.
.
.
.

50 алгоритмов, которые должен знать каждый программист

А. Глава 1: Обзор алгоритмов
Б. Глава 2: Структуры данных, используемые в алгоритмах
В. Глава 3: Алгоритмы сортировки и поиска
Г. Глава 4: Проектирование алгоритмов
Д. Глава 5: Алгоритмы графов
Е. Глава 6: Алгоритмы машинного обучения без надзора
Ж. Глава 7: Традиционные алгоритмы обучения под надзором
З. Глава 8: Алгоритмы нейронных сетей
И. Глава 9: Продвинутые алгоритмы глубокого обучения
К. Глава 10: Алгоритмы обработки естественного языка
Л. Глава 11: Движки рекомендаций
М. Глава 12: Алгоритмы данных
Н. Глава 13: Криптография
О. Глава 14: Крупномасштабные алгоритмы

Раздел 1: Основы и основные алгоритмы

В этом разделе представлены основные аспекты алгоритмов. Мы рассмотрим, что такое алгоритм и как его проектировать. Мы также узнаем о структурах данных, используемых в алгоритмах. В этом разделе также представлены алгоритмы сортировки и поиска, а также алгоритмы для решения графических задач. Главы, включённые в этот раздел, следующие:

- *Глава 1*, *Обзор алгоритмов*
- *Глава 2*, *Структуры данных, используемые в алгоритмах*
- *Глава 3*, *Алгоритмы сортировки и поиска*
- *Глава 4*, *Проектирование алгоритмов*

Глава 5, *Алгоритмы графов*

Эта книга охватывает информацию, необходимую для понимания, классификации, выбора и реализации важных алгоритмов. Помимо объяснения их логики, эта книга также рассматривает структуры данных, среды разработки и производственные среды, подходящие для различных классов алгоритмов. Это второе издание этой книги. В этом выпуске мы особенно сосредоточились на современных алгоритмах машинного обучения, которые становятся всё более важными. Помимо логики, приводятся практические примеры использования алгоритмов для решения реальных повседневных задач. Эта глава даёт представление о основах алгоритмов. Он начинается с раздела о базовых концепциях, необходимых для понимания работы различных алгоритмов. Чтобы дать историческую перспективу, этот раздел кратко описывает, как люди начали использовать алгоритмы для математического формулирования определённого класса задач. Также упоминаются ограничения различных алгоритмов. В следующем разделе объясняются различные способы задания логики алгоритма. Поскольку Python используется в этой книге для написания алгоритмов, объясняется, как настроить среду для выполнения примеров. Затем обсуждаются различные способы количественной оценки и сравнения производительности алгоритма с другими алгоритмами. Наконец, в этой главе

рассматриваются различные способы проверки конкретной реализации алгоритма. В итоге, эта глава охватывает следующие основные моменты:

- Что такое алгоритм?
- Фазы алгоритма
- Среда развития
- Методы проектирования алгоритмов
- Анализ эффективности

Валидация алгоритма

Что такое алгоритм?

Проще говоря, алгоритм — это набор правил для выполнения некоторых вычислений с целью решения задачи. Он предназначен для получения результатов для любого допустимого входа в соответствии с точно определёнными инструкциями. Если поискать слово «алгоритм» в словаре (например, American Heritage), оно определяется следующим образом:

> *Алгоритм — это конечное множество однозначных инструкций, которые при определённом наборе начальных условий могут выполняться в заданной последовательности для достижения определённой цели и имеют узнаваемый набор конечных условий.*

Проектирование алгоритма — это попытка создать математический рецепт максимально эффективным способом, который можно использовать для решения реальной задачи. Этот рецепт может служить основой для разработки более многократного и универсального математического решения, которое можно применить к более широкому набору подобных задач.

Фазы алгоритма

Различные этапы разработки, развертывания и, наконец, использования алгоритма показаны на *рисунке 1.1*:

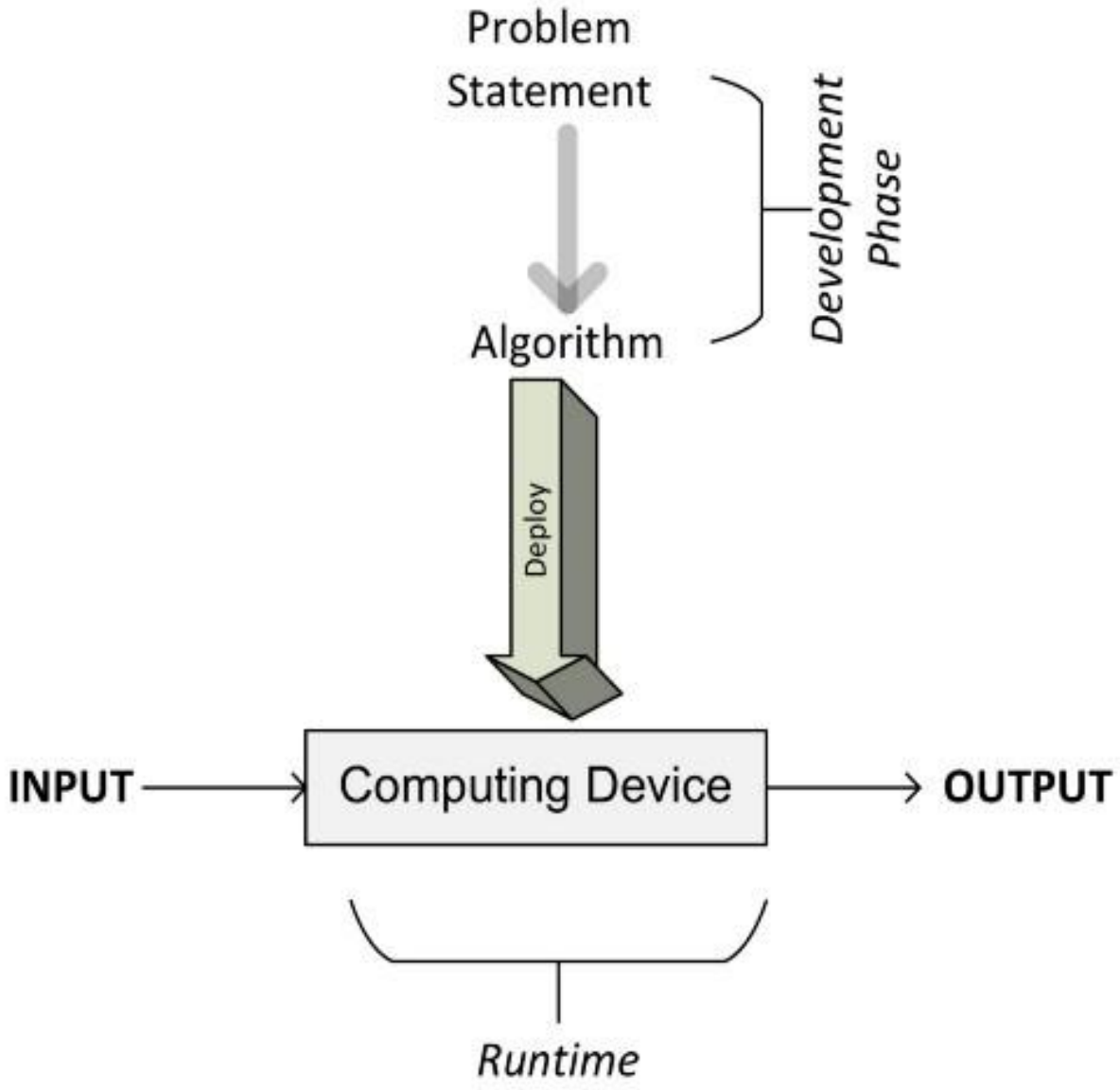


Рисунок 1.1: Различные этапы разработки, внедрения и использования алгоритма

Как видно, процесс начинается с понимания требований из формулировки задачи, в которых подробно описывается, что нужно сделать. Когда проблема чётко изложена, это приводит нас к этапу разработки. Этап разработки состоит из двух этапов:

A. **Фаза** проектирования: На этапе проектирования задумываются и документируются детали архитектуры, логики и реализации

алгоритма. При разработке алгоритма мы учитываем и точность, и производительность. При поиске наилучшего решения конкретной задачи во многих случаях у нас будет более одного кандидата алгоритмов. Этап проектирования алгоритма — это итеративный процесс, включающий сравнение различных кандидатов алгоритмов. Некоторые алгоритмы могут предлагать простые и быстрые решения, но могут идти на компромисс с точностью. Другие алгоритмы могут быть очень точными, но могут требовать значительного времени на выполнение из-за своей сложности. Некоторые из этих сложных алгоритмов могут быть эффективнее других. Перед выбором следует тщательно изучить все присущие компромиссы кандидатских алгоритмов. Особенно для сложной задачи важно разработать эффективный алгоритм. Правильно разработанный алгоритм приведёт к эффективному решению, способному одновременно обеспечивать удовлетворительную производительность и разумную точность.

Б. **Фаза** кодирования: на этапе кодирования разработанный алгоритм преобразуется в компьютерную программу. Важно, чтобы программа реализовала всю логику и архитектуру, предложенные на этапе проектирования.

Требования к бизнес-задаче можно разделить на функциональные и нефункциональные. Требования, которые прямо определяют ожидаемые характеристики решений, называются функциональными требованиями. Функциональные требования подробно описывают ожидаемое поведение решения. С другой стороны, нефункциональные требования касаются производительности, масштабируемости, удобства использования и точности алгоритма. Нефункциональные требования также формируют ожидания относительно безопасности данных. Например, предположим, что нам необходимо разработать алгоритм для кредитной компании, который сможет выявлять и отмечать мошеннические транзакции. Требования к функциям в этом примере задают ожидаемое поведение допустимого решения, предоставляя детали ожидаемого выхода при определённом наборе входных данных. В этом случае

входные данные могут содержать детали транзакции, а вывод — бинарный флаг, который обозначает транзакцию как мошенническую или немошеннический. В этом примере нефункциональные требования могут указывать время отклика каждого из предсказаний. Нефункциональные требования также устанавливают допустимые пороги точности. Поскольку мы имеем дело с финансовыми данными в этом примере, требования к безопасности, связанные с аутентификацией пользователей, авторизацией и конфиденциальностью данных, также ожидаются как часть нефункциональных требований. Обратите внимание, что функциональные и нефункциональные требования направлены на точное определение**, что** нужно сделать. Разработка решения — это понять**, как** оно будет реализовано. А реализация проектирования — это реальное решение на выбранном вами языке программирования. Создание дизайна, полностью удовлетворяющего как функциональным, так и нефункциональным требованиям, может потребовать много времени и усилий. Выбор подходящего языка программирования и среды разработки/производства может зависеть от требований задачи. Например, поскольку C/C++ — это язык более низкого уровня, чем Python, он может быть лучшим выбором для алгоритмов, требующих согласованного кода и низкоуровневой оптимизации. После завершения этапа проектирования и завершения кодирования алгоритм готов к развертыванию. Внедрение алгоритма включает проектирование непосредственной производственной среды, в которой будет выполняться код. Производственная среда должна быть спроектирована с учётом данных и потребностей алгоритма в обработке. Например, для параллелизируемых алгоритмов для эффективного выполнения алгоритма потребуется кластер с соответствующим количеством компьютерных узлов. Для алгоритмов, работающих с интенсивными данными, может потребоваться разработка конвейера входа данных и стратегию кэширования и хранения данных. Проектирование производственной среды подробно обсуждается в *главе 14 «Алгоритмы крупного* масштаба» и *главе 15 «Практические соображения*». После проектирования и реализации производственной среды алгоритм

внедряется, который принимает входные данные, обрабатывает их и генерирует выходные данные в соответствии с требованиями.

Среда развития

После разработки алгоритмы необходимо реализовать на языке программирования согласно проекту. Для этой книги мы выбрали язык программирования Python. Мы выбрали его, потому что Python — гибкий и открытый язык программирования. Python также является одним из языков, которые можно использовать в различных облачных инфраструктурах, таких как **Amazon Web Services (AWS**), Microsoft Azure и **Google Cloud Platform (GCP).** Официальная главная страница Python доступна по адресу https://www.p ython.org/, где также есть инструкции по установке и полезное руководство для новичков. Для лучшего понимания концепций, изложенных в этой книге, требуется базовое понимание Python. Для этой книги мы ожидаем, что вы будете использовать последнюю версию Python 3. На момент написания самой последней версией является версия 3.10, которую мы используем для выполнения упражнений в этой книге.

> Мы будем использовать Python на протяжении всей этой книги. Мы также будем использовать Jupyter Notebooks для запуска кода. Остальные главы этой книги предполагают, что Python установлен, а ноутбуки Jupyter корректно настроены и работают.

Python-пакеты

Python — это язык общего назначения. Она следует философии «батареи включены», что означает, что существует стандартная библиотека, которая доступна, без необходимости пользователя скачивать отдельные пакеты. Однако стандартные библиотечные модули предоставляют лишь минимальную функциональность. В зависимости от конкретного случая использования, над которым вы работаете, возможно, потребуется установить дополнительные пакеты. Официальный сторонний репозиторий пакетов на Python

называется PyPi, что расшифровывается как Python Package Index. Он размещает пакеты на Python как в виде исходного кода, так и в виде предварительно скомпилированного кода. В настоящее время на PyPi размещено более 113 000 пакетов на Python. Самый простой способ установить дополнительные пакеты — через систему управления пакетами pip. « `Pip` » — это занудная рекурсивная аббревиатура, широко встречающаяся в языке Python. Pip расшифровывается как «Pip Installs Python». Хорошая новость в том, что с версии 3.4 Python по умолчанию устанавливается pip. Чтобы проверить версию pip, можно ввести в командной строке: `pip --version`

Эту `команду pip` можно использовать для установки

дополнительных пакетов: `pip install 'PackageName'`

Пакеты, которые уже были установлены, требуют периодического обновления для получения последней функциональности. Это достигается с помощью флага обновления:

```
pip install 'PackageName' –upgrade
```

А чтобы установить конкретную версию пакета на

Python: `pip install 'PackageName==2.1'`

Добавление правильных библиотек и версий стало частью настройки среды программирования на Python. Одной из функций, помогающих поддерживать эти библиотеки, является возможность создания файла требований, в котором перечислены все необходимые пакеты. Файл требований — это простой текстовый файл, содержащий имена библиотек и связанные с ними версии. Пример файлов требований выглядит следующим образом:

```
scikit-learn==0.24.1

tensorflow==2.5.0

тензорборд==2.5.0
```

По конвенции `requirements.txt` размещается в верхнем каталоге проекта.

После создания файл требований можно использовать для настройки среды разработки, установив все библиотеки Python и связанные с ними версии с помощью следующей команды.

```
PIP Install -r requirements.txt
```

Теперь давайте рассмотрим основные пакеты, которые мы будем использовать в этой книге.

Экосистема SciPy

Scientific Python (SciPy) — произносится как sigh pie — это группа пакетов на Python, созданных для научного сообщества. Он содержит множество функций, включая широкий спектр генераторов случайных чисел, процедуры линейной алгебры и оптимизаторы. SciPy — это комплексный пакет, и со временем люди разработали множество расширений, чтобы настраивать и расширять пакет в соответствии со своими потребностями. SciPy работает эффективно, так как действует как тонкая обёртка вокруг оптимизированного кода, написанного на C/C++ или Fortran. Ниже приведены основные пакеты, входящие в эту экосистему:

- **NumPy**: Для алгоритмов очень важна возможность создавать многомерные структуры данных, такие как массивы и матрицы. NumPy предлагает набор типов данных массива и матриц, важных для статистики и анализа данных. Подробности о NumPy можно найти на http://www.nump y.org/.
- **scikit-learn**: Это расширение для машинного обучения — одно из самых популярных расширений SciPy. Scikit-learn предоставляет

широкий спектр важных алгоритмов машинного обучения, включая классификацию, регрессию, кластеризацию и валидацию моделей. Больше информации о scikit-learn можно найти на http://scikit-learn.org/.

- **pandas**: Pandas содержит табличную сложную структуру данных, широко используемую для ввода, вывода и обработки табличных данных в различных алгоритмах. Библиотека pandas содержит множество полезных функций и предлагает высоко оптимизированную производительность. Более подробную информацию о пандах можно найти на http://pandas.pydata.org/.
- **Matplotlib**: Matplotlib предоставляет инструменты для создания мощных визуализаций. Данные могут быть представлены в виде линейных графиков, диаграмм рассеяния, столбчатых диаграмм, гистограмм, круговых диаграмм и так далее. Больше информации можно найти на https://matplotlib.org/.

Использование Python Notebooks Мы будем использовать Jupyter Notebook и совместную лабораторию Google в качестве IDE. Подробнее о настройке и использовании Jupyter Notebooks и Colab можно найти в приложениях A и B.

Методы проектирования алгоритмов

Алгоритм — это математическое решение реальной задачи. При проектировании алгоритма мы учитываем следующие три аспекта проектирования при работе над проектированием и тонкой настройкой алгоритмов:

- **Вопрос 1**: Даёт ли этот алгоритм ожидаемый результат?
- **Вопрос 2**: Является ли это самым оптимальным способом
- получить такие результаты?

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

Важно понять сложность самой задачи, прежде чем разрабатывать её решение. Например, нам помогает разработать подходящее решение,

если мы характеризуем проблему с точки зрения её потребностей и сложности. В целом алгоритмы можно разделить на следующие типы в зависимости от характеристик задачи:

- **Алгоритмы, работающие с большим количеством** данных: алгоритмы, работающие с большим объемом данных, разработаны для работы с большим объёмом данных. Ожидается, что к ним будут относительно простые требования к обработке. Алгоритм сжатия, применяемый к огромному файлу, является хорошим примером алгоритмов, работающих с большими данными. Для таких алгоритмов размер данных ожидается значительно больше, чем память обработочного движка (одного узла или кластера), и может потребоваться разработка итеративного дизайна обработки данных для эффективной обработки данных в соответствии с требованиями.
- **Алгоритмы, интенсивные на вычислениях**: Алгоритмы с высокой нагрузкой на вычисление требуют значительных требований к обработке, но не требуют больших объёмов данных. Простой пример — алгоритм поиска очень большого простого числа. Поиск стратегии разделения алгоритма на разные фазы, чтобы хотя бы часть фаз были параллельны, является ключом к максимизации эффективности алгоритма.
- **Как алгоритмы, так и вычислительные** алгоритмы: существуют алгоритмы, которые работают с большим объёмом данных и требуют значительных вычислительных требований. Алгоритмы, используемые для анализа настроений в прямых видеопотоках, являются хорошим примером того, где и данные, и требования к обработке имеют огромное значение для выполнения задачи. Такие алгоритмы являются самыми ресурсоёмкими и требуют тщательного проектирования алгоритма и разумного распределения доступных ресурсов.

Чтобы охарактеризовать задачу с точки зрения её сложности и потребностей, полезно глубже изучить её данные и вычислить размерности, что мы сделаем в следующем разделе.

Измерение данных

Для категоризации измерения данных задачи рассматриваем её **объём**, **скорость** и **разнообразие** (**3V**), которые определяются следующим образом:

- **Объем**: Объём — это ожидаемый размер данных, которые алгоритм будет обрабатывать.
- **Скорость**: Скорость — это ожидаемая скорость генерации новых данных при использовании алгоритма. Это может быть ноль.
- **Разнообразие**: Многообразие количественно определяет, с какими типами данных алгоритм должен работать с разработкой.

На рисунке 1.5 показаны 3V данных более подробно. Центр этой диаграммы показывает самые простые возможные данные, с малым объёмом и малым разнообразием и скоростью. По мере того как мы отходим от центра, сложность данных увеличивается. Он может увеличиваться в одном или нескольких из трёх измерений. Например, в размерности скорости у нас самый простой пакетный процесс, затем периодический процесс, а затем процесс почти в реальном времени. Наконец, у нас есть процесс в реальном времени, который самый сложный для обработки в контексте скорости передачи данных. Например, коллекция прямых видеопотоков, собранных группой мониторинговых камер, будет иметь большой объём, высокую скорость и большое разнообразие и может потребовать соответствующего дизайна для эффективного хранения и обработки данных.

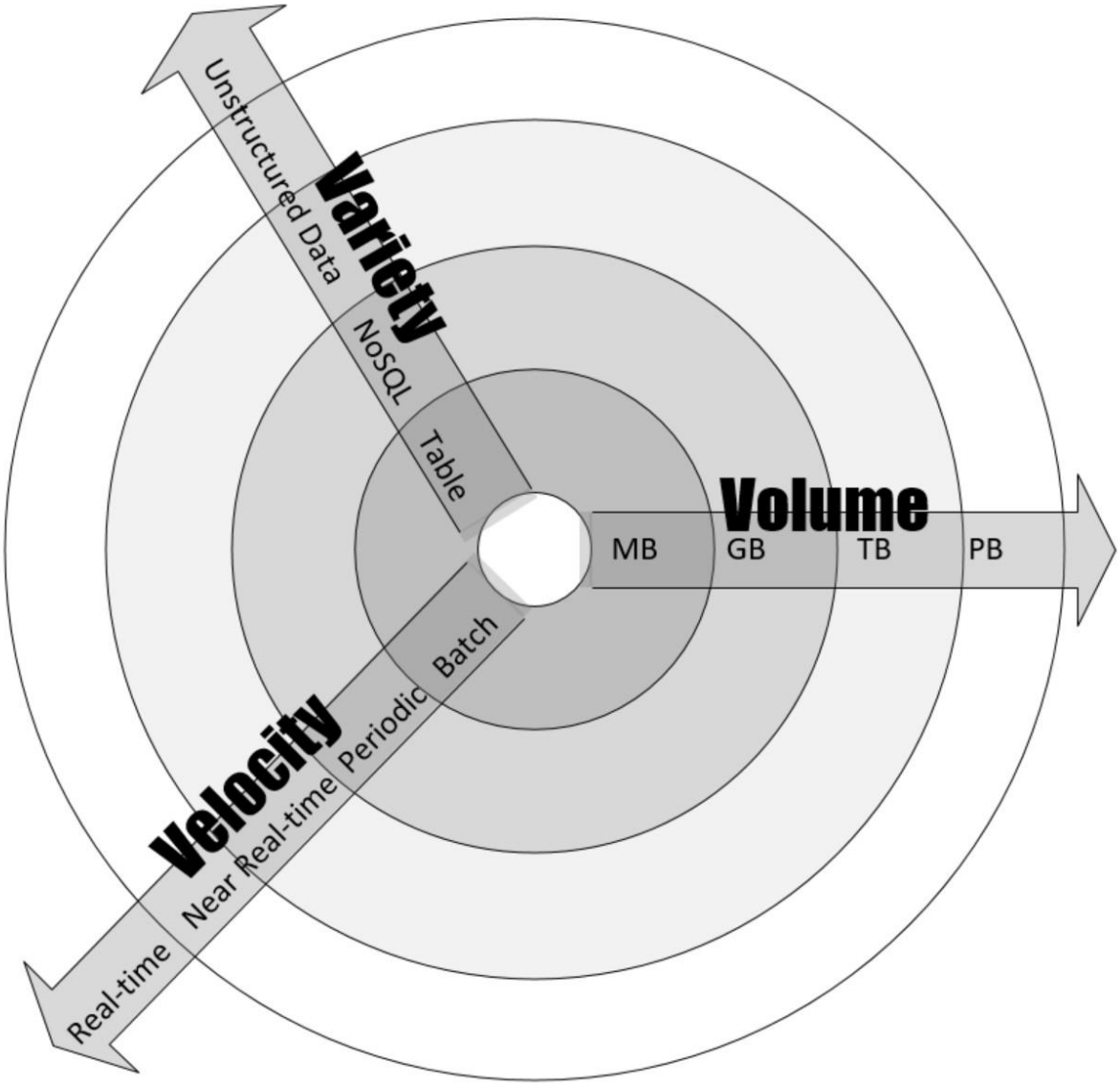


Рисунок 1.5: 3V данных: объём, скорость и разнообразие

Например, если входные данные — это простой `csv-файл`, то объём, скорость и разнообразие данных будут низкими. С другой стороны, если входные данные — это прямая трансляция видеокамеры наблюдения, то объём, скорость и разнообразие данных будут довольно высокими, и эту проблему следует учитывать при разработке алгоритма.

Вычислительная размерность

Для характеристики вычислительной размерности мы анализируем потребности в обработке конкретной задачи. Потребности алгоритма в обработке определяют, какой тип дизайна для него наиболее эффективен. Например, сложные алгоритмы в целом требуют большой вычислительной мощности. Для таких алгоритмов может быть важно иметь многоузловую параллельную архитектуру. Современные глубокие алгоритмы обычно требуют значительной численной обработки и могут нуждаться в мощности GPU или TUP, как обсуждается в *главе 15*, *Практические соображения*.

Анализ эффективности

Анализ производительности алгоритма является важной частью его проектирования. Один из способов оценки эффективности алгоритма — это анализ его сложности. Теория сложности — это изучение того, насколько сложны алгоритмы. Чтобы быть полезным, любой алгоритм должен обладать тремя ключевыми особенностями:

- **Должно быть правильным**: хороший алгоритм должен давать правильный результат. Чтобы подтвердить, что алгоритм работает корректно, его необходимо тщательно протестировать, особенно при тестировании крайних случаев.
- **Должно быть понятно**: хороший алгоритм должен быть понятен. Лучший алгоритм в мире не очень полезен, если его слишком сложно реализовать на компьютере.
- **Должен быть эффективным**: Хороший алгоритм должен быть эффективным. Даже если алгоритм даёт правильный результат, он мало поможет, если это займёт тысячу лет или потребует 1 миллиард терабайт памяти.

Существует два возможных типа анализа для количественной оценки сложности алгоритма:

- **Анализ пространственной сложности**: оценивает потребности в памяти во время выполнения алгоритма.
- **Анализ сложности** по времени: оценивает время, которое потребуется алгоритму для запуска.

Давайте изучим их по одному:

Анализ пространственной сложности

Анализ пространственной сложности оценивает объем памяти, необходимый алгоритму для обработки входных данных. При обработке входных данных алгоритм должен хранить временные структуры данных в памяти. Способ проектирования алгоритма влияет на количество, тип и размер этих структур данных. В эпоху распределённых вычислений и всё больших объёмов данных, требующих обработки, анализ пространственной сложности становится всё более значимым. Размер, тип и количество этих структур данных определяют требования к памяти для базового оборудования. Современные структуры данных в памяти, используемые в распределённых вычислениях, должны иметь эффективные механизмы распределения ресурсов, учитывающие требования к памяти на разных этапах выполнения алгоритма. Сложные алгоритмы, как правило, имеют итеративный характер. Вместо того чтобы собрать всю информацию в память сразу, такие алгоритмы итеративно заполняют структуры данных. Для вычисления пространственной сложности важно сначала классифицировать тип итеративного алгоритма, который мы планируем использовать. Итеративный алгоритм может использовать один из следующих трёх типов итераций.

- **Сходящиеся итерации**: по мере прохождения алгоритмом объём обрабатываемых ими данных в каждой отдельной итерации уменьшается. Другими словами, пространственная сложность уменьшается по мере прохождения алгоритма через свои итерации. Главная задача — справиться с пространственной сложностью начальных итераций. Современные

масштабируемые облачные инфраструктуры, такие как AWS и Google Cloud, лучше всего подходят для выполнения таких алгоритмов.

- **Расходящиеся итерации**: по мере прохождения алгоритмом объём обрабатываемых ими данных в каждой отдельной итерации увеличивается. По мере роста пространственной сложности по мере прогресса алгоритма через итерации важно устанавливать ограничения, чтобы предотвратить нестабильность системы. Ограничения можно устанавливать, ограничивая количество итераций и/или устанавливая лимит размера начальных данных.
- **Плоские итерации**: по мере прохождения алгоритмом объём обрабатываемых ими данных в каждой отдельной итерации остаётся постоянным. Поскольку сложность пространства не меняется, эластичность инфраструктуры не нужна.

Чтобы рассчитать пространственную сложность, нужно сосредоточиться на одной из самых сложных итераций. Например, для алгоритма, использующего конвертирующие итерации, мы можем выбрать одну из начальных итераций. После выбора мы оцениваем общий объём используемой алгоритмом памяти, включая память, используемую его временными структурами данных, выполнение и входные значения.
Это даст нам хорошую оценку пространственной сложности алгоритма. Ниже приведены рекомендации по минимизации сложности пространства:

- По возможности старайтесь проектировать алгоритм как
- итеративный. При проектировании итеративного алгоритма, когда есть выбор, предпочитайте большее количество итераций, а не меньшее. Ожидается, что при мелкозернистом большем количестве итераций будет меньше пространственной сложности.
- Алгоритмы должны передавать в память только ту информацию, необходимую для текущей обработки. Всё, что не нужно, должно быть вымыто из памяти.

Анализ пространственной сложности необходим для эффективного проектирования алгоритмов. Если при проектировании конкретного алгоритма не проводится правильный анализ пространственной сложности, недостаток доступности памяти для временных структур данных может привести к ненужным переливам диска, что может существенно повлиять на производительность и эффективность алгоритма. В этой главе мы глубже рассмотрим временную сложность. Пространственная сложность будет рассмотрена в *главе 14*, *«Крупномасштабные алгоритмы»*, более подробно, где мы рассмотрим крупномасштабные распределённые алгоритмы с сложными требованиями к памяти во время выполнения.

Анализ временной сложности

Анализ временной сложности оценивает, сколько времени потребуется алгоритму для выполнения назначенной задачи, исходя из его структуры. В отличие от пространственной сложности, временная сложность не зависит от аппаратного обеспечения, на котором будет работать алгоритм. Анализ временной сложности полностью зависит от структуры самого алгоритма. Общая цель анализа временной сложности — попытаться ответить на эти два важных вопроса: будет ли этот алгоритм масштабироваться? Хорошо спроектированный алгоритм должен быть полностью способен использовать современную эластичную инфраструктуру, доступную в облачных вычислениях. Алгоритм должен быть спроектирован так, чтобы он мог использовать доступ к большему количеству процессоров, вычислительных ядер, GPU и памяти. Например, алгоритм, используемый для обучения модели в задаче машинного обучения, должен уметь использовать распределённое обучение по мере увеличения числа процессоров. Такой алгоритм также должен использовать GPU и дополнительную память, если они доступны во время выполнения алгоритма. Насколько хорошо этот алгоритм справится с большими наборами данных? Чтобы ответить на эти вопросы, нам нужно определить влияние на производительность алгоритма по мере увеличения размера данных и убедиться, что алгоритм спроектирован так, чтобы он был не только точным, но и

хорошо масштабировался. Производительность алгоритма становится всё более важной для больших наборов данных в современном мире «больших данных». Во многих случаях у нас может быть несколько подходов к проектированию алгоритма. Цель проведения анализа временной сложности в данном случае будет следующей: *«Имея определённую задачу и более одного алгоритма, какой из них наиболее эффективен с точки зрения эффективности по времени?»*Существует два основных подхода к вычислению временной сложности алгоритма:

- **Подход профилирования после внедрения**: в этом подходе реализуются различные кандидаты алгоритмов и сравниваются их эффективность.
- **Теоретический подход до реализации**: в этом подходе производительность каждого алгоритма математически оценивается перед запуском алгоритма.

Преимущество теоретического подхода в том, что он зависит только от структуры самого алгоритма. Это не зависит от конкретного аппаратного обеспечения, которое будет использоваться для запуска алгоритма, выбора программного стека, выбранного во время выполнения, или языка программирования, используемого для реализации алгоритма.

Оценка производительности

Производительность типичного алгоритма будет зависеть от типа данных, которые ему предоставляются на вход. Например, если данные уже отсортированы в соответствии с контекстом задачи, которую мы пытаемся решить, алгоритм может работать очень быстро. Если отсортированный вход используется для бенчмаркинга этого конкретного алгоритма, то он даст нереалистично высокий показатель производительности, который не будет точным отражением его реальной производительности в большинстве сценариев. Чтобы справиться с этой зависимостью алгоритмов от

входных данных, у нас есть разные типы случаев, которые нужно учитывать при проведении анализа производительности.

Лучший вариант

В лучшем случае данные, предоставляемые в качестве входа, организованы так, чтобы алгоритм обеспечивал наилучшую производительность. Анализ в лучшем случае даёт верхнюю границу производительности.

Худший случай

Второй способ оценить производительность алгоритма — попытаться найти максимально возможное время, необходимое для выполнения задачи при заданном наборе условий. Этот анализ алгоритма в худшем случае весьма полезен, так как мы гарантируем, что независимо от условий производительность алгоритма всегда будет лучше, чем числа, полученные в результате нашего анализа. Анализ в худшем случае особенно полезен для оценки производительности при решении сложных задач с большими наборами данных. Анализ в худшем случае даёт нижнюю границу эффективности алгоритма.

Обычный случай

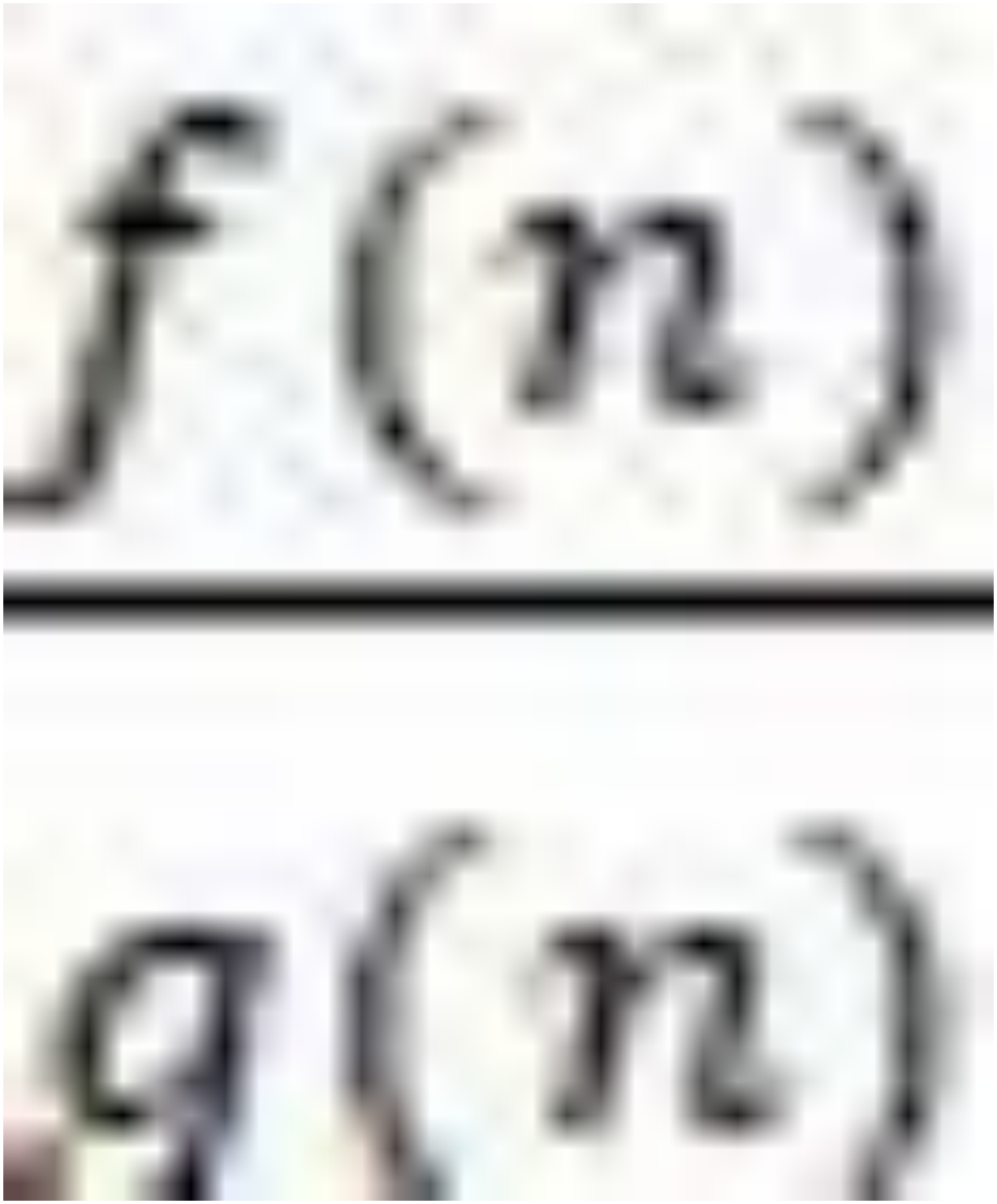
Это начинается с разделения различных возможных входных данных на различные группы. Затем проводится анализ эффективности на основе одного из представительных входных данных каждой группы. Наконец, он рассчитывает среднее значение результатов каждой из групп. Средний анализ случаев не всегда точен, так как необходимо учитывать все различные комбинации и возможности ввода в алгоритм, что не всегда легко сделать.

Выбор алгоритма

Как понять, какое решение лучше? Как понять, какой алгоритм работает быстрее? Анализ временной сложности алгоритма может ответить на такие вопросы. Чтобы понять, где это может быть полезно, рассмотрим простой пример, где цель — отсортировать список чисел. Существует множество алгоритмов, которые могут справиться с этой задачей. Вопрос в том, как выбрать правильную. Во-первых, можно сделать наблюдение: если в списке не слишком много чисел, то неважно, какой алгоритм мы выберем для сортировки чисел. Таким образом, если в списке всего 10 чисел (n=10), то неважно, какой алгоритм мы выберем, так как даже при очень простом алгоритме это займёт не больше нескольких микросекунд. Но по мере увеличения n выбор правильного алгоритма начинает иметь значение. Плохо спроектированный алгоритм может занять пару часов, а хорошо продуманный алгоритм может закончить сортировку списка за пару секунд. Поэтому для больших входных наборов имеет смысл потратить время и усилия, провести анализ производительности и выбрать правильно разработанный алгоритм, который выполнит необходимую задачу эффективно.

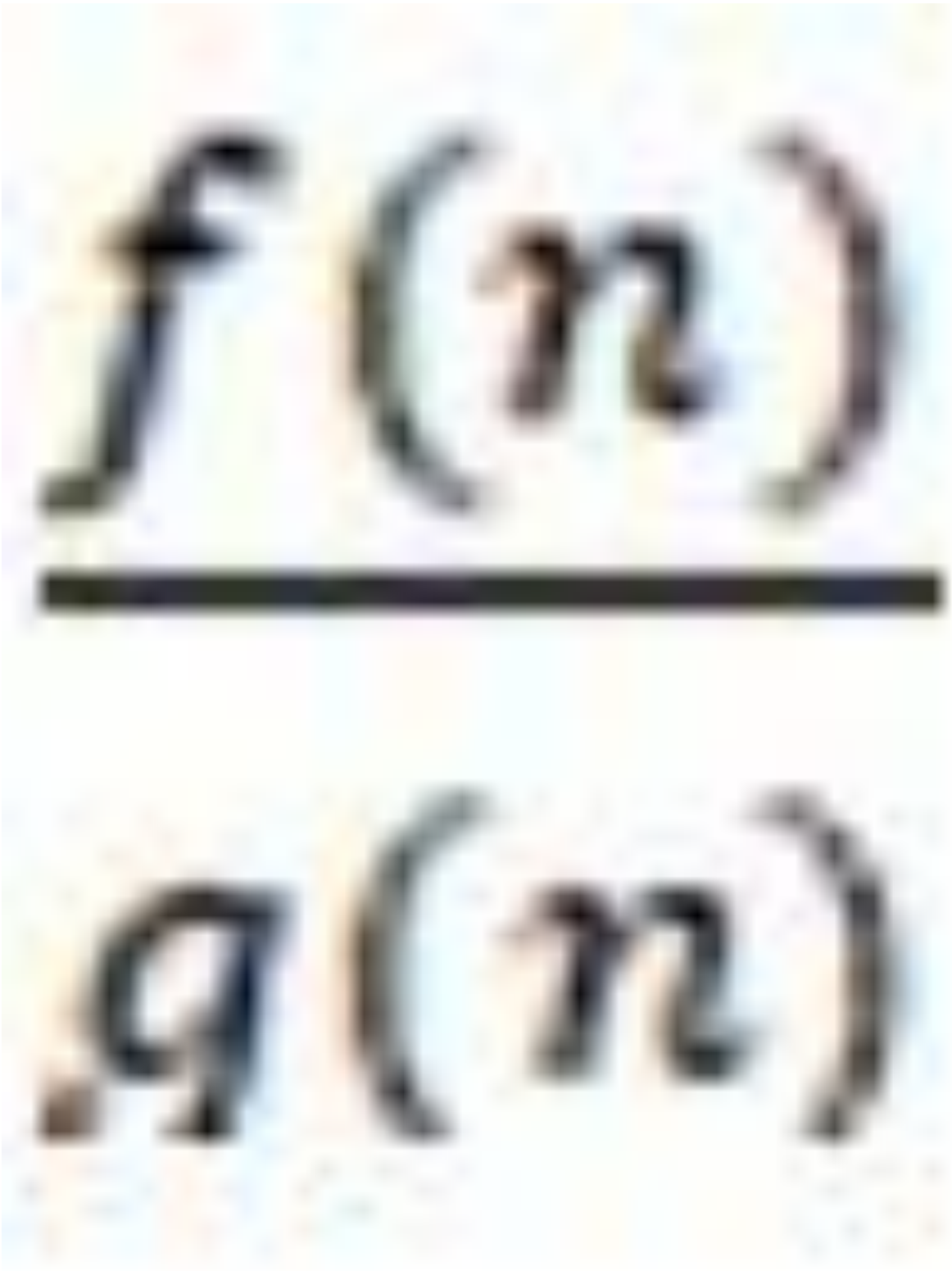
Нотация Big O

Нотация Big O была впервые введена Бахманом в 1894 году в научной работе для приближения роста алгоритма. Он писал: «... с помощью символа O(n) мы выражаем величину, порядок которой относительно n не превышает порядка n" (Bachmann 1894, с. 401)". Таким образом, нотация Big-O измеряет асимптотическую производительность алгоритма. Чтобы объяснить, как мы используем обозначение Big-O, рассмотрим две функции: f(n) и g(n) — как функции. Можно сказать, что f = O(g) тогда и только когда n увеличивается до бесконечности, если

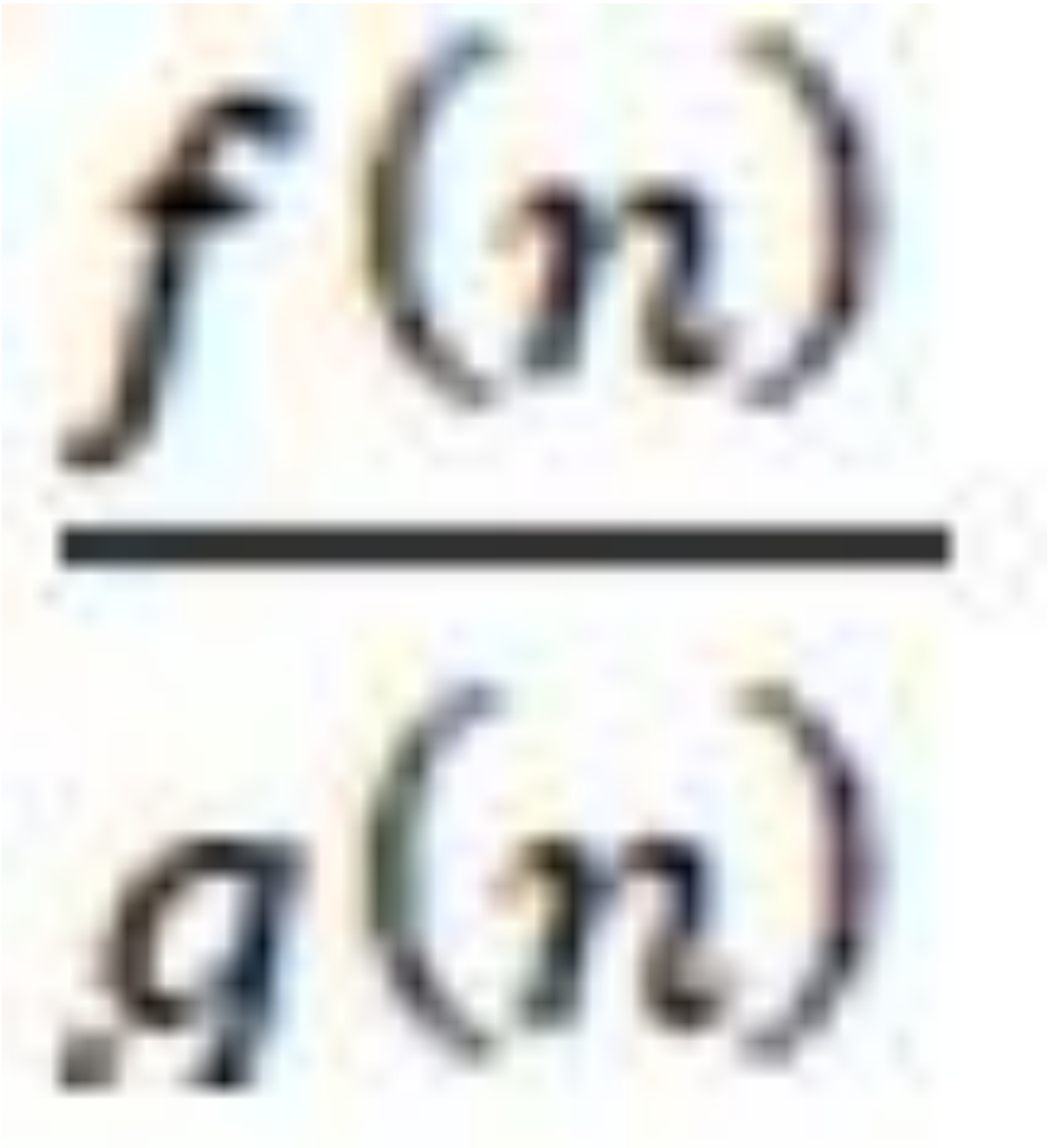


ограничен. Рассмотрим конкретную функцию: f(n) = 1000n2 + 100n + 10 и g(n) = n2. Обратите внимание, что обе функции будут приближаться к бесконечности, когда n приближается к бесконечности. Давайте выясним, что f = O(g), применив определение.

Сначала пусть вычислим



которая будет равна



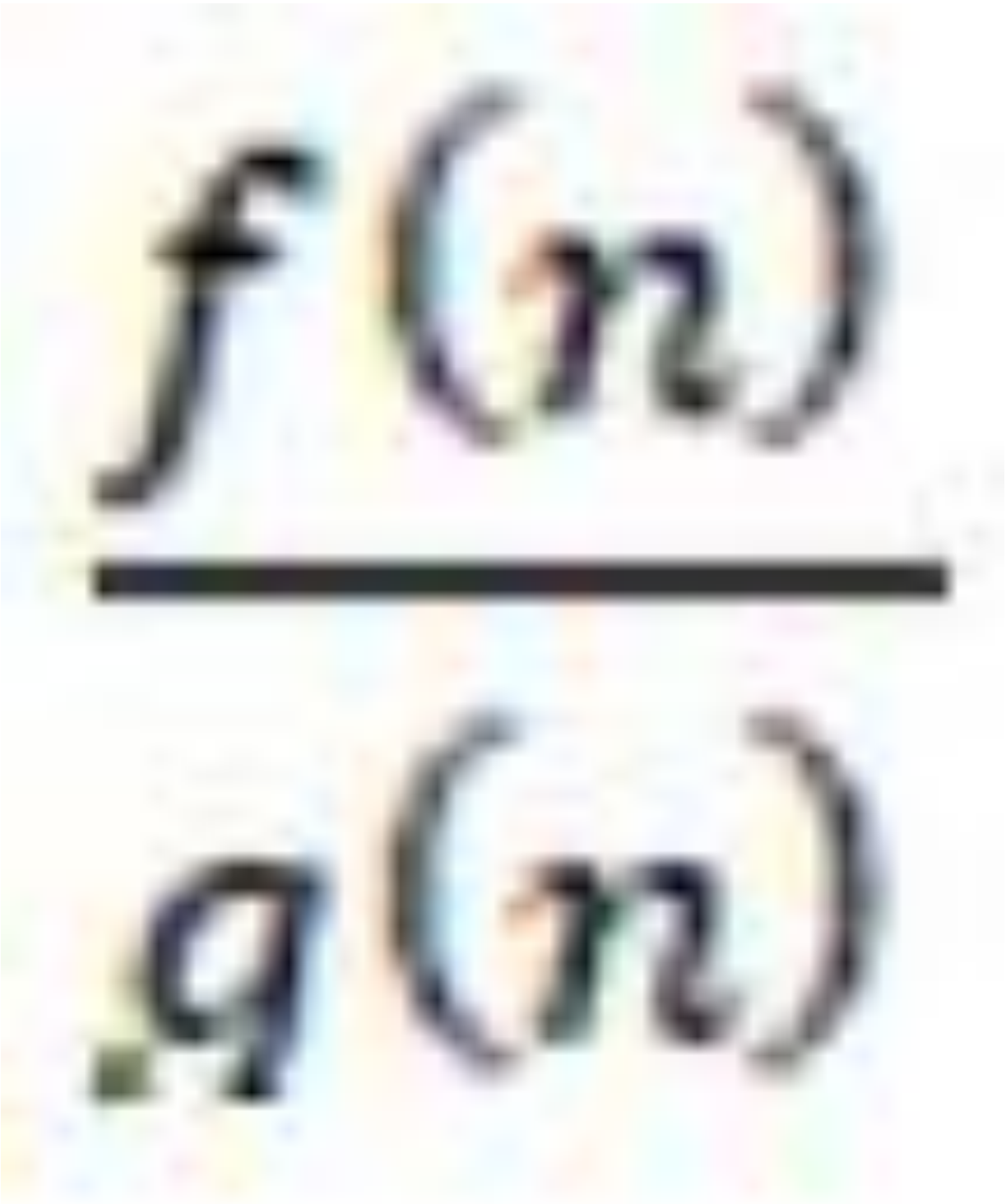
=

$$\frac{1000n^2+100n+10}{n^2}$$

= (1000+

$$\frac{100}{n}+\frac{10}{n^2}$$

)Очевидно, что



ограничен и не будет приближаться к n, приближающейся к бесконечности. Таким образом, f(n) = O(g) = O(n2). (n2) обозначает, что сложность этой функции увеличивается по мере квадрата входов n.

Если удвоить количество входных элементов, ожидается, что сложность увеличится на 4. Обратите внимание на следующие 4 правила при работе с нотацией Big-O.**Правило 1:** Давайте рассмотрим сложность циклов в алгоритмах. Если алгоритм выполняет определённую последовательность шагов n раз на производительность O(n). **Правило 2:** Давайте рассмотрим вложенные циклы алгоритмов. Если алгоритм выполняет функцию с циклом из n1 шагов, а для цикла выполняет ещё n2 шага, общая производительность алгоритма равна O(n1 × n2). Например, если алгоритм представляет собой внешние и внутренние петли с n шагов, то сложность алгоритма будет представлена как O(n*n) = O(n2)**Правило 3:** Если алгоритм выполняет функцию f(n), которая выполняет n1 шага, а затем выполняет другую функцию g(n) — n2 шага, то общая производительность алгоритма равна O(f(n)+g(n)).**Правило 4:** Если алгоритм принимает O(g(n) + h(n)), а функция g(n) больше больше(n) для большого n, производительность алгоритма можно упростить до O(g(n)). Это означает, что O(1+n) = O(n) и O(n2+ n3) = O(n2) **Правило 5:** При расчете сложности алгоритма игнорируйте постоянные кратные. Если k — постоянная, то O(kf(n)) равна O(f(n)). Также O(f(k × n)) — это то же самое, что O(f(n)). Таким образом, O(5n2) = O(n2)И O((3n2)) = O(n2)Обратите внимание, что:

- Сложность, количественно измеряемая нотацией Big O, является
- лишь оценкой. Для небольших объёмов данных нас не волнует временная сложность. n0 на графике определяет порог, выше которого нам интересно найти временную сложность. Затенённая область описывает эту область интереса, где мы анализируем временную сложность.
- T(n) временная сложность больше, чем исходная функция. Хороший выбор T(n) попытается создать чёткую верхнюю границу для F(n)

Следующая таблица суммирует различные типы обозначения Big O, обсуждаемые в этом разделе:

Название класса сложности Примеры операций

O(1) КонстантаДобавить, получить предмет, задать элемент.
O(logn) Логарифмические Поиск элемента в отсортированном массиве. O(n) Линейный копировать, вставлять, удалять, итерировать nLogn Линейно-логарифмическая сортировка списка, слияние — сортировка.
n2 Квадратичный Вложенные петли

Сложность постоянного времени (O(1))

Если алгоритм выполняется одинаково, независимо от размера входных данных, говорят, что он работает за постоянное время. Он представлен как O(1). Возьмём пример доступа к n-му элементу массива. Независимо от размера массива, получение результатов займёт постоянное время. Например, следующая функция вернёт первый элемент массива и имеет сложность O(1):

```
[10]  1 def getFirst(my_list):
      2     return my_list[0]
```

The output is shown as:

```
[11]  1 getFirst([1, 2, 3])

      1
```

```
[12]  1 getFirst([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

      1
```

- Добавление нового элемента в стек с помощью push или удаление

элемента из стека с помощью pop. Независимо от размера стека, добавление или удаление элемента займёт одинаковое время.
- Доступ к элементу хэш-таблицы (как обсуждается во *второй главе «Структуры данных, используемые в алгоритмах»).*
- Сортировка по ведру (как обсуждается во *второй главе «Структуры данных, используемые в алгоритмах»).*

Сложность линейного времени (O(n))

Алгоритм называется имеющим сложность линейного времени, представленную O(n), если время выполнения прямо пропорционально размеру входного сигнала. Простой пример — добавить элементы в одномерную структуру данных:

```
1 def get_sum(my_list):
2     sum = 0
3     for item in my_list:
4         sum = sum + item
5     return sum
```

Обратите внимание на основной цикл алгоритма. Количество итераций в главной петле увеличивается линейно с увеличением значения n, что приводит к O(n) сложности на следующем рисунке:

```
1 get_sum([1, 2, 3])
```

```
6
```

```
1 get_sum([1, 2, 3, 4])
```

```
10
```

Некоторые другие примеры операций с массивами следующие:

- Поиск элемента
- Нахождение минимального значения среди всех элементов массива

Сложность квадратичного времени (O(n2))

Говорят, что алгоритм работает в квадратичном времени, если время выполнения алгоритма пропорционально квадрату размера входа; например, простую функцию, которая суммирует двумерный массив следующим образом:

```
1 def get_sum(my_list):
2     sum = 0
3     for row in myList:
4         for item in row:
5             sum += item
6     return sum
```

Обратите внимание на вложенную внутреннюю петлю внутри другой основной петли. Этот вложенный цикл придаёт предыдущему коду сложность O(n2):

```
[13]  1 get_sum([[1, 2], [3, 4]])

      10
```

```
[15]  1 get_sum([[1 ,2 , 3],[4 , 5, 6]])

      21
```

Другой пример — **алгоритм сортировки пузырьков** (как описано во *второй главе «Структуры данных, используемые в алгоритмах»).*

Сложность логарифмического времени (O(logn))

Алгоритм считается работающим за логарифмическое время, если время выполнения алгоритма пропорционально логарифму размера входа. С каждой итерацией размер входа уменьшается на постоянный множитель. Примером логарифмического алгоритма является двоичный поиск. Бинарный алгоритм поиска используется для поиска определённого элемента в одномерной структуре данных, например, в списке на Python. Элементы в структуре данных должны быть отсортированы в убывающем порядке. Бинарный алгоритм поиска реализован в функции с названием searchBinary, следующим образом:

```
1  def search_binary(my_list, item):
2      first = 0
3      last = len(my_list)-1
4      found_flag = False
5      while(first <= last and not found_flag):
6          mid = (first + last)//2
7          if my_list[mid] == item:
8              found_flag = True
9          else:
10             if item < my_list[mid]:
11                 last = mid - 1
12             else:
13                 first = mid + 1
14     return found_flag
```

Основная петля использует тот факт, что список упорядочен. Он делит список пополам с каждой итерацией, пока не достигнет результата: После определения функции тестируется поиск определённого элемента в строках 11 и 12. Алгоритм бинарного поиска подробно рассматривается в *главе 3, «Алгоритмы сортировки и поиска»*. Обратите внимание, что из четырёх типов нотации Big O O(n2) имеет

худшую производительность, а O(logn) — лучшую. Фактически, производительность O(logn) можно считать золотым стандартом для работы любого алгоритма (хотя это не всегда достигается). С другой стороны, O(n2) не так плох, как O(n3), но алгоритмы, относящиеся к этому классу, нельзя использовать для больших данных, поскольку временная сложность ограничивает объем данных, которые они реально могут обработать. Один из способов снизить сложность алгоритма — это компромисс с его точностью, создавая тип алгоритма, называемый **приближённым алгоритмом**.

Валидация алгоритма

Проверка алгоритма подтверждает, что он действительно предоставляет математическое решение задачи, которую мы пытаемся решить. Процесс валидации должен проверять результаты на максимальное количество возможных значений и типов входных значений.

Точные, приближённые и рандомизированные алгоритмы

Валидация алгоритма также зависит от типа алгоритма, так как методы тестирования отличаются. Давайте сначала различаем детерминированные и рандомизированные алгоритмы. Для детерминированных алгоритмов определённый вход всегда генерирует точно одинаковый выход. Но для некоторых классов алгоритмов также принимается последовательность случайных чисел, что делает вывод разным при каждом запуске алгоритма. Алгоритм кластеризации kmeans, подробно описанный в *главе 6 «Алгоритмы неконтролируемого машинного обучения»*, является примером такого алгоритма:

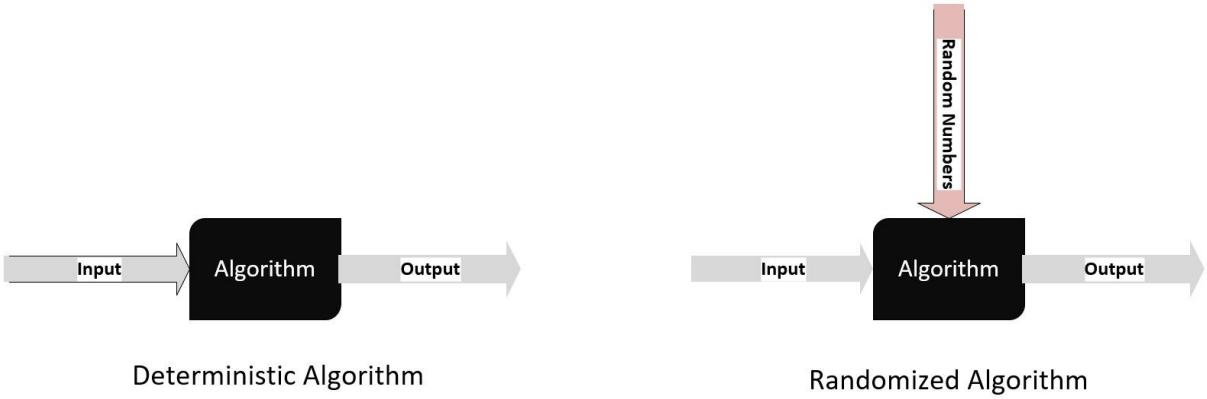


Рисунок 1.2 Детерминированные и рандомизированные алгоритмы

Алгоритмы также можно разделить на следующие два типа на основе предположений или приближений, используемых для упрощения логики и ускорения их работы:

- **Точный алгоритм**: От точные алгоритмы ожидается, что они дадут точное решение без введения предположений или приближений.

Приближённый алгоритм: когда сложность задачи слишком высока для заданных ресурсов, мы упрощаем задачу, делая некоторые предположения. Алгоритмы, основанные на этих упрощениях или предположениях, называются приближёнными алгоритмами, что не даёт точного решения. Давайте рассмотрим пример, чтобы понять разницу между точными и приближёнными алгоритмами — знаменитую задачу коммивояжера, которая была представлена в 1930 году. Коммивояжер задаёт вам задачу найти самый короткий маршрут для конкретного коммивояжера, который посещает каждый город (из списка городов), а затем возвращается в источник, поэтому его называют коммивояжером. Первая попытка найти решение будет включать генерацию всех перестановок городов и выбор самой дешёвой комбинации городов. Очевидно, что временная сложность начинает становиться неуправляемой за пределами 30 городов. Если число городов превышает 30, одним из способов снижения сложности является введение некоторых приближений и предположений. Для приближённых алгоритмов важно устанавливать ожидания точности при сборе требований. Валидация алгоритма аппроксимации

заключается в проверке того, что ошибка результатов находится в приемлемом диапазоне.

Объяснимость

Когда алгоритмы используются для критических случаев, становится важно иметь возможность объяснять причины каждого результата, когда это необходимо. Это необходимо для того, чтобы решения, основанные на результатах алгоритмов, не приводили к предвзятости. Способность точно определить признаки, которые используются прямо или косвенно для принятия конкретного решения, называется объяснимостью алгоритма. Алгоритмы, используемые в критических случаях использования, должны оцениваться на предмет предвзятости и предвзятости. Этический анализ алгоритмов стал стандартной частью процесса валидации тех алгоритмов, которые могут влиять на принятие решений, касающихся жизни людей. Для алгоритмов, работающих с глубоким обучением, объяснимость трудно достичь. Например, если алгоритм используется для отказа в ипотечной заявке человека, важно обладать прозрачностью и способностью объяснить причину. Алгоритмическая объяснимость — активная область исследований. Одной из недавно разработанных эффективных методов **являются Локальные интерпретируемые объяснения моделей** (**LIME),** предложенные в материалах 22-й **Ассоциации вычислительной техники** (**ACM**) при **Специальной группе по открытию знаний** (**SIGKDD**)) Международная конференция по поиску знаний и анализу данных в 2016 году. LIME основан на концепции, при которой входные данные для каждого экземпляра индуцируются небольшие изменения, а затем предпринимаются попытки отображить локальную границу решения для этого экземпляра. Затем он может количественно оценить влияние каждой переменной для данного случая.

Краткое содержание

Эта глава посвящена изучению основ алгоритмов. Сначала мы узнали о различных этапах разработки алгоритма. Мы обсудили различные

способы определения логики алгоритма, необходимые для его проектирования. Затем мы изучали, как проектировать алгоритм. Мы изучили два разных способа анализа эффективности алгоритма. Наконец, мы изучали различные аспекты валидации алгоритма. После изучения этой главы мы должны сможем понять псевдокод алгоритма. Мы должны понимать разные этапы разработки и внедрения алгоритма. Мы также научились использовать нотацию Big O для оценки производительности алгоритма. Следующая глава посвящена структурам данных, используемым в алгоритмах. Начнём с рассмотрения структур данных, доступных в Python. Затем мы рассмотрим, как можно использовать эти структуры данных для создания более сложных структур данных, таких как стеки, очереди и деревья, необходимые для разработки сложных алгоритмов.

OceanfPDF.com

2 Структуры данных, используемые в алгоритмах

Присоединяйтесь к Discord-пространству нашей книги

https:// packt.link/40Algos



Алгоритмам нужны необходимые структуры данных в памяти, которые могут хранить временные данные во время выполнения. Выбор правильных структур данных крайне важен для их эффективной реализации. Некоторые классы алгоритмов являются рекурсивными или итеративными в логике и требуют специально разработанных структур данных. Например, рекурсивный алгоритм может быть реализован проще, демонстрируя лучшую производительность, если использовать вложенные структуры данных. В этой главе структуры данных рассматриваются в

контексте алгоритмов. Поскольку мы используем Python в этой книге, эта глава сосредоточена на структурах данных Python, но представленные в ней концепции могут использоваться и в других языках, таких как Java и C++. К концу этой главы вы должны понять, как Python обрабатывает сложные структуры данных и какие из них следует использовать для определённого типа данных. Поэтому вот основные моменты, обсуждаемые в этой главе:

- Изучение встроенных типов данных в Python
- Использование рядов и датафреймов
- Изучение матриц и матричных операций
- Понимание абстрактных типов данных (ADT)

Изучение встроенных типов данных в Python

В любом языке структуры данных используются для хранения и обработки сложных данных. В Python структуры данных — это контейнеры для эффективного управления, организации и поиска данных. Они используются для хранения группы элементов данных, называемых коллекциями, которые необходимо хранить и обрабатывать вместе. В Python важные структуры данных, которые могут использоваться для хранения коллекций, приведены в *таблице 2.1*:

Данные Структура	Краткое объяснение	Пример
Списки	Упорядоченная, возможно вложенная, изменяемая последовательность элементы.	["John", 33,"Toronto", True]
Кортеж	Упорядочённая неизменная последовательность элементов	(«Красный», «Зелёный», «Синий», «Жёлтый»)
Словарь	Неупорядоченный набор пар ключ-значение	{'food': 'spam', 'taste': 'yum'}
Декорации	Неупорядоченный набор элементов	{'a', 'b', 'c'}

Таблица 2.1: Структуры данных Python

Давайте рассмотрим их подробнее в предстоящих подразделах.

Списки

В Python список — это основной тип данных, используемый для хранения изменяемой последовательности элементов. Последовательность элементов, хранящихся в списке, не обязательно должна быть одного типа. Список можно определить, заключив элементы в [ ], и их нужно разделить запятой. Например, следующий код создаёт вместе четыре элемента данных разных типов:

```
>>> list_a = [«Джон», 33, «Торонто», True]
>>> print(list_a)
[«Джон», 33 года, «Торонто», True]
```

В Python список — это удобный способ создания одномерных записываемых структур данных, которые особенно необходимы на различных внутренних этапах алгоритмов.

Использование списков

Утилитарные функции в структурах данных делают их очень полезными, так как их можно использовать для управления данными в списках. Давайте посмотрим, как их можно использовать:Индексация списка: Поскольку положение элемента в списке детерминировано, индекс может использоваться для получения элемента на определённой позиции. Следующий код демонстрирует эту концепцию:

```
>>> bin_colors=['Красный', 'Зелёный', 'Синий', 'Жёлтый']
>>> bin_colors[1]
«Зелёный»
```

Четырёхэлементный список, созданный этим кодом, показан на *рисунке 2.1*:

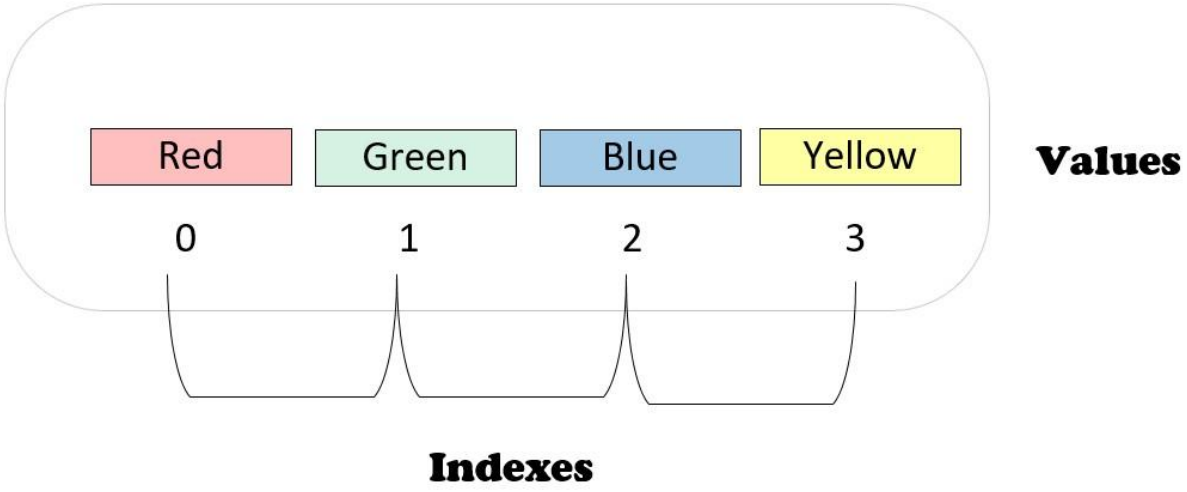


Рисунок 2.1: Четырёхэлементный список в Python

Обратите внимание, что в Python — это язык с нулевым индексированием. Это значит, что начальный индекс любой структуры данных, включая списки, будет равен 0. А `«Зелёный »`, то есть второй элемент, получается индексом `1` , то есть `bin_color[1]` . Слайсинг списков: Получение подмножества элементов списка путём указания диапазона индексов называется слайсингом. Следующий код может быть использован для создания среза списка:

```
>>> bin_colors=['Красный', 'Зелёный', 'Синий', 'Жёлтый']
```

```
>>> bin_colors[0:2]
['Красный', 'Зелёный']
```

Обратите внимание, что списки — одна из самых популярных одномерных структур данных в Python.

При нарезке списка диапазон обозначается следующим образом: первое число (включительно) и второе число (эксклюзивно). Например, `bin_colors[0:2]` будет
включайте `bin_color[0]` и `bin_color[1]`, но не `bin_color[2]`. При использовании списков это стоит помнить, так как некоторые пользователи языка Python жалуются, что это не очень интуитивно.

Давайте посмотрим на следующий фрагмент кода:

```
>>> bin_colors=['Красный', 'Зелёный', 'Синий', 'Жёлтый']
>>> bin_colors[2:]
['Синий', 'Жёлтый']
>>> bin_colors[:2]
['Красный', 'Зелёный']
```

Если начальный индекс не указан, это означает начало списка, а если конечный индекс не указан — это конец списка, как показано в предыдущем коде.
Отрицательная индексация: В Python также есть отрицательные индексы, которые учитываются из конца списка. Это продемонстрировано в следующем коде:

```
>>> bin_colors=['Красный', 'Зелёный', 'Синий', 'Жёлтый']
>>> bin_colors[:-1]
['Красный', 'Зелёный', 'Синий']
>>> bin_colors[:-2]
['Красный', 'Зелёный']
>>> bin_colors[-2:-1]
['Blue']
```

Обратите внимание, что отрицательные индексы особенно полезны, когда мы хотим использовать последний элемент в качестве точки отсчёта вместо первого.
Вложение: Элемент списка может быть любого типа данных. Это позволяет вложение в списки. Для итеративных и рекурсивных алгоритмов это обеспечивает важные возможности. Давайте рассмотрим следующий код, который является примером списка внутри списка (вложение):

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

Итерация: Python позволяет перебирать каждый элемент списка с помощью `цикла for` . Это продемонстрировано в следующем примере:

```
>>> bin_colors=['Красный', 'Зелёный', 'Синий', 'Жёлтый']
>>> для this_color в bin_colors: print(f"{this_color}
Square)
Красная площадь
Зелёная площадь
Голубая площадь
Жёлтый квадрат
```

Обратите внимание, что предыдущий код проходит по списку и печатает каждый элемент.

Функция дальности

Функцию диапазона можно использовать для создания большого списка чисел. Он используется для автозаполнения последовательностей чисел в списке. Функция `дальности` проста в использовании. Мы можем использовать его, просто указав количество нужных элементов в списке. По умолчанию он начинается с нуля и увеличивается на единицу:

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

Также можно указать конечное число и шаг:

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

Предыдущая функция диапазона даст нам нечётные числа, начинающиеся от 3 до 29.

Временная сложность списков

Временную сложность различных функций списка можно резюмировать следующим образом с помощью обозначения Big O:

Разные методы	Временная сложность
Вставьте элемент	O(1)
Удалить элемент	O(n) (в худшем случае может придётся повторять весь список)
Вырезание списка	O(n)
Извлечение элементов	O(n)
Принято	O(n)

Обратите внимание, что время, необходимое для добавления отдельного элемента, не зависит от размера списка. Другие операции, указанные в таблице, зависят от размера списка. По мере роста списка влияние на производительность становится всё более заметным.

Кортежи

Вторая структура данных, которая может использоваться для хранения коллекции, — это кортеж. В отличие от списков, кортежи — это неизменные (только для чтения) структуры данных. Кортежи состоят из нескольких элементов, окружённых ( ). Как и списки, элементы внутри кортежа могут быть разных типов. Они также допускают сложные типы данных для своих элементов. Таким образом, внутри кортежа может быть кортеж, предоставляющий способ создания вложенной структуры данных. Возможность создавать вложенные структуры данных особенно полезна в итеративных и рекурсивных алгоритмах. Следующий код демонстрирует, как создавать кортежи:

```
>>> bin_colors=(«Красный», «зелёный», «Синий», «Жёлтый»)
>>> bin_colors[1]
«Зелёный»
>>> bin_colors[2:]
(«Синий», «Жёлтый»)
>>> bin_colors[:-1]
(«Красный», «Зелёный», «Синий»)
# Вложенная структура данных
кортежа >>> a =
(1,2,(100,200,300),6)
>>> max(a[2])
300
>>> a[2][1]
200
```

По возможности следует предпочитать неизменяемые структуры данных (например, кортежи) перед изменяемыми структурами данных (например, списками) из-за высокой производительности. Особенно при работе с большими данными, неизменяемые структуры данных значительно быстрее изменяемых. Когда структура данных передаётся функции как неизменяемая, её копия не требуется, так как функция не может её изменить. Таким образом, выход может относиться к структуре входных данных. Это называется референтной прозрачностью и улучшает производительность. За возможность менять элементы данных в списках стоит платить, и нужно тщательно анализировать, действительно ли это необходимо, чтобы реализовать код в виде только читаемых кортежей, что будет гораздо быстрее.

Обратите внимание, что поскольку Python — это язык с нулевым индексом, `a[2]` относится к третьему элементу, который является кортежом `(100,200,300)`. `a[2][1]` относится ко второму элементу в этом кортеже, который равен `200` .

Временная сложность кортежей

Временную сложность различных функций кортежей можно резюмировать следующим образом (с использованием обозначения Big O):

Дополнение по времени

сложности функции O(1)

Обратите внимание, что Append — это функция, которая добавляет элемент ближе к концу уже существующего кортежа. Его сложность равна O(1). Словари и множестваВ этом разделе мы обсудим множества и словари, которые используются для хранения данных, не имеющих явного или неявного порядка. И словарь, и наборы довольно похожи. Разница в том, что словарь имеет ключ и пару значений. Набор можно представить как набор уникальных ключей. Давайте рассмотрим их по одному

Словари

Хранение данных как пар ключ-значение особенно важно в распределённых алгоритмах. В Python коллекция этих пар ключ-значение хранится в виде структуры данных, называемой **словарем**. Для создания словаря следует выбрать ключ как атрибут, который лучше всего подходит для идентификации данных в процессе обработки данных. Ограничение значения ключей заключается в том, что они должны быть хешируемыми типами. Хешируемые — это тип объектов, на которых мы можем запускать хеш-функцию, генерируя хеш-код, который никогда не меняется в течение своей жизни. Это гарантирует, что ключи уникальны, а поиск ключа быстрый. Числовые типы и плоские неизменяемые типы str — все хэшируемые и являются хорошим выбором для ключей словаря. Значение может быть элементом любого типа, например, числом или строкой. Python также всегда использует сложные типы данных, такие как списки, в качестве значений. Вложенные словари можно создать, используя словарь в качестве типа данных значения. Чтобы создать простой словарь, который присваивает цвета различным переменным, пары ключ-значение должны быть включены в { }. Например, следующий код создаёт простой словарь, состоящий из трёх пар ключ-значение:

```
>>> bin_colors ={
      "manual_color": "Жёлтый",
      "approved_color": "Зелёный",
"refused_color": "Красный"
    }
>>> print(bin_colors)
```

```
{'manual_color': 'Жёлтый', 'approved_color': 'Зелёный', 'refused_color':
'Красный'}
```
Три пары ключ-значение, созданные предыдущим кодом, также показаны на следующем скриншоте:

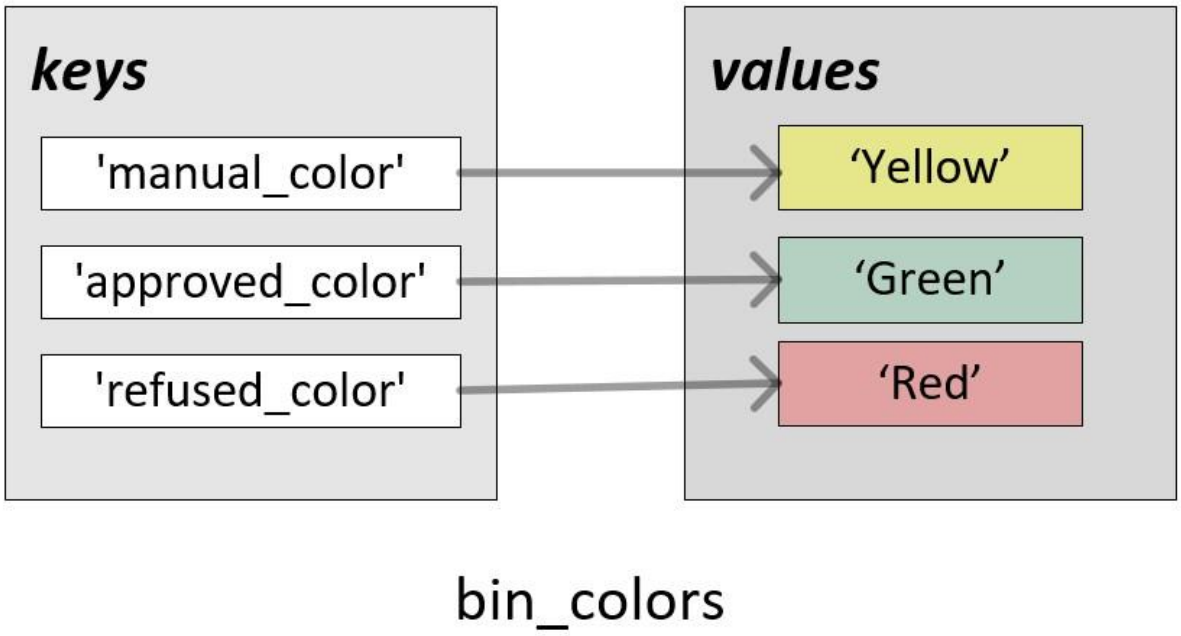


Рисунок 2.2: Пары ключ-значение в простом словаре

Теперь посмотрим, как получить и обновить значение, связанное с ключом: Чтобы получить значение, связанное с ключом, можно использовать либо `функцию get` , либо ключ в качестве индекса:

```
>>> bin_colors.get('approved_color')
«Зелёный»
>>> bin_colors['approved_color']
«Зелёный»
Чтобы обновить значение, связанное с ключом, используйте следующий код:
>>> bin_colors['approved_color']="Фиолетовый"
>>> print(bin_colors)
{'manual_color': 'Жёлтый', 'approved_color': 'Фиолетовый', 'refused_color':
'Красный'}
```

Обратите внимание, что предыдущий код показывает, как можно обновить значение, связанное с определённым ключем, в словаре.

Множества

Тесно связанным со словарём является множество, которое определяется как неупорядоченный набор различных элементов, которые могут быть разных типов.

