`HORSEPOWER` *(лошадиные силы)*	Непрерывный	Мощность двигателя в лошадиных силах
`ACCELERATION` *(ускорение)*	Непрерывный	Время, необходимое для ускорения от 0 до 60 миль в час (в секундах)

Целевой переменной для этой задачи является непрерывная переменная `MPG`, которая обозначает количество миль на галлон для каждого из ТС.

Спроектируем пайплайн обработки данных для нашей задачи.

Конструирование признаков с использованием пайплайна обработки данных

Приступим к проектированию многоразового пайплайна обработки данных для решения задачи регрессии. Как уже упоминалось, мы подготовим данные один раз, после чего используем их во всех алгоритмах. Выполним следующие шаги:

1. Импортируем набор данных:

```
dataset = pd.read_csv('auto.csv')
```

2. Предварительно рассмотрим набор данных:

```
dataset.head(5)
```

Вот как будет выглядеть набор данных (рис. 7.25).

	NAME	CYLINDERS	DISPLACEMENT	HORSEPOWER	WEIGHT	ACCELERATION	MPG
0	chevrolet chevelle malibu	8	307.0	130	3504	12.0	18.0
1	buick skylark 320	8	350.0	165	3693	11.5	15.0
2	plymouth satellite	8	318.0	150	3436	11.0	18.0
3	amc rebel sst	8	304.0	150	3433	12.0	16.0
4	ford torino	8	302.0	140	3449	10.5	17.0

Рис. 7.25

3. Далее перейдем к выбору признаков. Опустим столбец `NAME`, так как это идентификатор, необходимый только для автомобилей. Столбцы, которые используются для идентификации строк в нашем наборе данных, не имеют отношения к обучению модели. Отбросим эту колонку:

```
dataset=dataset.drop(columns=['NAME'])
```

4. Теперь преобразуем все входные переменные и заменим все пропущенные значения на нули:

```
dataset=dataset.drop(columns=['NAME'])
dataset= dataset.apply(pd.to_numeric, errors='coerce')
dataset.fillna(0, inplace=True)
```

Эта операция улучшает качество данных и подготавливает их к использованию для обучения модели. Перейдем к заключительному шагу:

5. Разделим данные на части для контроля и обучения:

```
from sklearn.model_selection import train_test_split
#from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.25, random_state = 0)
```

Это приведет к созданию следующих четырех структур:

- `X_train`: структура данных, содержащая признаки обучающих данных;
- `X_test`: структура данных, содержащая признаки контрольных данных;
- `y_train`: вектор, содержащий значения метки в наборе обучающих данных;
- `y_test`: вектор, содержащий значения метки в наборе контрольных данных.

Теперь испытаем подготовленные данные на трех различных регрессорах, чтобы оценить их производительность.

Линейная регрессия

Из всех методов машинного обучения с учителем алгоритм линейной регрессии является самым доступным для понимания. Сначала мы рассмотрим простую линейную регрессию, а затем расширим концепцию до множественной линейной регрессии.

Простая линейная регрессия

В своей простейшей форме линейная регрессия формулирует взаимосвязь между непрерывной зависимой переменной и непрерывной независимой переменной. Простая линейная регрессия демонстрирует, в какой степени изменения зависимой переменной (показанной на оси y) соотносятся с изменениями независимой переменной (показанной на оси x). Ее можно представить следующим образом:

$$y' = (X)w + \alpha.$$

Пояснения к формуле:

- y — зависимая переменная;
- X — независимая переменная;
- w — наклон (угловой коэффициент), который указывает, насколько прямая поднимается или опускается при каждом изменении X;
- α — свободный коэффициент, который указывает значение y при $X = 0$.

Ниже приведены некоторые примеры взаимосвязей между непрерывной зависимой переменной и непрерывной независимой переменной:

- вес человека и потребление им калорий;
- цена дома и его площадь в квадратных футах в конкретном районе;
- влажность воздуха и вероятность дождя.

Для линейной регрессии как входная (независимая) переменная, так и целевая (зависимая) переменная должны быть числовыми. Наилучшая взаимосвязь достигается путем минимизации суммы квадратов вертикальных расстояний от каждой точки до прямой, проведенной через все точки. Предполагается, что связь между переменной-предиктором и целевой переменной является линейной. Например, чем больше денег вложено в исследования и разработку продукта, тем выше его продажи.

Давайте рассмотрим конкретный пример. Попробуем сформулировать взаимосвязь между маркетинговыми расходами и продажами конкретного продукта. Оказывается, они напрямую связаны друг с другом. Маркетинговые расходы и продажи отражены на двумерном графике и показаны в виде голубых ромбиков. Взаимосвязь лучше всего можно аппроксимировать, проведя прямую линию, как показано ниже (рис. 7.26).

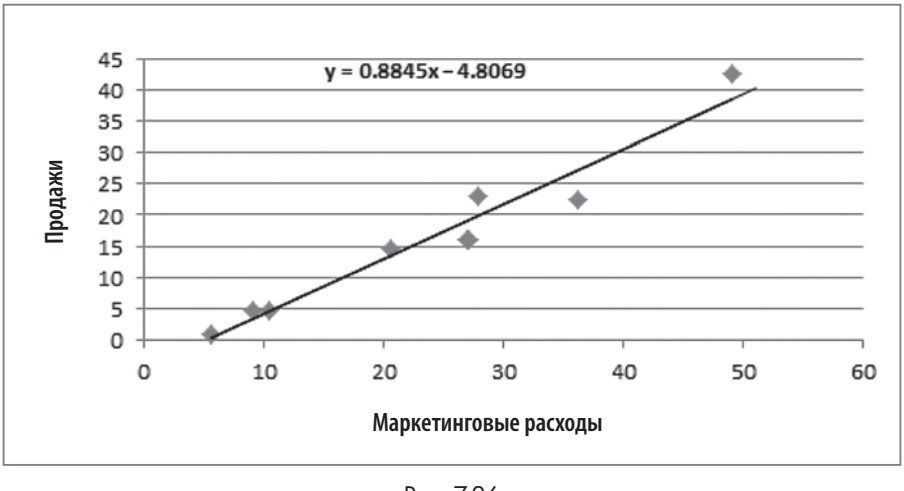


Рис. 7.26

Как только линия проведена, мы можем увидеть математическую зависимость между маркетинговыми расходами и продажами.

Оценка регрессоров

Линейная регрессия, которую мы построили, является аппроксимацией взаимосвязи между зависимой и независимой переменными. Даже самая лучшая линия будет иметь некоторое отклонение от фактических значений, как показано на рис. 7.27.

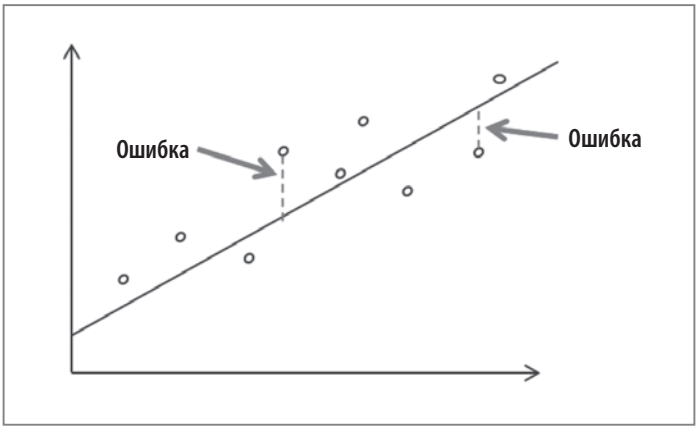


Рис. 7.27

Чтобы количественно оценить эффективность модели линейной регрессии, обычно используется *среднеквадратичная ошибка* (Root Mean Square Error, RMSE). Это математическое вычисление стандартного отклонения ошибок, допущенных обученной моделью. Для примера в наборе обучающих данных функция `loss` рассчитывается следующим образом:

$$\text{Loss}\,(y'^{(i)}, y^{(i)}) = 1/2(y'^{(i)} - y^{(i)})^2.$$

Это приводит к функции `cost`, которая сводит к минимуму потери для всех примеров в обучающем наборе:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y'^{(i)} - y^{i})^2}.$$

Давайте попробуем интерпретировать RMSE. Допустим, для нашей модели, которая предсказывает цену продукта, RMSE составляет 50 долларов. Это означает, что около 68,2 % прогнозов будут находиться в пределах 50 долларов от истинного значения (то есть α). Это также означает, что 95 % прогнозов будут находиться в пределах 100 долларов (то есть 2α) от фактического значения.

Наконец, 99,7 % прогнозов окажутся в пределах 150 долларов от фактического значения.

Множественная регрессия

В реальности при анализе чаще всего рассматривается более одной независимой переменной. *Множественная регрессия* является расширением простой линейной регрессии. Ключевое отличие заключается в том, что вводятся дополнительные бета-коэффициенты (β) для дополнительных переменных-предикторов. При обучении модели цель состоит в том, чтобы найти коэффициенты β, которые минимизируют ошибки линейного уравнения. Попробуем математически сформулировать взаимосвязь между зависимой переменной и набором независимых переменных (признаков).

Аналогично простому линейному уравнению, зависимая переменная y равна сумме свободного члена α и произведения коэффициентов β на значение x для каждого из i признаков:

$$y = \alpha + \beta_1 x_1 + \beta_2 \times 2 + ... + \beta_i x_i + \varepsilon.$$

Ошибка обозначается как ε и указывает на то, что прогнозы неидеальны.

Коэффициенты β позволяют каждому признаку отдельно влиять на значение y, поскольку y изменяется на величину β_i для каждого единичного увеличения x_i. Более того, свободный коэффициент α указывает ожидаемое значение y, когда все независимые переменные равны 0.

Обратите внимание, что все переменные в предыдущем уравнении могут быть представлены набором векторов. В этом случае целевые и предсказанные переменные становятся векторами со строкой. Коэффициенты регрессии β и ошибки ε также являются векторами.

Использование алгоритма линейной регрессии в задаче регрессии

Теперь давайте обучим модель, используя обучающую часть набора данных:

1. Начнем с импорта пакета линейной регрессии:

```
from sklearn.linear_model import LinearRegression
```

2. Создадим экземпляр модели линейной регрессии и обучим ее с помощью обучающего набора данных:

```
regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

3. Спрогнозируем результаты, используя контрольную часть набора данных:

```
y_pred = regressor.predict(X_test)
from sklearn.metrics import mean_squared_error
from math import sqrt
sqrt(mean_squared_error(y_test, y_pred))
```

4. Выходные данные, сгенерированные при выполнении этого кода, будут выглядеть так (рис. 7.28).

```
Out[10]: 4.36214129677179
```

Рис. 7.28

Как обсуждалось в предыдущем разделе, RMSE — это стандартное отклонение для ошибок. Оно указывает на то, что 68,2 % прогнозов будут находиться в пределах `4.36` от значения целевой переменной.

Когда используется линейная регрессия?

Линейная регрессия используется для решения многих реальных задач, например:

- Прогнозирование продаж.
- Прогнозирование оптимальных цен на продукцию.
- Количественная оценка причинно-следственной связи между событием и реакцией, например в клинических испытаниях лекарств, технических испытаниях безопасности или маркетинговых исследованиях.
- Выявление закономерностей, которые могут быть использованы для прогнозирования будущего поведения с учетом известных критериев. Например, для прогнозирования страховых выплат, ущерба от стихийных бедствий, результатов выборов или уровня преступности.

Слабые стороны линейной регрессии

Слабые стороны линейной регрессии заключаются в следующем:

- Она работает только с числовыми функциями.
- Категориальные данные должны быть предварительно обработаны.

- Она плохо справляется с пропущенными значениями.
- Она делает предположения относительно данных.

Алгоритм дерева регрессии

Алгоритм дерева регрессии (regression tree algorithm) аналогичен алгоритму дерева классификации, за исключением того, что целевая переменная является непрерывной переменной, а не категориальной.

Использование алгоритма дерева регрессии в задаче регрессии

В этом разделе мы увидим, как алгоритм дерева регрессии может быть использован в решении задачи регрессии.

1. Сначала обучим модель, используя алгоритм дерева регрессии (рис. 7.29).

```
In [43]: from sklearn.tree import DecisionTreeRegressor
         regressor = DecisionTreeRegressor(max_depth=3)
         regressor.fit(X_train, y_train)

Out[43]: DecisionTreeRegressor(criterion='mse', max_depth=4, max_features=None,
                               max_leaf_nodes=None, min_impurity_decrease=0.0,
                               min_impurity_split=None, min_samples_leaf=1,
                               min_samples_split=2, min_weight_fraction_leaf=0.0,
                               presort=False, random_state=None, splitter='best')
```

Рис. 7.29

2. Как только модель дерева регрессии будет обучена, используем ее для предсказания значений:

```
y_pred = regressor.predict(X_test)
```

3. Вычислим RMSE для количественной оценки производительности модели:

```
from sklearn.metrics import mean_squared_error
from math import sqrt
sqrt(mean_squared_error(y_test, y_pred))
```

Получим следующий результат (рис. 7.30).

```
Out[45]: 5.2771702288377
```

Рис. 7.30

Алгоритм градиентного бустинга для регрессии

Теперь попробуем применить для регрессии *алгоритм градиентного бустинга*. Он использует ансамбль деревьев решений, чтобы наилучшим образом выявить основные закономерности в данных.

Использование градиентного бустинга в задаче регрессии

В этом разделе мы выясним, как применить алгоритм градиентного бустинга в задаче регрессии.

1. Сначала обучим модель, используя градиентный бустинг (рис. 7.31).

```
In [5]: from sklearn import ensemble
        params = {'n_estimators': 500, 'max_depth': 4, 'min_samples_split': 2,
                  'learning_rate': 0.01, 'loss': 'ls'}
        regressor = ensemble.GradientBoostingRegressor(**params)

        regressor.fit(X_train, y_train)

Out[5]: GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
                                  learning_rate=0.01, loss='ls', max_depth=4,
                                  max_features=None, max_leaf_nodes=None,
                                  min_impurity_decrease=0.0, min_impurity_split=None,
                                  min_samples_leaf=1, min_samples_split=2,
                                  min_weight_fraction_leaf=0.0, n_estimators=500,
                                  n_iter_no_change=None, presort='auto',
                                  random_state=None, subsample=1.0, tol=0.0001,
                                  validation_fraction=0.1, verbose=0, warm_start=False)
```

Рис. 7.31

2. Как только модель обучена, используем ее для предсказания значений:

```
y_pred = regressor.predict(X_test)
```

3. Рассчитаем RMSE для количественной оценки производительности модели:

```
from sklearn.metrics import mean_squared_error
from math import sqrt
sqrt(mean_squared_error(y_test, y_pred))
```

4. Запустив код, получаем на выходе следующее значение (рис. 7.32).

```
Out[7]: 4.034836373089085
```

Рис. 7.32

Среди алгоритмов регрессии победителем становится...

Теперь сравним производительность трех алгоритмов регрессии, использованных для решения одной и той же задачи при одинаковых данных (табл. 7.6).

Таблица 7.6

Алгоритм	**RMSE**
Линейная регрессия	4.36214129677179
Дерево регрессии	5.2771702288377
Градиентный бустинг для регрессии	4.034836373089085

Из таблицы очевидно, что производительность градиентного бустинга является наилучшей, так как имеет самый низкий RMSE. За ним следует линейная регрессия. Алгоритм дерева регрессии показал худшие результаты для данной задачи.

ПРАКТИЧЕСКИЙ ПРИМЕР — КАК ПРЕДСКАЗАТЬ ПОГОДУ

Попробуем применить изученные концепции для прогноза погоды. Предположим, что мы хотим предсказать, будет ли завтра дождь, основываясь на данных, собранных за год для конкретного города.

Данные для обучения этой модели находятся в файле CSV под названием `weather.csv`.

1. Прежде всего импортируем данные в виде DataFrame pandas:

```
import numpy as np
import pandas as pd
df = pd.read_csv("weather.csv")
```

2. Рассмотрим столбцы DataFrame (рис. 7.33).
3. Далее изучим заголовки первых 13 столбцов данных `weather.csv` (рис. 7.34).
4. Рассмотрим оставшиеся 10 столбцов данных `weather.csv` (рис. 7.35).
5. Используем `x` для обозначения входных признаков. Удалим поле `Date` из списка признаков, так как оно бесполезно в контексте прогнозов. Также откажемся от метки `RainTomorrow`:

```
x = df.drop(['Date','RainTomorrow'],axis=1)
```

```
In [63]: df.columns

Out[63]: Index(['Date', 'MinTemp', 'MaxTemp', 'Rainfall', 'Evaporation', 'Sunshine',
                'WindGustDir', 'WindGustSpeed', 'WindDir9am', 'WindDir3pm',
                'WindSpeed9am', 'WindSpeed3pm', 'Humidity9am', 'Humidity3pm',
                'Pressure9am', 'Pressure3pm', 'Cloud9am', 'Cloud3pm', 'Temp9am',
                'Temp3pm', 'RainToday', 'RISK_MM', 'RainTomorrow'],
               dtype='object')
```

Рис. 7.33

```
In [124]: df.iloc[:,0:12].head()
```

Out[124]:

	Date	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindDir9am	WindDir3pm	WindSpeed9am	WindSpeed3pm
0	2007-11-01	8.0	24.3	0.0	3.4	6.3	7	30.0	12	7	6.0	20
1	2007-11-02	14.0	26.9	3.6	4.4	9.7	1	39.0	0	13	4.0	17
2	2007-11-03	13.7	23.4	3.6	5.8	3.3	7	85.0	3	5	6.0	6
3	2007-11-04	13.3	15.5	39.8	7.2	9.1	7	54.0	14	13	30.0	24
4	2007-11-05	7.6	16.1	2.8	5.6	10.6	10	50.0	10	2	20.0	28

Рис. 7.34

```
In [127]: df.iloc[:,12:25].head()
```

Out[127]:

	Humidity9am	Humidity3pm	Pressure9am	Pressure3pm	Cloud9am	Cloud3pm	Temp9am	Temp3pm	RainToday	RISK_MM	RainTomorrow
0	68	29	1019.7	1015.0	7	7	14.4	23.6	0	3.6	1
1	80	36	1012.4	1008.4	5	3	17.5	25.7	1	3.6	1
2	82	69	1009.5	1007.2	8	7	15.4	20.2	1	39.8	1
3	62	56	1005.5	1007.0	2	7	13.5	14.1	1	2.8	1
4	68	49	1018.3	1018.5	7	7	11.1	15.4	1	0.0	0

Рис. 7.35

6. Используем `y` для обозначения метки:

```
y = df['RainTomorrow']
```

7. Разделим данные на обучающие и контрольные с помощью `train_test_split`:

```
from sklearn.model_selection import train_test_split
train_x , train_y ,test_x , test_y = train_test_split(x,y ,
test_size = 0.2,random_state = 2)
```

8. Поскольку метка является двоичной переменной, мы обучаем классификатор. Для нашей задачи лучше всего подходит логистическая регрессия. Создадим экземпляр модели логистической регрессии:

```
model = LogisticRegression()
```

9. Применим `train_x` и `test_x` для обучения модели:

```
model.fit(train_x , test_x)
```

10. Как только модель обучена, ее можно использовать для прогнозирования:

```
predict = model.predict(train_y)
```

11. Определим долю правильных ответов для обученной модели (рис. 7.36).

```
In [89]: predict = model.predict(train_y)

In [90]: from sklearn.metrics import accuracy_score

In [91]: accuracy_score(predict , test_y)

Out[91]: 0.9696969696969697
```

Рис. 7.36

Теперь бинарный классификатор можно использовать для прогноза, будет ли завтра дождь.

РЕЗЮМЕ

Эту главу мы начали с изучения основ машинного обучения с учителем. Затем более подробно рассмотрели различные алгоритмы классификации, методы оценки производительности классификаторов и познакомились с регрессионными алгоритмами. Кроме того, научились оценивать производительность изученных алгоритмов.

В следующей главе мы обсудим нейронные сети и алгоритмы глубокого обучения. Мы рассмотрим методы, используемые для обучения нейронной сети, а также различные инструменты и платформы, доступные для ее оценки и развертывания.

8

Алгоритмы нейронных сетей

На сегодняшний день *искусственные нейронные сети, ИНС* (или ANN — Artificial Neural Networks) являются одним из наиболее важных методов машинного обучения. Этому способствуют необходимость решения все более сложных задач, лавинообразный рост объемов данных и появление таких технологий, как легкодоступные дешевые кластеры. Такие кластеры способны обеспечить вычислительную мощность, необходимую для разработки крайне сложных алгоритмов.

ИНС — стремительно развивающаяся область исследований, отвечающая за важнейшие достижения в таких отраслях, как робототехника, обработка естественного языка и создание беспилотных автомобилей.

Основной структурной единицей ИНС является нейрон. ИНС позволяет использовать потенциал большого количества нейронов, организуя их в многослойную архитектуру. Сигнал проходит через слои и различными способами обрабатывается в каждом из них до тех пор, пока не будет сгенерирован требуемый результат. Скрытые слои в ИНС выступают в качестве уровней абстракции, что позволяет осуществлять *глубокое обучение*. Этот метод широко используется при реализации мощных приложений, таких как Amazon Alexa, Google Photos и поиск по изображениям в Google.

В этой главе мы познакомимся с основными понятиями ИНС, рассмотрим компоненты и типы нейронных сетей, а также различные виды функций активации. Затем подробно обсудим алгоритм обратного распространения ошибки, который чаще всего используется для обучения нейронной сети. Далее мы познакомимся с *переносом обучения*, с помощью которого можно значительно упростить и частично автоматизировать обучение моделей. Наконец, разберем

практический пример использования глубокого обучения для выявления фальшивых документов.

Ниже приведены основные концепции, обсуждаемые в этой главе:

- Введение в ИНС.
- Эволюция ИНС.
- Обучение нейронной сети.
- Инструменты и фреймворки.
- Перенос обучения.
- Практический пример — использование глубокого обучения для выявления мошенничества.

Начнем с изучения основ ИНС.

ВВЕДЕНИЕ В ИНС

В 1957 году, вдохновившись работой нейронов в человеческом мозге, Фрэнк Розенблатт предложил концепцию нейронных сетей. Чтобы в полной мере понять ИНС, будет полезным получить представление о том, как связаны нейроны в человеческом мозге. Рассмотрим слоистую структуру нейронов, представленную на следующей схеме (рис. 8.1).

В человеческом мозге *дендриты* действуют как датчики, которые улавливают сигнал. Затем сигнал передается на *аксон*, представляющий собой длинный тонкий отросток нервной клетки. Работа аксона заключается в передаче сигнала мышцам, железам и другим нейронам. Прежде чем перейти к следующему нейрону, сигнал проходит через соединение, называемое *синапсом*. Обратите внимание, что по этому органическому конвейеру сигнал движется до тех пор, пока не достигнет нужной мышцы или железы, где он вызовет необходимую реакцию. Обычно требуется от семи до восьми миллисекунд, чтобы сигнал прошел через цепочку нейронов и достиг места назначения.

Вдохновившись этим шедевром обработки сигналов, созданным самой природой, Фрэнк Розенблатт спроектировал метод, позволяющий обрабатывать цифровую информацию послойно для решения сложных математических задач. Результатом его первой попытки создания нейронной сети стала довольно простая сеть, напоминающая модель линейной регрессии. Она не имела никаких скрытых слоев и была названа *перцептроном* (рис. 8.2).

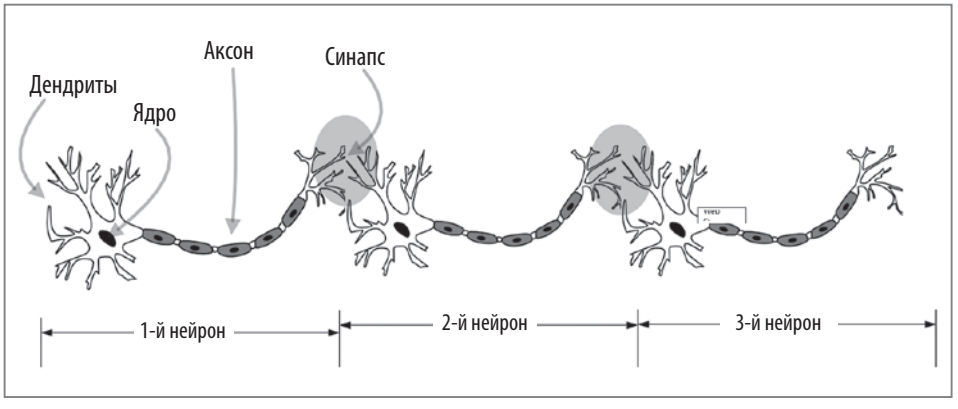


Рис. 8.1

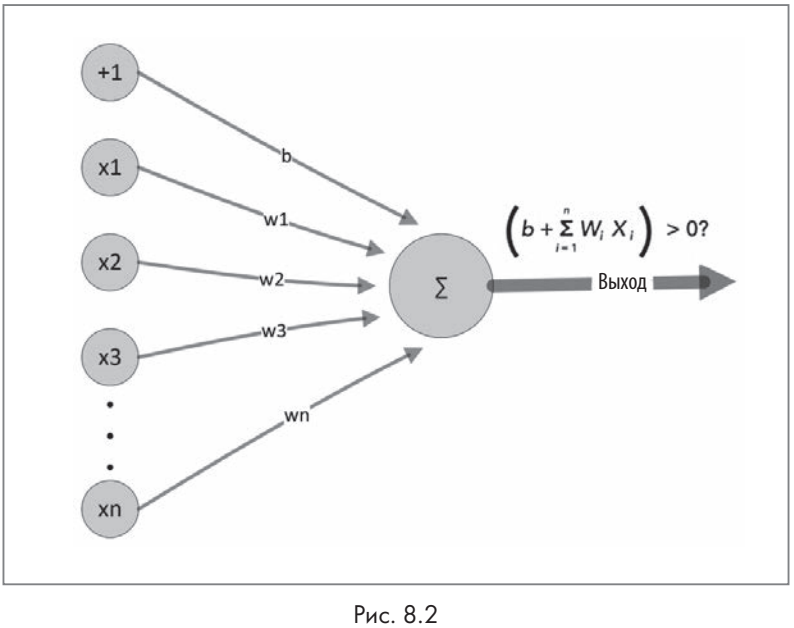


Рис. 8.2

Сформулируем математическое описание перцептрона. В левой части предыдущей схемы представлены входные сигналы. Каждый сигнал ($x_1, x_2 .. x_n$) умножается на присвоенный ему вес ($w_1, w_2 ... w_n$), а затем результаты суммируются, и получается *взвешенная сумма*:

$$\left(b+\sum_{i=1}^{n} w_i x_i\right) > 0?$$

Обратите внимание, что это бинарный классификатор, поскольку конечный выход перцептрона является истинным или ложным в зависимости от выхода *агрегатора* (на схеме показан как Σ). Агрегатор выдаст истинный сигнал, если обнаружит валидный сигнал хотя бы на одном из входов.

Давайте узнаем, как нейронные сети эволюционировали с течением времени.

ЭВОЛЮЦИЯ ИНС

В предыдущем разделе мы узнали о создании перцептрона, простой нейронной сети без каких-либо слоев. Позже было обнаружено, что перцептрон имеет серьезные ограничения. В 1969 году Марвин Мински и Сеймур Пейперт провели исследования и пришли к выводу, что перцептрон не способен обучаться какой-либо сложной логике.

Более того, эти исследования показали, что будет затруднительно обучить его даже такой простой логической функции, как XOR (исключающее «или»). Это вызвало снижение интереса к МО в целом и нейронным сетям в частности и положило начало эпохе, получившей название *зимы искусственного интеллекта*. Исследователи по всему миру не воспринимали ИИ всерьез, считая, что он неспособен решать хоть сколько-нибудь сложные задачи.

Одной из основных причин так называемой зимы ИИ стала ограниченность доступных в то время аппаратных возможностей. Либо не было необходимых вычислительных мощностей, либо они были непомерно дорогими. К концу 1990-х годов достижения в области распределенных вычислений обеспечили доступность инфраструктуры, что привело к окончанию зимы искусственного интеллекта. Эта оттепель активизировала исследования в области ИИ.

В конечном итоге наступила эпоха, которую можно назвать *весной искусственного интеллекта*. Ее характеризует большой интерес к ИИ в целом и нейронным сетям в частности.

Для более сложных задач исследователи разработали многослойную нейронную сеть — *многослойный перцептрон*. Эта сеть имеет несколько слоев, как показано на рис. 8.3:

- входной слой;
- скрытый слой (слои);
- выходной слой.

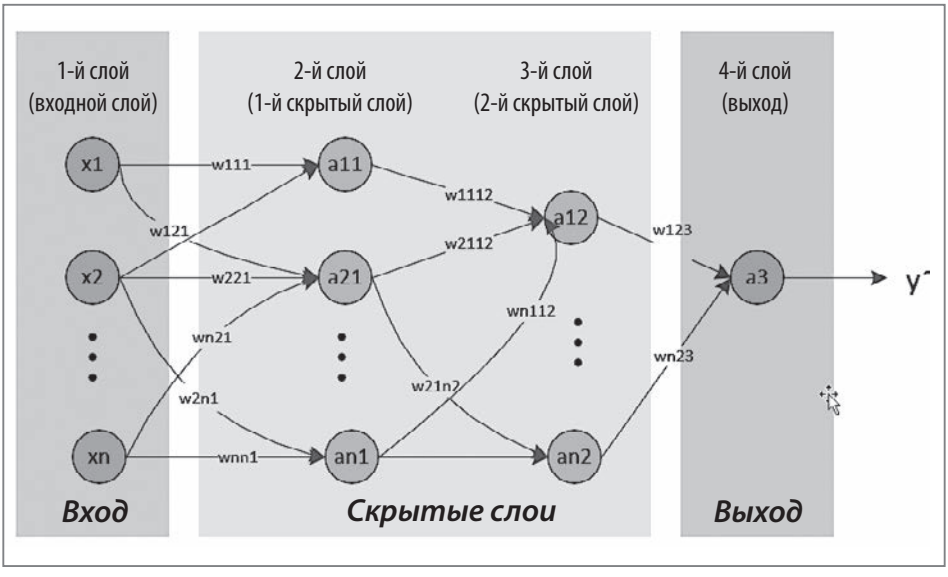


Рис. 8.3

Нейрон является основной единицей сети, и каждый нейрон любого слоя связан со всеми нейронами следующего слоя. Для сложных сетей количество таких соединений резко возрастает. Мы рассмотрим различные способы сокращения этих взаимосвязей без большой потери качества.

Сформулируем задачу.

Входными данными является вектор признаков x размерности n.

Нейронная сеть должна предсказывать значения. Прогнозируемые значения представлены в виде y'.

Наша задача — на основе некоторых входных данных определить вероятность того, что транзакция является мошеннической. Другими словами: при определенном значении x какой будет вероятность того, что $y = 1$? Математически это можно представить так:

$$y' = P(y = 1 \mid x), \text{ где } x \in \Re^{n_x}.$$

Обратите внимание, что x — это n_x-мерный вектор, где n_x — количество входных переменных.

Эта нейронная сеть состоит из четырех слоев. Слои между входом и выходом называются *скрытыми*. Количество нейронов в первом скрытом слое обозначается $n_h^{[1]}$. Связи между различными узлами умножаются на параметры, называемые *весами*. Обучение нейронной сети — это поиск правильных значений для весов.

Давайте узнаем, как обучить нейронную сеть.

ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ

Процесс построения нейронной сети с использованием заданного набора данных называется *обучением нейронной сети*. Рассмотрим анатомию типичной нейронной сети. Когда мы говорим об обучении нейронной сети, мы говорим о вычислении наилучших значений для весов. Обучение проводится итеративно с использованием набора примеров в виде обучающих данных. Примеры в обучающих данных содержат ожидаемые значения выходных данных для различных комбинаций входных значений. Процесс обучения нейронных сетей отличается от способа обучения традиционных моделей (которые обсуждались в главе 7).

Анатомия нейронной сети

Давайте разберем, из чего состоит нейронная сеть.

- *Слои*. Основные строительные блоки нейронной сети. Каждый слой представляет собой модуль обработки данных, который действует как фильтр. Он принимает один или несколько входных сигналов, обрабатывает их определенным образом, а затем выдает один или несколько выходных сигналов. Каждый раз, когда данные проходят через слой, они проходят этап обработки и демонстрируют закономерности, имеющие отношение к бизнес-вопросу, на который мы пытаемся ответить.
- *Функция потерь*. Обеспечивает сигнал обратной связи, используемый на различных итерациях процесса обучения. Функция потерь высчитывает отклонение для одного примера.
- *Функция стоимости*. Это функция потерь в полном наборе примеров.
- *Оптимизатор*. Определяет, как будет интерпретироваться сигнал обратной связи, предоставляемый функцией потерь.

- *Входные данные*. Данные, которые используются для обучения нейронной сети. Они определяют целевую переменную.
- *Веса*. Рассчитываются путем обучения сети. Веса приблизительно соответствуют важности каждого из входных сигналов. Например, если конкретный входной сигнал более важен, чем другие, после обучения ему присваивается большее значение веса, действующее как множитель. Даже слабый сигнал при этом будет усилен благодаря большому значению веса. Таким образом, вес изменяет сигналы в соответствии с их важностью.
- *Функция активации*. Значения умножаются на различные веса, а затем агрегируются. То, как именно они будут агрегированы и как будет интерпретироваться их значение, определяется типом выбранной функции активации.

Рассмотрим, как происходит процесс обучения нейронных сетей.

При обучении нейронных сетей мы перебираем примеры один за другим. Для каждого примера с помощью еще не обученной модели генерируются выходные данные. Далее вычисляется разница между ожидаемым и предсказанным результатами. Для каждого отдельного примера эта разница называется *потерей*. В совокупности потери по всему обучающему набору данных называются *стоимостью*. Продолжая тренировать модель, мы стремимся найти те значения весов, которые приведут к наименьшим потерям. На протяжении всего обучения значения весов продолжают корректироваться, пока не обнаружится набор значений, приводящий к минимально возможной общей стоимости. Как только мы достигнем минимальной стоимости, модель считается обученной.

Градиентный спуск

Цель обучения модели нейронной сети — найти верные значения весов. Обучение стартует со случайными или стандартными значениями весов. Затем итеративно используется алгоритм оптимизатора, например *градиентный спуск*, чтобы изменить веса в целях улучшения прогнозов.

Отправной точкой алгоритма градиентного спуска являются случайные значения весов, которые нужно оптимизировать по мере выполнения алгоритма. В каждой из последующих итераций алгоритм продолжает работу, изменяя значения весов таким образом, чтобы стоимость была минимизирована.

Следующая схема объясняет логику алгоритма градиентного спуска (рис. 8.4).

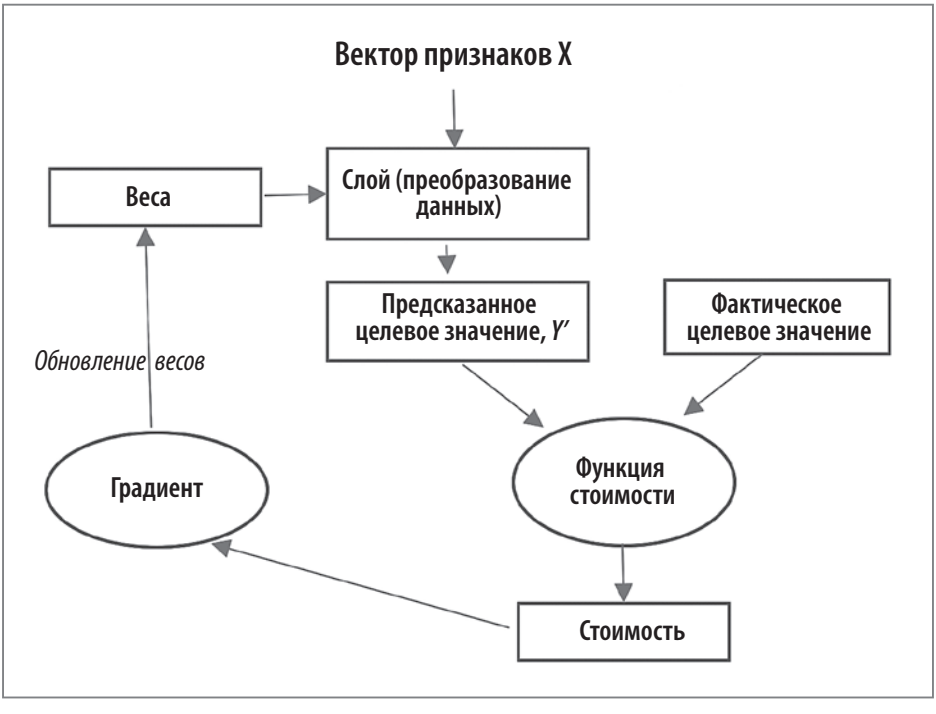


Рис. 8.4

На схеме входными данными является вектор признаков X. Фактическое значение целевой переменной равно Y, а ее предсказанное — Y'. Мы определяем отклонение фактического значения от предсказанных значений, обновляем веса и повторяем шаги до тех пор, пока стоимость не будет сведена к минимуму.

Изменение веса на каждой итерации алгоритма зависит от двух факторов.

- *Направление*. В каком направлении необходимо двигаться, чтобы получить минимум в функции потерь.
- *Скорость обучения*. Насколько значительными должны быть изменения в выбранном нами направлении.

Простой итерационный процесс показан на следующей схеме (рис. 8.5).

На схеме показано, как градиентный спуск пытается найти минимальное значение функции потерь, изменяя веса. Скорость обучения и выбранное направление определяют следующую точку на графике, которую нужно проверить.

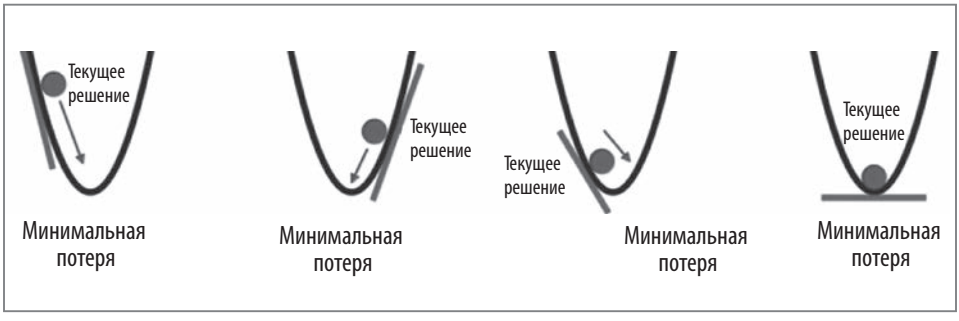


Рис. 8.5

Важно выбрать правильное значение для параметра скорости обучения. Если значение слишком низкое, то решение задачи займет много времени; если оно слишком высокое — задача не сойдется. На предыдущей схеме точка, представляющая наше текущее решение, будет продолжать колебаться между двумя противолежащими частями кривой на графике.

Теперь минимизируем градиент. Возьмем только две переменные, x и y. Градиент x и y рассчитывается следующим образом:

$$\text{градиент} = \frac{\Delta y}{\Delta x}.$$

Чтобы минимизировать градиент, можно использовать следующий подход:

```
while(gradient!=0):
    if (gradient < 0); move right
    if (gradient > 0); move left
```

Этот алгоритм также может быть использован для поиска оптимальных или близких к оптимальным значений весов для нейронной сети.

Обратите внимание, что расчет градиентного спуска выполняется в обратном направлении по всей сети. Сначала мы вычисляем градиент последнего слоя, затем предпоследнего, затем всех предыдущих, пока не достигнем первого. Это *метод обратного распространения ошибки* (backpropagation), предложенный Хинтоном, Уильямсом и Румельхартом в 1985 году.

Далее рассмотрим функции активации.

Функции активации

Функция активации формулирует, каким образом входные данные для конкретного нейрона будут обработаны для получения выходных данных.

Как показано на следующей схеме, каждый нейрон в нейронной сети имеет функцию активации, которая определяет то, как будут обрабатываться входные данные (рис. 8.6).

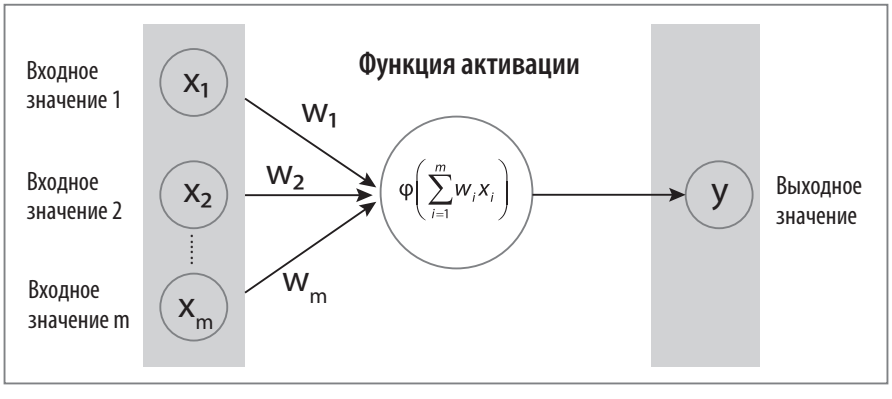


Рис. 8.6

На схеме мы видим, что результаты, полученные в процессе работы функции активации, передаются на выход. Функция задает критерии интерпретации входных значений для генерации выходных данных.

Для одних и тех же входных значений разные функции активации будут выдавать разные результаты. При решении задач с использованием нейронных сетей важно правильно выбирать функцию активации.

Рассмотрим несколько функций активации более подробно.

Пороговая функция

Самая простая из возможных функций активации — *пороговая функция* (threshold function). Выходные данные пороговой функции являются двоичными: 0 или 1. Она будет генерировать 1 в качестве выходного сигнала, если какое-либо входное значение больше 1. Это проиллюстрировано на следующей схеме (рис. 8.7).

Обратите внимание, что как только во взвешенных суммах входных данных обнаруживается какой-либо сигнал, выход (y) становится 1. Это делает функцию

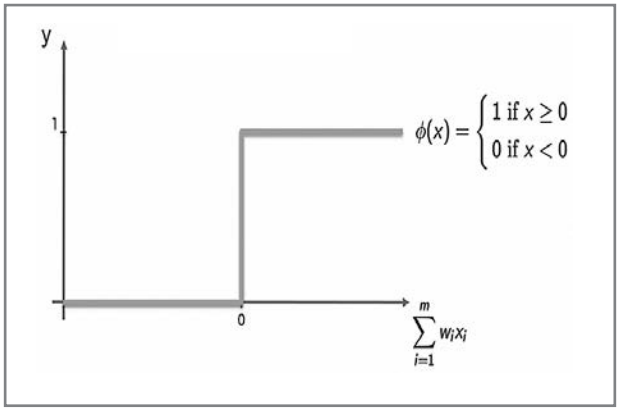


Рис. 8.7

пороговой активации очень чувствительной. Она склонна к ошибочному срабатыванию при малейшем сигнале на входе из-за сбоя или какого-либо шума.

Сигмоидная функция

Сигмоидную функцию (sigmoid function) можно рассматривать как улучшенный вариант пороговой. При работе с этой функцией активации мы получаем контроль над ее чувствительностью (рис. 8.8).

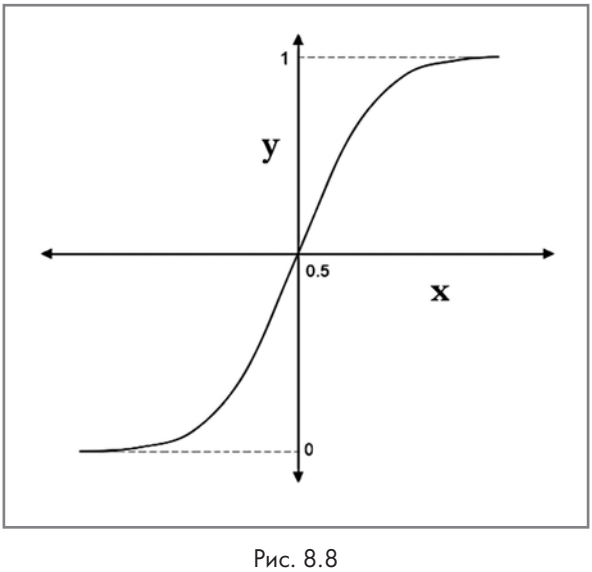


Рис. 8.8

Сигмоидная функция y определяется так:

$$y = f(x) = \frac{1}{1+e^{-x}}.$$

Ее можно реализовать на Python следующим образом:

```
def sigmoidFunction(z):
    return 1/ (1+np.exp(-z))
```

Снижая чувствительность функции активации, мы делаем сбои на входе менее разрушительными. Обратите внимание, что вывод сигмоидной функции активации по-прежнему двоичный, то есть 0 или 1.

ReLU

Первые две функции активации, представленные в этой главе, преобразуют набор входных переменных в двоичные выходные данные. *ReLU* (rectified linear unit; блок линейной ректификации) — это функция активации, которая принимает набор входных переменных и преобразует их в один непрерывный вывод. В нейронных сетях ReLU обычно используется в скрытых слоях, чтобы не преобразовывать непрерывные переменные в категориальные.

На следующей схеме представлен график функции активации ReLU (рис. 8.9).

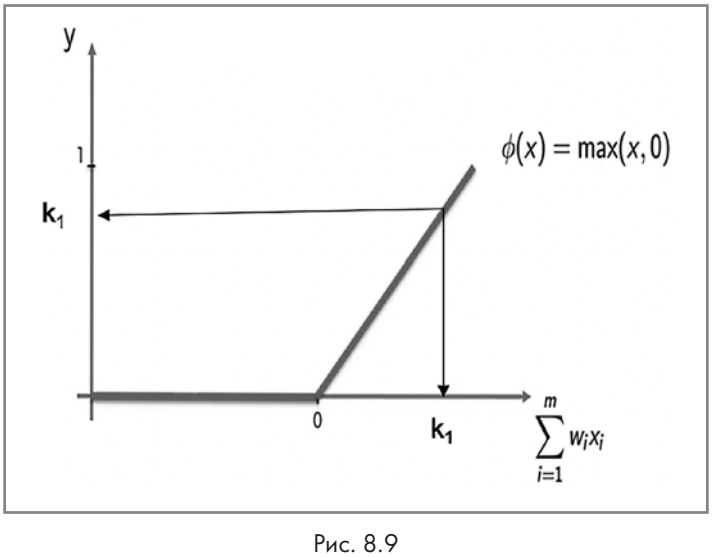


Рис. 8.9

Обратите внимание, что $x \leq 0$ означает, что $y = 0$. Таким образом, любой сигнал со входа, который равен или меньше нуля, преобразуется в нулевой выход:

$$y = f(x) = 0 \text{ для } x < 0$$
$$y = f(x) = x \text{ для } x \geq 0.$$

Как только x становится больше нуля, функция равна x.

Функция ReLU является одной из наиболее часто используемых функций активации в нейронных сетях. Ее можно реализовать на Python следующим образом:

```
def ReLU(x):
if x<0:
    return 0
else:
    return x
```

Теперь рассмотрим функцию Leaky ReLU, основанную на ReLU.

Leaky ReLU

В ReLU отрицательное значение для x приводит к нулевому значению для y. Это означает, что в процессе теряется часть информации, что удлиняет циклы обучения, особенно в начале обучения. Функция активации *Leaky ReLU* (ReLU с утечкой) избавляется от этой проблемы. Для Leaky ReLU применимо следующее:

$$y = f(x) = \beta x \text{ для } x < 0$$
$$y = f(x) = x \quad \text{для } x \geq 0.$$

График функции показан на следующей схеме (рис. 8.10).

Здесь β — параметр со значением меньше единицы.

Функцию можно реализовать на Python следующим образом:

```
def leakyReLU(x,beta=0.01):
    if x<0:
        return (beta*x)
    else:
        return x
```

Существуют три способа указать значение β:

- Указать значение по умолчанию для β.

- Сделать β параметром в нашей нейронной сети и позволить нейронной сети определять значение (*параметрическая ReLU*).
- Сделать β случайным значением (*рандомизированная ReLU*).

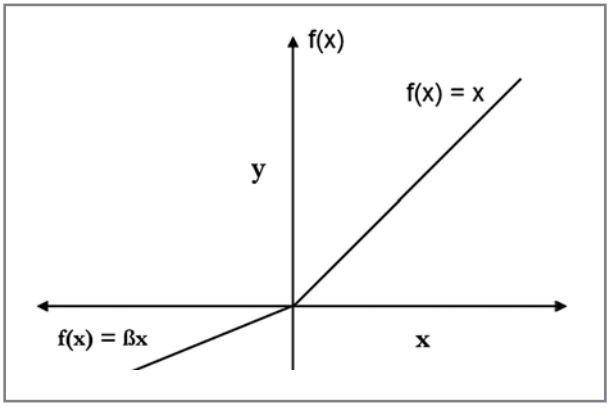


Рис. 8.10

Гиперболический тангенс (tanh)

Функция *tanh* аналогична сигмоидной функции, но она способна выдавать отрицательный сигнал. Это показано на следующей схеме (рис. 8.11).

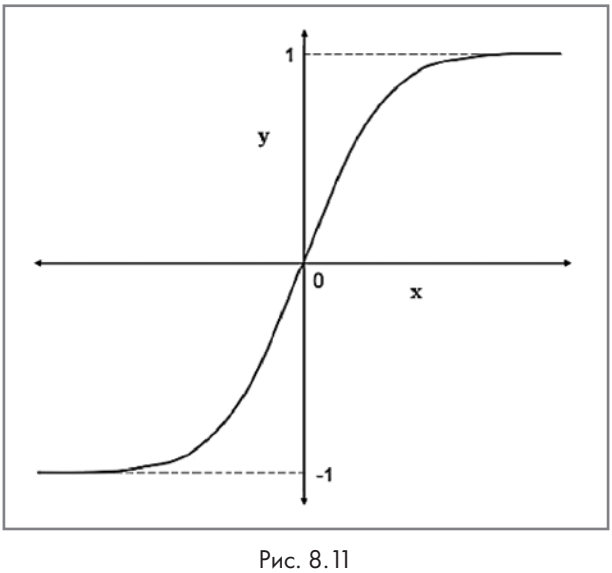


Рис. 8.11

Функция y выглядит следующим образом:

$$y = f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}.$$

Ее можно реализовать с помощью следующего кода Python:

```
def tanh(x):
    numerator = 1-np.exp(-2*x)
    denominator = 1+np.exp(-2*x)
    return numerator/denominator
```

Перейдем теперь к функции softmax.

Softmax

Иногда нам требуется более двух уровней для вывода функции активации. В таких случаях используется функция *softmax*. Она лучше всего подходит для задач многоклассовой классификации. Предположим, что у нас есть n классов и входные значения. Входные значения отображают классы следующим образом:

$$x = \{x^{(1)}, x^{(2)}, ... x^{(n)}\}.$$

Softmax основана на теории вероятности. Выходная вероятность e-го класса softmax рассчитывается следующим образом:

$$prob^{(s)} = \frac{e^{x^s}}{\sum_{i=1}^{n} e^{x^i}}.$$

Для бинарных классификаторов функция активации в конечном слое будет сигмоидной, а для многоклассовых классификаторов — softmax.

ИНСТРУМЕНТЫ И ФРЕЙМВОРКИ

В этом разделе мы подробно рассмотрим фреймворки и инструменты, доступные для реализации нейронных сетей.

К настоящему времени разработано множество различных фреймворков для реализации нейронных сетей. Каждый из них имеет свои сильные и слабые стороны. В этом разделе мы сосредоточимся на Keras с серверной частью TensorFlow.

Keras

Keras — одна из самых популярных и простых в использовании библиотек нейронных сетей, написанная на Python. Она была создана с прицелом на простоту использования и предлагает самый быстрый способ реализации глубокого обучения. Keras предоставляет только высокоуровневые операции и применяется на уровне модели.

Выбор серверного движка для Keras

Keras требуется библиотека глубокого обучения более низкого уровня для выполнения манипуляций с тензорами. Эта библиотека глубокого обучения более низкого уровня называется *серверным движком* или *бэкенд-движком* (backend engine). Внутренние движки, доступные для Keras:

- *TensorFlow* (www.tensorflow.org). Это самый популярный фреймворк в своем классе, с открытым исходным кодом, созданный Google;
- *Theano* (https://github.com/Theano/Theano). Разработан в лаборатории MILA в Университете Монреаля;
- *Microsoft Cognitive Toolkit (CNTK)*. Разработан корпорацией Microsoft.

Формат этого модульного стека технологий глубокого обучения показан на следующей схеме (рис. 8.12).

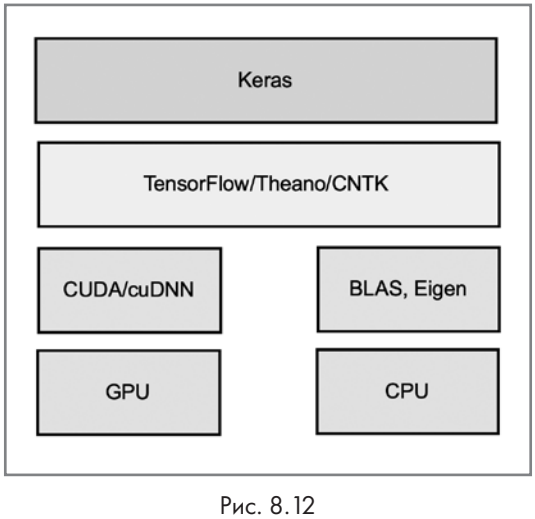


Рис. 8.12

Преимущество модульной архитектуры глубокого обучения заключается в том, что бэкенд-движок Keras можно изменять без переписывания кода. Например, если для конкретной задачи TensorFlow подходит лучше, чем Theano, можно просто сменить серверную часть на TensorFlow.

Низкоуровневые слои стека глубокого обучения

Все три упомянутых серверных движка способны работать как на центральном процессоре (CPU), так и на графическом (GPU), используя низкоуровневые слои стека. Для CPU существует библиотека тензорных операций, называемая *Eigen*. Для GPU TensorFlow использует библиотеку *CUDA Deep Neural Network* (cuDNN) от NVIDIA.

Определение гиперпараметров

Гиперпараметр — это параметр, значение которого выбирается перед запуском процесса обучения. Сначала мы задаем значения на основе здравого смысла, а затем пытаемся их оптимизировать. Для нейронных сетей важными являются следующие гиперпараметры:

- функция активации;
- скорость обучения;
- количество скрытых слоев;
- количество нейронов в каждом скрытом слое.

Давайте выясним, как строится модель с помощью Keras.

Построение модели Keras

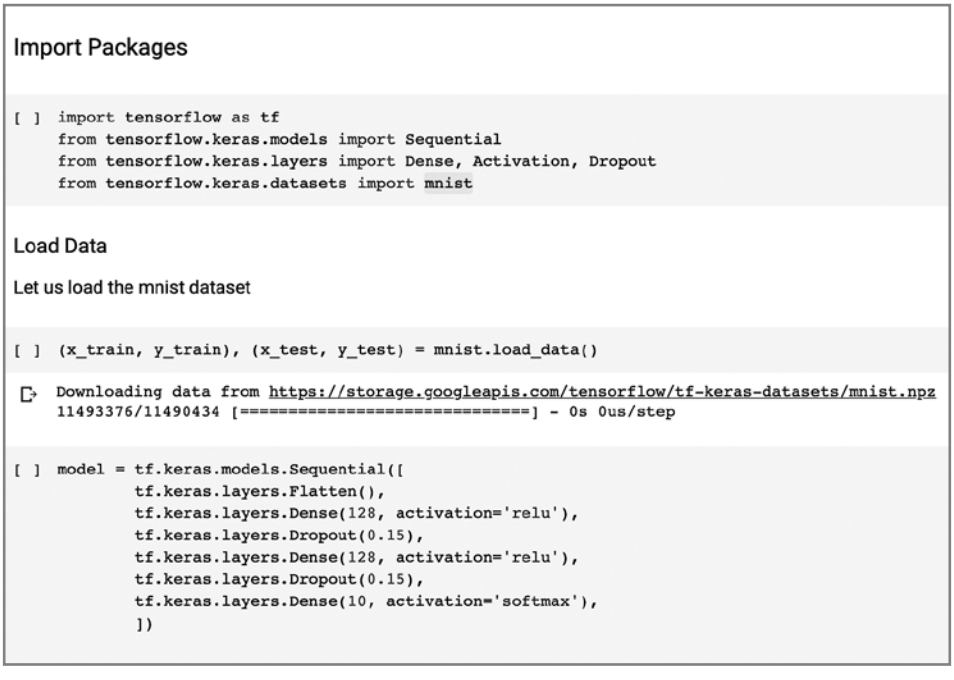
Создание завершенной модели Keras состоит из трех этапов:

1. *Конструирование слоев.*

   Существуют два способа построения модели с использованием Keras:

   - *Последовательный API* (Sequential API)[1]. Позволяет создавать модели для линейного стека слоев. Это типичный выбор при построении относительно простых моделей (рис. 8.13).

[1] API, application programming interface — интерфейс программирования приложений. — *Примеч. ред.*



Import Packages

```
[ ]  import tensorflow as tf
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Dense, Activation, Dropout
     from tensorflow.keras.datasets import mnist
```

Load Data

Let us load the mnist dataset

```
[ ]  (x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
     Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
     11493376/11490434 [==============================] - 0s 0us/step
```

```
[ ]  model = tf.keras.models.Sequential([
             tf.keras.layers.Flatten(),
             tf.keras.layers.Dense(128, activation='relu'),
             tf.keras.layers.Dropout(0.15),
             tf.keras.layers.Dense(128, activation='relu'),
             tf.keras.layers.Dropout(0.15),
             tf.keras.layers.Dense(10, activation='softmax'),
             ])
```

Рис. 8.13

Обратите внимание, что мы создали три слоя: первые два имеют функцию активации ReLU, а третий — softmax.

- *Функциональный API* (Functional API). Позволяет проектировать модели для ациклических графов. С его помощью могут быть созданы более сложные модели (рис. 8.14).

```
inputs = tf.keras.Input(shape=(128,128))
x = tf.keras.layers.Flatten()(inputs)
x = tf.keras.layers.Dense(512, activation='relu',name='d1')(x)
x = tf.keras.layers.Dropout(0.2)(x)
predictions = tf.keras.layers.Dense(10,activation=tf.nn.softmax, name='d2')(x)
model = tf.keras.Model(inputs=inputs, outputs=predictions)
```

Рис. 8.14

Одну и ту же нейронную сеть можно спроектировать, используя как последовательные, так и функциональные API. Выбор того или иного подхода не влияет на производительность.

2. *Настройка процесса обучения.*

   На этом этапе мы задаем три аргумента:

   - оптимизатор;
   - функция потерь;
   - метрики, которые будут количественно определять качество модели (рис. 8.15).

```
optimiser = tf.keras.optimizers.RMSprop
model.compile (optimizer= optimiser, loss='mse', metrics = ['accuracy'])
```

Рис. 8.15

   Для определения оптимизатора, функции потерь и метрик используется функция `model.compile`.

3. *Обучение модели.*

   Как только архитектура определена, пришло время обучить модель (рис. 8.16).

```
model.fit(x_train, y_train, batch_size=128, epochs=10)
```

Рис. 8.16

   Обратите внимание, что такие параметры, как `batch_size` (размер пакета) и `epochs` (количество эпох), являются настраиваемыми, что делает их гиперпараметрами.

Выбор последовательной или функциональной модели

Последовательная модель создает ИНС в виде простого стека слоев. Такую модель несложно понять и реализовать, но ее упрощенная архитектура накладывает серьезные ограничения. Каждый слой подключен ровно к одному входному и выходному тензору. Если же модель имеет несколько входов или выходов на любом из слоев (входном, выходном или скрытом), то она является функциональной.

Знакомство с TensorFlow

TensorFlow — одна из самых популярных библиотек для работы с нейронными сетями. В предыдущем разделе мы узнали о ее применении в качестве сервер-

ного движка Keras. На самом деле эта высокопроизводительная библиотека с открытым исходным кодом может использоваться для любых вычислительных задач. Если посмотреть на стек, становится ясно, что мы можем написать код TensorFlow на высокоуровневом языке (таком, как Python или C++), который затем интерпретируется распределенным механизмом выполнения. Это делает TensorFlow весьма полезной и популярной среди разработчиков.

Принцип работы TensorFlow заключается в том, что для представления вычислений создается *ориентированный граф* (directed graph). Ребра этого графа представляют собой массивы данных (входные и выходные данные), соединяющие вершины (математические операции).

Основные понятия TensorFlow

Давайте кратко рассмотрим понятия TensorFlow: скаляры, векторы и матрицы. Известно, что простое число, такое как три или пять, в традиционной математике называется *скаляром*. *Вектор* — это объект, имеющий величину и направление. Применительно к TensorFlow вектор используется для обозначения одномерных массивов. Расширив эту концепцию, мы получим двумерный массив, *матрицу*. Для трехмерного массива применяется термин *3D-тензор*. Термин *ранг* обозначает размерность структуры данных. Таким образом, *скаляр* — это структура данных *ранга 0*, *вектор* — структура данных *ранга 1*, а *матрица* — структура данных *ранга 2*. Эти многомерные структуры известны как *тензоры* и показаны на следующей схеме (рис. 8.17).

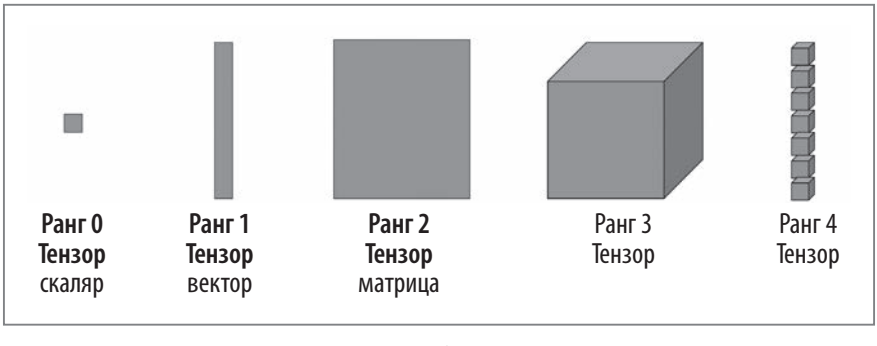


Рис. 8.17

Перейдем к другому параметру, *размеру,* называемому также *формой* (shape). Размер — это кортеж целых чисел, определяющих длину массива в каждом измерении. На следующей диаграмме объясняется концепция размера (рис. 8.18).

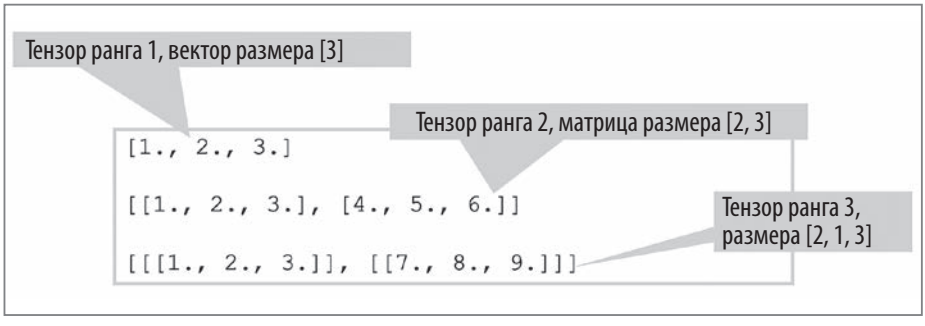


Рис. 8.18

Используя shape и rank, мы задаем параметры тензоров.

Понимание тензорной математики

Рассмотрим различные математические вычисления с использованием тензоров:

- Определим два скаляра, чтобы потом сложить и умножить их с помощью TensorFlow (рис. 8.19).

```
In [13]: print("Define constant tensors")
         a = tf.constant(2)
         print("a = %i" % a)
         b = tf.constant(3)
         print("b = %i" % b)

         Define constant tensors
         a = 2
         b = 3
```

Рис. 8.19

- Осуществим сложение и умножение, а затем выведем результаты (рис. 8.20).

```
In [14]: print("Running operations, without tf.Session")
         c = a + b
         print("a + b = %i" % c)
         d = a * b
         print("a * b = %i" % d)

         Running operations, without tf.Session
         a + b = 5
         a * b = 6
```

Рис. 8.20

- Создадим новый скаляр, сложив два тензора (рис. 8.21).

```
In [16]: c = a + b
         print("a + b = %s" % c)

         a + b = Tensor("add:0", shape=(2, 2), dtype=float32)
```

Рис. 8.21

- Выполним сложные тензорные функции (рис. 8.22).