Один из способов определения множества — заключить значения в { }. Например, посмотрите на следующий блок кода:

```
>>> зелёный = {'трава', 'листья'}
>>> принт (зелёный)
{'трава', 'листья'}
```

Определяющей характеристикой множества является то, что оно хранит только отдельное значение каждого элемента. Если мы попытаемся добавить ещё один избыточный элемент, он проигнорирует это, как показано ниже:

```
>>> зелёный = {'трава', 'листья','листья'}
>>> принт (зелёный)
{'трава', 'листья'}
```

Чтобы продемонстрировать, какие операции можно выполнять над множествами, давайте определим два множества: Множество с названием жёлтый, в котором есть желтые элементы. Другое множество с красным, где есть красные элементы. Обратите внимание, что некоторые общие черты между этими двумя множествами. Оба множества и их взаимосвязь можно представить с помощью следующей диаграммы Венна:

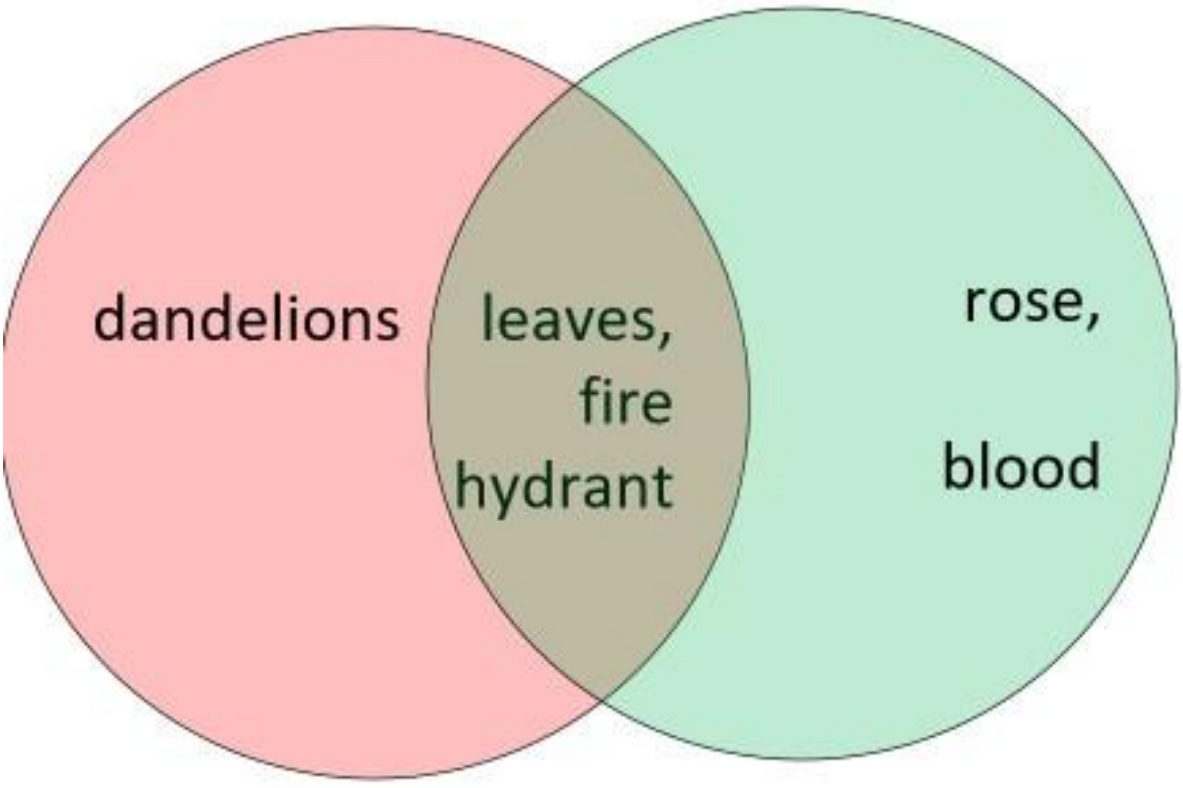


Рисунок 2.3: Диаграмма Венна, показывающая, как элементы хранятся в множествах

Если мы хотим реализовать эти два набора на Python, код будет выглядеть так:

```
>>> жёлтый = {'одуванчики', 'пожарный гидрант', 'листья'}
>>> красный = {'пожарный гидрант', 'кровь', 'роза', 'листья'}
```

Теперь рассмотрим следующий код, который демонстрирует операции множества с использованием Python:

```
>>> жёлтый|красный
{'одуванчики', 'пожарный гидрант', 'кровь', 'роза', 'листья'}
>>> жёлто-красный
{'пожарный гидрант','листья'}
```

Как показано в предыдущем фрагменте кода, множества в Python могут иметь операции, такие как объединения и пересечения. Как известно, операция объединения объединяет все элементы обоих множеств, а операция пересечения даст множество общих элементов между этими двумя множествами. Обратите внимание: `жёлтый|красный` используется для получения объединения двух предыдущих определённых множеств. `Жёлто-красный` используется для определения пересечения между жёлтым и красным.

Анализ временной сложности для словаря и множеств

Ниже приведён анализ временной сложности для множеств:

Множества	Сложность
Добавить элемент	O(1)
Получить элемент	O(1)
Принято	O(n)

Важно, что при анализе сложности словаря или наборов важно отметить, что они используют хеш-таблицы для достижения производительности O(1) как при сложении элементов, так и при поиске. Таким образом, время, необходимое для получения или установки значения ключа, полностью не зависит от размера словаря или множеств. Это связано с тем, что в словаре хеш-функция отображает ключи в целое число, которое используется для хранения пар ключ-значение. Библиотека pandas умело использует хеш-функцию, чтобы превратить произвольный ключ в индекс последовательности. Это означает, что время, необходимое для получения пары ключ-значение, в словарь размером три совпадает с временем, необходимым для получения пары ключ-значение в словарь размером один миллион. В

множествах хешируются и хранятся только ключи, что приводит к той же временной сложности.

Когда пользоваться словарем, а когда — набором?

Предположим, что мы ищем структуру данных для нашего телефонного справочника. Мы хотим хранить номер телефонов сотрудников компании. Для этого словарь — это правильная структура данных. Имя каждого сотрудника будет ключом, а значение — номером телефона

```
Телефонная книга = {
    «Икрема Хамза»: «555-555-5555»,
    «Джойс Достон»: «212-555-5555»,
}
```

Но если мы хотим хранить только уникальную ценность сотрудников, то это нужно делать с помощью наборов.

```
сотрудники = {
"Ikrema Hamza",
"Joyce Doston"
}
```

Использование рядов и кадров данныхОбработка данных — одна из ключевых задач, которые необходимо реализовать при реализации большинства алгоритмов. В Python обработка данных обычно осуществляется с использованием различных функций и структур данных библиотеки pandas. В этом разделе мы рассмотрим следующие две важные структуры данных библиотеки pandas, которые будут использоваться для реализации различных алгоритмов в этой книге.**Серия:** одномерный массив значенийDataFrame: двумерная структура данных, используемая для хранения табличных данныхДавайте сначала рассмотрим структуру данных рядов.

Серия

В библиотеке панд `Серия` — это одномерный массив значений для однородных данных. Можно представить Series как одну колонку в таблице. Можно представить, что ряды содержат разные значения определённой переменной. Ряд можно определить следующим образом:

```
[1]: import pandas as pd
     pd.Series(['red', 'green', 'blue', 'white', 'black'])
```

Обратите внимание, что в структурах данных на основе серий панды существует термин «ось», используемый для обозначения последовательности значений в определённом измерении. У ряда есть только «ось 0», потому что она имеет только одно измерение. Мы увидим, как эта концепция оси применяется к DataFrame в следующем разделе, который представляет собой двумерную структуру данных.

DataFrame

DataFrame построен на структуре данных Series и хранит двумерные табличные данные . Это одна из важнейших структур данных для алгоритмов, используемая для обработки традиционных структурированных данных. Рассмотрим следующую таблицу:

id name возраст

решение 1 Тарифы 32

Верно

Б Елена 23 Ложные

В Стивен 40 Тру

Теперь представим это с помощью `DataFrame` . Простой `DataFrame` можно создать, используя следующий код:

```
>>> импортировать панд как PD
>>> df = pd. DataFrame([
...              ['1', 'Fares', 32, True],
...              ['2', 'Elena', 23, ложь],
...              ['3', 'Стивен', 40, True]])
>>> df.columns = ['id', 'name', 'age', 'decision']
>>> df
  id name age решение 0 1
Тарифы 32 Верно
А  2 Елена 23 Ложь
Б  3 Стивен 40 Тру
```

Обратите внимание, что в предыдущем коде `df.column` — это список, указывающий имена столбцов. В DataFrame один столбец или строка называется **осью**.

DataFrame также используется в других популярных языках и фреймворках для реализации табличной структуры данных. Примеры — R и фреймворк Apache Spark.

Создание подмножества DataFrame

По сути, существует два основных способа создания подмножества DataFrame (например, название подмножества — `myDF`):

- Выбор столбцов
- Выбор строк

Давайте посмотрим их по одному.

Выбор колонок

В алгоритмах машинного обучения выбор правильного набора признаков является важной задачей. Из всех признаков, которые у нас могут быть, не все могут быть необходимы на определённом этапе алгоритма. В Python выбор признаков осуществляется с помощью выбора столбцов, что объясняется в этом разделе. Столбец можно найти по `названию` , как следующее:

```
>>>
df[['name','before']]
до 0 ярмарок 32
А   Елена 23
Б   Стивен 40
```

Положение столбца в DataFrame детерминировано. Столбец можно получить по её позиции следующим образом:

```
>>> df.iloc[:,3]
  Верно
А Ложь
Б Верно
```

Обратите внимание, что в этом коде мы получаем все строки DataFrame.

Выбор строк

Каждая строка в DataFrame соответствует точке данных в нашем проблемном пространстве. Нам нужно выполнить выбор строки, если мы хотим создать подмножество элементов данных, которые находятся в нашем проблемном пространстве. Это подмножество можно создать, используя один из двух следующих методов:

- Указывая их положение
- Задавая фильтр

Подмножество строк можно получить по его позиции следующим образом:

```
>>> c
  ID Name Age Decision
А 2 Елена 23 Ложь
```

```
Б 3 Стивен 40 Тру
```

Обратите внимание, что предыдущий код вернёт вторую и третью строки плюс все столбцы. Он использует метод iloc, который позволяет получать доступ к элементам по их числовому индексу. Чтобы создать подмножество с указанием фильтра, необходимо использовать один или несколько столбцов для определения критерия выбора. Например, подмножество элементов данных может быть выбрано следующим методом:

```
>>> df[df.возраст>30] Я бы
назвал возраст решение 0 1
Тарифы 32 Истинно
2 3 Стивен 40 Тру
>>>
df[(df.age<35)&(df.decision==True)] id
name age decision 0 1 Тарифы 32 True
```

Обратите внимание, что этот код создаёт подмножество строк, удовлетворяющих условию, установленному в фильтре.

Матрицы

Матрица — это двумерная структура данных с фиксированным количеством столбцов и строк. Каждый элемент матрицы можно называть своим столбцем и строкой. В Python матрицу можно создать с помощью `массива numpy` , как показано в следующем коде:

```
>>> 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))
<класс 'numpy.ndarray'>
```

Обратите внимание, что предыдущий код создаст матрицу с тремя строками и тремя столбцами.

Матричные операции

Существует множество операций для обработки матричными данными. Например, попробуем транспонировать предыдущую матрицу. Мы используем `функцию transpose()`, которая преобразует столбцы в строки, а строки — в столбцы:

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

Обратите внимание, что матричные операции часто используются в мультимедийной обработке данных. Теперь, когда мы узнали о структурах данных в Python, перейдём к абстрактным типам данных в следующем разделе.

Изучение абстрактных типов данных

Абстрактные типы данных (**ADT**) — это высокоуровневые абстракции, поведение которых определяется набором переменных и набором связанных операций. ADT определяют руководство по реализации «чего» следует ожидать, но дают программисту свободу в том, как именно это будет реализовано. Примеры — векторы, очереди и стеки. Это означает, что два разных программиста могут использовать два разных подхода для реализации стеков, подобных ADT. Скрывая детали на уровне реализации и предоставляя пользователю универсальную, независимую от реализации структуру данных, использование ADT создаёт алгоритмы, которые приводят к более простому и чистому коду. ADT могут быть реализованы на любом языке программирования, таком как C++, Java и Scala. В этом разделе мы реализуем ADT с использованием Python. Начнём с векторов.

Вектор

Вектор — это одномерная структура для хранения данных. Они являются одной из самых популярных структур данных в Python. Существует два способа создания векторов в Python: Использование списка Python: Самый простой способ создать вектор — использовать список Python, следующим образом:

```
>>> myVector = [22,33,44,55]
>>> print (myVector)
[22 33 44 55]
>>> print(type(myVector))
<Список классов>
```

Обратите внимание, что этот код создаст список из четырёх элементов. Использование `массива numpy` : Ещё один популярный способ создания вектора — использование массив NumPy, следующим образом:

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

Обратите внимание, что мы создали `myVector` с помощью `np.array` в этом коде.

В Python мы можем представлять целые числа с помощью подчёркивания для разделения частей. Это делает их более читаемыми и менее подверженными

ошибкам. Это особенно полезно при работе с большими числами. Таким образом, один миллиард можно представить как a=1

Стеки

Стек — это линейная структура данных для хранения одномерного списка. Он может хранить товары как в **формате Last-In, First-Out** (**LIFO),** так и **First In, Last-Out** (**FILO**). Определяющей особенностью стека является способ добавления и удаления элементов. Новый элемент добавляется на одном конце, и элемент удаляется только с этого конца. Ниже приведены операции, связанные со стеками:

- **isEmpty**: Возвращает истинное, если стек пуст
- **push**: Добавляет новый **pop** элемента: Возвращает
- добавленный последним элементом и удаляет его

Рисунок 2.4 показывает, как можно использовать операции push и pop для добавления и удаления данных из стека:

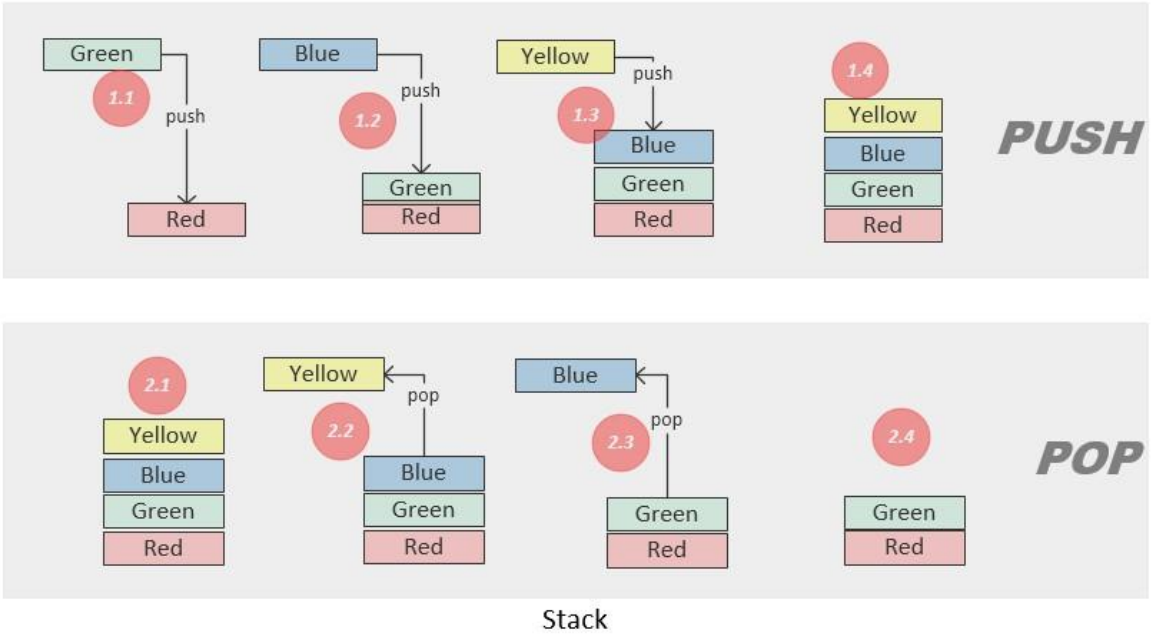


Рисунок 2.4: Операции push и pop

Верхняя часть *рисунка 2.4* показывает использование операций push для добавления элементов в стек. В шагах **1.1**, **1.2** и **1.3** операции push используются три раза для добавления трёх элементов в стек. Нижняя часть предыдущей диаграммы используется для извлечения сохранённых значений из стека. В шагах **2.2** и **2.3** операции pop используются для извлечения двух элементов из стека в формате LIFO. Давайте создадим класс под названием Stack на Python, где мы определим все операции, связанные с классом стека. Код этого класса будет следующим:

```
Стек классов:
     def __init__(self):
self.items = [] def
isEmpty(self): return
self.items == [] def
push(self, entem):
         self.items.append(item)
def pop(self):
         Вернуть self.items.pop()
Def Peek(self):
         Return self.items[len(self.items)-
1] def size(self): return len(self.items)
```

Для передачи четырёх элементов в стек можно использовать следующий код:

Populate the stack

```
In [2]: stack=Stack()
        stack.push('Red')
        stack.push('Green')
        stack.push("Blue")
        stack.push("Yellow")
```

Pop

```
In [3]: stack.pop()
Out[3]: 'Yellow'

In [7]: stack.isEmpty()
Out[7]: False
```

Обратите внимание, что предыдущий код создаёт стек из четырёх элементов данных.

Временная сложность стеков

Давайте рассмотрим временную сложность стеков (с использованием нотации Big O):

Операционная сложность

толкни	O(1)
поп	O(1)
размер	O(1)
пик	O(1)

Важно отметить, что производительность ни одной из четырёх операций, упомянутых в предыдущей таблице, не зависит от размера стека.

Практический пример

Стек используется в качестве структуры данных во многих случаях использования. Например, когда пользователь хочет просмотреть историю в веб-браузере, это шаблон доступа к данным LIFO, и для хранения истории можно использовать стек. Другой пример — когда пользователь хочет выполнить `операцию «Отменить` » в текстовом редакторе.

Очереди

Как и стеки, очередь хранит *n* элементов в одномерной структуре. Элементы добавляются и удаляются в **формате FIFO**. Один конец очереди называется **тылом**, а другой — **передним**. Когда элементы удаляются с фронта, операция называется **выходом из очереди**. Когда элементы добавляются сзади, операция называется **enqueue**. На следующей схеме верхняя часть показывает операцию enqueue. Шаги **1.1**, **1.2** и **1.3** добавляют три элемента в очередь, и полученная очередь отображается в **версии 1.4**. Обратите внимание, что **жёлтый** — **это задняя** часть, **а красный** — **передний**. Нижняя часть следующей схемы показывает операцию выхода из очереди. Шаги **2.2**, **2.3** и **2.4** удаляют элементы из очереди один за другим с первых рядов:

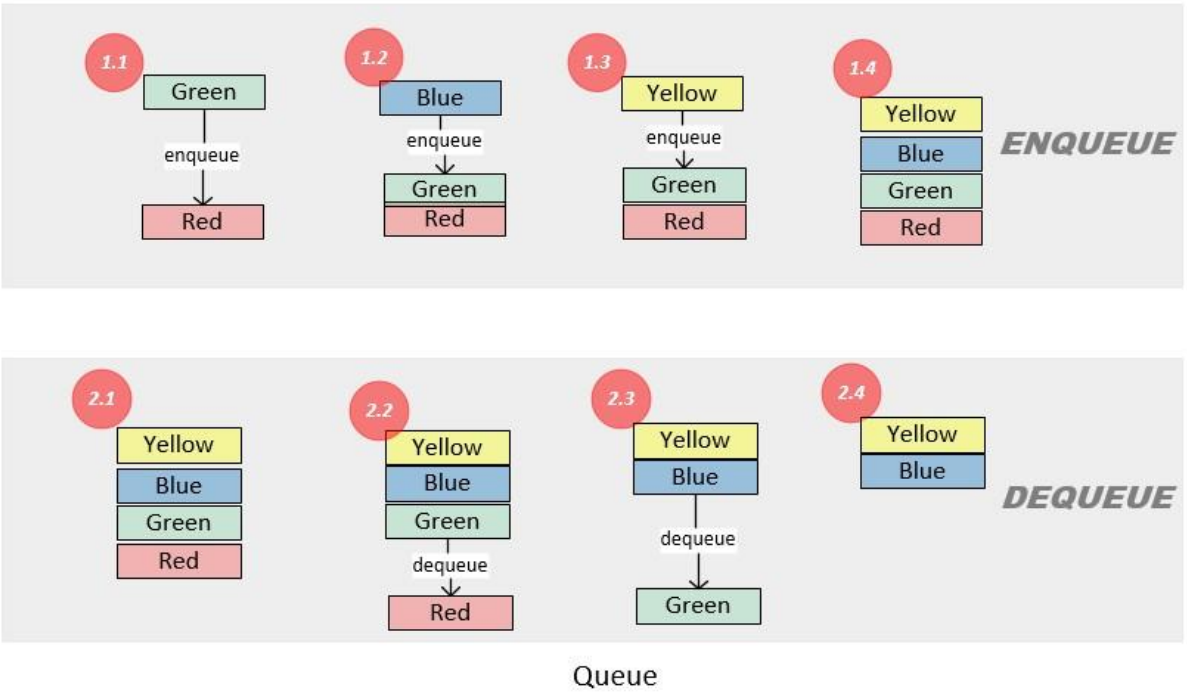


Рисунок 2.5: Операции вычисления в очередь и выхода из очереди

Очередь, показанная на предыдущей диаграмме, 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)
```

Давайте поставим в очередь и выведем из очереди элементы, как показано на предыдущей диаграмме с помощью следующего скриншота:

Using Queue Class

```
In [2]: queue = Queue()

In [3]: queue.enqueue('Red')

In [4]: queue.enqueue('Green')

In [5]: queue.enqueue('Blue')

In [6]: queue.enqueue('Yellow')

In [7]: print(queue.size())
        4

In [8]: print(queue.dequeue())
        Red

In [9]: print(queue.dequeue())
        Green
```

Обратите внимание, что предыдущий код сначала создаёт очередь, а затем помещает в неё четыре элемента.

Основная идея использования стеков и очередей

Давайте рассмотрим основную идею использования стеков и очередей с помощью аналогии. Предположим, что у нас есть таблица, куда мы кладём входящие письма от нашей почтовой службы, например, Canada Mail. Мы складываем их в стопку, пока не успеем открыть и посмотреть почту по одному. Есть два возможных способа сделать это: мы кладём письмо в стопку, и когда получаем новое письмо, кладём его сверху стопки. Когда мы хотим прочитать письмо, мы начинаем с того, что сверху. Это то, что мы называем стеком. Обратите внимание, что последнее письмо будет указано сверху и будет обработано первым. Поднятие буквы с верхней части списка называется *операцией pop*. Когда приходит новая буква, размещение её сверху называется *операцией push*. Если у нас получится большая стопка и постоянно приходит много писем, есть шанс, что мы никогда не успеваем добраться до очень важного письма, которое ждёт нас в нижней части стопки. Мы складываем письмо в стопку, но хотим сначала взять самое старое письмо: каждый раз, когда мы хотим посмотреть одно или несколько букв, мы стараемся сначала взять самое старое. Это то, что мы называем очередью. Добавление буквы в стопку называется *операцией enqueue*. Удаление буквы из стопки называется *операцией выхода из очереди*.

Дерево

В контексте алгоритмов дерево является одной из самых полезных структур данных благодаря своим иерархическим возможностям хранения данных. При проектировании алгоритмов мы используем деревья там, где нужно представить иерархические отношения между элементами данных, которые нужно хранить или обрабатывать. Давайте подробнее рассмотрим эту интересную и довольно важную структуру данных. Каждое дерево имеет конечный набор узлов, так что в нем есть начальный элемент данных, называемый корнем , и множество узлов, соединённых связями, называемыми **ветвями**.

Терминология

Давайте рассмотрим некоторые термины, связанные со структурой данных дерева:

Корень Узел без родителя называется корневым узлом. Например, в следующем узле Диаграмма, корневой узел — A. В алгоритмах, как правило, корневой узел содержит самое важное значение в структуре дерева.

Уровень Расстояние от корневого узла — это уровень узла. Например, в . следуя диаграмме, уровень узлов D, E и F равен двум.
узел

Братья и сестры Два узла в дереве называются братьями и сестрами, если они находятся на одном уровне. Например, если узлы Мы смотрим следующую диаграмму: узлы B и C — братья.

Ребёнок уровень и родительский узел
Узел F, является потомком узла C, если оба напрямую связаны и узел C меньше узла F. Наоборот, узел C является родителем узла F. Узлы C и F на следующей диаграмме показывают это отношение родитель-ребёнок.

Степень узла
Степень узла — это количество потомков, которые у него есть. Например, в следуя диаграмме, узел B имеет степень две.

Степень дерева
Степень дерева равна максимальной степени, которую можно найти среди составляющих узлов дерева. Например, дерево, представленное на следующей диаграмме, имеет степень два.

Поддерево
Поддерево дерева — это часть дерева, где выбранный узел является корневым узлом поддерева, а все дети — узлами дерева. Например, поддерево в узле E дерева, представленного на следующей диаграмме, состоит из узла E как корневого узла и узла G и H как двух потомков.

Лист узле
Узел в дереве без потомков называется листовым узлом. Например, в ниже на рисунке D, G, H и F — четыре листовых узла.

Внутренний
Любой узел, который не является ни корнем, ни листовым узлом, является внутренним узлом. Внутренний узел будет как минимум один родитель и как минимум один дочерний узел.

> Обратите внимание, что деревья — это своего рода сеть или граф, которые мы будем изучать в *главе 6*, *«Алгоритмы неконтролируемого машинного обучения»*. Для графов и сетевого анализа мы используем термины «ссылка» или «ребра» вместо ответвлений. Большинство остальных терминов осталось без изменений.

Виды деревьев

Существуют различные типы деревьев, которые объясняются следующим образом:
Бинарное дерево: если степень дерева равна две, это дерево называется *бинарным деревом*. Например, дерево, показанное на следующей диаграмме, является бинарным деревом, так как его степень равна две:

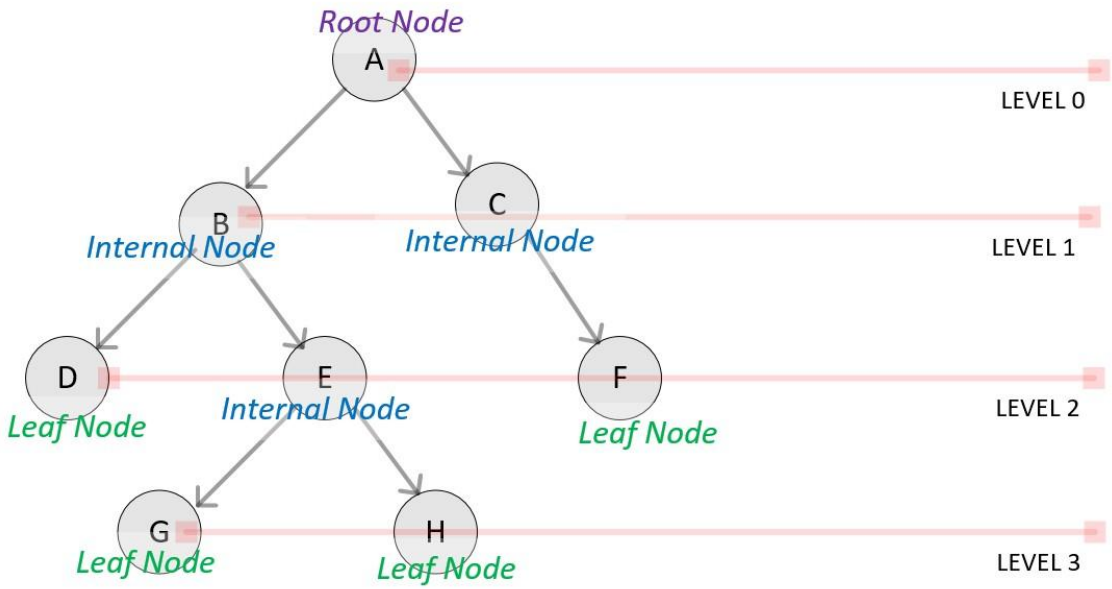


Рисунок 2.6: Двоичное дерево

Обратите внимание, что предыдущая диаграмма показывает дерево с четырьмя уровнями и восемью узлами. Полное дерево: полное дерево — это то, в котором все узлы одинаковой степени, которая будет равна степени дерева. Следующая диаграмма показывает виды деревьев, обсуждавшихся ранее:

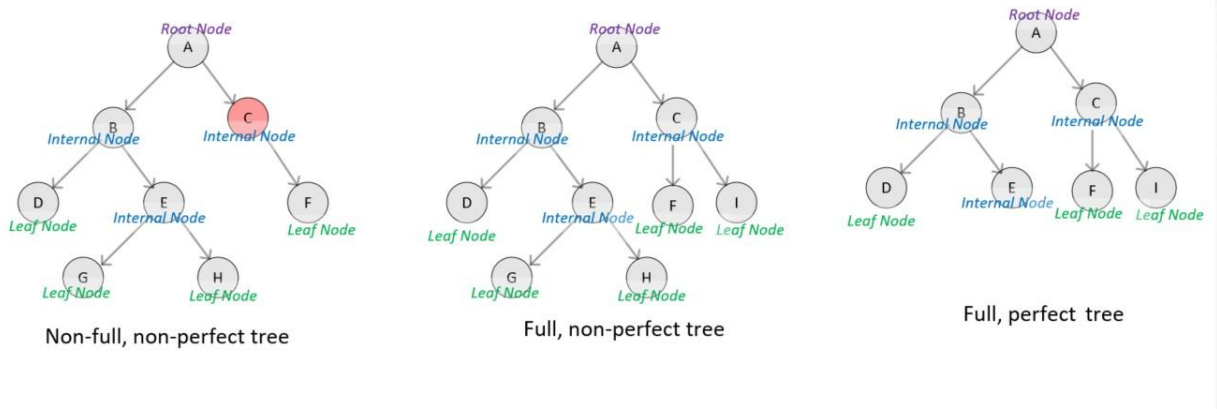


Рисунок 2.7: Полное дерево

Обратите внимание, что бинарное дерево слева не является полноценным деревом, так *как узел C* имеет степень один, а все остальные узлы — степень две. Дерево посередине и дерево слева — оба полные деревья. Идеальное дерево: Идеальное дерево — это особый тип полноценного дерева, у которого все листовые узлы находятся в

На том же уровне. Например, двоичное дерево справа, как показано на предыдущей диаграмме, является совершенным, полноценным деревом, так как все листовые узлы находятся на одном уровне, то есть на уровне 2. Упорядочённое дерево: Если дети узла организованы в определённом порядке по определённым критериям, дерево называется *упорядочённым деревом*. Например, дерево можно упорядочить слева направо в восходящем порядке, при этом узлы на одном уровне будут увеличиваться в стоимости при движении слева направо.

Практические примеры

Абстрактное дерево типов данных — одна из основных структур данных, используемых при разработке деревьев решений, как будет рассмотрено в *главе 7*, *«Традиционные алгоритмы контролируемого обучения*». Благодаря своей иерархической структуре он также популярен в алгоритмах, связанных с сетевым анализом, что будет подробно рассмотрено в *главе 6*, « *Алгоритмы неконтролируемого машинного обучения*». Деревья также используются в различных алгоритмах поиска и сортировки, где необходимо реализовать стратегии разделяй и властвуй.

Краткое содержание

В этой главе мы обсудили структуры данных, которые можно использовать для реализации различных типов алгоритмов. После изучения этой главы вы сможете выбрать правильную структуру данных для хранения и обработки данных алгоритмом. Вы также должны понимать последствия нашего выбора для производительности алгоритма. Следующая глава посвящена сортировке и поиску алгоритмов, где мы будем использовать некоторые структуры данных, представленные в этой главе, при реализации алгоритмов.

OceanfPDF.com

3 Алгоритмы сортировки и поиска

Присоединяйтесь к Discord-пространству нашей книги

https:// packt.link/40Algos



В этой главе мы рассмотрим алгоритмы, используемые для сортировки и поиска. Это важный класс алгоритмов, которые могут использоваться самостоятельно или служить основой для более сложных алгоритмов. К ним относятся обработка естественного языка (NLP) и алгоритмы извлечения шаблонов. Эта глава начинается с представления различных типов алгоритмов сортировки. В ней сравниваются результаты различных подходов к проектированию алгоритма сортировки. Затем подробно представлены некоторые алгоритмы поиска. Наконец, изучается практический пример алгоритмов сортировки и поиска, представленных в этой главе. К концу этой главы мы сможем понять различные алгоритмы, используемые для сортировки и поиска, и определить их сильные и слабые стороны. Поскольку алгоритмы поиска и сортировки являются строительными блоками для многих сложных алгоритмов, их подробное понимание поможет нам лучше понять современные сложные алгоритмы, как это также представлено в последующих главах. Основные концепции, обсуждаемые в этой главе:

- Введение алгоритмов сортировки
- Введение алгоритмов поиска
- Анализ производительности алгоритмов сортировки и поиска Практическое применение сортировки и поиска

Давайте сначала рассмотрим некоторые алгоритмы сортировки.

Введение алгоритмов сортировки

Возможность эффективно сортировать и искать элементы в сложной структуре данных важна, так как она необходима многим современным алгоритмам. Правильная стратегия сортировки и поиска данных будет зависеть от их размера и типа, как обсуждается в этой главе. Хотя конечный результат абсолютно одинаков, для эффективного решения реальной задачи потребуется правильный алгоритм сортировки и поиска. Поэтому важно тщательно анализировать работу этих алгоритмов. Алгоритмы сортировки широко используются в распределённых системах хранения данных, таких как современные NoSQL базы данных, которые позволяют создавать кластерные и облачные вычислительные архитектуры. В таких системах хранения данных элементы данных необходимо регулярно сортировать и хранить для эффективного извлечения. В этой главе представлены следующие алгоритмы сортировки:

- Сортировка пузырьков
- Слияние, сортировка
- Сортировка вставки
- Сортировка по оболочкам

Сортировка выбора

Но прежде чем представить эти алгоритмы, давайте сначала обсудим технику замены переменных в Python, которую мы будем использовать в коде, представленном в этой главе.

Перестановка переменных в Python

При реализации алгоритмов сортировки и поиска нужно поменять значения двух переменных. В Python существует стандартный способ поменять местами две переменные следующим образом:

```
var_1 = 1 var_2 = 2
var_1,var_2 = var_2,var_1
```

Этот простой способ обмена значениями используется во всех алгоритмах сортировки и поиска в этой главе. Начнём с алгоритма сортировки пузырьков в следующем разделе.

Сортировка пузырьков

Пузырчатая сортировка — один из самых простых и медленных алгоритмов, используемых для сортировки. Он устроен так, что наибольшее значение в списке данных *поднимается* наверх по мере прохождения итераций алгоритма. Пузырчатая сортировка требует мало памяти во время выполнения, так как весь упорядоченный процесс происходит внутри исходной структуры данных. Новые структуры данных не требуются в качестве временных буферов. Но в худшем случае производительность — O(N2), что является квадратичной временной сложностью (где N — количество сортируемых элементов). Как обсуждалось в следующем разделе, рекомендуется использовать только для небольших наборов данных. Фактические рекомендуемые пределы размера данных для использования сортировки с пузырьками зависят от памяти и доступных ресурсов обработки, но общее предложение считается сохранением количества элементов (N) ниже 1000.

Понимание логики сортировки пузырьков

Пузырчатая сортировка основана на различных итерациях, называемых **проходами**. Для списка *размера N* bubble sort будет иметь *проходы N-1*. Чтобы понять, что это работает, давайте сосредоточимся на первой итерации: первой. Цель первого прохода — поднять наивысшее значение в самый высокий индекс (в верхней части списка). Другими словами, мы увидим, как наивысшее значение списка *поднимается* наверх по мере продвижения первого прохода. Логика сортировки пузырьков основана на сравнении значений соседних соседей. Если значение при более высоком индексе выше, чем значение при более низком индексе, мы обмениваемся значениями. Эта итерация продолжается до конца списка. Это показано на *рисунке 3.1*:

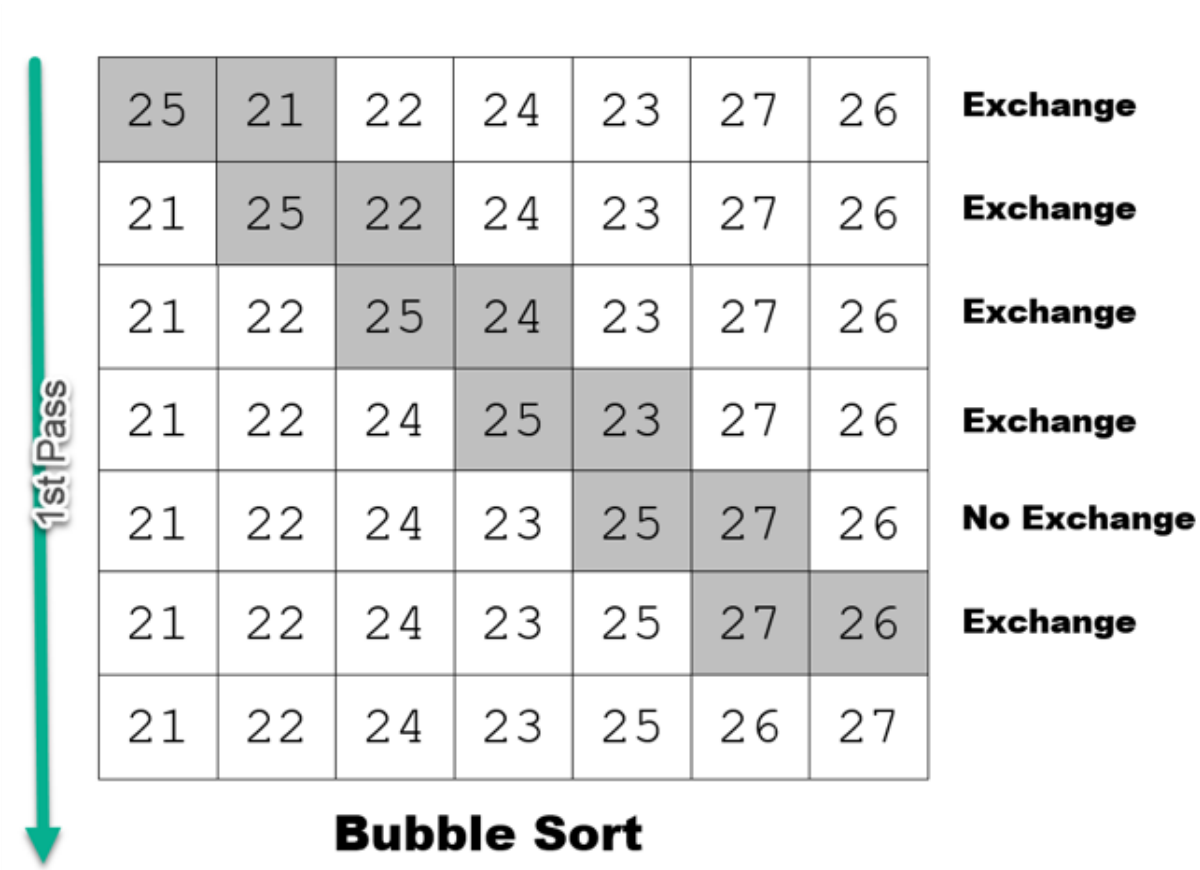


Рисунок 3.1: Алгоритм сортировки пузырьков

Давайте теперь посмотрим, как пузырчатая сортировка может быть реализована с помощью Python. Если мы реализуем передачу пузырьковой сортировки в Python, это будет выглядеть следующим образом:

```
last_element_index = len(list)-1
print(0,list) для idx в
диапазоне(last_element_index): if
list[idx]>list[idx+1]:
                    list[idx],list[idx+1]=list[idx+1],list[idx] print(idx+1,list)
```

```
  [25, 21, 22, 24, 23, 27, 26]
A [21, 25, 22, 24, 23, 27, 26]
Б [21, 22, 25, 24, 23, 27, 26]
В [21, 22, 24, 25, 23, 27, 26]
Г [21, 22, 24, 23, 25, 27, 26]
Д [21, 22, 24, 23, 25, 27, 26]
Е [21, 22, 24, 23, 25, 26, 27]
```

Обратите внимание, что после *первого прохода*:

- Наибольшее значение находится в верхней части списка, хранящегося в idx `+1` .
- При выполнении *первого прохода* алгоритм должен сравнить каждый элемент списка по отдельности, чтобы «пузырить» максимальное значение в этот верх.

После *первого прохода* алгоритм переходит ко *второму*. Цель *второго прохода* — переместить второе по величине значение на второй по величине индекс списка. Для этого алгоритм снова

сравнивает значения соседей, обмениваясь ими, если они не совпадают. Второй *проход* исключает значение в верхнем индексе, которое было поставлено в нужное место при *первом проходе.* Таким образом, у него будет меньше одного элемента данных. После завершения *второго прохода* алгоритм продолжает выполнять третий проход и последующие, пока все данные списка не будут в порядке роста. Алгоритму потребуется N-1 проходы для списка размера *N*, чтобы полностью отсортировать его. Реализация пузырьковой сортировки в Python выглядит следующим образом: Мы упомянули, что производительность — одно из ограничений `алгоритма сортировки пузырьков`. Давайте количественно оценим производительность `сортировки` пузырьков с помощью анализа эффективности алгоритма `сортировки` пузырьков:

```
def bubble_sort(список):
# Расставьте элементы в порядке last_element_index
= len(list)-1 для pass_no в
диапазоне(last_element_index,0,-1):
        для idx в диапазоне(pass_no):
если list[idx]>list[idx+1]:
                list[idx],list[idx+1]=list[idx+1],list[idx] return
list
```

Оптимизация сортировки пузырьков

Приведённая выше реализация пузырьковой сортировки не может использовать преимущества полностью отсортированного или частично отсортированного списка. Для сортировки данного списка нужно будет пройти через полные внутренние и внешние циклы, и это будет иметь временную сложность O(N2), даже если вход приведён полностью отсортированный список. Для оптимизации алгоритма сортировки пузырьков можно проверить, есть ли какая-либо операция перестановки при выполнении определённого прохода (внутренний цикл). Если перестановка не происходит ни в одном проходе, это означает, что данный список полностью отсортирован, и мы можем выйти из него. Так можно минимизировать количество проходов. Ниже приведена реализация оптимизированного алгоритма сортировки пузырьков:

Анализ производительности сортировки пузырьков

Легко заметить, что сортировка пузырьков включает два уровня циклов:

- Внешняя петля: это также называют проходами. Например, первый проход — это первая
- итерация внешней петли. Внутренний цикл: это когда оставшиеся неотсортированные элементы списка сортируются до тех пор, пока наибольшее значение не будет размещено вправо. В первом проходе будут *сравнения N-1*, во втором — *N-2*, а каждый следующий проход сокращает количество сравнений на одно.

Временная сложность алгоритма типа bubble выглядит следующим образом:

- **Лучший вариант**: если список уже отсортирован (или почти все элементы отсортированы), то сложность выполнения равна O(1)
- **Худший случай**: если отсортированы ни один или очень мало элементов, то наихудшая сложность во время выполнения — O(n2), так как алгоритму придётся полностью пройти через внутренние и внешние циклы.

Теперь давайте рассмотрим алгоритм сортировки по вставке.

Сортировка вставки

Основная идея сортировки через вставку заключается в том, что в каждой итерации мы удаляем точку данных из имеющейся структуры данных и затем вставляем её в нужное место. Вот почему мы называем это алгоритмом сортировки по вставке. В первой итерации мы выбираем две точки данных и сортируем их. Затем мы расширяем выбор, выбираем третью точку данных и находим её правильное положение по значению. Алгоритм движется до тех пор, пока все точки данных не будут перемещены на свои правильные позиции. Этот процесс показан на следующей диаграмме:

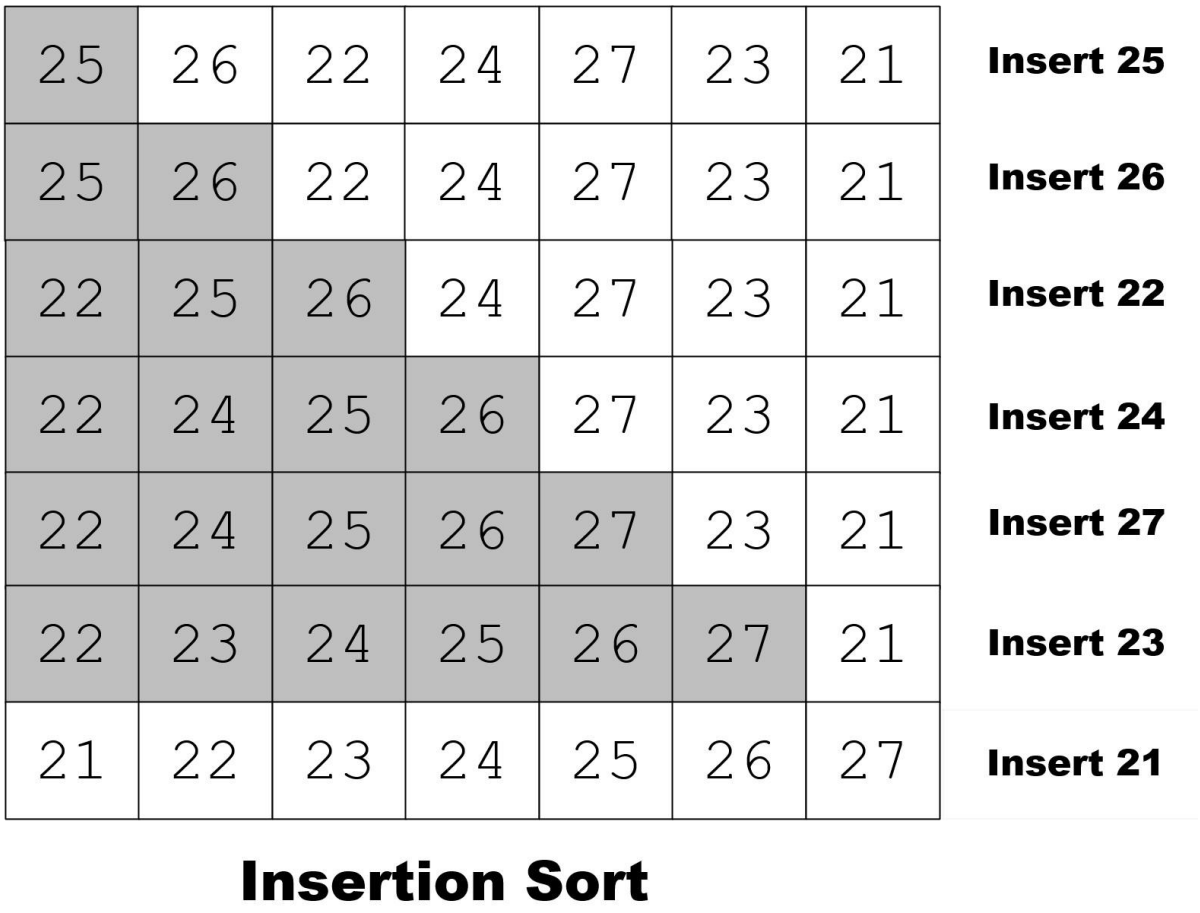


Рисунок 3.2 Алгоритм сортировки по вставке

Алгоритм сортировки по вставке может быть запрограммирован на Python следующим образом:

```
def insertion_sort (элементы):
    для i в диапазоне(1), len(элементы)):
        j = i-1
        Next = элементы[i]
# Сравните текущий элемент с следующим
            в то время как (элементы[j] >
следующим) и (j >= 0):
           элементы[j+1] = элементы[j]
j=j-1 элементы[j+1] = следующие
возвратные элементы
```

Обратите внимание, что в основном цикле мы повторяем весь список. В каждой итерации два соседних элемента — list `[j ]` (текущий элемент) и `list[i]` (следующий

стихия). В `list[j] > element_next` и `j >= 0` сравниваем текущий элемент со следующим элементом. Давайте рассмотрим производительность алгоритма сортировки вставки.

Анализ производительности сортировки вставки

Из описания алгоритма очевидно, что если структура данных уже отсортирована, вставка сортировки будет работать очень быстро. На самом деле, если структура данных отсортирована, то вставка сортировки будет иметь линейное время выполнения; то есть O(n). Худший случай — когда каждая из внутренних петель должна переместить все элементы списка. Если внутренний контур определен как *i*, то наихудшая производительность алгоритма сортировки вставки определяется следующим образом:

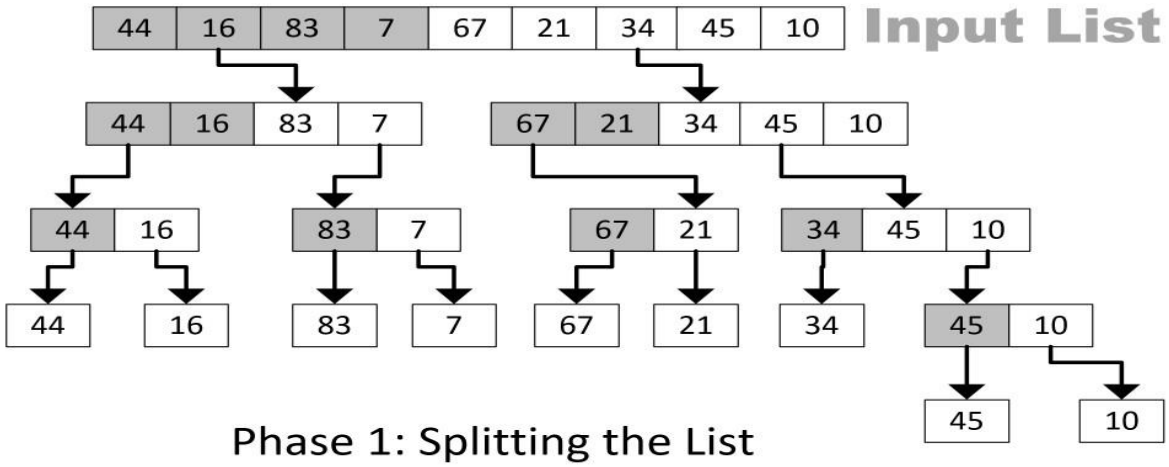
$$w(N) = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2} = \frac{N^2 - N}{2}$$

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

В целом, вставка может использоваться на небольших структурах данных. Для больших структур данных сортировка вставки не рекомендуется из-за квадратичной средней производительности.

Слияние, сортировка

На данный момент мы представили два алгоритма сортировки: сортировку пузырьков и сортировку вставки. Производительность обеих будет лучше, если данные частично отсортированы. Третий алгоритм, представленный в этой главе, — это алгоритм сортировки слияния, разработанный в 1940 году Джоном фон Нейманом. Определяющей особенностью этого алгоритма является то, что его производительность не зависит от того, отсортированы ли входные данные. Это хорошо работает для больших наборов данных. Как и другие алгоритмы больших данных, он основан на стратегии «разделяй и властвуй». В первом этапе, называемом расщеплением, алгоритм продолжает рекурсивно делить данные на две части, пока их размер не уменьшится за определённый порог. Во второй фазе, называемой слиянием, алгоритм продолжает слияние и обработку до получения конечного результата. Логика этого алгоритма объясняется на следующей диаграмме:



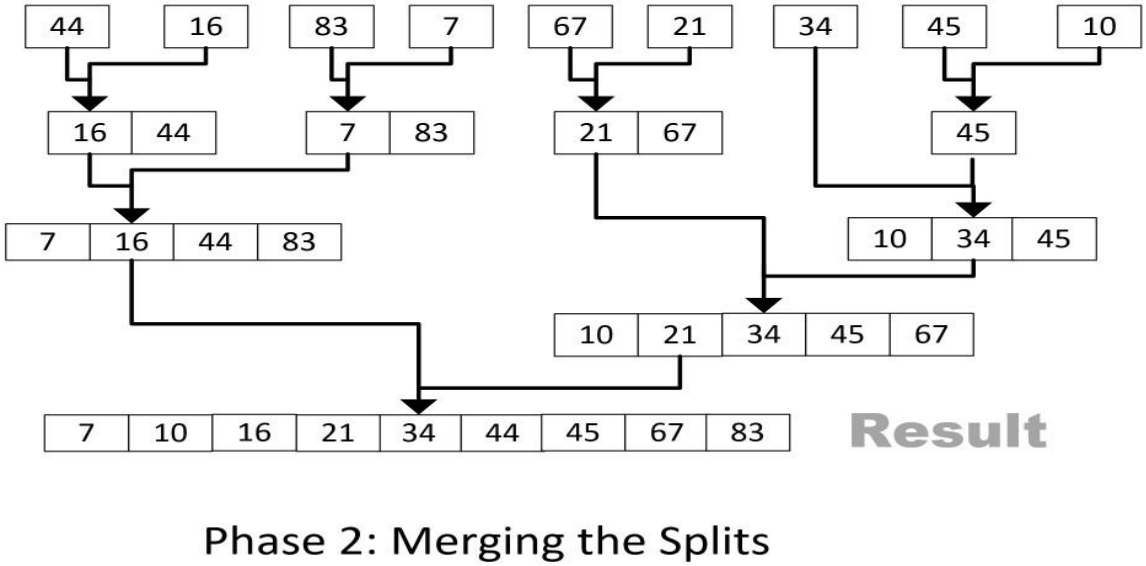


Рисунок 3.3 Алгоритм сортировки слияния

Давайте сначала рассмотрим псевдокод алгоритма сортировки слияния:

```
merge_sort (элементы, начало, конец)
if(начало < конец)
        midPoint = (конец - старт) / 2 + start merge_sort
(элементы, старт, серединаТочка) merge_sort (элементы,
серединаТочка + 1, старт) merge(элементы, старт, серединаТочка,
конец)
```

Как видно, алгоритм состоит из следующих трёх шагов:

А. Он делит список входных данных на две равные части, разделённые вокруг `середины точки`
Б. Используется рекурсия для разделения, пока длина каждого списка не станет 1
В. Затем он объединяет отсортированные части в отсортированный список и возвращает его

Код для реализации `merge_sort` приведён здесь:

```
 def merge_sort(элементы):
если len(элементы)>1:
        середина = len(элементы)//2 #splits список
пополам слева = элементы[:mid] право =
элементы[mid:]
        merge_sort(слева)#repeats пока длина каждого списка не
станет 1 merge_sort(справа) a = 0, b = 0 c = 0, a < len(слева) и
b < len(справа):
            если лево[a] <
правое[b]: элементы[c]=left[a] a =
a + 1, иначе:
                элементы[c]=правые[b]
b = b + 1 c = c + 1 в то время как a
< len(слева): элементы[c]=left[a] a
= a + 1 c = c + 1 в то время как b <
len(справа): элементы[c]=право[b] b
= b + 1 c = c + 1 возвратные
элементы
```

Сортировка по оболочкам

Алгоритм сортировки пузырьков сравнивает ближайших соседей и обменивается ими, если они не в порядке. С другой стороны, сортировка с вставками создаёт отсортированный список, перенося по одному элементу за раз. Если у нас частично отсортированный список, вставка сортировки должна обеспечивать разумную производительность. Но для совершенно неотсортированного списка *размером N* можно утверждать, что пузырчатая сортировка должна полностью пройти через *N-1* проходы, чтобы полностью отсортировать. Дональд Шелл предложил сортировку Shell (названную в его честь), которая ставит под сомнение важность выбора ближайших соседей для сравнения и обмена. Теперь давайте разберёмся с этой концепцией. В первом проходе вместо выбора непосредственных соседей мы используем элементы с фиксированным зазором, в итоге сортируем подсписок, состоящий из пары точек данных. Это показано на следующей диаграмме. Во втором проходе она сортирует подсписки, содержащие четыре точки данных (см. следующую диаграмму). В последующих проходах количество точек данных в каждом подсписке продолжает расти, а количество подсписков уменьшается, пока не достигается ситуации, когда остаётся только один подсписок, состоящий из всех точек данных. На этом этапе можно предположить, что список отсортирован:

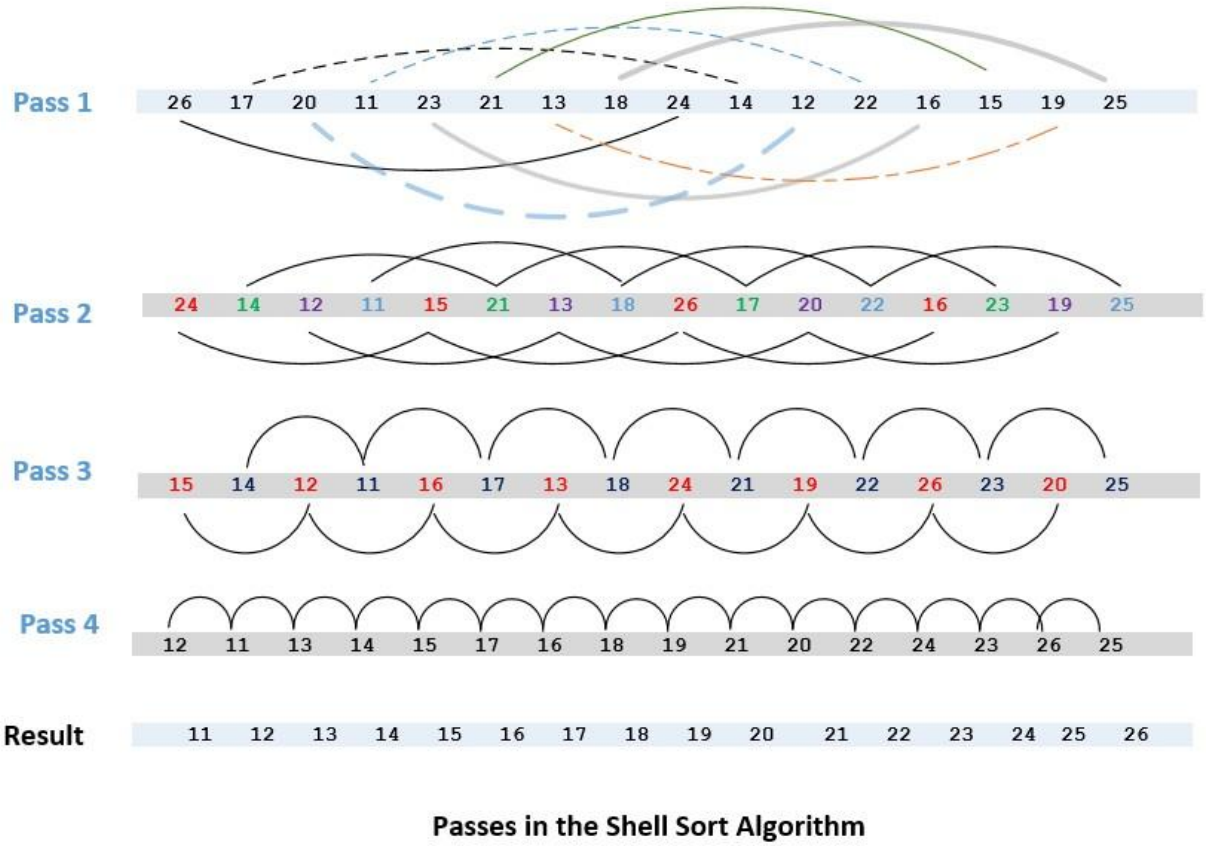


Рисунок 3.4 Проходы алгоритма сортировки оболочки

В Python код для реализации алгоритма сортировки Shell выглядит следующим образом:

```
def shell_sort(элементы):
    distance = len(элементы) // 2, при этом
расстояние > 0: для i в диапазоне(distance,
len(элементы)):
            temp = элементы[i] j
= i
# Отсортировать подсписок для этого расстояния, пока j > =
расстояние и элементы[j - расстояние] > температуре:
                list[j] = элементы[j -
расстояние] j = j-список расстояний[j] =
температура
# Уменьшить расстояние для следующего
элемента расстояние = расстояние//2
возвратных элемента
```

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

Анализ производительности сортировки Shell

Можно наблюдать, что в худшем случае алгоритм сортировки оболочки должен будет проходить через оба цикла, что придаёт ему сложность O(n2). Shell sort не предназначен для больших данных. Он используется для средних по размеру наборов данных. Грубо говоря, у него достаточно хорошая производительность в списке с до 6 000 элементов. Если данные частично расположены в

правильном порядке, производительность будет лучше. В лучшем случае, если список уже отсортирован, ему потребуется всего один проход через *N* элементов для проверки порядка, что даёт наилучшую производительность *O(N)*.

Сортировка выбора

Как мы видели ранее в этой главе, пузырчатая сортировка — один из самых простых алгоритмов сортировки. Сортировка выбора — это улучшение по сравнению с пузырьковой сортировкой, где мы стараемся минимизировать общее количество замен, требуемых алгоритмом. Он предназначен для одного обмена для каждого прохода, в отличие от *N-1* проходов с алгоритмом сортировки пузырьков. Вместо того чтобы набирать наибольшее значение вверх маленькими шагами (как при сортировке пузырьков, что приводит к *замене N-1*), мы ищем наибольшее значение в каждом проходе и перемещаем его наверх. Таким образом, после первого прохода наибольшее значение будет сверху. После второго прохода второе по величине значение будет рядом с верхним. По мере продвижения алгоритма последующие значения перемещаются на правильное место в зависимости от их значений. Последнее значение будет перемещено после *(N-1)*- го прохода. Таким образом, выборная сортировка выполняет *N-1* проходы для сортировки *N* элементов:

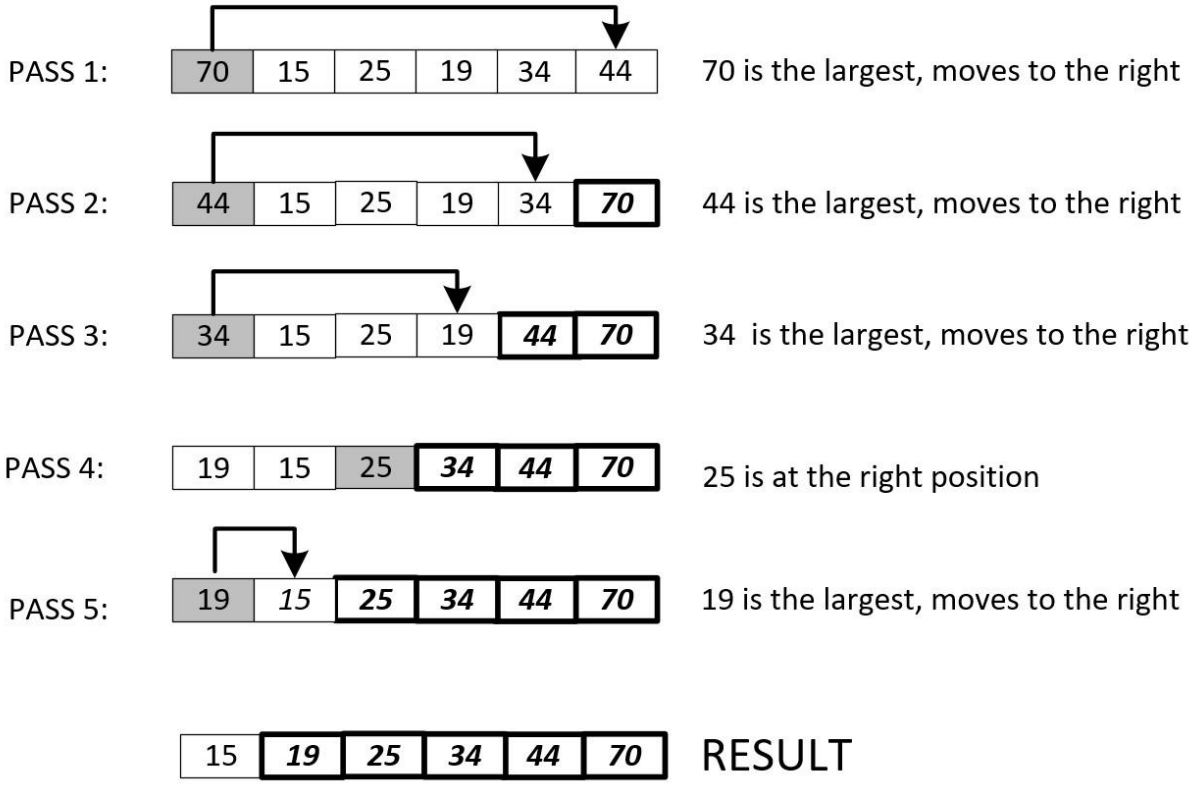


Рисунок 3.5: Алгоритм сортировки выбора

Реализация сортировки выбора в Python показана здесь:

```
def selection_sort(список): для fill_slot в
диапазоне (LEN(список) - 1, 0, -1):
        max_index = 0 для местоположения в
диапазоне(1, fill_slot + 1): если list[location]
> list[max_index]:
                max_index = местоположение
```

```
        list[fill_slot],list[max_index] = list[max_index],list[fill_slot] return
list
```

Производительность алгоритма сортировки выбора

Худшая производительность сортировки селекции — *O(N2).* Обратите внимание, что его худшая производительность похожа на сортировку пузырьков, и его не следует использовать для сортировки больших наборов данных. Тем не менее, выборочная сортировка — это лучше спроектированный алгоритм, чем пузырьковая сортировка, и его средняя производительность выше, чем в пузырьковой сортировке из-за уменьшения числа обменов.

Выбор алгоритма сортировки

Выбор правильного алгоритма сортировки зависит как от размера, так и от состояния текущих входных данных. Для небольших списков вводных вводов, которые отсортированы, использование продвинутого алгоритма внесёт ненужные сложности в код, при этом производительность значительно улучшается. Например, нам не нужно использовать сортировку слияния для небольших наборов данных. Bubble sort будет гораздо проще понять и реализовывать. Если данные частично отсортированы, мы можем воспользоваться этим, используя вставленную сортировку. Для больших наборов данных лучше всего использовать алгоритм сортировки слияния.

Введение в алгоритмы поиска

Эффективный поиск данных в сложных структурах данных — одна из важнейших функций. Самый простой способ, который не будет таким эффективным, — это поиск необходимых данных в каждой точке данных. Но по мере увеличения объема данных нам нужны более сложные алгоритмы для поиска данных. В этом разделе представлены следующие алгоритмы поиска:

- Линейный поиск
- Бинарный поиск
- Интерполяционный поиск

Давайте рассмотрим каждую из них подробнее.

Линейный поиск

Одна из самых простых стратегий поиска данных — просто проходить по каждому элементу в поисках цели. Каждая точка данных ищет совпадение, и при нахождении результатов возвращаются, и алгоритм выходит из цикла. В противном случае алгоритм продолжает искать, пока не достигнет конца данных. Очевидный недостаток линейного поиска в том, что он очень медленный из-за врожденного исчерпывающего поиска. Преимущество в том, что данные не требуют сортировки, как это требуют другие алгоритмы, представленные в этой главе. Давайте рассмотрим код линейного поиска:

```
def linear_search(элементы, предмет):
    index = 0
найдено = Ложно
# Сопоставьте значение с каждым элементом данных, при
этом индекс < len(элементы) и найдено — ложно: если
элементы[index] == элемент:
            найдено =
Истинное еще: индекс = индекс
+ 1 найденная доходность
```

Теперь рассмотрим результат предыдущего кода:

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

```
True
False
```

Обратите внимание, что при запуске `функции LinearSearch` при успешном `поиске данных возвращается истинное значение.`

Производительность линейного поиска

Как уже обсуждалось, линейный поиск — это простой алгоритм, выполняющий исчерпывающий поиск. В худшем случае его поведение — *O(N).* Больше информации можно найти на, возможно, вы также можете направить читателя на эту страницу: https://wiki.python.org/moin/TimeComplexity.

Бинарный поиск

Предварительным условием для алгоритма бинарного поиска являются отсортированные данные. Алгоритм итеративно делит список на две части и отслеживает самые низкие и самые высокие индексы, пока не найдёт нужное значение:

```
def binary_search(элементы, предмет):
    first = 0 last = len(элементы)-1,
a first<=last и не найден: midpoint
= (first + last)///2 if
list[midpoint] == элемент:
            return True, иначе: если
элемент < элементы[середина]:
                последний =
середина-1 иначе: первая =
середина + 1 возврат ложно
```

Результат выглядит следующим образом:

```
[48]: list = [12, 33, 11, 99, 22, 55, 90]
      sorted_list = BubbleSort(list)
      print(BinarySearch(list, 12))
      print(BinarySearch(list, 91))
```

```
True
False
```

Обратите внимание, что вызов `функции BinarySearch` вернёт `True`, если значение найдено в входном списке.

Производительность бинарного поиска

Бинарный поиск получил своё название потому, что при каждой итерации алгоритм делит данные на две части. Если данные содержат *N* элементов, для итерации потребуется максимум *O(logN*) шагов. Это означает, что алгоритм имеет *время выполнения* O(logN).

Интерполяционный поиск

Бинарный поиск основан на логике, что он фокусируется на средней части данных. Интерполяционный поиск более сложный. Он использует целевое значение для оценки положения элемента в отсортированном массиве. Давайте попробуем понять это на примере. Предположим, что мы хотим искать слово в английском словаре, например, слово river. Мы используем эту информацию для интерполяции и начала поиска слов, начинающихся на r. Более обобщённый интерполяционный поиск можно запрограммировать следующим образом:

```
def int_polsearch(list,x): idx0 = 0 idxn = (len(list) - 1) в то
время как idx0 < idxn и x >= list[idx0] и x <= list[idxn]:
# Найди середину
        mid = idx0 +int(((float(idxn - idx0)/( list[idxn] - list[idx0])) * ( x - list[idx0])))
# Сравните значение в середине с поисковым
значением, если list[mid] == x: вернуть True if
list[mid] < x: idx0 = mid + 1 return False
```

Результат выглядит следующим образом:

```
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
```

Обратите внимание, что перед использованием `IntPolsearch` массив сначала нужно отсортировать с помощью алгоритма сортировки.

Производительность интерполяционного поиска

Если данные распределены неравномерно, производительность алгоритма интерполяционного поиска будет низкой. Худшая производительность этого алгоритма — *O(N),* и если данные относительно однородны, то лучшая производительность — *O(log(log N)).*

Практическое применение

Возможность эффективного и точного поиска данных в данном хранилище данных критически важна для многих реальных приложений. В зависимости от выбора алгоритма поиска, возможно, сначала придётся отсортировать данные. Выбор правильных алгоритмов сортировки и поиска будет зависеть от типа и размера данных, а также от характера задачи, которую вы пытаетесь решить. Давайте попробуем использовать алгоритмы, представленные в этой главе, чтобы решить проблему сопоставления нового заявителя в иммиграционном департаменте определённой страны с историческими записями. Когда кто-то подаёт заявление на визу для въезда в страну, система пытается сопоставить заявителя с существующими историческими данными. Если найдено хотя бы одно совпадение, система дополнительно рассчитывает количество раз, когда человек был одобрен или отклонён в прошлом. С другой стороны, если совпадение не найдено, система классифицирует заявителя как нового и присваивает ему новый идентификатор. Возможность искать, находить и идентифицировать человека в исторических данных крайне важна для системы. Эта информация важна, потому что если кто-то уже подавал заявку раньше, и известно, что заявление было отклонено, это может негативно сказаться на текущей заявке этого человека. Аналогично, если известно, что чья-то заявка была одобрена в прошлом, это одобрение может увеличить шансы получить одобрение по текущей заявке. Обычно в исторической базе данных миллионы строк, и нам понадобится хорошо продуманное решение для сопоставления новых кандидатов в исторической базе данных. Предположим, что историческая таблица в базе данных выглядит следующим образом:

Личное удостоверение личности	Идентификатор заявления	Имя	Фамилия	Дата рождения	Дата принятия решения
45583	677862	John	Доу	2000-09-19	Утверждено 2018-08-07

54543	877653	Xman	Xsir	1970-03-10	Отклонено	2018-06-07
34332	344565	Агро	Вака	1973-02-15	Отклонено	2018-05-05
45583	677864	John	Доу	2000-09-19	Утверждено	2018-03-02
22331	344553	Кал	Сортировки	1975-01-02	Утверждено	2018-04-15

В этой таблице первая колонка, `Персональное удостоверение` , связана с каждым из уникальных претендентов в исторической базе данных. Если в исторической базе данных 30 миллионов уникальных заявителей, то будет 30 миллионов уникальных персональных ID. Каждое личное удостоверение личности идентифицирует заявителя в исторической системе баз данных. Второй столбец — `Application ID` . Каждый идентификатор приложения определяет уникальное приложение в системе. Человек мог подавать заявку более одного раза в прошлом. Это значит, что в исторической базе данных у нас будет больше уникальных идентификаторов приложений, чем персональных. У John Doe будет только одно личное удостоверение, но два идентификатора заявок, как показано в предыдущей таблице. В предыдущей таблице показана только выборка исторического набора данных. Предположим, что в нашем историческом наборе данных почти 1 миллион строк, содержащих записи заявок за последние 10 лет. Новые заявители постоянно приходят со средней скоростью около 2 заявок в минуту. Для каждого заявителя необходимо сделать следующее:

- Выдайте заявителю новый идентификатор заявки.
- Проверьте, есть ли совпадение с кандидатом в исторической базе данных.
- Если найдено совпадение, используйте личный идентификатор этого заявителя, как в
- исторической базе данных. Нам также нужно определить, сколько раз заявка была одобрена или отклонена в исторической базе данных. Если совпадения не найдены, нам нужно выдать новый личный удостоверение личности для этого человека.

Предположим, что новый человек приходит со следующими удостоверениями:

- `Имя: Джон`
- `Фамилия: Доу`
- `Дата рождения: 19.09.2000`

Теперь, как мы можем спроектировать приложение, которое сможет выполнять эффективный и экономически выгодный поиск? Одна из стратегий поиска нового приложения в базе данных может быть разработана следующим образом:

- Сортируйте историческую базу данных по `дате` рождения.
- Каждый раз, когда приходит новый человек, выдавайте ему новый идентификатор заявки.
- Возьмите все записи, соответствующие этой дате рождения. Это будет основной поиск.
- Из всех совпадений записей выполните дополнительный поиск по имени и фамилии. Если найдено совпадение, используйте `личное удостоверение` личности, чтобы обратиться к
- кандидатам. Рассчитайте количество одобрений и отказов.

Если совпадения не найдено, выдайте заявителю новый личный документ.

Давайте попробуем выбрать правильный алгоритм для сортировки исторической базы данных. Мы можем с уверенностью исключить сортировку пузырьков, так как размер данных огромен. Оболочная сортировка будет работать лучше, но только если у нас частично отсортированные списки. Поэтому сортировка по слиянию может быть лучшим вариантом для сортировки исторической базы данных.

Когда приходит новый человек, нам нужно найти и поискать его в исторической базе данных. Поскольку данные уже отсортированы, можно использовать как интерполяционный, так и бинарный поиск. Поскольку кандидаты, скорее всего, равномерно распределены по `правилам рождения` , мы можем безопасно использовать бинарный поиск. Изначально мы ищем по `дате` рождения, который возвращает набор заявок с одной датой рождения. Теперь нам нужно найти нужного человека среди небольшой группы людей, имеющих одну и ту же дату рождения. Поскольку мы успешно сократили данные до небольшого множества, любой алгоритм поиска, включая пузырчатую сортировку, может использоваться для поиска кандидата. Обратите внимание, что здесь мы немного упростили задачу вторичного поиска. Также нужно рассчитать общее количество одобрений и отказов, агрегируя результаты поиска, если найдено более одного совпадения. В реальной ситуации каждого человека необходимо идентифицировать во вторичном поиске с помощью нечеткого алгоритма поиска, так как имя и фамилия могут писаться немного иначе. Для реализации нечеткого поиска может потребоваться использование какого-то алгоритма расстояния, когда точки данных с сходством выше определённого порога считаются одинаковыми.

Краткое содержание

В этой главе мы представили набор алгоритмов сортировки и поиска. Мы также обсудили сильные и слабые стороны различных алгоритмов сортировки и поиска. Мы количественно оценили производительность этих алгоритмов и узнали, когда использовать каждый из них. В следующей главе мы изучим динамические алгоритмы. Мы также рассмотрим практический пример проектирования алгоритма и детали алгоритма ранжирования страниц. Наконец, мы изучим алгоритм линейного программирования.

OceanfPDF.com

4 Алгоритмы проектирования

Присоединяйтесь к Discord-пространству нашей книги

https:// packt.link/40Algos



В этой главе представлены основные концепции проектирования различных алгоритмов. В ней обсуждаются сильные и слабые стороны различных методов проектирования алгоритмов. Понимая эти концепции, мы научимся разрабатывать эффективные алгоритмы. Эта глава начинается с обсуждения различных вариантов, доступных нам при проектировании алгоритмов. Затем обсуждается важность характеристики конкретной проблемы, которую мы пытаемся решить. Далее используется знаменитая задача коммивояжера (TSP) в качестве примера использования и различные дизайнерские техники,

которые мы представим. Затем он вводит линейное программирование и обсуждает его применение. Наконец, он показывает, как линейное программирование может быть использовано для решения реальной задачи. К концу этой главы вы должны понять основные концепции проектирования эффективного алгоритма. В этой главе обсуждаются следующие концепции:

- Различные подходы к проектированию алгоритма
- Понимание компромиссов, связанных с выбором правильной конструкции алгоритма
- Лучшие практики формулировки реальной задачи
- Решение задачи оптимизации в реальном мире

Давайте сначала рассмотрим основные концепции проектирования алгоритма.

Введение основных понятий проектирования алгоритма

Алгоритм, согласно American Heritage Dictionary, определяется следующим образом: *«Конечный набор однозначных инструкций, которые при заданном наборе начальных условий могут выполняться в заданной последовательности для достижения определённой цели и имеют узнаваемый набор конечных условий.»*Проектирование алгоритма — это создание этого «конечного набора однозначных инструкций» максимально эффективным способом для «достижения определённой цели». Для сложной реальной задачи разработка алгоритма — это утомительная задача. Чтобы разработать хороший дизайн, сначала нужно полностью понять проблему, которую мы пытаемся решить. Мы начинаем с того, что определяем, что нужно сделать (то есть понимание требований), прежде чем рассматривать, как это будет реализовано (то есть с проектирования алгоритма).

Понимание задачи включает рассмотрение как функциональных, так и нефункциональных требований задачи. Давайте посмотрим, что это такое:

- Функциональные требования формально определяют входные и выходные интерфейсы задачи, которую мы хотим решить, а также связанные с ними функции. Функциональные требования помогают нам понять обработку данных, обработку данных и расчёты, которые необходимо реализовать для получения результата.
- Нефункциональные требования определяют ожидания относительно производительности и безопасности алгоритма.

Обратите внимание, что проектирование алгоритма направлено на наилучшее решение как функциональных, так и нефункциональных требований в данных условиях с учетом набора ресурсов, доступных для выполнения разработанного алгоритма. Чтобы дать хороший ответ, который соответствует функциональным и нефункциональным требованиям, наш дизайн должен учитывать следующие три аспекта, как обсуждается в *главе 1*, *Обзор алгоритмов:*

- Вопрос 1: Корректность: Даст ли разработанный алгоритм ожидаемый результат?
- Вопрос 2: Производительность: Является ли это оптимальным способом достижения этих результатов?

Вопрос 3: масштабируемость: как алгоритм будет работать на больших наборах данных?

В этом разделе давайте рассмотрим эти вопросы по одному.

Вопрос 1 — Правильность: даст ли разработанный алгоритм ожидаемый результат?

Алгоритм — это математическое решение реальной задачи. Чтобы быть полезным, он должен давать точные результаты. Проверка корректности алгоритма не должна быть второстепенной; Вместо этого это должно быть встроено в дизайн алгоритма. Прежде чем разрабатывать стратегию проверки алгоритма, нужно учитывать следующие два аспекта:

- **Определение** истины: Для проверки алгоритма нам нужны известные корректные результаты для заданного набора входных данных. Эти известные правильные результаты называются **истинами** в контексте задачи, которую мы пытаемся решить. Истина важна, так как используется как ориентир при итеративной работе над развитием алгоритма в сторону лучшего решения.
- **Выбор** метрик: Нам также нужно подумать о том, как мы будем количественно оценивать отклонение от определённой истины. Правильный выбор метрик поможет нам точно оценить качество нашего алгоритма.

Например, для алгоритмов контролируемого машинного обучения мы можем использовать существующие маркированные данные как истину. Мы можем выбрать одну или несколько метрик, таких как точность, воспоминание или точность, чтобы количественно оценить отклонение от истины. Важно отметить, что в некоторых случаях правильный вывод — это не одно значение. Вместо этого правильный выход определяется как диапазон для заданного набора входов. По мере работы над проектированием и разработкой нашего алгоритма цель будет — итеративно улучшать алгоритм до тех пор, пока он не достигнет диапазона, указанного в требованиях.

- **Рассмотрение крайних случаев:** крайний случай возникает, когда наш разработанный алгоритм работает на крайних параметрах работы. Крайний случай обычно встречается редко, но требует тщательного тестирования, так как он может привести к сбою нашего алгоритма. Нереберные случаи называются «счастливым путём», охватывающим все сценарии, которые обычно происходят, когда рабочие параметры находятся в пределах нормального диапазона. В подавляющем большинстве случаев алгоритм остаётся на «счастливом пути». К сожалению, невозможно вычислить все возможные крайние случаи для конкретного алгоритма, но следует рассмотреть как можно больше крайних случаев. Но без учёта и размышления о крайних случаях могут возникнуть проблемы.

Вопрос 2 – Производительность: является ли это оптимальным способом добиться таких результатов?

Вторая задача — найти ответ на следующий вопрос: Является ли это оптимальным решением и можем ли мы проверить, не существует ли другого решения, которое было бы лучше нашего решения? На первый взгляд этот вопрос кажется довольно простым для ответа. Однако для определённого класса алгоритмов исследователи десятилетиями безуспешно проверяли, является ли определённое решение, сгенерированное алгоритмом, лучшим и не

существует другого решения, способного обеспечить лучшую производительность. Поэтому важно сначала понять проблему, её требования и ресурсы, доступные для запуска алгоритма. Чтобы предложить наилучшее решение для определённой сложной задачи, нам нужно ответить на фундаментальный вопрос: стоит ли вообще стремиться найти оптимальное решение этой задачи? Если поиск и проверка оптимального решения — это очень трудоёмкая и сложная задача, то работоспособное решение может быть нашим лучшим вариантом. Эти приближённые рабочие решения являются *эвристиками*. Поэтому понимание проблемы и её сложности важно и помогает оценить потребности в ресурсах. Прежде чем мы углубляемся в это, давайте сначала определим несколько терминов:

- **Полиномиальный алгоритм:** если алгоритм имеет временную сложность *O(nk)*, мы называем его полиномиальным алгоритмом, где *k* — константа.
- **Сертификат:** Предлагаемое решение, созданное в конце итерации, называется **сертификатом**. По мере итеративного решения конкретной задачи мы обычно генерируем серию сертификатов. Если решение движется к сходимости, каждый сгенерированный сертификат будет лучше предыдущего. В какой-то момент, когда наш сертификат соответствует требованиям, мы выберем этот сертификат в качестве окончательного решения.

В *главе 1* «Обзор алгоритмов» мы представили нотацию Big O, которая может использоваться для анализа временной сложности алгоритма. В контексте анализа временной сложности мы рассматриваем следующие различные временные интервалы:

- Время, необходимое алгоритму для создания предлагаемого решения, называемого
- сертификатом (*tr) Время, необходимое для проверки предлагаемого решения (сертификата), ts*

Характеристика сложности задачи

За эти годы исследовательское сообщество разделило задачи на различные категории в зависимости от их сложности. Прежде чем мы пытаемся спроектировать решение задачи, логично сначала попытаться её охарактеризовать. В целом существует три типа задач:

- Тип 1: задачи, для которых можно гарантировать существование полиномиального алгоритма, который можно использовать для их решения
- Тип 2: задачи, для которых можно доказать, что их нельзя решить полиномиальным алгоритмом
- Тип 3: Задачи, для которых мы не можем найти полиномиальный алгоритм для их решения, но также не можем доказать, что полиномиальное решение для этих задач невозможно найти. Давайте рассмотрим различные классы задач:

- **Недетерминированный многочлен** (**NP):** Чтобы задача была NP-задачей, она должна соответствовать следующему условию:

Гарантируется, что существует полиномиальный алгоритм, который может быть использован для проверки оптимальности кандидата (сертификата).

- **Полиноминальные** (**P**): Это типы задач, которые можно рассматривать как подмножество NP. Помимо выполнения условия задачи NP, задачи P должны соответствовать ещё одному условию:

Гарантировано, что существует как минимум один полиномиальный алгоритм, который может быть использован для их решения. Взаимосвязь между задачами P и NP показана на *рисунке 4.1*:

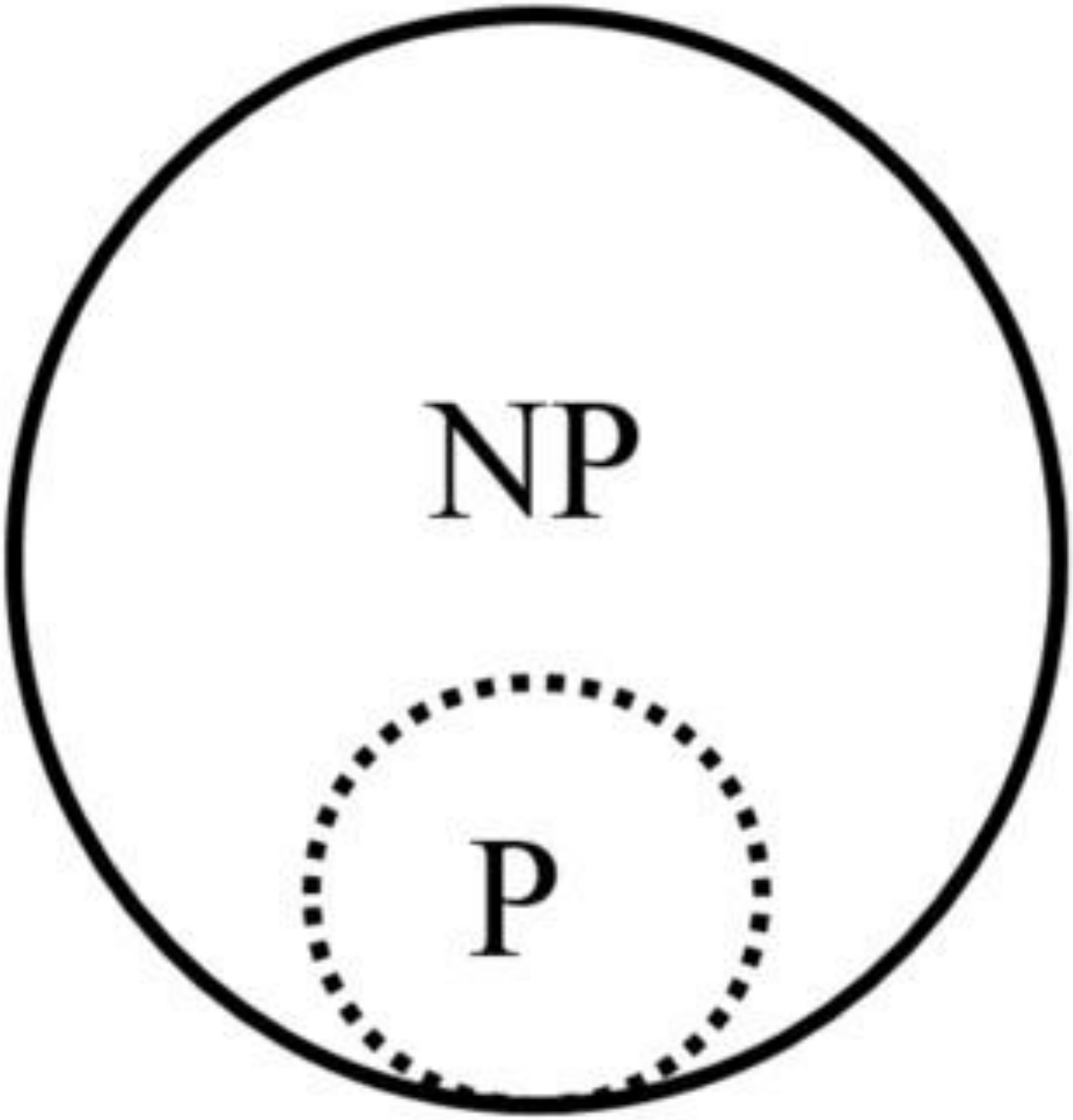


Рисунок 4.1: Взаимосвязь между задачами P и NP

Если проблема — это NP, то это тоже P? Это одна из величайших проблем в информатике, которая остаётся нерешённой. Компания Millennium Prize Problems, выбранная Институтом математики Клея, объявила о премии в размере 1 миллион долларов за решение этой задачи, поскольку она окажет значительное влияние на такие области, как искусственный интеллект, криптография и теоретические компьютерные науки:

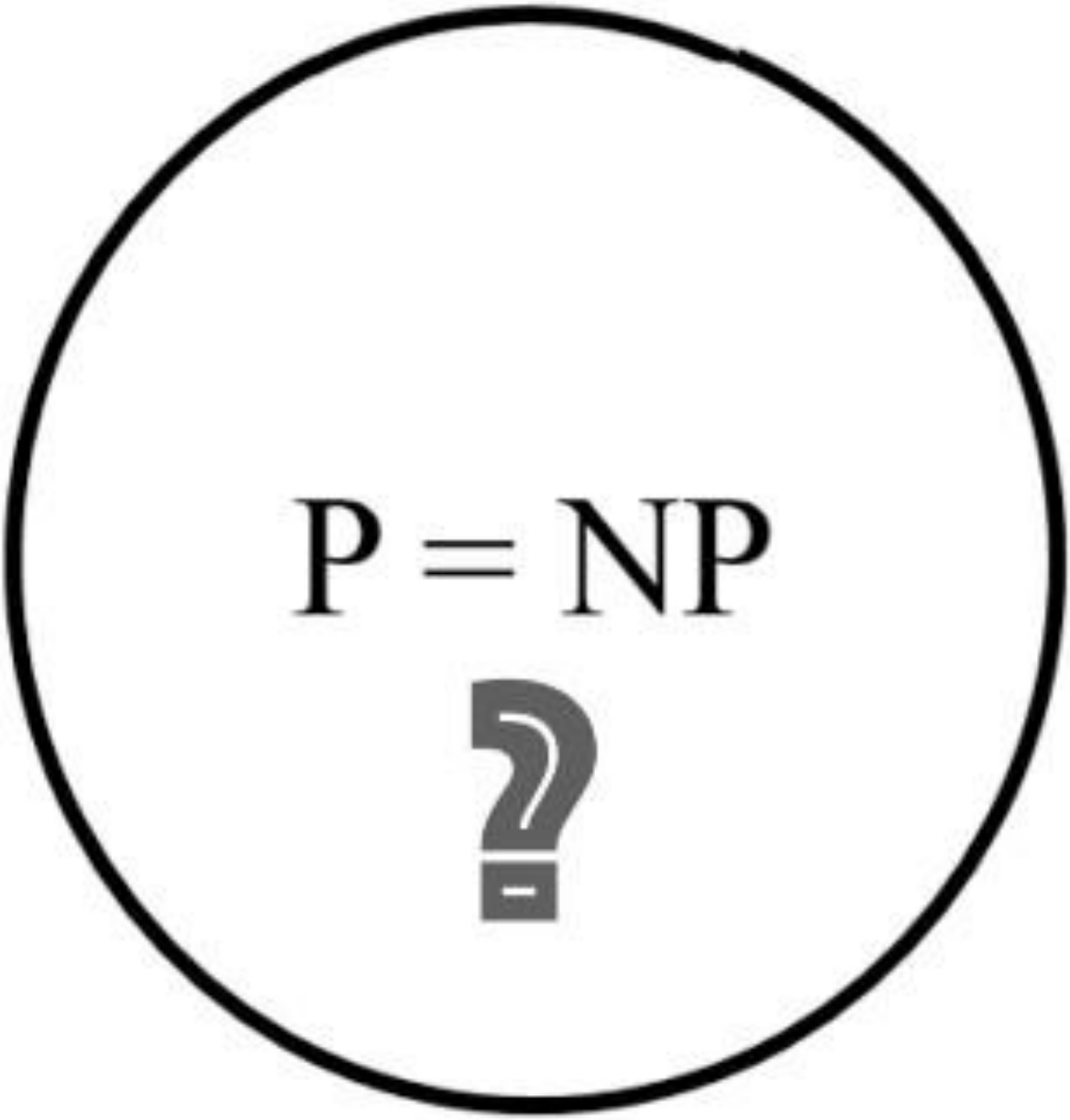


Рисунок 4.2: Является ли P = NP? Пока мы не знаем

Продолжим список различных классов задач:

- NP-полная: NP-полная категория содержит самые сложные задачи из всех NP-задач. NP-полная задача удовлетворяет следующим двум условиям:

Не существует известных полиномиальных алгоритмов для генерации сертификата. Существуют известные полиномиальные алгоритмы для проверки оптимальности предлагаемого сертификата.

- **NP-сложно**: NP-сложная категория содержит задачи, которые не менее сложны, чем любая задача из NP-категории, но сами по себе не обязательно должны быть в категории NP.

Теперь давайте попробуем нарисовать диаграмму, чтобы проиллюстрировать разные классы задач:

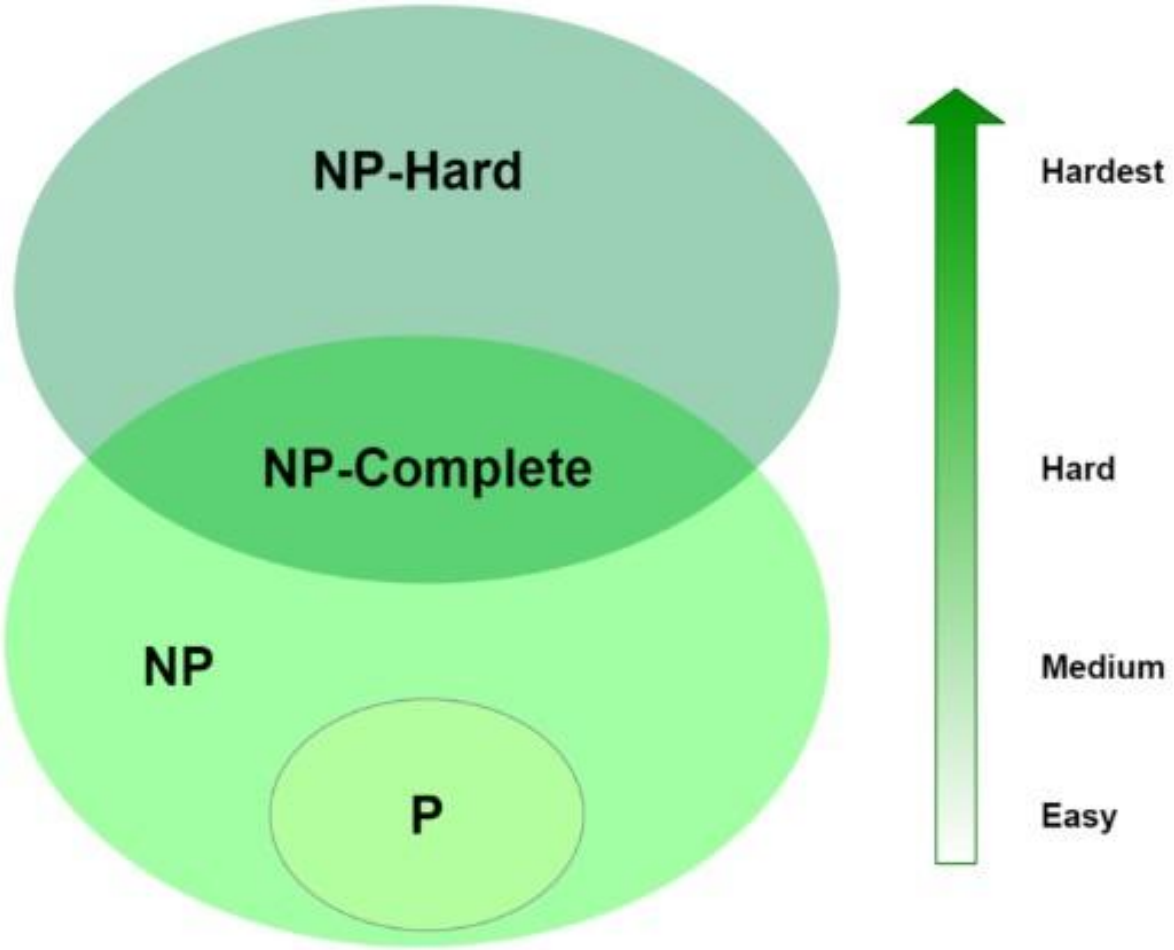


Рисунок 4.3: Взаимосвязь между P, NP, NP-Complete и NP-Hard

Обратите внимание, что исследовательское сообщество всё ещё предстоит доказать, что P = NP. Хотя это пока не доказано, крайне вероятно, что P ≠ NP. В этом случае для NP-полных задач полиномиального решения не существует. Обратите внимание, что предыдущая диаграмма основана на этом предположении.

Различие между P, NP, NP-полным и NP-твердым

К сожалению, различие между P, NP, NP-compete и NP-сложным не однозначно. Давайте подытожим и изучим несколько примеров, чтобы лучше понять концепции, обсуждаемые в этом разделе:

- **P**: Это класс задач, решаемых за полиномиальное время. Например:
  - Поиск по хеш-таблице
  - Алгоритмы по кратчайшему пути, такие как алгоритмы Джикстры
  - Линейные и двоичные алгоритмы поиска
- **NP-задача**: Задачи неразрешимы за полиномиальное время. Однако их решение можно проверить за полиномиальное время. Например: алгоритм шифрования RSA.

- **NP-сложно**: это сложные задачи, решения которых пока никто не может предложить. Но если решить решение, получится полиномиальное время. Например: оптимальная кластеризация с использованием алгоритма K-средних
- NP-полные: NP-полные задачи — самые «сложные» в NP. Они оба — NP-сложные и NP-сложные. Например:
  - Расчёт оптимального решения задачи коммивояжера

Инфобокс: Поиск решения для одного из классов (NP-сложного или NP-полного) означало бы решение для всех NP-жёстких/NP-полных задач.

Вопрос 3 — масштабируемость: как алгоритм будет работать на больших наборах данных?

Алгоритм обрабатывает данные определённым образом, чтобы получить результат. В целом, по мере увеличения размера данных требуется всё больше времени на их обработку и расчёт необходимых результатов. Термин «большие данные» иногда используется для приблизительного обозначения наборов данных, которые, как ожидается, будут сложны для работы инфраструктуры и алгоритмов из-за их объёма, разнообразия и скорости. Хорошо спроектированный алгоритм должен быть масштабируемым, то есть он должен быть спроектирован так, что, по возможности, он должен работать эффективно, использовать доступные ресурсы и получать правильные результаты в разумные сроки. Дизайн алгоритма становится ещё более важным при работе с большими данными. Чтобы количественно оценить масштабируемость алгоритма, нужно учитывать следующие два аспекта:

- **Увеличение требований к ресурсам по мере увеличения входных данных**: оценка такого требования называется анализом пространственной сложности.
- **Увеличение времени выполнения по мере увеличения входных данных**: оценка этого называется анализом временной сложности.

Обратите внимание, что мы живём в эпоху, определяемую взрывом данных. Термин «большие данные» стал распространённым, поскольку отражает размер и сложность данных, которые обычно требуются для обработки современными алгоритмами. Находясь на этапе разработки и тестирования, многие алгоритмы используют лишь небольшую выборку данных. При проектировании алгоритма важно учитывать аспект масштабируемости алгоритмов. В частности, важно тщательно анализировать (то есть тестировать или прогнозировать) влияние производительности алгоритма по мере увеличения размера наборов данных.

Эластичность облака и алгоритмическая масштабируемость

Облачные вычисления открыли новые возможности для решения требований алгоритма к ресурсам. Облачные вычислительные инфраструктуры способны предоставлять больше ресурсов по мере роста требований к обработке. Возможности облачных вычислений называются эластичностью инфраструктуры и теперь предоставляют больше возможностей для разработки алгоритма. При развертывании в облаке алгоритм может требовать дополнительных процессоров или виртуальных машин в зависимости от размера обрабатываемых данных. Типичный алгоритм глубокого обучения — хороший пример. Для обучения хорошей модели глубокого обучения требуется много маркированных данных. Для хорошо спроектированного алгоритма глубокого обучения обработка, необходимая для обучения модели глубокого обучения, прямо пропорциональна количеству примеров или близка к нему. При обучении модели глубокого обучения в облаке, по мере увеличения объема данных мы стараемся выделять больше ресурсов, чтобы поддерживать время обучения в управляемых пределах.

Понимание алгоритмических стратегий

Хорошо спроектированный алгоритм стремится максимально эффективно оптимизировать использование доступных ресурсов, разделяя задачу на более мелкие подзадачи, где это возможно. Существуют разные алгоритмические стратегии для проектирования алгоритмов. Алгоритмическая стратегия рассматривает следующие три аспекта списка алгоритмов, содержащие аспекты отсутствующего алгоритма. В этом разделе мы представим следующие три стратегии:

- Стратегия «разделяй и властвуй»
- Стратегия динамического программирования
- Стратегия жадного алгоритма

Понимание стратегии «разделяй и властвуй»

Одна из стратегий — найти способ разделить более крупную проблему на меньшие, которые можно решить независимо друг от друга. Подрешения, полученные этими подзадачами, затем объединяются для получения общего решения задачи. Это называется стратегией «разделяй и властвуй». Математически, если мы проектируем решение задачи (P) с n входами, требующей обработки набора данных d, мы делим задачу на k подзадач — от P1 до Pk. Каждая из подзадач обрабатывает раздел набора данных, d. Обычно мы используем обработку от P1 до Pk от d1 к dk. Давайте рассмотрим практический пример.

Практический пример — принцип разделяй и властвуй, применённый к Apache Spark

Apache Spark (https://spark.apболь. илиg/) — это фреймворк с открытым исходным кодом, используемый для решения сложных распределённых задач. Она реализует стратегию «разделяй и властвуй» для решения проблем. Чтобы обработать проблему, она делит её на различные подзадачи и обрабатывает их независимо друг от друга. Эти подзадачи могут выполняться на отдельных машинах, обеспечивая горизонтальное масштабирование. Мы продемонстрируем это на простом примере подсчёта слов из списка. Предположим, что у нас

есть следующий список слов: `words_list = ["Питон", "Ява", "Оттава", "Новости", "Ява", "Оттава"]`

Мы хотим рассчитать частоту каждого слова в этом списке. Для этого мы применим стратегию «разделяй и властвуй», чтобы эффективно решить эту проблему. Реализация «разделяй и властвуй» показана на следующей диаграмме:

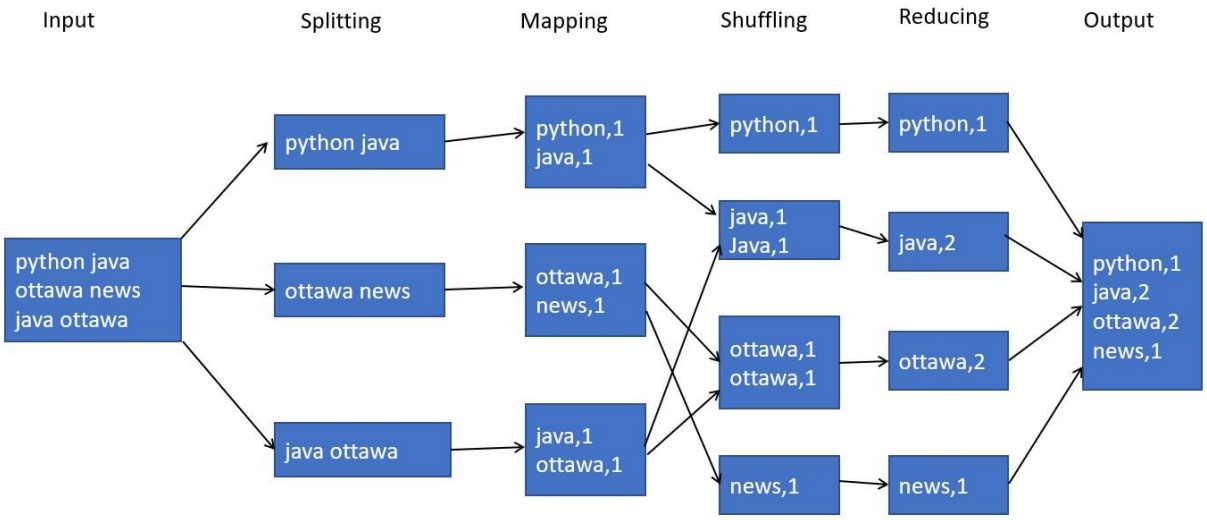


Рисунок 4.4: Разделяй и властвуй

Приведённая выше диаграмма показывает следующие фазы, на которые задача делится:

- **Разделение**: Входные данные делятся на разделы, которые могут обрабатываться независимо друг от друга. Это называется расщеплением. *В предыдущем рисунке* есть
- три раздела.

Отображение: Любая операция, которая может выполняться независимо на расщеплении, называется отображением. На предыдущей диаграмме операция отображения охватывает каждое слово из разбиения в пары ключ-значение.

В соответствии с тремя разделениями, параллельно работают три отображения.

- **Тасовка**: Тасовка — это процесс объединения похожих ключей. После того как похожие ключи объединены, можно запускать функции агрегации на их значениях. Обратите внимание, что *тасовка* — это операция, требующая высокой производительности, так как необходимо объединять похожие ключи, которые изначально можно распределять по сети.
- **Редуцирование**: Запуск агрегирующей функции на значениях схожих ключей называется редуцированием. На приведённой выше диаграмме нужно посчитать количество слов.

Давайте посмотрим, как мы можем написать код для реализации этого. Чтобы продемонстрировать стратегию разделяй и властвуй, нам нужна структура распределённых вычислений. Для этого мы запустим Python на Apache Spark:

- Во-первых, чтобы использовать Apache Spark, мы создадим контекст выполнения Apache Spark:

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

- Теперь давайте создадим примерный список с несколькими словами. Мы преобразуем этот список в нативную распределённую структуру данных Spark, называемую **устойчивым распределённым набором данных** (**RDD**):

```
wordsList = ['python', 'java', 'ottawa', 'ottawa', 'java','news']
wordsRDD = sc.parallelize(wordsList, 4) # Распечатать тип словRDD
print (wordsRDD.collect())
Он будет напечатан:

      [«Питон», «Ява», «Оттава», «Оттава», «Ява», «Новости»]
```

- Теперь давайте используем `функцию отображения, чтобы преобразовать слова в пару ключ-значение:`

```
wordPairs = wordsRDD.map(lambda w: (w, 1)) print
(wordPairs.collect())
```

Он будет напечатан:

```
[('Python', 1), ('Java', 1), ('Ottawa', 1), ('Ottawa', 1), ('Java', 1), ('News', 1)]
```

- Давайте используем `функцию reduce` для агрегирования и получения результата:

```
wordCountsCollected = wordPairs.reduceByKey(lambda x,y: x+y)
print(wordCountsCollected.collect())
```

Он печатает:

```
[('Питон', 1), ('Ява', 2), ('Оттава', 2), ('Новости', 1)]
```

Это показывает, как можно использовать стратегию разделяй и властвуй для подсчёта количества слов. Обратите внимание, что разделяй и властвуй полезен, когда задачу можно разделить на подзадачи, и каждую подзадачу можно хотя бы частично решить независимо от других подзадач. Это не лучший выбор для алгоритмов, требующих интенсивной итеративной обработки, таких алгоритмов оптимизации. Для таких алгоритмов подходит динамическое программирование, которое будет представлено в следующем разделе.

> Современные облачные вычислительные инфраструктуры, такие как Microsoft Azure, Amazon Web Services и Google Cloud, достигают масштабируемости, внедряя стратегию разделяй и властвуй прямо или косвенно за кулисами.

Понимание стратегии динамического программирования

В предыдущем разделе мы изучали метод «разделяй и властвуй», который является методом сверху вниз. В отличие от этого, динамическое программирование — это стратегия снизу вверх. Мы начинаем с самой маленькой подзадачи и продолжаем комбинировать решения. Мы продолжаем объединять, пока не придём к окончательному решению.

Динамическое программирование, как и метод «разделяй и властвуй», решает задачи путём комбинирования решений подзадач. Динамическое программирование — это стратегия, предложенная в 1950-х годах Ричардом Беллманом для оптимизации определённых классов алгоритмов. Обратите внимание, что в динамическом программировании слово «Программирование» в означает использование табличного метода и не связано с написанием кода. В отличие от стратегии разделяй и властвуй, динамическое программирование применимо, когда подзадачи не являются независимыми. Обычно он применяется к **задачам оптимизации** , в которых решение каждой подзадачи имеет определённое значение. Наша цель — найти решение с оптимальным значением. Алгоритм динамического программирования решает каждую подзадачу всего один раз, а затем сохраняет ответ в таблице, тем самым избегая работы по повторному вычислению ответа при каждом сталкивании с подзадачей.

Условия использования динамического программирования

Задача, которую мы пытаемся решить с помощью динамического программирования, должна иметь две характеристики.

А. **Оптимальная структура**: динамическое программирование даёт хорошие преимущества в производительности, когда задачу, которую мы пытаемся решить, можно разделить на подзадачи. Динамическое программирование работает за счёт комбинирования решений подзадач. Первый шаг к использованию динамического программирования для решения определённой задачи А — проверить, насколько проблема А имеет такие подструктуры.
Б. **Перекрывающиеся** подзадачи: он основан на интеллектуальном механизме кэширования, который пытается повторно использовать результаты сложных вычислений. Этот интеллектуальный механизм кэширования называется **мемоизацией**. Подзадачи частично связаны с вычислением, которое повторяется в этих подзадачах. Идея в том, чтобы выполнить этот вычисление один раз (что требует много времени), а затем использовать его повторно в других подзадачах. Это достигается с помощью мемоизации, что особенно полезно при решении рекурсивных задач, при которых можно многократно оценивать одни и те же входы. Обратите внимание, что **рекурсивная** функция решает конкретную задачу, вызывая свою копию и решая меньшие подзадачи исходных задач.

Вычисленные решения подзадач хранятся в таблице, так что их не нужно перечислять. Поэтому эта техника необходима там, где существует перекрывающаяся подзадача. Например

А. Бинарный поиск не имеет перекрывающихся подзадач.
Б. В то время как рекурсивные программы чисел Фибоначчи имеют множество пересекающихся подзадач.

Понимание жадных алгоритмов

Как следует из названия, жадный алгоритм относительно быстро даёт хорошее решение, но не может быть оптимальным. Как и динамическое программирование, жадные алгоритмы в

основном используются для решения задач оптимизации, где стратегия «разделяй и властвуй» нельзя. В жадном алгоритме решение рассчитывается по последовательности шагов. На каждом этапе делается локально оптимальный выбор.

Условия использования Жадного Программирования

Жадность — это стратегия, которая хорошо работает при решении задач с двумя характеристиками:

А. Глобальный от локального: Глобальный оптимум можно получить, выбрав локальный оптимум.
Б. Оптимальная подструктура: оптимальное решение задачи формируется на основе оптимальных решений подзадач.

Чтобы понять алгоритм жадности, давайте сначала определим два термина:

- **Алгоритмические накладные расходы**: Когда мы пытаемся найти оптимальное решение определённой задачи, это занимает время. По мере того как задачи, которые мы пытаемся оптимизировать, становятся всё сложнее, время поиска оптимального решения также увеличивается. Мы представляем алгоритмические накладные расходы с помощью *Ωi.*
- **Дельта от оптимального**: для заданной задачи оптимизации существует оптимальное решение. Обычно мы итеративно оптимизируем решение с помощью выбранного алгоритма. Для заданной задачи всегда существует совершенное решение, называемое **оптимальным решением** текущей задачи. Как уже обсуждалось, исходя из классификации задачи, которую мы пытаемся решить, оптимальное решение может быть неизвестным или что на его расчёт и проверку потребуется неразумно много времени. При условии, что оптимальное решение известно, разница от оптимального для текущего решения в *i-й итерации называется* **дельта от** оптимального и представлена как *Δi.*

Для сложных задач у нас есть две возможные стратегии:

- **Стратегия 1:** Потратить больше времени на поиск решения, наиболее близкого к оптимальному, чтобы *Δi был как можно меньше.*
- **Стратегия 2:** Минимизировать алгоритмические накладные расходы, *Ωi . Используйте быстрый и простой подход и просто работающее решение.*

Жадные алгоритмы основаны на стратегии 2, где мы не пытаемся найти глобальный оптимум и предпочитаем минимизировать накладные расходы алгоритма. Использование жадного алгоритма — это быстрая и простая стратегия поиска глобального оптимального значения для многоступенчатых задач. Он основан на выборе локальных оптимальных значений без попыток проверить, являются ли локальные оптимальные значения глобально оптимальными. Как правило, если нам не повезёт, жадный алгоритм не даст значение, которое можно считать глобально оптимальным. Однако поиск глобального оптимального значения занимает много времени. Таким образом, жадный алгоритм быстрее по

сравнению с алгоритмами разделяй и властвуй и динамическим программированием. Обычно жадный алгоритм определяется следующим образом:

- Предположим, что у нас есть набор данных, *D*. В этом наборе данных выберите элемент
- k.

Предположим, что кандидатом или сертификатом является *S.* Рассмотрим возможность включения *k* в решение — *S*. Если его можно включить, то решение — *Union(S, e)*.

Повторяйте процесс, пока *S* не заполнится или D не исчерпается. Пример: алгоритм Дерева классификации и регрессии (CART) — это жадный алгоритм, который ищет оптимальное разделение на верхнем уровне. Процесс повторяется на каждом последующем уровне. Обратите внимание, что алгоритм CART не рассчитывает и не проверяет, приведёт ли расщепление к минимальной возможной чистоте на несколько уровней ниже. CART использует жадный алгоритм, потому что поиск оптимального дерева известен *как NP-полная* задача. У неё алгоритмическая сложность составляет *O*(exp(*m*)) время.

Практическое применение — решение задачи путешествующего Сейлмана (TSP)

Давайте сначала рассмотрим формулировку задачи для TSP — хорошо известную задачу, введённую как вызов в 1930-х годах. TSP — это сложная задача, связанная с NP. Для начала мы можем случайным образом сгенерировать тур, который соответствует условию посещения всех городов, не заботясь о оптимальном решении. Затем мы можем работать над улучшением решения с каждой итерацией. Каждый тур, сгенерированный в итерации, называется кандидатным решением (также называемым сертификатом). Доказательство оптимальности сертификата требует экспоненциального увеличения времени. Вместо этого используются различные решения на основе эвристик, которые создают экскурсии, близкие к оптимальным, но не оптимальные. Коммивояжер должен посетить определённый список городов, чтобы выполнить свою работу:

ВХОД Список из n городов (обозначаемых как V) и расстояний между каждой парой городов, d ij ($1 \le i, j \le n$)

ВЫВОД Самый короткий тур, который посещает каждый город ровно один раз и возвращается в исходный город

Обратите внимание на следующее: Расстояния между городами в списке известны,

- Каждый город в данном списке нужно посетить *ровно* один раз.

Можем ли мы составить план путешествия для продавца? Какое решение будет оптимальным, чтобы минимизировать общее расстояние, пройденное коммивояжером? Ниже приведены расстояния между пятью канадскими городами, которые мы можем использовать для TSP: **Оттава, Монреаль, Кингстон, Торонто Садбери.**

Оттава	-	199	196	450	484

Монреаль 199	-	287	542	680
Кингстон 196	287	-	263	634
Toronto 450	542	263	-	400
Садбери 484	680	634	400	-

- Обратите внимание, что цель — получить экскурсию, которая начинается и заканчивается в первом городе. Например, типичный тур может быть Оттава–Садбери–Монреаль–Кингстон–Торонто–Оттава стоимостью *484 + 680 + 287 + 263 + 450 = 2 164*. Это тот тур, в котором продавец должен преодолеть минимальное расстояние? Какое решение будет оптимальным, чтобы минимизировать общее расстояние, пройденное коммивояжером? Я оставляю вам подумать и посчитать.

Использование стратегии грубой силы

Первое решение, которое приходит на ум для решения TSP, — это грубая сила, чтобы придумать самый короткий путь, при котором продавец посещает каждый город ровно один раз и возвращается в исходный город.
Таким образом, стратегия грубой силы работает следующим образом:

А. Оцените все возможные экскурсии.
Б. Выбираем тот, для которого мы получаем самое короткое расстояние.

Проблема в том, что для n городов существует (n-1)! возможные экскурсии. Это значит, что пять городов произведут 4! = 24 тура, и мы выберем тот, который соответствует наименьшему расстоянию. Очевидно, что этот метод сработает только тогда, когда у нас не слишком много городов. По мере увеличения числа городов стратегия грубой силы становится неразрешимой из-за большого количества перестановок, возникающих при использовании этого подхода. Давайте посмотрим, как можно реализовать стратегию грубой силы в Python. Во-первых, обратите внимание, что экскурсия {1,2,3} представляет собой экскурсию по городу из города 1 в город 2 и город 3. Общее расстояние в туре — это общее расстояние, пройденное за один тур. Мы предположим, что расстояние между городами — это самое короткое расстояние между ними (то есть евклидово расстояние). Сначала определим три функции полезности:

- `distance_points` : Вычисляет абсолютное расстояние между двумя точками
- `distance_tour` : Вычисляет общее расстояние, которое продавец должен преодолеть
- за заданный тур `generate_cities` : Случайным образом генерирует набор *из n* городов, расположенных в прямоугольнике шириной `500` и высота `300`

Рассмотрим следующий код:

```
Импорт случайным
из itertools импортные перестановки
alltours = перестановки
```

В предыдущем коде мы реализовали alltours из `функции перестановок` пакета itertools. Мы также обозначили расстояние комплексным числом. Это означает следующее:

- Вычисление расстояния между двумя городами, *a* и *b*, так же просто, как
- `расстояние (a,b)`. Мы можем создать *n* количества городов, просто вызвав `generate_cities(n )`.

```
distance_tour(aTour):
    return sum(distance_points(aTour[i - 1], aTour[i])
для i в диапазоне(len(aTour))) aCity = комплекс
DEF distance_points(Первый, Вторая): Возврат ABS(Первый -
Вторый) Def generate_cities (number_of_cities): Посев=111;
ширина=500; height=300 random.seed((number_of_cities, seed))
    return frozenset(aCity(random.randint(1, ширина), random.randint(1, высота)) для
с в диапазоне(number_of_cities))
```

Теперь определим функцию, `brute_force` , которая генерирует все возможные туры по городам. После того как он сгенерирует все возможные маршруты, он выберет тот, у которого самая короткая дистанция:

```
def brute_force(города):
    «Сгенерируйте все возможные туры по городам и выберите самый короткий тур.»
    Return shortest_tour(Alltours(Cities)) def
shortest_tour(Tours): return min(Tours, key=distance_tour)
```

Теперь давайте определим функции полезности, которые помогут нам построить цитаты. Определим следующие функции:

- `visualize_tour` : Прокладывает все города и связи в конкретном туре. Он также подчёркивает город
- С того места, где начался тур.`visualize_segment` : Используется `visualize_tour` для построения цитат и ссылок в сегменте.

Посмотрите на следующий код:

```
%matplotlib inline import
matplotlib.pyplot as plt def
visualize_tour(tour, style='bo-'):
    if len(tour) > 1000: plt.figure(figsize=(15, 10)) start
= тур[0:1]
    visualize_segment(тур + старт, стиль)
visualize_segment(старт, 'rD')
     def visualize_segment (сегмент,
стиль='bo-'):
    plt.plot([X(c) для с в сегменте], [Y(c) для с в сегменте], стиль, clip_on=False)
plt.axis('масштабированный') plt.axis('off')

def X(город): «Ось X»; return city.real def
Y(city): «Ось Y»; Return city.imag
```

Реализуем функцию `tsp()` , которая выполняет следующее:

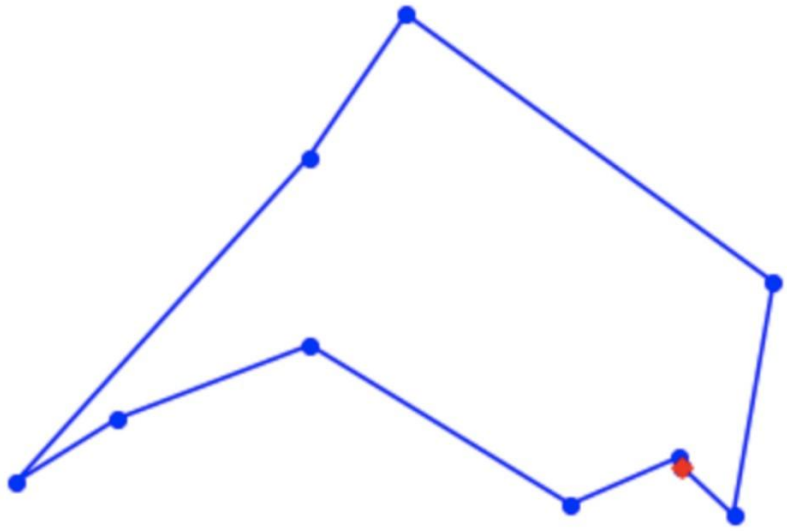
- Генерирует тур на основе алгоритма и количества запрошенных городов
- Вычисляет время, потребовавшееся для запуска
- алгоритма, Генерирует график

После `определения tsp()` мы можем использовать его для создания тура:

```
от времени импорта времени из
коллекций импорта Counter def
tsp(algorithm,cities): t0 =
clock() tour =
algorithm(cities) t1 = clock()
    # Каждый город появляется ровно один раз в туре
assert Counter(tour) == Counter(cities)
visalize_tour(тур)
    print("{}:{} города => длина тура {;. 0f} (в {:.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)
```



Обратите внимание, что мы использовали его для создания тура по 10 городам. При n = 10 он погенерирует (10-1)! = 362 880 возможных перестановок. Если n увеличивается, количество перестановок резко увеличивается, и метод грубой силы нельзя использовать.

Использование жадного алгоритма

Если мы используем жадный алгоритм для решения TSP, то на каждом этапе мы можем выбрать город, который кажется разумным, вместо того чтобы искать город для посещения, который приведёт к лучшему общему пути. Поэтому, когда нам нужно выбрать город, мы просто выбираем ближайший город, не утруждая себя проверкой, что этот выбор приведёт к глобально оптимальному пути. Подход жадного алгоритма прост:

А. Начинайте с любого города.
Б. На каждом этапе продолжайте строить экскурсию, переезжая в следующий город, где ближайший район ещё не бывал.

B. Повторите *шаг 2.*

Давайте определим функцию с названием `greedy_algorithm` , которая может реализовать такую логику:

```
def greedy_algorithm(города, начало=Нет):
    city_ = старт или первый тур по
городам = [city_]
    unvisited = set(cities - {city_}) при
непосещаемости:
        city_ = nearest_neighbor(city_, не
посещаем) тур.append(city_)
unvisited.remove(city_) возвращающийся тур
def first(collection): return next(iter(collection)) def
nearest_neighbor(city_a, cities): return min(cities, key=lambda city_:
distance_points(city_, city_a))
```

Теперь давайте воспользуемся greedy_algorithm, чтобы создать тур по 2000 городам:

```
tsp(greedy_algorithm, generate_cities(2000))
```

```
nn: 1991 cities ⇒ tour length 15846 (in 0.514 sec)
```

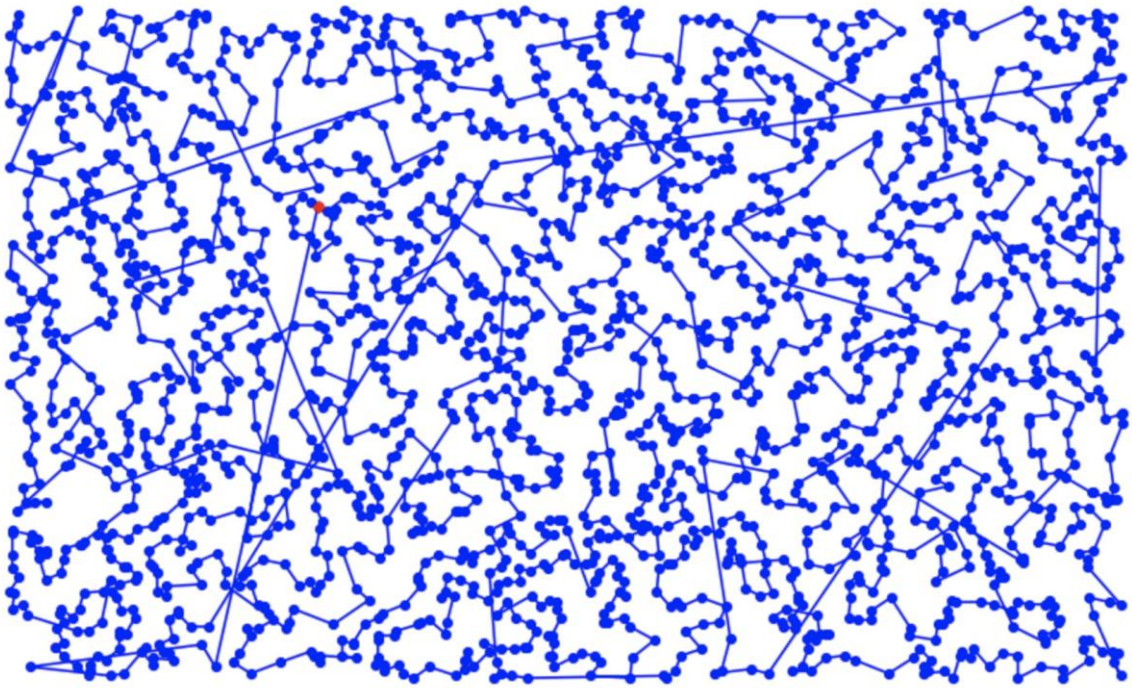


Рисунок 1: Города, отображаемые в Jupyter Notebook

Обратите внимание, что на создание тура по 2000 городам ушло всего 0,514 секунды. Если бы мы использовали метод грубой силы, он бы сгенерировал *(2000-1)!* Перестановки, что почти бесконечно. Обратите внимание, что жадный алгоритм основан на эвристиках, и нет доказательств, что решение будет оптимальным. Теперь давайте рассмотрим дизайн алгоритма PageRank.

Презентация алгоритма PageRank

В качестве практического примера рассмотрим алгоритм PageRank, который изначально использовался Google для ранжирования результатов поиска по пользовательскому запросу. Он генерирует число, которое количественно оценивает важность результатов поиска в контексте выполненного пользователем запроса. Его разработали два аспиранта, Ларри Пейдж и Сергей Брин, в Стэнфорде в конце 1990-х годов, которые также основали Google. *Алгоритм PageRank был назван в честь Ларри ПейджВпервые формально определили задачу, для которой изначально был разработан PageRank.*

Определение задачи

Каждый раз, когда пользователь вводит запрос в поисковую систему в интернете, обычно получается большое количество результатов. Чтобы результаты были полезны для конечного пользователя, важно ранжировать веб-страницы по определённым критериям. Отображаемые результаты используют это ранжирование для суммирования результатов для пользователя и зависят от критериев, определённых используемым алгоритмом.

Реализация алгоритма PageRank

Во-первых, при использовании алгоритма PageRank используется следующее представление:

- Веб-страницы представлены узлами в
- ориентированном графе. Рёбра графа соответствуют

гиперссылкам.

Самая важная часть алгоритма PageRank — найти лучший способ расчёта важности каждой страницы, который возвращается результатами запроса. Ранг конкретной веб-страницы в сети рассчитывается как вероятность того, что человек, случайным образом переходящий по рёбрам (то есть кликающий по ссылкам), попадёт на эту страницу. Кроме того, этот алгоритм параметризируется коэффициентом демпфирования альфа, который имеет значение по умолчанию 0,85. Этот фактор демпфирования — вероятность того, что пользователь продолжит кликать. Обратите внимание, что страница с самым высоким PageRank является самой привлекательной: независимо от того, с чего человек начинает, эта страница с наибольшей вероятностью станет её конечной целью. Алгоритм требует множества итераций или прохождения через набор веб-страниц, чтобы определить правильную важность (или значение PageRank) каждой страницы. Для вычисления числа от `0` до `1` , которое может количественно оценить важность конкретной страницы, алгоритм включает информацию из следующих двух компонентов:

- **Информация, специфическая для запроса, введённого пользователем**: этот компонент оценивает, в контексте запроса, введённого пользователем, насколько релевантно содержание веб-страницы.
  Содержание страницы напрямую зависит от автора страницы.
- **Информация, не относящаяся к запросу, введённому** пользователем: этот компонент пытается количественно оценить важность каждой веб-страницы в контексте её ссылок, просмотров и окрестности. Окружение веб-страницы — это группа веб-страниц,

непосредственно связанных с определённой страницей. Этот компонент сложно вычислить, так как веб-страницы неоднородны, и разработать критерии, которые можно применить в интернете, сложно.

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

```
импорт numpy как np импорт
networkx как nx импорт
matplotlib.pyplot как plt
```

Обратите внимание, что сеть — из https://networkx.org/. Для демонстрации предположим, что мы анализируем всего пять веб-страниц в сети. Давайте назовём этот набор страниц `my_pages` и вместе они входят в сеть под названием `my_web` :

```
my_web = nx. DiGraph()
my_pages = диапазон(1,5)
```

Теперь давайте подключим 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)]
my_web.add_nodes_from(my_pages) my_web.add_edges_from(connections)
```

Теперь давайте построим этот график:

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

Она создаёт визуальное представление нашей сети, следующим образом:

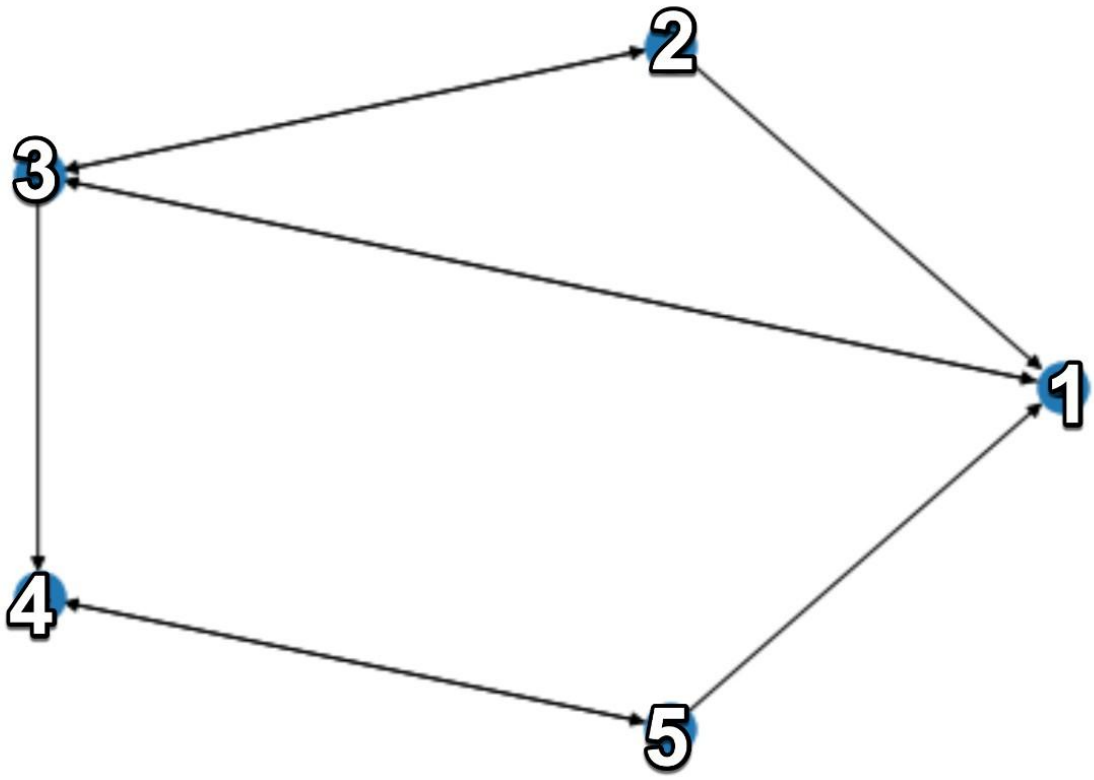


Рисунок 4.7: Визуальное представление сети

В алгоритме PageRank шаблоны веб-страницы содержатся в матрице, называемой переходной матрицей. Существуют алгоритмы, которые постоянно обновляют матрицу переходов, чтобы фиксировать постоянно меняющееся состояние веба. Размер матрицы переходов равен n x n, где n — количество узлов. Числа в матрице — это вероятность того, что посетитель перейдёт к этой ссылке из-за исходящей связи. В нашем случае предыдущий график показывает статическую паутину, которая у нас есть. Давайте определим функцию, которую можно использовать для создания матрицы переходов:

```
def create_page_rank(a_graph):
nodes_set = len(a_graph) M = nx.to
numpy_matrix(a_graph)
    наружу = np.squeeze(np.asarray (np. сумма (M, axis=1)))=
вероятно, наружу = np.array(
        [1.0/счёт
         если count>0 иначе 0.0 для счёта наружу]) 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 —
субстохастический возврат G, p
```

Обратите внимание, что эта функция вернёт G, которая представляет собой матрицу переходов для нашего графа. Давайте погенерируем матрицу переходов для нашего графа:

```
G,p = create_page_rank(my_web)
```

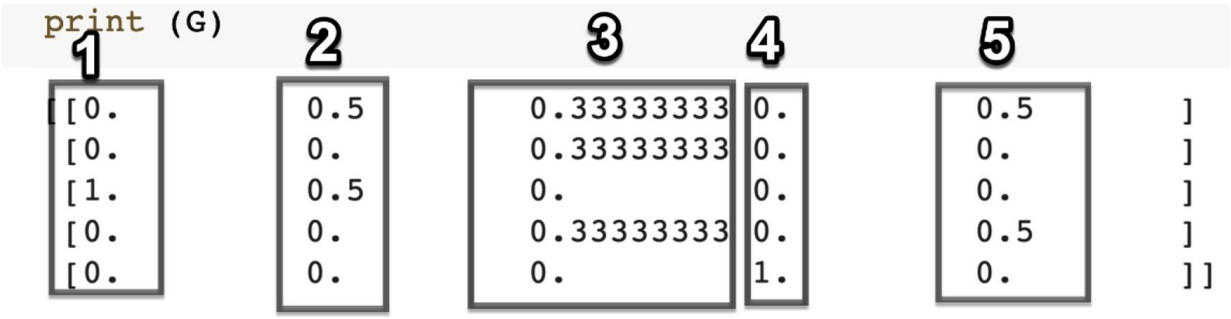


Рисунок 4.8: Матрица переходов

Обратите внимание, что матрица переходов для нашего графа равна 5 x 5. Каждый столбец соответствует каждому узлу графа. Например, столбец 2 относится ко второму узлу. Вероятность того, что посетитель переместится из узла 2 в узел 1 или узел 3, составляет 0,5. Обратите внимание, что диагональ матрицы переходов `равна 0` , как в нашем графе нет исходящей связи от узла к самому себе. В реальной сети это возможно. Обратите внимание, что матрица переходов — это разрежённая матрица. По мере увеличения числа узлов большинство его значений будут равны `0` . Таким образом, структура графа извлекается в виде *переходной матрицы*. В матрице транзакций, представляющей узлы в столбцах и строках:

- **Столбцы:** Указывает узел, на котором находится веб-серфер
- **Строки:** Указывает вероятность того, что серфер посетит другие узлы из-за исходящих связей

В реальной паутине матрица переходов, питающая алгоритм PageRank, строится за счёт непрерывного исследования ссылок пауками.

Понимание линейного программирования

Многие реальные задачи связаны с максимизацией или минимизацией цели с определёнными заданными ограничениями. Один из подходов — задать цель как линейную функцию некоторых переменных. Мы также формулируем ограничения на ресурсы как равенства или неравенства по этим переменным. Этот подход называется задачей линейного программирования. 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 планированием логистики снабжения и вместимости войск, работая на ВВС США. В конце Второй мировой войны Данциг начал работать в Пентагоне и развил свой алгоритм до техники, которую он назвал линейным программированием. Он использовался для военного боевого планирования. Сегодня он используется для решения важных реальных задач, связанных с минимизацией или максимизацией переменной на основе определённых ограничений. Некоторые примеры таких задач следующие:

- Минимизация времени на ремонт автомобиля в мастерской на основе ресурсов
- Распределение доступных распределённых ресурсов в распределённой вычислительной среде для минимизации времени отклика
- Максимизация прибыли компании на основе оптимального распределения ресурсов внутри компании

Формулировка задачи линейного программирования

Условия использования линейного программирования следующие:

- Мы должны иметь возможность сформулировать задачу через
- набор уравнений. Переменные, используемые в уравнении, должны быть линейными.

Определение целевой функции

Обратите внимание, что цель каждого из трёх предыдущих примеров заключается в минимизации или максимизации переменной. Эта цель математически формулируется как линейная функция других переменных и называется целевой функцией. Цель задачи линейного программирования — минимизировать или максимизировать целевую функцию, оставаясь в пределах заданных ограничений.

Определение ограничений

Когда пытаешься минимизировать или максимизировать что-то, существуют определённые ограничения в реальных задачах, которые нужно уважать. Например, при попытке минимизировать время ремонта автомобиля, нужно учитывать, что доступное количество механиков ограничено. Определение каждого ограничения с помощью линейного уравнения является важной частью формулировки задачи линейного программирования.

Практическое применение — планирование пропускной способности с линейным программированием

Рассмотрим практический сценарий, где линейное программирование может быть использовано для решения реальных задач. Предположим, что мы хотим максимизировать прибыль современного завода, который производит два разных типа роботов:

- **Продвинутая модель (A):** обеспечивает полную функциональность. Изготовление каждой единицы продвинутой модели приносит прибыль в размере $4,200.
- **Базовая модель (B):** Она предоставляет только базовую функциональность. Изготовление каждой единицы базовой модели приносит прибыль в размере $2,800.

Для создания робота требуется три типа людей. Точное количество дней, необходимое для производства робота каждого типа, следующее:

Тип роботаТехник — специалист по искусственному интеллекту — инженер

Робот A: продвинутая модель 3 дня 4 дня 4 дня

Робот B: базовая модель 2 дня 3 дня 3 дня

Завод работает по 30-дневным циклам. Один специалист по ИИ доступен в течение 30 дней в цикле. Каждый из двух инженеров возьмёт по 8 дней отдыха за 30 дней. Таким образом, инженер доступен только 22 дня в цикле. Существует один техник на 20 дней в течение 30-дневного цикла. В следующей таблице показано количество сотрудников на заводе:

	ТехникСпециалист по искусственному интеллекту Инженер		
Количество участников	1	1	2

Общее количество дней в цикле 1 x 20 = 20 дней 1 x 30 = 30 дней 2 x 22 = 44 дня

Это можно смоделировать следующим образом:

- Максимальная прибыль =
- 4200A + 2800B Это зависит от
  - следующего условия:
  - A ≥ 0: Количество произведенных продвинутых роботов может составлять `0` и
  - более.
  - Б ≥ 0: Количество выпущенных базовых роботов может составлять `0` и более.
  - 3A + 2B ≤ 20: Это ограничения доступности техника.

4A+3B ≤ 30: Это ограничения доступности специалиста по ИИ. 4A+ 3B ≤ 44: Это ограничения доступности машинистов.

Сначала импортируем Python-пакет с названием `pulp` , который используется

для реализации ; Линейное программирование: `импортная пульпа`

Затем мы вызываем `функцию LpProblem` в этом пакете для создания класса задачи. Мы называем `экземпляр задачей максимизации` прибыли:

```
# Воплотить нашу модель класса проблемы = целлюлоза.
LpProblem("Profit_maximising_problem", pulp. LpMaximize)
```

Затем определяем две линейные переменные — `A` и `B` . Переменная `A` обозначает количество произведенных продвинутых роботов, а переменная `B` — количество базовых роботов, которые производятся:

```
A = пульпа. LpVariable('A', lowBound=0, cat='Целое число')
Б = пульпа. LpVariable('B', lowBound=0, cat='Целое число')
```

Мы определяем целевую функцию и ограничения следующим образом:

```
# Целевая функция
модель += 5000 * A + 2500 * B, «Прибыль»
# Ограничения
модель += 3 * A + 2 * B < 20
моделей += 4 * A + 3 * B < 30
моделей += 4 * A + 3 * B <= 44
```

Используем `функцию решения` для генерации решения:

```
# Реши нашу проблемную модель.
реши() пульпу.
LpStatus[model.status]
```

Затем выводим значения `A` и `B` и значение целевой функции:

```
# Распечатать значения переменных
принятия решения печатать
(A.varValue) распечатать
(B.varValue)
```

Результат такой:

```
6.0
1.0
# Распечатать нашу целевую функцию,
значение печатать
(pulp.value(model.objective))
Он печатает:
32500.0
```

> Линейное программирование широко используется в производственной отрасли для поиска оптимального количества продуктов, которые должны быть использованы для оптимизации использования доступных ресурсов.

И вот мы подходим к концу этой главы! Давайте подытожим то, чему мы научились.

Краткое содержание

В этой главе мы изучали различные подходы к проектированию алгоритма. Мы рассмотрели компромиссы, связанные с выбором правильного дизайна алгоритма. Мы рассмотрели лучшие практики формулирования реальных проблем. Мы также научились решать реальные задачи оптимизации. Уроки, извлечённые из этой главы, можно использовать для реализации хорошо продуманных алгоритмов. В следующей главе мы сосредоточимся на алгоритмах на основе графов. Начнём с рассмотрения различных способов представления графов. Затем мы изучим методы установления окружения вокруг различных точек данных для проведения конкретного исследования. Наконец, мы изучим оптимальные способы поиска информации по графам.

OceanfPDF.com

5 Алгоритмы графов

Присоединяйтесь к Discord-пространству нашей книги

https:// packt.link/40Algos



Существует класс вычислительных задач, которые лучше всего представить с помощью графов. Такие задачи можно решить с помощью класса алгоритмов, называемых графовыми алгоритмами. Например, алгоритмы графов могут использоваться для эффективного поиска значения в графическом представлении данных. Для эффективной работы этим алгоритмам сначала нужно выявить структуру графа. Им также нужно найти правильную стратегию следования рёбрам графа, чтобы читать данные, хранящиеся в вершинах. Поскольку алгоритмы графов должны искать значения для работы, эффективные стратегии поиска лежат в центре проектирования эффективных алгоритмов графов. Использование алгоритмов графов — один из самых эффективных способов поиска информации в сложных, взаимосвязанных структурах данных, связанных осмысленными связями. В современную эпоху больших данных, социальных сетей и распределённых данных такие методы становятся всё более важными и полезными. В этой главе мы начнём с представления основных концепций, лежащих в основе алгоритмов графов. Затем мы представим основы теории анализа сетей. Далее мы рассмотрим различные методы, которые можно использовать для прохождения графов. Наконец, мы рассмотрим кейс-стади, показывающий, как алгоритмы графов могут использоваться для выявления мошенничества. В этой главе мы рассмотрим следующие концепции:

- Различные способы представления графов
- Введение анализа теории сетей
- Понимание обходов графов
- Кейс-стади: аналитика мошенничества

Методы установления окрестности в нашем проблемном пространстве

К концу этой главы вы хорошо поймёте, что такое графы и как с ними работать для представления взаимосвязанных структур данных, а также извлечения информации из сущностей, связанных прямыми или косвенными связями, а также использования их для решения сложных реальных задач.

Представления графов

Граф — это структура, представляющая данные в терминах вершин и рёбер. Граф представлен как `aGraph` = (V, E), где V — множество вершин, а E — множество рёбер. Обратите внимание, что `aGraph` имеет вершины |V| и рёбра |E. Вершина v ∈ V представляет собой реальный объект, такой как человек, компьютер или активность. Ребра, v ∈ E, соединяет две вершины в сети:e(v1, v2) | e ∈ E & vi ∈ VПа, приведённое уравнением, указывает, что в графе все рёбра принадлежат множеству E, а все вершины — к множеству, V. Ребро соединяет две вершины и, следовательно, представляет собой связь между ними. Например, она может представлять следующие отношения:

- Дружба между людьми
- Человек, связанный с другом в LinkedIn
- Физическое соединение двух узлов в кластере

Человек, посещающий исследовательскую конференцию

В этой главе мы будем использовать пакет networkx Python для представления графов. Давайте попробуем создать простой граф с помощью пакета `networtx` на Python. Для начала попробуем создать пустой граф, `aGraph`, без вершины или узла:

```
импортировать networkx
как nx G = nx. Graph()
```

Добавим одну вершину:

```
G.add_node("Майк")
```

Также можно добавить множество вершин с помощью списка:

```
G.add_nodes_from([«Амин», «Вассим», «Ник»])
```

Также можно добавить одно ребро между существующими вершинами, как показано:

```
G.add_edge("Майк", "Амин")
```

Теперь напечатаем рёбра и вершины:

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

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

Обратите внимание, что если мы добавляем ребро, это также приводит к добавлению соответствующих вершин, если они ещё не существуют, как показано здесь:

```
G.add_edge(«Амин», «Имран»)
```

Если мы напечатаем список узлов, то следующий результат, который мы наблюдаем:

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

Обратите внимание, что запрос на добавление уже существующей вершины молча игнорируется. Запрос игнорируется или рассматривается в зависимости от типа графа, который мы создали.

Эгоцентричные сети

Прямое окрестие конкретной вершины, m, может содержать достаточно важной информации для проведения окончательного анализа узла. Эгоцентр, или эгонет, основан на этой идее. Эгонет определённой вершины, m, состоит из всех вершин, непосредственно связанных с m, плюс самого узла m. Узел m называется эго, а однохопные соседи, к которым он соединён, — альтерами. Сеть эго конкретного узла 3 показана на следующем графике:

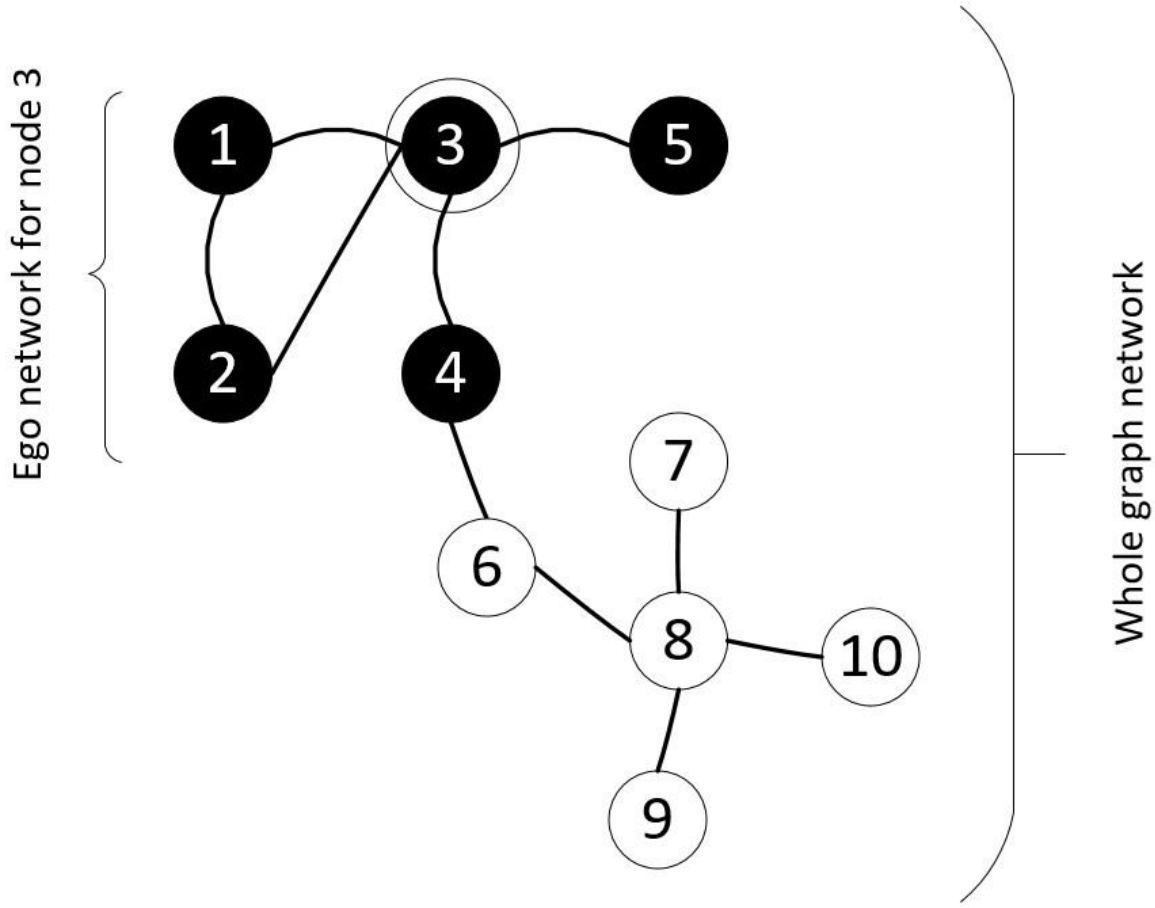


Рисунок 5.1:

Обратите внимание, что эгонет представляет окрестность одной степени. Это понятие можно расширить на n-градусные окрестности, которые состоят из всех вершин, находящихся на n-прыжке от интересующей вершины.

Анализ социальных сетей

Анализ социальных сетей (SNA) является одним из важных применений теории графов. Анализ сетевых графов считается анализом социальных сетей, если применимы следующие условия:

- Вершины графа представляют людей.
- Границы между ними отражают социальные отношения, такие как дружба, общее хобби, родство, сексуальные отношения, антипатии и так далее.
- Бизнес-вопрос, на который мы пытаемся ответить с помощью графического анализа, имеет сильный социальный аспект.

Человеческое поведение отражается в CHC и всегда должно учитываться при работе с CHC. Картографируя человеческие отношения в графике, SNA даёт хорошее понимание человеческих взаимодействий, что помогает нам понять их действия. Создавая вокруг каждого человека сообщество и анализируя его действия на основе его социальных отношений, вы можете получить интересные, а иногда и неожиданные инсайты. Альтернативные подходы к индивидуальному анализу отдельных людей, основанные на их индивидуальных должностных функциях, могут дать лишь ограниченные инсайты. Таким образом, SNA можно использовать для следующих целей:

- Понимание действий пользователей на платформах социальных сетей, таких как Facebook, Twitter или
- LinkedIn
- Понимание мошенничества

Понимание преступного поведения общества

LinkedIn внес значительный вклад в исследования и разработку новых методов, связанных с SNA. Фактически, LinkedIn можно считать пионером многих алгоритмов в этой области. Таким образом, SNA — благодаря своей распределённой и взаимосвязанной архитектуре социальных сетей — является одним из самых мощных сценариев применения теории графов. Другой способ абстрагировать граф — рассматривать его как сеть и применять алгоритм, разработанный для сетей. Вся эта область называется теорией анализа сетей, о которой мы поговорим далее.

Введение теории анализа сетей

Мы знаем, что взаимосвязанные данные могут быть представлены как сеть. В теории сетевого анализа мы изучаем детали методологий, разработанных для изучения и анализа данных, представленных в виде сети. Давайте рассмотрим некоторые важные аспекты теории анализа сетей в этом разделе. Во-первых, обратите внимание, что вершина в сети выступает в роли базовой единицы. Сеть — это взаимосвязанная сеть вершин, где каждое соединение отражает взаимоотношения между различными объектами, исследуемыми. Важно количественно оценить полезность и значимость вершины в сети в контексте задачи, которую мы пытаемся решить. Существует множество методов, которые могут помочь нам количественно оценить её значение. Давайте рассмотрим некоторые важные концепции, используемые в теории сетевого анализа.

Понимание самого короткого пути

Путь — это последовательность узлов между начальным узлом и конечным узлом, при которой ни один узел не появляется дважды на этом пути. Путь представляет собой маршрут между выбранной начальной и конечной вершиной. Это будет множество вершин p, соединяющих начальную вершину с конечной. Ни одна вершина не повторяется в p. Длина пути вычисляется путём подсчёта составляющих рёбер. Из всех вариантов путь с наименьшей длиной называется самым коротким путём. Вычисление самого короткого пути широко используется в алгоритмах теории графов, но не всегда легко вычислить. Существуют разные алгоритмы, которые позволяют найти кратчайший путь между начальным узлом и конечным узлом. Одним из самых популярных алгоритмов, которые можно использовать для поиска самого короткого расстояния между начальным узлом и конечным узлом, является алгоритм Дейкстры, опубликованный в конце 1950-х годов. Он вычисляет самый короткий путь в графе. Он может использоваться устройствами глобальной системы позиционирования (GPS) для расчёта минимального расстояния между источником и пунктом назначения. Алгоритм Дейкстры также используется в алгоритмах сетевой маршрутизации. Между Google и Apple идёт борьба за разработку лучшего алгоритма для работы на самых коротких расстояниях для Google Maps и Apple Maps. Их задача — сделать алгоритм достаточно быстрым, чтобы вычислить кратчайший путь за считанные секунды. Позже в этой главе мы рассмотрим алгоритм поиска по ширине (BFS), который можно модифицировать для преобразования в алгоритм Дейкстры. BFS

предполагает одинаковую стоимость прохождения каждого пути в данном графе. Для алгоритма Дейкстры стоимость прохождения графа может отличаться, и его необходимо включить для модификации BFS в алгоритм Дейкстры. Как указано, алгоритм Дейкстры — это алгоритм с одним источником, который вычисляет кратчайший путь. Если мы хотим решить все пары самых коротких путей, то можно использовать алгоритм Флойда-Уоршалла.

Создание района

Поиск стратегий создания окрестности вокруг интересующих узлов является ключевым для графовых алгоритмов. Методологии создания окрестностей основаны на выборе прямых ассоциированных с интересующей вершиной. Один из способов создания окрестности — выбрать стратегию порядка k, которая выбирает вершины, находящиеся на k прыжков от интересующей вершины. Давайте рассмотрим различные критерии создания районов.

Треугольники

В теории графов важно найти вершины, хорошо связанные друг с другом, для целей анализа. Один из методов — попытаться идентифицировать треугольники, которые представляют собой подграф, состоящий из трёх узлов, непосредственно связанных друг с другом, в сети. Давайте рассмотрим пример выявления мошенничества, который мы также использовали в качестве примера в конце этой главы. Если эгонет узла m состоит из трёх вершин, включая вершину m, то этот эгонет является треугольником. Вершина m будет эго, а две связанные вершины — альтерами, скажем, вершина A и вершина B. Если оба альтера известны как мошеннические, мы можем с уверенностью объявить вершину m мошенническим. Если один из альтеров замешан в мошенничестве, мы не можем найти убедительных доказательств, но нам нужно провести дополнительный поиск доказательств мошенничества.

Плотность

Давайте сначала определим полностью связанную сеть. Мы называем граф, где каждая вершина напрямую связана с каждой другой, полностью связанной сетью. Если у нас есть полностью связанная сеть N, то количество рёбер в сети можно представить следующим образом:

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

Вот тут и вступает в игру плотность. Плотность измеряет количество наблюдаемых рёбер с максимальным количеством рёбер, если EdgesObserved — это количество рёбер, которые мы хотим наблюдать. Его можно сформулировать следующим образом:

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

Обратите внимание, что для треугольника плотность сети равна `1` , и это максимально возможна связная сеть.

Понимание мер центральности

Существуют разные показатели для понимания центральной роли конкретной вершины в графе или подграфе. Например, они могут количественно оценить важность человека в социальной сети или значимость здания в городе. В графовом анализе широко используются следующие меры центральности:

- Степень
- Промежуточность
- Близость
- Собственный вектор

Давайте обсудим их подробно.

Степень

Количество рёбер, связанных с определённой вершиной, называется её степенью. Она может указывать на то, насколько хорошо связана конкретная вершина и её способность быстро распространять сообщение по сети. Рассмотрим `aGraph` = (V, E), где V представляет множество вершин, а E — множество рёбер. Напомним, что `y aGraph` есть вершины |V| и рёбра |E. Если делить степень узла на (|V| -1), это называется центральной степенью:

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

Теперь давайте рассмотрим конкретный пример. Рассмотрим следующий график:

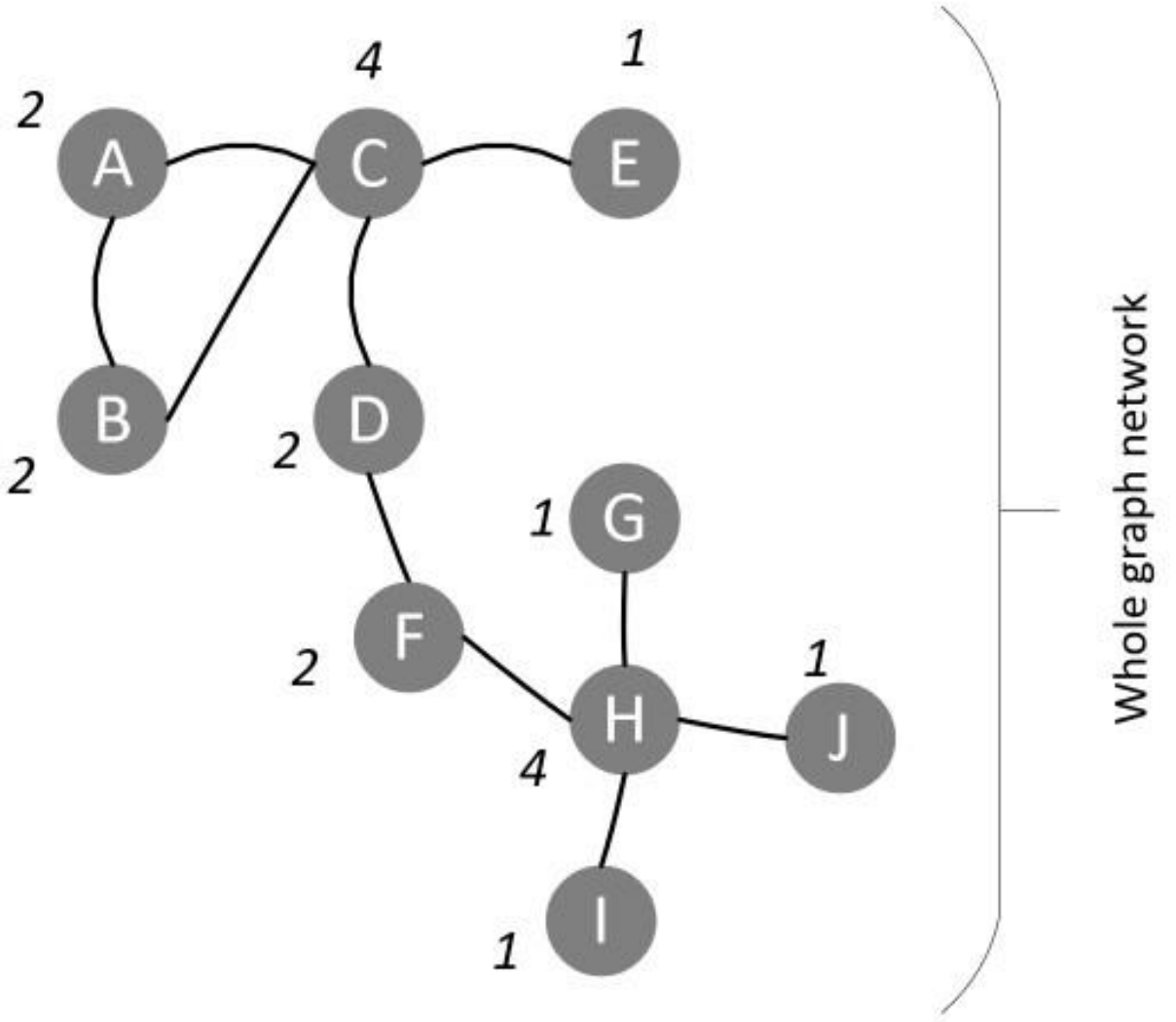


Рисунок 5.x:

Теперь, в предыдущем графе, вершина C имеет степень 4. Центральность степени может быть вычислена следующим образом:

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

Промежуточность

Промежуточность — это мера центральности в графе. В контексте социальных сетей она количественно оценивает вероятность того, что человек является частью коммуникации в подгруппе. Для компьютерной сети промежуточность количественно оценивает негативное влияние на связь между графовыми узлами в

случае отказа вершины. Чтобы вычислить промежуточность вершины a в определённом `aGraph` = (V, E), следуйте следующим шагам:

A. Вычислите самые короткие пути между каждой парой вершин в aGraph. Давайте представим это

$$n_{shortest_{Total}}$$

.

Б. От

$$n_{shortest_{Total}}$$

, Посчитайте количество самых коротких путей, проходящих через вершину a. Давайте представим это

$$n_{shortest_a}$$

.

B. Вычислим промежуточность c

$$C_{betweenness_a} = \frac{n_{shortest_a}}{n_{shortest_{Total}}}$$

.

Справедливость и близость

Давайте возьмём график, g. Справедливость вершины a в графе g определяется как сумма расстояния вершины a от других вершин. Обратите внимание, что центральность конкретной вершины количественно определяет её общее расстояние от всех остальных вершин. Противоположность справедливости — это близость.

Центральность собственного вектора

Центральность собственного вектора даёт оценки всем вершинам графа, измеряющих их важность в сети. Оценка будет индикатором связности конкретного узла с другими важными узлами всей сети. Когда

Google создала алгоритм PageRank, который присваивает балл каждой веб-странице в интернете (чтобы выразить её значимость), идея была выведена из меры центральности собственного вектора.

Расчёт метрик центральности с помощью Python

Давайте создадим сеть и попробуем рассчитать её метрики центральности. Следующий блок кода иллюстрирует это:

```
импортировать networkx как nx,
импортировать matplotlib.pyplot
как plt vertices = range(1,10)
рёбра = [(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(вершины)          G.add_edges_from(рёбра)
nx.draw(G,  with_labels=True,node_color='y',node_size=800)
```

Граф, полученный этим кодом, выглядит следующим образом:

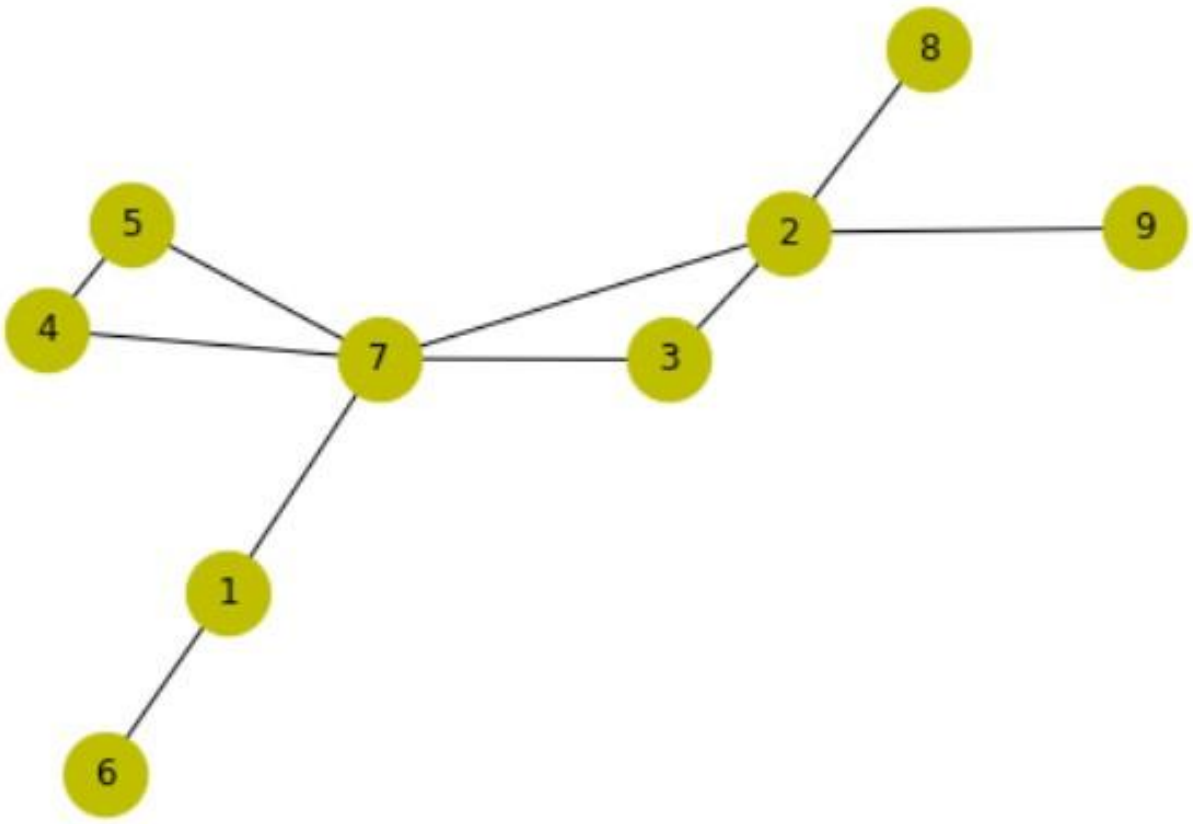


Рисунок 5.x:

До сих пор мы изучали разные показатели центральности. Давайте рассчитаем их для приведённого примера:

```
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')]
```

Обратите внимание, что метрики центральности должны давать меру центральной для конкретной вершины в графе или подграфе. Смотря на график, вершина с обозначением 7 кажется наиболее центральной точкой. Вершина 7 имеет самые высокие значения во всех четырёх метриках центральности, что отражает её важность в данном контексте. Теперь давайте посмотрим, как можно получить информацию из графиков. Графы — это сложные структуры данных с большим количеством информации, хранящейся как в вершинах, так и в рёбрах. Давайте рассмотрим некоторые стратегии, которые можно использовать для эффективной навигации по графам и получения информации с них для ответа на запросы.

Понимание обходов графов

Для использования графиков необходимо извлекать из них информацию. Обход графов определяется как стратегия, используемая для того, чтобы каждая вершина и ребро посещались в упорядоченном порядке. Старается убедиться, что каждая вершина и ребро посещаются ровно один раз; Ни больше, ни меньше. В целом, существует два разных способа перемещения по графу для поиска данных в нём. Поиск по

ширине называется поиском по ширине (BFS), а по глубине — поиском по глубине (DFS). Давайте посмотрим на них по одному.

Поиск по ширине

BFS работает лучше всего, когда в aGraph есть концепция слоёв или уровней окрестностей . Например, когда связи человека в LinkedIn выражены в графе, существуют связи первого уровня, а затем связей второго уровня, которые напрямую переводятся на эти слои. Алгоритм BFS начинается с корневой вершины и исследует вершины в окрестностях. Затем он переходит на следующий уровень соседства и повторяет процесс. Давайте рассмотрим алгоритм BFS. Для этого сначала рассмотрим следующий неориентированный граф:

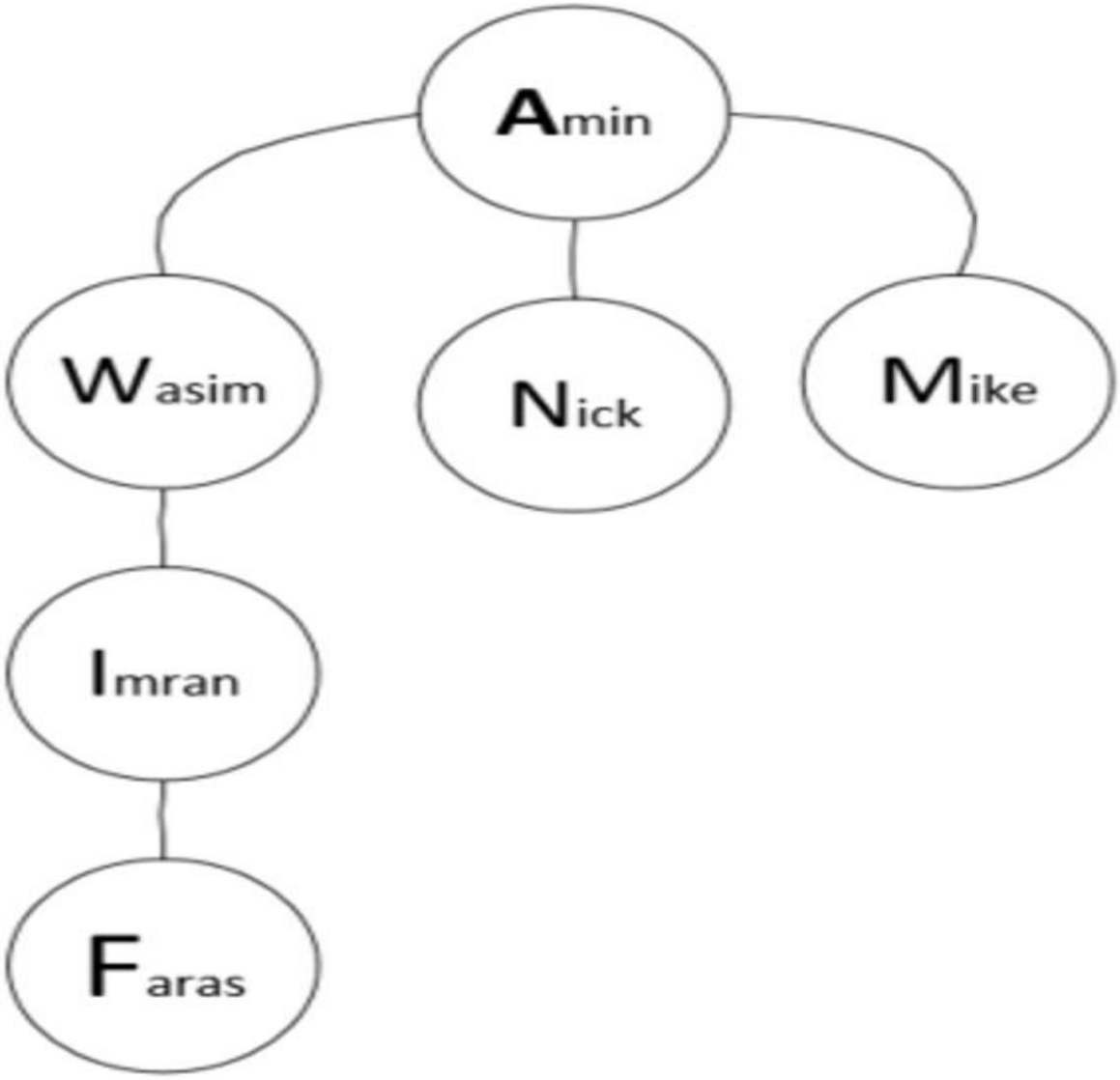


Рисунок 5.x:

Начнём с вычисления непосредственной окрестности каждой вершины и храним это в списке, называемом списком смежности. В Python мы можем использовать структуру данных словаря для его хранения:

```
graph={ 'Amin' : {'Wasim', 'Nick', 'Mike'},
         'Васим': {'Имран', 'Амин'},
         'Имран' : {'Васим','Фарас'},
         'Faras' : {'Imran'},
         'Майк': {'Амин'},
         'Nick' : {'Amin'}}
```

Чтобы реализовать это на Python, мы действуем следующим образом. Сначала мы объясним

инициализацию, а затем основной цикл. Инициализация

Мы будем использовать две структуры данных:

- посетивших: Здесь содержатся все вершины, которые были посещены.
- Изначально он будет пустым. `queue` : Он содержит все вершины, которые мы хотим

посетить в следующих итерациях. Главная петля

Далее мы реализуем основной цикл. Он будет циклироваться, пока в очереди не останется ни одного элемента. Для каждого узла в очереди, если он уже был посещен, он посещает соседа. Мы можем реализовать этот основной цикл в Python следующим образом:

1. Сначала выводим первый узел из очереди и выбираем его текущим узлом этой итерации.

```
узел = queue.pop(0)
```

1. Затем проверяем, что узел отсутствует в списке посещаемых. Если это не так, мы добавляем его в список посещаемых узлов и используем соседей для представления непосредственно связанных узлов

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

1. Теперь добавим соседей узлов в очередь:

```
для соседей в соседях: queue.append(neighbor)
```

А. После завершения основного цикла возвращается посещаемая структура данных, содержащая все пройденные узлы.

Б. Полный код, с инициализацией и основным циклом, будет выглядеть следующим образом:

```
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
```

Давайте рассмотрим исчерпывающий паттерн обхода поиска для графа, который мы определили с помощью BFS. Чтобы посетить все узлы, схема обхода показана на следующем рисунке. Можно заметить, что при выполнении он всегда поддерживает две структуры данных:

- **Visited**: содержит все посещённые узлы **Queue**:
- содержит узлы, которые ещё не были посещены

Вот как работает алгоритм:

А. Он начинается с первого узла, который является единственным узлом, Амином, на первом уровне.
Б. Затем он переходит на второй уровень и посещает все три узла — Васима, Ника и Майка по одному.
В. После этого он переходит на третий и четвёртый уровень, где есть только один узел — Имран и Фарас.

После посещения всех узлов они добавляются в структуру данных Visited, и итерации прекращаются:

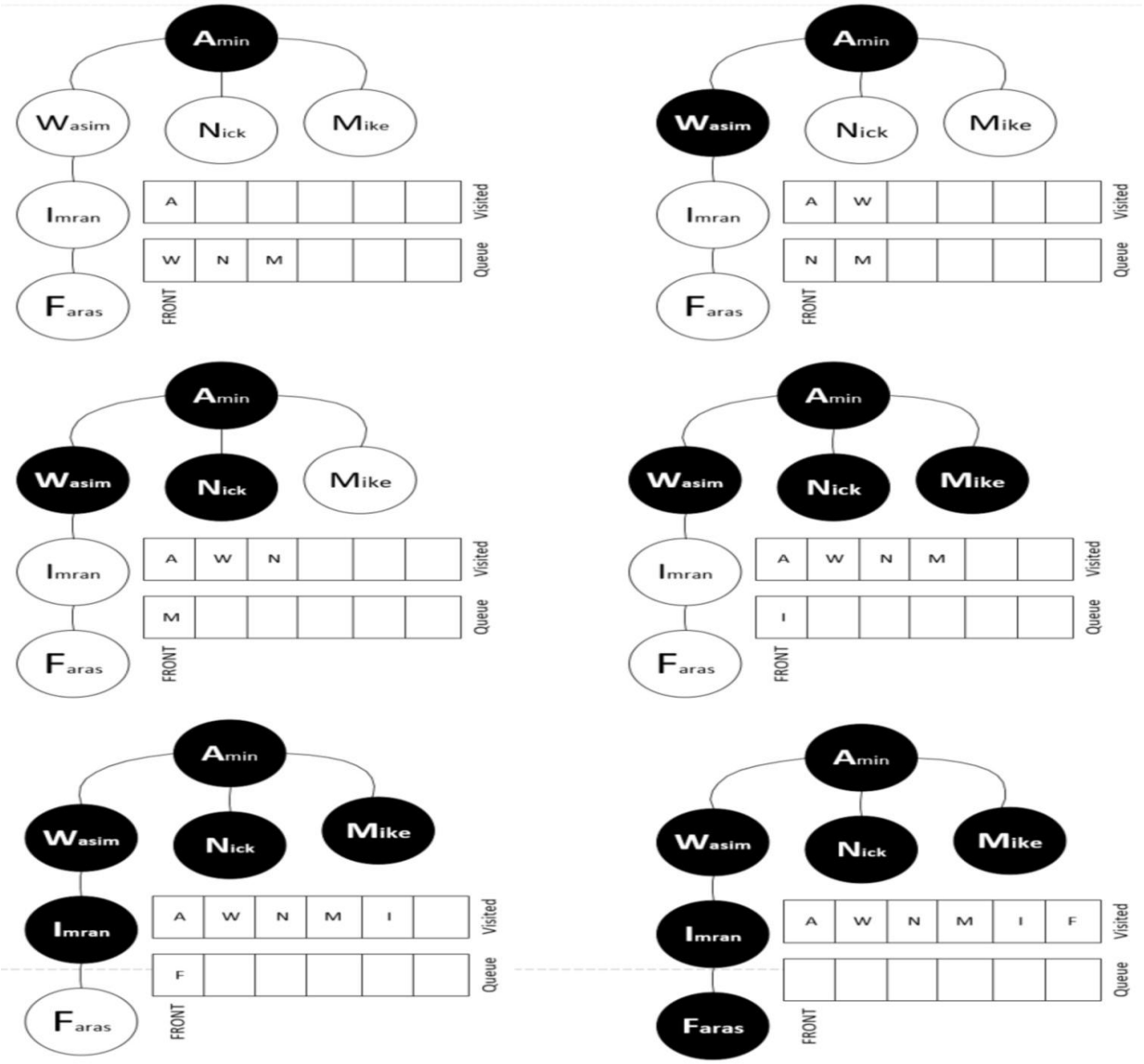


Рисунок 5.x:

Теперь давайте попробуем найти конкретного человека по этому графику с помощью BFS. Давайте уточним данные, которые мы ищем, и посмотрим на результаты:

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

Теперь давайте рассмотрим алгоритм поиска по глубине.

Поиск по глубине

DFS — это альтернатива BFS, используемая для поиска данных по графу. Фактор, отличающий DFS от BFS, заключается в том, что после начала с корневой вершины алгоритм опускается по максимуму по каждому

из уникальных отдельных путей по одному. Для каждого пути, достигнув конечной глубины, он отмечает все вершины, связанные с этим путём, как посещённые. После завершения пути алгоритм возвращается назад. Если алгоритм может найти другой путь от корневого узла, который ещё не был погружён, алгоритм повторяет предыдущий процесс. Алгоритм продолжает двигаться в новой ветке, пока все ветки не будут побыты. Обратите внимание, что граф может иметь циклический метод. Как уже упоминалось, мы используем булевый флаг для отслеживания обработанных вершин и избегать циклового повторения. Для реализации DFS мы будем использовать стековую структуру данных, которая подробно обсуждалась во *второй главе* «*Структуры данных, используемые в алгоритмах*». Помните, что стек основан на принципе **«Последний вошёл — первый вышел**» (**LIFO).** Это отличается от очереди, которая использовалась для BFS, и работает по принципу **First In, First Out** (**FIFO**). Для DFS используется следующий код:

```
def dfs(graph, start, visited=None):
если visited — None: visited = set()
visited.add(start) print(start) для
next in graph[start] - visited:
        dfs(график, следующий,
посещённый) Возвращение посетено
```

Давайте снова используем следующий код для проверки `ранее определённой функции dfs:`

```
graph={ 'Amin' : {'Wasim', 'Nick', 'Mike'},
         'Васим': {'Имран', 'Амин'},
         'Имран' : {'Васим','Фарас'},
         'Faras' : {'Imran'},
         'Mike' :{'Amin'},
         'Nick' :{'Amin'}}
```

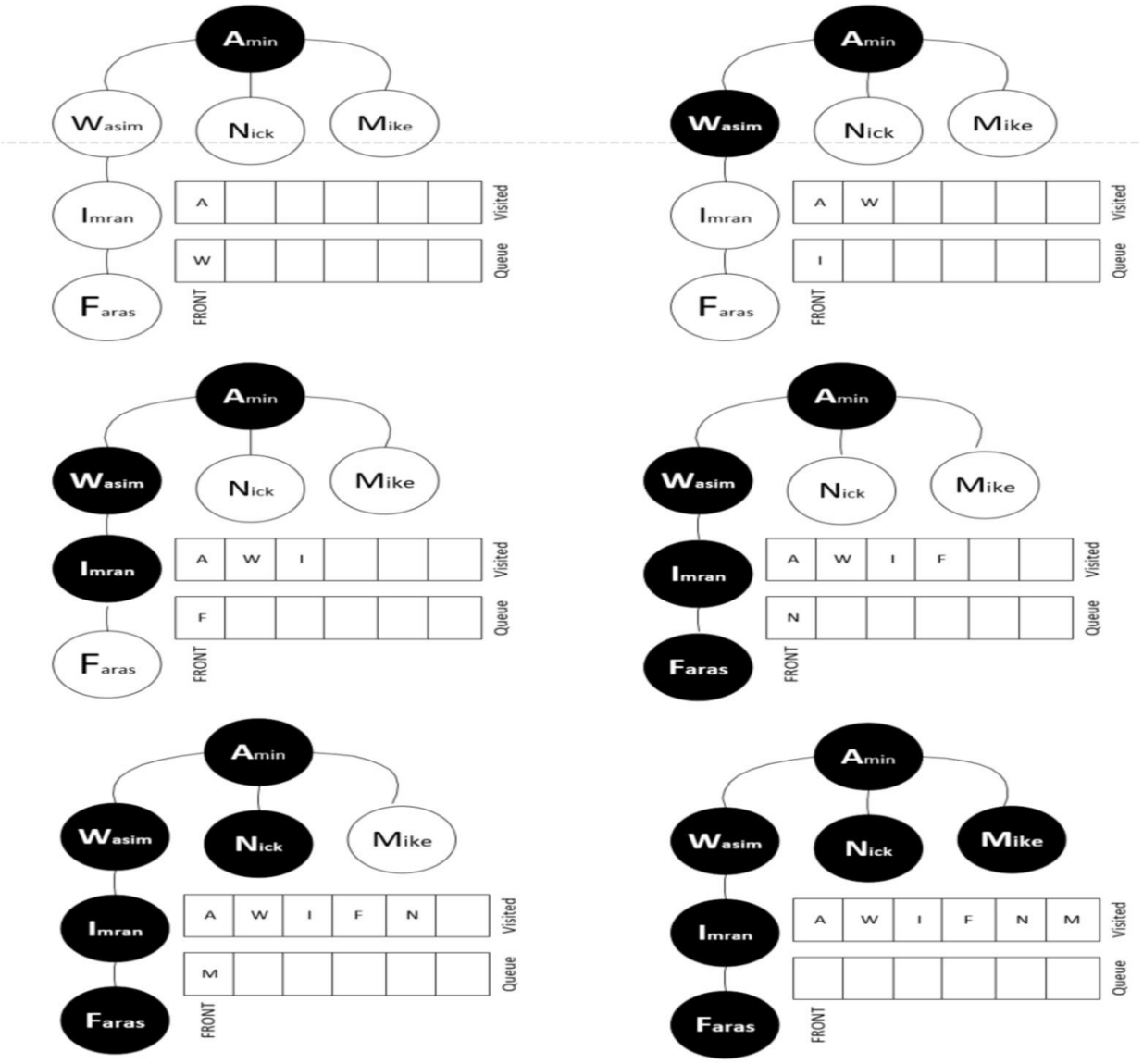
Если запустить этот алгоритм, результат будет выглядеть следующим образом:

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

Давайте рассмотрим исчерпывающую схему обхода этого графа с помощью методологии DFS:

А. Итерация начинается с верхнего узла — Амина.
Б. Затем переходит на второй уровень — Васим. Оттуда он движется к нижним уровням, пока не достигает конца — узлов Имран и Фарес.
В. После завершения первой полной ветки он возвращается назад, а затем направляется на второй уровень, чтобы навестить Ника и Майка.

Схема перемещения показана на следующем рисунке:



Обратите внимание, что DFS можно использовать и в деревьях.Теперь рассмотрим кейс-стади, который объясняет, как обсуждаемые в этой главе концепции могут быть использованы для решения реальных проблем.

Кейс-стади — аналитика мошенничества

Давайте посмотрим, как мы можем использовать SNA для выявления мошенничества. Поскольку люди — социальные существа, на человеческое поведение влияют окружающие тебя люди. Слово «гомофилия» было введено для обозначения влияния их социальной сети на человека. Расширяя эту концепцию, гомофильная сеть — это группа людей, которые, скорее всего, ассоциируются друг с другом по какому-то общему фактору; Например, иметь одинаковое происхождение или хобби, быть частью одной банды или одного университета, или какое-то сочетание других факторов. Если мы хотим проанализировать мошенничество в гомофильной сети, мы можем воспользоваться отношениями между расследуемым лицом и другими участниками сети, чей риск участия в мошенничестве уже тщательно рассчитан. Пометку человека из-за его компании иногда также называют чувством вины по ассоциации. Чтобы понять процесс,

давайте сначала рассмотрим простой случай. Для этого используем сеть с девятью вершинами и восемью рёбрами. В этой сети четыре вершины являются известными случаями мошенничества и классифицируются как мошенничество (F). Пятеро из оставшихся людей не имеют истории мошенничества и классифицируются как немошеннические (NF). Мы напишем код со следующими шагами для генерации этого графа:

1. Давайте импортируем те посылки, которые нам нужны:

```
Импортировать networkX как NX,
импортировать matplotlib.pyplot
как PLT
```

1. Определить структуры данных `вершин` и `рёбер` :

```
вершины = дальность(1,10)
рёбра= [(7,2), (2,3), (7,4), (4,5), (7,3), (7,5), (1,6),(1,7),(2,8),(2,9)]
```

1. Давайте сначала создадим график:

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

1. Теперь нарисуем график:

```
G.add_nodes_from(вершины)
G.add_edges_from(рёбра) pos=nx.spring_layout(G)
```

1. Определим узлы NF:

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

1. Теперь давайте создадим узлы, известные как причастные к мошенничеству:

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

1. Давайте создадим метки для узлов:

```
nx.draw_networkx_edges(G,pos,edges,width=3,alpha=0.5,edge_color='b')  метки={}  метки[1]=r'1  NF'
nx.draw_networkx_labels(G,pos,labels,font_size=16) После
```

выполнения предыдущего кода он покажет график следующего:

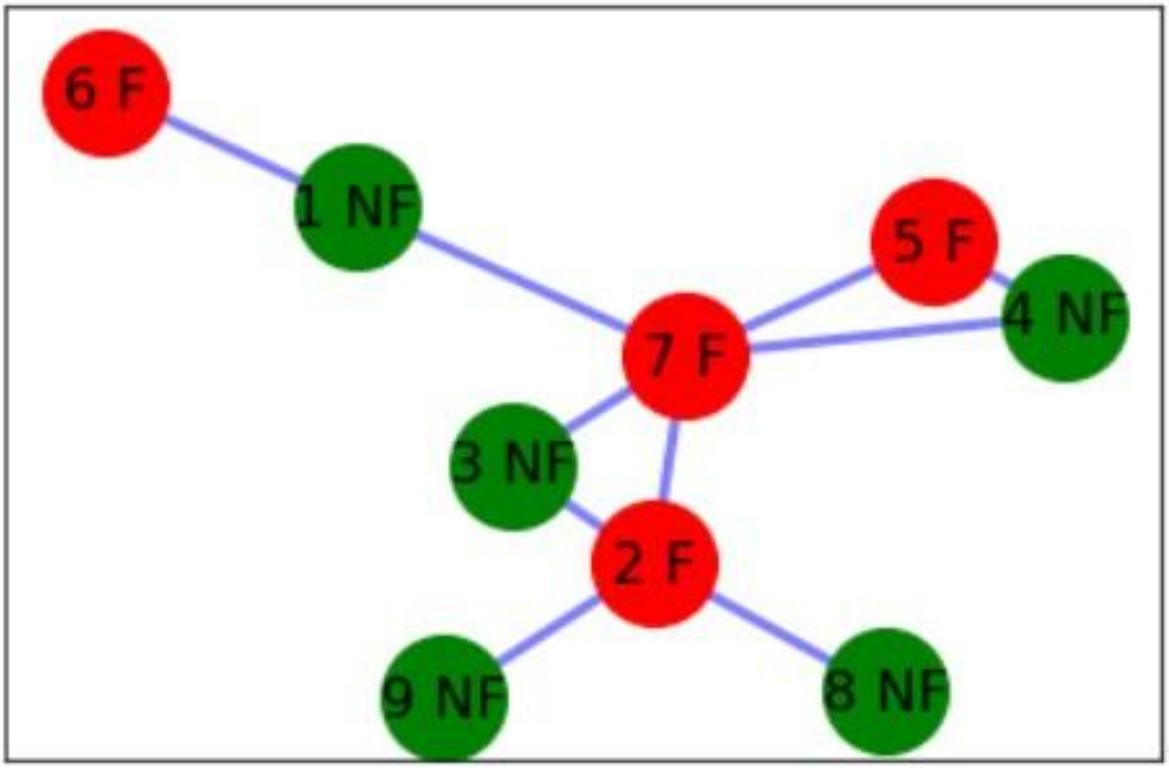


Рисунок 5.x:

Обратите внимание, что мы уже провели подробный анализ, чтобы классифицировать каждый узел как граф или неграф. Предположим, что мы добавляем в сеть ещё одну вершину с названием q, как показано на следующем рисунке. У нас нет предварительной информации об этом человеке и о том, причастен ли он к мошенничеству или нет. Мы хотим классифицировать этого человека как NF или F на основе его связей с существующими участниками социальной сети:

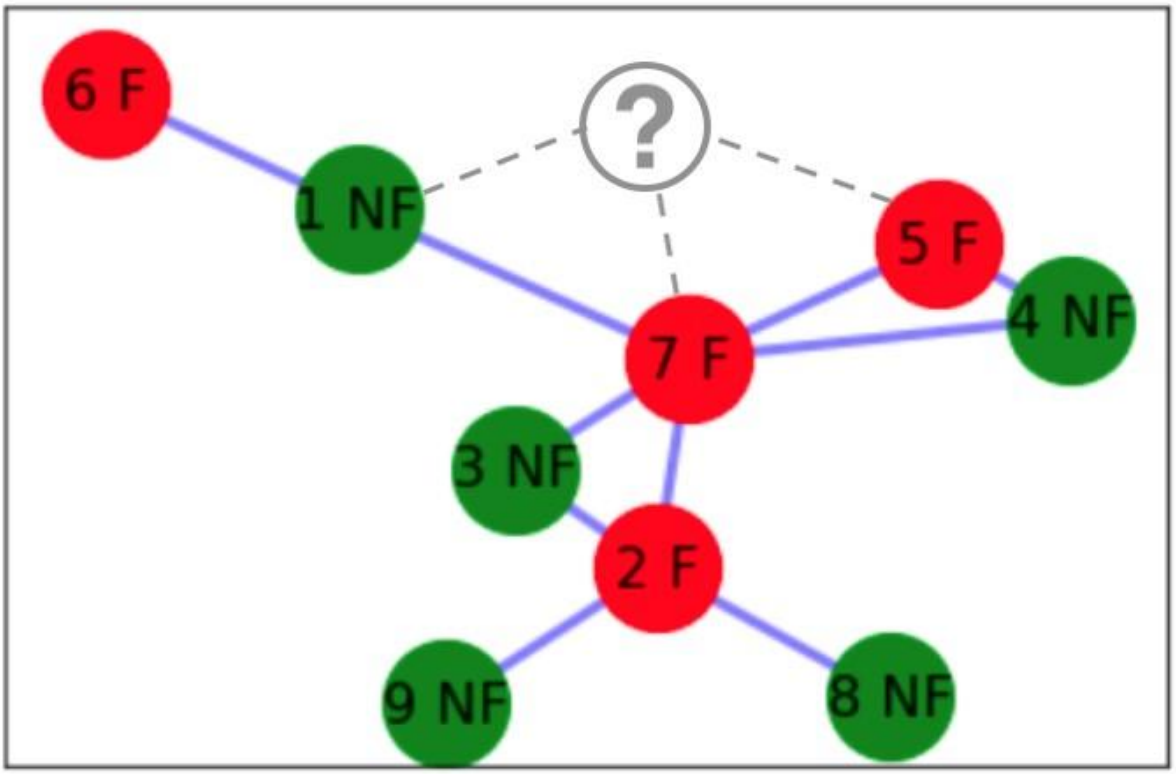


Рисунок 5.x:

Мы разработали два способа классифицировать этого нового человека, представленного узлом q, как F или NF:

- Использование простого метода, не использующего метрики центральности и дополнительной
- информации о типе мошенничества. Использование методологии сторожевой башни — продвинутой техники, использующей метрики центральности существующих узлов, а также дополнительную информацию о типе мошенничества

Мы подробно обсудим каждый метод.

Проведение простой аналитики мошенничества

Простая техника аналитики мошенничества основана на предположении, что в сети поведение человека зависит от людей, с которыми он связан. В сети две вершины с большей вероятностью проявляют похожее поведение, если они связаны друг с другом. Исходя из этого предположения, мы разрабатываем простую технику. Если мы хотим найти вероятность того, что определённый узел a, принадлежит F, вероятность представлена как P(F/q) и вычисляется следующим образом:

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

Применим это к предыдущему рисунку, где Neighborhoodn представляет окрестность вершины n, а w(n, nj) — вес связи между n и nj. Кроме того, степень q — это степень узла q. Тогда вероятность вычисляется следующим образом:

$$P(F|q) = \frac{1+1}{3} = \frac{2}{3} = .67$$

Согласно этому анализу, вероятность участия этого человека в мошенничестве составляет 67%. Нужно установить порог. Если порог 30%, то этот человек выше порогового значения, и мы можем с уверенностью отметить его как F. Обратите внимание, что этот процесс нужно повторять для каждого из новых узлов в сети. Теперь давайте рассмотрим продвинутый способ проведения аналитики мошенничества.

Презентация методологии аналитики мошенничества в Сторожевой башне

Предыдущая, простая техника анализа мошенничества имеет следующие два ограничения:

- Она не оценивает важность каждой вершины в социальной сети. Связь с хабом, вовлечённым в мошенничество, может иметь иные последствия, чем отношения с удалённым, изолированным
- человеком. Когда мы обозначаем кого-то как известный случай мошенничества в существующей сети, мы не учитываем тяжесть преступления.

Методология аналитики мошенничества в сторожевой башне решает эти два ограничения. Сначала давайте рассмотрим несколько концепций.

Оценка отрицательных результатов

Если известно, что человек участвует в мошенничестве, мы говорим, что у него есть негативные последствия. Не все негативные последствия одинаково серьёзны или тяжёлые. Человек, известный тем, что выдаёт себя за другого человека, столкнётся с более серьёзными негативными последствиями, по сравнению с тем, кто просто пытается использовать просроченную подарочную карту на $20 инновационным способом, чтобы сделать её действительной. От 1 до 10 мы оцениваем различные негативные результаты следующим образом:

Отрицательный исход	**Отрицательный результат**
Подражание	10
Участие в краже кредитных карт	8
Поддельный чек	7
Судимость	6
Нет записей	0

Обратите внимание, что эти оценки будут основаны на нашем анализе случаев мошенничества и их влиянии на основе исторических данных.

Степень подозрения

Степень подозрения (DOS) количественно определяет наш уровень подозрений, что человек может быть причастен к мошенничеству. Значение DOS 0 означает, что это человек с низким риском, а значение DOS 9 — что это человек с высоким риском. Анализ исторических данных показывает, что профессиональные мошенники занимают важные позиции в своих социальных сетях. Чтобы включить это, сначала мы вычисляем все четыре метрики центральности каждой вершины в нашей сети. Затем берём среднее значение этих вершин. Это отражает важность конкретного человека в сети. Если человек, связанный с вершиной, замешан в мошенничестве, мы иллюстрируем этот негативный исход, оценивая его по заранее заданным значениям, указанным в предыдущей таблице. Это делается так, чтобы тяжесть преступления отражалась в ценности каждого отдельного DOS. Наконец, мы умножаем среднее среднее метрик центральности и отрицательный результат, чтобы получить значение DOS. Мы нормализуем DOS, деля её на максимальное значение DOS в сети. Рассчитаем DOS для каждого из девяти узлов предыдущей сети:

	Узел 1	**Узел 2**	**Узел 3**	**Узел 4**	**Узел 5**	**Узел 6**	**Узел 7**	**Узел 8**	**Узел 9**
Степень	0.25	0.5	0.25	0.25	0.25	0.13	0.63	0.13	0.13
Центральная роль Промежуточность	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
Собственный вектор	0,24	0.45	0.36	0.32	0.32	0.08	0.59	0.16	0.16
Средние значения	0.31	0.51	0.29	0.26	0.26	0.14	0.66	0.17	0.17
Центральная роль Метрики									
Отрицательная Результат	0	6	0	0	7	8	10	0	0
DOS	0	3	0	0	1.82	1.1	6.625	0	0
Нормализовано DOS	0	0.47	0	0	0.27	0.17	1	0	0

Каждый из узлов и их нормализованная DOS показаны на следующем рисунке:

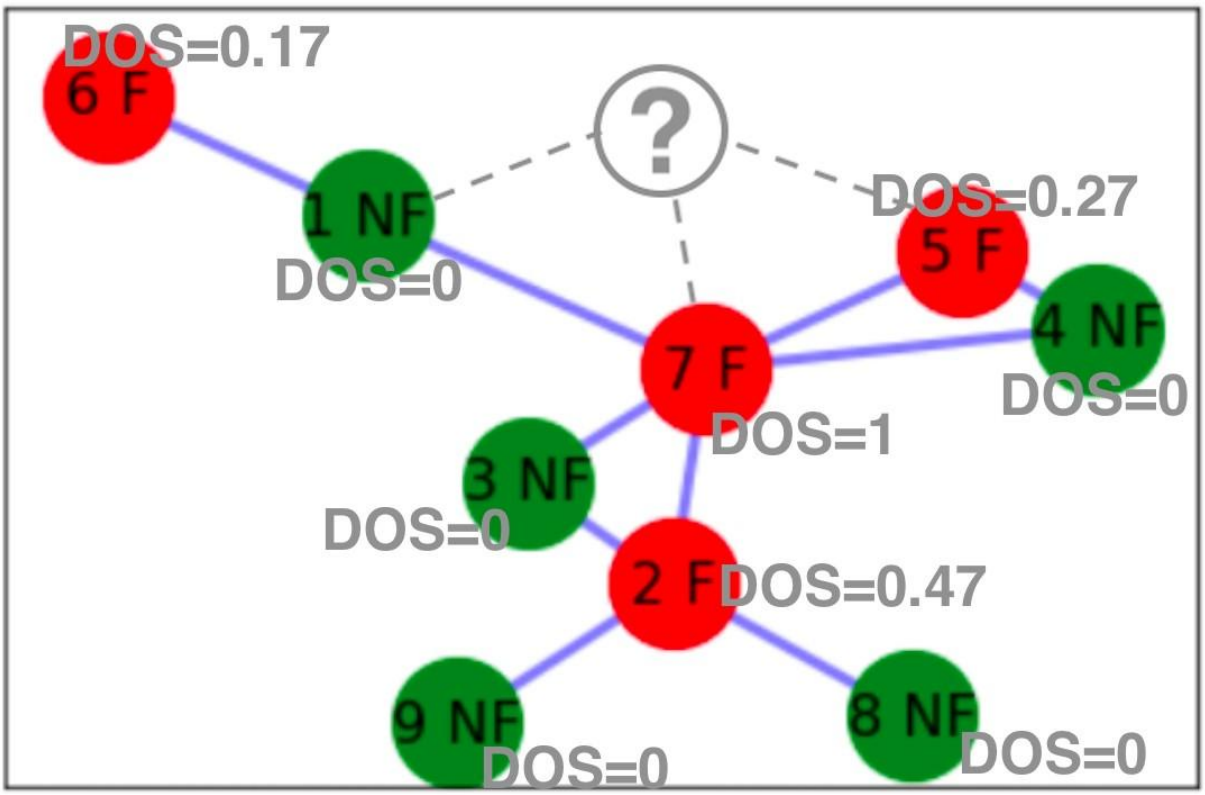


Рисунок 5.x:

Для вычисления DOS нового добавленного узла мы используем следующую формулу:

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

Используя соответствующие значения, мы вычислим DOS следующим образом:

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

Это будет указывать на риск мошенничества, связанного с добавлением нового узла в систему. Это означает, что по шкале от 0 до 1 у этого человека DOS-значение 0,42. Для DOS можно создавать разные группы рисков, следующим образом:

Значение классификации риска DOS

DOS = 0 Без риска

0<DOS<=0.10 Низкий риск

0.10<DOS<=0.3 Средний риск

DOS>0.3 Высокий риск

Исходя из этих критериев, можно увидеть, что новый человек относится к группе высокого риска и должен быть отмечен. Обычно при проведении такого анализа временное измерение не задействуется. Но сейчас существуют продвинутые методы, которые изучают рост графика со временем. Это позволяет исследователям изучать взаимосвязь между вершинами по мере развития сети. Хотя такой анализ временных рядов на графах значительно усложнит задачу, он может дать дополнительное понимание доказательств мошенничества, которое иначе было бы невозможно.

Краткое содержание

В этой главе мы узнали о алгоритмах на основе графов. В этой главе использовались различные методы представления, поиска и обработки данных, представленных в виде графов. Мы также развили навыки для вычисления самого короткого расстояния между двумя вершинами и построили окрестности в нашем проблемном пространстве. Эти знания должны помочь нам использовать теорию графов для решения таких задач, как обнаружение мошенничества. В следующей главе мы сосредоточимся на различных алгоритмах машинного обучения без надзора. Многие техники использования, обсуждаемые в этой главе, дополняют алгоритмы обучения без надзора, которые будут подробно рассмотрены в следующей главе. Обнаружение доказательств мошенничества в наборе данных — пример таких случаев использования.

OceanfPDF.com

Раздел 2: Алгоритмы машинного обучения

В этом разделе подробно объясняются различные типы алгоритмов машинного обучения, такие как алгоритмы неконтролируемого машинного обучения и традиционные алгоритмы с контролируемым обучением, а также представлены алгоритмы обработки естественного языка. Раздел заканчивается знакомством с рекомендательными системами. Главы, включённые в этот раздел, следующие:

- *Глава 6*, *Алгоритмы машинного обучения без надзора*
- *Глава 7*, *Традиционные алгоритмы контролируемого обучения*
- *Глава 8*, *Алгоритмы нейронных сетей*
- *Глава 9, Алгоритмы обработки естественного языка*

Глава 10, *Движки рекомендаций*

6 Неконтролируемые алгоритмы машинного обучения

Присоединяйтесь к Discord-пространству нашей книги

https:// packt.link/40Algos



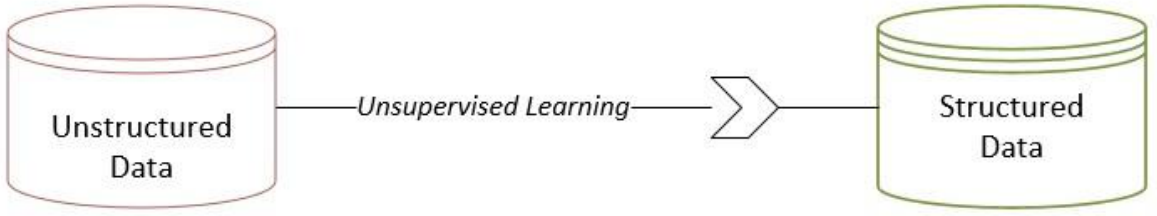
Эта глава посвящена алгоритмам машинного обучения без надзора. Глава начинается с введения в техники обучения без надзора. Затем мы узнаем о двух алгоритмах кластеризации: иерархическом кластерировании k-средних и алгоритмах DBSCAN. В следующем разделе рассматривается алгоритм уменьшения размерности, который может быть эффективен при большом числе входных переменных. Следующий раздел показывает, как неконтролируемое обучение может использоваться для обнаружения аномалий. Наконец, мы рассмотрим одну из самых мощных методов обучения без надзора — майнинг по правилам ассоциаций. В этом разделе также объясняется, как закономерности, выявленные при анализе правил ассоциаций, представляют интересные взаимосвязи между различными элементами данных в разных транзакциях, что может помочь нам в принятии решений на основе данных. К концу этой главы читатель должен понять, как обучение без надзора может быть использовано для решения реальных задач. Читатель поймёт базовые алгоритмы и методологии, которые в настоящее время используются для обучения без надзора. В этой главе мы рассмотрим следующие темы:

- Обучение без надзора
- Алгоритмы кластеризации
- Уменьшение размерности
- Алгоритмы обнаружения аномалий

Ассоциация управляет добычей полезных ископаемых

Введение обучения без надзора

Самое простое определение неконтролируемого обучения заключается в том, что это процесс создания некоторой структуры неструктурированных данных путём обнаружения и использования присущих им закономерностям. Если данные не производятся случайным процессом, между элементами данных в многомерном пространстве задач будут иметь определённые закономерности. Алгоритмы неконтролируемого обучения работают, обнаруживая эти шаблоны и используя их для создания структуры набора данных. Это понятие показано на следующей диаграмме:



Обратите внимание, что неконтролируемое обучение добавляет структуру, открывая новые особенности из существующих шаблонов.

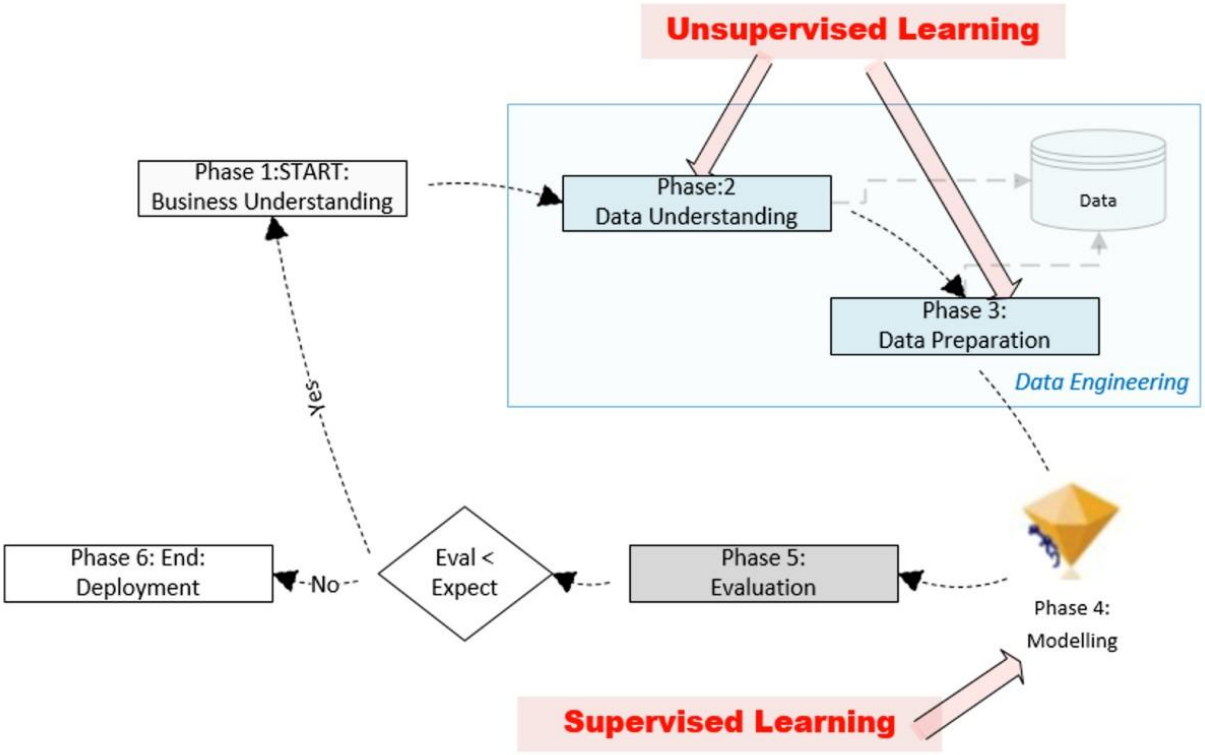
Неконтролируемое обучение в жизненном цикле майнинга данных

Чтобы понять роль обучения без надзора, важно сначала рассмотреть общий жизненный цикл процесса майнинга данных. Существуют различные методологии, которые делят жизненный цикл процесса анализа данных на независимые стадии, называемые **фазами**. В настоящее время существуют два популярных способа отображения жизненного цикла майнинга данных:

- **Жизненный цикл** CRISP-DM **(**Межотраслевой стандартный процесс для
- анализа данных)

Процесс анализа данных SEMMA **(**Sample, Explore, Modify, Model, Access)

CRISP-DM был разработан консорциумом майнеров данных, принадлежащих различным компаниям, включая
Chrysler и **SPSS** (**Статистический пакет для социальных наук**). SEMMA была предложена **системой статистического анализа (SAS).** Давайте рассмотрим одно из этих двух представлений жизненного цикла майнинга данных — CRISP-DM — и попробуем понять роль неконтролируемого обучения в жизненном цикле майнинга данных. Обратите внимание, что у SEMMA есть похожие этапы жизненного цикла. Если посмотреть на жизненный цикл CRISP-DM, можно увидеть, что он состоит из шести различных фаз, показанных на следующем рисунке:



Давайте разберёмся по каждому этапу:

- **Фаза 1: Понимание бизнеса**: Это сбор требований и попытка глубоко понять проблему с точки зрения бизнеса. Определение объёма задачи и правильная её формулировка в соответствии **с помощью машинного обучения** (**ML**) — важная часть этого этапа — например, для задачи бинарной классификации иногда полезно формулировать требования в терминах гипотезы, которую можно доказать или отвергнуть. Эта фаза также посвящена документированию ожиданий от модели машинного обучения, которая будет обучаться на этапе 4 — например, для задачи классификации необходимо задокументировать минимально приемлемую точность модели, которую можно использовать в производстве.

  Важно отметить, что первая фаза жизненного цикла CRISP-DM посвящена пониманию бизнеса. Она сосредоточена на том, что нужно сделать, а не на том, как это будет сделано.

- **Фаза 2: Понимание данных**: Речь идёт о понимании данных, доступных для анализа данных. На этом этапе мы выясним, доступны ли подходящие наборы данных для задачи, которую мы пытаемся решить. После идентификации наборов данных нужно понять их качество и их структуру. Нам нужно

выяснить, какие закономерности можно извлечь из данных, которые потенциально приведут нас к важным инсайтам. Мы также постараемся найти подходящую характеристику, которую можно использовать в качестве метки (или целевую переменную) в соответствии с требованиями, собранными в Фазе 1. Алгоритмы обучения без надзора могут сыграть важную роль в достижении целей второй фазы. Неконтролируемые алгоритмы могут использоваться для следующих целей:
    - Чтобы обнаружить закономерности в наборе данных
    - Понять структуру набора данных путём анализа обнаруженных
    - закономерностей Для выявления или вывода целевой переменной
- **Фаза 3: Подготовка данных**: Речь идёт о подготовке данных для модели машинного обучения, которую мы будем обучать в Фазе 4. Имеющиеся маркированные данные делятся на две неравные части. Большая часть называется **обучающими данными** и используется для обучения модели на этапе 4. Меньшая часть называется **тестовыми данными** и используется в Фазе 5 для оценки модели. На этом этапе алгоритмы машинного обучения без надзора могут быть используется как инструмент для подготовки данных — например, их можно преобразовать неструктурированные данные в структурированные, предоставляя дополнительные измерения, полезные для обучения модели.
- **Фаза 4: Моделирование**: Это этап, когда мы используем контролируемое обучение для формирования выявленных закономерностей. От нас ожидается успешная подготовка данных в соответствии с требованиями выбранного алгоритма контролируемого обучения. На этом этапе также будет идентифицирована конкретная особенность, которая будет использоваться в качестве метки. В третьей фазе мы разделили данные на тестовые и обучающие наборы. На этом этапе мы формируем математические формулировки, чтобы представить взаимосвязи в интересующих нам паттернах. Это достигается путём обучения модели с использованием обучающих данных, созданных на этапе 3. Как уже упоминалось, полученная математическая формулировка будет зависеть от выбора алгоритма.
- **Фаза 5: Оценка**: Эта фаза посвящена тестированию новой обученной модели с использованием тестовых данных Фазы 3. Если оценка соответствует ожиданиям, установленным в Фазе 1, то нам нужно пройти все предыдущие этапы заново, начиная с Фазы 1. Это показано на предыдущем изображении.
- **Этап 6: Внедрение**: Если оценка соответствует или превосходит ожидания, описанные в Фазе 5, обученная модель внедряется в производство и начинает генерировать решение проблемы, определённой в Фазе 1.

Этапы 2 (Понимание данных) и Фаза 3 (подготовка данных) жизненного цикла CRISP-DM посвящены пониманию данных и их подготовке к обучению модели. Эти этапы включают обработку данных. Некоторые организации нанимают специалистов для этого этапа инженерии данных. Очевидно, что процесс предложения решения проблемы полностью основан на данных. Для формулировки работоспособного решения используется сочетание контролируемого и неконтролируемого машинного обучения. В этой главе внимание уделяется части решения, связанной с обучением без надзора. Инженерия данных включает фазы 2 и 3 и является самой трудоёмкой частью машинного обучения. Это может занять до 70% времени и ресурсов типичного ML-проекта. Алгоритмы неконтролируемого обучения могут играть важную роль в инженерии данных. В следующих разделах приведена дополнительная информация о неконтролируемых алгоритмах.

Текущие тенденции исследований в области неконтролируемого обучения

В течение многих лет исследования алгоритмов машинного обучения были больше сосредоточены на методах обучения под надзором. Поскольку методы обучения под надзором могут использоваться напрямую для вывода, их преимущества в плане времени, стоимости и точности относительно легко

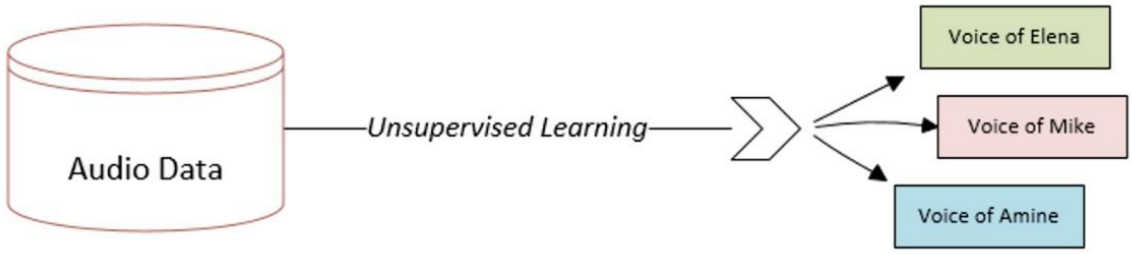
измеряются. В последнее время была признана мощь алгоритмов машинного обучения без надзора. Поскольку неконтролируемое обучение не направляется, оно менее зависит от предположений и потенциально может сходиться к решению в любом измерении. Хотя контролировать объем и требования к обработке алгоритмов обучения без надзора сложнее, они обладают большим потенциалом для выявления скрытых закономерностей. Исследователи также работают над объединением методов неконтролируемого машинного обучения с методами контролируемого обучения для создания новых мощных алгоритмов.

Практические примеры

В настоящее время для лучшего понимания данных и придания им структуры используется неконтролируемое обучение — например, для сегментации маркетинга, выявления мошенничества и анализа корзины рынка (что будет рассмотрено позже в этой главе). Давайте рассмотрим несколько примеров.

Категоризация голоса

Обучение без надзора может использоваться для классификации отдельных голосов в голосовом файле. Она использует тот факт, что голос каждого человека обладает уникальными характеристиками, создавая потенциально отдельные аудиопаттерны. Эти паттерны затем можно использовать для распознавания голоса — например, Google использует эту технику в своих устройствах Google Home, чтобы обучить их различать голоса разных людей. После обучения Google Home может индивидуально адаптировать ответ для каждого пользователя. Например, предположим, что у нас есть записанный разговор трёх человек, разговаривающих друг с другом в течение получаса. Используя алгоритмы неконтролируемого обучения, мы можем идентифицировать голоса разных людей в этом наборе данных. Обратите внимание, что с помощью неконтролируемого обучения мы добавляем структуру к данному набору неструктурированных данных. Эта структура даёт нам дополнительные полезные измерения в нашем проблемном пространстве, которые можно использовать для получения инсайтов и подготовки данных для выбранного алгоритма машинного обучения. Следующая схема показывает, как обучение без надзора используется для распознавания голоса:



Обратите внимание, что в данном случае неконтролируемое обучение подразумевает добавление новой функции с тремя отдельными уровнями.

Классификация документов

Алгоритмы неконтролируемого машинного обучения также могут применяться к хранилищу неструктурированных текстовых данных — например, если у нас есть набор данных PDF-документов, то неконтролируемое обучение можно использовать для выполнения следующих задач:

- Откройте для себя различные темы в наборе данных
- Связывайте каждый PDF-документ с одной из обнаруженных тем

Это использование неконтролируемого обучения для классификации документов показано на следующем рисунке. Это ещё один пример, в котором мы добавляем больше структуры к неструктурированным данным:

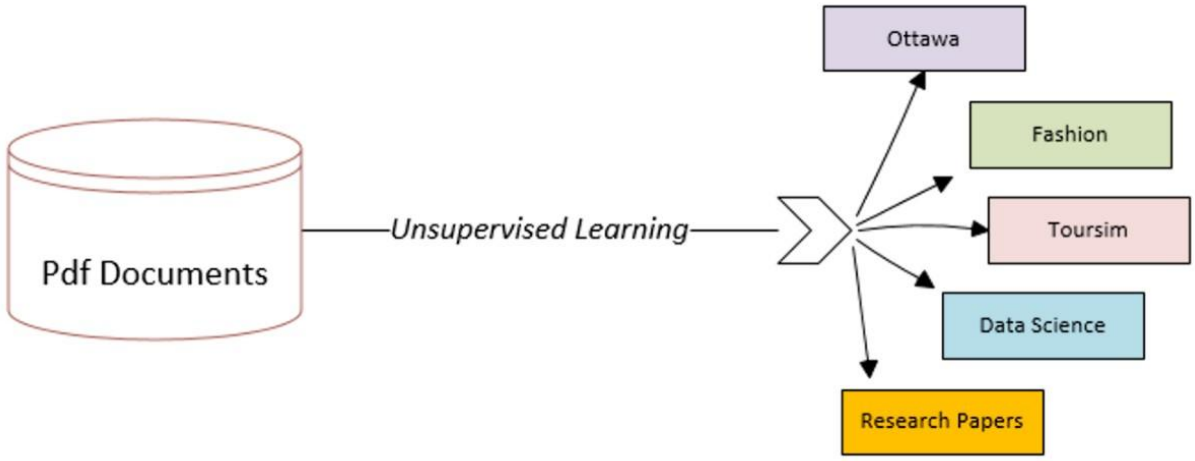


Рисунок 6.4: Использование неконтролируемого обучения для классификации документов

Обратите внимание, что в данном случае обучение без надзора предполагает добавление новой функции с пятью чёткими уровнями.

Понимание алгоритмов кластеризации

Один из самых простых и мощных методов, используемых в неконтролируемом обучении, основан на группировании схожих паттернов с помощью алгоритмов кластеризации. Он используется для понимания определённого аспекта данных, связанного с задачей, которую мы пытаемся решить. Алгоритмы кластеризации ищут естественную группировку в элементах данных. Поскольку группа не основана на какой-либо цели или предположениях, она классифицируется как метод обучения без надзора. Группировки, созданные различными алгоритмами кластеризации, основаны на поиске сходств между различными точками данных в проблемном пространстве. Лучший способ определить сходства между точками данных будет различаться в зависимости от задачи и зависит от характера задачи, с которой мы сталкиваемся. Давайте рассмотрим различные методы, которые можно использовать для расчёта сходств между различными точками данных.

Количественная оценка сходств

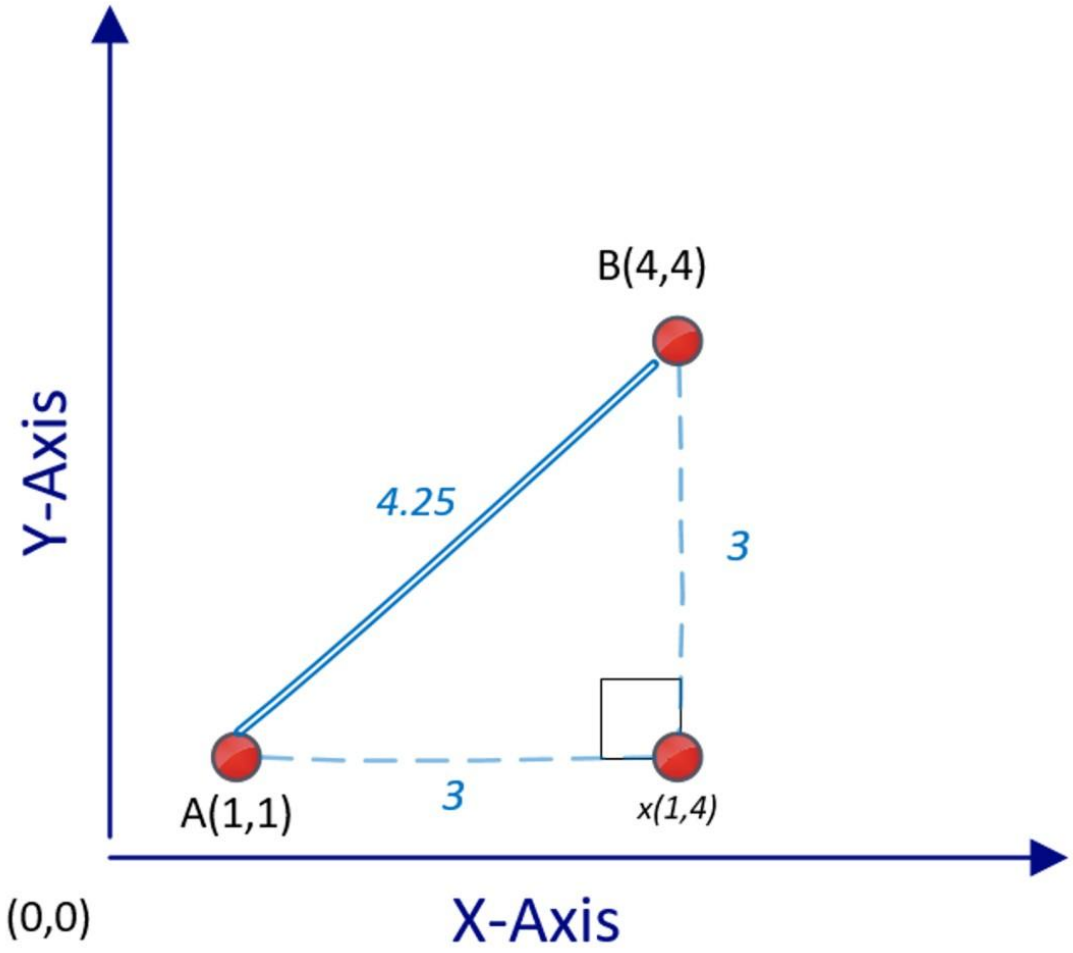
Надёжность группировки, создаваемой алгоритмами кластеризации, основана на предположении, что мы можем точно количественно оценить сходства или близость между различными точками данных в проблемном пространстве. Это делается с помощью различных измерений расстояния. Ниже приведены три самых популярных метода, используемых для количественной оценки сходств:

- Евклидово измерение расстояния
- Манхэттенское измерение
- расстояния по мере косинуса

Давайте подробнее рассмотрим эти показатели расстояния.

Евклидово расстояние

Расстояние между разными точками может количественно оценивать сходство между двумя точками данных и широко используется в неконтролируемых методах машинного обучения, таких как кластеризация. Евклидово расстояние — самая распространённая и простая мера расстояния. Он рассчитывается путём измерения наименьшего расстояния между двумя точками данных в многомерном пространстве. Например, рассмотрим две точки **— A(1,1**) и **B(4,4)** — в двумерном пространстве, как показано на следующем графике:



Для вычисления расстояния между **A** и **B** — то есть *d(A,B)*, можно использовать следующую пифагорейскую формулу:

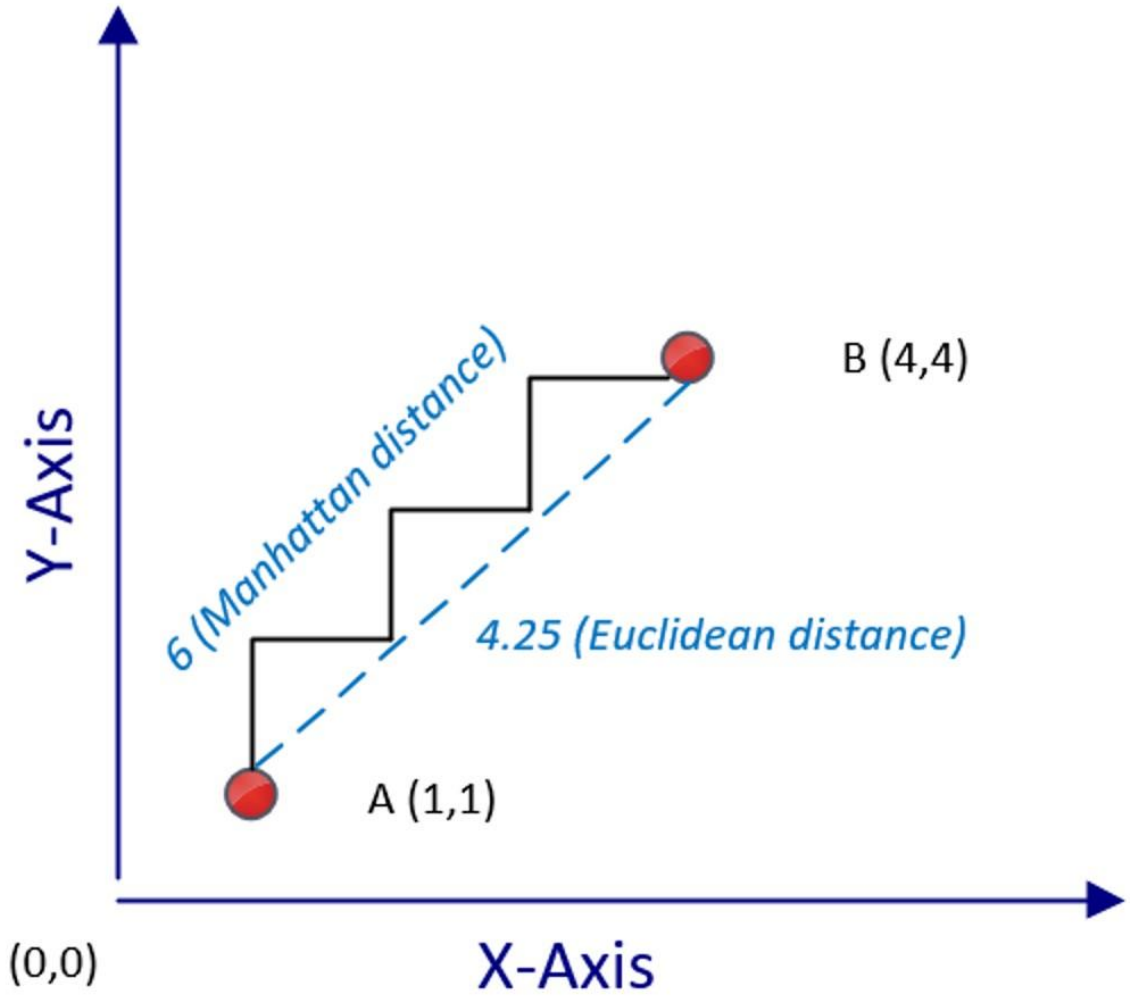
$$d(A,B) = \sqrt{(a_2 - b_2)^2 + (a_1 - b_1)^2} = \sqrt{(4-1)^2 + (4-1)^2} = \sqrt{9+9} = 4.25$$

Обратите внимание, что этот расчёт предназначен для двумерного пространства задач. Для n-мерного пространства задач можно вычислить расстояние между двумя точками **A** и **B** следующим образом:

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

Расстояние до Манхэттена

Во многих случаях измерение самого короткого расстояния между двумя точками с помощью евклидовой меры расстояния не будет действительно отражать сходство или близость между двумя точками — например, если две точки данных представляют местоположение на карте, то фактическое расстояние от точки А до точки Б с использованием наземного транспорта, такого как автомобиль или такси, будет больше расстояния, рассчитанного евклидовым расстоянием. В таких ситуациях мы используем манхэттенское расстояние, которое обозначает самый длинный путь между двумя точками и лучше отражает близость двух точек в контексте истока и пункта назначения, к которым можно добраться в оживлённом городе. Сравнение манхэттенских и евклидовых измерений расстояний показано на следующем графике:

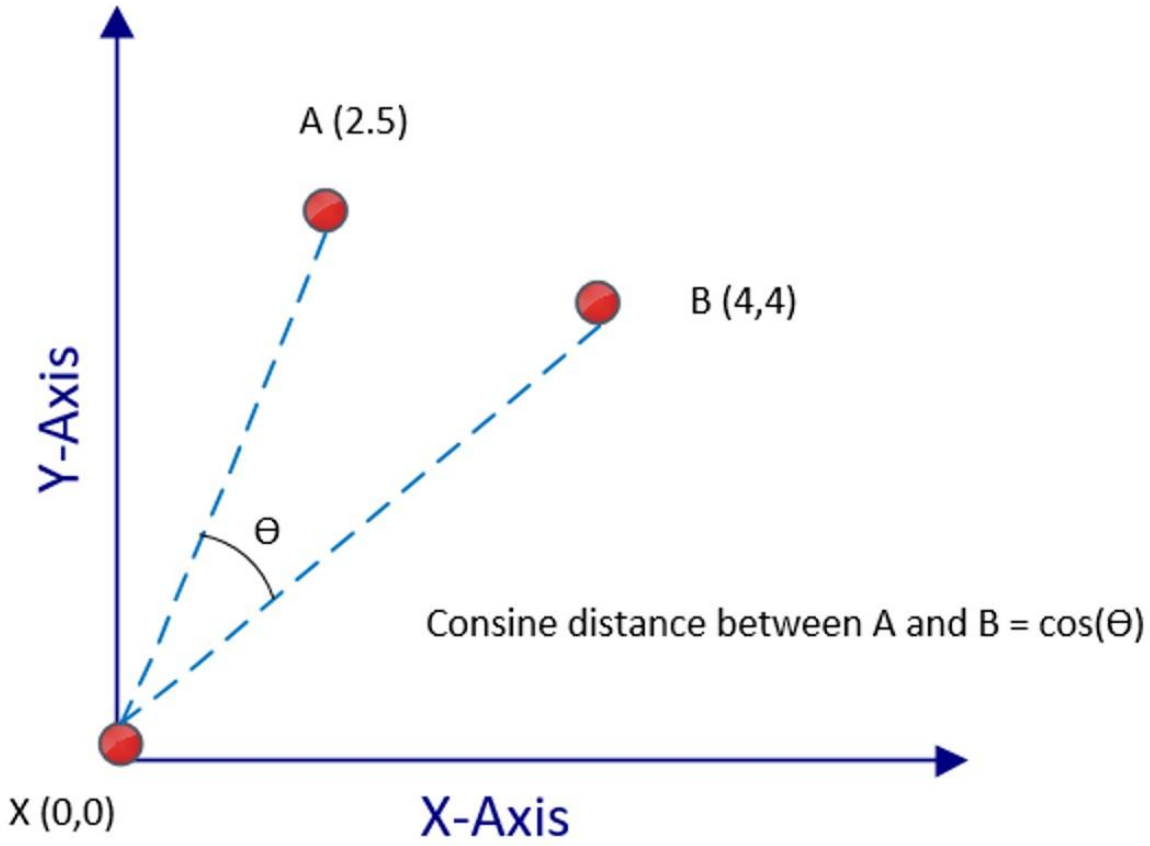


Обратите внимание, что манхэттенское расстояние всегда будет равным или больше, чем соответствующее евклидово расстояние, вычисленное.

Косинусовое расстояние

Евклидовы и манхэттенские меры расстояния плохо работают в пространстве с высокой размерностью. В пространстве задач с высокой размерностью расстояние косинуса точнее отражает близость между двумя точками данных в

Многомерное пространство задач. Мера косинусного расстояния вычисляется путём измерения угла косинуса, созданного двумя точками, соединёнными с опорной точкой. Если точки данных близки, то угол будет узким, независимо от их размеров. С другой стороны, если они находятся далеко, то угол будет большим:



Текстовые данные почти можно считать высокомерным пространством. Поскольку мера косинусного расстояния отлично работает с h-мерными пространствами, она является хорошим выбором при работе с текстовыми данными. Обратите внимание, что на предыдущем рисунке косинус угла между **A(2,5)** и **B(4.4)** равен расстоянию косинуса. Отсчёт между этими точками — это начало координат, то есть **X(0,0)**. Но на самом деле любая точка в проблемном пространстве может служить опорной точкой данных, и она не обязательно должна быть началом координат.

Алгоритм кластеризации K-средних

Название алгоритма кластеризации k-средних происходит от того, что он пытается создать ряд кластеров, k, вычисляя средства для поиска близости между точками данных. Он использует относительно простой подход к кластеризации, но всё ещё популярен благодаря масштабируемости и скорости. Алгоритмически

кластеризация k-средних использует итеративную логику, которая перемещает центры кластеров так, чтобы они отразили наиболее репрезентативную точку данных группировки, к которой они принадлежат. Важно отметить, что алгоритмы k-средних не обладают одной из самых базовых функций, необходимых для кластеризации. Недостающая функциональность заключается в том, что для данного набора данных алгоритм k-средних не может определить наиболее подходящее количество кластеров. Наиболее подходящее количество кластеров, *k*, зависит от количества естественных группировок в конкретном наборе данных. Философия этого упущения заключается в том, чтобы сделать алгоритм максимально простым, максимизируя его производительность. Такой дизайн с общитым и средним делает k-средние подходящими для больших наборов данных. Предполагается, что для вычисления k будет использоваться внешний механизм. Лучший способ *определить k* будет зависеть от задачи, которую мы пытаемся решить. В некоторых случаях *k* напрямую определяется контекстом задачи кластеризации — например, если мы хотим разделить класс студентов, изучающих науку о данных, на два кластера: один из студентов с навыками науки о данных, другой — с навыками программирования, *то k* будет два.
В некоторых других задачах значение *k* может быть неочевидным. В таких случаях для оценки наиболее подходящего количества кластеров для данного набора данных необходимо использовать итеративную процедуру проб и ошибок или эвристический алгоритм.

Логика кластеризации k-средних

В этом разделе описывается логика алгоритма кластеризации k-средних. Давайте посмотрим на них по одному.

Инициализация

Для их группировки алгоритм k-средних использует измерение расстояния для определения сходства или близости между точками данных. Перед использованием алгоритма k-средних необходимо выбрать наиболее подходящую меру расстояния. По умолчанию будет использоваться евклидово измерение расстояния. Кроме того, если в наборе данных есть выбросы, необходимо разработать механизм для определения критериев и удаления выбросов из набора данных.

Шаги алгоритма k-средних

Шаги, задействованные в алгоритме кластеризации k-средних, следующие:

Шаг 1 Выбираем количество кластеров, k .

Шаг 2 Среди точек данных мы случайным образом выбираем *k* точек в качестве центров кластера.

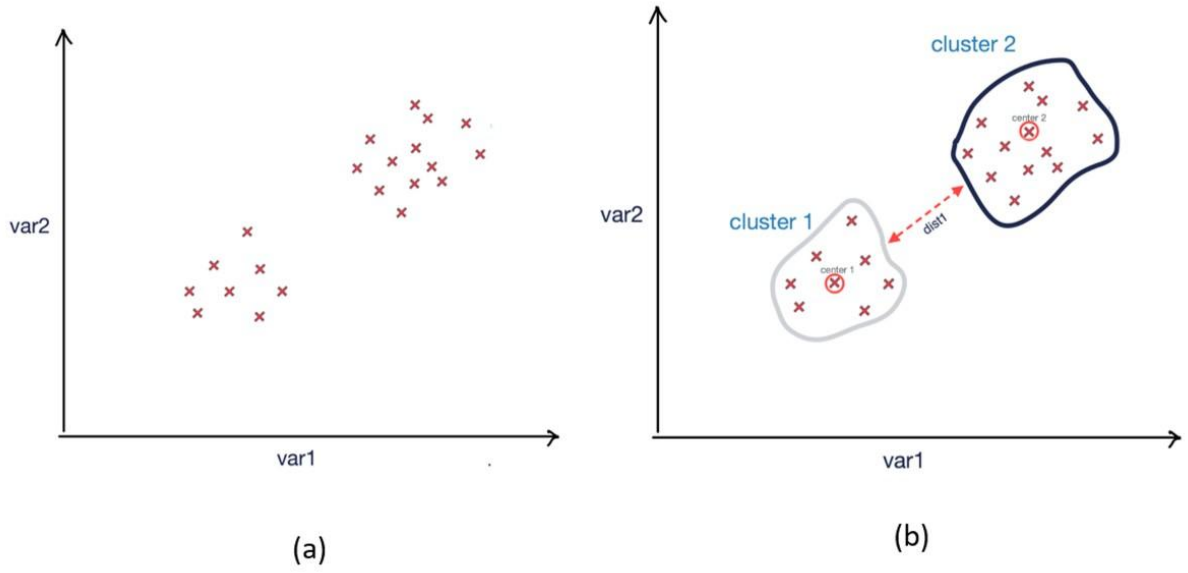
Шаг 3 На основе выбранной меры расстояния мы итеративно вычисляем расстояние от каждой точки в проблемном пространстве до каждого из k центров кластера. Исходя из размера набора данных, это может занять много времени шаг — например, если в кластере 10 000 точек и k = 3, это означает, что нужно вычислить 30 000 расстояний.

Шаг 4 Мы назначаем каждую точку данных в проблемном пространстве ближайшему центру кластера.

Шаг 5 Теперь каждая точка данных в нашем проблемном пространстве имеет назначенный центр кластера. Но мы ещё не закончили, так как выбор начальных центров кластеров основывался на случайном отборе. Нужно проверить, что текущие случайно выбранные центры кластеров действительно являются центром тяжести каждого скопления. Мы пересчитываем центры кластеров, вычисляя среднее значение составных точек данных каждого из k кластеров. Этот шаг объясняет, почему этот алгоритм называется k-средними.

Шаг 6 Если центры кластера сдвинулись на шаге 5, это означает, что нам нужно пересчитать назначение кластера для каждой точки данных. Для этого мы вернёмся к шагу 3, чтобы повторить этот вычислительный шаг. Если центры кластеров не сдвинулись или если наше заранее определённое условие остановки (например, количество максимальных итераций) было выполнено, то мы заканчиваем.

Следующий рисунок показывает результат запуска алгоритма k-средних в двумерном пространстве задач:



(a) Точки данных до кластеризации; (b) Результирующие кластеры после запуска алгоритма кластеризации k-средних

Обратите внимание, что два полученных кластера, созданных после выполнения k-средних, в этом случае хорошо дифференцированы.

Условие остановки

Для алгоритма k-средних по умолчанию условие остановки — когда на шаге 5 больше нет смещения центров кластеров. Но, как и во многих других алгоритмах, алгоритмы k-средних могут долго сходиться, особенно при обработке больших наборов данных в пространстве задач с высокой размерностью. Вместо того чтобы ждать сходимости алгоритма, можно явно определить условие остановки следующим образом:

- Задавая максимальное время выполнения:
  - **Условие остановки**: *t>tmax*, где *t* — текущее время выполнения, а *tmax* — максимальное время выполнения, установленное алгоритмом. Задавая максимальное количество итераций:
- **Условие** остановки: *если m>mmax*, где *m* — текущая итерация, а *mmax* — максимальное количество итераций, установленное алгоритмом.