```
In [17]: d = tf.matmul(a, b)
         print("a * b = %s" % d)

         a * b = Tensor("MatMul:0", shape=(2, 2), dtype=float32)
```

Рис. 8.22

Типы нейронных сетей

Существует несколько способов построения нейронных сетей. Если каждый нейрон в каждом слое соединен с каждым нейроном в другом слое, то перед нами *полносвязная* нейронная сеть. Рассмотрим некоторые другие формы нейронных сетей.

Сверточные нейросети

Сверточные нейросети, СНС (или CNN — Convolutional Neural Networks), обычно используются для анализа мультимедийных данных. Чтобы узнать больше о том, как СНС используется для анализа данных на основе изображений, нам необходимо иметь представление о следующих процессах:

- свертка (convolution);
- объединение или подвыборка (pooling).

Рассмотрим их по очереди.

Свертка

Процесс *свертки* выделяет нужный шаблон в конкретном изображении путем обработки его с применением другого изображения меньшего размера, называ-

емого *фильтром* или *ядром*. Например, если мы хотим найти границы объектов на изображении, мы можем свернуть его с помощью определенного фильтра. Выделение границ применяется для обнаружения объектов, их классификации и других задач. Иными словами, процесс свертки заключается в поиске характеристик и особенностей изображения.

Подход основан на поиске шаблонов, которые можно использовать повторно применительно к другим данным. Такие шаблоны называются фильтрами или ядрами.

Подвыборка (объединение)

Важной частью обработки мультимедийных данных для машинного обучения является *понижающая дискретизация* (downsampling). Она дает два преимущества:

- Уменьшается размерность задачи, существенно сокращается время, необходимое для обучения модели.
- Посредством агрегирования ненужные детали в мультимедийных данных абстрагируются. В результате данные становятся более обобщенными и более репрезентативными для аналогичных задач.

Понижающая дискретизация выполняется следующим образом (рис. 8.23).

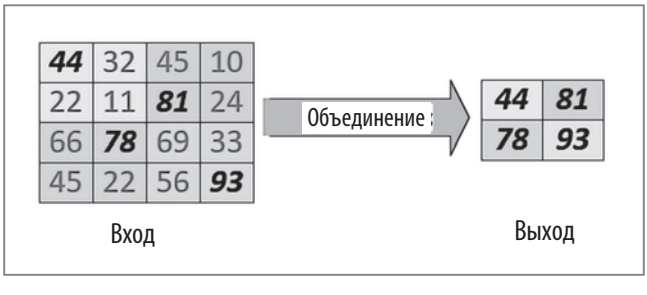


Рис. 8.23

Мы заменили каждый блок из четырех пикселей на один пиксель, выбрав для него наибольшее значение из четырех. Это означает, что выборка сократилась в четыре раза. Поскольку мы взяли максимальное значение в каждом блоке, этот процесс называется *объединением (подвыборкой) по максимальному значению*. Мы могли бы взять среднее значение; в данном случае это было бы *объединением (подвыборкой) средних значений*.

Рекуррентные нейросети

Рекуррентные нейросети, РНС (или RNN — Recurrent Neural Networks), — это особый тип нейронных сетей, основанных на циклической архитектуре. Именно поэтому они и называются *рекуррентными*. Важно отметить, что РНС обладают памятью. Это означает, что у них есть возможность хранить информацию из последних итераций. Они используются в таких задачах, как анализ структуры предложений (когда надо предсказать следующее слово в предложении).

Генеративно-состязательные сети

Генеративно-состязательные сети, ГСС (или GAN — Generative Adversarial Networks), — это тип нейронных сетей, которые генерируют синтетические данные. Эти сети были созданы в 2014 году Иэном Гудфеллоу и его коллегами. С помощью ГСС можно генерировать фотографии людей, которых никогда не существовало. Что еще более важно, такие сети могут создавать синтетические данные для аугментации (расширения) обучающих датасетов.

В следующем разделе мы рассмотрим, что такое перенос обучения.

ПЕРЕНОС ОБУЧЕНИЯ

В течение долгого времени многие организации, исследовательские группы и отдельные специалисты в сообществе открытого исходного кода совершенствовали сложные модели, обученные на гигантских объемах данных. В некоторых случаях на оптимизацию моделей были затрачены годы усилий. Некоторые из этих моделей с открытым исходным кодом можно использовать для следующих задач:

- Обнаружение объектов на видео.
- Обнаружение объектов на изображениях.
- Расшифровка аудио.
- Анализ эмоциональной окраски текста.

Всякий раз перед обучением новой модели МО следует задаться вопросом: можем ли мы просто адаптировать под наши цели готовую, предварительно обученную и испытанную модель (вместо того, чтобы начинать с нуля)? Другими словами, можно ли перенести обучение существующих моделей на новую модель так, чтобы ответить на бизнес-вопрос? Если это возможно, мы получаем три преимущества:

- Ускоряется процесс обучения модели.
- При использовании проверенной и зарекомендовавшей себя модели общее качество нашей модели, скорее всего, улучшится.
- *Перенос обучения* (transfer learning) решит проблему недостаточного объема обучающих данных.

Рассмотрим два практических примера.

- Допустим, нам нужно обучить робота. Для тренировки модели нейронной сети мы используем симуляцию. В ней воссоздаются все те редкие события, с которыми почти невозможно столкнуться в реальности. Затем мы применяем перенос обучения, чтобы обучить модель для реального мира.
- Предположим, нам нужна модель, которая выявляет на видеотрансляции ноутбуки Apple и Windows. Уже существуют готовые надежные модели с открытым исходным кодом для обнаружения объектов; эти модели умеют точно классифицировать различные предметы в видеопотоке. Можно использовать такие модели в качестве отправной точки и идентифицировать объекты как ноутбуки. После этого мы дополнительно обучим модель различать ноутбуки Apple и Windows между собой.

В следующем разделе мы применим рассмотренные концепции для построения нейронной сети, классифицирующей фальшивые документы.

ПРАКТИЧЕСКИЙ ПРИМЕР — ИСПОЛЬЗОВАНИЕ ГЛУБОКОГО ОБУЧЕНИЯ ДЛЯ ВЫЯВЛЕНИЯ МОШЕННИЧЕСТВА

Использование методов машинного обучения для выявления фальшивых документов является актуальной и сложной областью исследований. Исследователи пытаются понять, в какой степени для подобных целей можно использовать способность нейронных сетей распознавать шаблоны. Вместо ручного извлечения атрибутов для некоторых архитектурных структур глубокого обучения можно использовать необработанные пиксели.

Методология

В методике, представленной в этом разделе, используется тип архитектуры нейронной сети, называемый *сиамской нейронной сетью* (Siamese neural network).

Эта сеть состоит из двух ветвей с одинаковой архитектурой и параметрами. Использование сиамских нейронных сетей для маркировки фальшивых документов демонстрируется на следующей схеме (рис. 8.24).

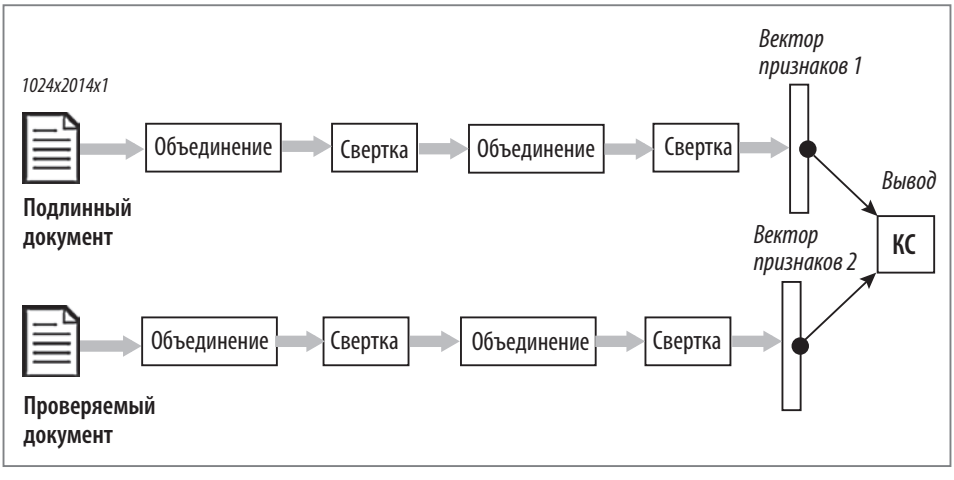


Рис. 8.24

Чтобы проверить документ на подлинность, нужно прежде всего классифицировать его по макету и типу, а затем сравнить с образцом. Если отклонение превышает определенный порог, документ помечается как поддельный; в противном случае он считается подлинным или достоверным. В особо важных ситуациях может понадобиться сравнить документы вручную. Это касается пограничных случаев, когда алгоритм не может окончательно классифицировать документ как подлинный или поддельный.

Чтобы сравнить документ с образцом, используем две идентичные *сверточные нейросети* в нашей сиамской архитектуре. СНС имеют преимущество в обучении оптимальных детекторов локальных признаков, инвариантных к сдвигу. Они могут создавать представления, устойчивые к геометрическим искажениям входного изображения. Это хорошо подходит для нашей задачи, поскольку мы будем проводить подлинные и проверяемые документы через одну и ту же сеть, а затем сравнивать результаты на предмет сходства. Для достижения этой цели мы осуществим следующие шаги.

Давайте предположим, что мы хотим проверить документ. Для каждого класса документов мы выполняем следующие действия.

1. Получить сохраненное изображение подлинного документа. Мы назовем его *истинным документом*. Проверяемый документ должен выглядеть как истинный.
2. Истинный документ передается через слои нейронной сети для создания вектора признаков, который является математическим представлением шаблонов истинного документа. Назовем его *вектором признаков 1* (см. схему выше).
3. Документ, который необходимо протестировать, называется *проверяемым документом*. Мы передаем его через нейронную сеть, идентичную той, которая использовалась для создания вектора признаков истинного документа. Вектор признаков проверяемого документа назовем *вектором признаков 2*.
4. Используем евклидово расстояние между вектором признаков 1 и вектором признаков 2 для вычисления показателя сходства между истинным и проверяемым документами. Показатель сходства называется *коэффициентом сходства, КС* (или MOS — measure of similarity). КС — это число в диапазоне от 0 до 1. Большее число означает меньшее расстояние между документами и большую вероятность того, что документы похожи.
5. Если показатель сходства, рассчитанный нейронной сетью, ниже заранее определенного порога, мы помечаем документ как поддельный.

Реализуем сиамскую нейронную сеть с помощью Python.

1. Импортируем необходимые библиотеки Python:

```
import random
import numpy as np
import tensorflow as tf
```

2. Определим нейронную сеть, которая будет использоваться для обработки каждой из ветвей сиамской сети:

```
def createTemplate():
    return tf.keras.models.Sequential([
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dropout(0.15),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dropout(0.15),
        tf.keras.layers.Dense(64, activation='relu'),
        ])
```

Обратите внимание, что мы указали коэффициент `dropout = 0.15` (dropout — исключение некоторого количества нейронов в процессе обучения для избежания переобучения).

3. Для реализации сиамской сети мы будем использовать изображения из базы данных MNIST. Они идеально подходят для проверки эффективности нашего подхода. Необходимо подготовить данные: каждый образец должен иметь два изображения и флаг бинарного сходства. Этот флаг является показателем того, что изображения принадлежат к одному классу. Теперь реализуем функцию подготовки данных под названием `prepareData()`:

```
def prepareData(inputs: np.ndarray, labels: np.ndarray):
    classesNumbers = 10
    digitalIdx = [np.where(labels == i)[0] for i in
range(classesNumbers)]
    pairs = list()
    labels = list()
    n = min([len(digitalIdx[d]) for d in range(classesNumbers)])
- 1
    for d in range(classesNumbers):
      for i in range(n):
          z1, z2 = digitalIdx[d][i], digitalIdx[d][i + 1]
          pairs += [[inputs[z1], inputs[z2]]]
          inc = random.randrange(1, classesNumbers)
          dn = (d + inc) % classesNumbers
          z1, z2 = digitalIdx[d][i], digitalIdx[dn][i]
          pairs += [[inputs[z1], inputs[z2]]]
          labels += [1, 0]
      return np.array(pairs), np.array(labels, dtype=np.float32)
```

   Обратите внимание, что `prepareData()` приведет к равному количеству образцов для всех чисел.

4. Подготовим обучающие и контрольные наборы данных:

```
 (x_train, y_train), (x_test, y_test) =
tf.keras.datasets.mnist.load_data()
x_train = x_train.astype(np.float32)
x_test = x_test.astype(np.float32)
x_train /= 255
x_test /= 255
input_shape = x_train.shape[1:]
train_pairs, tr_labels = prepareData(x_train, y_train)
test_pairs, test_labels = prepareData(x_test, y_test)
```

5. Теперь создадим две половины сиамской системы:

```
input_a = tf.keras.layers.Input(shape=input_shape)
enconder1 = base_network(input_a)
input_b = tf.keras.layers.Input(shape=input_shape)
enconder2 = base_network(input_b)
```

6. Реализуем *коэффициент сходства* (`measureOfSimilarity`), который количественно определит разницу между двумя документами, которые мы сравниваем:

```
distance = tf.keras.layers.Lambda(
    lambda embeddings: tf.keras.backend.abs(embeddings[0] -
embeddings[1])) ([enconder1, enconder2])
measureOfSimilarity = tf.keras.layers.Dense(1,
activation='sigmoid') (distance)
```

Перейдем к обучению модели. Для этого мы используем 10 эпох (рис. 8.25).

```
[10] # Build the model
     model = tf.keras.models.Model([input_a, input_b], measureOfSimilarity)
     # Train
     model.compile(loss='binary_crossentropy',optimizer=tf.keras.optimizers.Adam(),metrics=['accuracy'])

     model.fit([train_pairs[:, 0], train_pairs[:, 1]], tr_labels,
               batch_size=128,epochs=10,validation_data=([test_pairs[:, 0], test_pairs[:, 1]], test_labels))

     Epoch 1/10
     847/847 [==============================] - 6s 7ms/step - loss: 0.3459 - accuracy: 0.8500 - val_loss: 0.2652 - val_accuracy: 0.9105
     Epoch 2/10
     847/847 [==============================] - 6s 7ms/step - loss: 0.1773 - accuracy: 0.9337 - val_loss: 0.1685 - val_accuracy: 0.9508
     Epoch 3/10
     847/847 [==============================] - 6s 7ms/step - loss: 0.1215 - accuracy: 0.9563 - val_loss: 0.1301 - val_accuracy: 0.9610
     Epoch 4/10
     847/847 [==============================] - 6s 7ms/step - loss: 0.0956 - accuracy: 0.9665 - val_loss: 0.1087 - val_accuracy: 0.9685
     Epoch 5/10
     847/847 [==============================] - 6s 7ms/step - loss: 0.0790 - accuracy: 0.9724 - val_loss: 0.1104 - val_accuracy: 0.9669
     Epoch 6/10
     847/847 [==============================] - 6s 7ms/step - loss: 0.0649 - accuracy: 0.9770 - val_loss: 0.0949 - val_accuracy: 0.9715
     Epoch 7/10
     847/847 [==============================] - 6s 7ms/step - loss: 0.0568 - accuracy: 0.9803 - val_loss: 0.0895 - val_accuracy: 0.9722
     Epoch 8/10
     847/847 [==============================] - 6s 7ms/step - loss: 0.0513 - accuracy: 0.9823 - val_loss: 0.0807 - val_accuracy: 0.9770
     Epoch 9/10
     847/847 [==============================] - 6s 7ms/step - loss: 0.0439 - accuracy: 0.9847 - val_loss: 0.0916 - val_accuracy: 0.9737
     Epoch 10/10
     847/847 [==============================] - 6s 7ms/step - loss: 0.0417 - accuracy: 0.9853 - val_loss: 0.0835 - val_accuracy: 0.9749
     <tensorflow.python.keras.callbacks.History at 0x7ff1218297b8>
```

Рис. 8.25

Обратите внимание, что при использовании 10 эпох мы достигли точности 97,49 %. Увеличение их числа еще больше повысит уровень точности.

РЕЗЮМЕ

В этой главе мы познакомились с нейронными сетями: изучили историю их развития, типы и компоненты. Мы подробно разобрали алгоритм градиентного спуска, который используется для обучения нейронных сетей, обсудили различные функции активации и их применение. Мы также рассмотрели концепцию переноса обучения. Наконец, мы разобрали практический пример использования нейронной сети для обучения модели МО, выявляющей поддельные документы.

В следующей главе мы научимся использовать подобные алгоритмы для обработки естественного языка, обсудим концепцию эмбеддинга и использование рекуррентных сетей для обработки естественного языка. Наконец, мы познакомимся с анализом эмоциональной окраски текста.

9

Алгоритмы обработки естественного языка

В этой главе представлены алгоритмы *обработки естественного языка* (natural language processing, NLP). Мы будем постепенно двигаться от теории к практике. Сначала познакомимся с базовыми понятиями NLP и с основными алгоритмами. Далее рассмотрим одну из самых популярных нейронных сетей, которая широко применяется для решения важных задач по обработке текстовых данных. Затем обсудим ограничения NLP. Наконец, научимся использовать NLP для обучения модели МО, способной предсказывать полярность отзывов о кинофильмах.

Глава включает следующие разделы:

- Знакомство с NLP.
- Мешок слов (BoW).
- Эмбеддинги слов.
- Рекуррентные нейросети в NLP.
- Использование NLP для анализа эмоциональной окраски текста.
- Практический пример — анализ эмоциональной окраски в отзывах на фильмы.

К концу этой главы вы изучите основные техники обработки естественного языка и узнаете, как решать интересные практические задачи с помощью NLP.

Начнем с основных понятий.

ЗНАКОМСТВО С *NLP*

Обработка естественного языка (*NLP*) исследует методы формализации и описания взаимодействий между компьютером и человеческими (естественными) языками. *Это обширное направление* предполагает использование алгоритмов математической лингвистики и технологий человеко-машинного взаимодействия для обработки сложных неструктурированных данных. NLP может использоваться в различных областях, например:

- *Анализ содержания текста*. Выявление тем в хранилище текстов и классификация документов в соответствии с обнаруженными темами.
- *Анализ эмоциональной окраски текста*. Классификация текста в соответствии с содержащимися в нем положительными или отрицательными настроениями.
- *Машинный перевод*. Перевод текста с одного человеческого языка на другой.
- *Синтез речи*. Преобразование текста в устную речь.
- *Субъективная интерпретация*. Разумная интерпретация вопроса и способность ответить на него, используя доступную информацию.
- *Распознавание сущностей*. Идентификация в тексте сущностей (таких, как человек, место или вещь).
- *Выявление фейковых новостей*. Определение фейковых новостей исходя из их содержания.

Начнем с рассмотрения некоторых терминов, используемых в NLP.

Терминология NLP

NLP — это комплексная область знаний. Иногда в литературе, посвященной определенной сфере, для обозначения одного и того же явления используются разные термины. Рассмотрим некоторые базовые термины, связанные с NLP. Начнем с одного из основных видов NLP — нормализации, обычно выполняемой со входными данными.

Нормализация

Нормализация выполняется на входных текстовых данных, чтобы повысить их качество для обучения модели МО. Нормализация обычно включает в себя следующие этапы обработки:

- преобразование всего текста в верхний или нижний регистр;
- удаление знаков препинания;
- удаление чисел.

Как правило, перечисленные операции обязательны, однако фактические этапы обработки зависят от имеющейся задачи и меняются от случая к случаю. Например, если числа в тексте важны для контекста задачи, то их не потребуется удалять в процессе нормализации.

Корпус

Группа входных документов, которые мы используем для решения задачи, называется *корпусом*. Корпус выступает в качестве входных данных для задачи NLP.

Токенизация

При работе с NLP первоочередная задача состоит в том, чтобы разделить текст на список *токенов*. Этот процесс называется *токенизацией*. Степень детализации полученных токенов будет варьироваться в зависимости от цели. Например, токен может состоять из следующих элементов:

- одно слово;
- сочетание слов;
- предложение;
- абзац.

Распознавание именованных сущностей

В NLP часто возникает необходимость идентифицировать определенные слова и цифры в неструктурированных данных на принадлежность к заранее заданным категориям. Это могут быть номера телефонов, почтовые индексы, имена, места или страны. Так происходит упорядочивание неструктурированных данных. Этот процесс называется *распознаванием именованных сущностей* (named entity recognition, NER).

Стоп-слова

В результате токенизации мы получаем список всех слов, которые используются в тексте. Некоторые из них являются служебными частями речи, не несущи-

ми смысловой нагрузки: они есть в каждом документе. Эти слова не вносят никакой ценной информации в текст, в котором фигурируют. Они называются *стоп-словами* и обычно удаляются на этапе обработки данных. Некоторые примеры стоп-слов: «был», «они», «этот».

Анализ эмоциональной окраски текста

Анализ эмоциональной окраски текста (sentimental analysis), иначе называемый *анализом тональности* или *анализом мнений* (opinion mining), — это процесс извлечения положительных или отрицательных эмоций из текста.

Стемминг и лемматизация

В текстовых данных большинство слов, скорее всего, будет повторяться в нескольких разных формах. Сведение каждого слова к его корню или основе называется *стеммингом*. Стемминг используется для объединения близких по значению слов, чтобы уменьшить их общее количество для анализа. По сути, он уменьшает общую обусловленность задачи.

Например, {use, used, using, uses} => use.

Наиболее распространенным алгоритмом стемминга для английского языка служит *алгоритм Портера*.

Стемминг — это грубый процесс, который может привести к потере окончаний слов. Это может стать причиной ошибочного написания. Во многих случаях каждое слово служит просто идентификатором уровня в пространстве задачи, и ошибки не имеют значения. Если требуется правильное написание слов, то вместо стемминга следует использовать *лемматизацию*.

Алгоритмы не обладают разумом. Для человеческого мозга выявление похожих слов не представляет труда. Алгоритму же для ориентира необходимы критерии группировки.

Существуют три основных способа реализации NLP разной степени сложности:

- *мешок слов* (Bag-of-words, BoW);
- традиционные классификаторы NLP;
- глубокое обучение для NLP.

Библиотека NLTK

NLTK (natural language toolkit, инструментарий естественного языка) — широко используемая библиотека обработки естественного языка в Python. Это одна из старейших и наиболее популярных библиотек для NLP.

Библиотека NLTK крайне полезна, потому что, по сути, служит основой для построения любого процесса NLP. Она предоставляет базовые инструменты, которые можно объединять по мере необходимости, вместо того чтобы создавать их с нуля. В NLTK содержится множество инструментов, и в следующем разделе мы загрузим библиотеку и разберем некоторые из них.

Теперь рассмотрим *NLP* на основе модели «мешок слов».

МЕШОК СЛОВ (BoW)

Отображение входного текста в виде мультимножества токенов называется *мешком слов* (bag-of-words, BoW). Недостатком этого подхода является то, что мы отбрасываем большую часть грамматики, что совместно с токенизацией иногда приводит к потере контекста слов.

Применяя модель BoW, прежде всего необходимо оценить важность каждого слова во всех документах, которые мы хотим проанализировать.

Существуют три способа количественной оценки значимости слов в контексте документа:

- *Двоичный.* Если слово появляется в тексте, функция имеет значение 1, если нет — 0.
- *По количеству.* Значение функции равно числу появлений слова в тексте. Если слово отсутствует, значение функции равно 0.
- *Частота термина/обратная частота документа, TF-IDF* (term frequency/inverse document frequency). Значение функции представляет собой отношение того, насколько уникальным является слово в одном документе, к тому, насколько оно уникально во всем корпусе документов. Очевидно, что для распространенных слов, таких как «в» или «т. д.» (известных как стоп-слова), *оценка* TF-IDF будет низкой. Для более уникальных слов — например, терминов, относящихся к конкретной области, — оценка будет выше.

Обратите внимание, что, используя BoW, мы отбрасываем часть информации, а именно порядок слов в тексте. Этот подход работает, но может привести к снижению точности.

Рассмотрим конкретный пример. Подготовим модель, которая сможет распознавать хорошие и плохие отзывы о некотором ресторане. Входной файл представляет собой набор отзывов, которые будут классифицированы как положительные или отрицательные.

Сначала обработаем входные данные.

Этапы обработки представлены на следующей диаграмме (рис. 9.1).

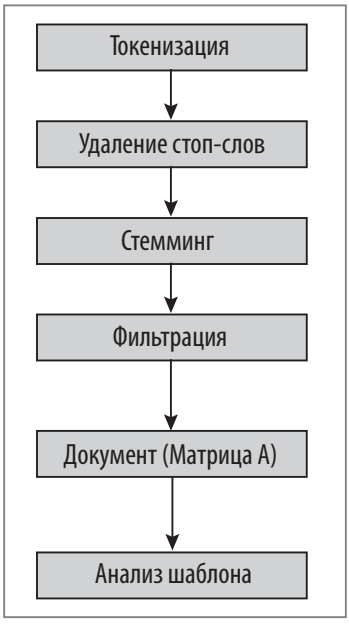


Рис. 9.1

Пайплайн обработки включает следующие шаги.

1. Прежде всего импортируем необходимые библиотеки:

```
import numpy as np
import pandas as pd
```

2. Затем импортируем набор данных из CSV-файла (рис. 9.2).

```
In [2]: # Importing the dataset
        dataset = pd.read_csv('Restaurant_Reviews.tsv', delimiter = '\t', quoting = 3)
        dataset.head()
```

Out[2]:

	Review	Liked
0	Wow... Loved this place.	1
1	Crust is not good.	0
2	Not tasty and the texture was just nasty.	0
3	Stopped by during the late May bank holiday of...	1
4	The selection on the menu was great and so wer...	1

Рис. 9.2

3. Далее очистим данные:

```
# Очистка текста
import re
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
corpus = []
for i in range(0, 1000):
    review = re.sub('[^a-zA-Z]', ' ', dataset['Review'][i])
    review = review.lower()
    review = review.split()
    ps = PorterStemmer()
    review = [ps.stem(word) for word in review if not word in
set(stopwords.words('english'))]
    review = ' '.join(review)
    corpus.append(review)
```

4. Определим признаки (представленные `y`) и метку (представленную `X`):

```
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(max_features = 1500)
X = cv.fit_transform(corpus).toarray()
y = dataset.iloc[:, 1].values
```

5. Разделим данные на обучающие и контрольные:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.20, random_state = 0)
```

6. Для обучения модели используем наивный байесовский классификатор:

```
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(X_train, y_train)
```

7. Теперь спрогнозируем результаты контрольного набора:

```
y_pred = classifier.predict(X_test)
```

8. Матрица ошибок выглядит следующим образом (рис. 9.3).

```
In [18]: # Making the Confusion Matrix
         from sklearn.metrics import confusion_matrix
         cm = confusion_matrix(y_test, y_pred)

In [19]: cm

Out[19]: array([[55, 42],
                [12, 91]])
```

Рис. 9.3

Исходя из матрицы ошибок, мы можем оценить ошибочность классификации.

ЭМБЕДДИНГИ СЛОВ

В предыдущем разделе мы узнали, как выполняется NLP с помощью мешка слов в качестве абстракции для входных текстовых данных. Одним из главных достижений в NLP является возможность создания значимого числового представления слов в виде плотных векторов. Эта техника называется *эмбеддингами слов* (word embedding). Йошуа Бенжио впервые ввел этот термин в своей статье «A Neural Probabilistic Language Model» («Нейронно-вероятностная языковая модель»). Любое слово в задаче NLP можно рассматривать как категориальный объект. Сопоставление каждого слова со списком чисел, представленных в виде вектора, называется эмбеддингом слов.

Иначе говоря, это методология преобразования слов в действительные числа. Особенность эмбеддинга состоит в том, что он использует плотный вектор, в отличие от традиционных подходов, применяющих разреженные векторы.

Существует две проблемы, связанные с использованием BoW для NLP:

- *Потеря семантического контекста*. При токенизации данных их контекст теряется. Слово может иметь разные значения в зависимости от того, где именно оно используется в предложении. Это становится еще более важным при интерпретации сложных особенностей человеческой речи, например юмора или сарказма.

- *Разреженный входной вектор*. При токенизации каждое слово становится признаком (как мы убедились на предыдущем примере). Это приводит к разреженным структурам данных.

Окружение слова

Ключевое понимание того, как представлять для алгоритма текстовые данные (в частности, отдельные слова или лексемы), приходит из лингвистики. В эмбеддингах мы обращаем внимание на *окружение* (neighbourhood) каждого слова. Оно помогает определить значение и важность слова. Окружение слова — это набор других слов, находящихся рядом с данным и задающих его контекст.

Обратите внимание, что в BoW слово теряет свой контекст, так как он зависит от окружения, в котором слово находится.

Свойства эмбеддингов слов

Хорошие эмбеддинги слов обладают следующими четырьмя свойствами.

- *Они плотные*. По своей сути эмбеддинги являются факторными моделями. Таким образом, каждый компонент вектора эмбеддинга представляет собой величину некоторого (скрытого) признака. Обычно мы не знаем, что представляет собой этот признак; однако у нас будет очень мало (если вообще будут) нулей, которые могут стать причиной разреженного ввода.
- *Имеют низкую размерность*. Эмбеддинги имеют предопределенную размерность (выбранную в качестве гиперпараметра). В предыдущем примере использования BoW нам потребовалось $|V|$ входов для каждого слова, так что общий размер входных данных составил $|V| * n$, где n — количество слов, которые мы используем в качестве входных данных. В эмбеддингах слов размер ввода будет $d * n$, где d обычно составляет от 50 до 300. Учитывая тот факт, что объем больших текстовых массивов часто намного превышает 300 слов, мы значительно экономим на размере входных данных. Это, как мы видели, может привести к повышению точности при меньшем общем количестве экземпляров данных.
- *Содержат семантику области*. Это свойство, вероятно, самое неожиданное, но при этом и самое полезное. При правильном обучении эмбеддинги получают информацию о содержании своей предметной области.
- *Их легко обобщать*. Эмбеддинги способны улавливать обобщенные абстрактные шаблоны. Например, мы можем обучать эмбеддинги на кошках, оленях,

собаках и т. д., а модель сама поймет, что мы имеем в виду животных. При этом модель, никогда не обучаясь на овцах, все же будет правильно их классифицировать. Используя эмбеддинг, мы можем рассчитывать на правильный ответ.

Теперь обсудим использование рекуррентных нейронных сетей для обработки естественного языка.

РЕКУРРЕНТНЫЕ НЕЙРОСЕТИ В NLP

Рекуррентная нейросеть (Recurrent Neural Networks, RNN) — это традиционная сеть прямого распространения с обратной связью. Чтобы понять RNN, нужно представить ее как нейронную сеть с *состояниями* (states). Рекуррентные нейросети используются с любым типом данных для генерации и прогнозирования различных последовательностей. Обучение модели RNN заключается в формулировании последовательностей данных. RNN можно применять и для текстовых данных, поскольку предложения — это просто последовательности слов. Используя рекуррентные сети для NLP, можно добиться следующего:

- предсказать следующее слово при вводе текста;
- создать новый текст, придерживаясь стиля, уже использованного в тексте ранее (рис. 9.4).

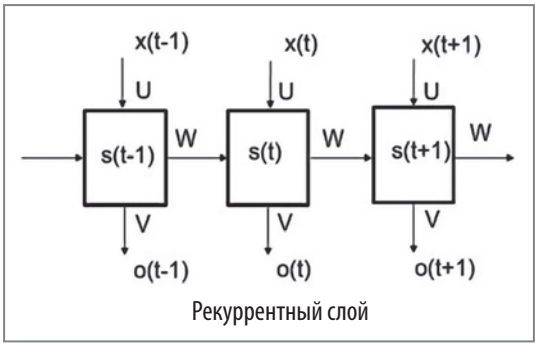


Рис. 9.4

Помните комбинацию слов, которая привела к правильному предсказанию? Процесс обучения RNN основан на тексте, который содержится в корпусе. Нейросеть обучается, уменьшая ошибку между предсказанным словом и фактическим следующим словом.

ИСПОЛЬЗОВАНИЕ NLP ДЛЯ АНАЛИЗА ЭМОЦИОНАЛЬНОЙ ОКРАСКИ ТЕКСТА

В этом разделе анализ эмоциональной окраски (или тональности) представлен на примере классификации плотного потока входящих твитов. Задача состоит в том, чтобы извлечь эмоциональную окраску твитов на определенную тему. Классификация эмоций количественно определяет полярность в режиме реального времени. После этого показатели суммируются в попытке отразить общие настроения по выбранной теме. Необходимо преодолеть трудности, связанные с содержанием и поведением потоковых данных Twitter, и при этом осуществить эффективный анализ в реальном времени. Для этого мы используем обученный классификатор NLP. Он подключается к потоку Twitter для определения полярности каждого твита (положительной, отрицательной или нейтральной). Далее происходят агрегирование и определение общей полярности всех твитов на выбранную тему. Теперь пошагово разберем этот процесс.

Сначала мы должны обучить классификатор. Для обучения классификатора нужен уже подготовленный набор исторических данных Twitter, обладающий той же структурой и трендами, что и данные реального времени. Поэтому мы используем набор данных с веб-сайта www.sentiment140.com. Он представляет собой размеченный вручную корпус текстов для анализа с огромным количеством твитов (более 1,6 миллиона). Твиты в этом наборе данных помечены одной из трех полярностей: 0 — для отрицательных, 2 — для нейтральных и 4 — для положительных. В дополнение к тексту твита корпус предоставляет идентификатор твита, дату, флаг и имя пользователя, который его написал. Рассмотрим операции, выполняемые с твитом в реальном времени, прежде чем он достигнет *обученного* классификатора:

1. Сначала твиты разбиваются на отдельные слова, называемые токенами (токенизация).
2. В результате токенизации возникает мешок слов, который представляет собой набор отдельных слов в тексте.
3. Твиты дополнительно фильтруются путем удаления цифр, знаков препинания и стоп-слов. Напомним, что стоп-слова — это распространенные слова и служебные части речи, такие как артикли, формы глагола «to be» и т. д. Они не содержат дополнительной информации, поэтому удаляются.
4. Кроме того, неалфавитные символы, такие как #@ и цифры, удаляются с помощью сопоставления с образцом, поскольку не имеют отношения к анализу тональности. Регулярные выражения используются только для сопостав-

ления буквенных символов, остальные же игнорируются. Это помогает уменьшить шум в потоке сообщений Twitter.

5. Результаты предыдущей фазы переносятся на этап стемминга. На этом этапе производные слова сводятся к своим корням — например, слово «*рыба*» имеет тот же корень, что и «*рыбалка*» и «*рыбак*». Для этого мы используем библиотеку NLTK, предоставляющую различные алгоритмы (например, стеммер Портера).
6. Как только данные обработаны, они преобразуются в структуру, называемую *терм-документной матрицей* (term document matrix TDM). TDM отображает термины и частоту их употребления в отфильтрованном корпусе.
7. Из TDM твит переходит к обученному классификатору (поскольку он обучен, то может обрабатывать твит), который вычисляет *важность эмоциональной полярности* (sentimental polarity importance, SPI) каждого слова, представляющей собой число от –5 до + 5. Положительный или отрицательный знак определяет тип эмоций, представленных этим конкретным словом, а величина отражает силу чувства. Это означает, что твит может быть классифицирован как положительный или отрицательный (рис. 9.5). Как только мы вычисляем полярность отдельных твитов, мы суммируем их общий SPI, чтобы агрегировать настроение источника. Например, общая полярность больше единицы указывает на то, что общее настроение твитов за наблюдаемый нами период времени является положительным.

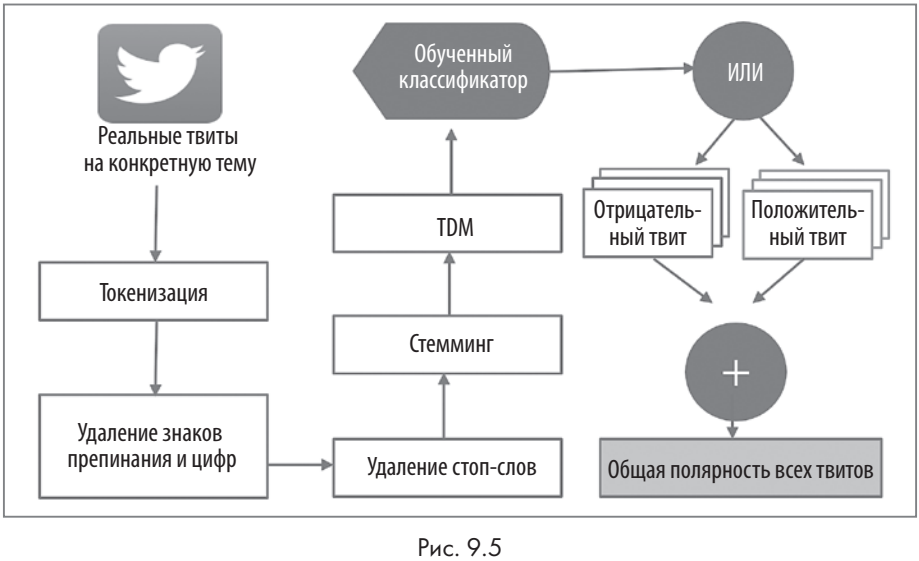


Рис. 9.5

Для получения необработанных твитов в реальном времени мы используем java-библиотеку Scala Twitter4J. Она предоставляет API для потоковой передачи из Twitter. API требует, чтобы пользователь зарегистрировал учетную запись разработчика в Twitter и ввел ряд параметров аутентификации. Этот интерфейс позволяет как получать случайные твиты, так и фильтровать их по выбранным ключевым словам. Для извлечения твитов мы использовали фильтры, определив ключевые слова.

Общая архитектура показана на рис. 9.5.

Анализ тональности текста применяется в разных сферах. К примеру, его можно использовать для классификации отзывов клиентов. Правительство может применить анализ полярности социальных сетей, чтобы определить эффективность своей политики. С помощью анализа тональности можно оценить успех рекламной кампании.

В следующем разделе мы разберем практический пример применения анализа тональности текстов для определения настроений в отзывах на фильмы.

ПРАКТИЧЕСКИЙ ПРИМЕР — АНАЛИЗ ТОНАЛЬНОСТИ В ОТЗЫВАХ НА ФИЛЬМЫ

Проанализируем настроения в отзывах на фильмы с помощью NLP. Для этого используем открытые источники данных с рецензиями, доступные по адресу http://www.cs.cornell.edu/people/pabo/movie-review-data/:

1. Импортируем набор данных, содержащий отзывы:

```
import numpy as np
import pandas as pd
```

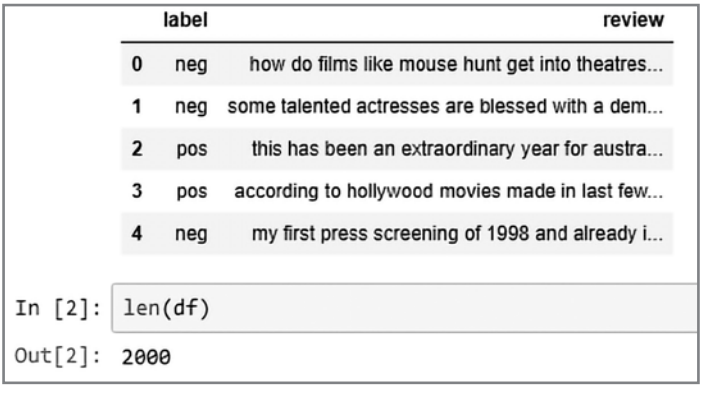
2. Загрузим данные и выведем несколько первых строк, чтобы увидеть их структуру (рис. 9.6):

```
df=pd.read_csv("moviereviews.tsv",sep='\t')
df.head()
```

   Обратите внимание, что набор данных содержит `2000` отзывов на фильмы. Из них половина — отрицательные, а половина — положительные.

3. Подготовим набор данных для обучения модели. Отбросим все пропущенные значения, которые есть в данных:

```
df.dropna(inplace=True)
```



	label	review
0	neg	how do films like mouse hunt get into theatres...
1	neg	some talented actresses are blessed with a dem...
2	pos	this has been an extraordinary year for austra...
3	pos	according to hollywood movies made in last few...
4	neg	my first press screening of 1998 and already i...

```
In [2]: len(df)
Out[2]: 2000
```

Рис. 9.6

4. Теперь нам нужно убрать пробелы. Пробелы не являются null-значениями, но их необходимо удалить. Для этого выполним итерацию по каждой строке во входном `DataFrame`. Используем `.itertuples()` для доступа к каждому полю:

```
blanks=[]

for i,lb,rv in df.itertuples():
    if rv.isspace():
        blanks.append(i)
df.drop(blanks,inplace=True)
```

   Обратите внимание, что мы использовали `i`, `lb` и `rv` для столбцов индекса, метки и обзора.

Разделим данные на контрольный и обучающий наборы:

1. Первый шаг — указать признаки и метку, а затем разделить данные на обучающие и контрольные:

```
from sklearn.model_selection import train_test_split

X = df['review']
y = df['label']

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

   Теперь у нас есть наборы данных для обучения и контроля.

2. Импортируем необходимые библиотеки и построим модель:

```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
```

```
# Наивный байесовский классификатор:
text_clf_nb = Pipeline([('tfidf', TfidfVectorizer()),
                        ('clf', MultinomialNB()),
])
```

Обратите внимание, что мы используем `tfidf` для количественной оценки важности точки данных в коллекции.

Далее обучим модель с использованием наивного байесовского классификатора, а затем протестируем обученную модель.

Выполним следующие действия:

1. Обучим модель, используя подготовленные наборы данных для контроля и обучения:

```
text_clf_nb.fit(X_train, y_train)
```

2. Запустим прогнозы и проанализируем результаты:

```
# Предсказанный набор
predictions = text_clf_nb.predict(X_test)
```

Теперь выведем матрицу ошибок, чтобы оценить производительность модели. Рассмотрим `precision` (точность), `recall` (полноту), `f1-score` (f1-меру) и `accuracy` (долю правильных ответов) (рис. 9.7).

```
In [23]: from sklearn.metrics import confusion_matrix,classification_report,accuracy_score

In [24]: print(confusion_matrix(y_test,predictions))
         [[259  23]
          [102 198]]

In [25]: print(classification_report(y_test,predictions))
                       precision    recall  f1-score   support

                  neg       0.72      0.92      0.81       282
                  pos       0.90      0.66      0.76       300

             accuracy                           0.79       582
            macro avg       0.81      0.79      0.78       582
         weighted avg       0.81      0.79      0.78       582


In [26]: print(accuracy_score(y_test,predictions))
         0.7852233676975945
```

Рис. 9.7

Эти метрики производительности и дают нам представление о качестве прогнозов. При `accuracy` 0.78 делаем вывод, что модель успешно обучена. Она способна предсказать, какой тип отзыва на конкретный фильм можно ожидать.

РЕЗЮМЕ

В этой главе мы обсудили алгоритмы, связанные с обработкой естественного языка, и изучили общую терминологию. Далее познакомились с методикой реализации стратегии NLP, называемой мешком слов. Затем разобрали концепцию эмбеддингов слов и использование нейронных сетей в NLP. Наконец, обратились к реальному примеру, в котором применили концепции, изученные в этой главе, для прогнозирования тональности отзывов на фильмы на основе текста отзыва. Теперь мы умеем использовать NLP для классификации текста и анализа тональности.

В следующей главе мы рассмотрим механизмы рекомендаций: изучим их типы и научимся использовать эти механизмы для решения реальных задач.

10

Рекомендательные системы

Рекомендательная система (recommendation engine) — это механизм предоставления рекомендаций, основанных на доступной информации о предпочтениях пользователей и характеристиках продукта. Цель рекомендательной системы состоит в том, чтобы выявить сходство между товарами (услугами) и/или сформулировать взаимодействие между пользователем и товаром.

Мы начнем с обсуждения основных понятий и различных типов рекомендательных систем. Далее узнаем, как именно они предлагают товары разным пользователям, а также рассмотрим их ограничения. Наконец, научимся использовать рекомендательные системы для решения реальных задач.

Глава включает следующие разделы:

- Введение в рекомендательные системы.
- Типы рекомендательных систем.
- Ограничения рекомендательных систем.
- Области практического применения.
- Практический пример — создание системы, которая рекомендует фильмы подписчикам.

К концу главы вы поймете, как использовать рекомендательные системы для предложения товаров на основе выбранных критериев предпочтения.

Начнем с изучения базовых концепций.

ВВЕДЕНИЕ В РЕКОМЕНДАТЕЛЬНЫЕ СИСТЕМЫ

Рекомендательные системы — это методы, первоначально разработанные исследователями для прогнозирования товаров, которые, скорее всего, заинтересуют клиента. Способность таких систем давать персонализированные рекомендации по товарам делает их, пожалуй, наиболее важной технологией в мире онлайн-покупок.

В приложениях электронной коммерции используются сложные алгоритмы для повышения качества обслуживания клиентов; эти алгоритмы позволяют поставщикам товаров и услуг настраивать свои предложения в соответствии с предпочтениями покупателей.

В 2009 году Netflix предложил 1 миллион долларов за алгоритм, способный улучшить существующий механизм рекомендаций (Cinematch) более чем на 10 %. Приз получила команда BellKor's Pragmatic Chaos.

ТИПЫ РЕКОМЕНДАТЕЛЬНЫХ СИСТЕМ

Существуют три типа рекомендательных систем:

- на основе контента;
- на основе коллаборативной, или совместной, фильтрации (collaborative filtering);
- гибридные.

Рекомендательные системы на основе контента

Основная идея *рекомендательной системы на основе контента* (content-based recommendation engine) состоит в том, чтобы предлагать позиции (товары), аналогичные тем, к которым пользователь ранее проявлял интерес. Эффективность такого механизма зависит от возможности количественно оценить сходство одного товара с другими.

Давайте посмотрим на следующую диаграмму. Если *Пользователь 1* прочитал *Документ 1*, то мы можем рекомендовать пользователю *Документ 2*, который аналогичен *Документу 1* (рис. 10.1).

Но как определить, какие позиции похожи между собой? Рассмотрим несколько методов поиска сходства.

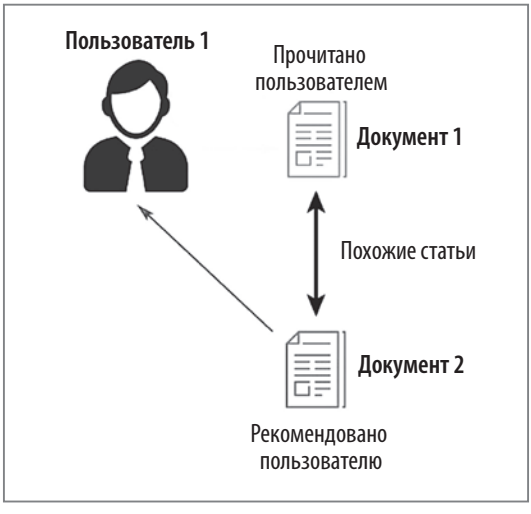


Рис. 10.1

Поиск сходства между неструктурированными документами

Одним из способов определения сходства между документами является предварительная обработка входных неструктурированных документов. В результате мы получаем структуру данных, которая называется *терм-документной матрицей (TDM)*. Она показана на следующей диаграмме (рис. 10.2).

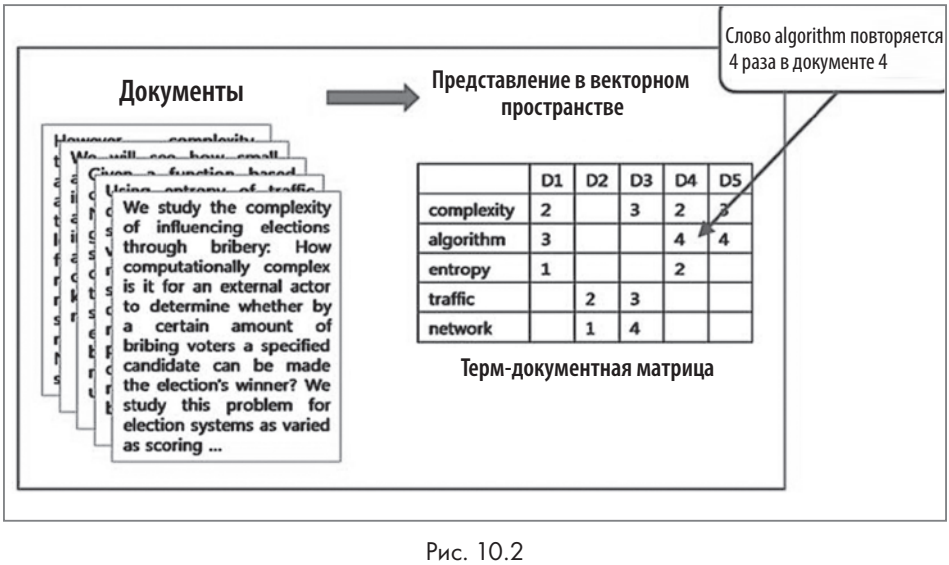


Рис. 10.2

TDM содержит весь глоссарий слов в виде строк и все документы в виде столбцов. С ее помощью можно выявить похожие документы на основе выбранного критерия. Например, Google News предлагает пользователю новости, базируясь на их сходстве с новостями, к которым он уже проявлял интерес.

Создав TDM, мы получаем два способа количественной оценки сходства между документами:

- *Подсчет частоты употребления*. Данный подход подразумевает, что важность слова прямо пропорциональна его частоте. Это самый простой способ расчета важности.
- *Использование TF-IDF*. Этот показатель представляет важность каждого слова в контексте задачи и является произведением двух сомножителей:
  - *Частота слова* (TF, term frequency). Это количество раз, когда слово или термин появляется в документе. TF напрямую коррелирует с важностью слова.
  - *Обратная частота документов* (IDF, inverse document frequency). Обратите внимание: *частота документов* (DF, document frequency) — это количество документов, содержащих искомое слово. В противоположность DF, IDF отражает меру уникальности слова и соотносит ее с важностью этого слова.
  - TF и IDF оценивают количественно важность слова в контексте задачи, а их комбинация, TF-IDF, служит хорошим показателем важности каждого слова и является более сложной альтернативой простому подсчету частоты.

Матрица совместной встречаемости

Использование *матрицы совместной встречаемости* (co-occurrence matrix) основано на предположении, что если в большинстве случаев два определенных товара покупаются одновременно, то они, скорее всего, похожи (либо, по крайней мере, принадлежат к одной и той же категории, товары из которой обычно покупаются вместе).

Например, гель для бритья и бритва почти всегда используются одновременно. Поэтому, если человек покупает бритву, разумно предположить, что он также купит и гель для бритья.

Проанализируем покупательские привычки четырех потребителей (табл. 10.1).

Таблица 10.1

	Бритва	Яблоко	Крем для бритья	Велосипед	Хумус
Майк	1	1	1	0	1
Тейлор	1	0	1	1	1
Елена	0	0	0	1	0
Амина	1	0	1	0	0

Создадим матрицу совместной встречаемости на основе этих данных (табл. 10.2).

Таблица 10.2

	Бритва	Яблоко	Крем для бритья	Велосипед	Хумус
Бритва	-	1	3	1	1
Яблоко	1	-	1	0	1
Крем для бритья	3	1	-	1	2
Велосипед	1	0	1	-	1
Хумус	1	1	2	1	-

Матрица совместной встречаемости суммирует вероятность покупки двух товаров вместе. Давайте посмотрим, как ее использовать.

Рекомендательные системы на основе коллаборативной фильтрации

Алгоритм *коллаборативной фильтрации* (collaborative filtering) основан на анализе покупательского поведения. Базовое предположение состоит в том, что если два пользователя проявляют интерес в основном к одним и тем же товарам, мы можем классифицировать этих покупателей как похожих. Другими словами, можно предположить следующее:

- Если совпадение в истории покупок двух пользователей превышает пороговое значение, их можно классифицировать как похожих.

- Товары, которые не пересекаются в истории покупок похожих пользователей, становятся предметом будущих рекомендаций на основе коллаборативной фильтрации.

Рассмотрим конкретный пример. У нас есть два пользователя, *Майк* и *Елена* (рис. 10.3).

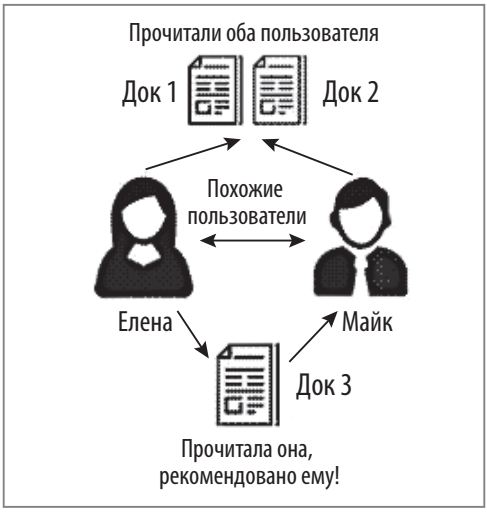


Рис. 10.3

Обратите внимание:

- Майк и Елена проявили интерес к одним и тем же позициям, *Документу 1* и *Документу 2*.
- Основываясь на схожих паттернах покупательского поведения, мы классифицируем пользователей как похожих.
- Если Елена в данный момент читает *Документ 3*, то мы можем предложить *Документ 3* и Майку.

Обратите внимание, что стратегия предложения позиций пользователям на основе их истории покупок будет срабатывать не всегда.

Предположим, что Елена и Майк проявили интерес к *Документу 1*, посвященному фотографии (потому что они оба любят фотографировать). Кроме того, Елена и Майк проявили интерес к *Документу 2*, который посвящен облачным вычислениям, опять же, потому что им обоим интересна эта тема. Основываясь

на коллаборативной фильтрации, мы классифицировали Елену и Майка как похожих пользователей. Теперь Елена читает *Документ 3* (журнал о женской моде). Если следовать алгоритму, мы должны предложить этот журнал и Майку, который, скорее всего, не проявит к нему особого интереса.

В 2012 году американский супермаркет Target экспериментировал с коллаборативной фильтрацией для рекомендации товаров покупателям. На основе профилей клиентов алгоритм классифицировал отца и его дочь-подростка как похожих покупателей. В итоге Target отправил отцу скидочный купон на подгузники, детскую смесь и детскую кроватку. Но отец не знал о беременности своей дочери...

Алгоритм коллаборативной фильтрации не зависит от какой-либо другой информации, является автономным и базируется на изменении поведения пользователей и совместных рекомендациях.

Гибридные рекомендательные системы

Итак, мы обсудили системы на основе контентной и коллаборативной фильтрации. Оба типа рекомендательных систем могут быть объединены для создания гибридной системы. Для этого нужно выполнить следующие действия:

- Создать матрицу сходства элементов.
- Создать матрицы предпочтений пользователей.
- Выработать рекомендации.

Рассмотрим подробнее обозначенные шаги.

Создание матрицы сходства элементов

Построение гибридной системы начнем с создания *матрицы сходства* (similarity matrix) с использованием рекомендаций на основе контента. Это можно проделать с помощью матрицы совместной встречаемости или любой меры расстояния для количественной оценки сходства между элементами.

Предположим, у нас есть пять товаров. Используя рекомендации на основе содержания, создадим матрицу, которая отражает сходство между товарами и выглядит следующим образом (рис. 10.4).

Далее объединим матрицу сходства с матрицей предпочтений для выработки рекомендаций.

	Товар 1	Товар 2	Товар 3	Товар 4	Товар 5
Товар 1	10	5	3	2	1
Товар 2	5	10	6	5	3
Товар 3	3	6	10	1	5
Товар 4	2	5	1	10	3
Товар 5	1	3	5	3	10

Рис. 10.4

Генерация векторов предпочтений пользователей

Основываясь на истории покупок, создадим *вектор предпочтений* (preference vector), который отражает интересы пользователей.

Предположим, мы должны настроить рекомендации для интернет-магазина под названием KentStreetOnline, в котором продается 100 уникальных товаров. KentStreetOnline — популярный магазин с миллионом активных подписчиков. Важно отметить, что нам нужно сгенерировать только одну матрицу сходства размером 100 на 100. Необходимо также сгенерировать вектор предпочтений для каждого пользователя (в общей сложности это 1 миллион векторов предпочтений).

Каждая запись вектора представляет предпочтение для одного товара. Значение первой строки означает, что вес предпочтения для *Товара 1* равен *4*; значение второй строки означает, что для *Товара 2* предпочтение не задано. Это наглядно показано на рис. 10.5.

Товар 1	4
Товар 2	0
Товар 3	0
Товар 4	5
Товар 5	0

Рис. 10.5

Перейдем к выработке рекомендаций на основе матрицы сходства (*S*) и матрицы предпочтений пользователя (*U*).

Выработка рекомендаций

Чтобы дать рекомендации, нужно перемножить матрицы. Пользователи с большей вероятностью заинтересуются товаром, часто встречающимся вместе с товаром, которому они дали высокую оценку:

$$Matrix[S] \times Matrix[U] = Matrix[R].$$

Этот расчет показан на следующей диаграмме (рис. 10.6).

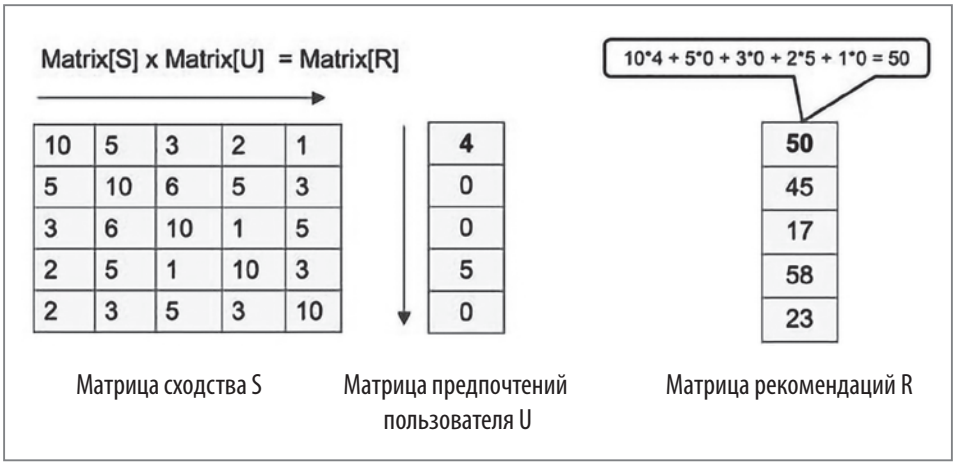


Рис. 10.6

Для каждого пользователя создается отдельная результирующая матрица. Числа в матрице рекомендаций, *Matrix[R]*, отражают прогнозируемый интерес пользователя к каждому из товаров. Например, в итоговой матрице четвертый элемент содержит наибольшее число — 58. Поэтому этот товар настоятельно рекомендуется данному конкретному пользователю.

Теперь рассмотрим ограничения различных рекомендательных систем.

ОГРАНИЧЕНИЯ РЕКОМЕНДАТЕЛЬНЫХ СИСТЕМ

Чтобы выработать рекомендации для широкого круга пользователей, используются алгоритмы прогнозирования. Это мощная технология, но нужно при-

нимать во внимание и ее недостатки. Рассмотрим различные ограничения рекомендательных систем.

Проблема холодного старта

Очевидно, что для осуществления коллаборативной фильтрации необходимы данные о предпочтениях пользователей. Для нового пользователя такие данные, скорее всего, отсутствуют. В результате алгоритм сходства будет основан на предположениях, которые могут оказаться неточными. Подробная информация о новых позициях также появляется не сразу, что затрудняет работу системы на основе контента. Требование наличия данных о позициях в ассортименте и пользователях для создания высококачественных рекомендаций называется *проблемой холодного старта* (cold start problem).

Требования к метаданным

Методы на основе контента требуют детального описания позиций для измерения сходства. Такие описания могут быть недоступны, что влияет на качество прогнозов.

Проблема разреженности данных

Среди огромного количества товаров пользователь, как правило, оценивает только несколько позиций, что приводит к очень разреженной матрице рекомендаций.

У Amazon около миллиарда пользователей и миллиард товаров. Говорят, что механизм рекомендаций Amazon содержит самые разреженные данные по сравнению с рекомендательными системами во всем мире.

Предвзятость из-за социального влияния

Социальное влияние играет важную роль при выработке рекомендаций. Взаимоотношения между людьми можно рассматривать как фактор, от которого зависят предпочтения пользователя. Друзья, как правило, покупают похожие товары, а также дают схожие оценки.

Ограниченные данные

Ограниченное количество отзывов затрудняет точное измерение сходства пользователей при работе рекомендательных систем.

ОБЛАСТИ ПРАКТИЧЕСКОГО ПРИМЕНЕНИЯ

Вот несколько примеров практического применения рекомендательных систем:

- Две трети фильмов на Netflix находятся в рекомендациях.
- 35 % продаж Amazon происходит благодаря рекомендациям.
- В новостях Google рекомендации генерируют на 38 % больше кликов.
- Прогноз пользовательских предпочтений в отношении товара базируется на рейтинге других товаров.
- Студентам университетов предлагаются курсы на основе их потребностей и предпочтений.
- На онлайн-порталах по поиску работы резюме сопоставляются с вакансиями.

Перейдем к решению реальной задачи с помощью механизма рекомендаций.

ПРАКТИЧЕСКИЙ ПРИМЕР — СОЗДАНИЕ РЕКОМЕНДАТЕЛЬНОЙ СИСТЕМЫ

Давайте создадим систему, которая сможет рекомендовать пользователям фильмы для просмотра. Мы используем данные, собранные исследовательской группой GroupLens Research Университета Миннесоты.

Выполним следующие действия.

1. Прежде всего импортируем необходимые библиотеки:

```
import pandas as pd
import numpy as np
```

2. Теперь импортируем наборы данных `user_id` и `item_id`:

```
df_reviews = pd.read_csv('reviews.csv')
df_movie_titles = pd.read_csv('movies.csv',index_col=False)
```

3. Объединим два DataFrame по ID фильма:

```
df = pd.merge(df_users, df_movie_titles, on='movieId')
```

Заголовки таблицы *df* после выполнения предыдущего кода выглядят следующим образом (рис. 10.7).

Out[5]:

	userId	movieId	rating	timestamp	title	genres
0	1	1	4.0	964982703	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	5	1	4.0	847434962	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	7	1	4.5	1106635946	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
3	15	1	2.5	1510577970	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
4	17	1	4.5	1305696483	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy

Рис. 10.7

Ниже представлена расшифровка названий столбцов:

- *userid*: уникальный идентификатор пользователя;
- *movieid*: уникальный идентификатор фильма;
- *rating*: оценка фильма от 1 до 5;
- *timestamp*: отметка времени, когда фильм был оценен;
- *title*: название фильма;
- *genres*: жанр фильма.

4. Чтобы увидеть общие тенденции входных данных, вычислим среднее значение и количество оценок для каждого фильма, используя groupby по столбцам `title` и `rating` (рис. 10.8).

Out[6]:

title	rating	number_of_ratings
'71 (2014)	4.0	1
'Hellboy': The Seeds of Creation (2004)	4.0	1
'Round Midnight (1986)	3.5	2
'Salem's Lot (2004)	5.0	1
'Til There Was You (1997)	4.0	2

Рис. 10.8

5. Чтобы подготовить данные для рекомендательной системы, преобразуем набор данных в матрицу со следующими характеристиками:
   - `title` — столбцы матрицы;
   - `userid` — будет использован как индекс;
   - `rating` — значение в таблице.

   Для этого используем функцию `pivot_table` из DataFrame:

```
movie_matrix = df.pivot_table(index='userId', columns='title',
values='rating')
```

   Обратите внимание, что этот код создаст очень разреженную матрицу.

6. Теперь применим созданную матрицу для рекомендации фильмов. Для этого выберем пользователя, который смотрел фильм «Аватар» (2009). Прежде всего найдем всех пользователей, которые проявили интерес к этому фильму:

```
Avatar_user_rating = movie_matrix['Avatar (2009)']
Avatar_user_rating = Avatar_user_rating.dropna()
Avatar_user_rating.head()
```

7. Далее попробуем предложить фильмы, которые соотносятся с «Аватаром». Рассчитаем корреляцию DataFrame `Avatar_user_rating` с `movie_matrix`:

```
similar_to_Avatar=movie_matrix.corrwith(Avatar_user_rating)
corr_Avatar = pd.DataFrame(similar_to_Avatar,
columns=['correlation'])
corr_Avatar.dropna(inplace=True)
corr_Avatar = corr_Avatar.join(df_ratings['number_of_ratings'])
corr_Avatar.head()
```

   Это даст следующий результат (рис. 10.9).

Out[12]:

title	correlation	number_of_ratings
'burbs, The (1989)	0.353553	17
(500) Days of Summer (2009)	0.131120	42
*batteries not included (1987)	0.785714	7
10 Things I Hate About You (1999)	0.265637	54
10,000 BC (2008)	-0.075431	17

Рис. 10.9

Эти фильмы можно использовать в качестве рекомендаций для пользователя.

РЕЗЮМЕ

В этой главе мы узнали о рекомендательных системах и научились выбирать подходящую в соответствии с задачей. Мы выяснили, как создать матрицу сходства, чтобы подготовить данные для рекомендательной системы. Наконец, мы применили эти знания для решения практической задачи — предложение фильмов пользователям на основе их прошлых предпочтений.

В следующей главе мы сосредоточимся на алгоритмах, которые используются для понимания и обработки данных.

Часть III

РАСШИРЕННЫЕ ВОЗМОЖНОСТИ

Как следует из названия, в этой части мы рассмотрим алгоритмы более высокого уровня. Ключевыми темами являются криптографические и крупномасштабные алгоритмы. В последней главе этой части (и всей книги) представлены практические рекомендации, полезные при реализации алгоритмов. Нас ждут следующие главы:

- Глава 11. Алгоритмы обработки данных.
- Глава 12. Криптография.
- Глава 13. Крупномасштабные алгоритмы.
- Глава 14. Практические рекомендации.

11

Алгоритмы обработки данных

Эта глава посвящена алгоритмам, ориентированным на данные. Мы сфокусируемся на трех задачах: хранении, потоковой передаче и сжатии данных. Начнем с краткого обзора алгоритмов обработки данных. Затем обсудим стратегии хранения данных, научимся применять алгоритмы к потоковым данным и изучим различные методологии сжатия. Наконец, применим изученные концепции для анализа тональности твитов в режиме реального времени.

К концу этой главы вы будете иметь представление о концепциях и компромиссах, связанных с разработкой алгоритмов, ориентированных на данные.

Итак, в главе рассматриваются следующие темы:

- Классификация данных.
- Алгоритмы хранения данных.
- Алгоритмы сжатия данных.
- Алгоритмы потоковой передачи данных.

Давайте познакомимся с основными понятиями.

ЗНАКОМСТВО С АЛГОРИТМАМИ ОБРАБОТКИ ДАННЫХ

Осознаем мы это или нет, но мы живем в эпоху больших данных. Чтобы получить представление о том, сколько данных генерируется каждую секунду, взгляните на некоторые цифры, опубликованные Google за 2019 год. Как мы знаем,

Google Фото — это мультимедийное хранилище для фотографий. В 2019 году в Google Фото ежедневно загружалось в среднем 1,2 миллиарда фотографий и видео. Кроме того, в среднем 400 часов видео (1 ПБ данных) ежеминутно выкладывалось на YouTube. Можно с уверенностью сказать, что объем генерируемых данных вырос просто лавинообразно.

Сегодня интерес к алгоритмам на основе данных связан с возможностью извлечь как ценную информацию, так и закономерности. При правильном использовании данные могут стать основой для принятия решений в сфере политики, маркетинга, управления и анализа тенденций.

Так что вполне понятно, что алгоритмы, которые работают с данными, приобретают все большее значение. Разработка таких алгоритмов — это область активных исследований. Нет сомнений в том, что поиск наилучших способов использования данных для получения измеримой выгоды находится в центре внимания различных организаций, предприятий и правительств по всему миру. Но данные в необработанном виде бесполезны. Чтобы извлечь информацию из таких данных, их необходимо обработать, подготовить и проанализировать.

Для начала сами данные нужно где-то хранить. Все большее значение приобретают эффективные методологии хранения данных. Из-за ограничений физической памяти однонодовых систем большие данные могут находиться только в распределенном хранилище. Оно состоит из нескольких нод, соединенных высокоскоростными каналами связи. Изучение алгоритмов обработки данных разумно начать с рассмотрения различных алгоритмов хранения.

Прежде всего разделим данные по категориям.

Классификация данных

Давайте разберемся, как классифицируются данные в контексте создания алгоритмов их обработки. Как обсуждалось в главе 1, для этого используются параметры объема, разнообразия и скорости (3V). Такая классификация может стать основой для разработки алгоритмов, предназначенных для хранения и обработки данных.

Рассмотрим эти характеристики по порядку.

- *Объем* (Volume) определяет количество данных, которые необходимо хранить и обрабатывать. По мере увеличения объема задача становится трудоемкой и требует выделения достаточного количества ресурсов для хранения, кэширования и обработки. Для обозначения огромных массивов данных, которые

не могут быть обработаны одной нодой, обычно используется термин *большие данные* (Big Data).

- *Скорость* (Velocity) обозначает скорость, с которой генерируются новые данные. Обычно высокоскоростные данные называют «горячими данными» или «горячим потоком», а низкоскоростные — «холодными данными» или «холодным потоком». Во многих приложениях данные будут представлять собой смесь горячих и холодных потоков, которые сначала необходимо подготовить и объединить в сводную таблицу и только потом использовать в алгоритме.
- *Разнообразие* (Variety) относится к различным типам структурированных и неструктурированных данных, которые необходимо объединить в таблицу, прежде чем они могут быть использованы алгоритмом.

В следующем разделе представлены связанные с этим компромиссы и различные варианты разработки алгоритмов хранения.

АЛГОРИТМЫ ХРАНЕНИЯ ДАННЫХ

Надежное и эффективное хранилище данных — это сердце распределенной системы. Если хранилище создано для аналитики, то оно также называется *озером* данных (data lake). Хранилище объединяет в одном месте данные из разных предметных областей. Прежде всего разберем некоторые вопросы, связанные с хранением данных в распределенной системе.

Стратегии хранения данных

В первые годы цифровой эры для проектирования хранилища данных использовалась однонодовая архитектура. По мере роста объемов данных основным способом стало распределенное хранение. Правильная стратегия хранения данных в распределенной среде зависит от типа данных и ожидаемой схемы их использования, а также от их нефункциональных требований. Для анализа требований к распределенному хранилищу нам понадобится теорема CAP. Она даст основу для разработки стратегии хранения данных в распределенной системе.

Теорема CAP

В 1998 году Эрик Брюер предложил теорему, которая позже стала известна как теорема CAP. В ней освещаются различные компромиссы, связанные с разработкой распределенной системы хранения данных.

Чтобы понять теорему CAP, определим следующие три характеристики распределенных систем хранения данных: *согласованность*, *доступность* и *устойчивость к разделению*. CAP — это аббревиатура, состоящая из первых букв этих понятий:

- *Согласованность* (consistency, C). Распределенное хранилище состоит из ряда нод. Любая из этих нод может использоваться для чтения, записи или обновления записей. Согласованность гарантирует, что в определенное время t_1 независимо от того, какую ноду мы используем для чтения данных, мы получим одинаковый результат.

  Каждая операция чтения либо возвращает последние данные, согласованные в рамках распределенной системы, либо выдает сообщение об ошибке.

- *Доступность* (availability, A). Эта характеристика гарантирует, что любая нода в распределенной системе способна немедленно обработать запрос с согласованностью или без нее.

- *Устойчивость к разделению* (partition tolerance, P). В распределенной системе несколько нод соединены через коммуникационную сеть. Устойчивость к разделению гарантирует, что в случае сбоя связи между небольшим подмножеством нод (одной или несколькими) система останется работоспособной. Обратите внимание, что для обеспечения устойчивости к разделению данные должны быть реплицированы на достаточное количество нод.

Используя эти характеристики, теорема CAP обобщает компромиссы, связанные с архитектурой и дизайном распределенной системы. В частности, теорема CAP гласит, что система хранения может обладать только двумя характеристиками из представленных.

Это демонстрируется на следующей схеме (рис. 11.1).

Рис. 11.1

Соответственно, существуют три типа распределенных систем хранения данных:

- система CA (согласованность + доступность);
- система AP (устойчивость к разделению + доступность);
- система CP (согласованность + устойчивость к разделению).

Рассмотрим их по очереди.

Системы CA

Традиционные системы с одной нодой — это *системы CA*: если система не является распределенной, то незачем беспокоиться об устойчивости к разделению. В этом случае система обладает согласованностью и доступностью.

Базы данных с одной нодой, такие как Oracle или MySQL, являются примерами систем CA.

Системы AP

Системы AP — распределенные системы хранения, направленные на доступность. Это высокочувствительные системы, способные жертвовать согласованностью, если нужно, для размещения высокоскоростных данных. Благодаря этому системы AP подходят для немедленной обработки запросов пользователей. Типичными запросами являются чтение или запись быстро меняющихся данных. Обычно AP используются в системах мониторинга в реальном времени, таких как сенсорные сети.

Высокоскоростная распределенная база данных Cassandra служит хорошим примером системы AP.

Давайте выясним, как можно использовать систему AP. Например, Transport Canada намеревается отслеживать движение на одной из автомагистралей в Оттаве с помощью сети датчиков, установленных в разных местах на шоссе. В этом случае лучшим решением будет использование системы AP для распределенного хранения данных.

Системы CP

Системы CP обладают как согласованностью, так и устойчивостью к разделению. Они гарантируют согласованность до того момента, пока процесс чтения не извлечет значение.

Типичный случай использования систем CP — это хранение текстовых файлов в формате `JSON`. Хранилища документов, такие как MongoDB, являются систе-

мами обработки данных, настроенными на согласованность в распределенной среде.

Распределенное хранилище становится все более важной частью современной ИТ-инфраструктуры. Оно должно быть тщательно спроектировано с учетом характеристик данных и требований задачи. Разделение систем хранения данных на CA, AP и CP полезно для понимания различных компромиссов, связанных с их проектированием.

Теперь обратимся к алгоритмам потоковой передачи данных.

АЛГОРИТМЫ ПОТОКОВОЙ ПЕРЕДАЧИ ДАННЫХ

Данные могут быть *ограниченными* (bounded) и *неограниченными* (unbounded). Ограниченные данные — это данные в состоянии покоя, которые обычно обрабатываются в пакетном режиме. Потоковая передача — это обработка неограниченных данных. Приведем пример. Предположим, мы анализируем банковские транзакции на предмет мошенничества. Чтобы выявить мошеннические транзакции, которые произошли 7 дней назад, нужно просмотреть данные в состоянии покоя, — это пример пакетного процесса.

Но если необходимо обнаружить мошенничество в режиме реального времени, это будет уже потоковой передачей. Итак, *алгоритмы потоковой передачи данных* обрабатывают потоки данных. Основная идея состоит в том, чтобы разделить поток входных данных на пакеты, которые затем обрабатываются нодой. Потоковые алгоритмы должны быть отказоустойчивыми и способными обрабатывать данные, поступающие с определенной скоростью. В наши дни спрос на анализ закономерностей в реальном времени постоянно растет, поэтому увеличивается потребность в потоковой обработке. Обратите внимание, что при потоковой передаче данные должны обрабатываться быстро, это необходимо учитывать при создании алгоритма.

Применение потоковой передачи

Существует множество примеров полезного применения потоковой передачи.

Вот лишь некоторые из них:

- Выявление мошенничества.
- Мониторинг системы.

- Умная маршрутизация заказов.
- Дашборды для мониторинга в реальном времени.
- Датчики движения на автомагистралях.
- Операции с кредитными картами.
- Действия игроков в многопользовательских онлайн-играх.

АЛГОРИТМЫ СЖАТИЯ ДАННЫХ

Алгоритмы сжатия данных обеспечивают уменьшение размера данных.

В этой главе представлен конкретный тип алгоритмов сжатия данных — *алгоритмы сжатия без потерь* (lossless compression algorithms).

Алгоритмы сжатия без потерь

Это алгоритмы, способные сжимать данные таким образом, чтобы в дальнейшем их можно было распаковать без какой-либо потери информации. Они используются, если после распаковки важно получить точные исходные файлы. Типичные случаи применения:

- Сжатие документов.
- Сжатие и упаковка исходного кода и исполняемых файлов.
- Преобразование большого количества небольших файлов в небольшое количество больших файлов.

Основные методы сжатия без потерь

Сжатие данных основано на принципе, согласно которому данные часто используют больше битов, чем необходимо (согласно энтропии данных). Напомним, что понятие энтропии используется для определения информативности данных. Это означает, что возможно более эффективное битовое представление одной и той же информации. Поиск и формулирование оптимального битового представления — основа разработки алгоритмов сжатия. Методы сжатия без потерь используют избыточность данных. В конце 80-х годов Зив и Лемпель предложили такие методы на основе словарей. Эти алгоритмы мгновенно обрели успех из-за своей скорости и хорошей степени сжатия. Они были использованы для создания некогда популярной утилиты `compress` на базе Unix. Кроме того, вездесущий формат изображений `gif` также использует эти методы

сжатия. Алгоритмы сжатия обрели популярность, поскольку они позволяли представлять одну и ту же информацию меньшим количеством битов, экономя место и пропускную способность. Позже они легли в основу разработки утилиты `zip` и ее вариантов. В основе стандарта сжатия V.44, используемого в модемах, лежат те же принципы.

Кодирование Хаффмана

Кодирование Хаффмана (Huffman coding) — один из старейших методов сжатия данных, основанный на построении дерева Хаффмана, которое используется как для кодирования, так и для декодирования данных. Кодирование Хаффмана представляет содержимое данных в более компактной форме, пользуясь тем, что некоторые данные (например, определенные символы алфавита) чаще появляются в потоке данных. При использовании кодировок разной длины (коротких — для наиболее часто встречающихся символов и длинных — для редко встречающихся) данные занимают меньше места.

Познакомимся с несколькими терминами, связанными с кодированием Хаффмана.

- *Кодирование*. Преобразование одной формы представления данных в другую. Итоговая форма должна быть как можно более сжатой.
- *Кодовое слово*. Определенный символ в закодированной форме.
- *Кодирование фиксированной длины*. Каждый закодированный символ, то есть кодовое слово, использует одинаковое количество битов.
- *Кодирование переменной длины*. Кодовые слова используют разное количество битов.
- *Оценка кода*. Ожидаемое количество битов на кодовое слово.
- *Коды без префикса*. Ни одно кодовое слово не является префиксом любого другого кодового слова.
- *Декодирование*. Код переменной длины не должен содержать никаких префиксов.

Чтобы понять последние два термина, обратимся к табл. 11.1.

Можно сделать следующий вывод:

- *Код фиксированной длины* для этой таблицы равен 3.
- *Код переменной длины* равен 45(*1) + 0.13(*3) + 0.12(*3) + 0.16(*3) + + 0.09(*4) + 0.05(*4) = 2.24.

Таблица 11.1

Символ	Частота	Код фиксированной длины	Код переменной длины
L	0.45	000	0
M	0.13	001	101
N	0.12	010	100
X	0.16	011	111
Y	0.09	100	1101
Z	0.05	101	1100

На следующей диаграмме показано дерево Хаффмана, созданное из предыдущего примера (рис. 11.2).

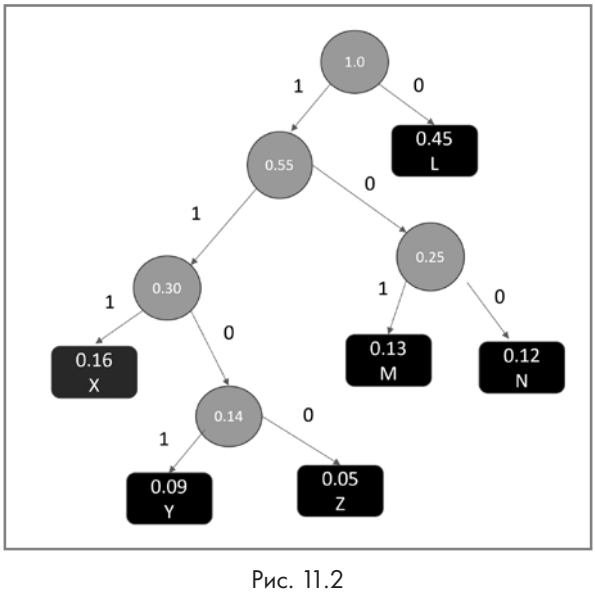


Рис. 11.2

Обратите внимание, что кодирование Хаффмана заключается в преобразовании данных в дерево Хаффмана, которое и обеспечивает сжатие. Декодирование (декомпрессия) возвращает данные в исходный формат.

ПРАКТИЧЕСКИЙ ПРИМЕР — АНАЛИЗ ТОНАЛЬНОСТИ ТВИТОВ В РЕЖИМЕ РЕАЛЬНОГО ВРЕМЕНИ

Известно, что в Твиттере каждую секунду появляется почти 7000 твитов на самые разные темы. Попробуем создать анализатор эмоциональной окраски (или тональности), который будет улавливать эмоции в новостях из разных источников в режиме реального времени. Начнем с импорта необходимых библиотек.

1. Импортируем библиотеки:

```
import tweepy,json,time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from vaderSentiment.vaderSentiment import
SentimentIntensityAnalyzer
analyzer = SentimentIntensityAnalyzer()
```

   Используем следующие две библиотеки:

2. `VADER` *(Valence Aware Dictionary and Sentiment Reasoner)*. Это один из распространенных инструментов анализа тональности на основе правил, разработанный для социальных сетей. Для установки выполним следующее:

```
pip install vaderSentiment
```

3. `Tweepy`, представляющий собой API на языке Python для доступа к Twitter. Выполним команду:

```
pip install Tweepy
```

4. Следующий шаг немного сложнее. Необходимо сделать запрос на создание учетной записи Twitter-разработчик, чтобы получить доступ к потоку твитов. Получив ключи API, можно представить их с помощью следующих переменных:

```
twitter_access_token = <your_twitter_access_token>
twitter_access_token_secret = <your_twitter_access_token_secret>
twitter_consumer_key = <your_consumer_key>
twitter_consumer_secret = <your_twitter_consumer_secret>
```

5. Далее настроим аутентификацию API `Tweepy`. Для этого предоставим ранее созданные переменные:

```
auth = tweepy.OAuthHandler(twitter_consumer_key,
twitter_consumer_secret)
auth.set_access_token(twitter_access_token,
twitter_access_token_secret)
api = tweepy.API(auth, parser=tweepy.parsers.JSONParser())
```

6. Начинается самое интересное. Зададим названия источников новостей, которые будем отслеживать для анализа тональности. Для этого примера выбраны следующие источники:

```
news_sources = ("@BBC", "@ctvnews", "@CNN","@FoxNews", "@dawn_com")
```

7. Теперь напишем основной цикл. Он начнется с пустого массива `array_sentiments`, предназначенного для хранения тональностей. Затем пройдемся по всем пяти источникам новостей и соберем по 100 твитов в каждом. Далее рассчитаем полярность для каждого твита (рис. 11.3).

```
In [12]: # We start extracting 100 tweets from each of the news sources
         print("...STARTING..... collecting tweets from sources")

         # Let us define an array to hold the sentiments
         array_sentiments = []

         for user in news_sources:
             count_tweet=100  # Setting the twitter count at 100
             print("Start tweets from %s"%user)
             for x in range(5):     # Extracting 5 pages of tweets
                 public_tweets=api.user_timeline(user,page=x)
                 # For each tweet
                 for tweet in public_tweets:
                     #Calculating the compound,+ive,-ive and neutral value for each tweet
                     compound = analyzer.polarity_scores(tweet["text"])["compound"]
                     pos = analyzer.polarity_scores(tweet["text"])["pos"]
                     neu = analyzer.polarity_scores(tweet["text"])["neu"]
                     neg = analyzer.polarity_scores(tweet["text"])["neg"]

                     array_sentiments.append({"Media":user,
                                              "Tweet Text":tweet["text"],
                                              "Compound":compound,
                                              "Positive":pos,
                                              "Negative":neg,
                                              "Neutral":neu,
                                              "Date":tweet["created_at"],
                                              "Tweets Ago":count_tweet})

                     count tweet-=1

         print("DONE with extracting tweets")

         ...STARTING..... collecting tweets from sources
         Start tweets from @BBC
         Start tweets from @ctvnews
         Start tweets from @CNN
         Start tweets from @FoxNews
         Start tweets from @dawn_com
         DONE with extracting tweets
```

Рис. 11.3

8. Создадим график, который показывает полярность новостей из этих источников (рис. 11.4).

```
In [21]: for media in source:
             mydf=sentiments_df[sentiments_df["Media"]==media]
             plt.scatter(mydf["Tweets Ago"],mydf["Compound"], marker="o", linewidth=0, alpha=0.8, label=media,
                         facecolors=mydf.Media.map({"BBC": "pink", "ctvnews" : "purple",  "CNN": 'red',
                                                    "FoxNews":"blue","dawn_com":"green"}))

         plt.legend(bbox_to_anchor = (1,1),title="Media Sources")
         plt.title("Sentiment Analysis of Media Tweets (%s)" % (time.strftime("%x")), fontsize=14)
         plt.xlabel("Tweets Ago")
         plt.ylabel("Tweet Polarity")
         plt.xlim(101,0)
         plt.ylim(-1,1)
         plt.grid(True)
         plt.savefig("Output/Sentiment Analysis of Media Tweets.png",bbox_inches='tight')
         plt.show()
```

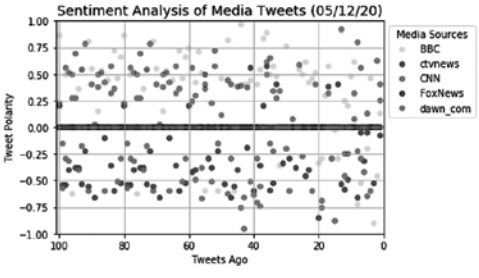


Рис. 11.4

9. Теперь обратимся к сводной статистике (рис. 11.5).

```
In [22]: # Calculating the mean for each Media channel and storing to a dataframe
         means_media_trends=sentiments_df.groupby("Media").mean()["Compound"].to_frame()
         #Resetting the index
         means_media_trends.reset_index(inplace=True)

         means_media_trends
```

Out[22]:

	Media	Compound
0	BBC	0.234137
1	CNN	0.057011
2	FoxNews	0.000000
3	ctvnews	-0.194618
4	dawn_com	-0.025389

Рис. 11.5

Данные цифры суммируют тенденции эмоциональной окраски. Например, тональность новостей BBC выглядит наиболее позитивной, а канадский новостной канал CTVNews, похоже, транслирует самые негативные эмоции.

РЕЗЮМЕ

В этой главе мы рассмотрели разработку алгоритмов, ориентированных на данные, и сосредоточились на трех задачах: хранение, сжатие и потоковая передача.

Мы узнали, как характеристики данных определяют дизайн хранилища данных, и изучили два типа алгоритмов сжатия. Затем мы на практике выяснили, как алгоритмы потоковой передачи данных используются для анализа тональности твитов в режиме реального времени.

В следующей главе мы рассмотрим криптографические алгоритмы и научимся использовать их возможности для защиты сообщений при передаче и хранении.

12

Криптография

В этой главе представлены *криптографические алгоритмы*. Мы начнем с основ, обсудим алгоритмы симметричного шифрования; затем нас ждут алгоритмы хеширования *MD5* и *SHA*. Далее мы познакомимся с ограничениями и уязвимостями симметричных алгоритмов. После обсудим алгоритмы асимметричного шифрования и создание цифровых сертификатов. Наконец, разберем практический пример, в котором обобщаются все эти методы.

К концу главы вы получите базовое представление о различных вопросах, связанных с криптографией.

В главе обсуждаются следующие темы:

- Введение в криптографию.
- Типы криптографических методов.
- Практический пример — проблемы безопасности при внедрении модели МО.

Давайте начнем с основных концепций.

ВВЕДЕНИЕ В КРИПТОГРАФИЮ

Методы защиты секретов существуют уже много веков. Самые ранние попытки обезопасить и скрыть данные от противников восходят к древним

надписям, обнаруженным на памятниках в Египте, где использовался специальный алфавит, известный лишь нескольким доверенным людям. Эта ранняя форма безопасности называется *неясностью* (obscurity) и используется в различных формах по сей день. Чтобы такой метод работал, крайне важно защитить ключ для понимания алфавита. В годы Первой и Второй мировых войн поиск надежных способов шифрования секретных сообщений стал задачей первостепенной важности. В конце XX века, с появлением электроники и компьютеров, были разработаны сложные алгоритмы защиты данных. Так возникла совершенно новая область — *криптография*. В этой главе представлены ее алгоритмические аспекты. Цель алгоритмов шифрования — обеспечить безопасный обмен данными между двумя процессами или пользователями. В криптографических алгоритмах используются математические функции.

Понимание важности самого слабого звена

Иногда при разработке цифровой инфраструктуры уделяется слишком много внимания защите отдельных элементов в ущерб сквозной безопасности. В результате из виду упускаются слабые места и уязвимости в системе, которые впоследствии могут быть использованы хакерами для доступа к конфиденциальным данным. Защищенность цифровой инфраструктуры в целом определяется защищенностью ее *самого слабого звена*. Воспользовавшись слабым звеном, хакер может получить доступ к конфиденциальным данным в обход систем безопасности. Нет смысла укреплять парадную дверь, если открыты двери с черного хода.

По мере того как алгоритмы и методы защиты цифровой инфраструктуры становятся все более и более сложными, злоумышленники также оттачивают свои приемы. Важно помнить, что использование уязвимостей — один из самых простых способов взлома системы для доступа к конфиденциальной информации.

В 2014 году кибератака на канадский Национальный исследовательский совет (NRC), по некоторым оценкам, обошлась в сотни миллионов долларов. Злоумышленникам удалось похитить исследовательские данные и материалы интеллектуальной собственности, собранные за 10 лет. Хакеры использовали лазейку в программном обеспечении Apache, которое было установлено на веб-серверах, и получили доступ к конфиденциальным данным.

В этой главе мы рассмотрим уязвимости различных алгоритмов шифрования.

Начнем с терминологии.

Основная терминология

Ниже представлена базовая терминология, связанная с криптографией.

- *Шифр*. Алгоритм, выполняющий определенную криптографическую функцию.
- *Открытый (исходный) текст*. Незашифрованные данные (текстовый файл, видео, растровое изображение или оцифрованная речь). Обозначим открытый текст как *P* (plain text).
- *Зашифрованный текст*. Это текст, полученный после применения алгоритмов криптографии к открытому тексту. Обозначим его как *C* (cipher text).
- *Набор шифров*. Набор компонентов криптографического ПО. Прежде чем обмениваться сообщениями с использованием криптографии, сначала необходимо согласовать этот набор. Важно убедиться, что используется одна и та же реализация криптографических функций.
- *Шифрование*. Процесс преобразования открытого текста (*P*) в зашифрованный текст (*C*). Математически это можно представить формулой *encrypt(P) = C*.
- *Расшифровка*. Процесс преобразования зашифрованного текста обратно в исходный текст.

  Математически это представляется как *decrypt(C) = P*.
- *Криптоанализ*. Методы, используемые для анализа надежности криптографических алгоритмов. Аналитик пытается восстановить исходный текст без доступа к секретному ключу.
- *Персональные данные* (personally identifiable information, PII). *Информация*, позволяющая определить личность человека (сама по себе или в совокупности с другими данными). Это может быть номер социального страхования, дата рождения или девичья фамилия матери.

Требования безопасности

Прежде всего необходимо точно определить требования безопасности системы. Исходя из них можно выбрать подходящий криптографический метод и обна-

ружить потенциальные лазейки в системе. Чтобы установить требования безопасности, нужно выполнить следующие три шага:

- идентифицировать субъекты;
- определить цели безопасности;
- установить степень конфиденциальности данных.

Рассмотрим эти шаги один за другим.

Идентификация субъектов

Один из способов определить субъекты — ответить на следующие четыре вопроса, чтобы понять потребности системы в контексте безопасности:

- Какие приложения необходимо защитить?
- От кого необходимо защищать приложения?
- Где их нужно защищать?
- Почему они должны быть защищены?

Как только ответы на вопросы получены, можно переходить к определению целей безопасности цифровой системы.

Определение целей безопасности

Криптографические алгоритмы обычно используются для достижения одной или нескольких целей безопасности:

- *Аутентификация*. Говоря простым языком, это проверка того, что пользователь является тем, за кого он себя выдает. В результате аутентификации подтверждается его личность. Сначала пользователь называет себя, а затем предоставляет информацию, которая известна только ему и может быть получена только от него.
- *Конфиденциальность*. Данные, которые необходимо защитить, называются *конфиденциальными*. Конфиденциальность — это концепция ограничения доступа к таким данным: доступ возможен только для авторизованных пользователей. Чтобы обеспечить конфиденциальность информации во время передачи или хранения, необходимо преобразовать данные таким образом, чтобы их могли прочитать только авторизованные пользователи. Для этого применяются алгоритмы шифрования, которые мы обсудим позже в этой главе.

- *Целостность*. Означает, что данные никоим образом не были изменены во время их передачи или хранения. Например, стек протоколов TCP/IP вычисляет контрольную сумму или использует *циклический избыточный код* (cyclic redundancy check, CRC) для проверки целостности данных.
- *Неотказуемость* (non-repudiation). Отправителю информации приходит подтверждение доставки данных, а получателю — подтверждение личности отправителя. Это обеспечивает неопровержимые доказательства того, что сообщение было отправлено или доставлено. Таким образом можно убедиться в получении данных или обнаружить точки сбоев связи.

Чувствительность информации

Важно оценить секретность информации и подумать о том, насколько серьезными будут последствия, если данные окажутся скомпрометированы. Выбрать подходящий криптографический алгоритм поможет классификация данных, основанная на чувствительности содержащейся в них информации. Рассмотрим типичные категории данных.

- *Общедоступные или несекретные данные*. Все, что доступно для публичного использования. Например, информация, найденная на веб-сайте компании или информационном портале правительства.
- *Внутренние или конфиденциальные данные*. Хотя они и не предназначены для общественного пользования, раскрытие таких данных не принесет большого вреда. Например, если будут обнародованы электронные письма сотрудника, жалующегося на своего менеджера, это способно поставить компанию в неловкое положение, но не приведет к разрушительным последствиям.
- *Чувствительные или секретные данные*. Данные, которые не предназначены для публичного использования, а их обнародование будет иметь пагубные последствия для отдельного лица или организации. Например, утечка подробностей о будущем iPhone может нанести ущерб бизнесу Apple и дать преимущество конкурентам, таким как Samsung.
- *Сверхчувствительные или сверхсекретные данные*. Это информация, которая при раскрытии нанесет огромный ущерб организации. Она может включать номера социального страхования клиентов, номера кредитных карт или другую сверхчувствительную информацию. Сверхсекретные данные защищены несколькими уровнями безопасности и требуют специального разрешения на доступ.

Как правило, более сложные системы безопасности работают намного медленнее, чем простые алгоритмы. Важно найти правильный баланс между безопасностью и производительностью системы.

Базовое устройство шифров

Разработка шифра заключается в создании алгоритма, который способен зашифровать данные так, чтобы вредоносная программа или неавторизованный пользователь не смогли получить к ним доступ. Хотя со временем шифры становятся все более и более сложными, их основа остается неизменной.

Начнем с рассмотрения ряда относительно простых шифров, чтобы понять основополагающие принципы, используемые при разработке криптографических алгоритмов.

Шифры подстановки

Различные формы шифров подстановки (substitution ciphers) использовались на протяжении сотен лет. Как следует из названия, они основаны на простой концепции — замене символов в открытом тексте на другие символы заранее определенным способом.

Рассмотрим необходимые для этого шаги.

1. Сопоставить каждый элемент с замещающим символом.
2. Преобразовать открытый текст в зашифрованный, заменяя каждый символ согласно таблице подстановок.
3. Для декодирования восстановить открытый текст с помощью таблицы подстановок.

Рассмотрим два примера.

- *Шифр Цезаря*

  В шифре Цезаря таблица подстановок создается путем замены каждого символа третьим символом справа от него. Этот процесс представлен на следующей диаграмме (рис. 12.1).

  Реализуем шифр Цезаря с помощью Python:

```
import string
rotation = 3
P = 'CALM'; C=''
for letter in P:
    C = C+ (chr(ord(letter) + rotation))
```

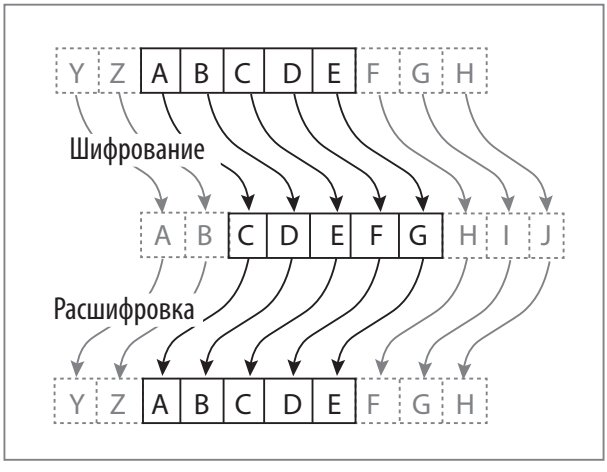


Рис. 12.1

Шифр применен к открытому тексту CALM.

Выведем зашифрованный текст после преобразования с помощью шифра Цезаря (рис. 12.2).