Кодирование алгоритма k-средних

Давайте посмотрим, как можно запрограммировать алгоритм k-means на Python:

1. Сначала импортируем пакеты, которые нам понадобятся для алгоритма k-средних. Обратите внимание, что мы импортируем пакет `sklearn` для кластеризации k-means:

```
Из sklearn Import Cluster
Import Pandas as PD Import
Numpy as NP
```

1. Чтобы использовать кластеризацию k-средних, давайте создадим 20 точек данных в двумерном пространстве задач, которое мы будем использовать для кластеризации k-средних:

```
Набор данных = 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] })
```

1. Давайте создадим два кластера (*k* = 2), а затем создадим кластер, вызвав функции `подгонки`:

```
myKmeans = кластер. KMeans(n_clusters=2)
myKmeans.fit(dataset)
```

1. Давайте создадим переменную с именем `centroid` — это массив, содержащий расположение центра сформированных кластеров. В нашем случае, при *k* = 2, массив будет иметь размер 2. Давайте также создадим другую переменную с названием `label` , которая представляет назначение каждой точки данных одному из двух кластеров. Поскольку существует 20 точек данных, размер массива будет 20:

```
центроиды = myKmeans.cluster_centers_ метки
= myKmeans.labels_
```

1. Теперь распечатаем эти два массива — `центроиды` и `метки` :

```
In [3]: print(labels)

        [1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1]

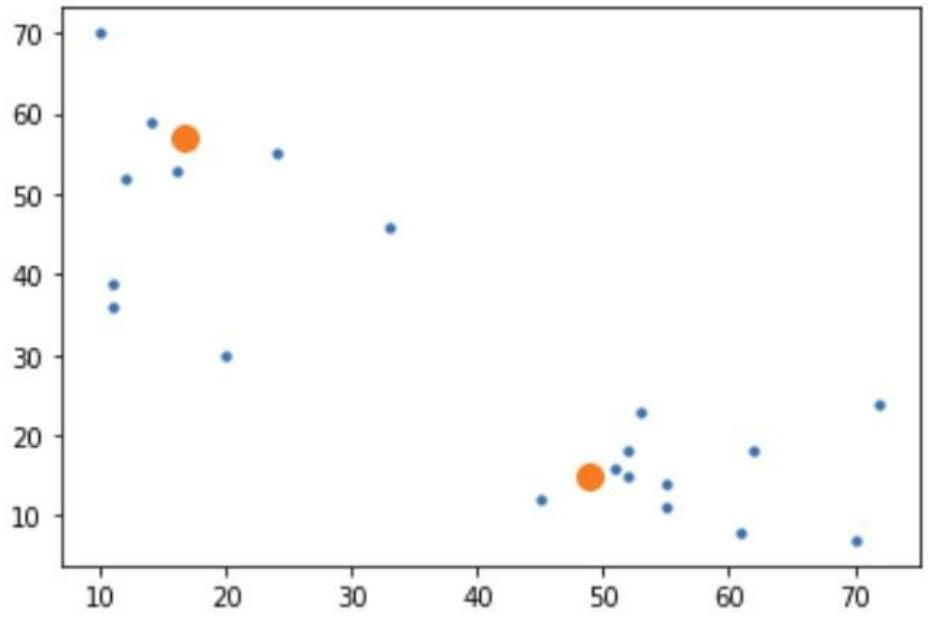
In [6]:
        print(centroids)

        [[57.09090909 15.09090909]
         [16.77777778 48.88888889]]
```

Обратите внимание, что первый массив показывает назначение кластера каждой точке данных, а второй — два центра кластера.

1. Давайте построим график и рассмотрим кластеры с помощью `matplotlib` :

```
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()
```



Обратите внимание, что крупные точки на графике — это центроиды, определённые алгоритмом k-средних.

Ограничение кластеризации k-средних

Алгоритм k-средних разработан как простой и быстрый алгоритм. Из-за намеренной простоты конструкции она имеет следующие ограничения:

- Главное ограничение кластеризации k-средних заключается в том, что начальное количество
- кластеров должно быть заранее определено. Первоначальное распределение центров кластеров случайно. Это означает, что каждый раз, когда алгоритм запускается, он может выдавать немного
- разные кластеры.

Каждая точка данных назначается только одному кластеру. k-средняя кластеризация чувствительна к выбросам.

Иерархическая кластеризация

К-средняя кластеризация использует подход сверху вниз, поскольку алгоритм начинается с самых важных точек данных, а именно центров кластера. Существует альтернативный подход кластеризации, при котором вместо начала сверху алгоритм начинается снизу. В данном контексте дно — это каждая из отдельных точек данных в проблемном пространстве. Решение — продолжать группировать похожие

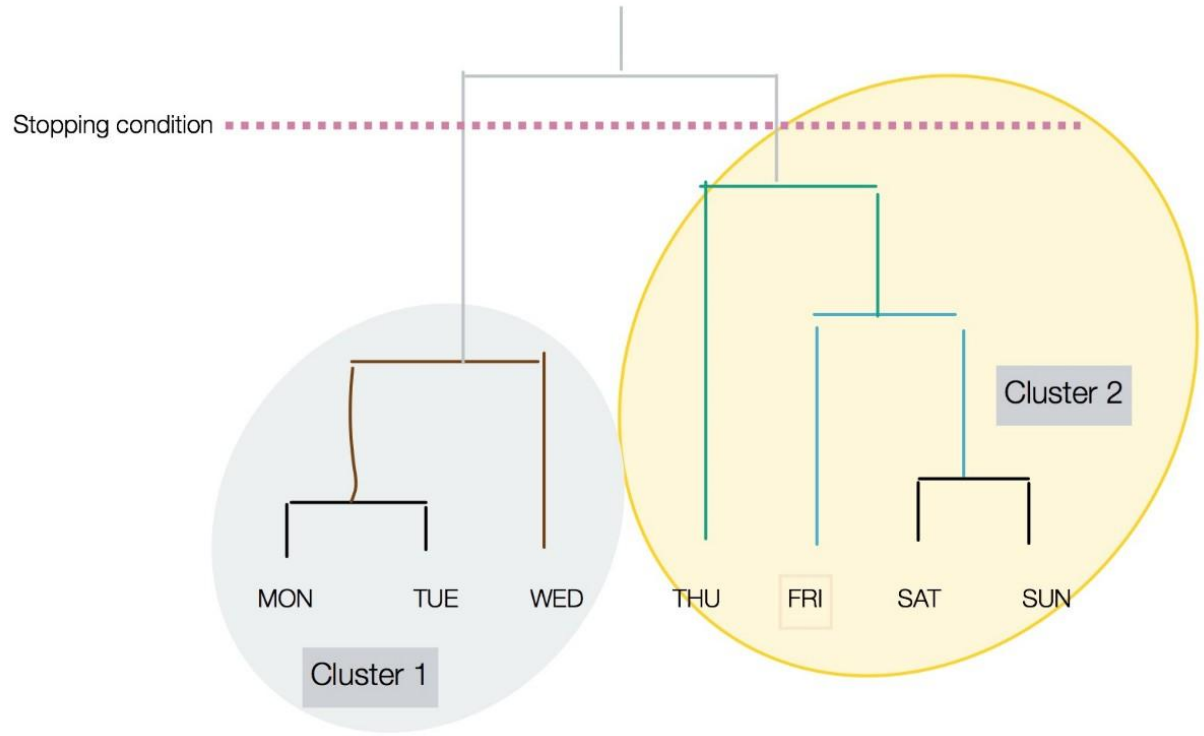
данные по мере продвижения к центрам кластера. Этот альтернативный подход снизу вверх используется алгоритмами иерархической кластеризации и обсуждается в этом разделе.

Этапы иерархической кластеризации

Иерархическая кластеризация включает следующие этапы:

А. Мы создаём отдельный кластер для каждой точки данных в нашем проблемном пространстве. Если наше пространство задач состоит из 100 точек данных, то оно начнётся со 100 кластеров.
Б. Мы группируем только те точки, которые находятся ближе всего друг к другу.
В. Мы проверяем условия остановки; Если условие остановки ещё не выполнено, повторяем шаг 2.

Полученная кластерная структура называется **дендрограммой**. В дендрограмме высота вертикальных линий определяет, насколько близко находятся элементы, как показано на следующей диаграмме:



Обратите внимание, что условие остановки показано пунктирной линией на предыдущем рисунке.

Кодирование иерархического алгоритма кластеризации

Давайте узнаем, как можно написать иерархический алгоритм на Python:

1. Сначала мы импортируем `AgglomerativeClustering` из библиотеки `sklearn.cluster` вместе с `пакетами pandas` и `numpy` :

```
from sklearn.cluster import AgglomerativeClustering
import pandas as pd import numpy as np
```

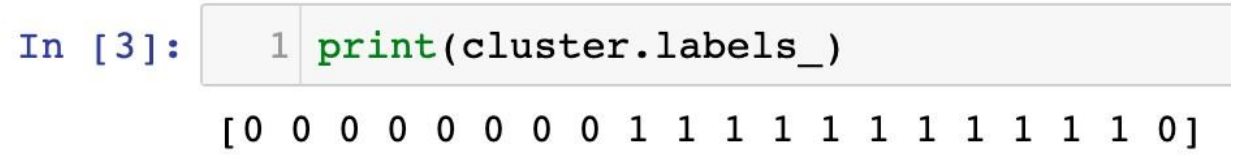
1. Затем мы создадим 20 точек данных в двумерном пространстве задач:

```
Набор данных = 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] })
```

1. Затем создаём иерархический кластер, задавая гиперпараметры. Мы используем `функцию fit_predict` для обработки алгоритма:

```
cluster = AgglomerativeClustering(n_clusters=2, affinity='евклидов', linkage='ward')
cluster.fit_predict(набор данных)
```

1. Теперь рассмотрим связь каждой точки данных с двумя созданными кластерами:



Вы можете увидеть, что распределение кластеров для иерархических и k-средних алгоритмов очень похоже.

Оценка кластеров

Цель качественной кластеризации — чтобы точки данных, принадлежащие отдельным кластерам, были дифференцируемыми. Это подразумевает следующее:

- Точки данных, относящиеся к одному кластеру, должны быть максимально
- похожими. Точки данных, относящиеся к отдельным кластерам, должны быть максимально разными.

Человеческая интуиция может использоваться для оценки результатов кластеризации через визуализацию кластеров, но существуют математические методы, которые могут количественно оценить качество кластеров. Анализ силуэта — один из таких методов, сравнивающий плотность и разделение в кластерах, созданных алгоритмом k-средних. Силуэт рисует график, который показывает близость каждой точки в конкретном кластере относительно других точек соседних скоплений. Он связывает число в диапазоне [-0, 1] с каждым кластером. Следующая таблица показывает, что означают цифры в этом диапазоне:

Значение диапазона	**Описание**
0.71–1.0 Отлично	Это означает, что кластеризация k-средних привело к формированию групп, которые довольно сильно различимы друг от друга.
0.51–0.70 Разумно	Это означает, что кластеризация k-средних привело к формированию групп, которые в некоторой степени являются различимы друг от друга.
0.26–0.50 Слабый	Это означает, что кластеризация k-средних приводила к группировке, но качество На группировку не стоит полагаться.
<0.25 Нет кластеризации невозможность было найти	Используя выбранные параметры и используемые данные, создать Группировка с помощью кластеризации K-средств.

Обратите внимание, что каждый кластер в задачном пространстве получит отдельную оценку.

DBSCAN

DBSCAN (пространственная кластеризация приложений с шумом, основанная на плотности) — это метод обучения без надзора, который осуществляет кластеризацию на основе плотности точек. Основная идея основана на предположениях, что если мы сгруппируем точки данных в перегруженном или высокоплотном пространстве, мы можем добиться значимой кластеризации. Этот подход к кластеризации имеет два важных вывода:

- Используя эту идею, алгоритм, скорее всего, сгруппирует существующие точки вместе, независимо от их формы. Эта методология помогает создавать кластер произвольных форм.
- В отличие от алгоритма k-средних, нам не нужно указывать количество кластеров, и алгоритм может обнаруживать соответствующее количество групп в данных.

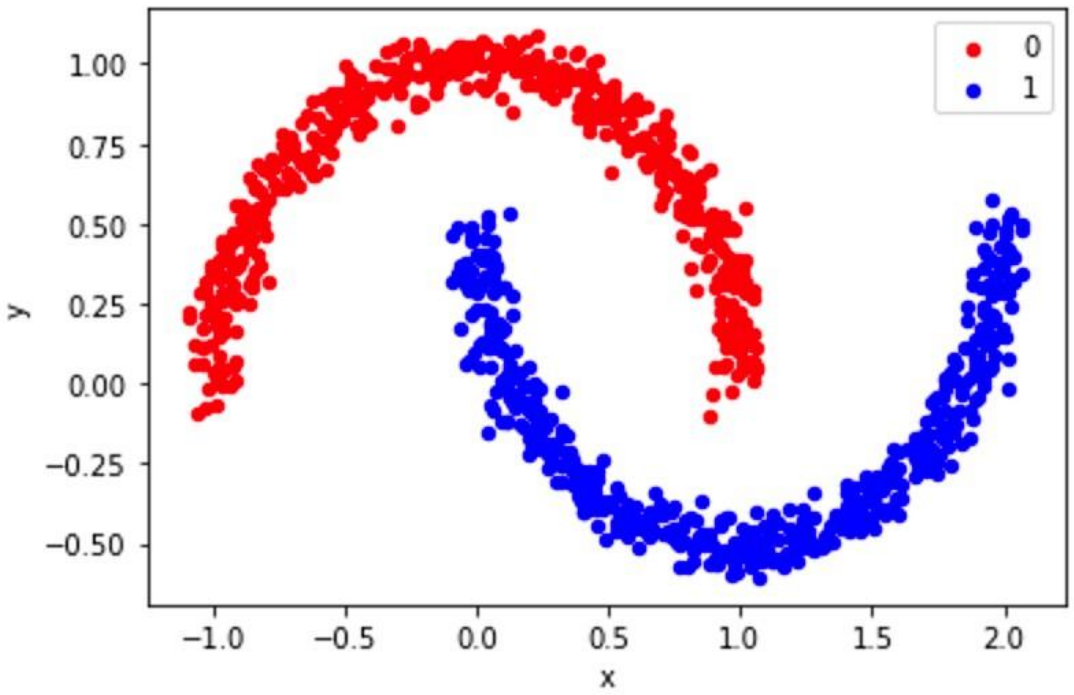
В алгоритме DBSCAN участвуют следующие шаги:

- Алгоритм устанавливает окрестность вокруг каждой точки данных. Это достигается путем подсчёта количества точек данных на расстоянии, обычно представленного *переменной eps*. Расстояние по умолчанию определяется евклидовой мерой расстояния.
- Далее алгоритм количественно определяет плотность каждой точки данных. Он использует переменные с именем min_samples. Если точка данных содержит более min_samples, то он идентифицируется как основной экземпляр. Очевидно, что в более перенаселенных или плотных регионах будет больше основных инстансов.
- Каждый из идентифицированных соседей идентифицирует кластер. Окрестность одного экземпляра ядра может содержать другие экземпляры ядра.
- Любая точка данных, не являющаяся экземпляром ядра или не находящаяся в окрестности экземпляра ядра, считается исключением.

Давайте посмотрим, как можно создавать кластеры с помощью DBSCAN в Python. Сначала мы импортируем необходимые функции из библиотеки sklearn.

```
In [13]: from sklearn.cluster import DBSCAN
         from sklearn.datasets import make_moons
```

Давайте попробуем DBSCAN с относительно сложной задачей кластеризации, называемой полулунами. Набор данных можно создать с помощью встроенной функции. Эта тестовая задача подходит для алгоритмов, способных изучать нелинейные границы классов. Для этого следующая `функция make_moons()` создаст вихревой узор, или две луны. Мы можем контролировать, насколько шумные формы Луны и сколько образцов нужно генерировать. Набор данных выглядит так:



Для использования DBSCAN необходимо предоставить параметры EPS и min_samples и обсудить:

```
In [ ]: dbscan = DBSCAN(eps=0.05, min_samples=5)

In [3]: dbscan.fit(X)

Out[3]: DBSCAN(eps=0.05)
```

Применение кластеризации

Кластеризация используется там, где нам нужно было выявить скрытые закономерности в наборах данных. В государственных случаях использования кластеризация может использоваться для следующих целей:

- Анализ горячих точек преступности
- Демографический социальный анализ

В маркетинговых исследованиях кластеризация может использоваться для следующих целей:

- Сегментация рынка
- Таргетированная реклама
- Классификация клиентов

Анализ основных компонентов (**PCA**) также используется для общего изучения данных и удаления шума из данных в реальном времени, таких как торговля на фондовом рынке.

Уменьшение размерности

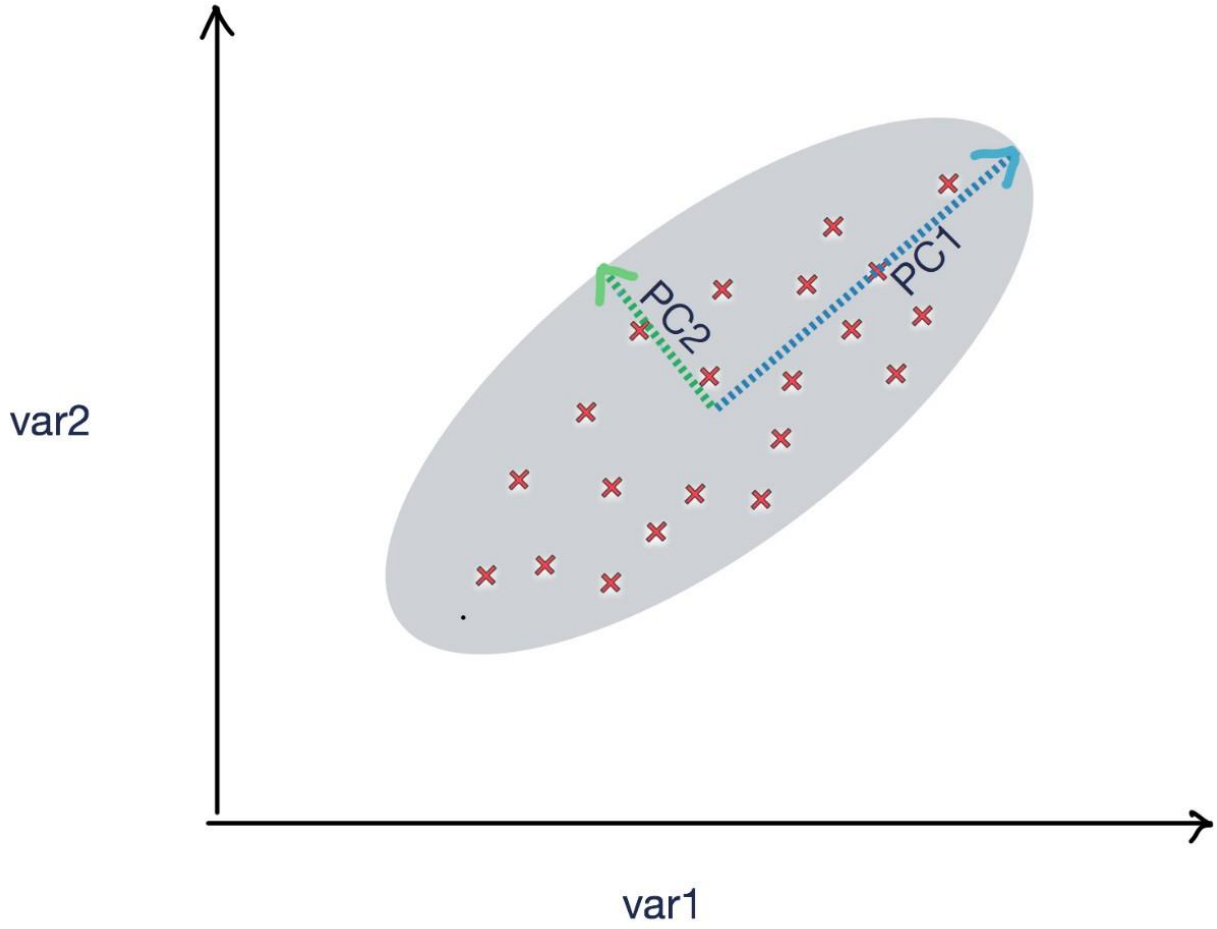
Каждая особенность в наших данных соответствует измерению в нашем проблемном пространстве. Минимизация числа признаков для упрощения пространства задач называется **уменьшением размерности**. Это можно сделать одним из следующих двух способов:

- **Выбор признаков**: выбор набора признаков, важных в контексте задачи, которую мы пытаемся реализовать
  решит
- **Агрегация** признаков: Объединение двух или более признаков для уменьшения размерности с помощью одного из следующих алгоритмов:
  - **PCA**: линейный алгоритм машинного обучения без надзора
  - **Линейный дискриминантный анализ** (**LDA**): линейный контролируемый алгоритм машинного **обучения Анализ основных компонентов ядра**: нелинейный алгоритм

Давайте подробнее рассмотрим один из популярных алгоритмов уменьшения размерности, а именно PCA.

Анализ основных компонентов

PCA — это метод неконтролируемого машинного обучения, который можно использовать для уменьшения размерностей с помощью линейного преобразования. На следующем рисунке мы видим два основных компонента — **PC1** и **PC2**, которые показывают форму распределения точек данных. PC1 и PC2 можно использовать для суммирования данных с соответствующими коэффициентами:



Рассмотрим следующий код:

```
из 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 рассмотрим коэффициенты нашей модели PCA:

```
In [36]: print(pd.DataFrame(pca.components_,columns=X.columns))
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
0	0.361387	-0.084523	0.856671	0.358289
1	0.656589	0.730161	-0.173373	-0.075481
2	-0.582030	0.597911	0.076236	0.545831
3	-0.315487	0.319723	0.479839	-0.753657

Coefficients for PC1
Coefficients for PC2
Coefficients for PC3
Coefficients for PC4

Обратите внимание, что оригинальный DataFrame имеет четыре особенности: `Длина Сепаль` , `Ширина Лепестка`, `Длина` лепестка и `Ширина` лепестка. Предыдущий DataFrame задаёт коэффициенты четырёх основных компонентов: PC1, PC2, PC3 и PC4 — например, первая строка указывает коэффициенты PC1, которые можно использовать для замены исходных четырёх переменных. Исходя из этих коэффициентов, мы можем вычислить компоненты PCA для нашего входного DataFrame X:

```
pca_df=(pd. DataFrame(pca.components_,columns=X.columns))
# Вычислим PC1 с помощью порождённых коэффициентов
X['PC1'] = X['Sepal.Length']* pca_df['Sepal.Length'][0] + X['Sepal.Width']* pca_df['Sepal.Width']
# Давайте вычислим PC2
X['PC2'] = X['Sepal.Length']* pca_df[''Sepal.Length'][1] + X['Sepal.Width']*
pca_df['Sepal.Width'] #Let вычислить PC3
X['PC3'] = X['Sepal.Length']* pca_df['Sepal.Length'][2] + X['Sepal.Width']* pca_df['Sepal.Width']
# Давайте вычислим PC4
X['PC4'] = X['Sepal.Length']* pca_df['Sepal.Length'][3] + X['Sepal.Width']* pca_df['Sepal.Width']
```

Теперь распечатаем X после вычисления компонентов PCA:

	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

Теперь распечатаем коэффициент дисперсии и попробуем понять последствия использования PCA:

```
In [37]: print(pca.explained_variance_ratio_)
[0.92461872 0.05306648 0.01710261 0.00521218]
```

Коэффициент дисперсии указывает на следующее:

- Если мы решим заменить исходные четыре признака на PC1, то сможем охватить примерно 92,3% дисперсии исходных переменных. Мы введём некоторые приближения, не учитывая 100% дисперсии исходных четырёх признаков.
- Если мы решим заменить исходные четыре признака на PC1 и PC2, то получим дополнительно 5,3 % дисперсии исходных переменных.
- Если мы решим заменить исходные четыре функции на PC1, PC2 и PC3, то теперь получим ещё 0,017 % дисперсии исходных переменных.

Если мы решим заменить исходные четыре признака четырьмя главными компонентами, то получим 100% дисперсии исходных переменных (92,4 + 0,053 + 0,017 + 0,005), но замена четырёх исходных признаков на четыре основных компонента бессмысленна, так как мы не уменьшили размерности и ничего не достигли.

Ограничения PCA

Ниже приведены ограничения PCA:

- PCA может использоваться только для непрерывных переменных и не имеет значения для
- переменных категорий.

При агрегировании PCA аппроксимирует компонентные переменные; Это упрощает проблему размерности в ущерб точности. Этот компромисс следует тщательно изучить перед использованием PCA.

Ассоциация управляет добычей полезных ископаемых

Паттерны в конкретном наборе данных — это сокровище, которое нужно найти, понять и извлечь для получения содержащейся в них информации. Существует важный набор алгоритмов, которые пытаются сосредоточиться на анализе паттернов в данном наборе данных. Один из наиболее популярных алгоритмов этого класса алгоритмов называется **алгоритмом майнинга правил ассоциации** , который предоставляет нам следующие возможности:

- Способность измерять частоту паттерна
- Способность устанавливать *причинно-следственную связь между паттернами.*
- Способность количественно оценивать полезность паттернов путём сравнения их точности с случайными догадками

Примеры использования

Майнинг правил ассоциаций используется при исследовании причинно-следственных связей между различными переменными в наборе данных. Ниже приведены примеры вопросов, на которые может помочь ответить:

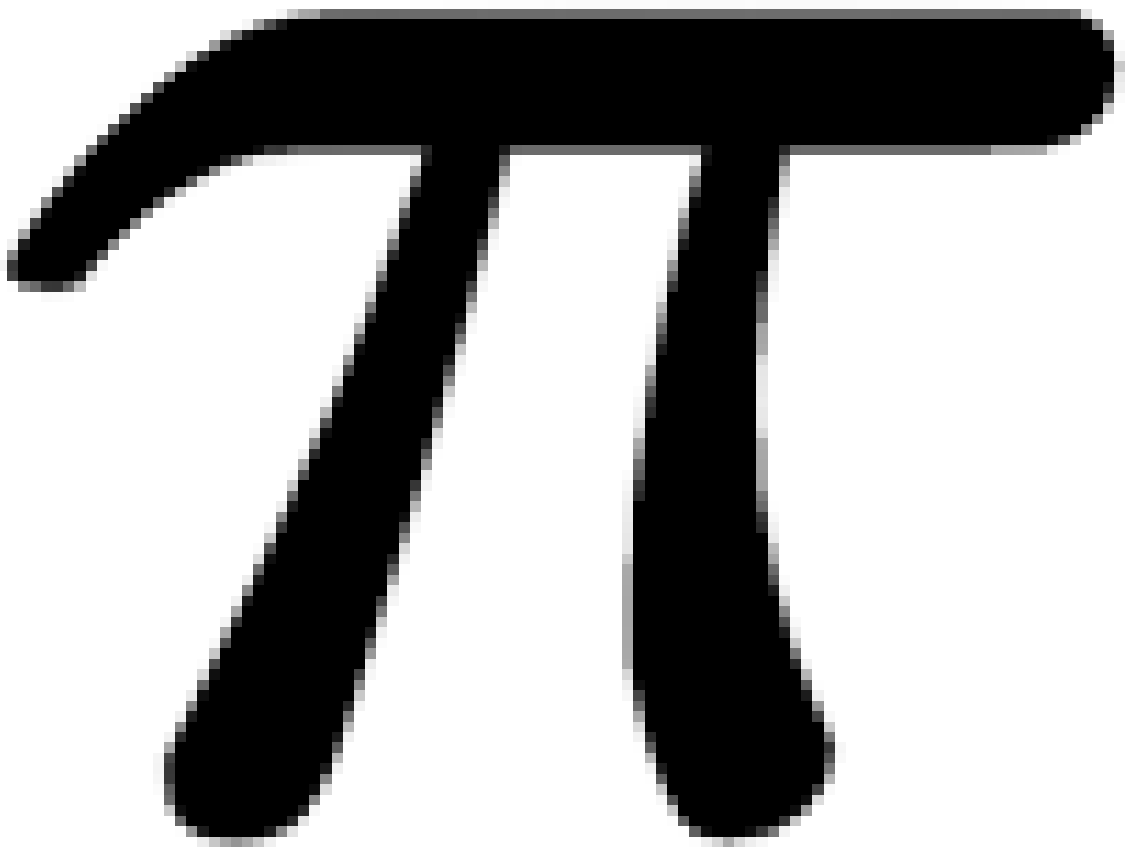
- Какие значения влажности, облачности и температуры могут привести к дождю завтра?
- Какой тип страхового иска может указывать на мошенничество?
- Какие комбинации лекарств могут привести к осложнениям у пациентов?

Анализ корзины рынка

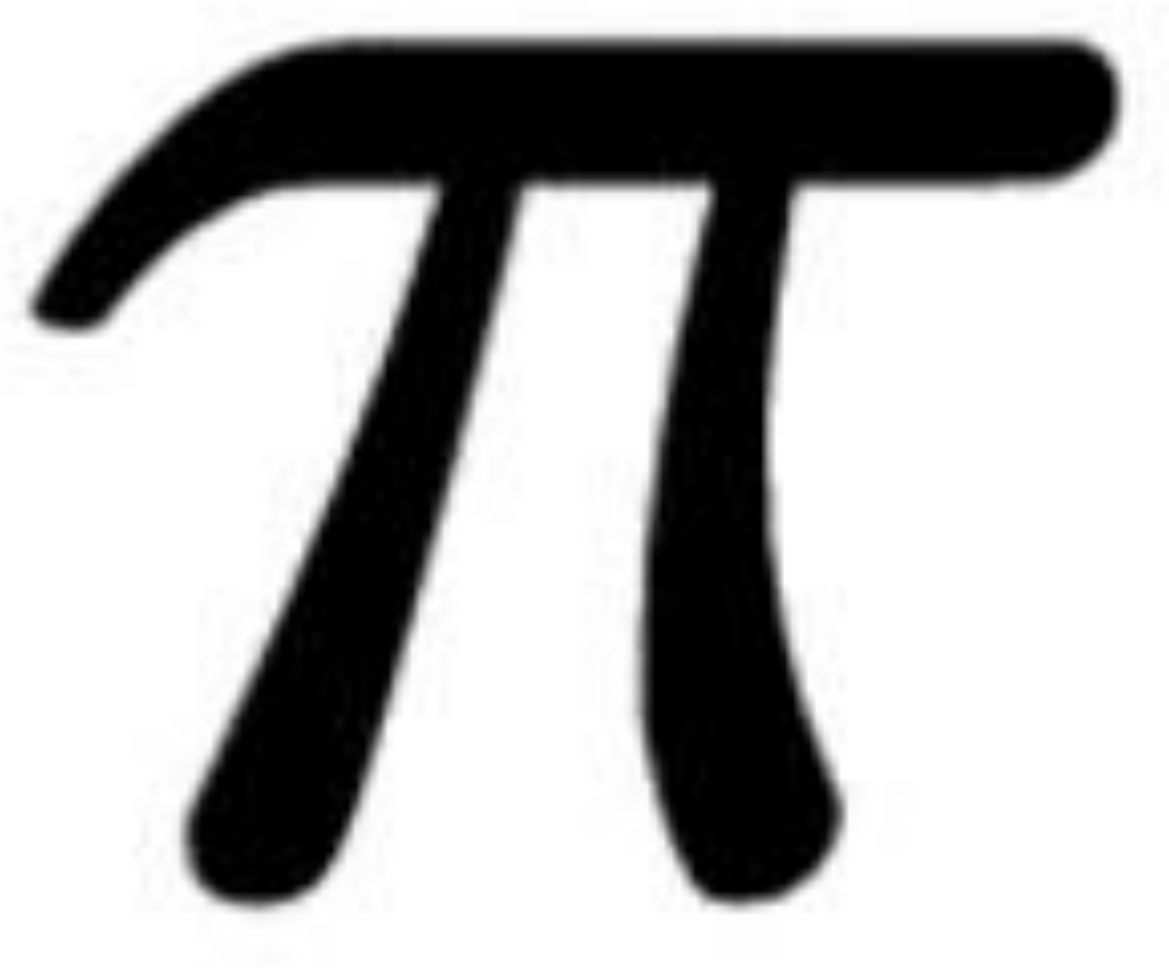
В этой книге двигатели рекомендаций обсуждаются в *главе 8*, «*Алгоритмы нейронных сетей*». Анализ корзины — это более простой способ изучения рекомендаций. В анализе корзин наши данные содержат только информацию о том, какие товары были куплены вместе. В нём нет информации о пользователе или о том, понравились ли ему отдельные предметы. Обратите внимание, что получить эти данные гораздо проще, чем рейтинги. Например, такие данные генерируются при покупке в Walmart, и для их получения не требуется специальная техника. Эти данные, собираемые в течение определённого времени, называются **транснациональными данными**. Когда анализ правил ассоциации применяется к транснациональным наборам данных о тележках, используемых в магазинах дома, супермаркетах и сетях быстрого питания, это называется **анализом корзины рынка**. Он измеряет условную вероятность совместной покупки набора товаров, что помогает ответить на следующие вопросы:

- Как оптимально размещать товары на полке?
- Как должны отображаться товары в маркетинговом каталоге?
- Что следует рекомендовать, исходя из покупательских моделей пользователя?

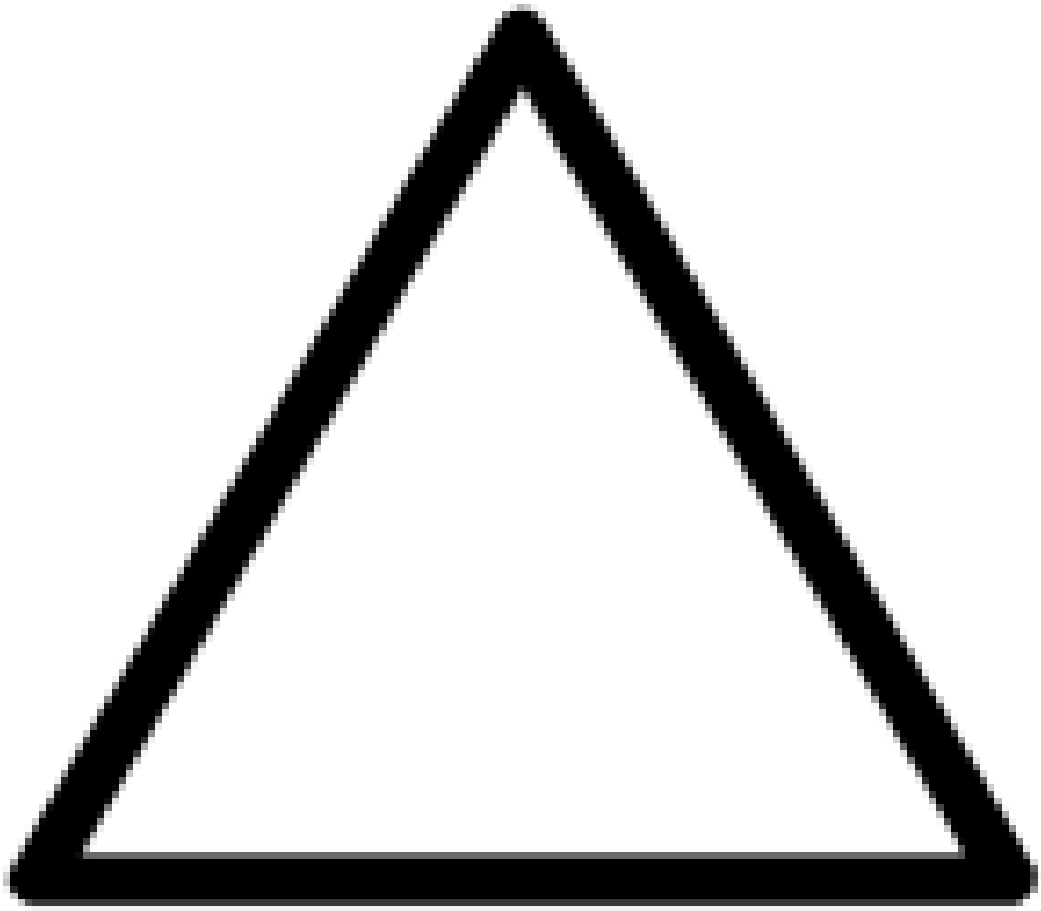
Поскольку анализ корзины рынка позволяет оценить, как товары взаимосвязаны друг с другом, его часто используют в массовой розничной торговле, такой как супермаркеты, магазины дома, аптеки и сети быстрого питания. Преимущество анализа корзины в том, что результаты почти сами по себе понятны, то есть их легко понять бизнес-пользователям. Давайте посмотрим на типичный супермаркет. Все уникальные предметы, доступные в магазине, могут быть представлены набором,



= {пункт 1 , элемент $_2$, . . . , элемент $_m$ }. Если этот супермаркет продаёт 500 разных товаров, то



будет набор размера 500. Люди будут покупать товары в этом магазине. Каждый раз, когда кто-то покупает товар и оплачивает на кассе, он добавляется в набор товаров в конкретной транзакции, называемый **набором товаров**. За определённый период времени транзакции объединяются в множество, представленное как



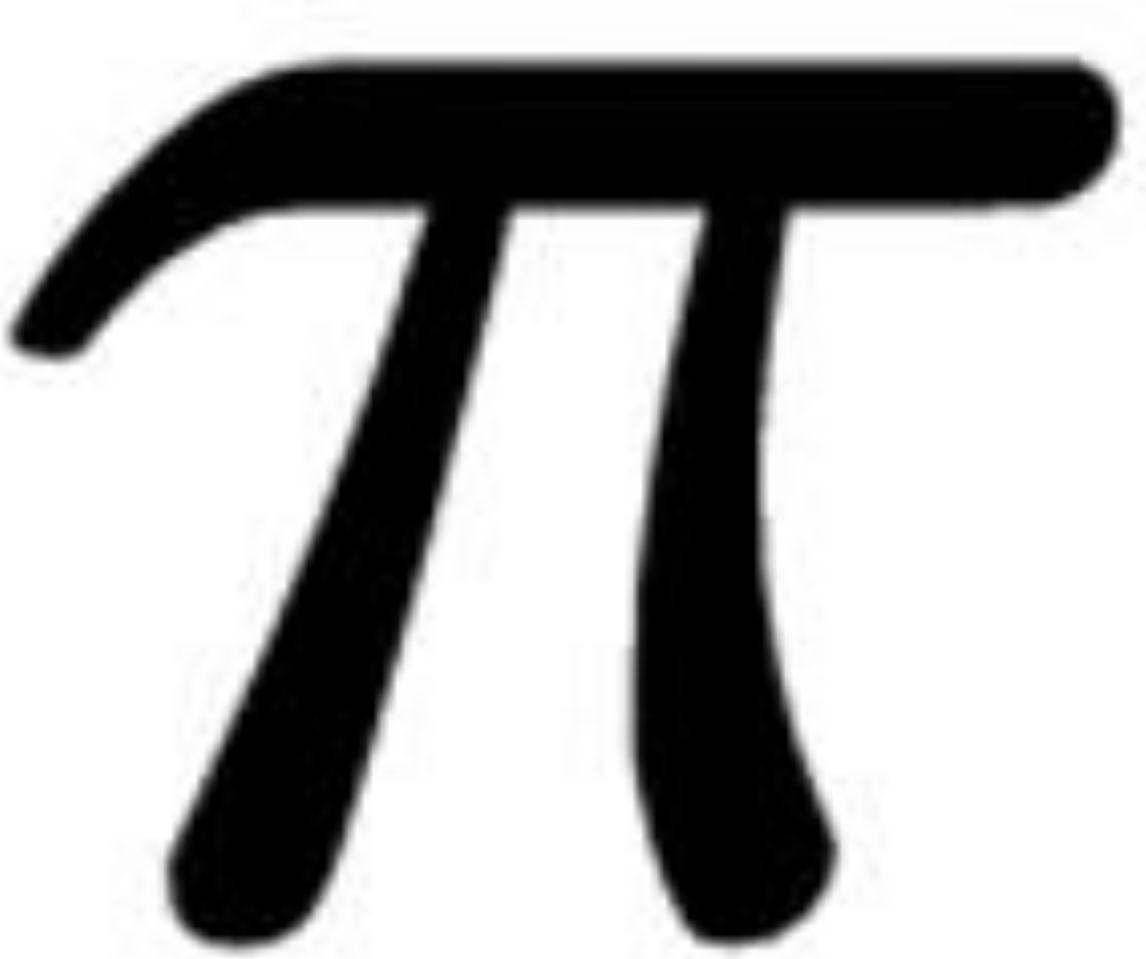
, где



= {t_1 ,t_2 , . . . ,t_n }. Рассмотрим следующие простые данные транзакций, состоящие всего из четырёх транзакций. Эти сделки обобщены в следующей таблице:

Т1 Калитки, накладки Т2
Бита, калитки, накладки,
шлем Т3 Шлем, мяч Т4
Бита, накладки, шлем

Давайте рассмотрим этот пример подробнее:



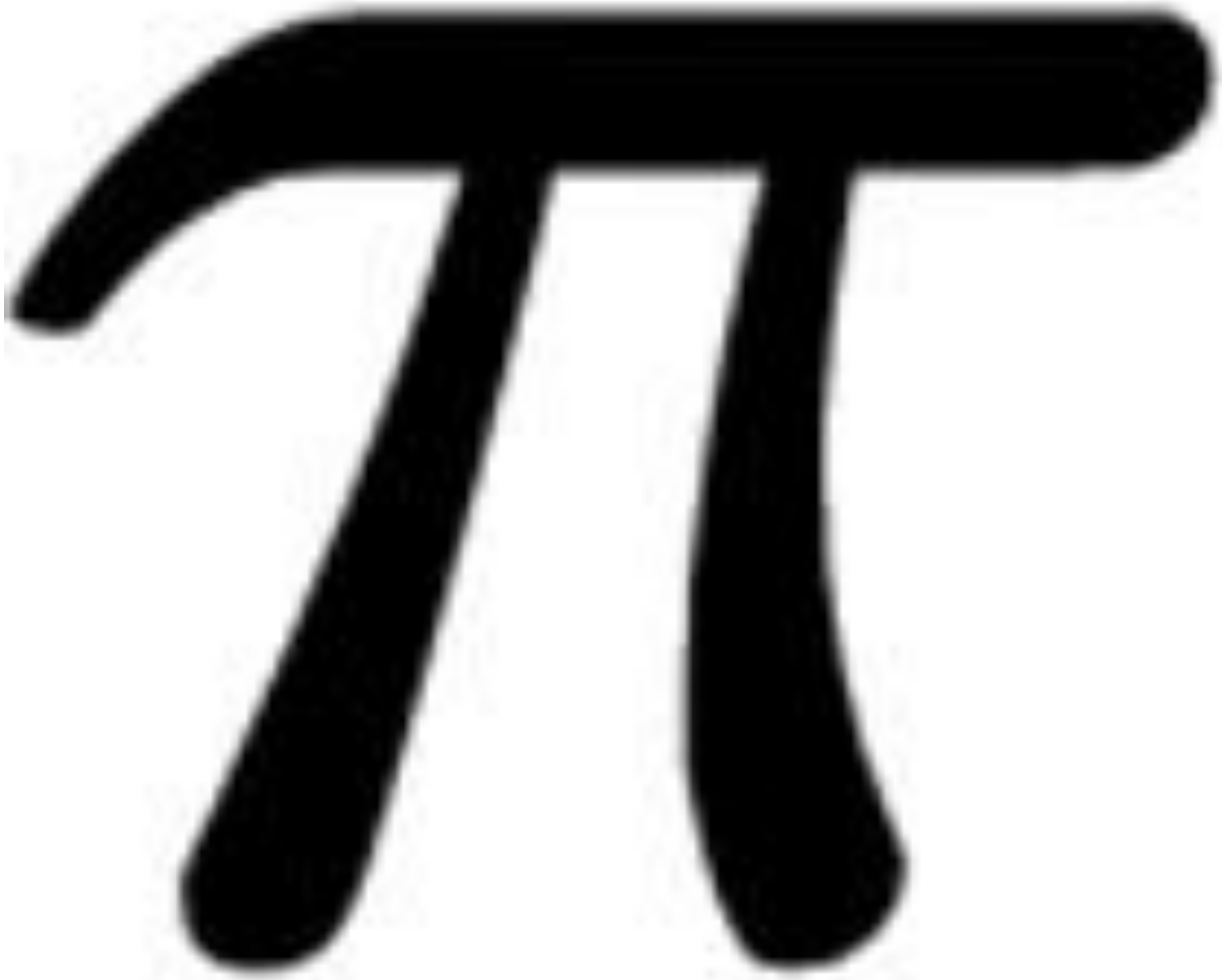
= {бита, калитки, накладки, шлем, мяч}, что представляет все уникальные предметы, доступные в магазине. Рассмотрим одну из транзакций, t3, из



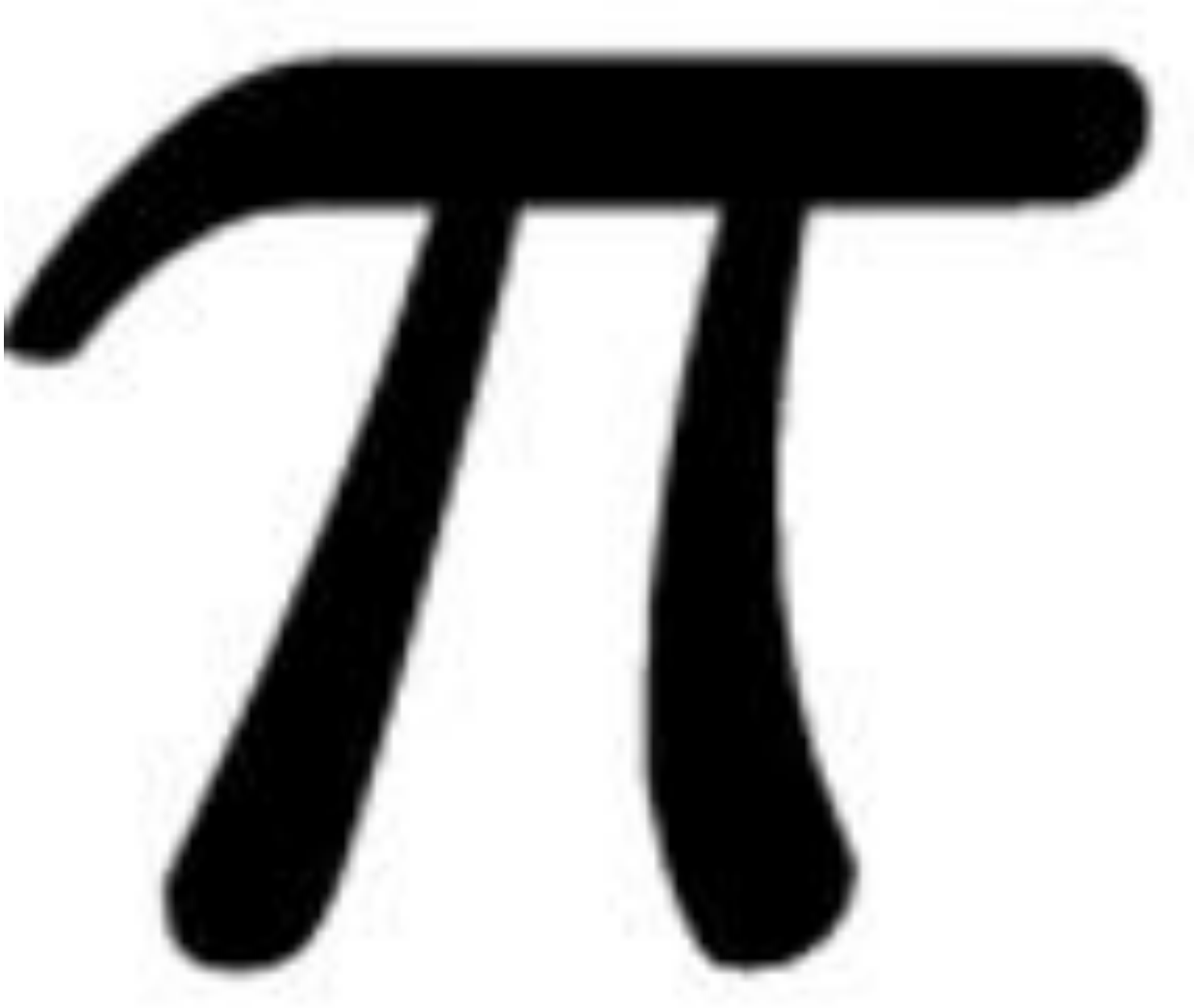
. Обратите внимание, что предметы, купленные в t3, могут быть представлены в itemsett3= {helmet,ball}, что означает, что клиент купил два предмета. Поскольку в этом наборе предметов два предмета, размер itemsett5 называется двумя.

Правила ассоциации

Правило ассоциации математически описывает элементы отношений, участвующие в различных сделках. Это делается путём исследования взаимосвязи между двумя наборами предметов в виде *X* => *Y*, где *X* ⊂



,Y⊂



. Кроме того, *X* и *Y* — это неперекрывающиеся элементы; что означает, что

$$X \bigcap Y = \varnothing$$

. Правило ассоциации можно описать следующим образом:

```
{шлем, мячи}=> {велосипед}
```

Здесь {helmet,ball} — *это X* , а {ball} — *Y*.

Виды правил

Запуск ассоциативных алгоритмов анализа обычно приводит к генерации большого количества правил из набора транзакций. Большинство из них бесполезны. Чтобы выбрать правила, которые могут дать полезную информацию, мы можем классифицировать их как один из следующих трёх типов:

- Тривиально
- Необъяснимое,
- подлежащее

действию

Давайте рассмотрим каждый из этих типов подробнее.

Тривиальные правила

Среди множества сгенерированных правил многие из них окажутся бесполезными, поскольку они обобщают общие знания о бизнесе. Их называют тривиальными правилами. Даже если доверие к тривиальным правилам высоко, они остаются бесполезными и не могут быть использованы для принятия решений на основе данных. Мы можем спокойно игнорировать все мелкие правила. Ниже приведены примеры тривиальных правил:

- Любой, кто прыгнет с высотного здания, скорее всего, погибнет.
- Усердная работа ведёт к лучшим результатам на экзаменах.
- Продажи обогревателей увеличиваются с понижением температуры
- Вождение автомобиля с превышением ограниченной скорости на шоссе увеличивает вероятность аварии.

Необъяснимые правила

Среди правил, которые генерируются после запуска алгоритма правил ассоциации, самые сложные в использовании без очевидного объяснения. Обратите внимание, что правило может быть полезным только в том случае, если оно помогает нам обнаружить и понять новый паттерн, который, как ожидается, в конечном итоге приведёт к определённому курсу действий. Если это не так, и мы не можем объяснить, почему событие *X* привело к событию *Y*, то это необъяснимое правило, потому что это просто математическая формула, которая в итоге исследует бессмысленную связь между двумя событиями, не связанными и независимыми. Ниже приведены примеры необъяснимых правил:

- Люди, носящие красные рубашки, обычно лучше сдают экзамены.
- Зелёные велосипеды чаще подвергаются краже.
- Люди, которые покупают огурцы, в итоге покупают подгузники.

Практические правила

Практические правила — это золотые правила, которые мы ищем. Их понимает бизнес, и они приводят к инсайтам. Они могут помочь нам выявить возможные причины события, если представить аудитории, знакомой с бизнес-сферой — например, практические правила могут указывать на лучшее размещение конкретного товара в магазине на основе текущих покупательских моделей. Они также могут предложить, какие товары размещать вместе, чтобы увеличить шансы на продажу, так как пользователи обычно покупают их вместе. Ниже приведены примеры применимых правил и соответствующих им действий:

- **Правило 1:** Отображение рекламы на аккаунтах пользователей в социальных сетях увеличивает вероятность продаж.

Практический элемент: предлагает альтернативные способы рекламы продукта

- **Правило 2:** Создание большего количества ценовых категорий увеличивает вероятность продаж.

Действующий товар: один товар может быть рекламирован на распродаже, а цена другого может повышаться.

Правила ранжирования

Правила ассоциации измеряются тремя способами:

- Поддержка (частота) элементов
- Подъём
- уверенности

Давайте рассмотрим их подробнее.

Поддержка

Показатель поддержки — это число, которое количественно определяет, насколько часто рассматриваемая закономерность находится в нашем наборе данных. Он рассчитывается сначала путём подсчёта количества случаев нашего интересного паттерна, а затем деления на общее количество всех транзакций. Рассмотрим следующую формулу для конкретного *itemsseta*: *numItemseta = Количество транзакций, содержащих itemsetanumtotal = Общее количество транзакций*

$$support(itemset_a) = \frac{numItemset_a}{num_{total}}$$

Просто посмотрев на поддержку, можно понять, насколько редко возникает паттерн. Низкая поддержка означает, что мы ищем редкое событие. Например, если *itemseta = {шлем, мяч}* встречается в двух из шести транзакций, то support (itemseta) = 2/6 = 0,33.

Уверенность

Доверие — это число, которое количественно определяет, насколько сильно мы можем связать левую часть (*X*) с правой (*Y*) путём вычисления условной вероятности. Он вычисляет вероятность того, что событие *X* приведёт к событию *Y*, если произошло событие *X* . Математически рассмотрим правило *X* = > *Y*. Уверенность этого правила представлена как уверенность (*X* => *Y*) и измеряется следующим образом:

$$confidence(X \Rightarrow Y) = \frac{support(X \cup Y)}{support(X)}$$

Давайте рассмотрим пример. Рассмотрим следующее правило:

{шлем, мяч} => {калитки}

Уверенность этого правила вычисляется 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 значит, что если у кого-то есть {шлем, мячи} в корзине, то вероятность того, что у него тоже будут калитки, составляет 0,5 или 50 процентов.

Подъёмник

Другой способ оценить качество правила — рассчитать подъёмную силу. Подъём возвращает число, которое количественно показывает, насколько улучшение было достигнуто по правилу при предсказании результата по сравнению с простым предположением результата справа в уравнении. Если бы наборы *элементов X* и *Y* были независимы, то подъём вычисляется следующим образом:

$$Lift(X \Rightarrow Y) = \frac{support(X \cup Y)}{support(X) \times support(Y)}$$

Алгоритмы анализа ассоциаций

В этом разделе мы рассмотрим следующие два алгоритма, которые можно использовать для анализа ассоциаций:

- **Алгоритм Априори**: предложен Агравалом, Р. и Срикантом в 1994 году.
- **Алгоритм роста** FP: улучшение, предложенное Ханом и др. в 2001 году.

Давайте рассмотрим каждый из этих алгоритмов.

Априорный алгоритм

Априорный алгоритм — это итеративный и многофазный алгоритм, используемый для генерации правил ассоциаций. Она основана на подходе поколения и тестирования. Перед выполнением априорного алгоритма необходимо определить две переменные: порог поддержки и порог доверия. Алгоритм состоит из следующих двух этапов:

- **Фаза генерации кандидатов**: генерируются наборы кандидатов, которые содержат наборы всех элементов выше порога поддержки.
- **Фаза** фильтрации: он фильтрует все правила ниже ожидаемого порога доверия.

После фильтрации ответом становятся полученные правила.

Ограничения априорного алгоритма

Основным узким местом в априорном алгоритме является генерация кандидатов правил в первой фазе — например,



= {элемент 1 , элемент $_2$, . . . , элемент $_m$ } может дать 2 млн возможных наборов элементов. Благодаря многофазному дизайну он сначала генерирует эти наборы предметов, а затем работает над поиском частых наборов. Это ограничение представляет собой огромное узкое место производительности и делает алгоритм априорного использования непригодным для крупных элементов.

Алгоритм роста FP

Алгоритм **частого роста паттернов** (**FP-рост**) является улучшением априорного алгоритма. Начинается с отображения FP-дерева частых транзакций, которое является упорядочённым деревом. Он состоит из двух этапов:

- Заполнение FP-дерева
- Частые схемы добычи

Давайте рассмотрим эти шаги по одному.

Заполнение FP-дерева

Рассмотрим данные транзакций, показанные в следующей таблице. Сначала представим это как разрежённую матрицу:

ID	Бита	Калитки	Накладки	Шлем	Мяч
А	0	1	1	0	0
Б	1	1	1	1	0
В	0	0	0	1	1
Г	1	0	1	1	0

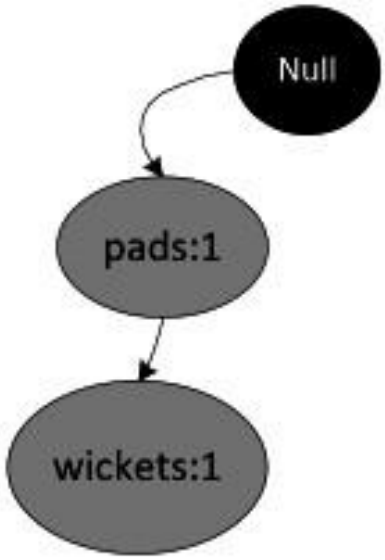
Давайте вычислим частоту каждого элемента и отсортируем их в убывающем порядке по частоте:

Предмет	Частота
Колодки	3
шлема 3 бита	2
калитки 2 мяча	1

Теперь давайте пересмотрим транзакционные данные в зависимости от частоты:

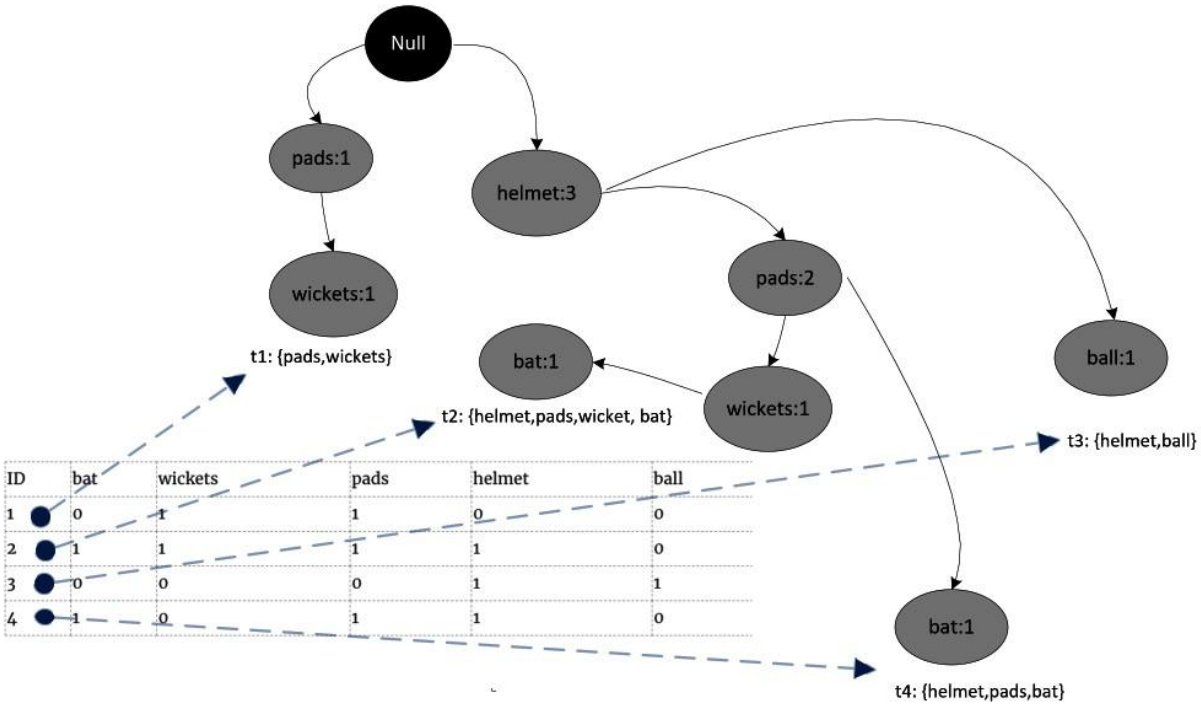
ID	Оригинальные предметы	Перезаказ
Предметы	T1 Калитки, накладки	Накладки, калитки
t2	Бита, калитки, накладки, шлем	Шлем, накладки, калитки, бита
t3	Шлем, мяч	Шлем, мяч
t4	Бита, накладки, шлем	Шлем, накладки, бита

Чтобы построить FP-дерево, начнём с первой ветви FP-дерева. FP-дерево начинается с **Null** в качестве корня. Чтобы построить дерево, можно представить каждый элемент узлом, как показано на следующей диаграмме (дерево t1 показано здесь). Обратите внимание, что метка каждого узла — это название элемента, а частота добавляется после двоеточия. Также обратите внимание, что элемент **с прокладками** 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

Используя тот же шаблон, нарисуем все четыре транзакции, в результате чего получится полное FP-дерево. FP-дерево состоит из четырёх листовых узлов, каждый из которых представляет набор элементов, связанных с четырьмя транзакциями. Обратите внимание, что нужно считать частоты каждого элемента и увеличивать её при многократном использовании — например, при добавлении t2 в FPtree частота **шлема** увеличивалась до двух. Аналогично, при добавлении t4, его ещё раз увеличили до трёх. Полученное дерево показано на следующей диаграмме:



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

Обратите внимание, что FP-дерево, сгенерированное на предыдущей диаграмме, является упорядочённым деревом.

Частые схемы добычи

Вторая фаза FP-роста включает извлечение частых паттернов из FP-дерева. Создавая упорядочённое дерево, цель — создать эффективную структуру данных, по которой можно легко ориентироваться для поиска частых шаблонов. Мы начинаем с листового узла (то есть конечного узла) и двигаемся вверх — например, начнём с одного из элементов листового узла, **летучей мыши**. Затем нужно вычислить условную базу паттерна для **летучей мыши**. База условного шаблона вычисляется путём указания всех путей от узла элемента листа до верхней части. Условная основа паттерна для **bat** будет следующей:

Калитка: 1 Накладки: 1 Шлем: 1
Колодка: 1 Шлем: 1

Частый паттерн для **бэта** будет следующим:

{калитка, накладки, шлем} : бита {накладка, шлем} : бита

Код для использования FP-роста

Давайте посмотрим, как можно сгенерировать правила ассоциаций с помощью алгоритма FP-growth в Python. Для этого мы будем использовать `пакет pyfpgrowth`. Во-первых, если мы никогда раньше не пользовались `pyfpgrowth`, давайте сначала установим его:

```
!pip install pyfpgrowth
```

Затем импортируем пакеты, которые нам нужны для реализации этого алгоритма:

```
Импорт Pandas как PD
импорт numpy как NP
импорт pyfpgrowth как FP
```

Теперь мы создадим входные данные в виде `transactionSet` :

```
dict1 = { 'id':[0,1,2,3],
 'предметы':[["калитки", "колодки"],
 [«бита», «калитки», «накладки», «шлем»],
 ["шлем", "пад"],
 [«летучая мышь», «накладки», «шлем»]]
} transactionSet = pd. DataFrame
(dict1)
```

После генерации входных данных мы сгенерируем шаблоны, основанные на параметрах, которые мы передали в `find_frequent_patterns()` . Обратите внимание, что второй параметр, передаваемый этой функции, — это минимальная поддержка, которая в данном случае равна 1:

```
паттерны = fp.find_frequent_patterns(transactionSet['items'],1)
```

Шаблоны были сгенерированы. Теперь распечатаем узоры. В выкройках перечислены комбинации предметов с их опорами:

```
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}
```

Теперь давайте сгенерируем правила:

```
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)}
```

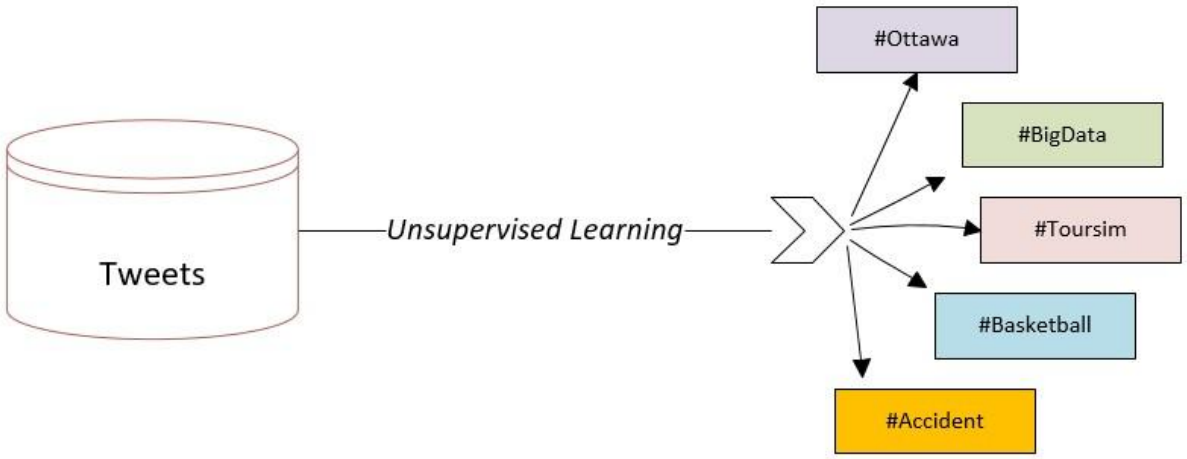
Каждое правило имеет левую и правую стороны, разделённые двоеточием (:). Он также даёт нам поддержку каждого из правил в нашем входном наборе данных.

Практическое применение — объединение похожих твитов вместе.

Алгоритмы неконтролируемого машинного обучения также могут применяться в реальном времени для объединения похожих твитов. Они делают следующее:

- ШАГ 1 — **Моделирование тем**: Откройте для себя различные темы из заданного набора твитов
- ШАГ 2 — **Кластеризация:** Ассоциируйте каждый твит с одной из обнаруженных тем

Это использование неконтролируемого обучения показано на следующей диаграмме:



Обратите внимание, что в этом примере требуется обработка входных данных в реальном времени. Давайте рассмотрим эти шаги по очереди.

Тематическое моделирование

Тематическое моделирование — это процесс обнаружения концепций в наборе документов, которые можно использовать для их различия. В контексте твитов речь идёт о поиске наиболее подходящих тем, на которые можно разделить набор твитов. Латентное распределение Дирихле — популярный алгоритм, используемый для моделирования тем. Поскольку каждый из твитов — это короткий документ из 144 символов, обычно посвящённый очень конкретной теме, мы можем написать более простой алгоритм для целей моделирования тем. Алгоритм описывается следующим образом:

А. Токенизируйте твиты.
Б. Предварительно обработайте данные. Уберите стопворды, цифры, символы и выполните стемминг
В. Создайте матрицу термина-документа (TDM) для твитов. Выберите топ-200 слов, которые чаще всего встречаются в уникальных твитах.
Г. Выберите топ-10 слов, которые прямо или косвенно отражают концепцию или тему. Например, мода, Нью-Йорк, программирование, случайность. Эти 10 слов теперь являются темами, которые мы успешно обнаружили и которые станут центрами кластера для твитов.

Перейдём к следующему этапу — кластеризации

Кластеризация

Как только мы откроем темы, мы выберем их в центр кластера. Затем можно запустить алгоритм кластеризации k-means, который назначит каждый твит одному из центров кластера. Итак, это пример того, как набор твитов можно сгруппировать в изученные темы.

Алгоритмы обнаружения аномалий

Словарное определение *аномалии* — это нечто, что отличается, аномально, необычное или трудно классифицировать. Это отклонение от общего правила. В контексте науки о данных аномалия — это точка данных, которая сильно отклоняется от ожидаемой закономерности. Методы обнаружения таких точек данных называются методами обнаружения аномалий. Теперь давайте рассмотрим некоторые применения алгоритмов обнаружения аномалий:

- Мошенничество с кредитными картами
- Обнаружение злокачественной опухоли на **магнитно-резонансной томографии (МРТ**)
- Предотвращение неисправностей в кластерах
- Имитация на экзаменах

Аварии на шоссе

В следующих разделах мы увидим различные методы обнаружения аномалий.

Использование кластеризации

Алгоритмы кластеризации, такие как k-средние, могут использоваться для группировки похожих точек данных. Можно определить порог, и любая точка за его пределами может быть классифицирована как аномалия. Проблема такого подхода в том, что группировка, создаваемая кластеризацией k-средних,

может быть предвзятой из-за наличия аномальных точек данных и влиять на полезность и точность подхода.

Использование обнаружения аномалий на основе плотности

Подход, основанный на плотности, пытается найти плотные окрестности. Для этой цели может использоваться **алгоритм k-ближайших** соседей (**KNN).** Аномалии, находящиеся далеко от обнаруженных густонаселенных районов, отмечаются как аномалии.

Использование опорных векторных машин

Алгоритм **Support Vector Machine** (**SVM**) может использоваться для изучения границ точек данных. Любые точки за пределами этих открытых границ определяются как аномалии.

Краткое содержание

В этой главе мы рассмотрели различные неконтролируемые методы машинного обучения. Мы рассмотрели обстоятельства, при которых разумно попытаться уменьшить размерность задачи, которую пытаемся решить, и различные методы её выполнения. Мы также изучили практические примеры, где методы неконтролируемого машинного обучения могут быть очень полезны, включая анализ рыночных корзин и обнаружение аномалий. В следующей главе мы рассмотрим различные методы обучения под контролем. Начнём с линейной регрессии, а затем рассмотрим более сложные методы контролируемого машинного обучения, такие как алгоритмы на основе дерева решений, SVM и XGBoast. Мы также изучим наивный алгоритм Байеса, который лучше всего подходит для неструктурированных текстовых данных.

OceanfPDF.com

7 Традиционные алгоритмы контролируемого обучения

Присоединяйтесь к Discord-пространству нашей книги

https:// packt.link/40Algos



«Искусственный интеллект — это новая электричество.» ~Эндрю Нг.

В этой главе мы сосредоточимся на алгоритмах контролируемого машинного обучения, которые являются одним из важнейших типов современных алгоритмов. Отличительной чертой алгоритма контролируемого машинного обучения является использование маркированных данных для обучения модели. В этой главе мы представим все алгоритмы контролируемого машинного обучения, за исключением нейронных сетей. Следующие три главы посвящены реализации алгоритмов контролируемого машинного обучения с использованием нейронных сетей. Правда в том, что при таком постоянном развитии в этой области нейронные сети — это комплексная тема, для которой требуется более одной главы для полноценного обсуждения... В этой главе мы сначала познакомимся с фундаментальными понятиями контролируемого машинного обучения. Далее мы представим два типа моделей контролируемых машин — классификаторы и регрессоры. Чтобы продемонстрировать способности классификаторов, сначала мы представим реальную задачу в качестве вызова. Затем мы представим шесть различных алгоритмов классификации, которые используются для решения одной и той же задачи. Далее мы сосредоточимся на алгоритмах регрессии, сначала представив аналогичную задачу, которую нужно решить для регрессоров. Затем представим три алгоритма регрессии и используем их для решения одной и той же задачи. Наконец, мы сравним результаты, чтобы подытожить концепции, представленные в этой главе. Общая цель этой главы — понять различные типы техники контролируемого машинного обучения и понять, какие методы контролируемого машинного обучения являются лучшими для определённых классов задач. В этой главе обсуждаются следующие концепции:

- Понимание контролируемого машинного обучения
- Понимание алгоритмов классификации
- Методы оценки производительности классификаторов
- Понимание алгоритмов регрессии

Методы оценки эффективности регрессионных алгоритмов

Давайте начнём с рассмотрения основных концепций контролируемого машинного обучения.

Понимание контролируемого машинного обучения

Машинное обучение сосредоточено на использовании подходов, основанных на данных, для создания автономных систем, которые помогают нам принимать решения с человеческим контролем или без него.

Для создания таких автономных систем машинное обучение использует набор алгоритмов и методологий для обнаружения и формулирования повторяемых закономерностей в данных. Одной из самых популярных и мощных методологий в машинном обучении является подход с контролируемым машинным обучением. В контролируемом машинном обучении алгоритму задаётся набор входных данных, называемых признаками, и соответствующими им выходами, **называемыми метками**. Используя данный набор данных, алгоритм контролируемого машинного обучения обучает модель, отражающую сложную связь между признаками и метками, представленными математической формулой. Эта обученная модель является базовым транспортным средством, используемым для прогнозирования.*Способность учиться на существующих данных в контролируемом обучении схожа со способностью человеческого мозга учиться на опыте. Эта способность к обучению в контролируемом обучении использует одну из характеристик человеческого мозга и является фундаментальным способом открытия дверей, чтобы передать машинам силу и интеллект для принятия решений.*Рассмотрим пример, когда мы хотим использовать техники контролируемого машинного обучения для обучения модели, которая может классифицировать набор писем на легитимные (называемые легитимными) и нежелательные (**называемые спамом**). Во-первых, чтобы начать, нам нужны примеры из прошлого, чтобы машина могла понять, какой тип писем следует классифицировать как спам. Эта задача обучения текстовых данных на основе содержания является сложным процессом и реализуется с помощью одного из контролируемых алгоритмов машинного обучения. Некоторые примеры алгоритмов контролируемого машинного обучения, которые можно использовать для обучения модели в этом примере, включают деревья принятия решений и наивные классификаторы Байеса, о которых мы рассмотрим позже в этой главе.

Формулирование задач с контролируемым машинным обучением

Прежде чем углубиться в детали алгоритмов контролируемого машинного обучения, давайте определим некоторые базовые термины в области контролируемого машинного обучения:

Терминология	**Объяснение**
Лейбл	Метка — это переменная, которую мы хотим, чтобы наша модель предсказывала, называется меткой. В модели контролируемого машинного обучения может быть только одна метка.
Особенности	Набор входных переменных, используемых для предсказания метки, называется признаками.
Особенности машинного обучения	Преобразование функций для подготовки к выбранному алгоритму контролируемого — это инженерия называется инженерией особенностей.
Вектор признаков	Перед передачей входных данных алгоритму контролируемого машинного обучения все признаки объединяются в структуру данных, называемую вектором признаков.
Исторические данные	Данные из прошлого, используемые для формулировки связи между меткой и признаками, называются историческими данными. Исторические данные приводят примеры.
Обучение/тестирование данных, называемый обучающими данными	Исторические данные с примерами делятся на две части — больший набор и меньший набор данных, называемый тестовыми данными.
Модель	Математическая формулировка паттернов, которая лучше всего отражает связь между меткой и признаками.
Обучение	Создание модели на основе обучающих данных.

Тестирование Оценка качества обученной модели с использованием тестовых данных. Прогноз Использование обученной модели для прогнозирования метки

Обученная модель с контролируемым машинным обучением способна делать прогнозы, оценивая метку на основе признаков. Давайте представим обозначения, которые мы будем использовать в этой главе для обсуждения методов машинного обучения:

Значение переменных

y Реальный лейбл *ý*
Прогнозируемая метка *d*
Общее количество примеров *b* Количество обучающих примеров *c* Количество примеров тестирования

Теперь давайте посмотрим, как некоторые из этих терминологий формулируются на практике. Как мы обсуждали, вектор признаков определяется как структура данных, в которой хранятся все признаки. Если количество признаков равно *n*, а количество обучающих примеров — *b*, то `X_train` представляет собой вектор обучающих признаков. Для обучающего набора данных вектор признаков представлен X_train . Если *в обучающем наборе* данных есть примеры b, `то X_train` будут *строки b*. Если есть *n* переменных, то обучающий набор данных будет иметь размерность *n* x *b. Используем верхний индекс для обозначения номера строки обучающего примера. Это конкретный пример в нашем помеченном наборе представлен как (Features*$^{(1)}$,label$^{(1)}$) = ($X^{(1)}$, $y^{(1)}$). Итак, наш помеченный набор данных представлен D = {$X^{(1)}$,$y^{(1)}$), ($X^{(2)}$,$y^{(2)}$), ... ,
($X^{(d)}$,$y^{(d)}$)}. Мы делим это на две части — Dtrain и$_{D\text{-тест}}$. Таким образом, наш набор обучения можно представить Dtrain = {$X^{(1)}$,$y^{(1)}$), ($X^{(2)}$,$y^{(2)}$), ... , ($X^{(b)}$,$y^{(b)}$)}. Цель обучения модели заключается в том, чтобы для любого *i-го* примера в обучающем наборе предсказанное значение целевого значения было максимально близко к фактическому значению в примерах. Другими словами,

$$\acute{y}(i) \approx y(i)\,;\; for\; 1 \leq i \leq b$$

Таким образом, наш набор тестирования может быть представлен Dtest = {$X^{(1)}$,$y^{(1)}$), ($X^{(2)}$,$y^{(2)}$), ... , ($X^{(c)}$,$y^{(c)}$))}.

Значения метки представлены вектором: *Y*:Y ={ $y^{(1)}$, $y^{(2)}$, ..., $y^{(m)}$}

Понимание условий, способствующих

Контролируемое машинное обучение основано на способности алгоритма обучать модель с помощью примеров. Алгоритм контролируемого машинного обучения требует выполнения определённых условий для выполнения работы. Эти условия следующие:

- **Достаточно** примеров: алгоритмы контролируемого машинного обучения требуют достаточного
- количества примеров для обучения модели. **Закономерности в исторических** данных: примеры, используемые для обучения модели, должны содержать закономерности. Вероятность возникновения нашего интересующего события должна зависеть от сочетания закономерностей, тенденций и событий. Без них мы имеем дело с случайными данными, которые нельзя использовать для обучения модели.

- **Допустимые предположения**: Когда мы обучаем модель контролируемого машинного обучения на примерах, мы ожидаем, что предположения, применимые к примерам, будут также актуальны в будущем. Давайте рассмотрим реальный пример. Если мы хотим обучить модель машинного обучения для правительства, которая сможет предсказывать вероятность выдачи визы студенту, понимается, что законы и политики не изменятся при использовании модели для прогнозов. Если после обучения модели применяются новые политики или законы, модель может потребоваться переобучения для включения этой новой информации.

Различие между классификаторами и регрессорами

В модели машинного обучения метка может быть переменной категории или непрерывной переменной. Тип метки определяет, какой тип модели контролируемого машинного обучения у нас есть. По сути, у нас есть два типа моделей контролируемого машинного обучения:

- **Классификаторы**: если метка является переменной категории, модель машинного обучения называется классификатором. Классификаторы могут использоваться для ответа на следующие бизнес-вопросы:
  - Является ли этот аномальный рост ткани злокачественной опухолью?
  - Учитывая текущие погодные условия, будет ли завтра дождь?
  - Исходя из профиля конкретного заявителя, следует ли одобрить его ипотечную заявку?
- **Регрессоры**: если метка — непрерывная переменная, мы обучаем регрессора. Регрессоры можно использовать для ответа на следующие типы бизнес-вопросов:
  - Учитывая текущие погодные условия, сколько дождя будет завтра?
  - Какова будет цена конкретного дома с определёнными характеристиками?

Давайте подробнее рассмотрим как классификаторы, так и регрессоры.

Понимание алгоритмов классификации

В контролируемом машинном обучении, если метка является переменной категории, модель классифицируется как классификатор:

- Исторические данные называются **маркированными данными**.
- Производственные данные, для которых необходимо предсказать маркировку, называются **немаркированными данными**.

*Возможность точно маркировать немаркированные данные с помощью обученной модели — это настоящая мощь алгоритмов классификации. Классификаторы предсказывают метки для немаркированных данных, чтобы ответить на конкретный бизнес-вопрос.*Прежде чем представить детали алгоритмов классификации, давайте сначала представим бизнес-задачу, которую мы используем в качестве вызова для классификаторов. Затем мы используем шесть различных алгоритмов для решения одной и той же задачи, что поможет сравнить их методологию, подход и эффективность.

Презентация вызова классификаторов

Сначала мы представим общую задачу, которую используем в качестве вызова для тестирования шести различных алгоритмов классификации. Эта распространённая проблема в этой главе называется вызовом классификаторов. Использование всех шести классификаторов для решения одной задачи поможет нам двумя способами:

- Все входные переменные должны быть обработаны и собраны в сложную структуру данных, называемую вектором признаков. Использование одного и того же вектора признаков помогает
- избежать повторяющихся подготовок данных для всех шести алгоритмов. Мы можем точно сравнивать производительность различных алгоритмов, так как используем один и тот же вектор признаков для входа.

Задание классификаторов направлено на предсказание вероятности того, что человек совершит покупку. В розничной торговле одним из способов максимизации продаж является лучшее понимание поведения клиентов. Это можно сделать, анализируя закономерности, обнаруженные в исторических данных. Давайте сначала изложим проблему.

Формулировка проблемы

Учитывая исторические данные, можем ли мы обучить бинарный классификатор, который сможет предсказывать, купит ли конкретный пользователь продукт на основе своего профиля? Сначала рассмотрим исторически маркированный набор данных, доступный для решения этой задачи:x € Rb, y € {0,1}Для конкретного примера, когда *y* = 1, мы называем это положительным классом, а y = 0 — отрицательным классом.*Хотя уровень положительного и отрицательного классов можно выбрать произвольно, хорошей практикой является определение положительного класса как интересующее событие. Если мы пытаемся отметить мошенническую транзакцию для банка, то положительный класс (то есть y = 1) должен быть мошеннической операцией, а не наоборот.*Теперь давайте рассмотрим следующее:

- Фактическая метка, обозначаемая как *y*
- Предсказанная метка, обозначаемая как *y'*

Обратите внимание, что для нашей задачи классификаторов фактическое значение метки, найденной в примерах, представлено как *y*. Если в нашем примере кто-то купил товар, мы говорим y =1. Прогнозируемые значения представлены как *y'*. Вектор входных признаков, *x*, имеет размерность 4. Мы хотим определить, какова вероятность того, что пользователь совершит покупку, учитывая конкретный вход. Итак, мы хотим определить вероятность того, что *y* = 1 имеет определённое значение вектора признаков *x.* Математически это можно представить следующим образом:

$$\acute{y} = P(y = 1|x) : where; x \in \Re^{n_x}$$

Теперь давайте посмотрим, как мы можем обрабатывать и собирать различные входные переменные в векторе признаков x. Методология сборки различных частей *x* с помощью процессорного конвейера подробно рассматривается в следующем разделе.

Инжиниринг признаков с использованием конвейера обработки данных

Подготовка данных для выбранного алгоритма машинного обучения называется **инженерией признаков** и является важной частью жизненного цикла машинного обучения. Разработка функций выполняется на разных этапах или этапах. Многоступенчатый код обработки, используемый для обработки данных, в совокупности называется **конвейером данных**. Создание конвейера данных с использованием стандартных этапов обработки, где это возможно, делает его повторно используемым и снижает затраты на обучение моделей. Использование более проверенных программных модулей также повышает качество кода. Давайте посмотрим, как спроектировать многоразовый процессорный конвейер для задачи

классификаторов. Как уже упоминалось, мы подготовим данные один раз, а затем используем их для всех классификаторов.

Импорт данных

Исторические данные этой задачи, содержащие примеры, хранятся в файле под названием `dataset` в формате `.csv`. Мы используем функцию `pd.read_csv` из pandas для импорта данных в виде кадра данных:

```
dataset = pd.read_csv('Social_Network_Ads.csv')
```

Выбор функций

Процесс выбора признаков, релевантных контексту задачи, которую мы хотим решить, называется **выбором признаков**. Это неотъемлемая часть инженерии функций. После импорта файла мы убираем столбец `User ID`, который используется для идентификации человека и должен быть исключён при обучении модели: `dataset = dataset.drop(columns=['User ID'])`

Теперь давайте рассмотрим набор данных:

```
dataset.head(5)
```

Набор данных выглядит так:

	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

Теперь давайте посмотрим, как мы можем дальше обработать входной набор данных.

Однофазное кодирование

Многие алгоритмы машинного обучения требуют, чтобы все признаки были непрерывными переменными. Это значит, что если некоторые признаки являются переменными категорий, нам нужно

найти стратегию их преобразования в непрерывные переменные. Однофазное кодирование — один из самых эффективных способов выполнения этого преобразования. Для этой конкретной задачи единственной переменной категории является `гендер` . Давайте преобразуем это в непрерывную переменную с помощью однофазного кодирования:

```
enc = sklearn.preprocessing.OneHotEncoder()
enc.fit(dataset.iloc[:,[0]])
onehotlabels = enc.transform(dataset.iloc[:,[0]]).toarray()
Гендеры = PD. DataFrame({'Female': onehotlabels[:, 0], 'Male': onehotlabels[:, 1]})
result = pd.concat([genders,dataset.iloc[:,1:]], axis=1, sort=False) result.head(5)
```

После конвертации давайте снова посмотрим на набор данных:

	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

Обратите внимание, что для преобразования переменной из категории в непрерывную переменную одногорячая кодировка преобразовала `гендер` в два отдельных столбца — `Мужской и Женский` .

Указание характеристик и маркировки

Давайте уточним характеристики и ярлыки. В этой книге мы будем использовать `y` для обозначения метки, а `X` — для обозначения набора признаков:

```
y=результат['Куплено']
X=result.drop(columns=['Куплено'])
```

`X` представляет вектор признаков и содержит все входные переменные, необходимые для обучения модели.

Разделение набора данных на части тестирования и обучения

Теперь давайте разделим обучающий набор данных на 25% тестирование и 75% обучающих частей с использованием `sklearn.model_selection импорта train_test_split` :

```
#from sklearn.cross_validation импортировать train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0,25, random_state = 0)
```

Это создало следующие четыре структуры данных:

- `X_train` : Структура данных, содержащая особенности обучающих данных
- `X_test` : Структура данных, содержащая признаки обучающего `теста`
- `y_train` : Вектор, содержащий значения метки в обучающем наборе
- `данных y_test` : Вектор, содержащий значения метки в тестовом наборе данных

Масштабирование характеристик

Для многих алгоритмов машинного обучения хорошая практика — масштабировать переменные от `0` до `1` . Это также называется **нормализацией признаков**. Давайте применим масштабируемое преобразование, чтобы добиться этого:

```
от sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train) X_test
= sc.transform(X_test)
```

После масштабирования данных они готовы к использованию в качестве входных данных для различных классификаторов, которые мы представим в следующих разделах.

Оценка классификаторов

После обучения модели нужно оценить её производительность. Для этого мы используем следующий процесс:

А. Мы разделим набор данных маркировки на две части — обучающий раздел и тестовый. Мы будем использовать раздел тестирования для оценки обученной модели.
Б. Мы будем использовать функции нашего тестового раздела для генерации меток для каждой строки. Это наш набор предсказанных меток.
В. Мы сравним набор прогнозируемых меток с реальными метками, чтобы оценить модель.

*Если мы не пытаемся решить что-то довольно тривиальное, при оценке модели будут возникать ошибки. То, как мы интерпретируем эти неправильные классификации для определения качества модели, зависит от того, какие метрики эффективности мы выбираем.*Когда у нас есть как набор фактических меток, так и прогнозируемых меток, для оценки моделей можно использовать множество метрик производительности. Лучшая метрика для количественной оценки модели будет зависеть от требований бизнес-задачи, которую мы хотим решить, а также от характеристик обучающего набора данных.

Матрица путаницы

Матрица путаницы используется для суммирования результатов оценки классификатора. Матрица путаницы для бинарного классификатора выглядит следующим образом:

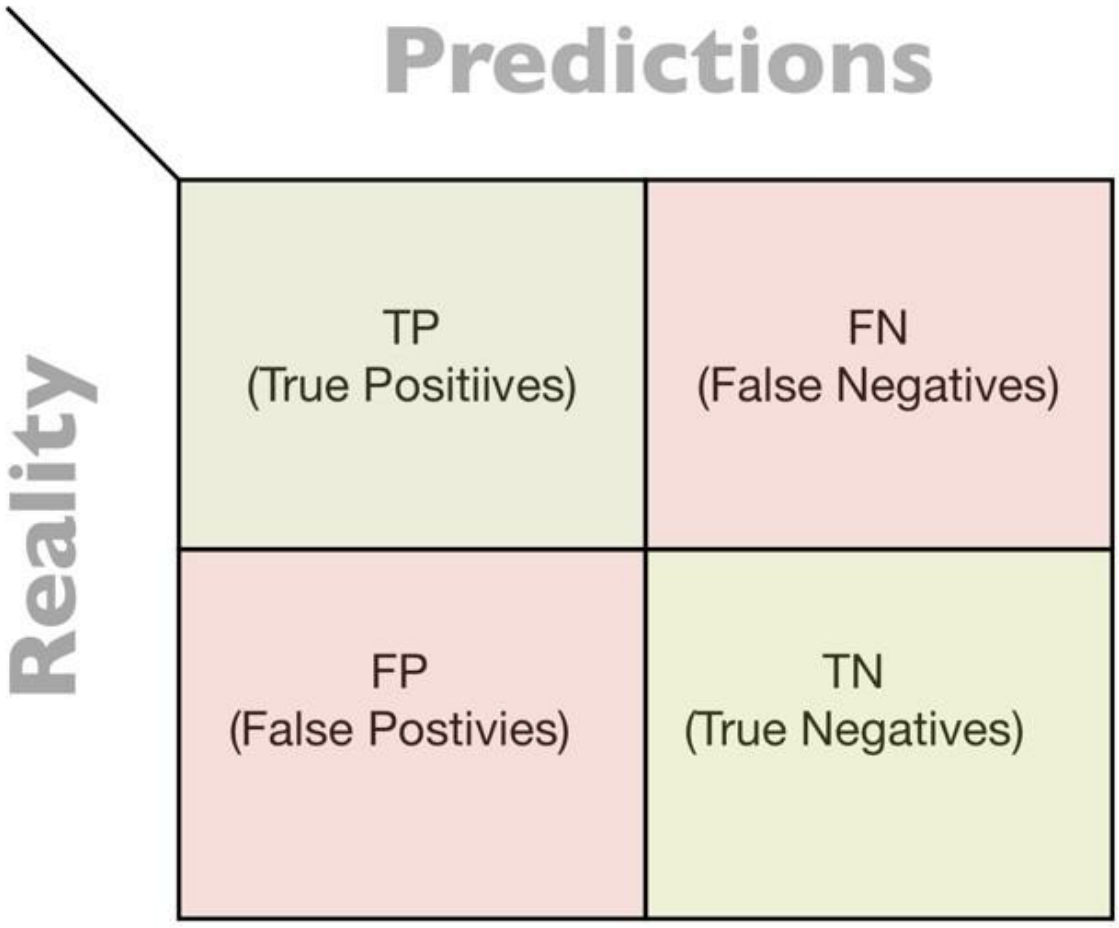


Рисунок 7.1: Матрица путаницы

Если метка классификатора, который мы обучаем, имеет два уровня, она называется **бинарным классификатором***. Первым критическим примером использования контролируемого машинного обучения — в частности, бинарного классификатора — было во время Первой мировой войны для различия между самолётом и летающими птицами.*Классификацию можно разделить на следующие четыре категории:

- **Истинные** положительные результаты (**TP**): положительные классификации, которые были правильно
- классифицированы
- **Истинные** отрицательные оценки (**TN):** Отрицательные классификации, которые были правильно
- классифицированы

Ложноположительные результаты (**FP**): положительные классификации, которые на самом деле были отрицательными

Ложноотрицательные результаты (**FN**): отрицательные классификации, которые действительно были положительными

Давайте посмотрим, как мы можем использовать эти четыре категории для создания различных метрик производительности. Матрица путаницы даёт вам количество правильных классификаций и ошибок на основе подсчёта TP, TN, FP и FN. Точность модели определяется как доля корректировочных классификаций среди всех предсказаний и легко может быть получена из матрицы путаницы следующим

образом: Если наши данные имеют сбалансированные классы (то есть общее количество положительных примеров примерно равно числу отрицательных примеров), то точность даст нам хорошее представление о Качество нашей обученной модели. Точность — это доля классификаций коррекций среди всех предсказаний.

$$Accuracy = \frac{Correct\ Classifications}{Total\ number\ of\ Classifications} = \frac{TP+FP}{TP+FP+FN+TN}$$

Понимание воспоминания и точности

При расчёте точности мы не различаем TP и TN. Оценка модели по точности проста, но когда данные имеют дисбалансные классы, качество обученной модели не будет точно оценить. Когда данные имеют дисбалансные классы, две дополнительные метрики лучше количественно оценивают качество обученной модели, воспоминание и точность. Мы приведём пример популярного процесса добычи алмазов, чтобы объяснить концепции этих двух дополнительных метрик. На протяжении веков добыча аллювиальных алмазов была одним из самых популярных способов добычи алмаза из песков русел рек по всему миру. Известно, что тысячелетняя эрозия смывала алмазы из их первичных месторождений в речные русла в разных частях мира. Для добычи алмазов люди собирают пески с берега реки в большой открытый карьер. После тщательной промывки в яме остаётся много камней. Подавляющее большинство этих промытых камней — это просто обычные камни. Идентификация одной из пород как алмаза — редкое, но очень важное событие. Владельцы шахты экспериментируют с помощью компьютерного зрения, чтобы определить, какие из промытых камней являются обычными камнями, а какие — алмазами. Они используют форму, цвет и отражение для классификации промытых камней с помощью компьютерного зрения. В контексте этого примера:

TP Промытый камень, правильно идентифицированный как алмаз

TN Вымытый камень, правильно идентифицированный как камень

FP Камень, ошибочно идентифицированный как алмаз

FN Алмаз, ошибочно идентифицированный как камень

Давайте объясним, что такое отзыв и точность, учитывая этот процесс добычи алмаза из шахты.

- **Вспомнить**: Это рассчитывает *процент попаданий* — долю выявленных событий, представляющих интерес, в огромном хранилище событий. Другими словами, эта метрика оценивает нашу способность находить или «попадать» в большинство интересующих событий, оставляя как можно меньше неидентифицированными. В контексте выявления алмазных ям, содержащих большое количество промытых камней, «воспоминание» — это количественная оценка успеха «охоты за сокровищами». Для определённой ямы, заполненной промытыми камнями, будет использоваться соотношение количества найденных алмазов к общему числу алмазов в карьере.



$$Recall = \frac{No.\ of\ correctly\ identified\ diamonds}{Total\ number\ of\ diamonds\ in\ the\ pit} \frac{TP}{TP + FN}$$

- Предположим, что в яме было 10 бриллиантов, каждый стоимостью $1000. Наш алгоритм машинного обучения смог выявить 9 из них. Таким образом, отзыв будет 9/10 = 0,90

Таким образом, мы смогли вернуть 90% наших сокровищ. По долларовой стоимости нам удалось выделить $9,000 сокровищ из общей стоимости $10,000

- **Точность**: В точности мы сосредотачиваемся только на тех точках данных, которые обученная модель отмечает как положительные, и отбрасывает всё остальное. Если мы отфильтруем только те события, отмеченные как положительные в нашей обученной модели (то есть TP и FP), то вычислим точность, это называется точностью.

Теперь давайте рассмотрим точность в контексте примера с добычей алмазов. Рассмотрим ситуацию, когда мы хотим с помощью компьютерного зрения определить алмазы среди ямы с вымытыми камнями и отправить их клиентам. Процесс должен быть автоматизирован. Абсолютно нежелательное событие — это когда алгоритм ошибочно классифицирует камень как алмаз, из-за чего конечный клиент получает его по почте и получает за него плату. Поэтому должно быть очевидно, что для осуществления этого процесса точность должна быть высокой. Для примера с добычей алмазов:



$$precision = \frac{No.\ of\ correctly\ identifed\ diamonds}{Total\ number\ of\ rocks\ flagged\ as\ diamonds} = \frac{TP}{TP + FP}$$

Понимание компромисса между отзывом и точностью

Чтобы понять врождённый компромисс, связанный с точностью и запоминанием, важно сначала понять, как классификатор принимает решение. Придумать решение для каждой точки данных, чтобы обозначить её одним из двух классов, представляет собой двухэтапный процесс:ШАГ 1: Классификатор генерирует оценку решения, которая представляет собой число от 0 до 1.ШАГ 2: Классификатор использует значение параметра, называемого порогом решения, чтобы назначить один из двух классов текущей точке данных. Любой порог принятия решения (балл > решение) прогнозируется как положительный, а любой точку данных с порогом решения (балл < решение) прогнозируется как отрицательный. Чтобы объяснить компромисс между точностью и отзывом, мы вернёмся к примеру добычи алмазов. На рисунке 7.x показаны несколько камней, организованных в порядке восходящего результата по ШАГУ 1 алгоритма.

Обратите внимание, что камни слева имеют самый низкий балл и наименее вероятно являются алмазами. Камни справа имеют наивысший балл справа и, скорее всего, представляют собой алмаз. Для совершенного классификатора камни справа от порога принятия решения должны быть бриллиантами, а камни на левой стороне — камнями. Теперь сначала рассмотрим рисунок 7.x (a). Порог принятия решений находится в центре. Если посмотреть справа на границе решения (в заштраненном блоке), вы увидите 3 правильные классификации и одну ошибку. То есть, вы найдёте 3 TP и 1 FP. Слева от порога принятия решений вы найдёте 2 правильных классификации и 2 неправильных классификации. Это 2TN и 2FN. Рассчитаем воспоминание и точность для рисунка 7.x (a)

$$\frac{\square}{\square} = \frac{\square}{\square} = \frac{\square}{\square} = 0.5$$

$$\frac{\square\,\square\,\square}{\square\,\square\,\square} = 0.75$$

$$Recall = \frac{No.\ of\ correctly\ identified\ diamonds}{Total\ number\ of\ diamonds\ in\ the\ pit}\ \frac{TP}{TP+FN}\ \frac{3}{6}\ precision$$

$$= \frac{No.\ of\ correctly\ identifed\ diamonds}{Total\ number\ of\ rocks\ flagged\ as\ diamonds} = \frac{TP}{TP+FP}$$

$$= \frac{3}{4}$$

Из рисунка 7.x (b) ясно, что при повышении порога принятия решения до 0.8 мы делаем критерии для классификации камня гораздо строгими. Подразумевается, что точность увеличивается, но воспоминание снижается. На рисунке 7.x (c) мы уменьшили порог принятия решения. Другими словами, мы снизили критерии для классификации камня как алмаза. Таким образом, FN (сокровище промахивается) будет уменьшаться, но FP (ложный сигнал) тоже увеличивается. Итак, игра с ценностью границы принятия решения — это управление компромиссом между возвратом и точностью. Мы увеличиваем границу принятия решения, чтобы получить лучшую точность, и можем рассчитывать на большее количество отзывов. И мы снижаем границу решения, чтобы получить лучший отзыв, и можем рассчитывать на меньшую точность.

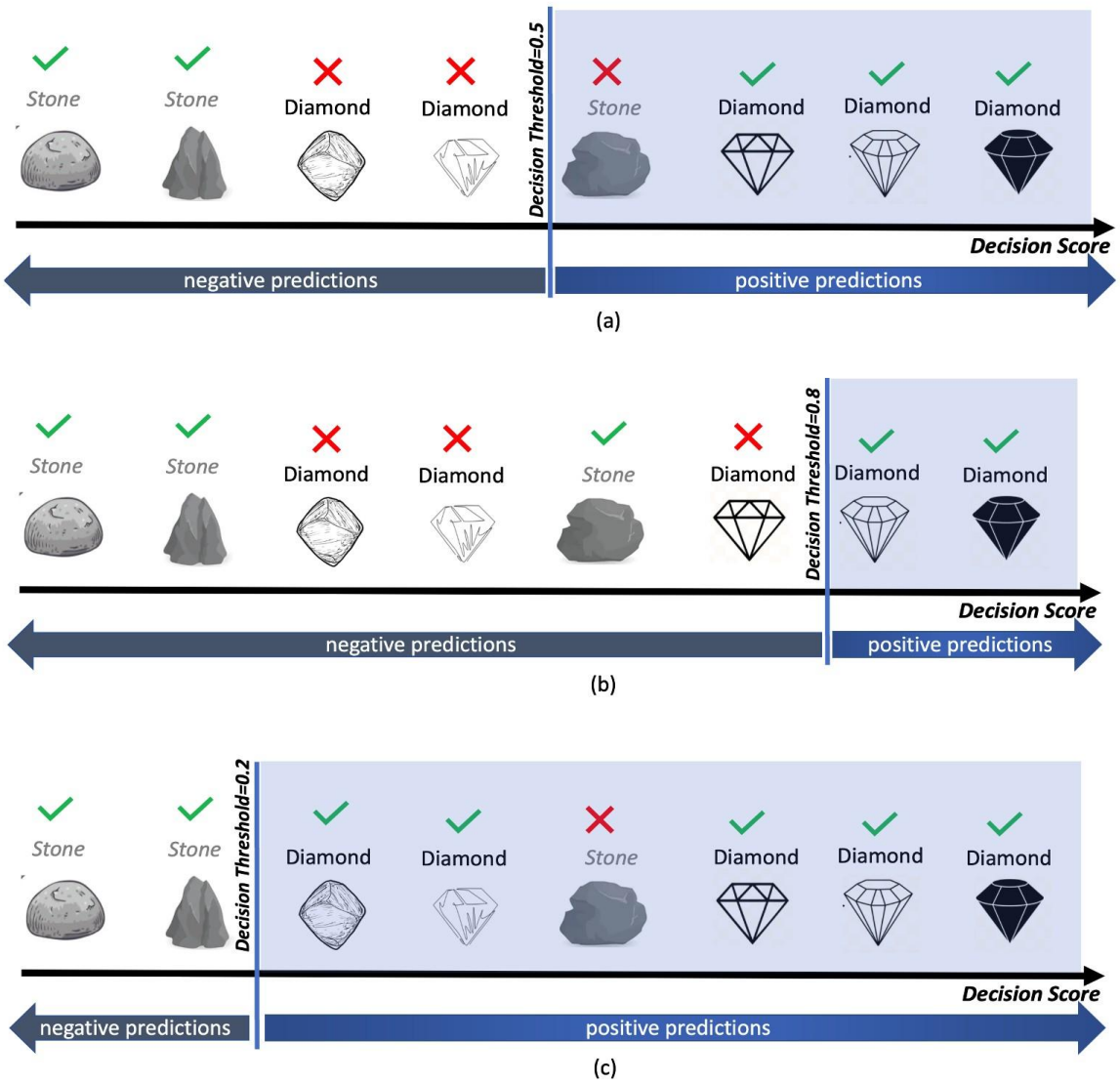


Рисунок 7.2: Компромисс между точностью и воспоминанием: камни ранжируются по оценке классификатора. Те, кто выше порога принятия решения, считаются алмазами. Обратите внимание, что чем выше порог, тем выше точность, но тем ниже — отсвояване,

Давайте проведём график между точностью и воспоминанием, чтобы лучше понять компромисс.

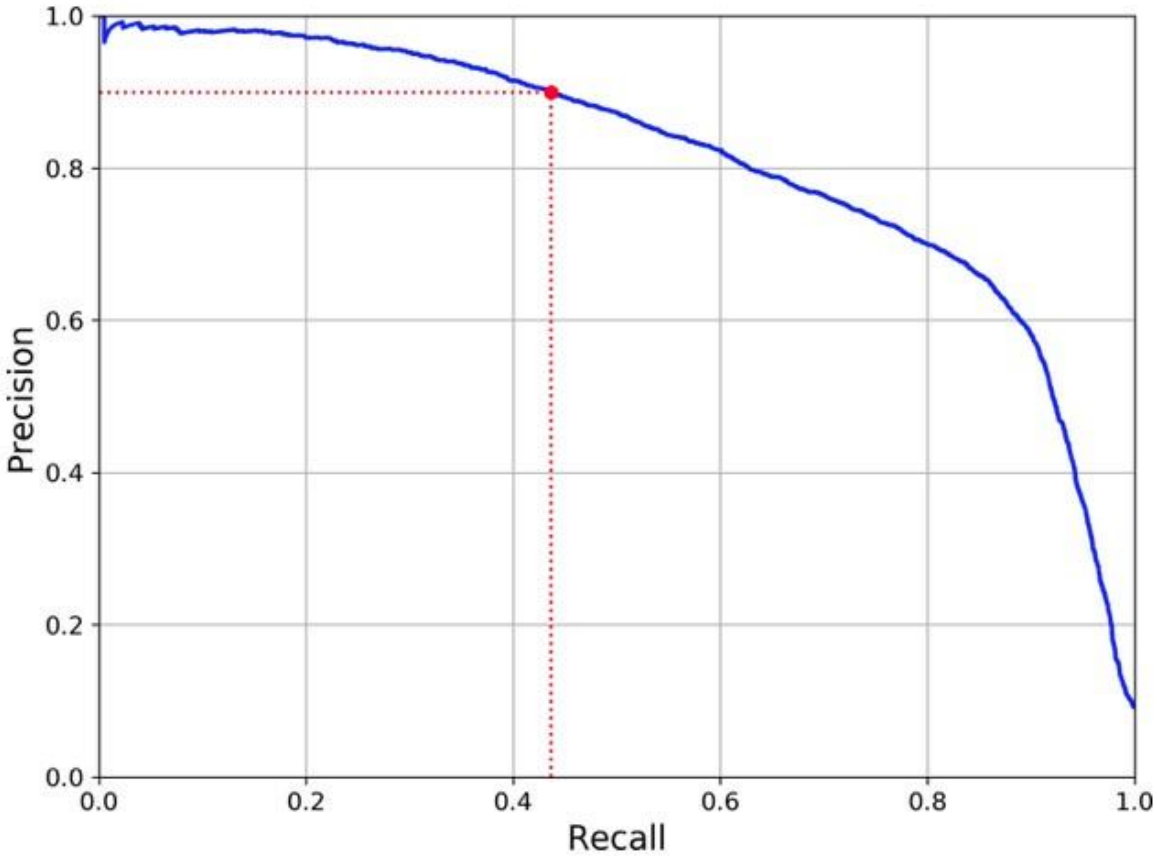


Рисунок 7.2: Точность против отзыва

Какой выбор является правильным для воспроизведения и точности? Увеличение запоминания достигается за счёт снижения критериев для определения точки данных как положительных. Ожидается, что точность снизится, но на рисунке выше она резко падает примерно на 0,8. Это момент, когда мы можем выбрать правильное значение воспоминания и точности. На приведённом выше графике, если выбрать 0,8 в качестве отзыва, точность равна 0,75. Мы можем интерпретировать это так, будто смогли отметить 80% всех интересных точек данных. Точность для пометки этих точек данных составляет 75%. Если нет конкретных бизнес-требований, для универсального случая использования это может быть разумным компромиссом. Другой способ показать внутренний компромисс между точностью и отзывом — использовать приёмную операционную кривую (ROC). Для этого определим два термина: истинно положительный коэффициент (TPR) и FPR (FPR). Давайте рассмотрим кривую ROC с помощью рисунка x. Чтобы рассчитать TPR и FPR, нам нужны алмазы в шахте.

$$TPR = \frac{\square}{\square} = \frac{\square}{\square}$$

$$\frac{\square}{\square} = \frac{\square}{\square}$$

$$\frac{\square}{\square} = \frac{\square}{\square} \frac{Truely\ identified\ diamonds}{All\ diamonds\ in\ the\ pit} \frac{TP}{TP+FN} TNR = \frac{Truely\ identified\ stones}{All\ stones\ in\ the\ pit} \frac{TN}{TN+FP} FPR = \frac{Falsely\ indentified\ diamonds}{All\ stones\ in\ the\ pit} \frac{FP}{TN+FP}$$

Обратите внимание:

А. TPR равен возвращению или скорости попаданий.
Б. TNR можно считать отзывом или частотой попадания негативного события. Он определяет наш успех в правильном выявлении негативного события. Её также называют специфичностью.
В. FPR = 1 – TNR = 1 - Специфичность

Очевидно, что TPR и FPR для этих показателей можно рассчитать следующим образом:

Рисунок номер	**TPR**	**FPR**
Xx (a)	3/5=0,6	1/3=0,33
Xxx(b)	2/5=0,4	0/3 = 0
Xxx(c)	5/5 = 1	1/3 = 0,33

Обратите внимание, что TPR или отзыв будут увеличены за счёт снижения порога принятия решения. В попытке получить большую часть алмазов из шахты мы снизим критерии для классификации промытого камня как алмаза. В результате всё больше камней ошибочно классифицируются как алмазы, что увеличивает FPR. Обратите внимание, что качественный алгоритм классификации должен уметь предоставлять оценку решения для каждого камня в карьере, которая примерно соответствует вероятности того, что камень является алмазом. Вывод такого алгоритма показан на рисунке xx. Алмазы должны быть справа, а камни — слева. На рисунке, когда мы снизили порог принятия решения с 0,8 до 0,2, ожидается значительно более высокий рост TRP, чем FPR. Фактически, резкий рост TRP с небольшим увеличением FPR — один из лучших показателей качества бинарного классификатора. Алгоритм классификации смог генерировать оценки решений, которые напрямую связаны с вероятностью того, что камень является алмазом. Если алмазы и камни расположены случайным образом на оси очков решения, с равной вероятностью снижение порога решения приведёт к флагу камней или бриллиантов. Это был бы худший возможный бинарный классификатор, также называемый рандомизатором.

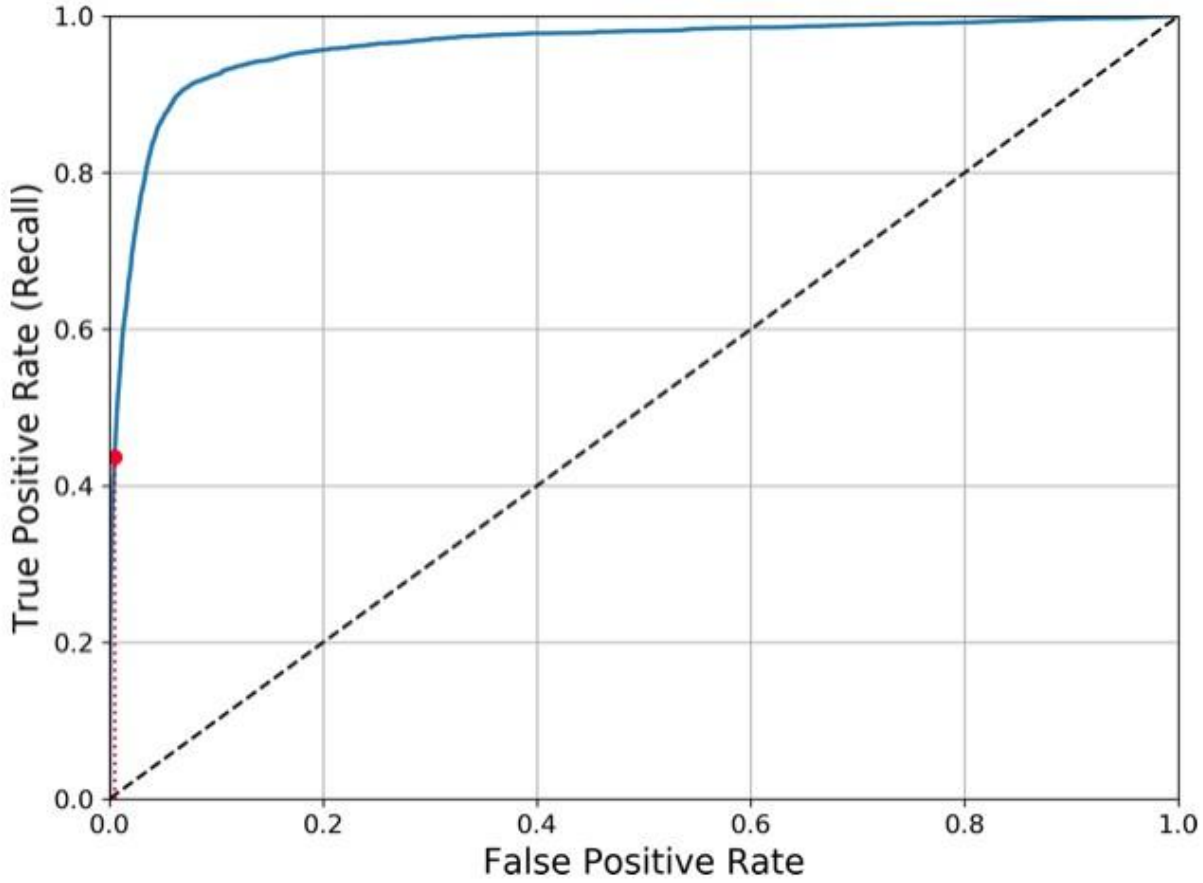


Рисунок 7.3: Кривая ROC

Понимание переподгонки

Если модель машинного обучения отлично работает в среде разработки, но заметно ухудшается в производственной, мы считаем, что модель перегружена. Это означает, что обученная модель слишком близко следует обучающему набору данных. Это указывает на то, что в правилах, созданных моделью, слишком много деталей. Компромисс между дисперсией модели и смещением лучше всего отражает суть идеи. Давайте рассмотрим эти концепции по одному.

Смещение

Любая модель машинного обучения обучается на основе определённых предположений. В общем случае эти предположения являются упрощёнными приближениями некоторых реальных явлений. Эти предположения упрощают реальные связи между признаками и их характеристиками и облегчают обучение модели. Больше предположений — значит больше предвзятости. Таким образом, при обучении модели более упрощённые предположения = высокая смещённость, а реалистичные предположения, более репрезентативные для реальных явлений = низкая смещённость.*В линейной регрессии нелинейность признаков игнорируется, и они аппроксимируются как линейные переменные. Таким образом, линейные регрессионные модели по своей природе уязвимы к высокой смещенности.*

Дисперсия

Дисперсия количественно определяет, насколько точно модель оценивает метку, если для обучения используется другой набор данных. Он количественно определяет, является ли математическая формулировка нашей модели хорошим обобщением базовых закономерностей. Специфические правила перенастройки, основанные на конкретных сценариях и ситуациях = высокая дисперсия, а также правила, обобщённые и применимые к различным ситуациям и ситуациям = низкая дисперсия.*Наша цель в машинном обучении — обучать модели, демонстрирующие низкую смещённость и низкую дисперсию. Достичь этой цели не всегда просто и обычно не даёт дата-сайентисту спать по ночам.*

Компромисс между смещением и дисперсией

При обучении конкретной модели машинного обучения сложно определить правильный уровень обобщения для правил, составляющих обученную модель. Борьба за правильный уровень обобщения отражается через компромисс между смещением и дисперсией.*Обратите внимание, что более упрощённые предположения = больше обобщения = низкая дисперсия = высокая дисперсия*.Этот компромисс между смещением и дисперсией определяется выбором алгоритма, характеристиками данных и различными гиперпараметрами. Важно достичь правильного компромисса между смещением и дисперсией, исходя из требований конкретной задачи, которую вы пытаетесь решить.

Определение фаз классификаторов

После подготовки маркированных данных разработка классификаторов включает обучение, оценку и внедрение. Эти три этапа реализации классификатора показаны в жизненном цикле **CRISP-DM** (**Межотраслевой стандартный процесс для добычи данных**) на следующей диаграмме (жизненный цикл CRISP-DM подробно описан в *главе 5*, *Алгоритмы* графов)

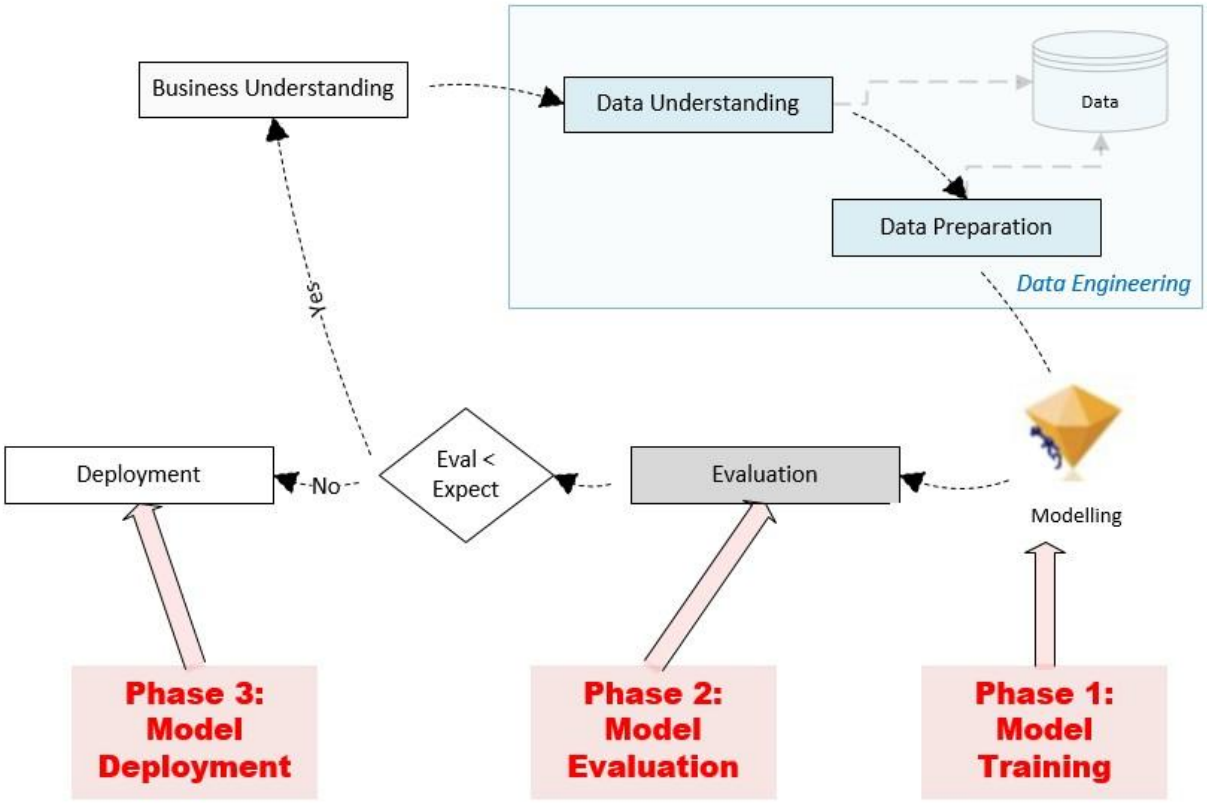


Рисунок 7.4: Жизненный цикл CRISP DM

На первых двух этапах внедрения классификатора — этапах тестирования и обучения — мы используем маркированные данные. Маркированные данные делятся на два раздела — больший раздел, называемый обучающими данными, и меньший раздел, называемый тестовыми данными. Метод случайной выборки используется для разделения входных помеченных данных на обучающие и тестовые разделы, чтобы убедиться, что оба раздела содержат согласованные шаблоны. Обратите внимание, что, как показано на предыдущей диаграмме, сначала есть этап обучения, когда обучающие данные используются для обучения модели. После завершения этапа обучения обучаемая модель оценивается на основе данных тестирования. Для количественной оценки производительности обученной модели используются различные матрицы производительности. После оценки модели наступает этап развертывания, когда обученная модель развертывается и используется для вывода с целью решения реальных задач путём маркировки немаркированных данных. Теперь давайте рассмотрим некоторые алгоритмы классификации. В следующих разделах мы рассмотрим следующие алгоритмы классификации:

- Алгоритм дерева принятия решений
- Алгоритм XGBoost
- Алгоритм случайного леса
- Алгоритм логистической регрессии
- Алгоритм **Support Vector Machine** (**SVM**)

Наивный алгоритм Байеса

Начнём с алгоритма дерева решений.

Алгоритм классификации по дереву принятия решений

Дерево принятия решений основано на подходе рекурсивного разбиения (разделяй и властвуй), который генерирует набор правил, которые можно использовать для предсказания метки. Он начинается с корневой вершины и разделяется на несколько ветвей. Внутренние узлы представляют тест по определённому атрибуту, а результат теста представлен ветвлением на следующий уровень. Дерево решений заканчивается листовыми узлами, содержащими решения. Процесс останавливается, когда разбиение перестаёт улучшать результат.

Понимание алгоритма классификации по дереву принятия решений

Отличительной чертой классификации дерева решений является генерация иерархии правил, интерпретируемых человеком, которые используются для прогнозирования метки во время выполнения. Алгоритм имеет рекурсивную природу. Создание этой иерархии правил включает следующие шаги:

А. **Найдите самую важную особенность**: из всех признаков алгоритм определяет ту особенность, которая лучше всего различает точки данных в обучающем наборе данных по метке. Расчёт основан на таких метриках, как получение информации или примеси Джини.

Б. **Bifurcate**: Используя наиболее известную важную особенность, алгоритм создаёт критерий, который разделяет обучающий набор данных на две ветки:

- Точки данных, соответствующие
- критерию. Точки данных, не

соответствующие критерию

В. **Проверьте наличие листовых узлов**: если какая-либо результирующая ветвь в основном содержит метки одного класса, ветвление становится финальной и получается листовой узел.

Г. **Проверьте условия остановки и повторите**: если указанные условия остановки не выполнены, алгоритм вернётся к *шагу* 1 для следующей итерации. В противном случае модель отмечается как обученная, и каждый узел результирующего дерева решений на самом низком уровне обозначается как листовой узел. Условие остановки может быть таким простым, как определение количества итераций, или использовать стандартное условие остановки, при котором алгоритм останавливается, как только достигает определённого уровня однородности для каждого из листовых узлов.

Алгоритм дерева принятия решений можно объяснить следующей диаграммой:

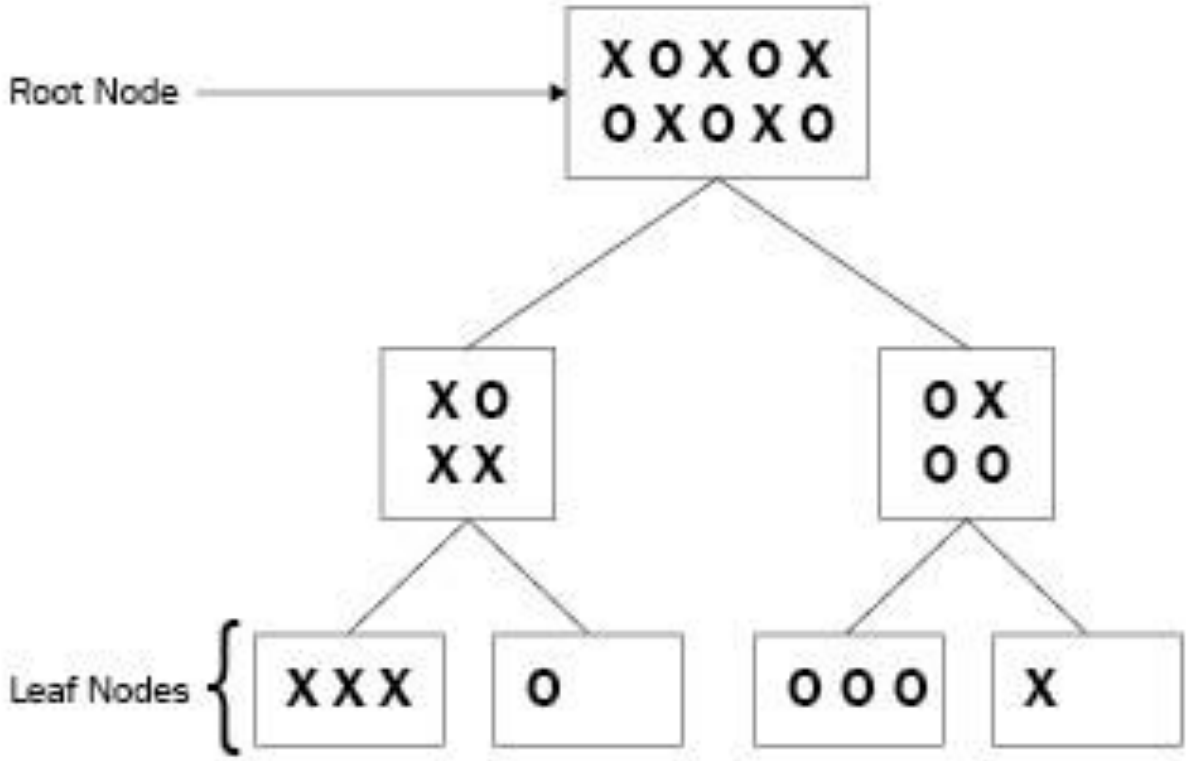


Рисунок 7.5: Дерево принятия решений

На предыдущей диаграмме корень содержит множество окружностей и крестов. Алгоритм создаёт критерий, который пытается отделить окружности от крестов. На каждом уровне дерево принятия решений создаёт разбиения данных, которые, как ожидается, будут становиться всё более однородными начиная с первого уровня и выше. Совершенный классификатор имеет листовые узлы, содержащие только окружности или кресты. Обучение совершенных классификаторов обычно сложно из-за присущей случайности обучающего набора данных.

Использование алгоритма классификации дерева принятия решений для задачи классификаторов

Теперь давайте используем алгоритм классификации дерева принятия решений для распространённой задачи, которую мы ранее определили, чтобы предсказать, купит ли клиент продукт:

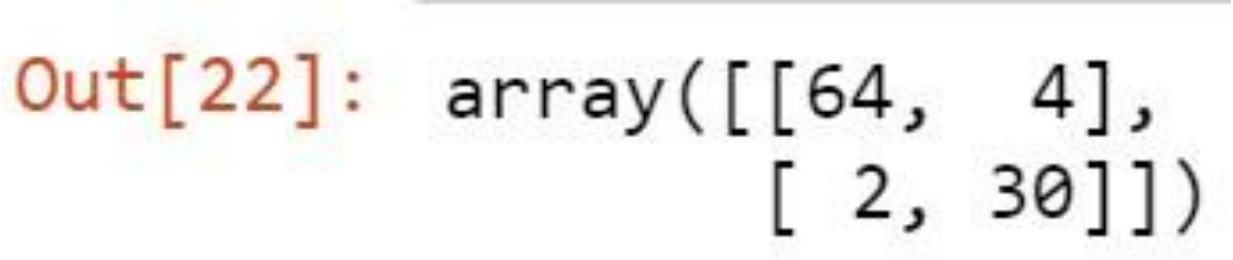
1. Для начала создадим алгоритм классификации дерева принятия решений и обучим модель, используя обучающую часть данных, подготовленную для наших классификаторов:

```
Classifier = sklearn.tree.DecisionTreeClassifier(criterion = 'энтропия', random_state = 100,
max_d classifier.fit(X_train, y_train)
```

1. Теперь давайте используем нашу обученную модель, чтобы предсказать метки для тестовой части наших маркированных данных. Давайте создадим матрицу путаницы, которая поможет резюмировать производительность нашей обученной модели:

```
импортировать sklearn.metrics как метрики
y_pred = classifier.predict(X_test) cm =
metrics.confusion_matrix(y_test, y_pred) cm
```

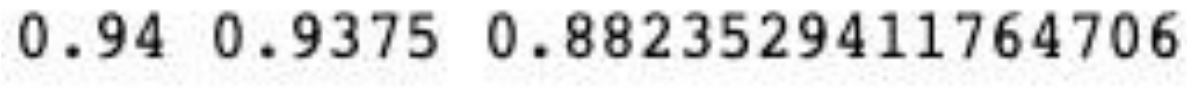
Это даёт следующий результат:



1. Теперь вычислим значения `точности , воспоминания , и` точности `для созданного классификатора с помощью алгоритма классификации дерева решений:`

```
точность = 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)
```

1. Запуск предыдущего кода даст следующий результат:



Показатели эффективности помогают сравнивать разные методы моделирования тренировок друг с другом.

Сильные и слабые стороны классификаторов дерева решений

В этом разделе давайте рассмотрим сильные и слабые стороны использования алгоритма классификации по дереву решений.

Сильные сторони

Ниже приведены сильные стороны классификаторов дерева решений:

- Правила моделей, созданных с использованием алгоритма дерева решений, интерпретируемы человеком. Такие модели называются **белыми ящиками.** Модели белого ящика необходимы всякий раз, когда требуется прозрачность для отслеживания деталей и причин принятых решений, принимаемых моделью. Эта прозрачность крайне важна в приложениях, где мы хотим предотвратить предвзятость и защитить уязвимые сообщества. Например, модель белого ящика обычно является необходимым для критически важных случаев использования в государственных и страховых отраслях.
- Классификаторы дерева решений разработаны для извлечения информации из дискретного пространства задач. Это означает, что большинство признаков являются переменными категорий, поэтому использование дерева решений для обучения модели — хороший выбор.

Слабые места

Ниже приведены слабые места классификаторов дерева решений:

- Если дерево, сгенерированное классификатором дерева решений, заходит слишком глубоко, правила охватывают слишком много деталей, что приводит к перенаправленной модели. При использовании алгоритма дерева принятия решений нужно учитывать, что деревья решений уязвимы к перенагону, поэтому необходимо обрезать дерево, когда это необходимо, чтобы этого предотвратить.
- Недостатком классификаторов дерева решений является их неспособность захватывать нелинейные отношения в правилах, которые они создают.

Сценарии использования

В этом разделе давайте рассмотрим сценарии использования, для которых используется алгоритм дерева решений.

Классификационные рекорды

Классификаторы деревьев принятия решений могут использоваться для классификации точек данных, например, в следующих примерах:

- **Ипотечные заявки**: Обучение бинарного классификатора для определения вероятности дефолта
- заявителя.**Сегментация клиентов**: Классифицировать клиентов на клиентов с высокой ценностью,
- средней и низкой стоимостью, чтобы маркетинговые стратегии можно было адаптировать под
- каждую категорию.

Медицинская диагностика: Обучать классификатор, который может классифицировать доброкачественное или злокачественное образование.

Анализ эффективности лечения: Обучение классификатора, который может отмечать пациентов, положительно отреагировавших на конкретное лечение.

Выбор функций

Алгоритм классификации дерева принятия решений выбирает небольшое подмножество признаков для создания правил. Этот выбор признаков можно использовать для выбора признаков для другого алгоритма машинного обучения, если у вас много функций.

Понимание методов ансамбля

Ансамбль — это метод в машинном обучении, позволяющий создать более одной слегка отличающейся модели с использованием различных параметров и затем объединить их в агрегатную модель. Чтобы создать эффективные ансамбли, нам нужно найти критерий агрегации для генерации результирующей модели. Давайте рассмотрим некоторые ансамблевые алгоритмы.

Реализация градиентного бустинга с помощью алгоритма XGBoost

XGBoost был создан в 2014 году и основан на принципах градиентного бустинга. Он стал одним из самых популярных алгоритмов классификации ансамблей. Он генерирует множество взаимосвязанных деревьев и использует градиентный спуск для минимизации
остаточная ошибка. Это делает его идеальным для распределённых инфраструктур, таких как Apache Spark, или для облачных вычислений, таких как Google Cloud или **Amazon Web Services** (**AWS).**Теперь посмотрим, как можно реализовать градиентное бустинг с помощью алгоритма XGBoost:

1. Сначала мы создадим классификатор XGBClassfier и обучим модель с помощью обучающей части данных:

```
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)
```

1. Затем мы сгенерируем прогнозы на основе новой обученной модели:

```
y_pred = классификатор.predict(X_test) cm =
metrics.confusion_matrix(y_test, y_pred) cm
```

Даёт следующий результат:

```
Out[21]: array([[64,  4],
                [ 3, 29]])
```

1. Наконец, мы 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 даёт следующий результат:

```
0.93 0.90625 0.8787878787878788
```

Далее рассмотрим алгоритм случайного леса.

Используя алгоритм случайного леса

Случайный лес — это тип ансамблевого метода, который работает путём объединения нескольких деревьев принятия решений для уменьшения как смещения, так и дисперсии.

Обучение алгоритма случайного леса

В процессе обучения этот алгоритм берет *N* выборок из обучающих данных и создаёт *m* подмножеств наших общих данных. Эти подмножества создаются путём случайного выбора некоторых строк и

столбцов входных данных. Алгоритм строит *m* независимых деревьев решений. Эти классификационные деревья представлены `от C1` до `Cm` .

Использование случайного леса для предсказаний

После обучения модели её можно использовать для маркировки новых данных. Каждое отдельное дерево генерирует метку. Окончательный прогноз определяется голосованием по этим индивидуальным прогнозам, как показано:

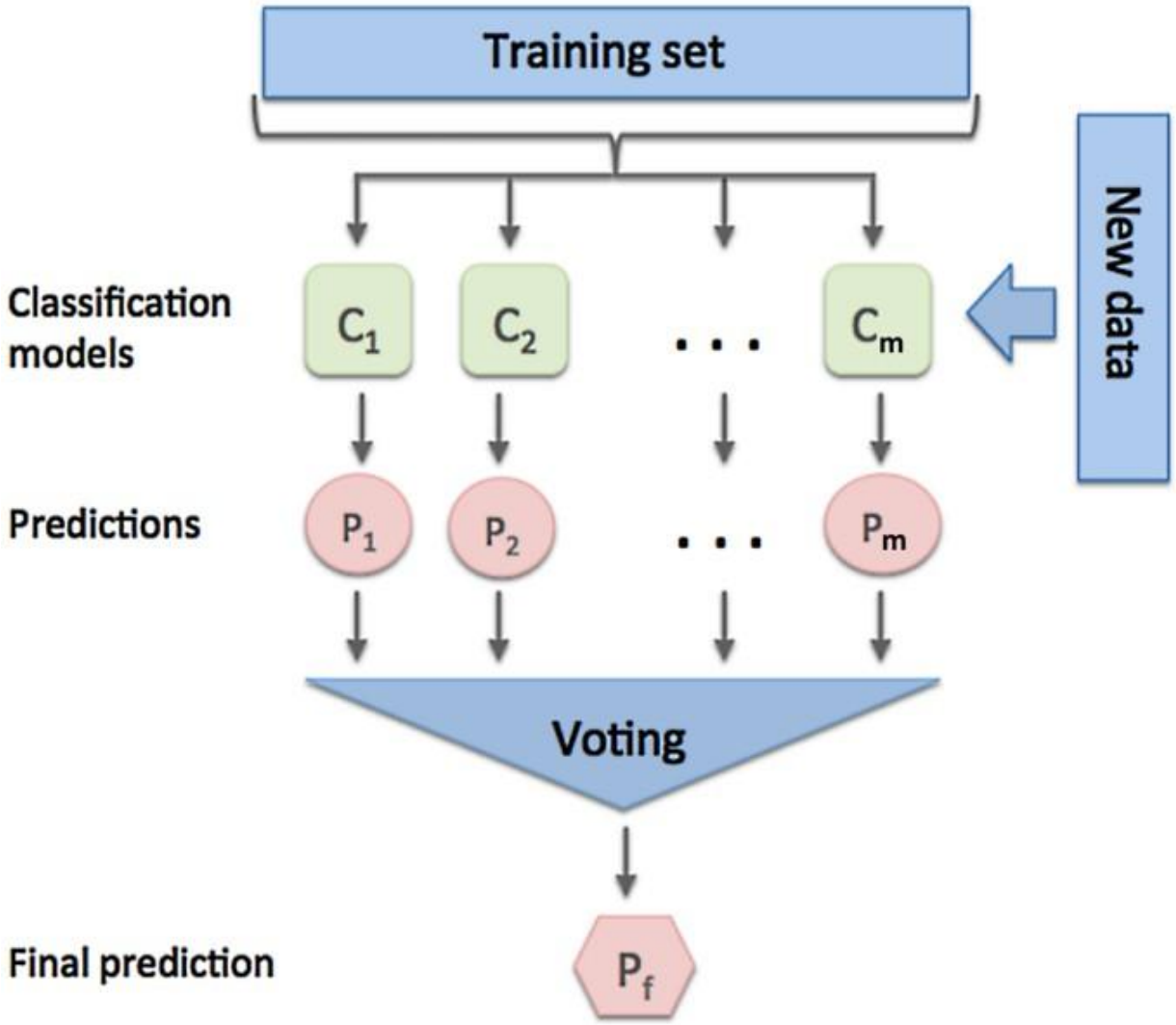


Рисунок 7.6: Случайный лес

Обратите внимание, что на предыдущей диаграмме m деревьев обучаются, что представлено от `C1` до `Cm` . То есть Деревья = {C1,..,Cm}Каждое из деревьев генерирует предсказание, представленное множеством:Индивидуальные предсказания = P= {P1,..., Pm}Окончательное предсказание представлено как Pf . Он определяется большинством индивидуальных прогнозов.

Функция `режима` может использоваться для поиска решения большинства (`режим` — это число, которое повторяется чаще всего и находится в большинстве). Индивидуальное предсказание и итоговое предсказание связаны следующим образом: Pf = режим (P)

Различие алгоритма случайного леса от ансамблевого бустинга

Каждое дерево, порождённое алгоритмом случайного леса, полностью независимо друг от друга. Он не знает деталей других деревьев в ансамбле. Это отличает его от других техник, таких как усиление ансамбля.

Использование алгоритма случайного леса для задачи классификаторов

Давайте создадим алгоритм случайного леса и используем его для обучения нашей модели с использованием обучающих данных. Здесь мы рассмотрим два ключевых гиперпараметра:

-
- `n_estimators`
  `max_depth`

`Гиперпараметр n_estimators` определяет, сколько отдельных деревьев решений построено, а `гиперпараметр max_depth` определяет, насколько глубоко может проникнуть каждое из этих отдельных деревьев решений. Иными словами, дерево решений может продолжать расщеплять и делиться, пока не появится узел, представляющий каждый конкретный пример в обучающем наборе. Устанавливая `max_depth` , мы ограничиваем, сколько уровней разделений он может получить. Это контролирует сложность модели и определяет, насколько она соответствует обучающим данным. Если обратиться к следующему результату, `n_estimators` управляет шириной модели случайного леса, а `max_depth` — глубиной модели:

```
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)
```

После обучения модели случайного леса используем её для предсказаний:

```
y_pred = классификатор.predict(X_test) cm =
metrics.confusion_matrix(y_test, y_pred) cm
```

Что даёт результат как:

```
Out[10]: array([[64,  4],
                [ 3, 29]])
```

Теперь давайте количественно 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)` Мы увидим следующий результат:

```
0.93 0.90625 0.8787878787878788
```

Далее рассмотрим логистическую регрессию.

Логистическая регрессия

Логистическая регрессия — это алгоритм классификации, используемый для бинарной классификации. Он использует логистическую функцию для формулировки взаимодействия между входными признаками и меткой. Это один из самых простых методов классификации, используемый для моделирования бинарной зависимой переменной.

Предположения

Логистическая регрессия предполагает следующее: Обучающий набор данных не имеет отсутствующего значения.

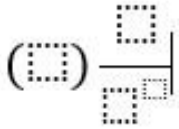
- Метка является бинарной переменной категории.

Метка является порядком — другими словами, категориальной переменной с упорядочёнными значениями.

- Все признаки или входные переменные независимы друг от друга.

Установление отношений

Для логистической регрессии предсказанное значение вычисляется следующим образом: ý= σ (ω X +j)Предположим, что z = ωX+ jИтак:



$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Приведённое соотношение можно графически показать следующим образом:

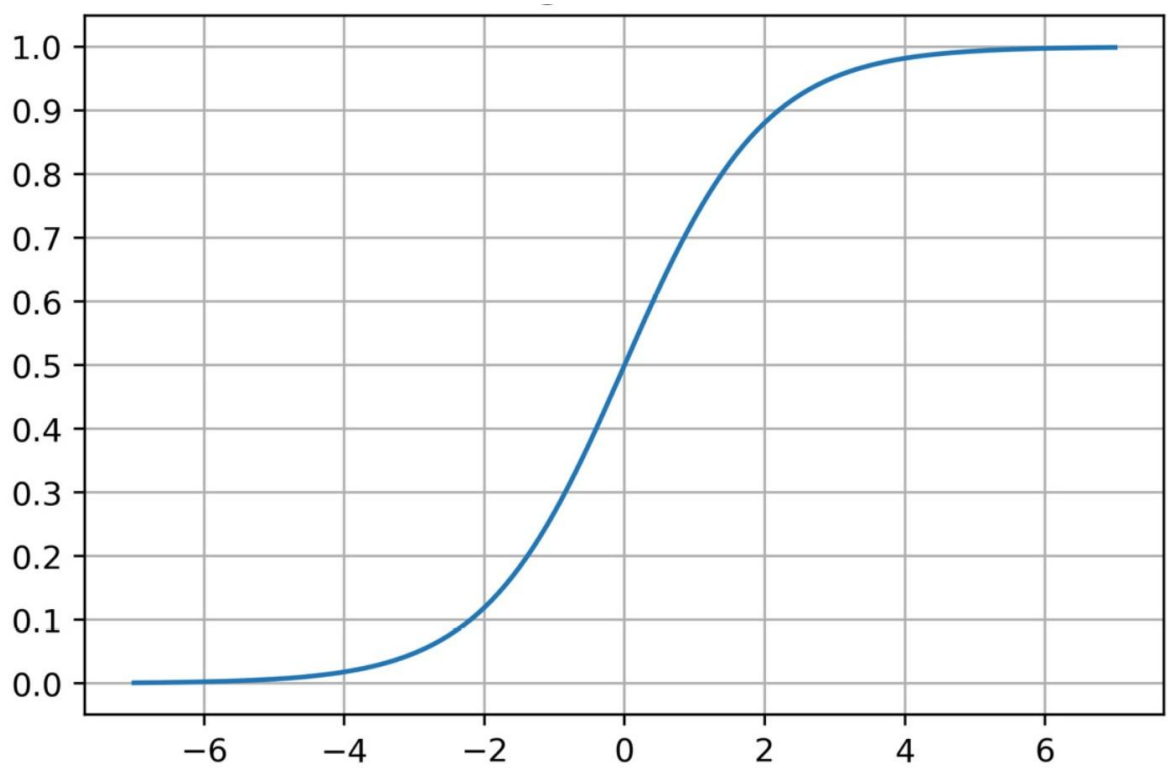


Рисунок 7.7: Построение сигмовидной функции

Обратите внимание, что если *z* велико, то σ (*z*) будет равен `1` . Если *z* очень мало или большое отрицательное число, σ (z) будет равна `0` . Также, когда z равно 0, то σ (*z*)=0,5. Сигмоид — естественная функция для представления вероятностей, так как он строго ограничен между 0 и 1. Цель обучения логистической регрессионной модели — найти правильные значения для *w* и *j.Логистическая регрессия называется по функции для её формулирования* — **логистической** *или* **сигмоидной функцией**.

Функции убытков и затрат

Функция `потерь` определяет, как мы хотим количественно оценить ошибку для конкретного примера в наших обучающих данных. `Функция затрат` определяет, как мы хотим минимизировать ошибку во всём нашем обучающем наборе данных. Таким образом, `функция потерь` используется для одного из примеров в обучающем наборе данных, а `функция затрат` — для общей стоимости, которая количественно оценивает общее отклонение фактических и прогнозируемых значений. Он зависит от выбора *w* и *h*. Функция `потерь`, используемая в логистической регрессии для определённого примера i в обучающем наборе, выглядит следующим образом: *Потеря (ý(i), y(i)) = - (y(i) log ý(i) + (1-y(i)) log (1-ý(i)))* *Обратите* внимание, что когда *y(i) = 1, Loss(ý(i), y(i)) = logý(i)*. Минимизация потерь приведёт к большому значению ý$^{(i)}$. Будучи сигмоидной функцией, максимальное значение будет 1 . Если *y(i) = 0, потеря (ý(i), y(i)) = - log (1-ý(i))*). Минимизация потерь приведёт к тому, что *ý(i)* будет максимально маленьким, то есть 0 . Функция затрат логистической регрессии следующая:

$$Cost(w,b) = \frac{1}{b}\sum Loss(\acute{y}^{(i)}, y^{(i)})$$

Когда использовать логистическую регрессию

Логистическая регрессия отлично подходит для бинарных классификаторов. Логистическая регрессия работает плохо, когда данные огромны, но качество данных не очень хорошее. Он может отражать отношения, которые не слишком сложны. Хотя обычно она не даёт максимальной производительности, она задаёт очень хороший ориентир для начала.

Использование алгоритма логистической регрессии для задачи классификаторов

В этом разделе мы рассмотрим, как можно использовать алгоритм логистической регрессии для задачи классификаторов:

1. Сначала создадим модель логистической регрессии и обучим её, используя обучающие данные:

```
из sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state = 0)
classifier.fit(X_train, y_train)
```

1. Давайте предсказаем значения тестовых данных и создадим матрицу путаницы:

```
y_pred = классификатор.predict(X_test) cm =
metrics.confusion_matrix(y_test, y_pred) cm
```

При запуске предыдущего кода получаем следующий результат:

```
Out[11]: array([[65,  3],
                [ 6, 26]])
```

1. Теперь давайте рассмотрим показатели производительности:

```
точность = 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)
```

1. При запуске предыдущего кода получаем следующий результат:

```
0.91 0.8125 0.896551724137931
```

Далее давайте посмотрим на **SVM**.

Алгоритм SVM

Теперь давайте посмотрим на SVM. SVM — это классификатор, который находит оптимальную гиперплоскость, максимизирующую замежность между двумя классами. В SVM наша цель оптимизации — максимизировать маржу. Поля определяется как расстояние между разделяющей гиперплоскостью (границей решения) и обучающими образцами, ближайшими к этой гиперплоскости, **называемыми опорными векторами**. Итак, давайте начнём с очень простого примера с двумя измерениями — *X1* и *X2*. Нам нужна линия, отделяющая круги от крестов. Это показано на следующей диаграмме:

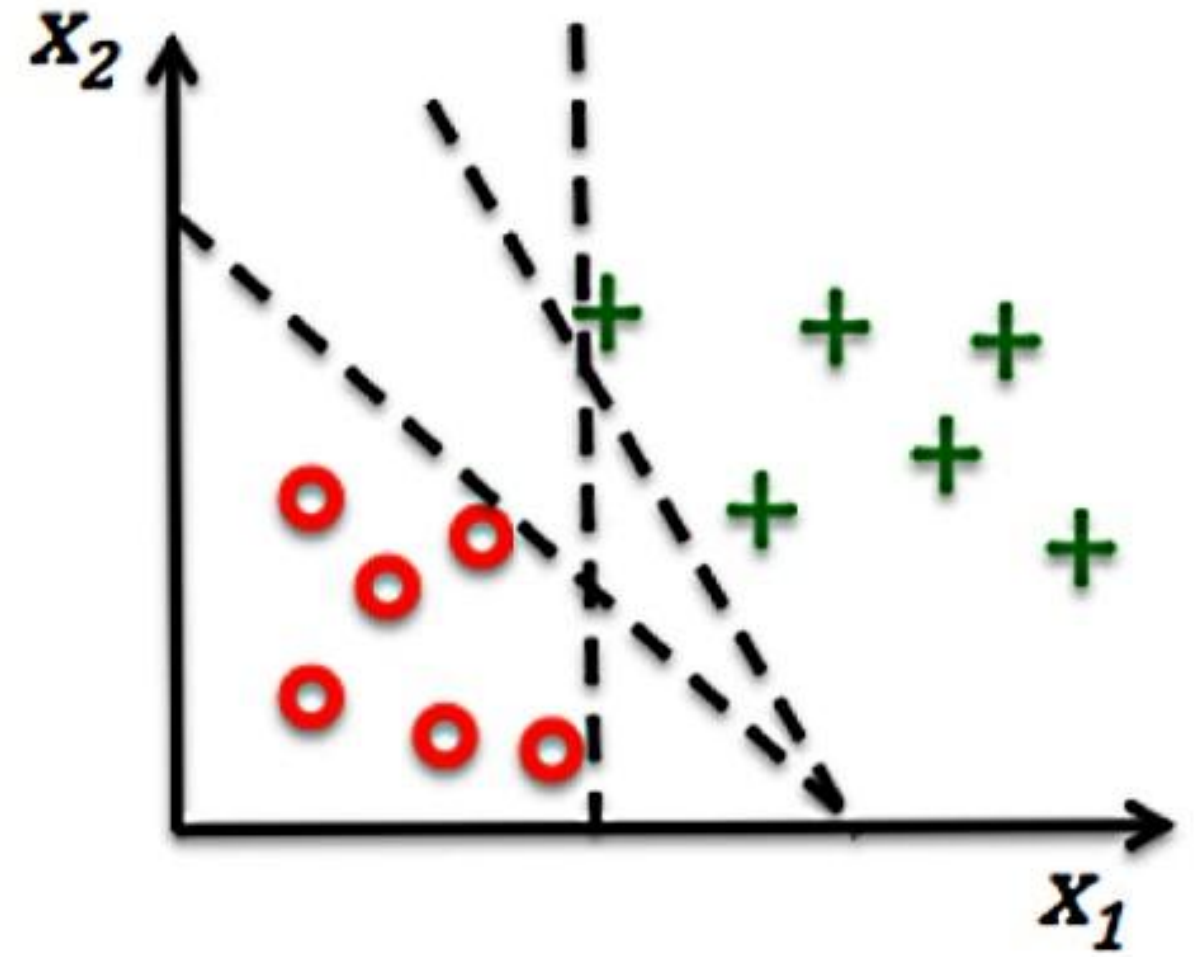


Рисунок 7.8: Алгоритм SVM

Мы провели две линии, и обе идеально отделяют кресты от кругов. Однако должна быть оптимальная линия или граница решения, которая даст нам наилучшие шансы правильно классифицировать большинство дополнительных примеров. Разумным выбором может быть линия, равномерно расположенная между этими двумя классами, чтобы создать небольшой буфер для каждого класса, как показано:

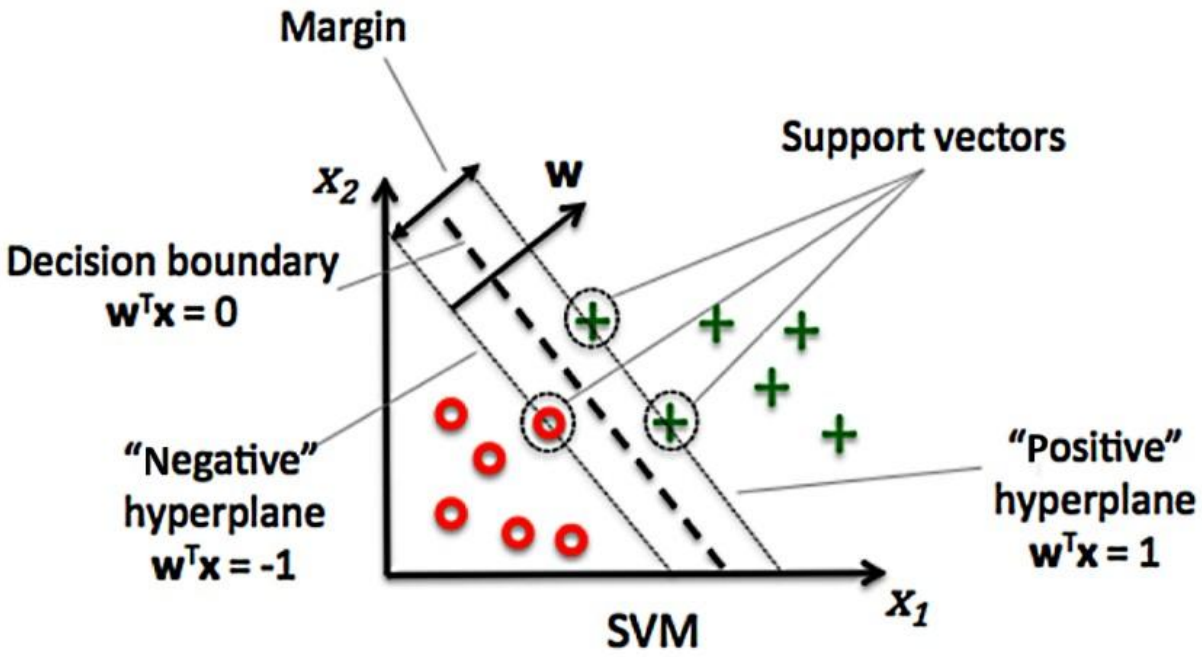


Рисунок 7.9: Понятия, связанные с SVM

Теперь давайте посмотрим, как мы можем использовать SVM для обучения классификатора для нашей задачи.

Использование алгоритма SVM для задачи классификаторов

Сначала создадим классификатор SVM, а затем используем обучающую часть помеченных данных для его обучения. Гиперпараметр ядра определяет тип преобразования, применяемого к входным данным, чтобы сделать их линейно разделяемыми:

```
из sklearn.svm импорт SVC
классификатор = SVC(ядро = «линейное», random_state =
0) classifier.fit(X_train, y_train)
```

1. После обучения давайте сгенерируем прогнозы и посмотрим на матрицу путаницы:

```
y_pred = классификатор.predict(X_test) cm =
metrics.confusion_matrix(y_test, y_pred) cm
```

1. Обратите внимание на следующий результат:

```
Out[9]: array([[66,  2],
               [ 9, 23]])
```

1. Теперь давайте рассмотрим различные показатели производительности:

```
точность = 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 запуска предыдущего кода получаем следующие значения в качестве выхода:

```
0.89 0.71875 0.92
```

Понимание наивного алгоритма Байеса

Основываясь на теории вероятностей, наивный Байес — один из самых простых алгоритмов классификации. При правильном использовании он может давать точные прогнозы. Наивный алгоритм Байеса назван s0 по двум причинам:

- Она основана на наивном предположении, что между признаками и входной переменной
- существует независимость. Она основана на теореме Байеса.

Этот алгоритм пытается классифицировать экземпляры на основе вероятностей предыдущих атрибутов/экземпляров, предполагая полную независимость атрибутов. Существует три типа мероприятий:

- **Независимые** мероприятия не влияют на вероятность повторения нового события (например, получение письма с предложением бесплатного входа на техническое *событие и* реорганизация в
- вашей компании).

Зависимые события влияют на вероятность повторения другого события; то есть они связаны каким-то образом (например, вероятность того, что вы успеете на конференцию вовремя, может зависеть из-за забастовки сотрудников авиакомпаний или рейсов, которые могут не выполняться вовремя).

- **Взаимно исключающие** события не могут происходить одновременно (например, вероятность бросать тройку и шестерку на одном броске кубика равна 0 — эти два исхода взаимоисключают).

Теорема Байеса

Теорема Байеса используется для вычисления условной вероятности между двумя независимыми событиями — *A* и *B*. Вероятность произошедшего событий *A* и *B* представлена как P(*A*) и P(*B*). Условная вероятность представлена как P(*B*|*A*), что является условной вероятностью произошедшего события B при условии, что событие *A* произошло:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Вычисление вероятностей

Наивный Байес основан на основании вероятностных принципов. Вероятность произошения одного события (наблюдательная вероятность) рассчитывается путём деления количества таких случаев на общее число процессов, которые могли привести к этому событию. Например, колл-центр получает более 100 звонков в поддержку в день, 50 раз за месяц. Вам нужно знать вероятность того, что на звонок откликнутся менее чем за 3 минуты, исходя из предыдущих случаев. Если колл-центр сумел повторить этот временной рекорд 27 раз, то наблюдательная вероятность того, что 100 звонков будут приняты менее чем за 3 минуты, будет следующей: *P(100 звонков поддержки менее чем за 3 минуты) = (27 / 50) = 0,54 (54%)* 100 звонков можно принять менее чем за 3 минуты примерно за половину времени, Основываясь на записях о 50 случаях, когда это произошло в прошлом.

Правила умножения для событий AND

Чтобы рассчитать вероятность происходящего одновременно двух или более событий, рассмотрим, являются ли события независимыми или зависимыми. Если они независимы, используется простое правило умножения:*P(результат 1 И результат 2) = P(результат 1) * P(результат 2)*Например, для расчёта вероятности получения письма с бесплатным доступом к техническому событию *и* реорганизацией на рабочем месте используется это простое правило умножения. Эти два события независимы, так как возникновение одного не влияет на вероятность повторения 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(электронная почта И реорганизация) = P(электронная почта) * P(реорганизация) = (0,31) * (0,82) = 0,2542 (25%)

Общее правило умножения

Если два или более событий зависят, используется общее правило умножения. Эта формула действительна как в случаях независимых, так и в зависимых событиях:*P(исход 1 И исход 2)=P(исход 1)*P(результат 2 | результат 1)*Обратите внимание, что `P(исход 2 | исход 1)` относится к условной вероятности возникновения `исхода 2`, если исход `1` уже произошёл. Формула учитывает зависимость между событиями. Если события независимы, то условная вероятность не имеет значения, так как один исход не влияет на вероятность повторения другого, а `P(исход 2 | исход 1)` — это просто `P(исход 2)`. Обратите внимание, что формула в этом случае просто становится простым правилом умножения.

Правила добавления для мероприятий операционной

При расчёте вероятности произшествия одного или другого события (взаимно исключающее) используется следующее простое правило сложения:*P(исход 1 ИЛИ исход 2) = P(исход 1) + P(исход 2)Например*, какова вероятность выпадения шестерки или тройки? Чтобы ответить на этот вопрос, сначала обратите внимание, что оба исхода не могут происходить одновременно. Вероятность выбросить шестерку равна (1 / 6), и то же самое можно сказать для броска 3:*P(6 ИЛИ 3) = (1 / 6) + (1 / 6) = 0,33 (33%)*Если события не взаимоисключают друг друга и могут происходить одновременно, используйте следующую формулу сложения, которая всегда действует как в случаях взаимной исключительности, так и при невзаимной исключительности: *P(исход 1 ИЛИ результат 2) = P(исход 1) + P(исход 2) P(исход 1 И результат 2)*

Использование наивного алгоритма Байеса для задачи классификаторов

Теперь давайте воспользуемся наивным алгоритмом Байеса для решения задачи классификаторов:

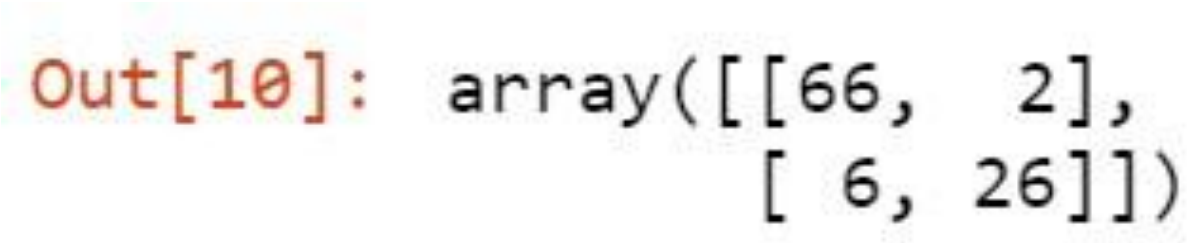
1. Сначала импортируем функцию `GaussianNB()` и используем её для обучения модели:

```
из sklearn.naive_bayes import GaussianNB
классификатор = GaussianNB()
classifier.fit(X_train, y_train)
```

1. Теперь давайте используем обученную модель для прогнозирования результатов. Мы используем его для прогнозирования меток для нашего тестового раздела, который X_test :

```
Прогнозирование результатов тестового
множества y_pred =
classifier.predict(X_test) cm =
metrics.confusion_matrix(y_test,
y_pred) cm
```

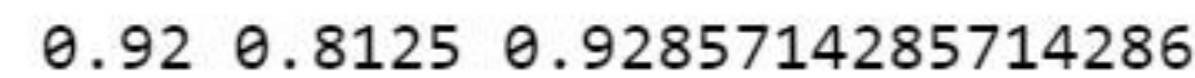
1. Теперь рассмотрим матрицу путаницы:



1. Теперь распечатаем матрицы производительности, чтобы количественно оценить качество нашей 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)
```

Что даёт результат как:



Для алгоритмов классификации победителем является

Давайте рассмотрим показатели производительности различных алгоритмов, которые мы представили. Это обобщено в следующей таблице:

Алгоритм	**Точность**	**Воспоминание**	**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	0.81	0.89	

0,91

SVM	0.89	0.71	0.92
Наивный Байес	0.92	0.81	0.92

Смотря на приведённую таблицу, можно заметить, что классификатор дерева решений показывает лучшие результаты по точности и запоминаниям. Если мы ищем точность, то есть связь между SVM и наивным Байесом, так что любой вариант нам подойдёт.

Понимание алгоритмов регрессии

Модель контролируемого машинного обучения использует один из регрессионных алгоритмов, если метка является непрерывной переменной. В этом случае модель машинного обучения называется регрессором. В этом разделе мы представим различные алгоритмы, которые можно использовать для обучения модели регрессии с контролируемым машинным обучением — или, проще говоря, регрессоров. Прежде чем перейти к деталям алгоритмов, давайте сначала создадим задачу для тестирования их производительности, возможностей и эффективности.