```
In [37]: print(C)
         FDOP
```

Рис. 12.2

Считается, что шифр Цезаря использовался Юлием Цезарем для связи со своими советниками.

- *Rotation 13 (ROT13)*

  ROT13 — это еще одно шифрование на основе подстановки. В ROT13 таблица подстановок создается путем замены каждого символа 13-м символом справа от него. Это показано на следующей схеме (рис. 12.3).

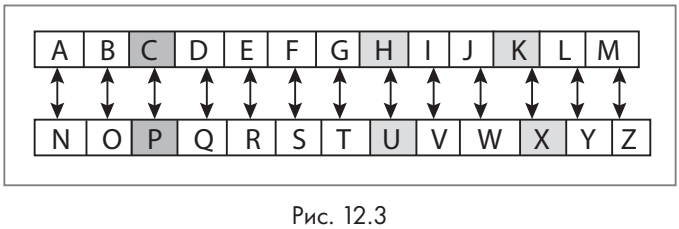


Рис. 12.3

Если `ROT13()` является функцией, реализующей ROT13, то можно записать это так:

```
import codecs
P = 'CALM'
C=''
C=codecs.encode(P, 'rot_13')
```

Теперь выведем закодированное значение `C` (рис. 12.4).

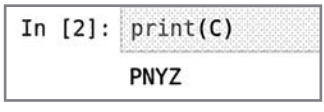


Рис. 12.4

Криптоанализ шифров подстановки

Шифры подстановки просты в реализации и понимании. К сожалению, их столь же легко взломать. Простой криптоанализ показывает, что если в шифре используется английский (или любой другой известный) алфавит, то все, что нужно знать для взлома, — это величина сдвига. Можно проверять каждую букву алфавита одну за другой, пока текст не будет расшифрован. Это значит, что для восстановления открытого текста потребуется около 25 попыток (для английского языка).

Теперь рассмотрим другой метод простого шифрования — перестановочные шифры.

Перестановочные шифры

В *перестановочных шифрах* (transposition ciphers) символы открытого исходного текста меняются местами. Для этого необходимо проделать следующие шаги.

1. Создать матрицу перестановок, выбрав ее размер. Она должна быть достаточно большой, чтобы вместить строку открытого текста.
2. Заполнить матрицу, записав все символы строки по горизонтали.
3. Прочитать в матрице все символы строки по вертикали.

Рассмотрим пример.

Давайте возьмем открытый текст `Ottawa Rocks` (*P*).

Закодируем *P*. Для этого используем матрицу 3 × 4 и запишем символы открытого текста по горизонтали (табл. 12.1).

Таблица 12.1

O	t	t	a
w	a	R	o
c	k	s	

Процесс чтения будет считывать символы по вертикали, что приведет к созданию зашифрованного текста — `OwctaktRsao`.

Во время Первой мировой войны немцы использовали шифр под названием ADFGVX, в котором использовались как шифры перестановки, так и подстановки. Позже он был взломан Жоржем Пенвеном.

Это лишь некоторые методы шифрования. Теперь рассмотрим ряд криптографических методов, которые используются в настоящее время.

ТИПЫ КРИПТОГРАФИЧЕСКИХ МЕТОДОВ

Разные типы криптографических методов используют различные алгоритмы и применяются в различных обстоятельствах.

В широком смысле криптографические методы можно разделить на следующие три типа:

- Хеширование.
- Симметричные методы.
- Асимметричные методы.

Рассмотрим их по очереди.

Криптографические хеш-функции

Криптографическая хеш-функция — это математический алгоритм, который может использоваться для создания уникального цифрового отпечатка сообще-

ния. Открытый текст преобразуется в вывод фиксированного размера, называемый *хешем*.

Математически это выглядит следующим образом:

$$C_1 = hashFunction(P_1)$$

В формуле:

- P_1 — открытый текст, представляющий входные данные;
- C_1 — хеш фиксированной длины, который генерируется криптографической хеш-функцией.

Процесс показан на диаграмме (рис. 12.5). Данные переменной длины преобразуются в хеш фиксированной длины с помощью односторонней хеш-функции.

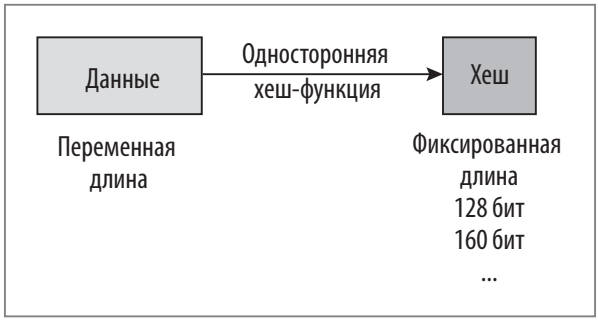


Рис. 12.5

Хеш-функция обладает следующими пятью характеристиками:

- Она детерминирована. Один и тот же открытый текст генерирует одинаковый хеш.
- Уникальные входные строки генерируют уникальные выходные хеш-значения.
- Независимо от входного сообщения хеш-функция имеет фиксированную длину.
- Даже небольшие изменения в открытом тексте генерируют новый хеш.
- Это односторонняя функция, то есть открытый текст P_1 не может быть сгенерирован из зашифрованного текста C_1.

Ситуация, когда не у каждого уникального сообщения есть уникальный хеш, называется *коллизией*. Иными словами, если при хешировании двух текстов, P_1 и P_2, возникнет коллизия, это означает, *что* $hashFunction(P_1) = hashFunction(P_2)$.

Независимо от используемого алгоритма хеширования коллизии происходят достаточно редко. В противном случае хеширование было бы бесполезно. Однако для некоторых приложений конфликты недопустимы. В таких случаях следует использовать более сложный алгоритм хеширования с гораздо меньшей вероятностью коллизии генерируемых хеш-значений.

Реализация криптографических хеш-функций

Криптографические хеш-функции могут быть реализованы с использованием различных алгоритмов. Остановимся на двух из них.

Алгоритм MD5

Алгоритм дайджеста сообщений MD5 (Message-Digest 5) был разработан Рональдом Л. Ривестом в 1991 году для замены MD4. Он генерирует 128-битный хеш. MD5 — это относительно простой алгоритм, который подвержен коллизиям. В приложениях, где коллизии недопустимы, MD5 использовать не рекомендуется.

Рассмотрим пример. Чтобы сгенерировать хеш MD5 на Python, используем библиотеку `passlib`. Это одна из самых популярных библиотек с открытым исходным кодом, реализующая более 30 алгоритмов хеширования паролей. Установите ее, используя следующий код в ноутбуке Jupyter:

```
!pip install passlib
```

Сгенерируем хеш MD5 в Python (рис. 12.6).

```
In [36]: myHash
Out[36]: '$1$a6sQqHlF$j5iHhbCzmOzVwrxWDxnUu.'
```

Рис. 12.6

Как уже упоминалось, MD5 генерирует хеш в 128 бит.

Сгенерированный хеш можно использовать в качестве отпечатка исходного текста, в нашем случае это текст `myPassword`. Вот как это делается на Python (рис. 12.7).

```
In [37]: md5_crypt.verify("myPassword", myHash)
Out[37]: True

In [38]: md5_crypt.verify("myPassword2", myHash)
Out[38]: False
```

Рис. 12.7

Обратите внимание, что полученный хеш для строки `myPassword` соответствует исходному хешу, который вернул значение `True`. Однако он вернул значение `False`, как только открытый текст был изменен на `myPassword2`.

Перейдем к другому алгоритму хеширования — *SHA*.

Алгоритм SHA

Алгоритм безопасного хеширования SHA (Secure Hash Algorithm) был разработан *Национальным институтом стандартов и технологий (NIST)*. Давайте посмотрим, как используется SHA на Python для создания хеша:

```
from passlib.hash import sha512_crypt
sha512_crypt.using(salt = "qIo0foX5",rounds=5000).hash("myPassword")
```

Обратите внимание на использование параметра `salt (соль)`. «Соление» — это процедура добавления случайных символов перед хешированием.

Выполнение кода дает следующий результат (рис. 12.8).

```
In [13]: myHash
Out[13]: '$6$qIo0foX5$a.RA/OyedLnLEnWovzqngCqhyy3EfqRtwacvWKsIoYSvYgRxCRetM3XSwrgMxwdPqZt4KfbXzCp6y
         NyxI5j6o/'
```

Рис. 12.8

Важно отметить, что при использовании алгоритма SHA генерируемый хеш составляет 512 байт.

Применение криптографических хеш-функций

Хеш-функции используются для проверки целостности файла после его копирования. Когда файл загружается из источника в место назначения (например, при скачивании с веб-сервера на локальный компьютер), вместе с ним копируется его хеш. Этот оригинальный хеш, $h_{original}$, является отпечатком исходного

файла. После копирования файла мы генерируем хеш копии файла — h_{copied}. Если $h_{original} = h_{copied}$, то есть сгенерированный хеш совпадает с исходным, это подтверждает, что файл не изменился и никакие данные не были утеряны в процессе загрузки. Для создания хеша в этих целях можно использовать любую криптографическую хеш-функцию, например MD5 или SHA.

Теперь рассмотрим симметричное шифрование.

Симметричное шифрование

В криптографии ключ — это комбинация чисел, используемая для кодирования открытого текста с использованием выбранного алгоритма. При *симметричном шифровании* для кодирования и декодирования применяется один и тот же ключ. Если ключ, используемый для симметричного шифрования, равен K, то выполняется следующее уравнение:

$$E_K(P) = C.$$

Здесь P — это открытый текст, а C — зашифрованный.

Для расшифровки мы используем тот же ключ, K, преобразуя его обратно в P:

$$D_K(C) = P.$$

Данный процесс показан на диаграмме (рис. 12.9).

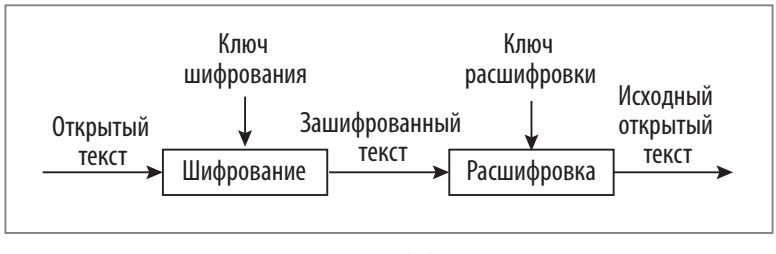


Рис. 12.9

Теперь посмотрим, как симметричное шифрование реализуется на Python.

Реализация симметричного шифрования

В этом разделе для демонстрации симметричного шифрования используется библиотека Python под названием `cryptography`. Это комплексная библиотека,

реализующая множество криптографических алгоритмов, таких как симметричные шифры и различные *алгоритмы дайджеста сообщений* (message digests). Установим ее с помощью команды `pip`:

```
!pip install cryptography
```

После установки используем библиотеку для реализации симметричного шифрования.

1. Прежде всего импортируем нужные библиотеки:

```
import cryptography as crypt
from cryptography.fernet import Fernet
```

2. Далее сгенерируем ключ (рис. 12.10).

```
In [29]: key = Fernet.generate_key()
         print(key)

         b'NbzXiNqKR25SEv_O8EpuW2Lr_QO2vDStTDV4ex4WA5U='
```

Рис. 12.10

3. Теперь давайте откроем ключ:

```
file = open('mykey.key', 'wb')
file.write(key)
file.close()
```

4. Используя ключ, зашифруем сообщение:

```
file = open('mykey.key', 'rb')
key = file.read()
file.close()
```

5. Теперь расшифруем сообщение с помощью этого же ключа:

```
from cryptography.fernet import Fernet
message = "Ottawa is really cold".encode()

f = Fernet(key)
encrypted = f.encrypt(message)
```

6. Расшифруем сообщение и присвоим его переменной с именем `decrypt`:

```
decrypted = f.decrypt(encrypted)
```

7. Выведем `decrypt`, чтобы проверить, удастся ли получить одно и то же сообщение (рис. 12.11).

```
In [46]: print(decrypted)
         b'Ottawa is really cold'
```

Рис. 12.11

Рассмотрим преимущества и недостатки симметричного шифрования.

Преимущества симметричного шифрования

Хотя производительность симметричного шифрования зависит от конкретного алгоритма, в целом оно намного быстрее, чем асимметричное (которое мы рассмотрим ниже).

Недостатки симметричного шифрования

Когда два пользователя или процесса планируют использовать для связи симметричное шифрование, им необходимо обменяться ключами через защищенный канал. Это ведет к следующим проблемам:

- *Защита ключа*. Как защитить симметричный ключ шифрования?
- *Распространение ключа*. Как передать ключ симметричного шифрования от источника к месту назначения?

Перейдем теперь к асимметричному шифрованию.

Асимметричное шифрование

В 1970-х годах для устранения упомянутых в предыдущем разделе недостатков симметричного шифрования было разработано асимметричное шифрование.

Первым шагом в асимметричном шифровании является создание двух ключей, которые выглядят совершенно по-разному, но алгоритмически связаны. Один из них выбирается в качестве *закрытого ключа* (private key) K_{pr}, а другой — в качестве *открытого ключа* (public key), K_{pu}. Математически это можно представить так:

$$E_{Kpr}(P) = C.$$

Здесь P — это открытый текст, а C — зашифрованный.

Можно расшифровать его следующим образом:

$$D_{Kpu}(C) = P.$$

Открытые ключи могут свободно распространяться, а закрытые — хранятся владельцем пары ключей в секрете.

Фундаментальный принцип заключается в том, что если данные закодированы с помощью первого, открытого ключа, единственный способ расшифровать их — использовать второй, закрытый ключ. Рассмотрим один из основных протоколов асимметричного шифрования — *SSL/TLS*. Это протокол подтверждения связи, или *протокол рукопожатия* (handshake protocol), обеспечивающий установление соединения между двумя нодами с использованием асимметричного шифрования.

Протокол SSL/TLS

SSL (Secure Sockets Layer — уровень защищенных сокетов) изначально был разработан для повышения безопасности HTTP. Со временем SSL был заменен более эффективным и надежным протоколом *TLS* (Transport Layer Security — безопасность транспортного уровня). Безопасный сеанс связи HTTP базируется на процедуре установки *TLS-рукопожатия*. TLS-соединение происходит между двумя участвующими объектами — *клиентом* и *сервером*. Данный процесс показан на диаграмме (рис. 12.12).

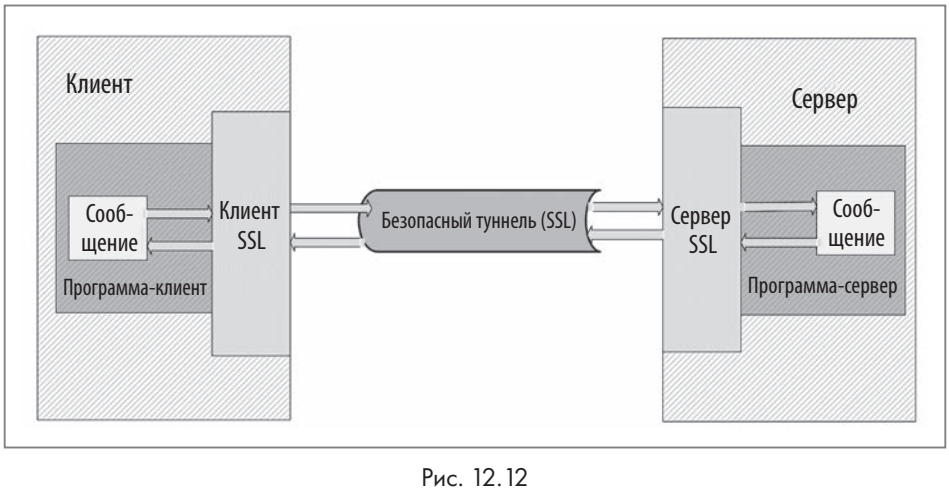


Рис. 12.12

TLS-соединение устанавливает безопасную связь между участвующими нодами. Ниже приведены этапы данного процесса.

1. Клиент отправляет серверу сообщение `client hello`. В сообщении также содержится:
   - Версия используемого TLS.
   - Список наборов шифров, поддерживаемых клиентом.
   - Алгоритм сжатия.
   - Случайная строка байтов, `byte_client`.
2. Сервер в ответ отправляет клиенту приветственное сообщение `server hello`. В сообщении также содержится следующее:
   - Набор шифров, выбранный сервером из списка, предоставленного клиентом.
   - ID сеанса.
   - Случайная строка байтов, `byte_server`.
   - Цифровой сертификат сервера, `cert_server`, содержащий открытый ключ сервера.
   - Если сервер требует цифровой сертификат для аутентификации клиента, его запрос включает также:
     - Уникальные названия допустимых центров сертификации.
     - Типы поддерживаемых сертификатов.
3. Клиент проверяет `cert_server`.
4. Клиент генерирует случайную байтовую строку `byte_client2` и шифрует ее с помощью открытого ключа сервера, предоставленного через `cert_server`.
5. Клиент генерирует случайную строку байтов и осуществляет шифрование с помощью собственного закрытого ключа.
6. Сервер проверяет сертификат клиента.
7. Клиент отправляет на сервер сообщение `finished`, зашифрованное секретным ключом.
8. Чтобы подтвердить получение, сервер отправляет клиенту сообщение `finished`, также зашифрованное секретным ключом.
9. Сервер и клиент установили безопасный канал связи. Теперь они могут обмениваться сообщениями, которые симметрично зашифрованы с помощью общего секретного ключа. Вся методология выглядит следующим образом (рис. 12.13).

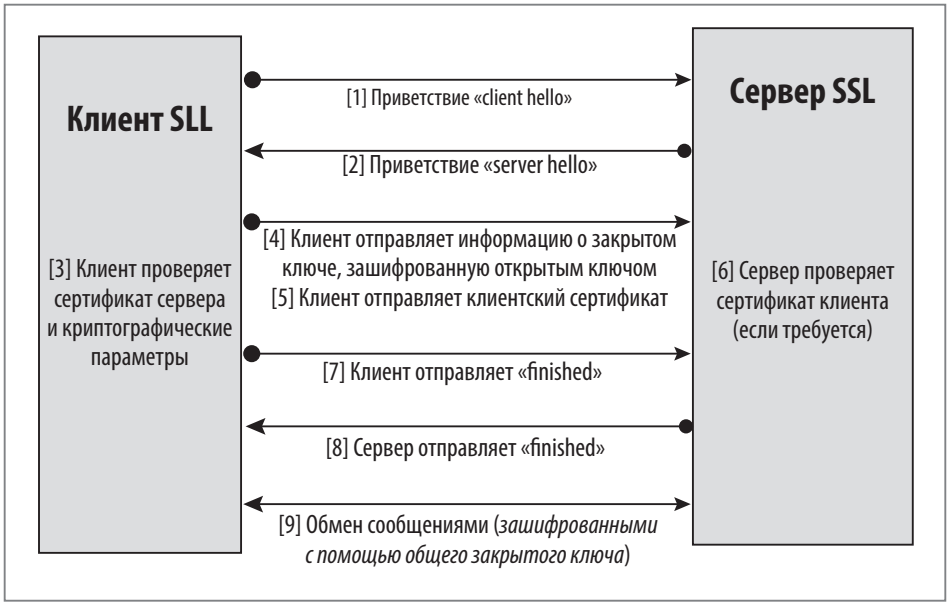


Рис. 12.13

Теперь обсудим, как используется асимметричное шифрование для создания *инфраструктуры открытых ключей* (public key infrastructure, PKI), предназначенной для достижения одной или нескольких целей безопасности организации.

Инфраструктура открытых ключей (PKI)

Для реализации *инфраструктуры открытых ключей (PKI)* используется асимметричное шифрование. PKI является одним из самых популярных и надежных способов управления ключами шифрования в рамках организации. Все участники доверяют надежному органу — *центру сертификации* (certification authority, CA). Центр сертификации идентифицирует отдельное лицо или организацию, а затем выдает им цифровые сертификаты. Сертификат содержит копию открытого ключа лица (или организации) и его идентификационные данные. Тем самым центр подтверждает, что публичный ключ на самом деле принадлежит лицу или организации.

Центр сертификации просит пользователя подтвердить свою личность, при этом для отдельных лиц и организаций применяются разные стандарты. Это может быть простая проверка принадлежности доменного имени или более тщательный процесс, включающий подтверждение личности (в зависимости от типа циф-

рового сертификата, который пытается получить пользователь). Если подлинность пользователя подтверждается, он передает центру сертификации открытый ключ по защищенному каналу. С учетом этой информации создается цифровой сертификат с цифровой подписью центра. После этого пользователь может предъявить свой сертификат любому, кто хочет удостоверить его личность. При этом нет необходимости отправлять сертификат по защищенному каналу, поскольку он не содержит никакой конфиденциальной информации. Лицо, получающее сертификат, не обязано напрямую проверять личность пользователя. Этот человек может просто убедиться в том, что сертификат действителен, проверив цифровую подпись центра сертификации. Подпись подтверждает, что открытый ключ, содержащийся в сертификате, действительно принадлежит лицу (или организации), указанному в нем.

Закрытый ключ центра сертификации организации является самым слабым звеном в цепочке доверия PKI. Например, если имитатор завладеет закрытым ключом Microsoft, он сможет установить вредоносное ПО на миллионы компьютеров по всему миру, выдавая себя за центр обновления Windows.

ПРАКТИЧЕСКИЙ ПРИМЕР — ПРОБЛЕМЫ БЕЗОПАСНОСТИ ПРИ РАЗВЕРТЫВАНИИ МОДЕЛИ МО

В главе 6 мы рассмотрели *жизненный цикл CRISP-DM*, который представляет собой этапы обучения и развертывания модели МО. Как только модель обучена и оценена, завершающим этапом становится ее развертывание. Если это особо важная модель, необходимо убедиться, что все цели безопасности достигнуты.

Проанализируем типичные проблемы, с которыми приходится сталкиваться при развертывании подобной модели, и способы их решения с помощью концепций, рассмотренных в этой главе. Обсудим стратегии защиты обученной модели от следующих трех угроз:

- Атака посредника.
- Маскарадинг (нелегальное проникновение).
- Искажение данных.

Рассмотрим их по очереди.

Атака посредника (MITM)

Одна из возможных атак, от которой необходимо защитить модель, — это *атака посредника*, или *атака MITM* (Man-in-the-Middle — «человек посередине»). Атака MITM происходит, когда злоумышленник пытается перехватить не предназначенное для него сообщение при внедрении обученной модели МО.

Поэтапно разберем атаку MITM, обратившись к примеру возможного сценария.

Предположим, Боб и Алиса собираются обменяться сообщениями с помощью PKI.

1. Боб использует $\{Pr_{Боб}, Pu_{Боб}\}$, а Алиса — $\{Pr_{Алисас}, Pu_{Алиса}\}$. Боб создал сообщение $M_{Боб}$, и Алиса создала сообщение $M_{Алиса}$. Они хотят безопасно обменяться этими сообщениями друг с другом.
2. Сначала им необходимо обменяться открытыми ключами, чтобы установить друг с другом безопасное соединение. Это означает, что Боб использует $Pu_{Алиса}$ для шифрования $M_{Боб}$ перед отправкой сообщения Алисе.
3. Предположим, что существует злоумышленник X, который использует $\{PR_X, Pu_X\}$. Он может перехватить обмен открытыми ключами между Бобом и Алисой и заменить их собственным открытым сертификатом.
4. Боб отправляет $M_{Боб}$ Алисе, шифруя его с помощью Pu_X вместо $Pu_{Алиса}$, ошибочно полагая, что это публичный сертификат Алисы. Подслушивающий X перехватывает сообщение $M_{Боб}$ и расшифровывает его с помощью $Pr_{Боб}$.

Эта атака MITM продемонстрирована на следующей диаграмме (рис. 12.14).

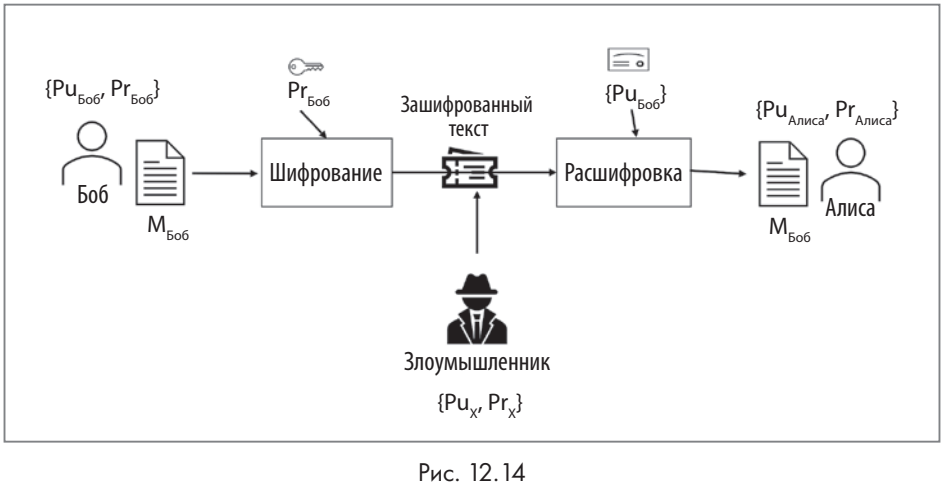


Рис. 12.14

Давайте узнаем, как можно предотвратить атаки MITM.

Предотвращение атаки MITM

Предотвратить атаки MITM возможно путем добавления в систему центра сертификации. Допустим, название этого центра — myTrustCA. Цифровой сертификат имеет свой открытый ключ, $Pu_{myTrustCA}$, встроенный в него. myTrustCA отвечает за подписание сертификатов для всех участников системы, включая Алису и Боба. Это означает, что и Боб, и Алиса имеют сертификаты, подписанные myTrustCA. Подписывая их сертификаты, myTrustCA проверяет, что они действительно являются теми, за кого себя выдают.

После введения этого условия вернемся к взаимодействию между Бобом и Алисой.

1. Боб использует $\{PR_{Боб}, Pu_{Боб}\}$, а Алиса использует $\{PR_{Алиса}, Pu_{Алиса}\}$. Оба открытых ключа встроены в их цифровые сертификаты, подписанные myTrustCA. Боб создал сообщение $M_{Боб}$, а Алиса создала сообщение $M_{Алиса}$. Они хотят безопасно обменяться этими сообщениями друг с другом.
2. Боб и Алиса обмениваются цифровыми сертификатами, содержащими их открытые ключи. Они примут ключи только в том случае, если те встроены в сертификаты, подписанные myTrustCA. Они должны обменяться открытыми ключами, чтобы установить безопасное соединение друг с другом. Таким образом, Боб применит $Pu_{Алиса}$ для шифрования $M_{Боб}$, прежде чем отправить сообщение Алисы.
3. Появляется злоумышленник X, который использует $\{PR_X, Pu_X\}$. Злоумышленник может перехватить открытые ключи Боба и Алисы и заменить их собственным открытым сертификатом Pu_X.
4. Боб отклоняет попытку X, так как цифровой сертификат злоумышленника не подписан myTrustCA. Безопасное соединение прерывается, попытка атаки регистрируется с отметкой времени и всеми деталями, и возникает исключение безопасности.

При внедрении обученной модели МО вместо Алисы используется сервер развертывания. Боб развертывает модель только после установления защищенного канала, выполнив все упомянутые шаги.

Реализуем это на Python.

Сначала импортируем необходимые библиотеки.

```
from xmlrpc.client import SafeTransport, ServerProxy import ssl
```

Теперь создадим класс, который может проверить сертификат.

```
class CertVerify(SafeTransport):
    def __init__(self, cafile, certfile=None, keyfile=None):
    SafeTransport.__init__(self)
    self._ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
    self._ssl_context.load_verify_locations(cafile)
    if cert:
        self._ssl_context.load_cert_chain(certfile, keyfile)
    self._ssl_context.verify_mode = ssl.CERT_REQUIRED

def make_connection(self, host):
    s = super().make_connection((host, {'context': self._ssl_context}))
    return s

# Создаем клиентский прокси-сервер
s = ServerProxy('https://cloudanum.com:15000',
transport=VerifyCertSafeTransport('server_cert.pem'), allow_none=True)
```

Рассмотрим другие угрозы, с которыми может столкнуться развернутая модель.

Избежание маскарадинга

Злоумышленник *X* притворяется авторизованным пользователем, Бобом, и получает доступ к конфиденциальным данным (в нашем случае это обученная модель). Необходимо защитить модель от любых несанкционированных изменений.

Один из способов защиты от маскарадинга — шифрование модели с помощью закрытого ключа авторизованного пользователя. После этого любой может прочитать и использовать модель, расшифровав ее с помощью открытого ключа авторизованного пользователя, который содержится в его цифровом сертификате. Никто не может вносить какие-либо несанкционированные изменения в модель.

Шифрование данных и моделей

После развертывания модели неразмеченные данные реального времени (предоставленные в качестве входных) также могут быть изменены. Обученная модель используется для вывода и размечает эти данные. Чтобы обезопасить данные от несанкционированного доступа, необходимо защищать их как в состоянии покоя, так и при передаче. В первом случае для кодирования можно использовать симметричное шифрование. Для передачи данных можно установить защищенные каналы на основе SSL/TLS, обеспечивающие безопасный туннель. Он ис-

пользуется для передачи симметричного ключа. Данные расшифровываются на сервере до того, как будут предоставлены обученной модели.

Это один из наиболее эффективных и надежных способов защиты данных от несанкционированного доступа.

Симметричное шифрование также используется для кодирования модели после обучения (перед развертыванием на сервере). Это предотвратит любой несанкционированный доступ к модели до ее внедрения.

Рассмотрим пошагово кодирование обученной модели в источнике, используя симметричное шифрование, а затем декодируем ее в месте назначения перед использованием.

1. Сначала обучим простую модель, используя набор данных Iris:

```
import cryptography as crypt
from sklearn.linear_model
import LogisticRegression
from cryptography.fernet
import Fernet from sklearn.model_selection
import train_test_split
from sklearn.datasets import load_iris
iris = load_iris()

X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y)
model = LogisticRegression()
model.fit(X_train, y_train)
```

2. Теперь определим имена файлов, в которых будет храниться модель:

```
filename_source = 'myModel_source.sav'
filename_destination = "myModel_destination.sav"
filename_sec = "myModel_sec.sav"
```

   Обратите внимание, что `filename_source` — это файл, в котором хранится обученная незашифрованная модель в источнике. `filename_destination` — файл, в котором находится обученная незашифрованная модель в месте назначения, а `filename_sec` — это зашифрованная обученная модель.

3. Используем `pickle` для помещения обученной модели в файл:

```
from pickle import dump
dump(model, open(filename_source, 'wb'))
```

4. Определим функцию с именем `write_key()`, которая генерирует симметричный ключ и хранит его в файле с именем `key.key`:

```
def write_key():
     key = Fernet.generate_key()
     with open("key.key", "wb") as key_file:
        key_file.write(key)
```

5. Теперь определим функцию с именем `load_key()`; она считывает сохраненный ключ из файла `key.key`:

```
def load_key():
    return open("key.key", "rb").read()
```

6. Определим функцию `encrypt()`, которая шифрует и обучает модель, а также сохраняет ее в файле с именем `filename_sec`:

```
def encrypt(filename, key):
     f = Fernet(key)
     with open(filename_source,"rb") as file:
        file_data = file.read()
     encrypted_data = f.encrypt(file_data)
     with open(filename_sec,"wb") as file:
        file.write(encrypted_data)
```

7. Применим эти функции для генерации симметричного ключа и сохранения его в файле. Затем прочитаем этот ключ и используем его для хранения обученной модели в файле с именем `filename_sec`:

```
write_key()
encrypt(filename_source,load_key())
```

Модель зашифрована. Она будет передана в место назначения, где будет использоваться для прогнозирования. Для этого проделаем следующие шаги.

8. Определим функцию с именем `decrypt()` для расшифровки модели из `filename_sec` в `filename_destination` с использованием ключа, хранящегося в файле `key.key`:

```
def decrypt(filename, key):
     f = Fernet(key)
     with open(filename_sec, "rb") as file:
        encrypted_data = file.read()
     decrypted_data = f.decrypt(encrypted_data)
     with open(filename_destination, "wb") as file:
file.write(decrypted_data)
```

9. Теперь используем эту функцию для расшифровки модели и сохраним ее в файле с именем `filename_destination`:

```
decrypt(filename_sec,load_key())
```

10. Далее применим этот незашифрованный файл, чтобы загрузить модель и использовать ее для прогнозирования (рис. 12.15).