Презентация вызова регрессоров

Подобно подходу, который мы использовали с алгоритмами классификации, сначала мы представим задачу для решения как вызов для всех алгоритмов регрессии. Мы назовём эту распространённую проблему вызовом регрессоров. Затем мы используем три различных регрессионных алгоритма для решения этой задачи. Этот подход с использованием общей задачи для различных регрессионных алгоритмов имеет два преимущества:

- Мы можем подготовить данные один раз и использовать подготовленные данные для всех трёх
- регрессионных алгоритмов.

Мы можем сравнить производительность трёх регрессионных алгоритмов с осмысленным способом, так как будем использовать их для решения одной и той же задачи.

Давайте рассмотрим формулировку задачи задания.

Формулировка проблемы вызова регрессоров

В наши дни важно прогнозировать пробег разных автомобилей. Эффективный автомобиль полезен для окружающей среды и также экономически эффективен. Пробег можно оценить по мощности двигателя и характеристикам автомобиля. Давайте создадим вызов для регрессоров, чтобы обучить модель, способную предсказывать **количество миль на галлон** (**MPG**) автомобиля на основе его характеристик. Давайте посмотрим на исторический набор данных, который мы будем использовать для обучения регрессоров.

Изучение исторического набора данных

Ниже приведены особенности исторических данных набора данных, которые у нас есть:

Название	**Тип**	**Описание**
`НАЗВАНИЕ`	Категория	Определяет конкретное транспортное средство
`ЦИЛИНДРЫ`	Непрерывное	Количество цилиндров (от 4 до 8)
`ОБЪЁМ`	Непрерывный	Рабочий объём двигателя в кубических дюймах
`ЛОШАДИНАЯ МОЩНОСТЬ`	Непрерывная	мощность двигателя
`УСКОРЕНИЕ`	Непрерывное	Время, необходимое для разгона с 0 до 60 миль в час (за секунды)

Метка для этой задачи — это непрерывная переменная, MPG, которая указывает расход mpg для каждого транспортного средства. Давайте сначала спроектируем конвейер обработки данных для этой задачи.

Инжиниринг признаков с использованием конвейера обработки данных

Давайте посмотрим, как мы можем спроектировать повторно используемый конвейер обработки, чтобы решить проблему регрессоров. Как уже упоминалось, мы подготовим данные один раз, а затем используем их во всех алгоритмах регрессии. Давайте следуем следующим шагам:

1. Начинаем с импорта набора данных, следующим

образом: `dataset = pd.read_csv('auto.csv')`

1. Теперь давайте предварительно рассмотрим

набор данных: `dataset.head(5)`

Вот как будет выглядеть набор данных:

	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

1. Теперь перейдём к выбору функций. Давайте уберём столбец NAME, так как это только идентификатор, необходимый для автомобилей. Столбцы, используемые для идентификации строк в нашем наборе данных, не имеют значения для обучения модели. Давайте уберём эту колонку:

```
dataset=dataset.drop(columns=['NAME'])
```

1. Давайте преобразуем все входные переменные и присвоим все нулевые значения:

```
dataset=dataset.drop(columns=['NAME'])
dataset= dataset.apply(pd.to_numeric, errors='coerce') dataset.fillna(0,
inplace=True)
```

Импутация улучшает качество данных и готовит их к обучению модели. Теперь давайте посмотрим на финальный шаг:

1. Давайте разделим данные на тестовые и обучающие разделы:

```
От sklearn.model_selection импорт train_test_split #from
sklearn.cross_validation импорт train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0,25, random_state = 0)
```

Это создало следующие четыре структуры данных:

- `X_train` : Структура данных, содержащая признаки обучающих
- данных `X_test` : Структура данных, содержащая признаки
- обучающего теста `y_train` : Вектор, содержащий значения метки в обучающем наборе `данных y_test` : Вектор, содержащий значения метки в тестовом наборе данных

Теперь давайте используем подготовленные данные по трём разным регрессорам, чтобы сравнить их эффективность.

Линейная регрессия

Из всех методов контролируемого машинного обучения алгоритм линейной регрессии самый простой для понимания. Сначала мы рассмотрим простую линейную регрессию, а затем расширим концепцию до множественной линейной регрессии.

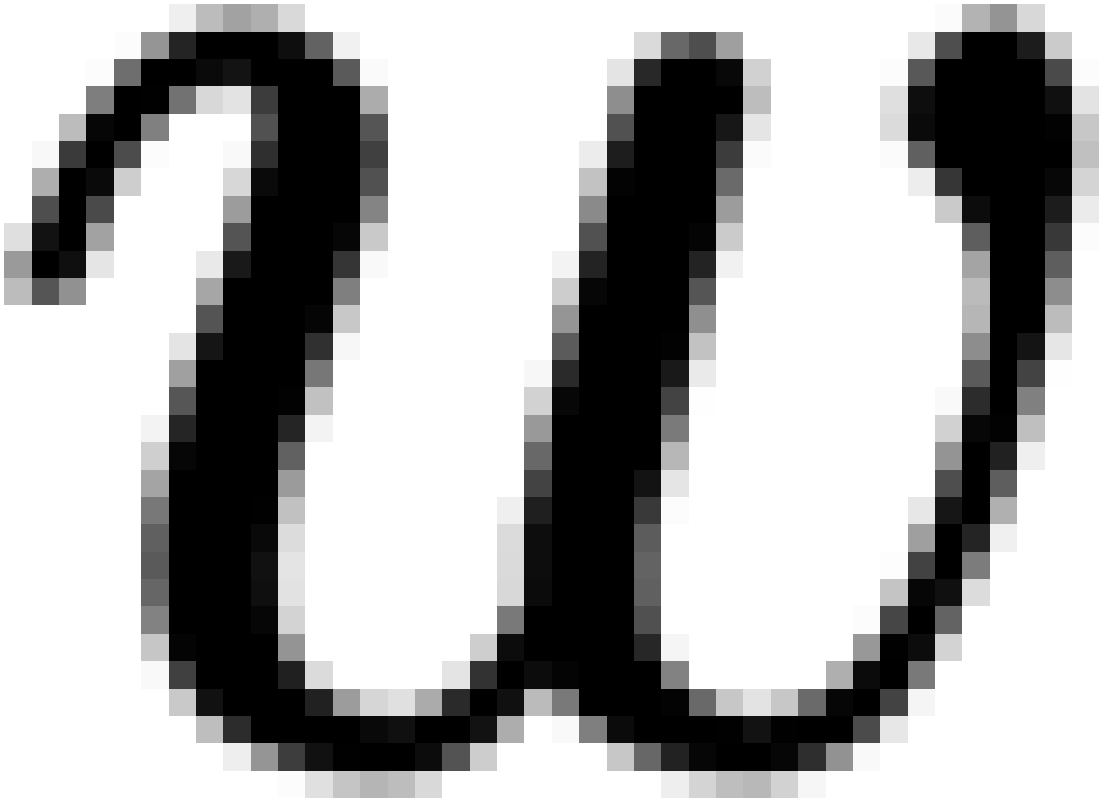
Простая линейная регрессия

В самой простой форме линейная регрессия формулирует связь между одной непрерывной независимой переменной и одной непрерывной независимой переменной. (Простая) регрессия используется для демонстрации степени, в которой изменения зависимой переменной (отображаемой по *оси y*) могут быть связаны с изменениями объяснительной переменной (показанной на *xaxis).* Его можно представить следующим образом:

$$\acute{y} = (X)w + \alpha$$

Эту формулу можно объяснить следующим

- y — зависимая переменная.
- X — независимая переменная.



-
- a — это пересечение, указывающее значение y, когда $X = 0$.

— это наклон, который показывает, насколько линия поднимается или X при каждом

Некоторые примеры связей между одной непрерывной зависимой переменной и одной непрерывной независимой переменной включают следующие:

- Вес человека и его потребление калорий
- Стоимость дома и его площадь в квадратных футах в конкретном районе,
- влажность воздуха и вероятность дождя

Для линейной регрессии и входная (независимая) переменная, и целевая (зависимая) переменная должны быть численными. Наилучшее соотношение достигается минимизацией суммы квадратов вертикальных расстояний каждой точки по прямой, проведённой через все точки. Предполагается, что связь между предикторной переменной и меткой является линейной. Например, чем больше денег вкладывается в исследования и разработки, тем выше продажи. Давайте рассмотрим конкретный пример.

Давайте попробуем сформулировать связь между маркетинговыми расходами и продажами конкретного продукта. Они оказываются напрямую взаимосвязаны друг с другом. Маркетинговые расходы и продажи отображаются на двухмерном графике и отображаются синими бриллиантами. Связь лучше всего приближается, проведя прямую линию, как на следующем графике:

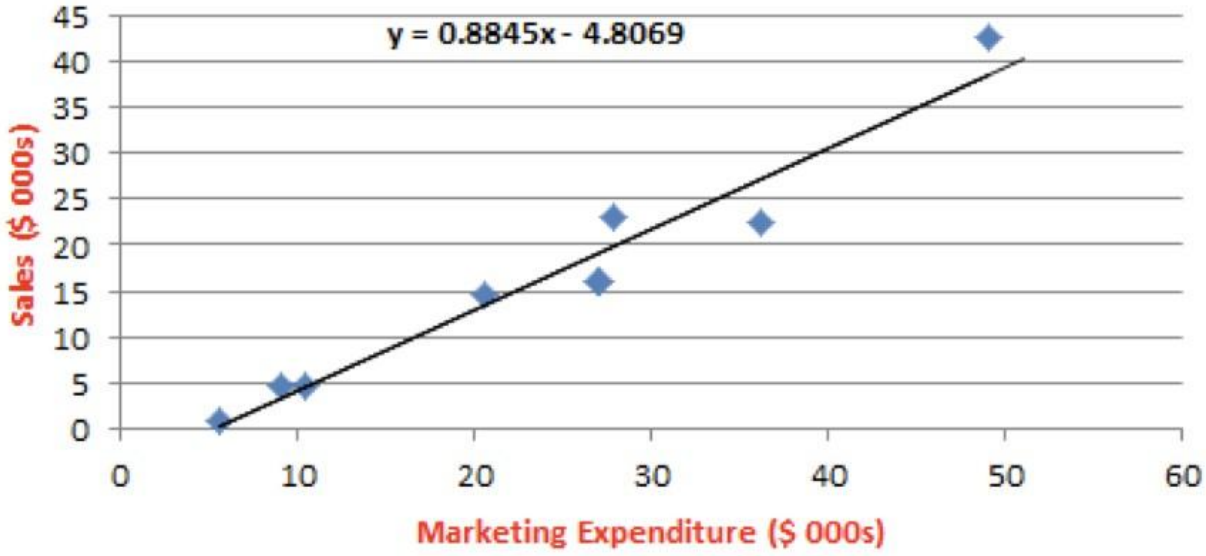


Рисунок 7.10: Линейная регрессия

Когда линейная линия проведена, мы можем увидеть математическую связь между маркетинговыми расходами и продажами.

Оценка регрессоров

Линейная линия, которую мы провели, представляет собой приближение связи между зависимыми и независимыми переменными. Даже лучшая строка будет иметь некоторое отклонение от реальных значений, как показано:

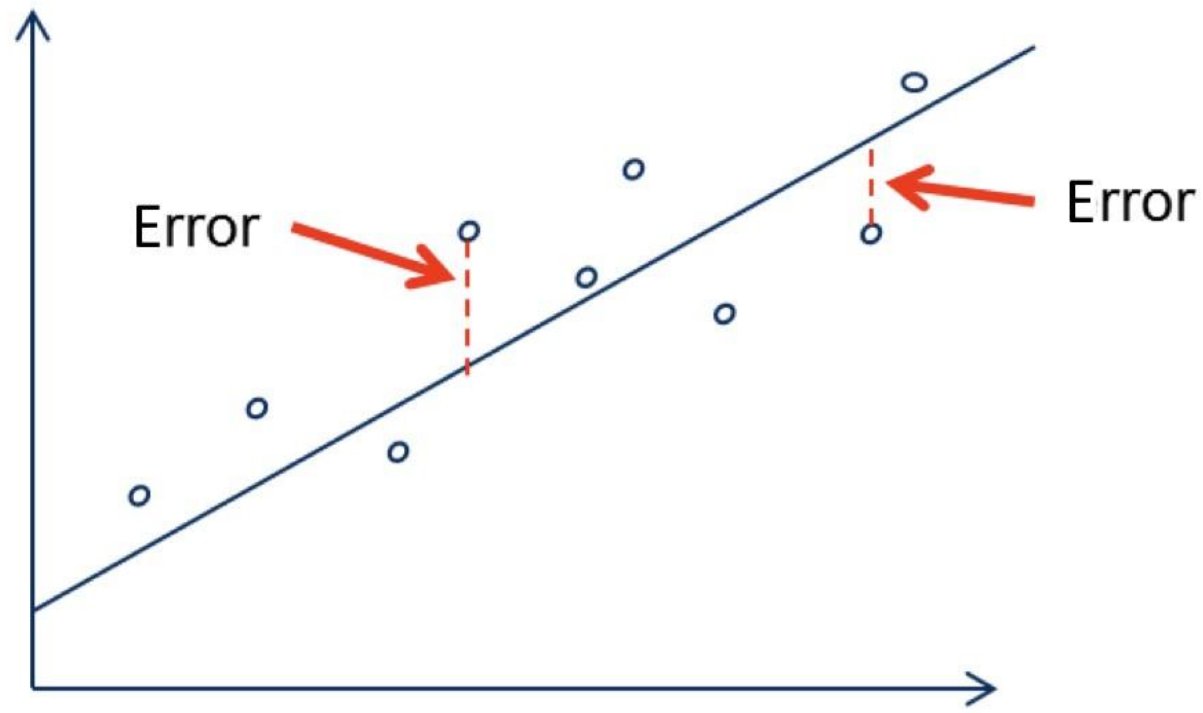


Рисунок 7.11: Оценка регрессоров

Типичный способ количественной оценки производительности линейных регрессионных моделей — использование **корневой среднеквадратической ошибки** (**RMSE**). Это математически вычисляет стандартное отклонение ошибок, допущенных обученной моделью. Например, в обучающем наборе данных функция потерь рассчитывается следующим образом: Потери ($\acute{y}$(i)$, y$(i)$) = 1/2(\acute{y}$(i)$- y$(i)$)$2Это приводит к следующей функции затрат, которая минимизирует потери всех примеров в обучающем наборе:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\acute{y}^{(i)} - y^{i})^2}$$

Давайте попробуем интерпретировать RMSE. Если RMSE — это $50 для нашего примера модели, предсказывающей цену продукта, это означает, что около 68,2% прогнозов будут соответствовать действительному значению в пределах $50 (то есть α). Это также означает, что 95% прогнозов будут соответствовать фактической стоимости в пределах $100 (то есть 2α). Наконец, 99,7% прогнозов будут соответствовать реальной стоимости в пределах $150.

Множественная регрессия

Факт в том, что большинство реальных анализов имеют более одной независимой переменной. Множественная регрессия — это расширение простой линейной регрессии. Ключевое отличие заключается в том, что для дополнительных предикторных переменных существуют дополнительные коэффициенты бета. При обучении модели цель — найти бета-коэффициенты, минимизирующие ошибки линейного уравнения. Давайте попробуем математически сформулировать связь между зависимой переменной и множеством независимых переменных (признаков). Подобно простому линейному уравнению, зависимая переменная *y* количественно определяется как сумма пересечения члена плюс произведение *коэффициентов b*, умноженное на *значение x* для каждой из i-признаков: $y = \alpha + \beta 1 x 1 + \beta 2 x 2 +...+ \beta i x_i + \varepsilon$ Ошибка представлена ε и указывает на то, что предсказания не идеальны. *Коэффициенты b* позволяют каждой признаке оказывать отдельное оценочное влияние на значение *y* из-за *изменений y* на β i при каждом единичном увеличении *xi*. Кроме того, пересечение (*α*) указывает на ожидаемое значение *y*, когда все независимые переменные равны 0. Обратите внимание, что все переменные в предыдущем уравнении могут быть представлены множеством векторов. Целевая и предикторная переменные теперь являются векторами с строкой, а коэффициенты регрессии β и ошибки *ε* также являются векторами.

Использование алгоритма линейной регрессии для задачи с регрессорами

Теперь обучаем модель с помощью обучающей части набора данных:

1. Начнём с импорта пакета линейной регрессии:

```
из импорта sklearn.linear_model LinearRegression
```

1. Затем создадим модель линейной регрессии и обучим её с помощью обучающего набора данных:

```
regressor = линейная регрессия()
regressor.fit(X_train, y_train)
```

1. Теперь предскажем результаты, используя тестовую часть набора данных:

```
y_pred = regressor.predict(X_test) из
sklearn.metrics импорт mean_squared_error из
math import sqrt
sqrt(mean_squared_error(y_test, y_pred))
```

1. Выход, полученный при выполнении предыдущего кода, погенерирует следующее:



Как обсуждалось в предыдущем разделе, RMSE — это стандартное отклонение ошибки. Он показывает, что 68,2% прогнозов будут соответствовать значения метки в пределах `4,36` .

Когда используется линейная регрессия?

Линейная регрессия используется для решения многих реальных задач, включая следующие:

- Прогнозирование продаж
- Прогнозирование оптимальных цен на продукцию
- Количественная оценка причинно-следственной связи между событием и ответом, например, в клинических испытаниях лекарств, инженерных испытаниях безопасности или маркетинговых исследованиях

Выявление закономерностей, которые можно использовать для прогнозирования будущего поведения с учётом известных критериев — например, прогнозирования страховых случаев, ущерба от стихийных бедствий, результатов выборов и уровня преступности

Слабые места линейной регрессии

Слабые места линейной регрессии следующие:

- Он работает только с числовыми признаками.
- Категориальные данные должны быть предварительно обработаны.
- Он плохо справляется с отсутствующими данными. Он делает предположения относительно данных.

Алгоритм регрессионного дерева

Алгоритм регрессионного дерева похож на алгоритм классификационного дерева, за исключением того, что метка — это непрерывная переменная, а не переменная категории.

Использование алгоритма регрессионного дерева для задачи с регрессорами

В этом разделе мы рассмотрим, как алгоритм регрессионного дерева может быть использован для задачи регрессоров:

1. Сначала обучаем модель с помощью алгоритма регрессионного дерева:

```
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')
```

1. После обучения модели регрессионного дерева используем обученную модель для предсказания значений: `y_pred = regressor.predict(X_test)`

1. Затем мы рассчитываем RMSE для количественной оценки эффективности модели:

```
Из sklearn.metrics импорт mean_squared_error из
math                import                 sqrt
```

`sqrt(mean_squared_error(y_test, y_pred))` мы получаем следующий результат:



Алгоритм регрессии с градиентным бустом

Теперь рассмотрим алгоритм регрессии с градиентным бустом. Он использует ансамбль деревьев принятия решений для более точной формулировки базовых закономерностей в данных.

Использование алгоритма регрессии с градиентным бустом для задачи регрессоров

В этом разделе мы рассмотрим, как можно использовать алгоритм регрессии с градиентным бустом для задачи с регрессорами:

1. Сначала обучаем модель с помощью алгоритма регрессии с градиентным бустом:

```
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)
```

1. После обучения модели алгоритма градиентной регрессии мы используем её для

предсказания значений: `y_pred = regressor.predict(X_test)`

1. Наконец, мы рассчитываем RMSE для количественной оценки производительности модели:

```
Импорт из sklearn.metrics mean_squared_error из
math import sqrt
sqrt(mean_squared_error(y_test, y_pred))
```

1. Запустив это, мы получим выходное значение, следующим образом:

```
Out[7]:  4.034836373089085
```

Для регрессионных алгоритмов победителем является

Давайте рассмотрим производительность трёх регрессионных алгоритмов, которые мы использовали на одних и тех же данных и в точно одном и том же сценарии использования:

Алгоритм	**RMSE**
Линейная регрессия	4.36214129677179
Регрессионное дерево	5.2771702288377
Регрессия с градиентным бустом	4.034836373089085

Если посмотреть на производительность всех алгоритмов регрессии, очевидно, что производительность градиентно-буст-регрессии наилучшая, так как у неё самый низкий RMSE. За этим следует линейная регрессия. Алгоритм регрессионного дерева показал худшие результаты для этой задачи.

Практический пример — как предсказать погоду

Давайте посмотрим, как мы можем использовать концепции, разработанные в этой главе, чтобы предсказать погоду. Предположим, что мы хотим предсказать, будет ли завтра дождь, основываясь на данных, собранных за год по конкретному городу. Данные, доступные для обучения этой модели, находятся в CSV-файле под названием `weather.csv` :

1. Импортируем данные в виде кадра данных панды:

```
Импортировать Numpy как NP
Импорт Панд как PD DF =
pd.read_csv("weather.csv")
```

1. Давайте рассмотрим столбцы фрейма данных:

```
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')
```

1. Далее рассмотрим заголовок первых 13 столбцов данных weather.csv:

```
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

1. Теперь давайте посмотрим на последние 10 столбцов данных weather.csv:

```
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

1. Давайте используем x для обозначения входных признаков. Мы уберём поле Date для списка признаков, так как оно не полезно в контексте предсказаний. Мы также уберём метку `RainTomorrow`: `x = df.drop(['Date','RainTomorrow'],axis=1)`

1. Давайте используем `y` для обозначения метки:

```
y = df['ДождьЗавтра']
```

1. Теперь разделим данные на `train_test_split` :

```
От 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)
```

1. Поскольку метка является бинарной переменной, мы обучаем классификатор. Поэтому логистическая регрессия здесь будет хорошим выбором. Сначала давайте представим модель логистической регрессии:

```
модель = Логистическая регрессия()
```

1. Теперь мы можем использовать `train_x` и `test_x` для обучения модели:

```
model.fit(train_x , test_x)
```

1. После обучения модели давайте используем её для предсказаний:

```
predict = model.predict(train_y)
```

1. Теперь давайте определим точность нашей обученной модели:

```
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
```

Теперь этот бинарный классификатор можно использовать для прогнозирования, будет ли завтра дождь. С

Краткое содержание

В этой главе мы начали с изучения основ контролируемого машинного обучения. Затем мы более подробно рассмотрели различные алгоритмы классификации. Далее мы рассмотрели различные методы оценки производительности классификаторов и изучили различные алгоритмы регрессии. Мы также рассмотрели различные методы, которые можно использовать для оценки эффективности изученных алгоритмов. В следующей главе мы рассмотрим нейронные сети и алгоритмы глубокого обучения. Мы рассмотрим методы обучения нейронной сети, а также различные инструменты и фреймворки, доступные для оценки и развертывания нейронной сети.

OceanfPDF.com

8 Алгоритмы нейронных сетей

Присоединяйтесь к Discord-пространству нашей книги

https:// packt.link/40Algos



«Алгоритма для юмора не существует». -- Роберт Манкофф

Сочетание различных факторов сделало **нейронные сети** одной из важнейших современных методов машинного обучения. К этим факторам относятся необходимость решения всё более сложных задач, взрывной рост данных и появление технологий, таких как облачные вычисления, которые обеспечивают вычислительную мощность, необходимую для проектирования и использования очень сложных алгоритмов. На самом деле, именно эта область исследований быстро развивается и ответственна за большинство основных достижений, заявляемых в передовых технологических областях, таких как

робототехника, периферийные вычисления, обработка естественного языка и беспилотные автомобили. В этой главе сначала представлены основные понятия и компоненты типичной нейронной сети. Далее представлены различные типы нейронных сетей и объясняются различные типы активационных функций, используемых в этих нейронных сетях. Далее подробно обсуждается алгоритм обратного распространения, который является самым широко используемым алгоритмом для обучения нейронной сети. Далее объясняется метод трансферного обучения, который может значительно упростить и частично автоматизировать обучение моделей. Наконец, анализ того, как использовать глубокое обучение для выявления мошеннических документов, рассматривается с помощью реального примера приложения. Основные концепции, обсуждаемые в этой главе:

- Понимание нейронных сетей
- Эволюция нейронных сетей
- Обучение нейронной сети
- Инструменты и фреймворки
- Трансферное обучение

Кейс-стади: использование глубокого обучения для выявления мошенничества

Давайте начнём с основ нейронных сетей.

Эволюция нейронных сетей

Изучая структуру нейронной сети, её базовой единицей является нейрон. Истинная сила нейронной сети заключается в её способности использовать силу нескольких нейронов, организуя их в многослойную архитектуру. Нейронная сеть создаёт многослойную архитектуру, связывая нейроны в различных слоях. Сигнал проходит через эти слои и обрабатывается разными способами в каждом из слоёв, пока не будет получен окончательный необходимый выход. Как мы увидим в этой главе, скрытые слои, используемые нейронными сетями, выступают в роли слоёв абстракции, обеспечивая глубокое обучение, которое широко используется для реализации мощных приложений, таких как Alexa от Amazon, поиск изображений Google и Google Photos.

Немного истории

Вдохновлённый работой нейронов в человеческом мозге, концепция нейронных сетей была предложена Фрэнком Розенблаттом в 1957 году. Чтобы полностью понять архитектуру, полезно кратко рассмотреть многослойную структуру нейронов в человеческом мозге. (См. рисунок 1, чтобы понять, как нейроны человеческого мозга связаны между собой.)В человеческом мозге **дендриты** действуют в роли сенсоров, фиксирующих сигнал. Сигнал затем передаётся **аксону** — длинной, тонкой проекции нервной клетки. Функция аксона — передавать этот сигнал мышцам, железам и другим нейронам. Как показано на следующей схеме, сигнал проходит через взаимосвязанные ткани, называемые **синапсами**, и затем передаётся другим нейронам. Обратите внимание, что через этот органический канал сигнал продолжает распространяться, пока не достигнет целевого мышцы или железы, где он вызывает необходимое действие. Обычно сигналу требуется семь–восемь миллисекунд, чтобы пройти через цепочку нейронов и достичь своей цели:

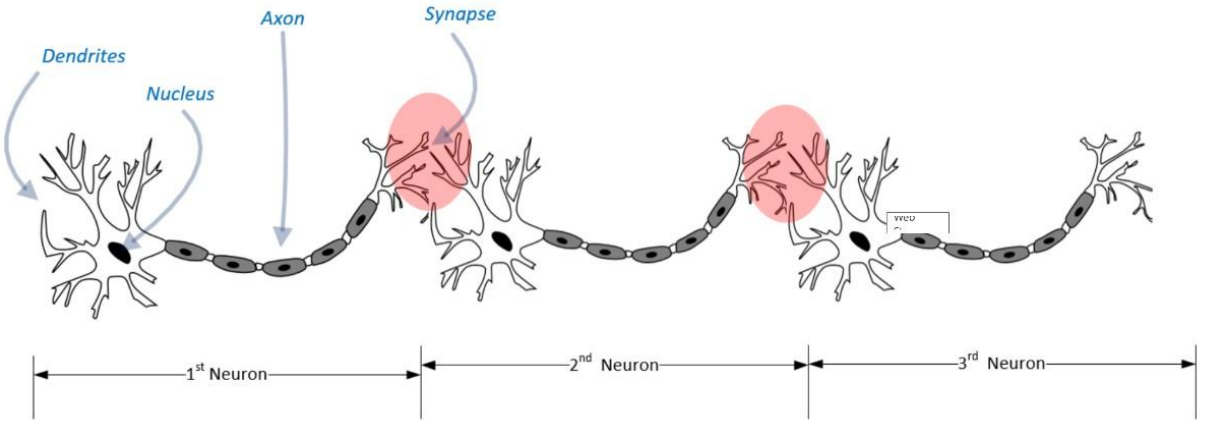


Рисунок 8.1: Нейроны, связанные в цепи в человеческом мозге

Вдохновившись этим естественным архитектурным шедевром обработки сигналов, Фрэнк Розенблатт разработал технику, позволяющую обрабатывать цифровую информацию слоями для решения сложной математической задачи. Его первоначальная попытка спроектировать нейронную сеть была довольно простой и напоминала модель линейной регрессии. Эта простая нейронная сеть не имела скрытых слоёв и называлась *перцептроном*. Простая нейронная сеть без слоёв, называемая перцептроном, стала базовым блоком для нейронных сетей.

AI Зима и рассвет AI Весны

К сожалению, когда весь интерес к предлагаемому перцептрону утих, тщательный и детальный анализ перцептрона оказался серьёзным ограничением. В 1969 году Марвин Мински и Сеймур Паперт проводили исследования, которые привели к выводу, что перцептрон неспособен изучать сложную логику. На самом деле они показали, что перцептрону будет сложно освоить даже такие простые логические функции, как XOR. Это привело к снижению интереса к машинному обучению в целом и к нейронным сетям в частности, и положило начало эпохе, ныне известной как **ИИ-зима**. Исследователи по всему миру не воспринимали бы ИИ всерьёз, считая, что он не способен решать сложные задачи. Оглядываясь назад, одной из главных причин так называемой AI-зимы было ограничение аппаратных возможностей, доступных в то время. Либо необходимая вычислительная мощность не была доступна, либо она была непомерно дорогой. К концу 1990-х годов произошёл резкий сдвиг в отношении имиджа ИИ и его воспринимаемого потенциала. Катализатором этих изменений стали достижения в распределённых вычислениях, которые обеспечили легко доступную и доступную инфраструктуру. Увидев потенциал, новые IT-гиганты того времени (такие как Google) сделали ИИ своим центром исследований и разработок. Возрожденный интерес к ИИ привёл к оттепию так называемой AI-зимы. Оттепель вдохновила новую жизнь в исследования в области ИИ. В итоге это привело к превращению нынешней эпохи в эпоху, которую можно назвать **весной ИИ**, когда существует огромный интерес к ИИ и нейронным сетям.

Понимание нейронных сетей

Сначала начнём с сердца нейронной сети — перцептрона. Можно рассматривать один перцептрон как самую простую возможную нейронную сеть, и он является основой современных сложных многоуровневых архитектур. Начнём с понимания работы перцептрона.

Понимание Персептрона

Один перцептрон имеет несколько входов и один выход, который управляется или активируется функцией активации. Это показано на рисунке 8.2.

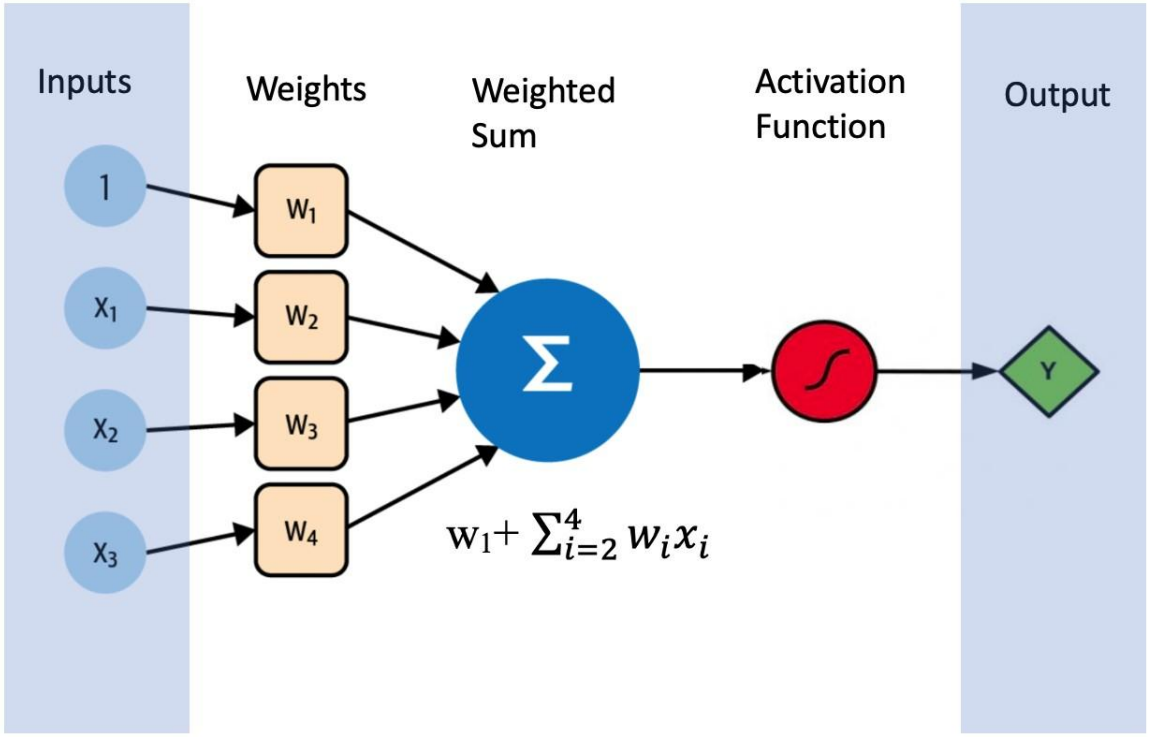


Рисунок 8.2: Персептрон

Перцептрон, показанный на рисунке 8.2, имеет три входных признака; x1, x2 и x3. Также добавляем постоянный сигнал, называемый смещением. Входные признаки и смещение умножаются на веса и суммируются в взвешенную сумму



Таким образом, взвешенная сумма передаётся активационной функции, которая генерирует выход y. Возможность использовать широкий спектр активационных функций для формулирования сложных связей между признаками и метками — одна из сильных сторон нейронных сетей. Разнообразие функций активации можно выбрать через гиперпараметры.

Понимание интуиции, лежащей в основе нейронных сетей

В последней главе мы обсудили некоторые традиционные алгоритмы машинного обучения. Эти традиционные алгоритмы машинного обучения отлично подходят для многих важных сценариев

применения. Но у них есть и ограничения. Когда подчёркиваемые шаблоны в обучающем наборе данных начинают становиться нелинейными и многомерными, они начинают выходить за рамки возможностей традиционных алгоритмов машинного обучения, чтобы точно отражать сложные взаимосвязи между признаками и меткой. Эти некомплексные, несколько упрощённые математические формулировки сложных паттернов приводят к неоптимальной производительности обученных моделей для этих сценариев использования. В реальной жизни во многих случаях связь между ярлыком и характеристиками проявляется более сложные закономерности. Нейронные сети приходят на помощь, предоставляя нам мощный вариант для обучения моделей с некоторыми ограничениями, обсуждаемыми позже в этой главе. Давайте разберёмся с некоторыми часто встречающимися закономерностями и проблемами, которые они создают для более простых алгоритмов машинного обучения, подобных используемому линейной регрессией. Предположим, что мы хотим предсказать зарплату дата-сайентиста, используя «годы, проведённые в образовании». У нас есть два разных набора данных, собранных из двух разных организаций.**Набор данных 1, рисунок 3(a)**: Показана относительно простая связь между признаком и меткой, демонстрирующей линейность. Но при попытке математически сформулировать её с помощью линейного алгоритма существует две сложности в этой простой схеме: во-первых, мы знаем, что зарплата не может быть отрицательной. Это значит, что независимо от значения x (то есть зарплаты), y не может быть отрицательным. Похоже, что есть как минимум один младший специалист по данным, который, возможно, только что окончил школу, проведя один год в образовании, и, соответственно, не имеет никакой зарплаты, возможно, проходя стажировку. Таким образом, для значений x от 0 до x1 значение y остаётся равным нулю. См. рис. 3(a). Как мы увидим позже, благодаря ректифицированной линейной активационной функции, доступной в нейронной сети, мы сможем захватывать такие сложные связи между меткой и признаком.**Набор данных 2, рисунок 3(b):** Он показывает нелинейность взаимосвязи между признаком и меткой, см. рис. 3(b). Заработная плата (y) остаётся равной 0, так как x изменяется от 0 до x1. Он растёт относительно резко по мере приближения значения x к x2. Но по мере увеличения значения y выше x2 значение y выравняется. Как мы увидим позже, благодаря функции активации сигмовидной системы мы сможем захватывать такие взаимосвязи, используя нейронные сети.

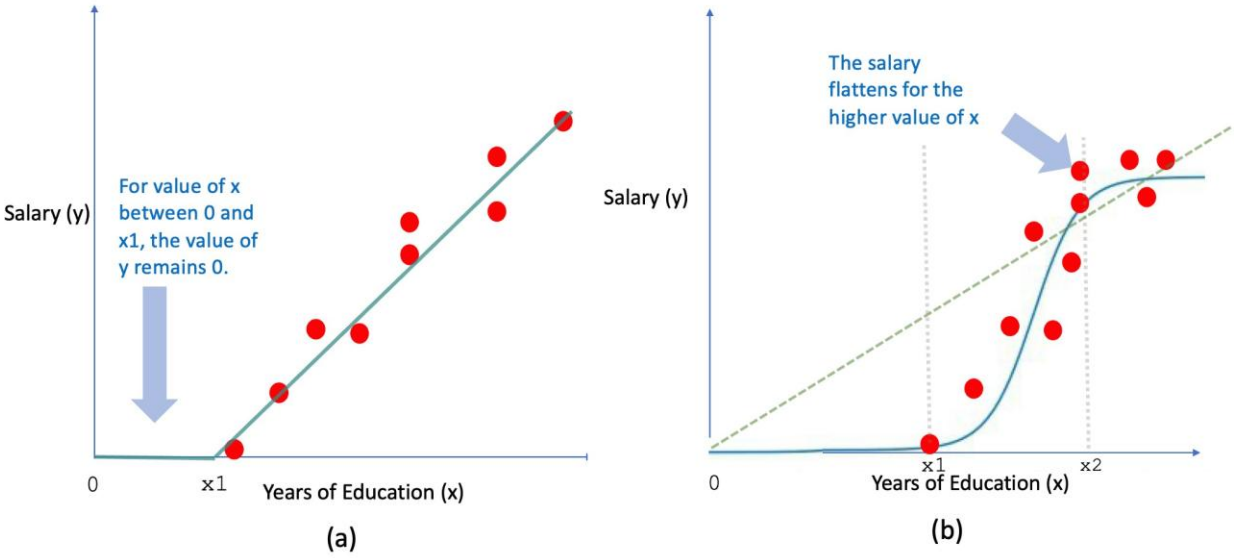


Рисунок 8.3: Зарплата ~ Годы образования

Набор данных 1: Линейные паттерны (b) Набор данных 2: Нелинейные паттерны

Понимание многоуровневых архитектур глубокого обучения

Для более сложных задач исследователи разработали многослойную нейронную сеть, называемую **многослойным перцептроном**. Многоуровневая нейронная сеть состоит из нескольких слоёв, как показано на следующей диаграмме. Эти слои следующие:

- **Входный слой:** первый слой — это входный слой. На входном уровне значения признаков подаются как входные данные в сеть.
- **Скрытые слои:** за входным слоем следует один или несколько скрытых слоёв. Каждый скрытый слой — это массивы аналогичной функции активации.
- **Выходной слой:** последний слой называется выходным слоем.

Простая нейронная сеть имеет один скрытый слой. Глубокая нейронная сеть — это нейронная сеть с двумя или более скрытыми слоями. См. рисунок (8.4)

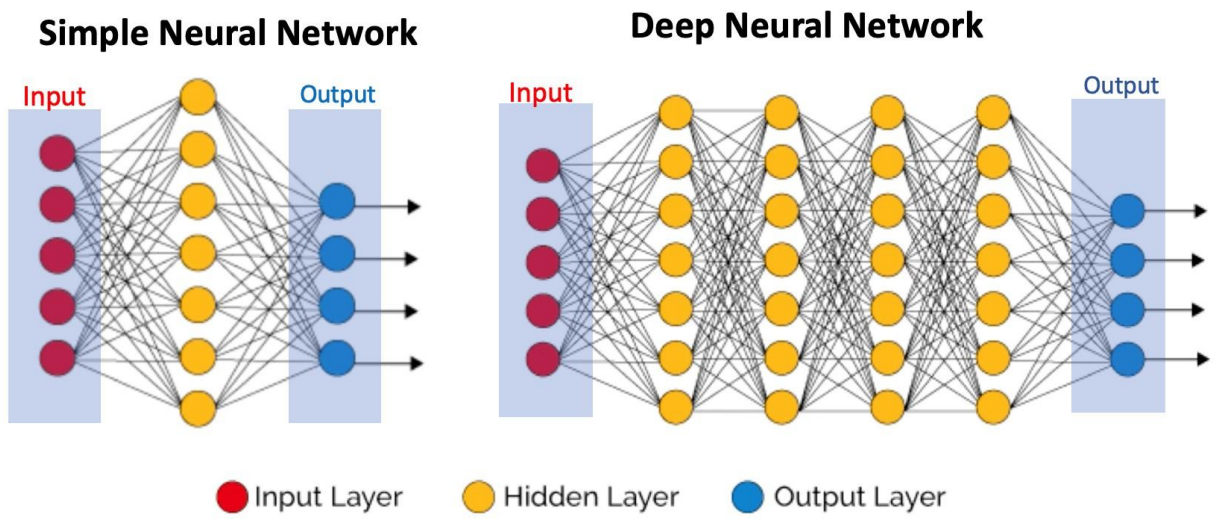


Рисунок 8.4: Простая нейронная сеть и глубокая нейронная сеть

Далее попробуем понять функцию скрытых слоёв.

Развитие интуиции для скрытых слоёв

Чтобы глубже понять, как скрытые слои объединяют и абстрируют входные признаки, давайте снова рассмотрим проблему прогнозирования зарплаты дата-сайентиста. Предположим, что у нас есть пять входных характеристик: 1) Опыт работы 2) Годы образования 3) Навыки SQL, 4) Навыки программирования и 5) Алгоритмические Knowledge.То понимание интуиции, лежащую в основе скрытых слоёв, которые мы создали на скрытом уровне с тремя узлами. Вход в первый узел скрытого слоя — это опыт работы и годы образования. Верхний узел скрытого слоя предоставит один узел, который абстрагирует эффект комбинации этих двух переменных. Мы назвали его «Background». Входные данные в средний узел скрытых слоёв — это SQL Skills и Programming Skills, которые называются Data Engineering Skills. Третий узел скрытого слоя содержит входные данные, такие как навыки программирования и алгоритмические знания, и называется навыками машинного обучения.

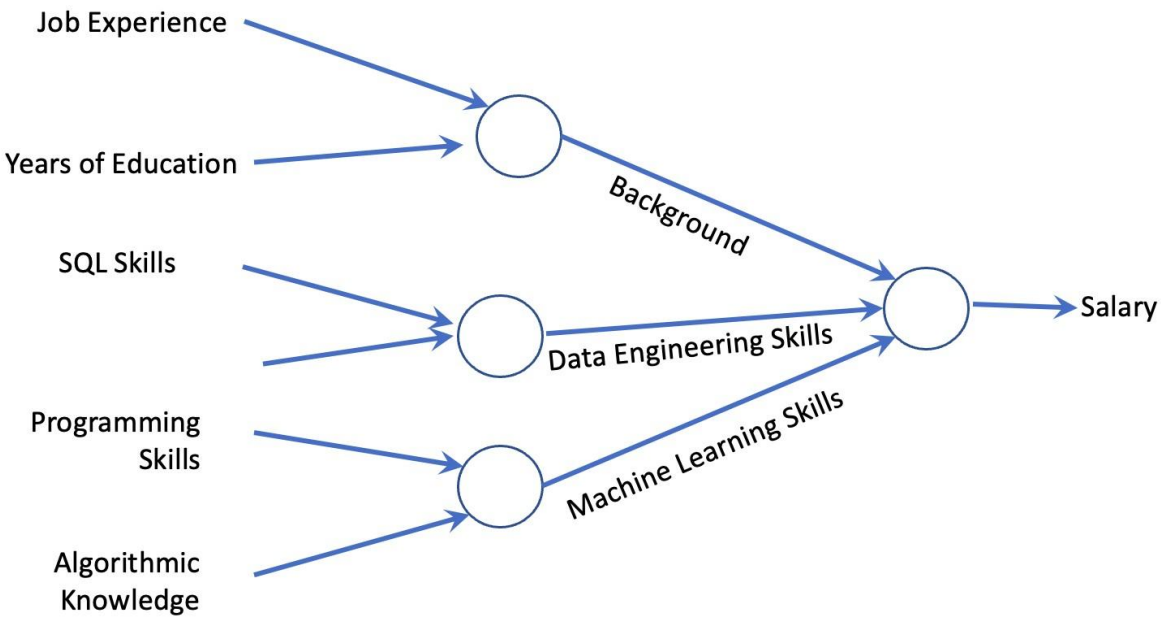


Рисунок 8.5: Решение задач с использованием нескольких слоёв

Сколько скрытых слоёв следует использовать?

Обратите внимание, что оптимальное количество скрытых слоёв будет варьироваться в зависимости от задачи. Для некоторых задач следует использовать однослойные нейронные сети. Для других стоит добавить несколько слоёв для лучшей производительности. В целом это зависит от сложности присущих закономерностей данных, которые мы пытаемся сформулировать с помощью нейронных сетей. Для задач с упрощёнными узорами использование большого количества скрытых слоёв приведёт к перенагону. С другой стороны, если речь идёт о сложных задачах, небольшое количество скрытых слоёв может не сможет полностью охватить и сформулировать закономерности, приводящие к недооборудованной модели. Тщательный анализ проблемы и эксперименты с базовой экспериментальной моделью помогут нам найти нужное количество скрытых слоёв.

Математические основы нейронных сетей

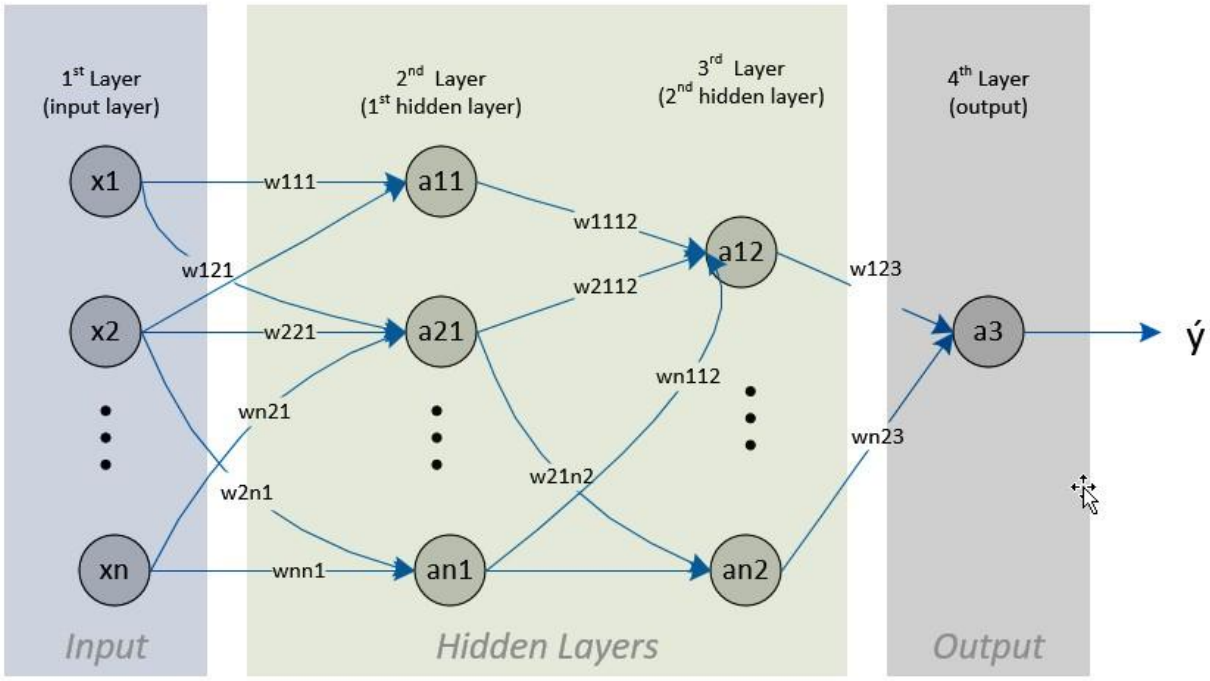
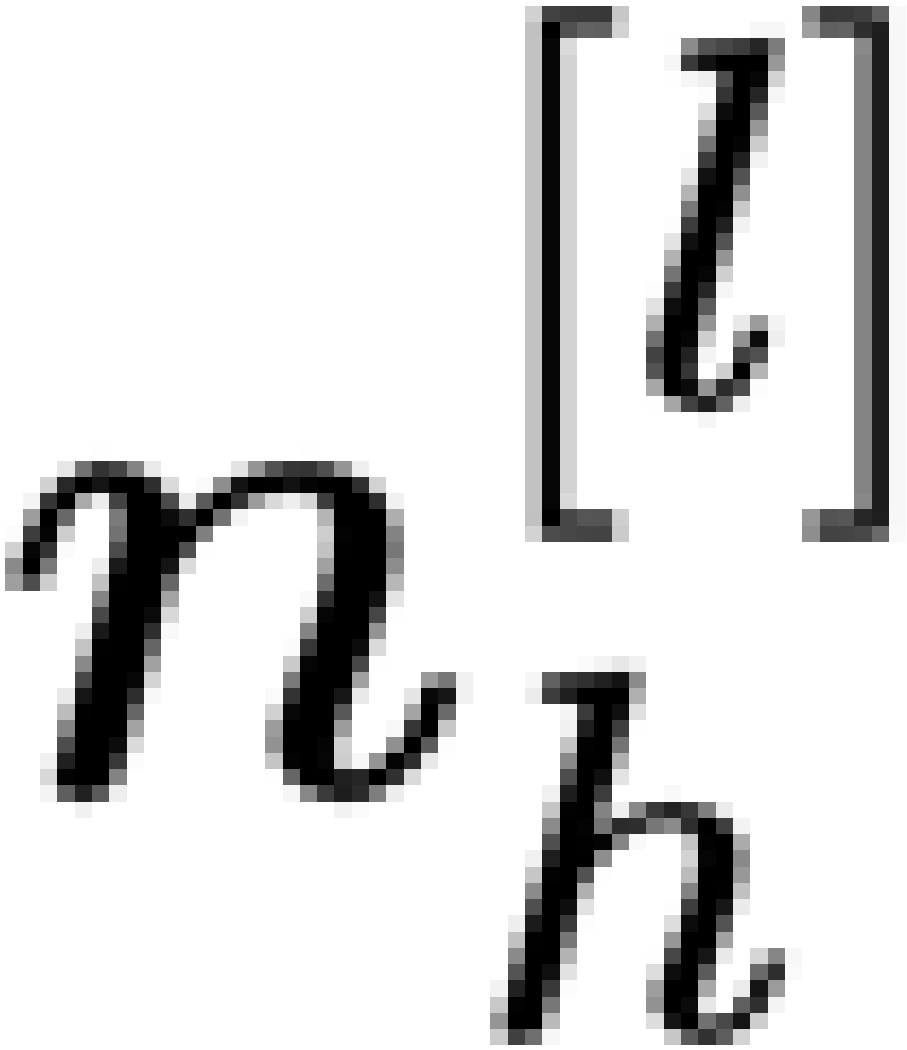


Рисунок 8.6: Многослойный перцептрон

Важно отметить, что нейрон является базовой единицей этой сети, и каждый нейрон слоя связан со всеми нейронами следующего слоя. Для сложных сетей количество этих взаимосвязей резко возрастает, и мы рассмотрим разные способы их сокращения, не жертвуя слишком большим качеством. Сначала давайте попробуем сформулировать задачу, которую мы пытаемся решить. Вход — это вектор признаков *x* размерности *n*. Мы хотим, чтобы нейронная сетьнейронная сеть предсказывала значения. Предсказанные значения представлены как *ý*. Математически мы хотим определить, учитывая конкретный вход, вероятность того, что транзакция является мошенническим. Другими словами, при определённом значении *x* какова вероятность того, что *y* = 1? Математически это можно представить следующим образом:

$$\acute{y} = P(y = 1|x) : where; x \in \Re^{n_x}$$

Обратите внимание, что x — nx-мерный вектор, где *nx* — количество входных переменных. Эта нейронная сеть имеет четыре слоя. Слои между входом и выходом — это скрытые слои. Количество нейронов в первом скрытом слое обозначается как



. Связи между различными узлами умножаются на параметры, называемые *весами*. Обучение нейронной сети — это поиск правильных значений для весов. Давайте посмотрим, как можно обучить нейронную сеть.

Обучение нейронной сети

Процесс построения нейронной сетевой сети с использованием заданного набора называется обучением нейронной сетевой нейронной сети. Давайте рассмотрим анатомию типичной нейронной сетевой сети. Когда мы говорим о тренировке нейронной сети, речь идёт о расчёте наилучших значений для весов. Обучение проводится итеративно, используя набор примеров в виде обучающих данных. Примеры в обучающих данных содержат ожидаемые значения выхода для различных комбинаций входных значений. Процесс обучения нейронных нейронных сетей отличается от способа обучения традиционных моделей (которые обсуждались в *главе 7*, *Традиционные алгоритмы контролируемого* обучения).

Понимание анатомии нейронной сети

Давайте посмотрим, из чего состоит нейронная сеть:

- **Слои:** Слои — это основные строительные блоки нейронной сетевой сети. Каждый слой — это модуль обработки данных, который выступает в роли фильтра. Он принимает один или несколько входов, обрабатывает их определённым образом и затем производит один или несколько выходов. Каждый раз, когда данные проходят через слой, они проходят фазу обработки и показывают закономерности, релевантные бизнес-вопросу, на который мы пытаемся ответить.
- **Функция потерь:** функция потерь обеспечивает сигнал обратной связи, используемый в различных итерациях процесса обучения. Функция потерь даёт отклонение для одного примера.
- **Функция затрат:** Функция затрат — это функция потерь на полном наборе примеров.
- **Оптимизатор:** оптимизатор определяет, как будет интерпретироваться сигнал обратной связи, предоставляемый функцией потерь.
- **Входные данные:** Входные данные используются для обучения нейронной сетевой сети. Он задаёт целевую переменную.

Веса: Веса рассчитываются с помощью тренировки сети. Веса примерно соответствуют важности каждого из входных данных. Например, если определённый вход важнее других, после тренировки ему присваивается большее значение веса, выступая в роли множителя. Даже слабый сигнал для этого важного входа набирает силу из большого значения веса (которое действует как множитель). Таким образом, вес в итоге поворачивает каждый вход в зависимости от их важности.

- **Функция активации:** значения умножаются на разные веса и затем агрегируются. Точное то, как они будут агрегироваться и как их значение будет интерпретироваться, будет определяться типом выбранной функции активации.

Давайте теперь рассмотрим очень важный аспект обучения нейронным сетям. При обучении нейронных сетей мы рассматриваем каждый пример по одному. Для каждого из примеров мы генерируем результат с помощью нашей модели недостаточного обучения. Мы рассчитываем разницу между ожидаемым и прогнозируемым результатом. Для каждого отдельного примера эта разница называется **потерей**. В совокупности потеря по всему учебному набору называется **стоимостью**. Продолжая обучение модели, мы стремимся найти правильные значения весов, которые приведут к минимальному значению потерь. В течение тренировки мы постоянно корректируем значения весов, пока не найдём набор значений для весов, который приведёт к минимальной возможной общей стоимости. Когда достигаем минимальной стоимости, мы отмечаем модель как обученную.

Определение градиентного спуска

Цель обучения модели нейронной сети — найти правильные значения для весов. Мы начинаем тренировать нейронную сеть с помощью случайных или стандартных значений весов. Затем мы итеративно используем алгоритм оптимизатора, например градиентный спуск, чтобы изменить веса так, чтобы наши прогнозы улучшались. Отправной точкой алгоритма градиентного спуска являются случайные значения весов, которые необходимо оптимизировать при прохождении алгоритма. В каждой из последующих итераций алгоритм действует путем изменения значений весов таким образом, чтобы минимизировать стоимость. Следующая диаграмма объясняет логику алгоритма градиентного спуска:

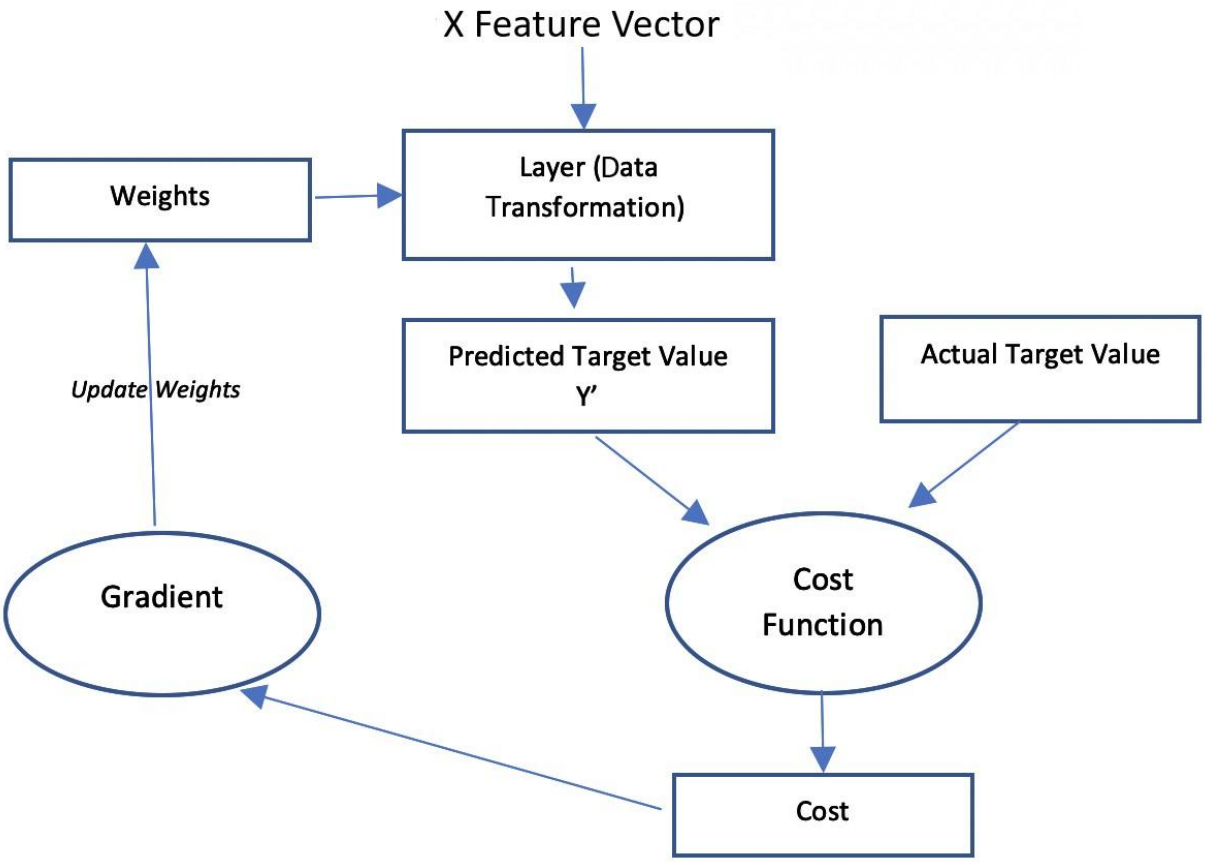


Рисунок 8.6: Алгоритм спуска по градиенту

На приведенной выше диаграмме входом является вектор признаков **X**. Фактическое значение целевой переменной равно **Y**, а предсказанное значение целевой переменной — **Y'**. Мы определяем отклонение фактического значения от предсказуемых. Мы обновляем веса и повторяем шаги, пока стоимость не будет минимизирована. То, как изменять вес в каждой итерации алгоритма, зависит от следующих двух факторов:

- **Направление:** В каком направлении двигаться, чтобы получить минимальную функцию потерь
- **Темп обучения:** насколько значительным должно быть изменение в выбранном направлении

Простой итеративный процесс показан на следующей диаграмме:

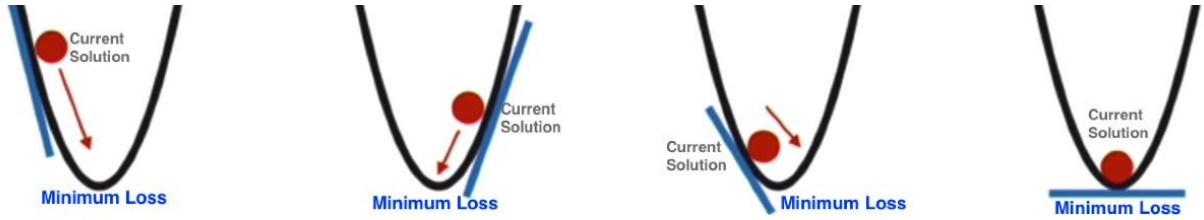


Рисунок 8.8: Алгоритм градиентного спуска, поиск минимумов

Диаграмма показывает, как, изменяя веса, градиентный спуск пытается найти минимальную стоимость. Скорость обучения и выбранное направление определят следующую точку на графике для изучения.

*Выбор правильного значения для скорости обучения важен. Если скорость обучения слишком мала, задача может долго сближаться. Если скорость обучения слишком высока, задача не сходится. На предыдущей диаграмме точка, представляющая наше текущее решение, будет колебаться между двумя противоположными линиями графа.*Теперь давайте посмотрим, как минимизировать градиент. Рассмотрим только две переменные: *x* и *y*. Градиент *x* и *y* вычисляется следующим образом:

$$gradient = \frac{\triangle y}{\triangle x}$$

Для минимизации градиента можно использовать следующий подход:

```
while(градиент!=0):
    если (градиент < 0); сдвиг
вправо если (градиент > 0); Двигайся
налево
```

Этот алгоритм также может использоваться для поиска оптимальных или близких к оптимальных значений весов нейронной сети. Обратите внимание, что расчёт градиентного спуска происходит обратно по всей сети. Мы начинаем с вычисления градиента последнего слоя, затем предпоследнего слоя, а затем предыдущего, пока не достигнем первого слоя. Это называется обратным распространением, которое было введено Хинтоном, Уильямсом и Румельхартом в 1985 году. Далее рассмотрим активационные функции.

Функции активации

Функция активации формулирует, как входные данные к конкретному нейрону будут обрабатываться для генерации выхода. Как показано на следующей диаграмме, каждый нейрон нейронной нейронной сети имеет активационную функцию, определяющую, как будут обрабатываться входные данные:

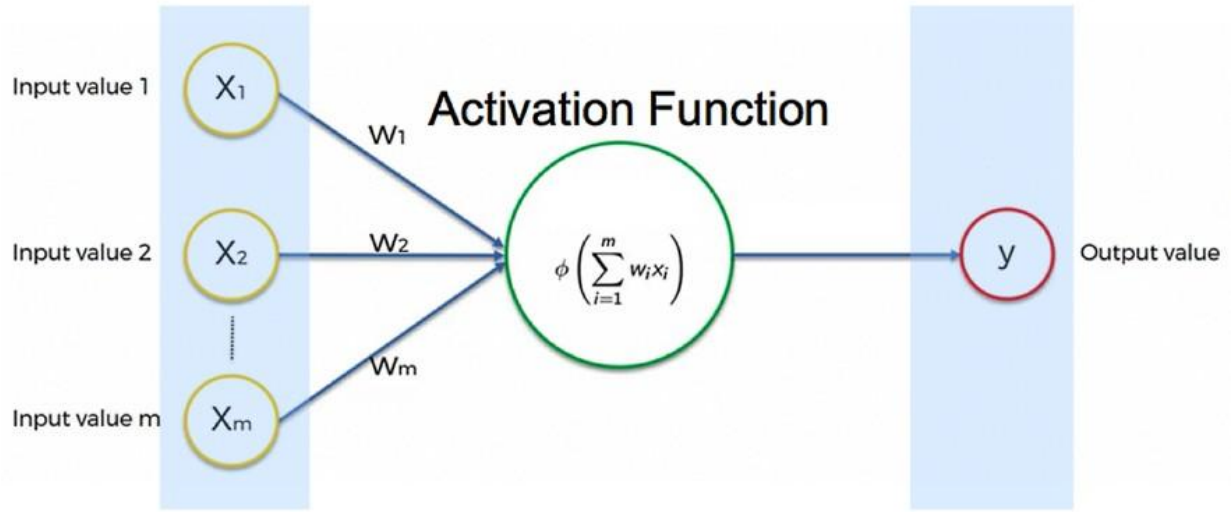


Рисунок 8.9: Функция активации

На приведённой выше диаграмме видно, что результаты, полученные активационной функцией, передаются на выход. Функция активации задаёт критерии интерпретации значений входных данных для генерации выхода. Для точно одних и тех же входных значений разные функции активации будут производить разные выходы. Понимание того, как выбрать правильную активационную функцию, важно при использовании нейронных сетей для решения задач. Теперь давайте рассмотрим эти функции активации по одной.

Шаговая функция

Самая простая возможная функция активации — это пороговая функция. Выход пороговой функции бинарен: 0 или 1. Он генерирует 1 как выход, если любой вход больше 1. Это можно объяснить на следующей схеме:

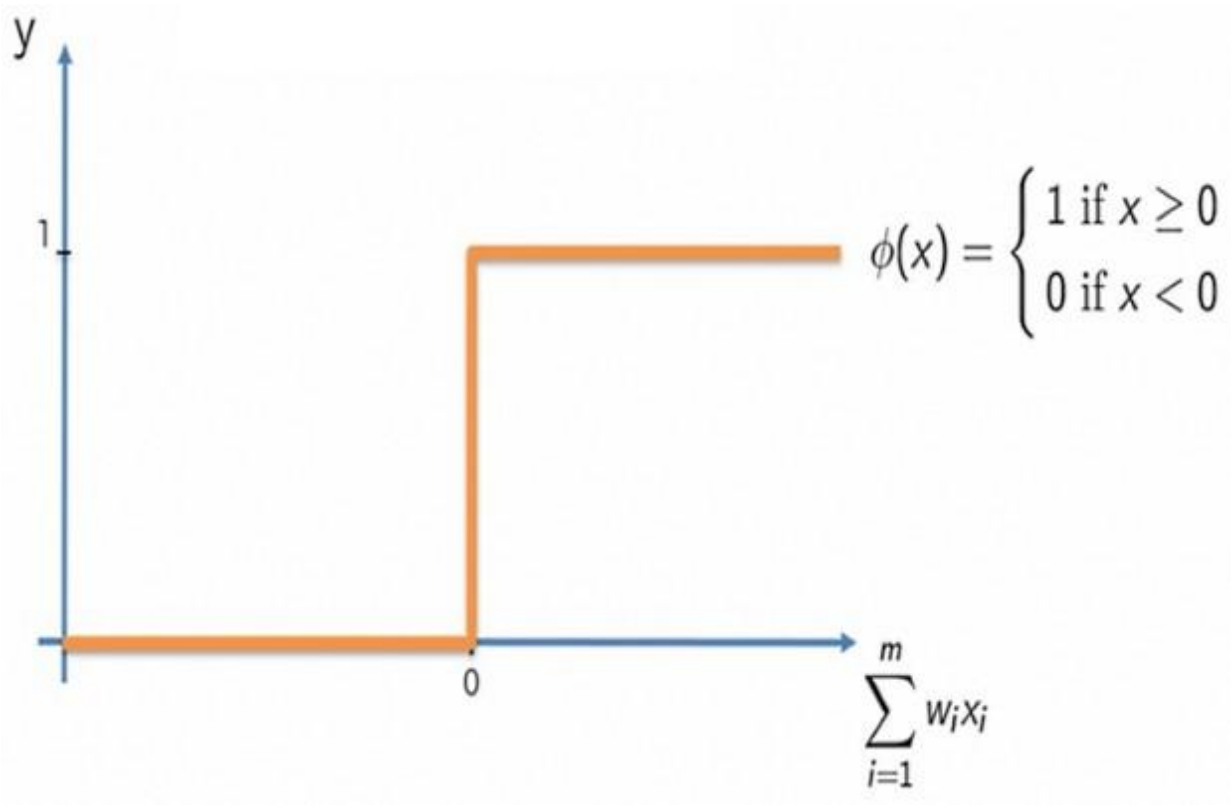


Рисунок 8.10: Шаговая функция

Обратите внимание, что как только в взвешенных суммах входов обнаруживаются признаки жизни, выход (*y*) превращается в 1. Это делает пороговую активационную функцию очень чувствительной. Он довольно уязвим к ошибочному срабатыванию при малейшем сигнале на входе из-за сбоя или шума.

Сигмоид

Сигмовидную функцию можно рассматривать как улучшение пороговой функции. Здесь мы контролируем чувствительность функции активации:

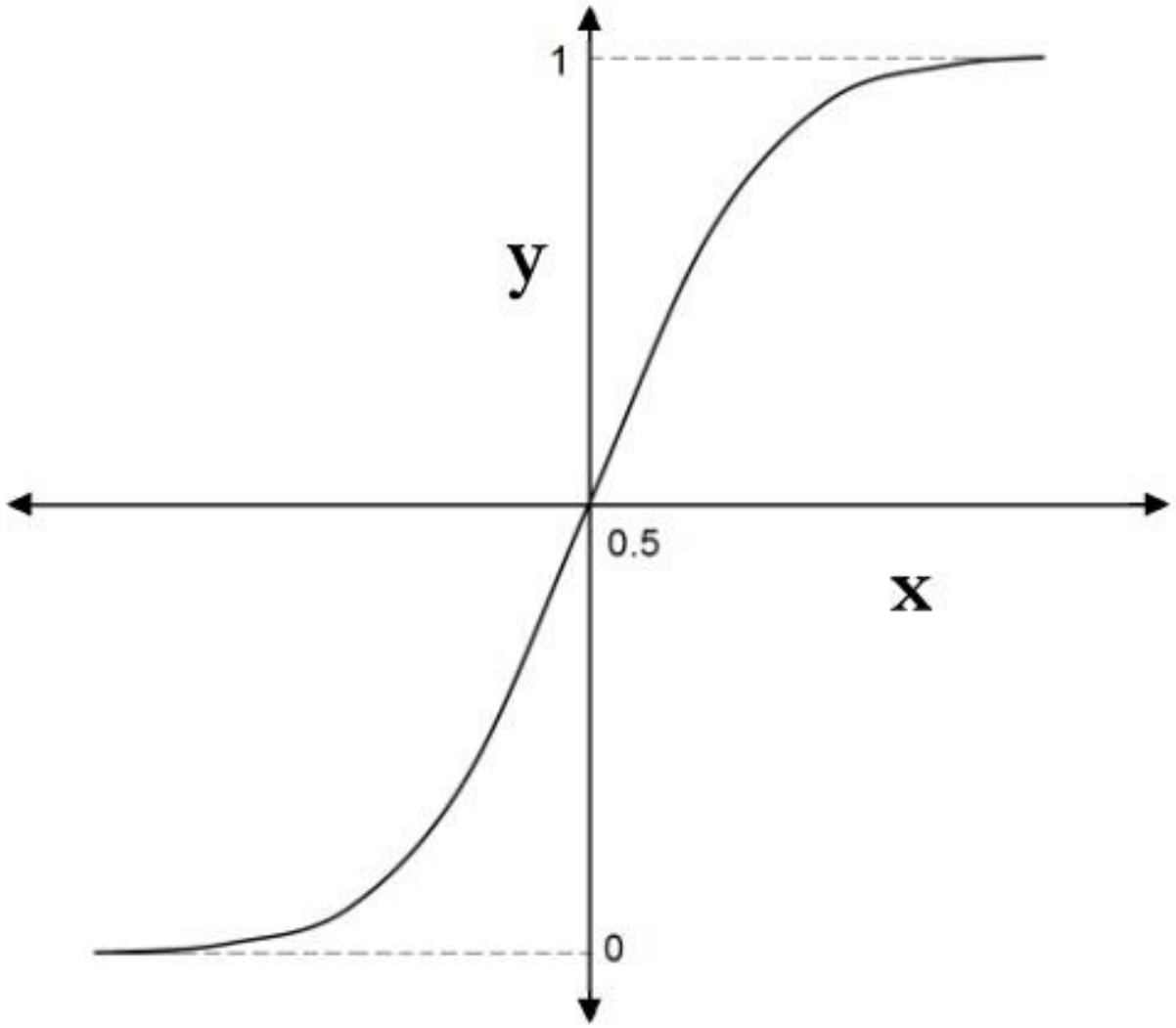


Рисунок 8.11: Функция активации сигмовидной формы

Сигмовидная функция y определяется следующим образом:

$$y = f(x) = \frac{1}{1 + e^{-x}}$$

Его можно реализовать на Python следующим образом:

```
def sigmoidFunction(z): return
1/ (1+np.exp(-z))
```

Обратите внимание, что снижая чувствительность функции активации, мы делаем сбои на входе менее нарушающими. Обратите внимание, что выход функции активации сигмовидной системы всё ещё бинарен, то есть 0 или 1.

Ректифицированная линейная единица (ReLU)

Вывод для первых двух функций активации, представленных в этой главе, был бинарным. Это значит, что они берут набор входных переменных и преобразуют их в двоичные выходы. ReLU — это функция активации, которая принимает набор входных переменных и преобразует их в один непрерывный выход. В нейронных сетях ReLU является самой популярной активационной функцией и обычно используется в скрытых слоях, где мы не хотим преобразовывать непрерывные переменные в категорические переменные. Следующая диаграмма обобщает функцию активации ReLU:

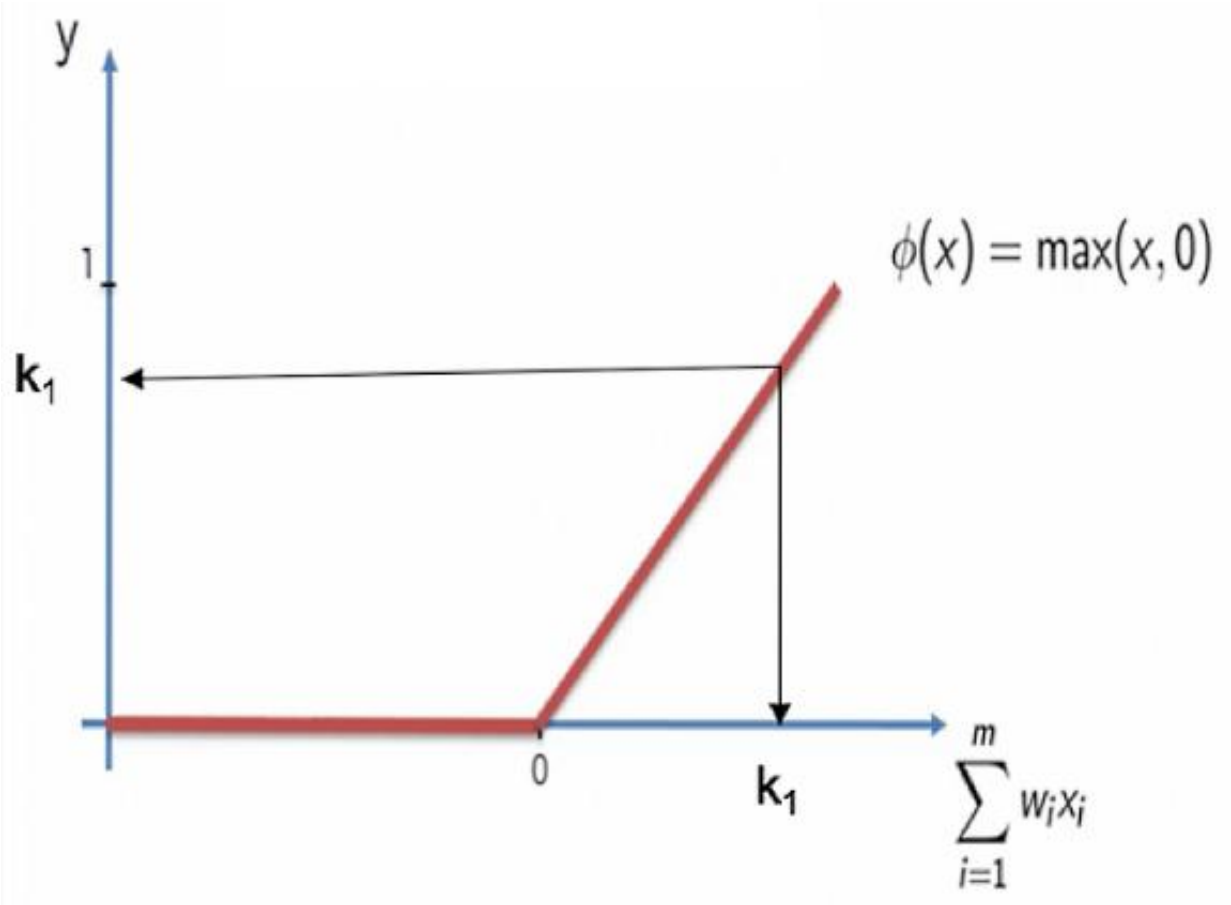


Рисунок 8.12: Ректифицированная линейная единица

Обратите внимание, что когда $x \leq 0$, это значит $y = 0$. Это означает, что любой сигнал от входа, равный нулю или меньше нуля, переводится в нулевый выход:

$$y = f(x) = 0;$$

$$x < 0$$

для

$$y = f(x) = x$$

для

$$x >= 0$$

Как *только x* становится больше нуля, он становится *x.* Функция ReLU является одной из самых часто используемых активационных функций в нейронных сетях. Его можно реализовать на Python следующим образом:

```
def ReLU(x):
если x<0:
    Вернуть 0
иначе:
вернуть x
```

Теперь давайте рассмотрим Leaky ReLU, который основан на ReLU.