```
In [21]: loaded_model = pickle.load(open(filename_destination, 'rb'))
         result = loaded_model.score(X_test, y_test)
         print(result)

         0.9473684210526315
```

Рис. 12.15

Обратите внимание, что для кодирования модели использовалось симметричное шифрование. Тот же метод может быть применен и для шифрования данных.

РЕЗЮМЕ

В этой главе мы узнали о криптографических алгоритмах. Мы начали с определения целей безопасности, затем обсудили различные криптографические методы, а также подробно рассмотрели инфраструктуру PKI. Наконец, мы разобрали различные способы защиты обученной модели МО от распространенных атак. Теперь мы знакомы с основами алгоритмов безопасности, используемых для защиты современных ИТ-инфраструктур.

Следующая глава посвящена разработке крупномасштабных алгоритмов. Мы рассмотрим сопутствующие проблемы и компромиссы, а также изучим использование GPU и кластеров для решения сложных задач.

13

Крупномасштабные алгоритмы

Крупномасштабные алгоритмы, или *алгоритмы решения задач большой размерности* (large-scale algorithms), предназначены для решения невероятно сложных задач. Они нуждаются в нескольких механизмах выполнения ввиду крупных объемов данных и серьезных требований к обработке. В этой главе мы рассмотрим типы алгоритмов, требующих параллельного выполнения, и обсудим распараллеливание. Далее мы познакомимся с архитектурой *CUDA* и выясним, как ускорять алгоритмы с помощью одного или нескольких графических процессоров (GPU). Мы научимся изменять алгоритм таким образом, чтобы эффективно использовать мощность *GPU*. Наконец, коснемся кластерных вычислений. Мы узнаем, как наборы данных *RDD* в Apache Spark используются для чрезвычайно быстрой параллельной реализации стандартных алгоритмов.

Прочтя эту главу, вы усвоите основные стратегии разработки крупномасштабных алгоритмов.

Итак, в главе представлены следующие темы:

- Введение в крупномасштабные алгоритмы.
- Разработка параллельных алгоритмов.
- Алгоритмы для использования *GPU*.
- Алгоритмы, использующие кластерные вычисления.
- Применение *GPU* для запуска крупномасштабных алгоритмов.

- Использование возможностей кластеров для запуска крупномасштабных алгоритмов.

Начнем с основ.

ВВЕДЕНИЕ В КРУПНОМАСШТАБНЫЕ АЛГОРИТМЫ

Людям нравится преодолевать трудности. На протяжении веков мы придумываем инновационные способы решения сложных проблем. От предсказания района нашествия саранчи и до вычисления наибольшего простого числа, методы поиска ответов на трудные вопросы продолжают развиваться. С появлением компьютеров мы получили новый мощный способ решения сложных задач.

Определение эффективного крупномасштабного алгоритма

Хорошо продуманный крупномасштабный алгоритм обладает следующими двумя характеристиками:

- Он справляется с гигантским объемом данных и обширными требованиями к обработке, наилучшим образом используя доступные ресурсы.
- Он масштабируется. По мере усложнения проблемы алгоритм просто задействует больше ресурсов.

Наиболее практичный способ реализации крупномасштабных алгоритмов — стратегия «разделяй и властвуй». Это разбиение крупной задачи на более мелкие, которые решаются независимо друг от друга.

Терминология

Рассмотрим некоторые термины, используемые для количественной оценки качества крупномасштабных алгоритмов.

Задержка

Задержка (latency) — это общее время, необходимое для выполнения одного вычисления. Допустим, C_1 — это одно вычисление, которое начинается в t_1 и заканчивается в t_2, тогда можно сказать следующее:

$$Latency = t_2 - t_1.$$

Пропускная способность

В контексте параллельных вычислений *пропускная способность* (throughput) — это количество отдельных вычислений, которые могут выполняться одновременно. Например, если при t_1 можно выполнить четыре одновременных вычисления, C_1, C_2, C_3 и C_4, то пропускная способность равна четырем.

Полоса бисекции сети

Полоса пропускания между двумя равными частями сети называется *полосой бисекции сети* (network bisection bandwidth). Это самый важный параметр, влияющий на эффективность распределенных вычислений. При недостаточной полосе бисекции скорость соединения будет медленной. Таким образом, будет потеряно преимущество, полученное благодаря наличию нескольких механизмов выполнения.

Эластичность

Способность инфраструктуры среагировать на внезапное увеличение требований к обработке и выделить большее количество ресурсов называется *эластичностью*.

Три гиганта облачных вычислений, Google, Amazon и Microsoft, способны обеспечить высокоэластичную инфраструктуру. Их общий пул ресурсов огромен, и существует очень мало компаний, способных добиться такой же эластичности.

Если инфраструктура эластична, она способна создать масштабируемое решение для задачи.

РАЗРАБОТКА ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ

Важно отметить, что параллельные алгоритмы не являются панацеей. Даже лучшие параллельные архитектуры могут не обеспечить ожидаемой производительности. Рассмотрим закон, который широко применяется для разработки параллельных алгоритмов, — закон Амдала.

Закон Амдала

Джин Амдал был одним из первооткрывателей параллельной обработки в 1960-х годах. Он предложил закон, который актуален до сих пор. Закон Ам-

дала может служить основой для понимания различных компромиссов, связанных с разработкой решений для параллельных вычислений.

Согласно закону Амдала, не все части вычислительного процесса могут выполняться параллельно. Всегда будет последовательная часть процесса, которая не может быть распараллелена.

Рассмотрим конкретный пример. Предположим, необходимо прочитать большое количество файлов, хранящихся на компьютере, и обучить модель МО, используя полученные данные.

Назовем этот процесс *P*. Очевидно, что *P* можно разделить на два подпроцесса:

- *P*1: просканировать файлы в каталоге, создать список имен файлов, соответствующих входному файлу, и передать список дальше.
- *P*2: прочитать файлы, создать пайплайн обработки данных, обработать файлы и обучить модель.

Анализ последовательного процесса

Время выполнения *P* представлено как $T_{seq}(P)$. Время выполнения *P*1 и *P*2 представлено как $T_{seq}(P1)$ и $T_{seq}(P2)$. Очевидно, что при работе на одной ноде мы будем наблюдать следующее:

- *P*2 не может начать работу до завершения *P*1. Это представляется как *P*1 -->*P2*.
- $T_{seq}(P) = T_{seq}(P1) + T_{seq}(P2)$.

Предположим, что запуск *P* на одной ноде в целом занимает 11 секунд . Из них выполнение *P1* занимает 2 секунды, а выполнение *P*2 — 9 секунд. Работа алгоритма показана на следующей схеме (рис. 13.1).

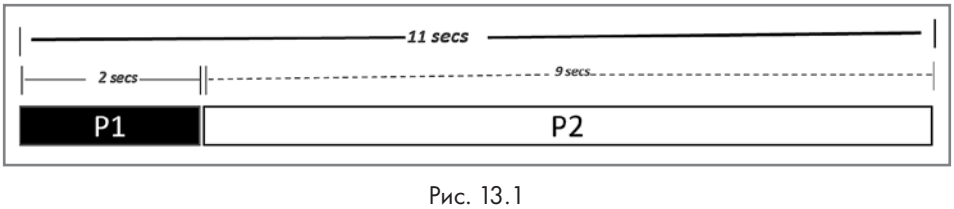


Рис. 13.1

Важно отметить, что *P*1 является последовательным по своей природе. Этот процесс невозможно ускорить, сделав его параллельным. Вместе с тем *P*2 легко

делится на подзадачи, которые могут выполняться одновременно. Их параллельный запуск ускоряет работу процесса.

Основным преимуществом облачных вычислений является наличие большого пула ресурсов, и многие из них используются параллельно. План применения этих ресурсов для конкретной задачи называется планом выполнения. Закон Амдала используется для тщательного выявления ограничений задачи и пула ресурсов.

Анализ параллельного выполнения

Если используется более одной ноды для ускорения *P*, это повлияет на *P*2 только с коэффициентом $s > 1$:

$$T_{par}(P) = T_{seq}(P1) + \frac{1}{s}T_{seq}(P2).$$

Ускорение процесса *P* можно легко рассчитать следующим образом:

$$S(P) = \frac{T_{seq}(P)}{T_{par}(P)}.$$

Отношение распараллеливаемой части процесса к его общему количеству представлено *b* и рассчитывается так:

$$b = \frac{T_{seq}(P2)}{T_{seq}(P)}.$$

Например, в предыдущем сценарии *b = 9/11 = 0.8182*.

Упрощение этих уравнений даст нам закон Амдала:

$$S(P) = \frac{1}{1 - b + \frac{b}{s}}.$$

Итак, мы имеем следующее:

- *P* — это общий процесс;
- *b* — отношение распараллеливаемой части *P;*
- *s* — ускорение, достигнутое в распараллеливаемой части *P*.

Предположим, процесс *P* запускается на трех параллельных нодах:

- *P*1 является последовательным и не может быть сокращен с помощью параллельных нод. Он по-прежнему длится 2 секунды.
- *P*2 теперь занимает 3 секунды вместо 9.

Таким образом, общее время, затрачиваемое процессом *P*, сокращается до 5 секунд, как показано на следующей диаграмме (рис. 13.2).

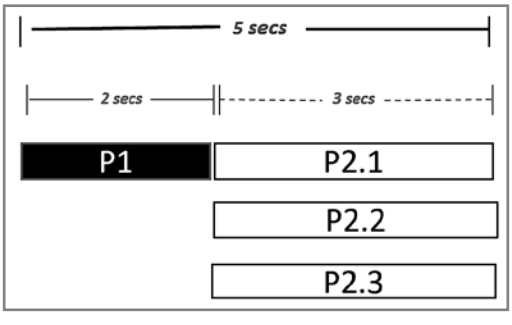


Рис. 13.2

Вычислим также:

- n_p = количество процессоров = 3;
- b = параллельная часть = 9/11 = 81,82 %;
- s = ускорение = 3.

Теперь взглянем на типичный график, объясняющий закон Амдала (рис. 13.3).

На этой диаграмме график строится между s и n_p для разных значений b.

Гранулярность задачи

При распараллеливании алгоритма большая задача делится на несколько параллельных подзадач. Их оптимальное количество не всегда очевидно. Если подзадач слишком мало, параллельные вычисления не принесут особой пользы; слишком большое количество подзадач чересчур увеличит затраты ресурсов. Эта проблема называется *гранулярностью задачи*.

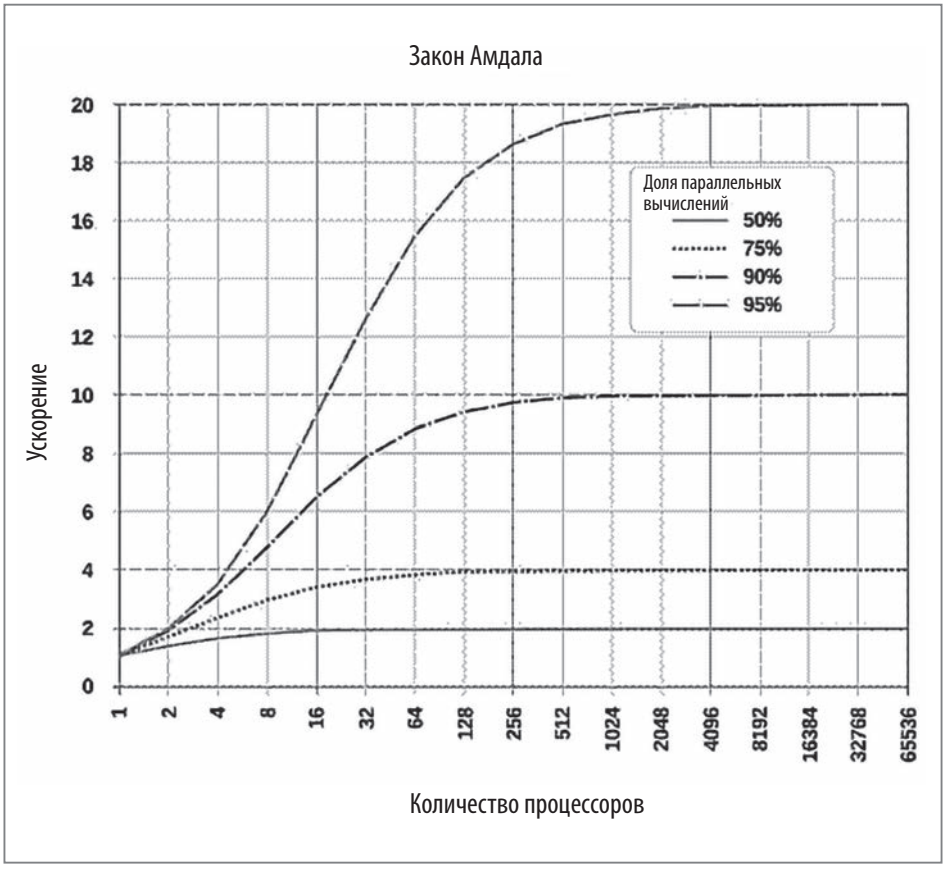


Рис. 13.3

Балансировка нагрузки

В параллельных вычислениях за выбор ресурсов для выполнения задачи отвечает *планировщик*. Оптимальной *балансировки нагрузки* достичь сложно, а при ее отсутствии ресурсы используются не в полной мере.

Проблема расположения

При параллельной обработке не рекомендуется перемещать данные. По возможности их следует обрабатывать в той ноде, в которой они находятся. В противном случае качество распараллеливания снижается.

Запуск параллельной обработки на Python

Самый простой способ запустить параллельную обработку на Python — это клонировать текущий процесс, который запустит новый параллельный процесс, называемый *дочерним*.

Программисты Python (хотя они и не биологи) создали собственный процесс клонирования. Как и в случае с клонированной овцой, клонированный процесс является точной копией исходного процесса.

РАЗРАБОТКА СТРАТЕГИИ МУЛЬТИПРОЦЕССОРНОЙ ОБРАБОТКИ

Первоначально крупномасштабные алгоритмы использовались для работы на огромных машинах, называемых *суперкомпьютерами*.

Эти суперкомпьютеры использовали одну область памяти. Все ресурсы были локальными — физически размещены на одной машине. Благодаря этому связь между процессорами (CPU) была очень быстрой, и они могли использовать одну и ту же переменную в общей области памяти. В результате развития систем и роста потребности в крупномасштабных алгоритмах суперкомпьютеры получили *распределенную общую память* (distributed shared memory, DSM), где каждая нода обработки владела частью физической памяти. Наконец, были разработаны слабо связанные кластеры, которые полагаются на обмен сообщениями между нодами. Для крупномасштабных алгоритмов необходимо найти несколько механизмов выполнения, работающих параллельно, чтобы решать сложные задачи (рис. 13.4).

Существуют три стратегии задействования механизмов выполнения:

- *Поиск внутри.* Задействовать ресурсы, уже имеющиеся на компьютере. Использовать сотни ядер GPU для запуска крупномасштабного алгоритма.
- *Поиск снаружи.* С помощью распределенных вычислений найти больше вычислительных ресурсов, которые можно одновременно применить для решения масштабной проблемы.
- *Гибридная стратегия.* Использовать распределенные вычисления и при этом на каждой ноде задействовать GPU (или массив GPU) для ускорения выполнения алгоритма.

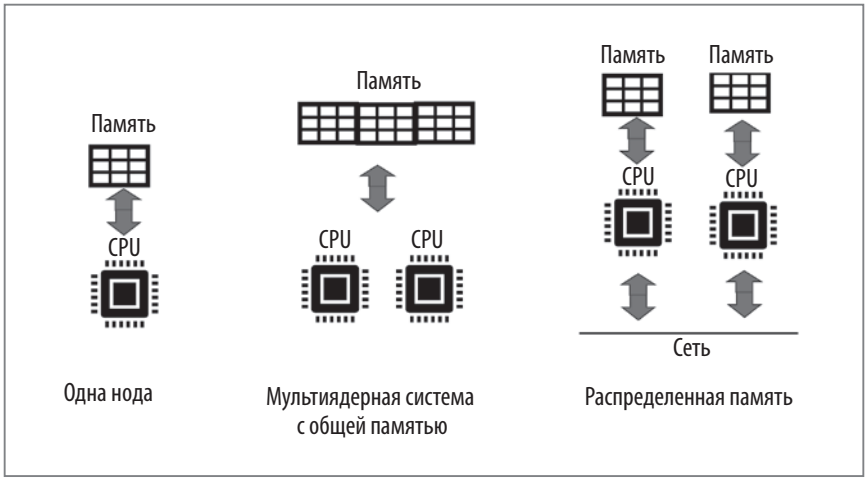


Рис. 13.4

Введение в CUDA

Графические процессоры изначально были разработаны для обработки графики, оптимизации работы с мультимедийными данными обычного компьютера. В связи с этим они обладают определенными особенностями, которые отличают их от процессоров (CPU). Например, у них тысячи ядер по сравнению с ограниченным числом ядер CPU. Их тактовая частота намного ниже, чем у CPU. GPU имеют свою собственную DRAM (динамическую память). Например, RTX 2080 от Nvidia имеет 8 ГБ оперативной памяти. Обратите внимание, что GPU являются специализированными устройствами обработки и не имеют некоторых функций CPU, таких как прерывание или адресация к устройству (например, клавиатуры и мыши). Вот как выглядит архитектура графических процессоров (рис. 13.5).

Вскоре после того, как GPU стали распространенным явлением, специалисты по обработке данных начали изучать их на предмет возможности эффективного выполнения параллельных операций. Поскольку типичный GPU имеет тысячи *ALU* (arithmetic logic unit; арифметико-логическое устройство), он может генерировать тысячи одновременных процессов. Это делает GPU архитектурой, оптимизированной для параллельных вычислений. Следовательно, алгоритмы параллельных вычислений лучше всего запускать на GPU. Например, поиск объектов в видео выполняется по меньшей мере в 20 раз быстрее на GPU по сравнению с CPU. Графовые алгоритмы, которые обсуждались в главе 5, также работают намного быстрее на графических процессорах, чем на обычных.

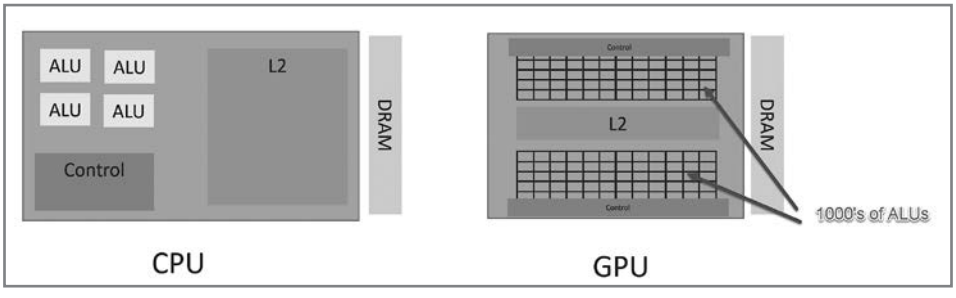


Рис. 13.5

Чтобы мечты дата-сайентистов о полноценном использовании GPU для алгоритмов воплотились в жизнь, в 2007 году Nvidia создала платформу с открытым исходным кодом под названием *CUDA* (Compute Unified Device Architecture). CUDA превращает CPU и GPU в *хост* (host) и *устройство* (device) соответственно. Хост, то есть центральный процессор, отвечает за вызов устройства, которым является графический процессор. Архитектура CUDA имеет различные уровни абстракций, которые можно представить следующим образом (рис. 13.6).

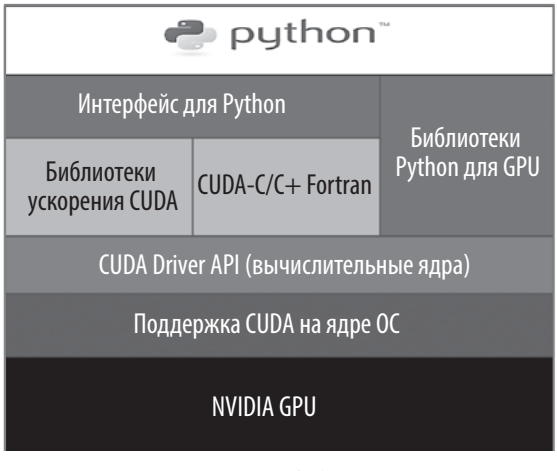


Рис. 13.6

Обратите внимание, что CUDA работает поверх графических процессоров Nvidia. Она нуждается в поддержке ядром операционной системы. Изначально CUDA поддерживалась ядром Linux. Совсем недавно она получила полную поддержку Windows. Кроме того, существует CUDA Driver API, который действует как мост между API языка программирования и драйвером CUDA. На верхнем уровне расположена поддержка C, C+ и Python.

Проектирование параллельных алгоритмов на CUDA

Давайте разберемся, как именно GPU ускоряет определенные операции обработки. Как известно, процессоры предназначены для последовательной обработки данных. Это значительно увеличивает время выполнения некоторых приложений. Рассмотрим пример обработки изображения размером 1920 × 1200 (можно подсчитать, что это 2 204 000 пикселей). Последовательная обработка на традиционном CPU займет много времени. Современные GPU, такие как Tesla от Nvidia, способны генерировать невероятное количество параллельных потоков для обработки этих 2 204 000 пикселей. В большинстве мультимедийных приложений пиксели могут обрабатываться независимо друг от друга, что дает значительное ускорение. Если сопоставить каждый пиксель с потоком, все они обработаются за постоянное время O(1).

Но обработка изображений — не единственная область применения параллелизма данных для ускорения процесса. Его можно использовать и при подготовке данных для библиотек машинного обучения. Также GPU может значительно сократить время выполнения таких распараллеливаемых алгоритмов, как:

- майнинг криптовалюты;
- крупномасштабные симуляции;
- анализ ДНК;
- анализ видео и фотографий.

GPU не предназначены для работы по принципу SPMD (single program, multiple data, единая программа — множество данных). Например, если нужно вычислить хеш для блока данных, используется одна программа, которую нельзя распараллелить. В подобных сценариях GPU будут работать медленнее.

Код для запуска на GPU помечен специальными ключевыми словами CUDA — *kernels* (ядра). Они используются для обозначения функций, которые будут вызываться на GPU для параллельной обработки. Основываясь на ключевых словах, компилятор GPU отделяет код, который должен выполняться на GPU, от кода для CPU.

Использование GPU для обработки данных на Python

GPU отлично подходят для обработки данных в многомерной структуре данных. Эти структуры по своей природе являются распараллеливаемыми. Давайте узнаем, как используется GPU в Python.

1. Прежде всего импортируем необходимые библиотеки Python:

```
import numpy as np
import cupy as cp
import time
```

2. Создадим многомерный массив в NumPy, традиционной библиотеке Python, использующей CPU .
3. Затем создадим многомерный массив, задействуя массив CuPy, который использует уже GPU. Теперь сравним время выполнения:

```
### Запуск на CPU с помощью Numpy
start_time = time.time()
myvar_cpu = np.ones((800,800,800))
end_time = time.time()
print(end_time - start_time)

### Запуск на GPU с помощью CuPy
start_time = time.time()
myvar_gpu = cp.ones((800,800,800))
cp.cuda.Stream.null.synchronize()
end_time = time.time()
print(end_time - start_time)
```

Если запустить этот код, он сгенерирует следующий вывод (рис. 13.7).

```
1.130657434463501
0.012250661849975586
```

Рис. 13.7

Обратите внимание, что для создания массива в NumPy потребовалось около 1.13 секунды, а в CuPy — около 0.012 секунды, что ускоряет инициализацию этого массива на GPU в 92 раза.

Кластерные вычисления

Кластерные вычисления — один из способов реализации параллельной обработки для крупномасштабных алгоритмов. В кластерных вычислениях используется несколько нод, соединенных посредством высокоскоростной сети. Крупномасштабные алгоритмы представляются в виде заданий. Отдельное задание разделено на различные задачи, каждая из которых выполняется на отдельной ноде.

Apache Spark — одна из самых популярных платформ для реализации кластерных вычислений. В Apache Spark данные преобразуются в устойчивые распределенные наборы данных, которые называются *RDD* (resilient distributed datasets). RDD являются ключевой абстракцией Apache Spark и представляют собой неизменяемые наборы элементов, которыми можно управлять параллельно. Они разбиты на разделы и распределены по нодам следующим образом (рис. 13.8).

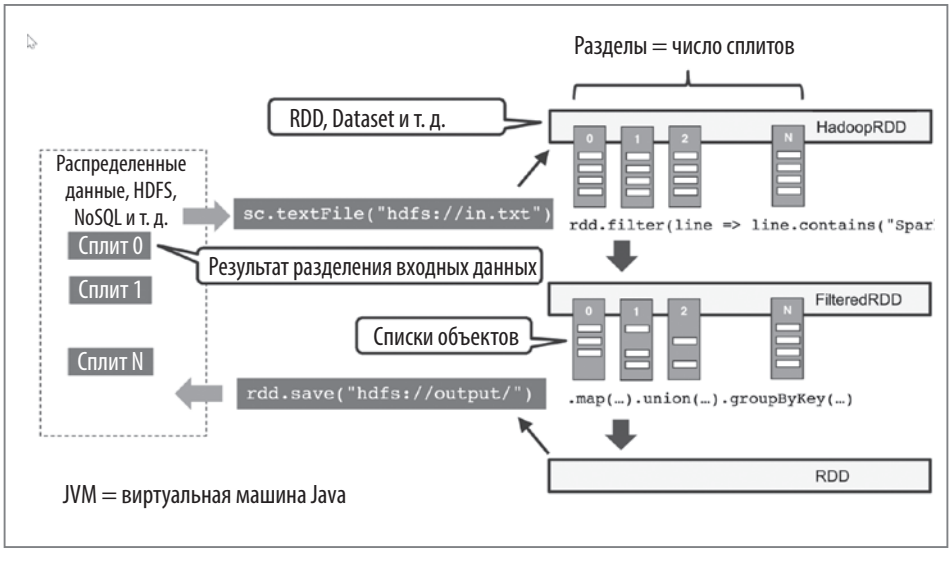


Рис. 13.8

Благодаря такой структуре данных мы можем запускать алгоритмы параллельно.

Реализация обработки данных в Apache Spark

Давайте узнаем, как создать RDD в Apache Spark и запустить распределенную обработку по всему кластеру.

1. Сначала необходимо создать новый сеанс Spark:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('cloudanum').getOrCreate()
```

2. Далее используем CSV-файл в качестве источника RDD. Запустим следующую функцию с именем `df` — она создаст RDD, который преобразуется

в DataFrame. Возможность преобразовать RDD в DataFrame была добавлена в Spark 2.0; это упрощает обработку данных:

```
df = spark.read.csv('taxi2.csv',inferSchema=True,header=True)
```

Посмотрим на столбцы DataFrame (рис. 13.9).

```
In [3]: df.columns

Out[3]: ['pickup_datetime',
         'dropoff_datetime',
         'pickup_longitude',
         'pickup_latitude',
         'dropoff_longitude',
         'dropoff_latitude',
         'passenger_count',
         'trip_distance',
         'payment_type',
         'fare_amount',
         'tip_amount',
         'tolls_amount',
         'total_amount']
```

Рис. 13.9

3. Создадим временную таблицу из DataFrame:

```
df.createOrReplaceTempView("main")
```

4. Как только временная таблица создана, можно запустить SQL-операторы для обработки данных (рис. 13.10).

```
In [9]: data=spark.sql("SELECT payment_type,Count(*) AS COUNT,AVG(fare_amount),
                    AVG(tip_amount) AS AverageFare from main GROUP BY payment_type")
        data.show()

        +------------+-----+------------------+-----------------+
        |payment_type|COUNT|  avg(fare_amount)|      AverageFare|
        +------------+-----+------------------+-----------------+
        |         CRD|10000|32.384988999999784| 7.61713200000006|
        |         Cas| 3080| 34.64730519480518|7.497457792207749|
        +------------+-----+------------------+-----------------+
```

Рис. 13.10

Важно отметить, что, хотя это выглядит как обычный DataFrame, это всего лишь высокоуровневая структура данных. На самом деле это RDD, который распределяет данные по кластеру. Подобным образом при запуске SQL-функций они преобразуются в параллельные трансформеры и редьюсеры и всецело используют мощности кластера для обработки кода.

Гибридная стратегия

Облачные вычисления становятся все более популярными для запуска крупномасштабных алгоритмов. Это дает возможность сочетать стратегии *поиска снаружи* и *внутри*. Для этого один или несколько GPU инициализируются на ряде виртуальных машин, как показано на рис. 13.11.

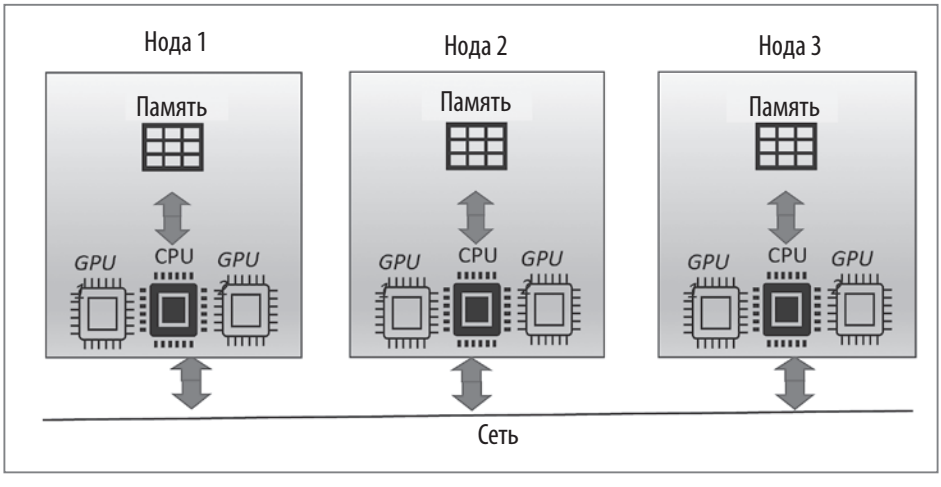


Рис. 13.11

Оптимальное использование гибридной архитектуры — задача не из легких. Для этого нужно сначала разбить данные на несколько разделов. Вычислительные задачи, требующие меньшего объема данных, распараллеливаются в каждой ноде на GPU.

РЕЗЮМЕ

В этой главе мы познакомились с параллельными алгоритмами и проблемами проектирования крупномасштабных алгоритмов. Мы обсудили применение параллельных вычислений и графических процессоров, а также кластеров Spark для реализации крупномасштабных алгоритмов.

Кроме того, мы рассмотрели проблемы, связанные с распараллеливанием алгоритмов, и потенциальные трудности, возникающие при этом.

В следующей главе представлен ряд практических аспектов реализации алгоритмов.

14

Практические рекомендации

В этой книге представлено множество алгоритмов для решения реальных задач. Данная глава посвящена некоторым практическим рекомендациям по их применению.

Глава начинается с введения. Затем раскрывается важная тема объяснимости алгоритма. Она отражает, в какой степени его внутренняя механика может быть описана в понятных терминах. Далее обсуждаются этика алгоритмов и возможность возникновения предвзятости при их реализации. Также рассматриваются методы решения NP-трудных задач и факторы, влияющие на выбор алгоритма.

К концу этой главы вы усвоите практические рекомендации, которые важно учитывать при использовании алгоритмов.

Итак, в главу включены следующие темы:

- Введение в практические рекомендации.
- Объяснимость алгоритма.
- Этика алгоритмов.
- Снижение предвзятости в моделях.
- Решение NP-трудных задач.
- Когда следует использовать алгоритмы.

Начнем с введения.

ВВЕДЕНИЕ В ПРАКТИЧЕСКИЕ РЕКОМЕНДАЦИИ

Помимо проектирования, разработки и тестирования алгоритма часто необходимо учесть определенные практические аспекты реализации. Это сделает решение задачи более эффективным. Например, для некоторых алгоритмов может понадобиться надежный способ внесения новой важной информации, которая будет меняться даже в процессе выполнения. Повлияет ли включение новых данных на качество уже проверенного алгоритма? Если да, то как он с этим справится? Для некоторых алгоритмов, использующих глобальные шаблоны, может потребоваться отслеживание изменений в мировой геополитической ситуации в режиме реального времени. В некоторых случаях, чтобы решение было эффективным, нужно учитывать нормативно-правовые изменения, вносимые во время использования алгоритма.

Используя алгоритм для решения реальной задачи, мы в некотором смысле полагаемся на компьютер. Даже самые сложные алгоритмы основаны на упрощениях и предположениях и не способны справляться с неожиданностями. Человечество даже близко не подошло к тому, чтобы полностью возложить принятие важных решений на алгоритмы.

Например, разработанные Google алгоритмы рекомендаций недавно столкнулись с нормативными ограничениями Европейского союза из-за соображений конфиденциальности. Возможно, эти алгоритмы — одни из самых продвинутых в своей области. Но если их запретить законодательно, они окажутся бесполезны, поскольку их больше нельзя будет применять для решения задач, для которых они были разработаны.

К сожалению, практические рекомендации к применению алгоритмов все еще оказываются запоздалыми соображениями, которые обычно не рассматриваются на начальном этапе проектирования. Часто бывает так, что как только алгоритм развернут и эйфория от найденного решения прошла, именно практические аспекты и последствия использования, обнаруживаемые с течением времени, определяют успех или неудачу проекта.

Рассмотрим пример, когда, проигнорировав практические рекомендации, одна из лучших IT-компаний провалила громкий проект.

Печальная история ИИ-бота в Твиттере

Давайте обратимся к классическому примеру *Tay*, который был представлен как первый в истории искусственного интеллекта Twitter-бот, созданный

Microsoft в 2016 году. Управляемый алгоритмом ИИ, Tay должен был учиться у своих собеседников и совершенствоваться. К сожалению, прожив пару дней в киберпространстве, Tay стал повторять расистские и грубые твиты пользователей. Очень скоро он начал писать собственные оскорбительные твиты. С одной стороны, бот проявил интеллект и быстро научился создавать индивидуальные твиты на основе событий в реальном времени (как и было задумано). С другой — он начал серьезно оскорблять людей. Microsoft отключила Tay и попыталась его перенастроить, но это не сработало. В конечном счете компании пришлось удалить бота. Это был печальный конец амбициозного проекта.

Хотя интеллект Tay был весьма впечатляющим, Microsoft проигнорировала последствия внедрения самообучающегося Twitter-бота. Использованные алгоритмы NLP и машинного обучения, возможно, были лучшими в своем классе, но из-за очевидных недостатков это был практически бесполезный проект. Сегодня Tay стал хрестоматийным примером неудачи из-за игнорирования практических последствий предоставления алгоритмам возможности учиться на лету. Уроки, извлеченные из провала Tay, определенно повлияли на проекты ИИ более поздних лет. Специалисты по обработке данных также начали уделять больше внимания прозрачности алгоритмов. Это подводит нас к следующей теме, в которой исследуются необходимость и способы сделать алгоритмы объяснимыми.

ОБЪЯСНИМОСТЬ АЛГОРИТМА

Алгоритм «черного ящика» — это алгоритм, логика которого не поддается интерпретации человеком либо из-за его сложности, либо из-за слишком запутанного объяснения. В то же время логика *алгоритма «белого ящика»* прозрачна и понятна человеку. Объяснимость позволяет человеку понять, почему алгоритм дает те или иные результаты. Степень объяснимости — это мера того, насколько конкретный алгоритм понятен для человеческого разума. Многие классы алгоритмов, особенно связанные с машинным обучением, классифицируются как «черный ящик». Если алгоритм используется для принятия критического решения, крайне важно знать причины, лежащие в основе полученных результатов. К тому же преобразование алгоритма из «черного ящика» в «белый» позволяет лучше понять внутреннюю работу модели. Например, объяснимый алгоритм дает врачу возможность понять, какие функции на самом деле использовались для деления пациентов на больных и здоровых. Если у врача есть какие-либо сомнения по поводу результатов, можно вернуться и перепроверить конкретные характеристики на точность.

Алгоритмы машинного обучения и объяснимость

Объяснимость алгоритма имеет особое значение для алгоритмов машинного обучения. Во многих приложениях МО от пользователей требуется доверять модели, которая помогает им принимать решения. Объяснимость в подобных случаях обеспечивает необходимую прозрачность.

Разберем более подробно конкретный пример. Предположим, что с помощью МО нам нужно спрогнозировать цены на дома в районе Бостона на основе их характеристик. Местные законы позволяют нам использовать МО, только если по запросу мы предоставим подробное обоснование любых прогнозов. Эта информация нужна, чтобы провести аудит и убедиться, что определенные сегменты рынка жилья не подвергаются искусственному манипулированию. Если обученная модель будет объяснимой, то мы сможем предоставить такую информацию.

Давайте рассмотрим различные варианты реализации объяснимости обученной модели.

Стратегии объяснимости

В машинном обучении, по сути, существуют две стратегии, которые обеспечивают объяснимость алгоритмов:

- *Глобальная объяснимость* предоставляет детальную информацию о модели в целом.
- *Локальная объяснимость* дает обоснование для одного или нескольких отдельных прогнозов, сделанных обученной моделью.

Одной из стратегий глобальной объяснимости является метод *TCAV* (Testing with Concept Activation Vectors; тестирование с помощью векторов активации концепций). Данный метод используется, чтобы обеспечить объяснимость моделей классификации изображений. TCAV основан на вычислении производных по направлению для оценки степени взаимосвязи между концепцией, которую задает пользователь, и классификацией изображений. Например, TCAV может определить, насколько классификация человека на фото как мужчины зависит от наличия бороды на этом фото. Существуют и другие стратегии глобальной объяснимости, позволяющие раскрыть механизм работы обученной модели. Это *графики частичной зависимости* (partial dependence plots) и вычисление *важности перестановки* (permutation importance). Стратегии как глобальной, так и локальной объяснимости могут быть либо моделезависимыми, либо моделе-

независимыми. В первом случае стратегии применимы только к определенным типам моделей, во втором — к широкому спектру моделей.

На следующей диаграмме представлены различные стратегии для объяснимости машинного обучения (рис. 14.1).

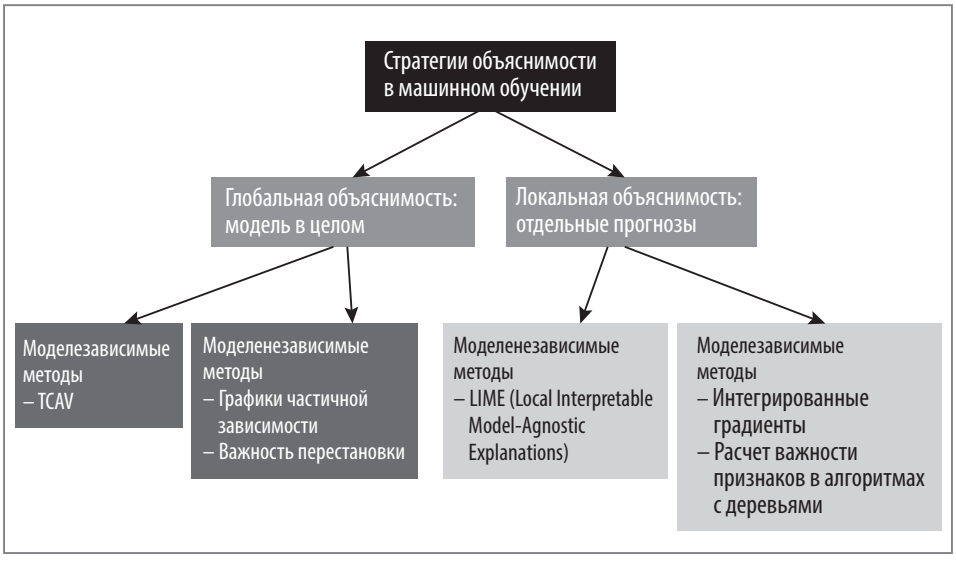


Рис. 14.1

Теперь посмотрим, как реализуется объяснимость с применением одной из этих стратегий.

Реализация объяснимости

LIME (local interpretable model-agnostic explanations; локально интерпретируемые моделенезависимые объяснения) — это моделенезависимый подход, способный объяснить отдельные прогнозы, сделанные обученной моделью. Он пригоден для интерпретации большинства типов обученных моделей МО.

LIME интерпретирует решения, внося небольшие изменения во входные данные для каждого экземпляра. Он собирает данные о воздействии этих изменений на локальную границу принятия решений для конкретной модели. LIME циклично повторяется, чтобы в итоге предоставить подробную информацию для каждой переменной. Глядя на выходные данные, можно увидеть, какая переменная оказывает наибольшее влияние на конкретный экземпляр.

Попробуем использовать LIME, чтобы сделать объяснимыми индивидуальные прогнозы модели, предсказывающей цены на жилье.

1. Для начала установим библиотеку с помощью `pip`:

```
!pip install lime
```

2. Затем импортируем нужные библиотеки Python:

```
import sklearn as sk
import numpy as np
from lime.lime_tabular import LimeTabularExplainer as ex
```

3. Подготовим модель, которая сможет прогнозировать цены на жилье в конкретном городе. Для этого импортируем набор данных, который хранится в файле `housing.pkl`. Затем рассмотрим содержащиеся в нем признаки (рис. 14.2).

```
In [2]: pkl_file = open("housing.pkl","rb")
        housing = pickle.load(pkl_file)
        pkl_file.close()
        housing['feature_names']

Out[2]: array(['crime_per_capita', 'zoning_prop', 'industrial_prop',
               'nitrogen_oxide', 'number_of_rooms', 'old_home_prop',
               'distance_from_city_center', 'high_way_access',
               'property_tax_rate', 'pupil_teacher_ratio', 'low_income_prop',
               'lower_status_prop', 'median_price_in_area'], dtype='<U25')
```

Рис. 14.2

Основываясь на этих признаках, нам нужно предсказать цену дома.

4. Перейдем к обучению модели. Для обучения используем регрессор случайного леса. Сначала разделим данные на контрольную и обучающую части, а затем обучим модели:

```
from sklearn.ensemble import RandomForestRegressor
X_train, X_test, y_train, y_test =
sklearn.model_selection.train_test_split(
    housing.data, housing.target)

regressor = RandomForestRegressor()
regressor.fit(X_train, y_train)
```

5. Далее определим столбцы категорий:

```
cat_col = [i for i, col in enumerate(housing.data.T)
                        if np.unique(col).size < 10]
```

6. Теперь создадим экземпляр объяснителя LIME с требуемыми параметрами конфигурации. Обратите внимание, что наша метка — `'price'`. Именно она представляет цены на дома в Бостоне:

```
myexplainer = ex(X_train,
    feature_names=housing.feature_names,
    class_names=['price'],
    categorical_features=cat_col,
    mode='regression')
```

7. Попробуем детально разобраться в предсказаниях. Для этого сначала импортируем `pyplot` из `matplotlib` в качестве графической библиотеки:

```
exp = myexplainer.explain_instance(X_test[25], regressor.predict,
       num_features=10)
exp.as_pyplot_figure()
from matplotlib import pyplot as plt
plt.tight_layout()
```

8. Поскольку объяснитель LIME работает с отдельными предсказаниями, нужно выбрать прогнозы для анализа. Выбираем прогнозы, проиндексированные как `1` и `35` (рис. 14.3).

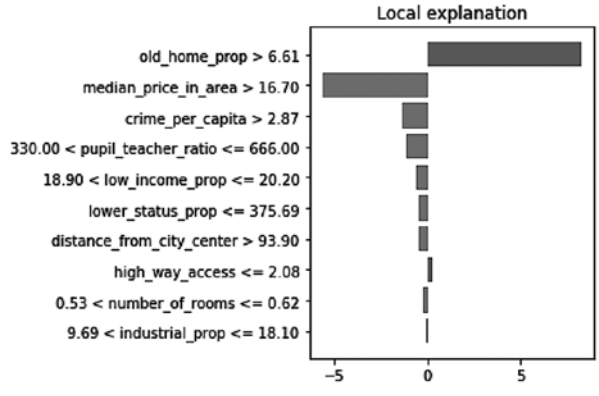
Попробуем проанализировать объяснение LIME, которое говорит нам следующее:

- *Список признаков, используемых в отдельных прогнозах*. Они указаны на оси *y* на рис. 14.3.
- *Относительная важность признаков при определении решения*. Чем больше столбец, тем больше значение. Значение числа находится на оси *x*.
- *Положительное или отрицательное влияние каждого входного признака на метку*. Полосы слева от вертикальной оси показывают отрицательное влияние, а справа — положительное влияние определенного признака.

ЭТИКА И АЛГОРИТМЫ

Представление закономерностей в виде алгоритма может прямо или косвенно привести к принятию неэтичных решений. При разработке алгоритма трудно предвидеть весь спектр потенциальных этических последствий. Особенно это касается крупномасштабных алгоритмов, в создании которых могут участвовать несколько разработчиков. Это еще более затрудняет анализ влияния человеческой субъективности.

```
In [9]: for i in [1, 35]:
            exp = myexplainer.explain_instance(X_test[i], regressor.predict,
                    num_features=10)
            exp.as_pyplot_figure()
            plt.tight_layout()
```



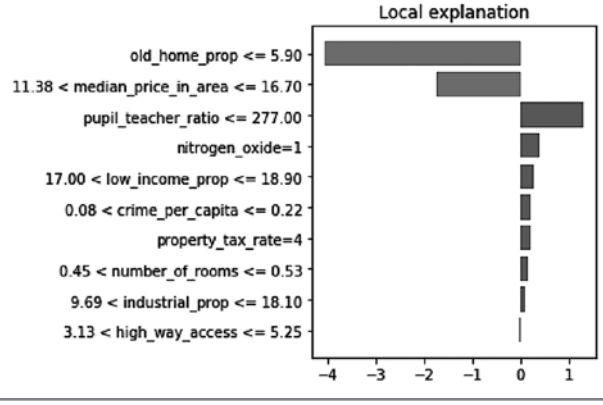


Рис. 14.3

Все больше компаний делают этический анализ алгоритма частью его проектирования. Но истина в том, что проблемы могут не проявиться, пока мы не столкнемся с ними на практике.

Проблемы обучающихся алгоритмов

Алгоритмы, способные адаптироваться в соответствии с изменяющимися закономерностями в данных, называются *обучающимися алгоритмами*. Они обучаются в режиме реального времени. В результате это может привести к спорным

решениям с этической точки зрения. Алгоритмы обучения созданы таким образом, чтобы непрерывно развиваться, поэтому проводить их этический анализ на постоянной основе практически невозможно.

По мере роста сложности алгоритмов становится все труднее полностью понять долгосрочные последствия их работы для отдельных лиц и социальных групп.

Понимание этических аспектов

Алгоритмические решения — это сухие математические формулировки. Разработчики несут ответственность за то, чтобы алгоритмы соответствовали этическим нормам в контексте задачи. Эти соображения зависят от типа алгоритма.

Рассмотрим некоторые алгоритмы и их этические аспекты. Ниже приведены примеры мощных алгоритмов, для которых необходима тщательная этическая проработка:

- Алгоритмы классификации в случае применения к обществу определяют способы организации и управления отдельными лицами и группами.
- Алгоритмы, используемые в рекомендательных системах, сопоставляют резюме с соискателями как в случае отдельных лиц, так и групп.
- Алгоритмы майнинга данных используются для получения информации от пользователей и предоставления этих данных лицам, принимающим решения, и правительствам.
- Алгоритмы МО начинают использоваться правительствами для выдачи или отказа в выдаче виз заявителям.

Таким образом, этические аспекты алгоритмов зависят от конкретного случая и лиц, на которых они прямо или косвенно влияют. Прежде чем использовать алгоритм для принятия важных решений, необходимо провести тщательный этический анализ. В следующих разделах мы рассмотрим факторы, которые следует учитывать при подробном анализе алгоритмов.

Неубедительные доказательства

Данные, используемые для обучения алгоритма МО, могут быть спорными. Например, в клинических испытаниях эффективность препарата может быть

не доказана из-за недостатка имеющихся данных. Аналогичным образом, можно найти несколько неубедительных доказательств того, что конкретный почтовый индекс в определенном городе с большой вероятностью причастен к мошенничеству. Нужно крайне осторожно оценивать результаты работы алгоритмов, использующих такие ограниченные данные.

Решения, основанные на неубедительных доказательствах, могут привести к неправомерным действиям.

Трассируемость

Разрыв между этапами обучения и тестирования в алгоритмах МО означает, что, если алгоритм наносит какой-либо вред, проблему очень трудно отследить и устранить. Кроме того, трудно определить людей, которых она затронула.

Искажения в данных

Алгоритмы — это математические формулировки, основанные на данных. Принцип *GIGO (Garbage-in, Garbage-out; «мусор на входе» = «мусор на выходе»)* означает, что результаты работы алгоритма надежны ровно настолько, насколько достоверны входные данные. Если в данных есть искажения, это отразится и на результате.

Несправедливые результаты

Использование алгоритмов может привести к нанесению ущерба уязвимым сообществам и группам, которые уже находятся в невыгодном положении.

Кроме того, не раз было доказано, что алгоритмы распределения финансирования исследований проявляют *предвзятость* в пользу мужчин. Алгоритмы, используемые для предоставления возможности иммиграции, иногда предвзяты в отношении уязвимых групп населения.

Несмотря на использование высококачественных данных и сложных математических формулировок, результат может оказаться несправедливым. И тогда все эти усилия принесут больше вреда, чем пользы.

СНИЖЕНИЕ ПРЕДВЗЯТОСТИ В МОДЕЛЯХ

В современном мире существуют известные, документально подтвержденные общие предубеждения, основанные на поле, расе и сексуальной ориентации. Это означает, что собранные нами данные будут содержать такие предубеждения

(если только мы не имеем дело со средой, в которой были предприняты усилия для устранения таких искажений до сбора данных).

Алгоритмическая предвзятость прямо или косвенно обусловлена человеческими предубеждениями. Они могут быть отражены либо в данных, используемых алгоритмом, либо в описании самого алгоритма. Для типичного проекта машинного обучения, следующего жизненному циклу *CRISP-DM*, который был описан в главе 6, предвзятость выглядит следующим образом (рис. 14.4).

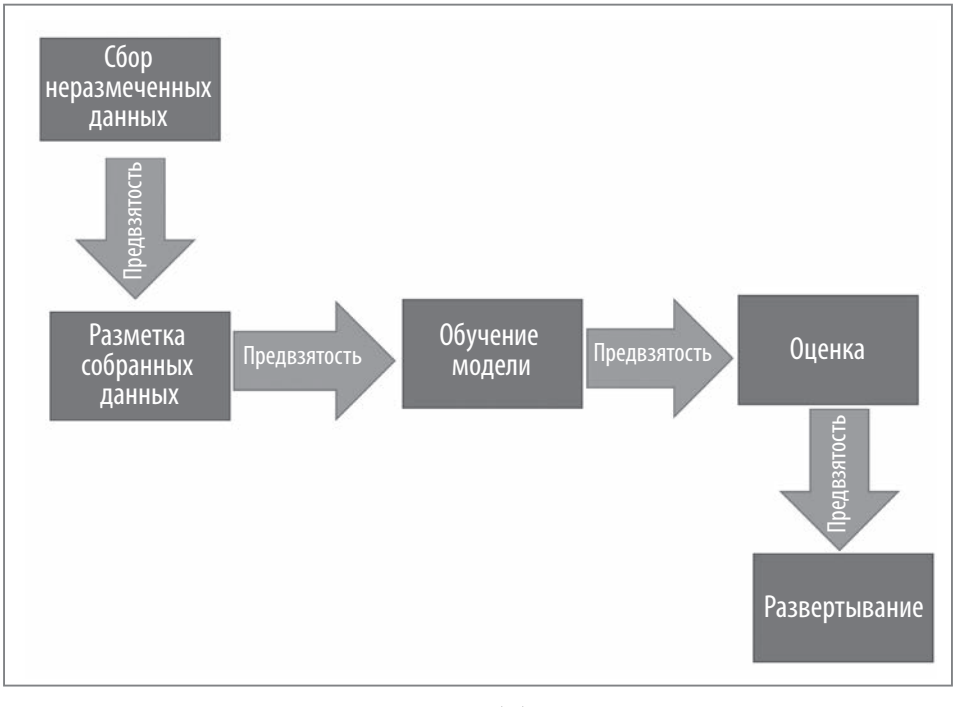


Рис. 14.4

При снижении предвзятости труднее всего в первую очередь выявить и локализовать невольное (неосознаваемое) предубеждение.

РЕШЕНИЕ NP-ТРУДНЫХ ЗАДАЧ

NP-трудные задачи подробно обсуждались в главе 4. Некоторые задачи этого класса крайне важны, а значит, необходимо разрабатывать алгоритмы для их решения.

Если NP-трудная задача кажется нерешаемой из-за сложности или ограниченности имеющихся ресурсов, можно воспользоваться одним из следующих подходов:

- упрощение задачи;
- адаптация известного решения аналогичной задачи;
- вероятностный метод.

Давайте рассмотрим каждый подход.

Упрощение задачи

Задачу можно упростить, приняв определенные допущения. В этом случае решение будет хотя и не идеальным, но приемлемым и все равно полезным. Чтобы это сработало, допущения должны быть как можно менее ограничивающими.

Пример

Взаимосвязь между признаками и метками в задачах регрессии редко бывает идеально линейной. Но ее можно считать линейной в пределах нашего обычного рабочего диапазона. Такая аппроксимация значительно упрощает алгоритм и весьма широко применяется. Однако это отрицательно влияет на точность алгоритма. Нужно тщательно изучить компромисс между допущениями и точностью и выбрать оптимальное соотношение, подходящее для заинтересованных сторон.

Адаптация известного решения аналогичной задачи

Если существует решение похожей задачи, то его можно использовать в качестве отправной точки и адаптировать для новой задачи. Идея *переноса обучения* основана именно на этом принципе. Результат работы уже обученных моделей используется в качестве стартовой точки для разработки нового алгоритма.

Пример

Предположим, нам нужно обучить бинарный классификатор, который во время корпоративного обучения различает ноутбуки Apple и Windows на видеотрансляции, используя распознавание объектов. На первом этапе мы научим модель определять на видео различные объекты и выявлять ноутбуки. После этого можно перейти ко второму этапу — разработке правил, которые помогут различать ноутбуки Apple и Windows.

Уже существуют хорошо обученные и протестированные модели с открытым исходным кодом, которые могут справиться с первым этапом обучения алгоритма. Почему бы не использовать их в качестве отправной точки? Можно сразу перейти ко второму этапу разработки, применив готовое решение. Это даст нам хороший старт, а результат будет более надежным, поскольку первый этап уже хорошо протестирован.

Вероятностный метод

Вероятностный метод (probabilistic method) дает достаточно хорошее решение, которое является рабочим, но не наилучшим. В главе 7 мы применили к задаче алгоритм дерева решений и получили результат, основанный на вероятностном подходе. Мы не доказали, что это идеальное решение, но оно было достаточно хорошим. В итоге был получен приемлемый результат, с учетом ограничений, описанных в требованиях задачи.

Пример

Многие алгоритмы МО отталкиваются от случайного решения, а затем итеративно его улучшают. Окончательное решение может быть эффективным, но доказать, что оно является лучшим, невозможно. Этот метод используется для решения сложных задач в разумные сроки. Вот почему для многих алгоритмов МО единственный способ получить воспроизводимые результаты — это использовать одну и ту же последовательность случайных чисел, применяя одно и то же *начальное значение* (seed).

КОГДА СЛЕДУЕТ ИСПОЛЬЗОВАТЬ АЛГОРИТМЫ

Алгоритмы подобны инструментам в арсенале рабочего. Прежде всего необходимо понять, какой инструмент лучше всего использовать в конкретных обстоятельствах. Для этого нужно ответить на некоторые вопросы. Существует ли вообще алгоритм для этой задачи? Когда наступит подходящее время для его внедрения? Даст ли алгоритм действительно полезное решение (в отличие от альтернативных вариантов)? Нужно проанализировать эффект от использования алгоритма с точки зрения трех аспектов:

- *Затраты*. Оправдывает ли результат затраты, связанные с реализацией алгоритма?
- *Время*. Повышает ли алгоритм производительность в сравнении с более простыми альтернативами?

- *Точность*. Дает ли более он точные результаты, нежели другие, не столь сложные алгоритмы?

Чтобы выбрать подходящий алгоритм, нам придется ответить на следующие вопросы:

- Возможно ли упростить задачу, введя ряд допущений?
- Как оценивать алгоритм? Каковы ключевые показатели?
- Как именно будет развернут и использован этот алгоритм?
- Должен ли он быть объяснимым?
- Учитываются ли три важных нефункциональных требования — безопасность, производительность и доступность?
- Установлен ли срок выполнения работ?

Практический пример — события типа «черный лебедь»

Алгоритмы получают, обрабатывают и математически описывают данные, чтобы решить поставленную задачу. Но что, если собранные данные касаются чрезвычайно важного и очень редкого события? Как использовать алгоритмы на основе данных, сгенерированных таким событием, а также обстоятельствами, которые привели к этому «большому взрыву»?

Нассим Талеб в 2001 году в своей книге «Одураченные случайностью» обозначил чрезвычайно редкие явления метафорой *«черный лебедь»*.

До того как черные лебеди были впервые обнаружены в дикой природе, на протяжении веков они символизировали то, чего не бывает (все лебеди — белые). После открытия этих птиц понятие осталось в обиходе, но его употребление изменилось. Теперь «черный лебедь» обозначает настолько редкое событие, что его невозможно предсказать.

Талеб представил четыре критерия классификации события как «черного лебедя».

Четыре критерия классификации события как «черного лебедя»

Не так уж просто определить, является ли редкое событие «черным лебедем». Для этого оно должно соответствовать следующим четырем критериям.

1. Само это событие должно вызвать у наблюдателей шок. Примером может служить атомная бомбардировка Хиросимы.
2. Событие должно стать сенсационным — разрушительным и масштабным, таким как вспышка испанского гриппа.
3. Как только событие произойдет и пыль уляжется, дата-сайентисты, включенные в группу наблюдателей, должны осознать, что на самом деле оно не было таким уж непредсказуемым. Просто сами наблюдатели никогда не обращали внимания на некоторые важные подсказки. Если бы у них были возможности и желание, событие типа «черный лебедь» можно было бы предсказать. Например, на вспышку испанского гриппа указывали некоторые признаки. Но они, как известно, игнорировались до того, как она стала глобальной. Манхэттенский проект осуществлялся в течение многих лет до того, как атомная бомба была фактически сброшена на Хиросиму. Люди в группе наблюдателей просто не смогли связать факты воедино.
4. В то время как для наблюдателей «черный лебедь» оказывается самым большим потрясением в жизни, у некоторых людей он может вообще не вызвать удивления. Например, для ученых, годами работавших над созданием атомной бомбы, ее использование не стало неожиданностью, а было вполне предсказуемым событием.

Применение алгоритмов к событиям типа «черный лебедь»

Ниже представлены главные аспекты применения алгоритмов к событиям типа «черный лебедь».

- Существует множество сложных алгоритмов прогнозирования. Но если мы надеемся использовать стандартные методы, чтобы предсказать и предупредить событие типа «черный лебедь», то это не сработает. Это даст лишь чувство ложной безопасности.
- После того как произошло событие типа «черный лебедь», точно предсказать его последствия для всех сфер социальной жизни (экономики, общества, государства), как правило, невозможно. Поскольку это редкое событие, не существует нужных данных для передачи их алгоритмам. К тому же не всегда понятна взаимосвязь между разными социальными проявлениями (возможно, ее никогда до этого не исследовали).
- Важно отметить, что события типа «черный лебедь» не являются случайными. У нас просто не было возможности обратить внимание на сложные обстоятельства, которые в конечном итоге привели к этим событиям. Это та область, где алгоритмы могут сыграть важную роль. В будущем необходимо

выработать стратегию прогнозирования и обнаружения таких, на первый взгляд незначительных, событий, которые со временем, объединяясь, порождают событие типа «черный лебедь».

Вспышка COVID-19 в начале 2020 года — лучший пример события типа «черный лебедь» нашего времени.

Предыдущий пример демонстрирует, насколько важно сначала детально проанализировать стоящую перед нами задачу. Затем нужно определить области, в которых применение алгоритма будет способствовать решению. Без всестороннего анализа, как уже было отмечено ранее, использование алгоритмов может решить только часть сложной задачи и не оправдает ожиданий.

РЕЗЮМЕ

В этой главе мы узнали о практических аспектах, которые следует учитывать при разработке алгоритмов. Мы рассмотрели концепцию алгоритмической объяснимости и различные способы, которыми ее можно обеспечить на разных уровнях. Мы также обсудили потенциальные этические проблемы, с которыми можно столкнуться, и факторы, влияющие на выбор алгоритма.

Алгоритмы — это механизмы нового автоматизированного мира, в котором мы сегодня живем. Важно изучать, экспериментировать и осознавать последствия их использования. Понимание сильных сторон и недостатков алгоритмов, а также этических последствий их применения может внести значительный вклад в улучшение нашей жизни. И эта книга — попытка достичь этой, несомненно, важной цели в постоянно меняющемся и развивающемся мире.

Имран Ахмад

40 алгоритмов, которые должен знать каждый программист на Python

Перевел с английского Р. Чикин

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Е. Строганова*
Научный редактор	*В. Кадочников*
Литературный редактор	*Ю. Лесняк*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Т. Никифорова*
Верстка	*Л. Егорова*

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: 01.2023. 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 25.11.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 29,670. Тираж 1000. Заказ 0000.

Алекс Петров

РАСПРЕДЕЛЕННЫЕ ДАННЫЕ. АЛГОРИТМЫ РАБОТЫ СОВРЕМЕННЫХ СИСТЕМ ХРАНЕНИЯ ИНФОРМАЦИИ

Когда дело доходит до выбора, использования и обслуживания базы данных, важно понимать ее внутреннее устройство. Как разобраться в огромном море доступных сегодня распределенных баз данных и инструментов? На что они способны? Чем различаются?

Алекс Петров знакомит нас с концепциями, лежащими в основе внутренних механизмов современных баз данных и хранилищ. Для этого ему пришлось обобщить и систематизировать разрозненную информацию из многочисленных книг, статей, постов и даже из нескольких баз данных с открытым исходным кодом.

Вы узнаете о принципах и концепциях, используемых во всех типах СУБД, с акцентом на подсистеме хранения данных и компонентах, отвечающих за распределение. Эти алгоритмы используются в базах данных, очередях сообщений, планировщиках и в другом важном инфраструктурном программном обеспечении. Вы разберетесь, как работают современные системы хранения информации, и это поможет взвешенно выбирать необходимое программное обеспечение и выявлять потенциальные проблемы.

Брэдфорд Такфилд

АЛГОРИТМЫ НЕФОРМАЛЬНО. ИНСТРУКЦИЯ ДЛЯ НАЧИНАЮЩИХ ПИТОНИСТОВ

Алгоритмы — это не только задачи поиска, сортировки или оптимизации, они помогут вам поймать бейсбольный мяч, проникнуть в «механику» машинного обучения и искусственного интеллекта и выйти за границы возможного.

Вы узнаете нюансы реализации многих самых популярных алгоритмов современности, познакомитесь с их реализацией на Python 3, а также научитесь измерять и оптимизировать их производительность.

А. Бхаргава

ГРОКАЕМ АЛГОРИТМЫ. ИЛЛЮСТРИРОВАННОЕ ПОСОБИЕ ДЛЯ ПРОГРАММИСТОВ И ЛЮБОПЫТСТВУЮЩИХ

Алгоритмы — это всего лишь пошаговые инструкции решения задач, и большинство таких задач уже были кем-то решены, протестированы и проверены. Можно, конечно, погрузиться в глубокую философию гениального Кнута, изучить многостраничные фолианты с доказательствами и обоснованиями, но хотите ли вы тратить на это свое время? Откройте великолепно иллюстрированную книгу и вы сразу поймете, что алгоритмы — это просто. А грокать алгоритмы — веселое и увлекательное занятие.

Ришал Харбанс

ГРОКАЕМ АЛГОРИТМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Искусственный интеллект — часть нашей повседневной жизни. Мы встречаемся с его проявлениями, когда занимаемся шопингом в интернет-магазинах, получаем рекомендации «вам может понравиться этот фильм», узнаем медицинские диагнозы…

Чтобы уверенно ориентироваться в новом мире, необходимо понимать алгоритмы, лежащие в основе ИИ.

«Грокаем алгоритмы искусственного интеллекта» объясняет фундаментальные концепции ИИ с помощью иллюстраций и примеров из жизни. Все, что вам понадобится, — это знание алгебры на уровне старших классов школы, и вы с легкостью будете решать задачи, позволяющие обнаружить банковских мошенников, создавать шедевры живописи и управлять движением беспилотных автомобилей.

КУПИТЬ

Кори Альтхофф

COMPUTER SCIENCE ДЛЯ ПРОГРАММИСТА-САМОУЧКИ. ВСЕ ЧТО НУЖНО ЗНАТЬ О СТРУКТУРАХ ДАННЫХ И АЛГОРИТМАХ

Книги Кори Альтхоффа вдохновили сотни тысяч людей на самостоятельное изучение программирования.

Чтобы стать профи в программировании, не обязательно иметь диплом в области computer science, и личный опыт Кори подтверждает это: он стал разработчиком ПО в eBay и добился этого самостоятельно.

Познакомьтесь с наиболее важными темами computer science, в которых должен разбираться каждый программист-самоучка, мечтающий о выдающейся карьере, — это структуры данных и алгоритмы. «Computer Science для программиста-самоучки» поможет вам пройти техническое интервью, без которого нельзя получить работу в «айти».

Книга написана для абсолютных новичков, поэтому у вас не должно возникнуть трудностей, даже если ранее вы ничего не слышали о computer science.

КУПИТЬ