Дырявый ReLU

В ReLU отрицательное значение для *x* даёт нулевое значение для *y*. Это означает, что часть информации теряется в процессе, что увеличивает циклы обучения, особенно в начале. Функция активации Leaky ReLU решает эту проблему. Для Leaky ReLu применимо следующее:

$$y = f(x) = ßx$$

$$x < 0$$

; для

$$y = f(x) = x$$

$$x >= 0$$

для

Это показано на следующей диаграмме:

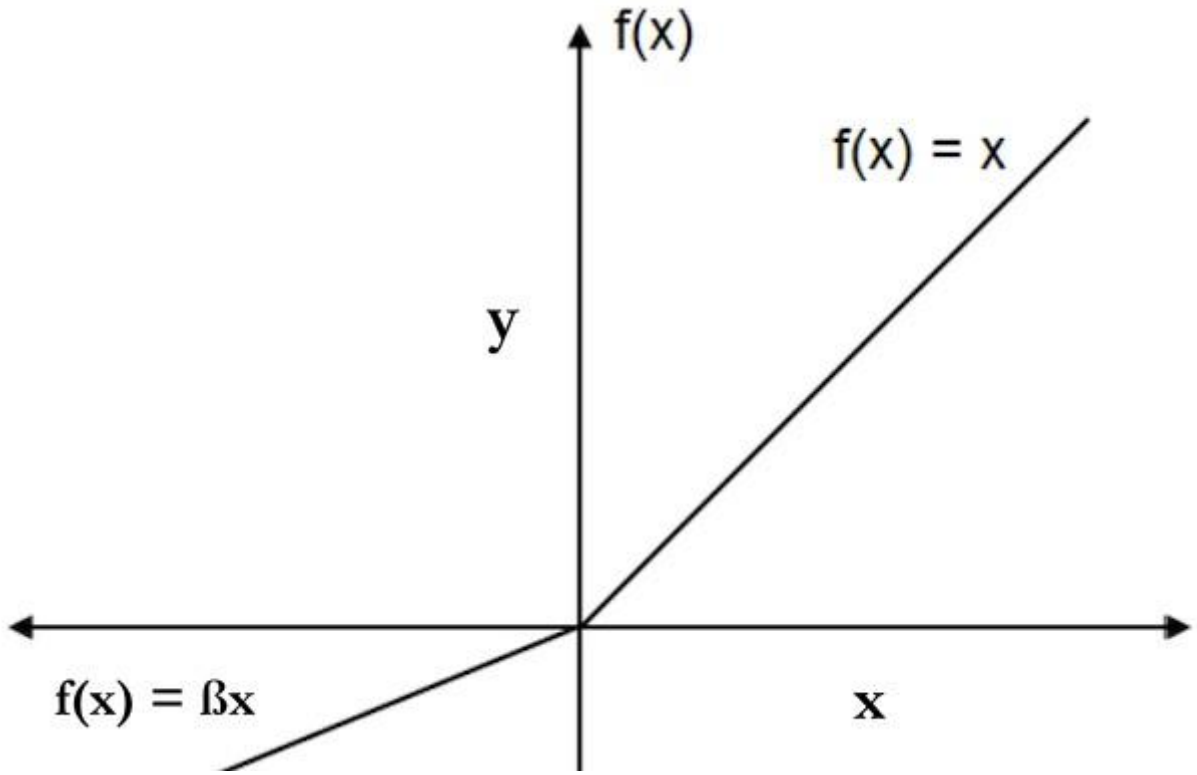


Рисунок 8.13: Дырявый ReLu

Здесь *ß* — параметр со значением меньше единицы. Его можно реализовать на Python следующим образом:

```
def leakyReLU(x,beta=0.01):
если x<0:
        Return (beta*x)
В другом случае: Return
x
```

Существует три способа указания значения для ß:

- Можно задать значение по умолчанию — ß.
- Мы можем сделать ß параметром в нашей нейронной сетевой сети и позволить нейронной сети определить значение (это называется **параметрическим ReLU**).
- Мы можем сделать ß случайным значением (это называется **рандомизированным ReLU).**

Гиперболический тангенс (tanh)

Функция Тань похожа на сигмовидную функцию, но способна также давать отрицательный сигнал. Следующая схема иллюстрирует это:

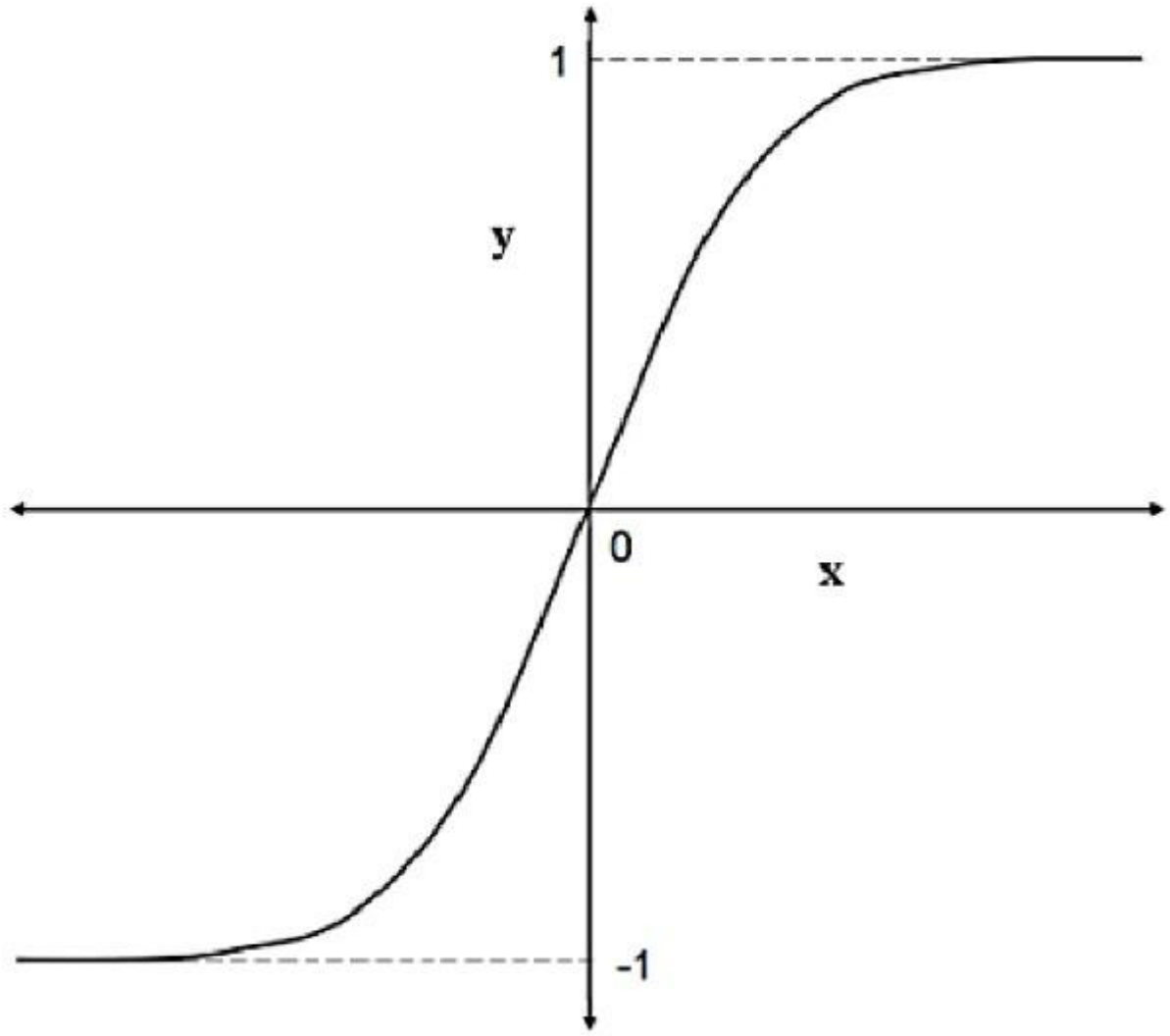


Рисунок 8.14: Гиперболическая касательная

Функция y выглядит следующим образом:

$$y = f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Его можно реализовать следующим кодом на Python:

```
Def Tanh(x):
    Числитель   =  1-np.exp(-2*x)
знаменатель    =   1+np.exp(-2*x)
```

возвращает числитель/ знаменатель Теперь рассмотрим функцию softmax.

Softmax

Иногда для выхода активационной функции требуется более двух уровней. Softmax — это функция активации, которая предоставляет нам более двух уровней для выхода. Он лучше всего подходит для задач многоклассовой классификации. Предположим, что у нас есть *n* классов. У нас есть входные значения. Входные значения отображают классы следующим образом: *x = {x(1),x(2),.... x(n)}*Softmax работает на основе теории вероятностей. Выходная вероятность *класса eth* мягкого максимума рассчитывается следующим образом:

$$prob^{(s)} = \frac{e^{x^s}}{\sum_{i=1}^{n} e^{x^i}}$$

Для бинарных классификаторов функция активации в последнем слое будет сигмоидной, а для мультиклассовых классификаторов — softmax.

Инструменты и фреймворки

В этом разделе мы подробно рассмотрим фреймворки и инструменты для реализации нейронных сетей. Со временем было разработано множество различных фреймворков для реализации нейронных сетей. У разных фреймворков есть свои сильные и слабые стороны. В этом разделе мы сосредоточимся на Keras с бэкендом TensorFlow.

Керас

Keras — одна из самых популярных и простых в использовании библиотек нейронных сетей, написана на Python. Он был написан с учётом удобства использования и предоставляет самый быстрый способ внедрения глубокого обучения. Keras предоставляет только высокоуровневые блоки и рассматривается на уровне модели.

Задние двигатели Keras

Keras нужна библиотека низкоуровневого глубокого обучения для выполнения тензорных манипуляций. Эта библиотека низкоуровневого глубокого обучения называется *бэкенд-движком*. Возможные бэкендные движки для Keras включают следующие:

- **TensorFlow** (https://www.tensorflow.org): Это самый популярный фреймворк такого рода, открытый Google.
- **Теона** (`deeplearning.net/software/theano`): Это было разработано в лаборатории MILA в Университете Монреаля.
- **Microsoft Cognitive Toolkit** (**CNTK**): Это было разработано компанией Microsoft.

Формат этого модульного стека технологий глубокого обучения показан на следующей диаграмме:

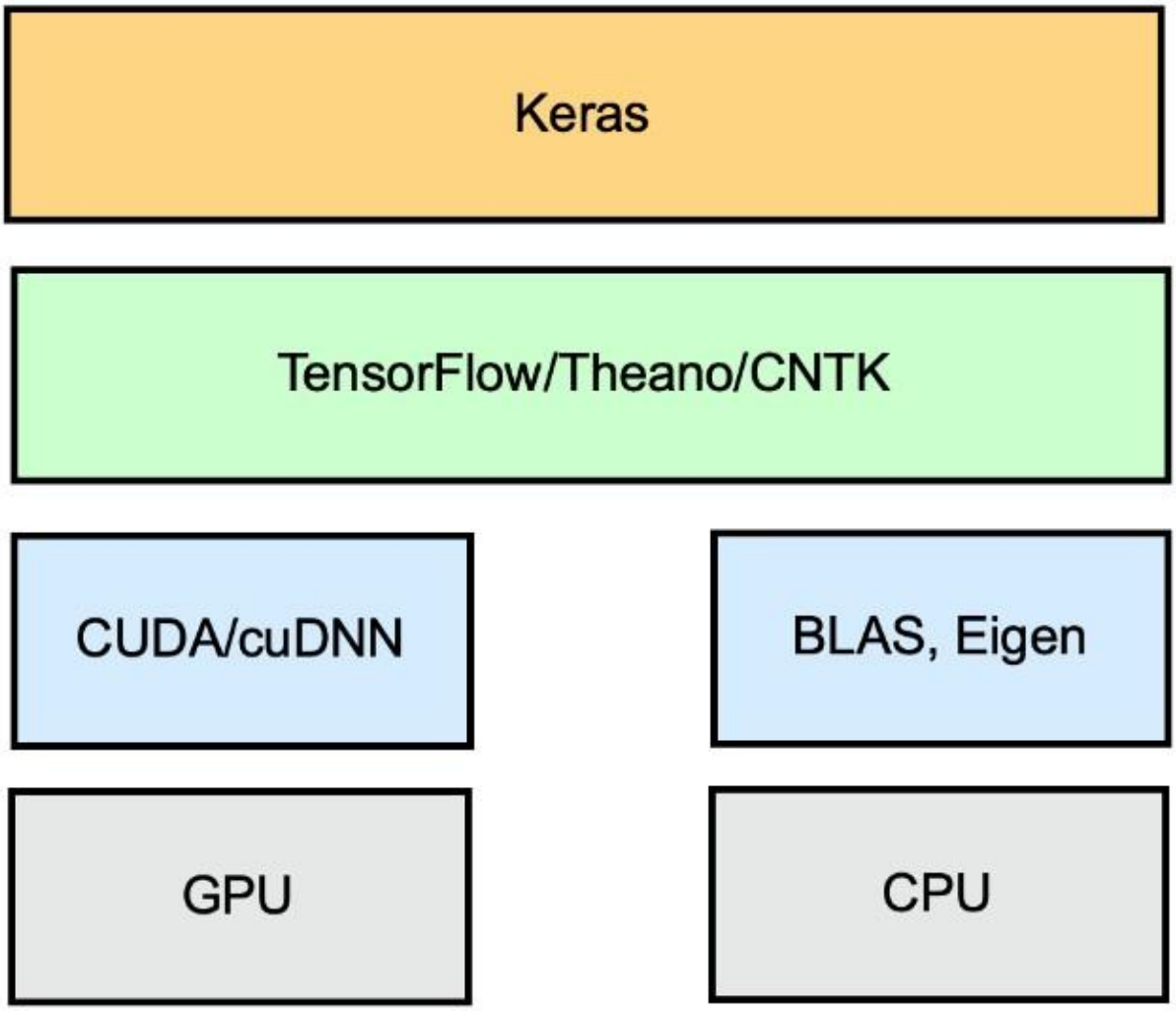


Рисунок 8.15: Архитектура Кераса

Преимущество такой модульной архитектуры глубокого обучения в том, что бэкенд Keras можно менять без переписывания кода. Например, если мы считаем, что TensorFlow лучше Theona для конкретной задачи, мы можем просто изменить бэкенд на TensorFlow без переписывания кода.

Низкоуровневые слои стека глубокого обучения

Три упомянутых нами бэкенд-движка могут работать как на процессорах, так и на GPU, используя низкоуровневые слои стека. Для процессоров используется низкоуровневая библиотека тензорных

операций, называемая **Eigen**. Для GPU TensorFlow использует библиотеку **CUDA Deep Neural** Network (**cuDNN**) от NVIDIA.

Определение гиперпараметров

Как обсуждается в *главе 6* «*Алгоритмы неконтролируемого машинного обучения*», гиперпараметр — это параметр, значение которого выбирается до начала процесса обучения. Мы начинаем с разумных ценностей, а потом пытаемся их оптимизировать. Для нейронных сетей важные гиперпараметры следующие:

- Функция активации
- Скорость обучения
- Количество скрытых слоёв
- Количество нейронов в каждом скрытом слое

Давайте рассмотрим, как можно определить модель с помощью Keras.

Определение модели Кераса

Для определения полной модели Кераса состоит три этапа:

1. **Определите слои**.

Модель с использованием Keras можно построить двумя возможными способами:

- **Последовательный API:** Это позволяет нам проектировать модели для линейного стека слоёв. Он используется для относительно простых моделей и является обычным выбором для сборки моделей:

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'),
            ])
```

Обратите внимание, что здесь мы создали три слоя — первые два слоя имеют функцию активации ReLU, а третий — softmax в качестве функции активации.

- **Функциональный API:** Это позволяет нам создавать модели для ациклических графов слоёв. Более сложные модели можно создавать с помощью функционального API.

```
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)
```

Обратите внимание, что мы можем определить одну и ту же нейронную сеть, используя как последовательные, так и функциональные API. С точки зрения производительности не имеет значения, какой подход вы выбираете для определения модели.

1. **Определите процесс обучения**.

На этом этапе мы определяем три вещи:

- Оптимизатор
- Функция потерь
- Метрики, которые количественно оценят качество модели:

```
optimiser = tf.keras.optimizers.RMSprop
model.compile (optimizer= optimiser, loss='mse', metrics = ['accuracy'])
```

Note that we use the model.compile function to define the optimizer, loss function, and metrics.

1. **Train the model**.

Once the architecture is defined, it is time to train the model:

```
model.fit(x_train, y_train, batch_size=128, epochs=10)
```

Note that parameters such as batch_size and epochs are configurable parameters, making them hyperparameters.

Choosing sequential or functional model

The sequential model creates the neural network as a simple stack of layer. The sequential model is easy and straightforward to understand and implement, but its simplistic architecture also has a major restriction. Each layer is connected to exactly one input and output tensor. This means that if our model has multiple inputs or multiple outputs either at the input or output or at any of the hidden layers, then we cneural networkot use a sequential model. In this case, we will have to use the functional model.

Understanding TensorFlow

TensorFlow is one of the most popular libraries for working with neural neural networks. In the preceding section, we saw how we can use it as the backend engine of Keras. It is an open source, high-performance library that can actually be used for any numerical computation. If we look at the stack, we can see that we can write TensorFlow code in a high-level language such as Python or C++, which gets interpreted by the TensorFlow distributed execution engine. This makes it quite useful for and popular with developers.The way TensorFlow works is that you create a **Directed Graph** (**DG**) to represent your computation. Connecting the nodes are the edges, the input, and the output of the mathematical operations. Also, they represent arrays of data.

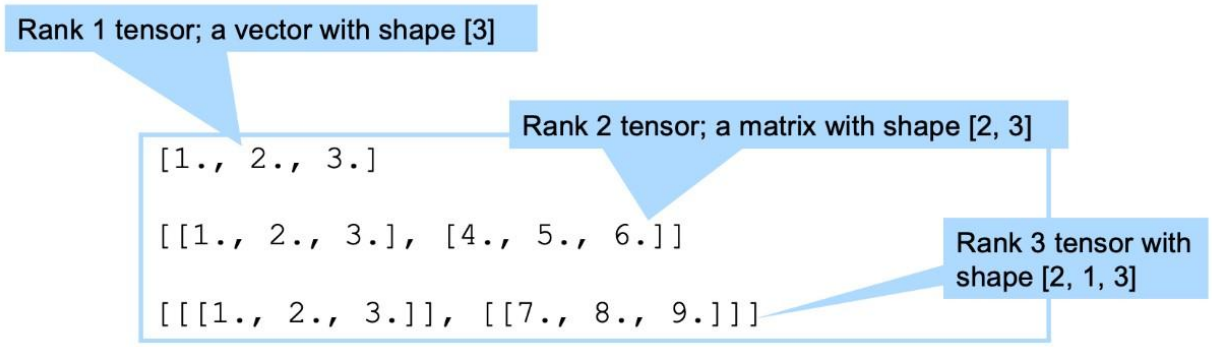
Presenting TensorFlow's Basic Concepts

Let's take a brief look at TensorFlow concepts such as scalars, vectors, and matrices. We know that a simple number, such as three or five, is called a **scalar** in traditional mathematics. Moreover, in physics, a **vector** is something with magnitude and direction. In terms of TensorFlow, we use a vector to mean one-dimensional arrays. Extending this concept, a two-dimensional array is a **matrix**. For a three-dimensional array, we use the term **3D tensor**. We use the term **rank** to capture the dimensionality of a data structure. As such, a **scalar** is a **rank 0** data structure, a **vector** is a **rank 1** data structure, and a **matrix** is a **rank 2** data structure. These multi-dimensional structures are known as **tensors** and are shown in the following diagram:



Figure 8.16: Multi-dimensional structures or tensors

As we can see in the preceding diagram, the rank defines the dimensionality of a tensor.Let's now look at another parameter, shape. shape is a tuple of integers specifying the length of an array in each dimension. The following diagram explains the concept of shape:



Using shape and rank, we can specify the details of tensors.

Understanding Tensor Mathematics

Let's now look at different mathematical computations using tensors:

- Let's define two scalars and try to add and multiply them using TensorFlow:

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
```

- We can add and multiply them and display the results:

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
```

- We can also create a new scalar tensor by adding the two tensors:

In [16]:
```
c = a + b
print("a + b = %s" % c)
```

```
a + b = Tensor("add:0", shape=(2, 2), dtype=float32)
```

- We can also perform complex tensor functions:

In [17]:
```
d = tf.matmul(a, b)
print("a * b = %s" % d)
```

```
a * b = Tensor("MatMul:0", shape=(2, 2), dtype=float32)
```

Understanding the Types of neural networks

There is more than one way that neuralneural networks can be built. If every neuron in each layer is connected to each of the neurons in another layer, then we call it a dense or fully connected neural networkneural network. Let's look at some other forms of neuralneural networks.

Convolutional neural networks

Convolution neural networks (**CNNs**) are typically used to analyze multimedia data. In order to learn more about how a CNN is used to analyze image-based data, we need to have a grasp of the following processes:

- Convolution
- Pooling

Let's explore them one by one.

Convolution

The process of convolution emphasizes a pattern of interest in a particular image by processing it with another smaller image called a **filter** (also called a **kernel**). For example, if we want to find the edges of objects in an image, we can convolve the image with a particular filter to get them. Edge detection can help us in object detection, object classification, and other applications. So, the process of convolution is about finding characteristics and features in an image.The approach to finding patterns is based on finding patterns that can be reused on different data. The reusable patterns are called filters or kernels.

Pooling

An important part of processing multimedia data for the purpose of machine learning is downsampling it. This provides two benefits:

- It reduces the overall dimensionality of the problem, decreasing the time needed to train the model in a major way.
- Through aggregation, we abstract the unnecessary details in the multimedia data, making it more generic and more representative of similar problems.

Downsampling is performed as follows:

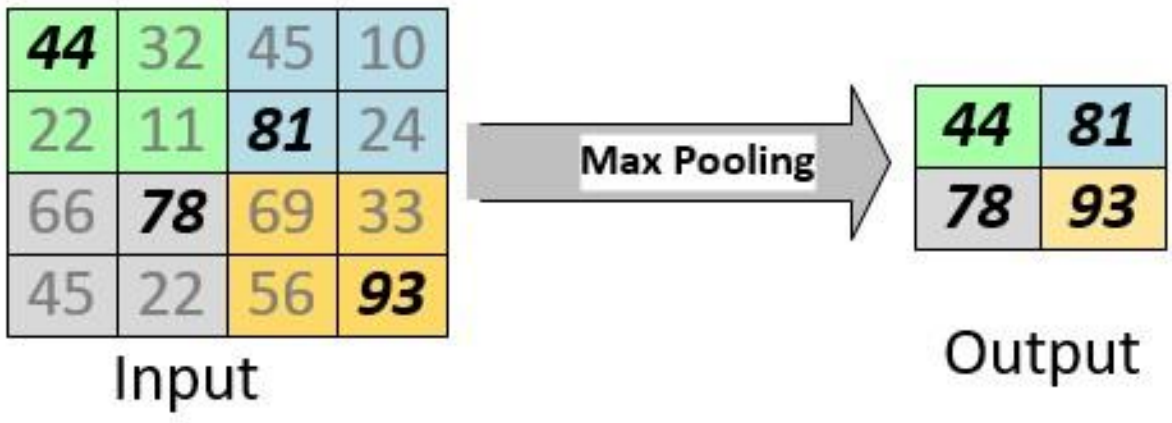


Figure 8.17: Downsampling

Note that we have replaced every block of four pixels with one pixel, choosing the highest value of the four pixels to be the value of that one pixel. This means that we have downsampled by a factor of four. As we have chosen the maximum value in each block, this process is called **max pooling**. We could have chosen the average value; in that case, it would be average pooling.

Recurrent neural networks

Recurrent neural networks (**RNNs**) are a special type of neuralneural network that are based on looped architecture. This is why they are called *recurrent*. The important thing to note is that RNNs have memory. This means that they have the ability to store information from recent iterations. They are used in areas such as analyzing sentence structures to predict the next word in a sentence.

Generative Adversarial Networks

Generative Adversarial Networks (**GANs**) are a type of neural networkneural network that generate synthetic data. They were created in 2014 by Ian Goodfellow and his colleagues. They can be used to generate photographs of people that have never existed. More importantly, they are used to generate synthetic data to augment training datasets.In the upcoming section, we will see what transfer learning is.

Transfer Learning

Over the years, many organizations, research groups, and individuals within the open source community have perfected some complex models trained using gigantic amounts of data for generic use cases. In some cases, they have invested years of effort in optimizing these models. Some of these open source models can be used for the following applications:

- Object detection in video
- Object detection in images
- Transcription for audio
- Sentiment analysis for text

Whenever we start working on training a new machine learning model, the question to ask ourselves is this: instead of starting from scratch, can we simply customize a well-established pre-trained model for our purposes? In other words, can we transfer the learning of existing models to our custom model so that we can answer our business question? If we can do that, it will provide three benefits:

- Our model training efforts will be given a jump-start.
- By using a well-tested and well-established model, the overall quality of our model is likely to improve. If we do not have enough data for the problem we are working on, using a pre-trained model through transfer learning may help.

Let's look into two actual examples where this would be useful:

- When training a robot, we could first use a simulation game to train a neural networkneural network model. In that simulation, we could create all those rare events that are very hard to find in the real world. Once trained, we could use transfer learning to train the model for the real world.
- Let's assume that we want to train a model that can classify Apple And Windows laptops from a video feed. There are already well-established object detection models available as open source that can accurately classify various objects in a video feed. We can use these models as a starting point and identify objects as

laptops. Once we have identified the objects as laptops, we can further train the model to differentiate between Apple and Windows laptops.

In the next section, we will apply the concepts that we have covered in this chapter to building a fraudulent document classifying neuralneural network.

Case study – using deep learning for fraud detection

Using **Machine Learning** (**ML**) techniques to identify fraudulent documents is an active and challenging field of research. Researchers are investigating to what extent the pattern recognition power of neural neural networks can be exploited for this purpose. Instead of manual attribute extractors, raw pixels can be used for several deep learning architectural structures.

Methodology

The technique presented in this section uses a type of neural networkneural network architecture called **Siamese neuralneural networks**, which features two branches that share identical architectures and parameters. The use of Siamese neural neural networks to flag fraudulent documents is shown in the following diagram:

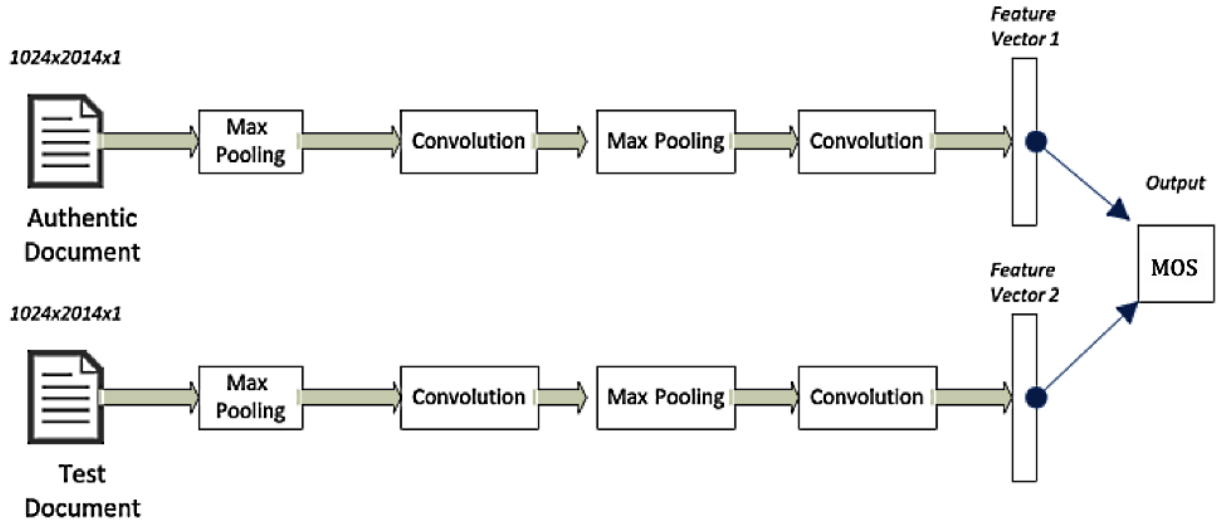


Figure 8.18: Siamese neural networks

When a particular document needs to be verified for authenticity, we first classify the document based on its layout and type, and then we compare it against its expected template and pattern. If it deviates beyond a certain threshold, it is flagged as a fake document; otherwise, it is considered an authentic or true document. For critical use cases, we can add a manual process for borderline cases where the algorithm cneural networkot conclusively classify a document as authentic or fake.To compare a document against its expected template, we use two identical CNNs in our Siamese architecture. CNNs have the advantage of learning optimal shift-invariant local feature detectors and can build representations that are robust to geometric distortions of the input image. This is well suited to our problem since we aim to pass authentic and test documents through a single network, and then compare their outcomes for similarity. To achieve this goal, we implement the following steps.Let's assume that we want to test a document. For each class of document, we perform the following steps:

A. Get the stored image of the authentic document. We call it the **true document**. The test document should look like the true document.

Б. The true document is passed through the neural networkneural network layers to create a feature vector, which is the mathematical representation of the patterns of the true document. We call it F**eature Vector 1**, as shown in the preceding diagram.

В. The document that needs to be tested is called the **test document**. We pass this document through a neuralneural network similar to the one that was used to create the feature vector for the true document. The feature vector of the test document is called F**eature Vector 2**.

Г. We use the Euclidean distance between feature vector 1 and feature vector 2 to calculate the similarity score between the true document and the test document. This similarity score is called the **Measure Of Similarity** (**MOS**). The MOS is a number between 0 and 1. A higher number represents a lower distance between the documents and a greater likelihood that the documents are similar.

Д. If the similarity score calculated by the neural networkneural network is below a pre-defined threshold, we flag the document as fraudulent.

Let's see how we can implement Siamese neural neural networks using Python:

1. First, let's import the Python packages that are required:

```
import random import
numpy as np import
tensorflow as tf
```

1. Next, we will define the neuralneural network that will be used to process each of the branches of the Siamese network:

```
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'),
])
```

Note that, in order to reduce overfitting, we have also specified a dropout rate of 0.15.

1. To implement Siamese networks, we will use MNIST images. MNIST images are ideal for testing the effectiveness of our approach. Our approach entails preparing the data in such a way that each sample will have two images and a binary similarity flag. This flag is an indicator that they are from the same class. Let's now implement the function named prepareData, which can prepare the data for us:

```
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)
```

Note that prepareData() will result in an equal number of samples across all digits.

1. We will now prepare the training and testing datasets:

```
(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)
```

1. Now, let's create the two halves of the Siamese system:

```
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)
```

1. Now, we will implement the **measure of similarity** (**MOS**), which will quantify the distance between two documents that we want to compare:

```
distance = tf.keras.layers.Lambda(
    lambda embeddings: tf.keras.backend.abs(embeddings[0] - embeddings[1])) ([enconder1, enconder
measureOfSimilarity = tf.keras.layers.Dense(1, activation='sigmoid') (distance)
```

Now, let's train the model. We will use 10 epochs to train this model:

```
[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>
```

Note that we reached an accuracy of 97.49% using 10 epochs. Increasing the number of epochs will further improve the level of accuracy.

Summary

In this chapter, we first looked at the details of neural neural networks. We started by looking at how neural neural networks have evolved over the years. We studied different types of neural neural networks. Then, we looked at the various building blocks of neural neural networks. We studied in depth the gradient descent algorithm, which is used to train neural neural networks. We discussed various activation functions and studied the applications of activation functions in a neural networkneural network. We also looked at the concept of transfer learning. Finally, we looked at a practical example of how a neural networkneural network can be used to train a machine learning model that can be deployed to flag forged or fraudulent documents.Looking ahead, in

the next chapter, we will look into how we can use such algorithms for natural language processing. We will also introduce the concept of web embedding and will look into the use of recurrent networks for natural language processing. Finally, we will also look into how to implement sentiment analysis.

OceanofPDF.com

Section 3: Advanced Topics

Join our book's Discord space

https://packt.link/40Algos



As the name suggests, we will deal with some selected advanced topics related to algorithms in this section. Cryptography and large scale algorithms are key highlights of this section. We will also look into the issues related to large scale infrastructure needed to train complex algorithm. The last chapter of this section explores the practical considerations one should keep in mind while implementing algorithms. The chapters included in this section are:

- *Chapter 12*, *Data Algorithms*
- *Chapter 13*, *Cryptography*
- *Chapter 14*, *Large Scale Algorithms Chapter*
- *15*, *Practical Considerations*

12 Data Algorithms

"Data is the New Oil of the Digital Economy": Wired Magazine

decision making around the world is increasingly data driven. Data captures the information related to phenomenon and algorithms can help us extract and utilize this in a meaningful way. This chapter is about such data-centric algorithms and, in particular, it focuses on three aspects of data-centric algorithms: storage, streaming, and compression. This chapter starts with a brief overview of data-centric algorithms, then we will discuss various strategies that can be used in data storage. Next, how to apply algorithms to streaming data is described, then different methodologies to compress data are discussed. Finally, we will learn how we can use the concepts

developed in this chapter to monitor the speeds of cars traveling on a highway using a state-of-the-art sensor network.By the end of this chapter, you should be able to understand the concepts and trade-offs involved in the design of various data-centric algorithms.This chapter discusses the following concepts:

- Data classification
- Data storage algorithms
- How to use algorithms to compress data
- How to use algorithms to stream data

Let's first introduce the basic concepts.

Introduction to data algorithms

Whether we realize it or not, we are living in an era of big data. Just to get an idea about how much data is constantly being generated, just look into some of the numbers published by Google for 2019. As we know, Google Photos is the multimedia repository for storing photos created by Google. In 2019, an average of 1.2 billion photos and videos were uploaded to Google Photos every day. Also, an average of 400 hours of video (amounting to 1 PB of data) were uploaded every minute each day to YouTube. We can safely say the amount of data that is being generated has simply exploded.The current interest in data-driven algorithms is driven by the fact that data contains valuable information and patterns. If used in the right way, data can become the basis of policy-making decisions, marketing, governance, and trend analysis.For obvious reasons, algorithms that deal with data are becoming more and more important. Designing algorithms that can efficiently process data is an active area of research. There is no doubt that exploring the best ways to use data to provide some quantifiable benefit is the focus of various organizations, businesses, and governments all over the world. But data in its raw form is seldom useful. To mine the information from the raw data, it needs to be processed, prepared, and analyzed.For that, we need to find a strategy to store it. Efficient methodologies to store data are becoming more and more important. Note that due to the physical storage limitations of single-node systems, big data can only be stored in distributed storage consisting of more than one node connected by high-speed communication links. So, it makes sense that, for learning data algorithms, we start by looking at different data storage algorithms.First, let's classify data into various categories.

Data classification

Let's look into how we can classify data in the context of designing data algorithms. As discussed in *Chapter 2, Data Structures Used in Algorithms*, quantifying the volume,

variety, and velocity of the data can be used to classify it. This classification can become a basis to design data algorithms that can be used for its storage and processing.Let's look into these characteristics one by one in the context of data algorithms:

- **Volume** quantifies the amount of data that needs to be stored and processed in an algorithm. As the volume increases, the task becomes data-intensive and requires provisioning enough resources to store, cache, and process data. **Big data** is a term that vaguely defines a large volume of data that cannot be handled by a single node.
- **Velocity** defines the rate at which new data is being generated. Usually, highvelocity data is called **hot data** or a **hot stream** and low-velocity data is called a **cold stream** or simply **cold data**. In many applications, data will be a mix of hot and cold streams that will first need to be prepared and combined into a single table before it can be used with the algorithm.
- **Variety** refers to different types of structured and unstructured data that needs to be combined into a single table before it can be used by the algorithm.

The next section will help us to understand the trade-offs involved and will present various design choices when designing storage algorithms.

Presenting data storage algorithms

A reliable and efficient data repository is the heart of a distributed system. If this data repository is created for analytics, then it is also called a **data lake**. A data repository brings together data from different domains into a single location. Let's start with first understanding different issues related to the storage of data in a distributed repository.

Understanding data storage strategies

In the initial years of digital computing, the usual way of designing a data repository was by using a single node architecture. With the ever-increasing sizes of datasets, distributed storage of data has now become mainstream. The right strategy to store data in a distributed environment depends on the type of data and its expected usage pattern as well as its non-functional requirements. To further analyze the requirements of a distributed data store, let's start with the **Consistency Availability PartitionTolerance** (**CAP**) theorem, which provides us with the basis of devising a data storage strategy for a distributed system.

Presenting the CAP theorem

In 1998, Eric Brewer proposed a theorem that later became famous as the CAP theorem. It highlights the various trade-offs involved in designing a distributed storage system.To understand the CAP theorem, first, let's define the following three characteristics of distributed storage systems: consistency, availability, and partition tolerance. CAP is, in fact, an acronym made up of these three characteristics:

- **Consistency** (or simply **C**): The distributed storage consists of various nodes. Any of these nodes can be used to read, write, or update records in the data repository. Consistency guarantees that at a certain time, *t1*, independent of which node we use to read the data, we will get the same result. Every *read* operation either returns the latest data that is consistent across the distributed repository or gives an error message.
- **Availability** (or simply **A**): Availability guarantees that any node in the distributed storage system will be able to immediately handle the request with or without consistency.
- **Partition Tolerance** (or simply **P**): In a distributed system, multiple nodes are connected via a communication network. Partition tolerance guarantees that, in the event of communication failure between a small subset of nodes (one or more), the system remains operational. Note that to guarantee partition tolerance, data needs to be replicated across a sufficient number of nodes.

Using these three characteristics, the **CAP** theorem carefully summarizes the trade-offs involved in the architecture and design of a distributed system. Specifically, the **CAP** theorem states that, in a storage system, we can only have two of the following characteristics: **consistency** or **C**, **availability** or **A**, and **partition tolerance** or **P**.This is shown in the following diagram:

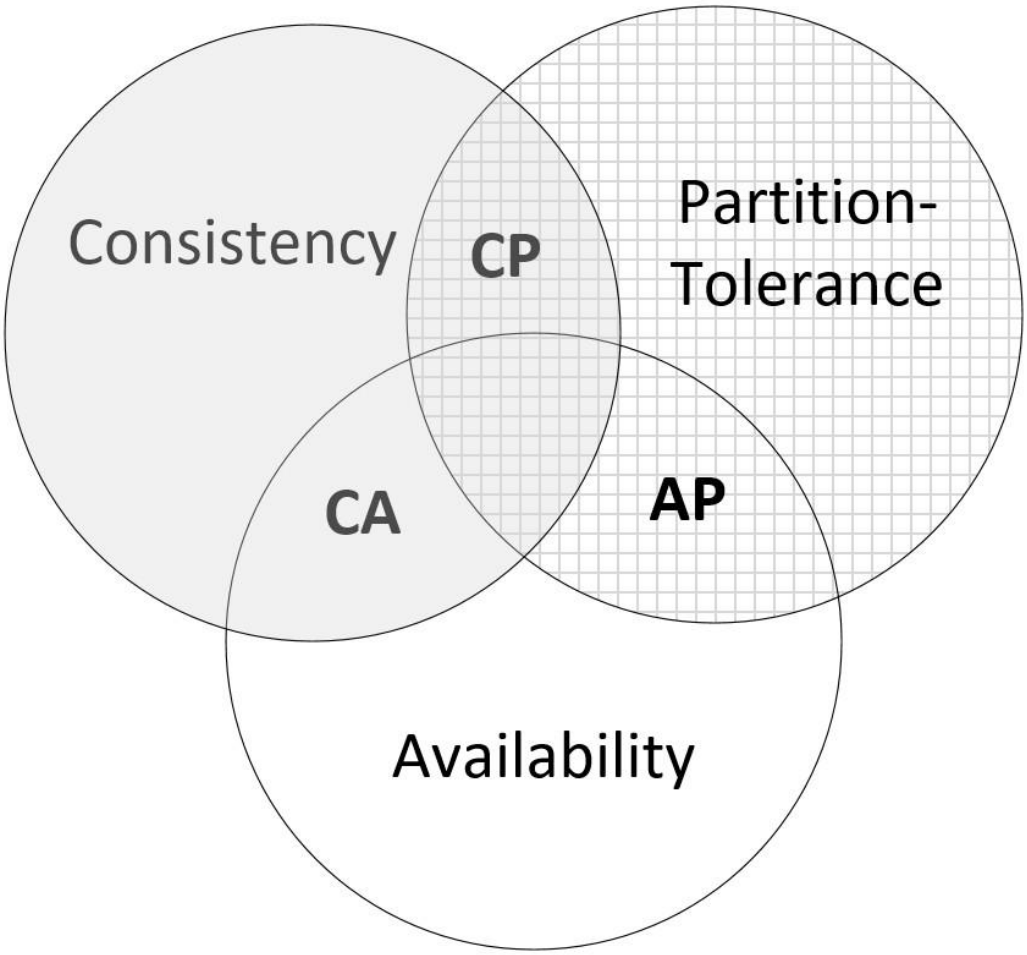


Figure 12.1: Write title here

The CAP theorem also means that we can have three types of distributed storage systems:

- A CA system (implementing Consistency-Availability)
- An AP system (implementing Availability-Partition Tolerance) A
- CP system (implementing Consistency-Partition Tolerance)

Let's look into them, one by one.

CA systems

Traditional single-node systems are CA systems. This is because if we do not have a distributed system, then we do not need to worry about partition tolerance. In that case, we can have a system that has both consistency and availability, that is, a CA

system.Traditional single-node databases such as Oracle or MySQL are all examples of CA systems.

AP systems

AP systems are the distributed storage systems tuned for availability. Designed as highly-responsive systems, they can sacrifice consistency, if needed, to accommodate high-velocity data. This means that these are distributed storage systems that are designed to immediately handle requests from users. Typical user requests are to read or write fast-changing data. Typical AP systems are used in real-time monitoring systems such as sensor networks.High-velocity distributed databases, such as Cassandra, are good examples of AP systems.Let's look into where an AP system can be used. If Transport Canada wants to monitor traffic on one of the highways in Ottawa through a network of sensors installed at different locations on the highway, an AP system is recommended for implementing distributed data storage.

CP systems

CP systems have both consistency and partition tolerance. This means that these are the distributed storage systems that are tuned to guarantee consistency before a read process can fetch a value.A typical use case for CP systems is when we want to store document files in JSON format. Document datastores such as MongoDB are CP systems tuned for consistency in a distributed environment.Distributed data storage is increasingly becoming the most important part of a modern IT infrastructure. Distributed data storage should be carefully designed based on the characteristics of the data and the requirements of the problem we want to solve. Classifying data storage into CA, AP, and CP systems help us to understand the various trade-offs involved when designing data storage systems.Now, let's look into streaming data algorithms.

Presenting streaming data algorithms

Data can be categorized as bounded or unbounded. Bounded data is data at rest and is usually processed through a batch process. Streaming is basically data processing on unbounded data. Let's look into an example. Let's assume that we are analyzing fraudulent transactions at a bank. If we want to look for fraud transactions 7 days ago, we have to look at the data at rest; this is an example of a batch process.On the other hand, if we want to detect fraud in real-time, that is an example of streaming. Hence, streaming data algorithms are those algorithms that deal with processing data streams. The fundamental idea is to divide the input data stream into batches, which are then processed by the processing node. Streaming algorithms need to be

faulttolerant and should be able to handle the incoming velocity of data. As the demand for real-time trend analysis is increasing, the demand for stream processing is also increasing these days. Note that, for streaming to work, data has to be processed fast and while designing algorithms, this needs to be always kept in mind.

Applications of streaming

IThere are many applications of streaming data and its utilization in a meaningful way. Some of the applications are as follows:

- Fraud detection
- System monitoring
- Smart order routing
- Live dashboards
- Traffic sensors along highways
- Credit card transactions

User moves in multi-user online gaming

Now, let's see how we can implement streaming using Python.

Presenting data compression algorithms

Data compression algorithms are involved in the process of reducing the size of data.In this chapter, we will delve into a specific data compression algorithm named the lossless compression algorithm.

Lossless compression algorithms

These are algorithms that are capable of compressing data in such a way that it can be decompressed without any loss of information. They are used when it is important to retrieve the exact original files after decompression. Typical uses of lossless compression algorithms are as follows:

- To compress documents
- To compress and package source code and executable files
- To convert a large number of small files into a small number of large files

Understanding the basic techniques of lossless compression

Data compression is based on the principle that the majority of data is known to be using more bits than its entropy indicates is optimal. Recall that entropy is a term used to specify the information that the data carries. It means that a more optimal bit representation of the same information is possible. Exploring and formulating more efficient bit representation becomes the basis for devising compression algorithms. Lossless data compression takes advantage of this redundancy to compress data without losing any information. In the late '80s, Ziv and Lempel proposed dictionarybased data compression techniques that can be used to implement lossless data compression. These techniques were an instant hit due to their speed and good compression rate. These techniques were used to create the popular Unixbased *compress* tool. Also, the ubiquitous `gif` image format uses these compression techniques, which proved to be popular as they could be used to represent the same information in a lesser number of bits, saving space and communication bandwidth. These techniques later became the basis of developing the `zip` utility and its variants. The compression standard, V.44, used in modems is also based on it.Let's now look at the techniques one by one in the upcoming sections.

Huffman coding

Huffman coding is one of the oldest methods of compressing data and is based on creating a Huffman tree, which is used to both encode and decode data. Huffman coding can represent data content in a more compact form by exploiting the fact that some data (for instance certain characters of the alphabet) appears more frequently in a data stream. By using encodings of different length (shorter for the most frequent characters and longer for the least frequent ones), the data consumes less space.Now, let's learn a few terminologies related to Huffman coding:

- **Coding**: Coding in the context of data represents the method of representing data from one form to another. We would like the resultant form to be concise.
- **Codeword**: A particular character in encoded form is called a codeword. **Fixed-**
- **length coding**: This is when each encoded character, that is, codeword, uses the same number of bits.
- **Variable-length coding**: Codewords can use a different number of bits.
- **Evaluation of code**: This is the expected number of bits per codeword. **Prefix**
- **free codes**: This means that no codeword is a prefix of any other codeword. **Decoding**: This means that a variable-length code must be free from any prefix.
-

For understanding the last two terms, you need to have a look at this table first:

character frequency fixed length code variable length code

L	.45	000	0
M	.13	001	101
N	.12	010	100
X	.16	011	111
Y	.09	100	1101
Z	.05	101	1100

Now, we can infer the following:

- **Fixed length code**: The fixed-length code for this table is 3.
- **Variable length code**: The variable-length code for this table is *45(1) + .13(3) + .12(3) + .16(3) + .09(4) + .05(4) = 2.24*.

The following diagram shows the Huffman tree created from the preceding example:

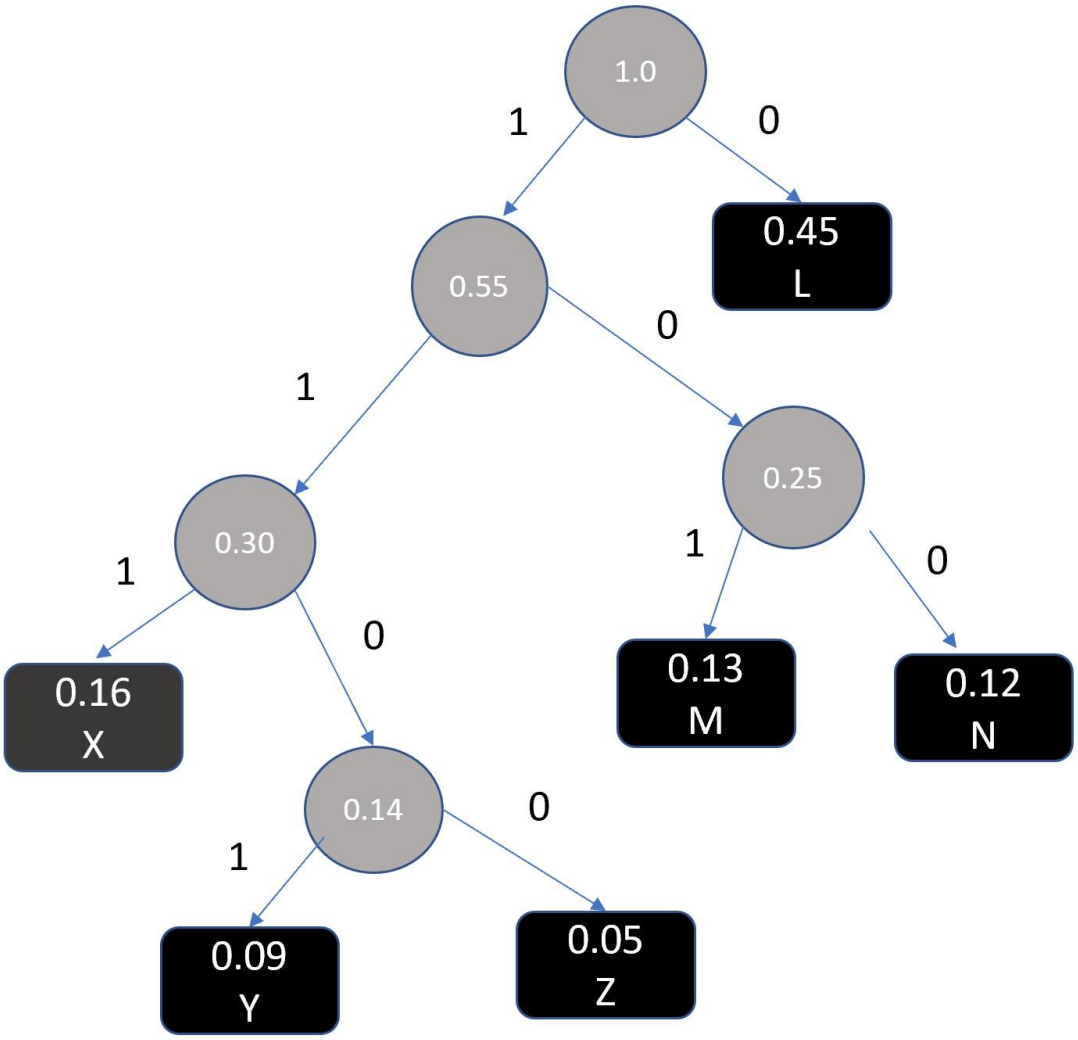


Figure 12.2: Write title here

Note that Huffman encoding is about converting data into a Huffman tree that enables compression. Decoding or decompression brings the data back to the original format.

A practical example – Twitter real-time sentiment analysis

Twitter is said to have almost 7,000 tweets every second on a wide variety of topics. Let's try to build a sentiment analyzer that can capture the emotions of the news from different news sources in real time. We will start by importing the required packages:

1. Import the needed packages:

```
import tweepy,json,time import
numpy as np import pandas as pd
import matplotlib.pyplot as plt
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
analyzer = SentimentIntensityAnalyzer()
```

A. Note that we are using the following two packages:
Б. **VADER** sentiment analysis, which stands for **Valence Aware Dictionary and Sentiment Reasoner**. It is one of the popular rule-based sentiment analysis tools that is developed for social media. If you have never used it before, then you will first have to run the following:

```
pip install vaderSentiment
```

1. `Tweepy` , which is a Python-based API to access Twitter. Again, if you have never used it before, you need to first run this:

```
pip install Tweepy
```

1. The next step is a bit tricky. You need to make a request to create a developer account with Twitter to get access to the live stream of tweets. Once you have the API keys, you can represent them with the following variables:

```
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>
```

1. Let's then configure the `Tweepy` API authentication. For that, we need to provide the previously created variables:

```
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())
```

1. Now comes the interesting part. We will choose the Twitter handles of the news sources that we want to monitor for sentiment analysis. For this example, we have chosen the following news sources:

```
news_sources = ("@BBC", "@ctvnews", "@CNN","@FoxNews", "@dawn_com")
```

1. Now, let's create the main loop. This loop will start with an empty array called `array_sentiments` to hold the sentiments. Then, we will loop through all five news sources and collect 100 tweets each. Then, for each tweet, we will calculate its polarity:

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

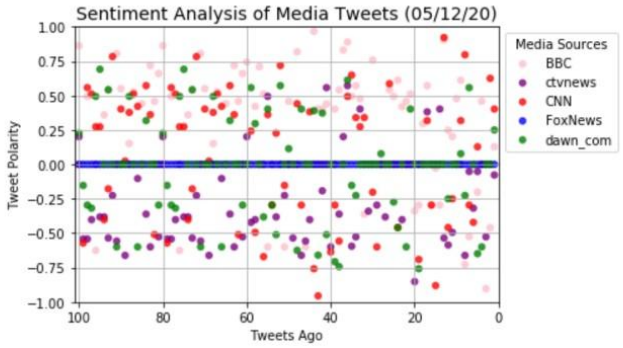
print("DONE with extracting tweets")
```

```
...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
```

1. Now, let's create a graph that shows the polarity of the news from these individual news sources:

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()
```



A. Note that each of the news sources is represented by a different color.

Б. Now, let's look at the summary statistics:

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

1. The preceding numbers summarize the trends of the sentiments. For example, the sentiments of BBC are found to be the most positive, and the Canadian news channel, CTVnews, seems to be carrying the most negative emotions.

Summary

In this chapter, we looked at the design of data-centric algorithms. We focused on three aspects of data-centric algorithms: storage, compression, and streaming.We looked into how the characteristics of data can dictate the data storage design. We looked into two different types of data compression algorithms. Then, we studied a practical example of how data streaming algorithms can be used to count words from a stream of textual data.In the next chapter, we will look at cryptographic algorithms. We will learn how we can use the power of these algorithms to secure exchanged and stored messages.

OceanofPDF.com

13 Cryptography

Join our book's Discord space

https://packt.link/40Algos



"I carry my unwritten poems in cipher on my face!"-- George Eliot

This chapter introduces you to algorithms related to cryptography. We will start by presenting the background, then we will discuss symmetric encryption algorithms. We will then explain the **Message-Digest 5** (**MD5**) algorithm and the **Secure Hash Algorithm** (**SHA**) and present the limitations and weaknesses of implementing symmetric algorithms. Next, we will discuss asymmetric encryption algorithms and how they are used to create digital certificates. Finally, we will present a practical example that summarizes all of these techniques.By the end of this chapter, you will have a basic understanding of various issues related to cryptography.The following topics are discussed in this chapter:

- Introduction to cryptography
- Understanding the types of cryptography techniques
- Example – security concerns when deploying a machine learning model

Let's start by looking at the basic concepts.

Introduction to Cryptography

Techniques to protect secrets have been around for centuries. The earliest attempts to secure and hide data from adversaries date back to ancient inscriptions discovered on monuments in Egypt, where a special alphabet that was known by only a few trusted people was used. This early form of security is called obscurity and is still used in different forms today. In order for this method to work, it is critical to protect the secret, which would be the secret meaning of the alphabet in the above example. Later in time, finding foolproof ways of protecting important messages was important in both World War One and World War Two. In the late-20th century, with the introduction of electronics and computers, sophisticated algorithms were developed to secure data, giving rise to a whole new field called cryptography. This chapter discusses the algorithmic aspects of cryptography. The purpose of these algorithms is to allow secure data exchange between two processes or users. Cryptographic algorithms find strategies of using mathematical functions to ensure the stated security goals.First, we will look at the importance of "the weakest link" in the infrastructure.

Understanding the Importance of the Weakest Link

Sometimes, when architecting the security of digital infrastructure, we put too much emphasis on the security of individual entities and don't pay the necessary attention to end-to-end security. This can result in us overlooking some loopholes and vulnerabilities in the system, which can later be exploited by hackers to access sensitive

data. The important point to remember is that a digital infrastructure, as a whole, is only as strong as its weakest link.
For a hacker, this weakest link can provide backdoor access to the sensitive data in the digital infrastructure. Beyond a certain point, there is not much benefit in fortifying the front door without closing all the back doors.As the algorithms and techniques for keeping digital infrastructure become more and more sophisticated, attackers keep upgrading their techniques as well. It is always important to remember that one of the easiest ways for attackers to hack digital infrastructure is by exploiting these vulnerabilities to access sensitive information.

> In 2014, a cyber attack on a Canadian federal research institute—the National Research Council (NRC)—is estimated to have cost hundreds of millions of dollars. The attackers were able to steal decades of research data and intellectual property material. They used a loophole in the Apache software that was used on the web servers to gain access to the sensitive data.

In this chapter, we will highlight the vulnerabilities of various encryption algorithms.Let's first look at the basic terminology used.

The Basic Terminology

Let's look at the basic terminology related to cryptography:

- **Cipher**: An algorithm that performs a particular cryptographic function.
- **Plain text**: The plain data, which can be a text file, a video, a bitmap, or a digitized voice. In this chapter, we will represent plain text as *P*.
- **Cipher text**: The scrambled text that is obtained after applying cryptography to the plain text. In this chapter, we will represent this as *C*.
- **Cipher suite**: A set or suite of cryptographic software components. When two separate nodes want to exchange messages using cryptography, they first need to agree on a cipher suite. This is important in ensuring that they use exactly the same implementation of the cryptographic functions.
- **Encryption**: The process of converting plain text, *P*, into cipher text, *C*, is called encryption. Mathematically, it is represented by *encrypt(P) = C*.
- **Decryption**: The process of converting cipher text back into plain text. Mathematically, it is represented by *decrypt(C) = P*.
- **Cryptanalysis**: The methods used to analyze the strength of the cryptographic algorithms. The analyst tries to recover the plain text without access to the secret.
- **Personally Identifiable Information** (**PII**): PII is the information that can be used to trace an individual's identity when used alone or with other relevant data. Some examples include protected information, such as a social security number, date of birth, or mother's maiden name.

Let us first understand the security needs of a system.

Understanding the Security Requirements

It is important to first understand the exact security needs of a system. Understanding this will help us use the correct cryptographic technique and discover the potential loopholes in a system. One way of developing better understanding of the security needs of a system is by answering the following four questions:

- Which individuals or processes need to be protected?
- Who are we protecting the individuals and processes from?
- Where should we protect them?
- Why are we protecting them?

Let us take the example of the a Virtual Private Cloud (VPC) in AWS Cloud. VPC allows us to create a logically isolation network where resources like virtual machines are added to it. In order, to understand the security requirements of a VPC, it is important to first identity the identities by answering the above four questions.

- How many individuals are planning to use this system?
- What sort of information needs to be protected?
- Should we protect the VPC only, or we are passing message to the system that need to be encrypted and communicated to the VPC?
- What is the security classification of the data? What are the potential risks? Why would anyone have an incentive to try to hack the system?

Most of the answers to these questions will come performing the following three steps:

- Step 1: Identify the entities.
- Step 2: Establish the security goals.
- Step 3: Understand the sensitivity of the data.

Let's look at these steps one by one.

Step 1: Identifying the Entities

An entity can be defined as an individual, a process, or a resource that part of an information system. We first need to identify that how users, resources, and processes are present at the runtime. Then, we will quantify the security needs of these identified entities, either as individually or as a group. Once we better understand these requirements, we can establish the security goals of our digital system.

Step 2: Establishing the Security Goals

The goal of designing a security system is to protect information from being stolen, compromised, or attacked. Cryptographic algorithms are typically used to meet one or more security goals:

- **Authentication**: Simply put, authentication is how we prove that a user is who they claim to be. Through the process of authentication, we ensure that a user's identity is verified. The process of authentication starts by having the user provide their identity. This is followed by providing information that is only known by the user and so can only be produced by them.
- **Confidentiality**: Data that needs to be protected is called **sensitive data**. Confidentiality is the concept of restricting sensitive data to authorized users only. To protect the confidentiality of sensitive data during its transit or in storage, you need to render the data so that it is unreadable except by authorized users. This is accomplished by using encryption algorithms, which we will discuss later on in this chapter.
- **Integrity**: Integrity is the process of establishing that data has not been altered in any way during its transit or storage. For example, **TCP/IP** (**Transmission Control Protocol/Internet Protocol**) uses checksum or **Cyclic Redundancy Check** (**CRC**) algorithms to verify data integrity.
- **Non-repudiation**: Non-repudiation is the ability to produce unforgeable and irrefutable evidence that that a message was sent or received. This evidence can be used later to prove the receipt of data.

Step 3: Understanding the Sensitivity of the Data

It is important to understand the classified nature of data. The data is categorized by the regulatory authorities such as governments, agencies or organization based on how serious the consequence will be if it is compromised. The categorization of the data helps us choose the correct cryptographic algorithm. There is more

than one way to categorize data, based on the sensitivity of the information it contains. Let's look at the typical ways of classifying data:

- **Public data or unclassified data**: Anything that is available for consumption for the public. For example, information found on a company's website or a government's info portal.
- **Internal data or confidential data**: Although not for public consumption, exposing this data to the public may not have damaging consequences. For example, if an employee's emails complaining about their manager are exposed, it may be embarrassing for the company but this may not have damaging consequences.
- **Sensitive data or secret data**: Data that is not supposed to be for public consumption and exposing it to the public has damaging consequences for an individual or an organization. For example, leaking the details of a future iPhone may harm Apple's business goals and can give an advantage to rivals, such as Samsung.
- **Highly sensitive data**: Also called **top-secret data**. This is information that if disclosed, would be extremely damaging to the organization. This can include customer social security numbers, credit card numbers, or other very sensitive information. Top-secret data is protected through multiple layers of security and requires special permission to access.

In general, more sophisticated security designs are much slower than simple algorithms. It is important to strike the right balance between the security and the performance of the system.

Understanding the Basic Design of Ciphers

Designing ciphers is about coming up with an algorithm that can scramble sensitive data so that a malicious process or an unauthorized user cannot access it. Although over time ciphers have become more and more sophisticated, the underlying principles that ciphers are based on remain unchanged.Let's start by looking at some relatively simple ciphers that will help us understand the underlying principles that are used in the design of cryptographic algorithms.

Presenting Substitution Ciphers

Substitution ciphers have been in use for hundreds of years in various forms. As the name indicates, substitution ciphers are based on a simple concept—substituting characters in plain text with other characters in a predetermined, organized way.Let's look at the exact steps involved in this:

A. First, we map each character to a substitute character.
Б. Then, encode and convert the plaintext into cipher text by replacing each character in the plain text with another character in the ciphertext using substitution mapping.
B. To decode, bring back the plaintext by using substitution mapping.

Following are the examples of the substitutions based ciphers.

- Caesar cipher
- Rotation13

Let us look into them in more details

Caesar cipher

In Caesar ciphers are based on a substitution mapping. A substitution mapping changes the actual string in a deterministic way by apply a simple formula which is kept secret. The substitution mapping is created by replacing each character with the third character to the right of it. This mapping is described in the following diagram:

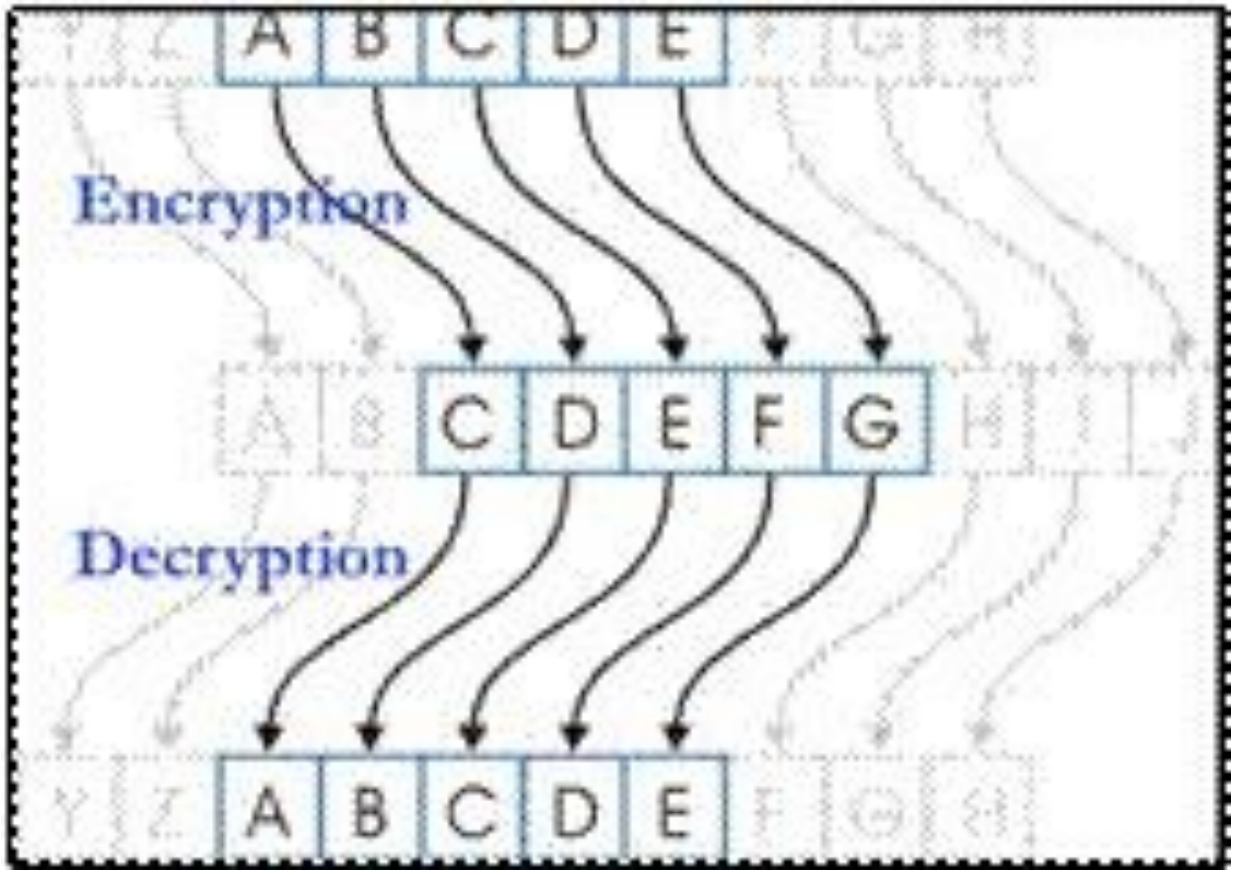


Figure 1: The Substitution Mapping of Caesar Ciphers

Let's see how we can implement a Caesar cipher using Python:

```
import string rotation = 3 P = 'CALM';
C='' for letter in P:    C = C+
(chr(ord(letter) + rotation))
```

We can see that we applied a Caesar cipher to the plaintext, `CALM` .Let's print the cipher text after encrypting with the Caesar cipher:

```
In [37]: print(C)

FDOP
```

Caesar ciphers are said to have been used by Julius Caesar to communicate with his advisers.

A Caesar cipher is a simple cipher and is easy to implement. The downside is that it is not too difficult to crack as a hacker could simply iterate through all the possible shift of the alphabet (all 2626 of them). And see if any

sensible message appears. Given the current processing abilities of the computers, this is a relatively small number of combinations to do. It should not be used to protect highly sensitive data.

Rotation 13 (ROT13)

ROT13 is another of the substitution-based encryptions. In ROT13, the substitution mapping is created by replacing each character with the 13th character to the right of it. The following diagram illustrates this:

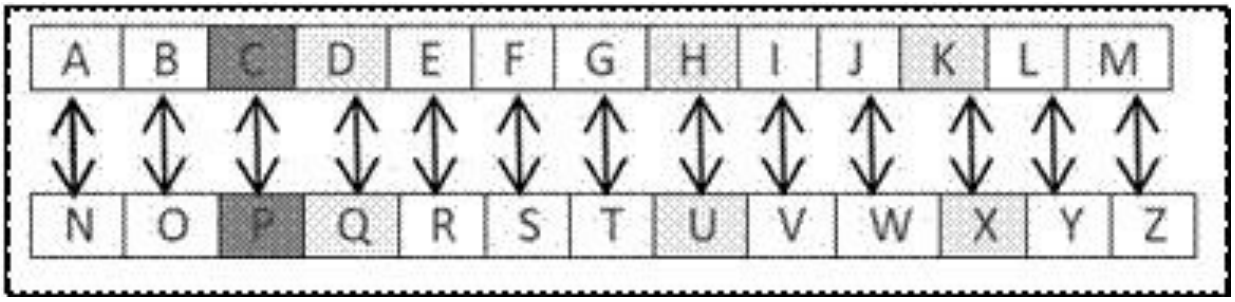


Figure 13.2: Working of ROT13

This means that if `ROT13()` is the function that implements ROT13, then the following applies:

```
import codecs a_message = 'CALM'
c='' c=codecs.encode(a_message,
'rot_13')
```

Now, let's print the encoded value of `C` :

```
      >> print(c)
PNYZ
```

Cryptanalysis of substitution ciphers

Substitution ciphers are simple to implement and understand. Unfortunately, they are also easy to crack. A simple cryptanalysis of substitution ciphers shows that if we use the English language alphabet, then all we need to determine to crack the cipher is how much are we rotating by. We can try each letter of the English alphabet one by one until we are able to decrypt the text. This means that it will take around 25 attempts to reconstruct the plain text.Now, let's look at another type of simple cipher—transposition ciphers.

Understanding Transposition Ciphers

In transposition ciphers, the characters of the plain text are transposed. Let's look at the steps involved for this:

A. Create the matrix and choose the size of the transposition matrix. It should be large enough to fit the plain text string.
Б. Fill in the matrix by writing in all of the characters of the string horizontally.
B. Read all of the characters of the string vertically in the matrix.

Let's look at an example.Let's take the `Ottawa Rocks` plain text (P).First, let's encode P. For that, we will use a 3 x 4 matrix and write in the plaintext horizontally:

O t t a
w a R o
c k s

The `read` process will read the string vertically, which will generate the cipher text— `OwctaktRsao` .

> *The Germans used a cipher named ADFGVX in the First World War, which used both the transposition and substitution ciphers. Years later, it was cracked by George Painvin.*

So, these are some of the types of ciphers. Now, let's look at some of the cryptographic techniques that are currently used.

Understanding the Types of Cryptographic Techniques

Different types of cryptographic techniques use different types of algorithms and are used under different sets of circumstances.Broadly, cryptographic techniques can be divided into the following three types:

- Hashing
- Symmetric
- Asymmetric

Let's look at them one by one.

Using the Cryptographic Hash Function

The cryptographic hash function is a mathematical algorithm that can be used to create unique fingerprint of a message. It creates a fixed-sized output, called a hash, from plain text.Mathematically, this looks as follows:

C1 = hashFunction(P1)

This is explained as follows:

- *P*1 is the plain text representing the input data.
- *C*1 is a fixed-length hash that is generated by the cryptographic hash function.

This is shown in the following diagram. The variable-length data is converted into a fixed-length hash through a one-way hash function:

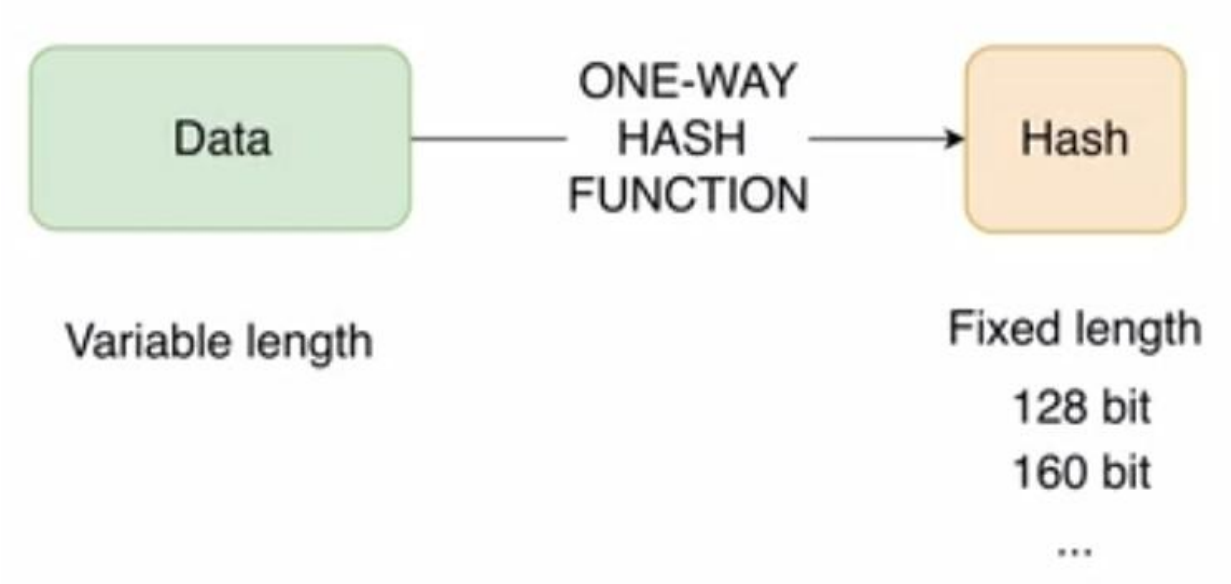


Figure 13.3: Figure 3: One-way hash functions

hash function has the following five characteristics:

А. It is deterministic. The same plain text generates the same hash.
Б. Unique input strings should generate unique output hash values.
В. Regardless of the input message, it has a fixed length.
Г. Even small changes in the plain text generate a new hash.
Д. It is a one-way function, which means that plain text, *P*1, cannot be generated from the cipher text, *C*1.

If we have a situation where each unique message does not have a unique hash, we call it a collision. That is, if we have two texts, P1 and P2 , in the case of collision, it means *hashFunction(P1) = hashFunction(P2)*.Regardless of the hashing algorithm used, collisions are rare. Otherwise, hashing wouldn't be useful. However, for some applications, collisions cannot be tolerated. In those cases, we need to use a hashing algorithm that is more complex but much less likely to generate hash values that collide.

Implementing cryptographic hash functions

Cryptographic hash functions can be implemented by using various algorithms. Let's take a deeper look at two of them.

Understanding MD5-tolerated

MD5 was developed by Poul-Henning Kamp in 1994 to replace MD4. It generates a 128-bit hash. MD5 is a relatively simple algorithm that is vulnerable to collision. In applications where a collision cannot be tolerated, MD5 should not be used.Let's look at an example. In order to generate an MD5 hash in Python, we will use the `passlib` library, which one of the most popular open source libraries, implementing over 30 passwordhashing algorithms. If it is not already installed on your device, install it by using the following code in a Jupyter Notebook:

```
!pip install passlib
```

In Python, we can generate the MD5 hash as follows:

```
In [36]: myHash
Out[36]: '$1$a6sQqHlF$j5iHhbCzmOzVwrxWDxnUu.'
```

Note that MD5 generates a hash of 128 bits.As mentioned, we can use this generated hash as a fingerprint of the original text, which was `myPassword` . Let's see how we can do this in Python:

```
In [37]: md5_crypt.verify("myPassword", myHash)
Out[37]: True

In [38]: md5_crypt.verify("myPassword2", myHash)
Out[38]: False
```

Note that the generated hash for the `myPassword` string matched the original hash, which generated a `True` value. However, it returned `False` as soon as the plain text was changed to `myPassword2` .

When to Use MD5

Looking back at history, weeknesses with MD5 were discovered in the late 1990s. Despite severalissues MD5 usage is still popular. It is ideal to be used for integrity checks for data. Note that MD5 message digest does not uniquely associates the hash with its owner. As the MD5 digest is not a signed hash. The MD5 is used to prove that a file has not been changed since the hash was computed. It is not used to prove the authenticity of a file.Now, let's look at another hashing algorithm—Secure Hash Algorithm (SHA).

Understanding Secure Hashing Algorithm (SHA)

SHA was developed by the National Institute of Standards and Technology (NIST). Let's see how we can use Python to create a hash using the SHA algorithm:

```
from passlib.hash import sha512_crypt
sha512_crypt.using(salt = "qIo0foX5",rounds=5000).hash("myPassword")
myHash=sha512_crypt.using(salt="qIo0foX5",rounds=5000).hash("myPassword")
```

Note the use of a parameter called `salt` . Salting is the procedure of adding random characters before hashing.Running this code will give us the following result:

```
In [13]: myHash
Out[13]: '$6$qIo0foX5$a.RA/OyedLnLEnWovzqngCqhyy3EfqRtwacvWKsIoYSvYgRxCRetM3XSwrgMxwdPqZt4KfbXzCp6y
         NyxI5j6o/'
```

Note that when we use the SHA algorithm, the hash generated is of 512 bytes.

An Application of the Cryptographic Hash Function

Hash functions are used to check the integrity of a file after making a copy of it. To achieve this, when a file is copied from a source to a destination (for example, when downloaded from a web server), a corresponding hash is also copied with it. This original hash, *horiginal*, acts as a fingerprint of the original file. After copying the file, we generate the hash again from the copied version of the file—that is, *hcopied*. If *horiginal = hcopied*—that is, the generated hash matches the original hash—this verifies that the file has not changed and none of the data was lost during the download process. We can use any cryptographic hash function, such as MD5 or SHA, to generate a hash for this purpose.Now, let's look at symmetric encryption.

Using Symmetric Encryption

In cryptography, a key is a combination of numbers that is used to encode plain text using an algorithm of our choice. In symmetric encryption, we use the same key for encryption and decryption. If the key used for symmetric encryption is *K*, then for symmetric encryption, the following equation holds:*EK(P) = C*Here, *P* is the plain text and *C* is the cipher text.For decryption, we use the same key, *K*, to convert it back to *P*:*DK(C) = P*This process is shown in the following diagram:

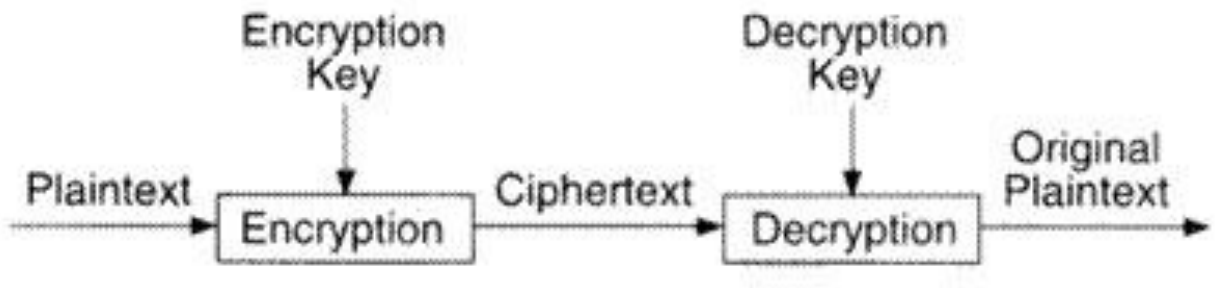


Figure 3.4: Symmetric Encryption

Now, let's look at how we can use symmetric encryption with Python.

Coding Symmetric Encryption

We will use Python's `cryptography` package, in this section, to demonstrate symmetric encryption. It is a comprehensive package that implements many cryptographic algorithms, such as symmetric ciphers and different message digests. When using it for the first time, we first need to install it using the `pip` command:

```
!pip install cryptography
```

Once installed, we can now use the package to implement symmetric encryption, as follows:

1. First, let's import the packages we need:

```
import cryptography as crypt from
cryptography.fernet import Fernet
```

1. Let's generate the key:

```
In [29]: key = Fernet.generate_key()
         print(key)

         b'NbzXiNqKR25SEv_O8EpuW2Lr_QO2vDStTDV4ex4WA5U='
```

1. Now, let's open the key:

```
file = open('mykey.key', 'wb')
file.write(key)  file.close()
```

1. Using the key, let's now try to encrypt the message:

```
file = open('mykey.key', 'rb')
key = file.read()  file.close()
```

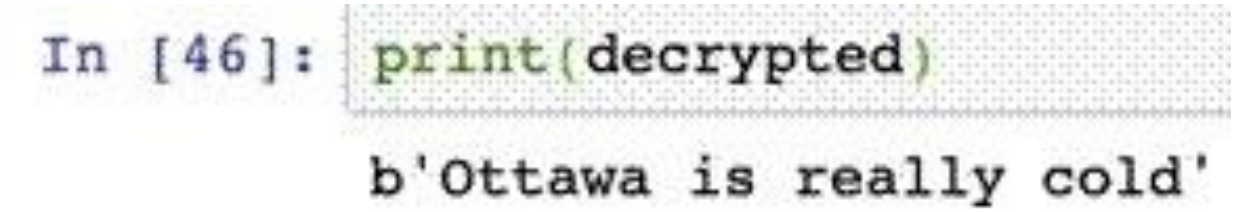
1. Now, let's decrypt the message using the same key:

```
from cryptography.fernet import Fernet
message = "Ottawa is really cold".encode()
f = Fernet(key) encrypted =
f.encrypt(message)
```

1. Let's decrypt the message and assign it to a variable named `decrypt` :

```
decrypted = f.decrypt(encrypted)
```

1. Let's now print the `decrypt` variable to verify whether we are able to get the same message:



Let's look at some of the advantages of symmetric encryption.

The Advantages of Symmetric Encryption

Although the performance of the symmetric encryption is dependent on the exact algorithm that is used, in general, it is a lot faster than asymmetric encryption.

The Problems with Symmetric Encryption

When two users or processes plan to use symmetric encryption to communicate, they need to exchange keys using a secure channel. This gives rise to the following two problems:

- **Key protection**: How to protect the symmetric encryption key.
- **Key distribution**: How to share the symmetric encryption key from the source to the destination.

Now, let's look at asymmetric encryption.

Asymmetric Encryption

In the 1970s, asymmetric encryption was devised to address some of the weaknesses of symmetric encryption that we discussed in the previous section.The first step in asymmetric encryption is to generate two different keys that look totally different but are algorithmically related. One of them is chosen as the private key, *Kpr*, and the other one is chosen as the public key, *Kpu*. Mathematically, we can represent this as follows: *EKpr(P) = C*

Here, *P* is the plain text and *C* is the cipher text.We can decrypt it as follows:
DKpu(C) = P

Public keys are supposed to be freely distributed and private keys are kept secret by the owner of the key pair.The fundamental principle is that if you encrypt with one of the keys, the only way to decrypt it is by using the other key. For example, if we encrypt the data using the public key, we will need to decrypt it using the other key—that is, the private key. Now, let's look at one of the fundamental protocols of asymmetric encryption—the **Secure Sockets Layer** (**SSL**)/**Transport Layer Security** (**TLS**) handshake—which is responsible for establishing a connection between two nodes using asymmetric encryption.

The SSL/TLS Handshaking Algorithm

SSL was originally developed to add security to HTTP. Over time, SSL was replaced with a more efficient and more secure protocol, called TLS. TLS handshakes are the basis of how HTTP creates a secure communication session. A TLS handshake occurs between the two participating entities—the client and the server. This process is shown in the following diagram:

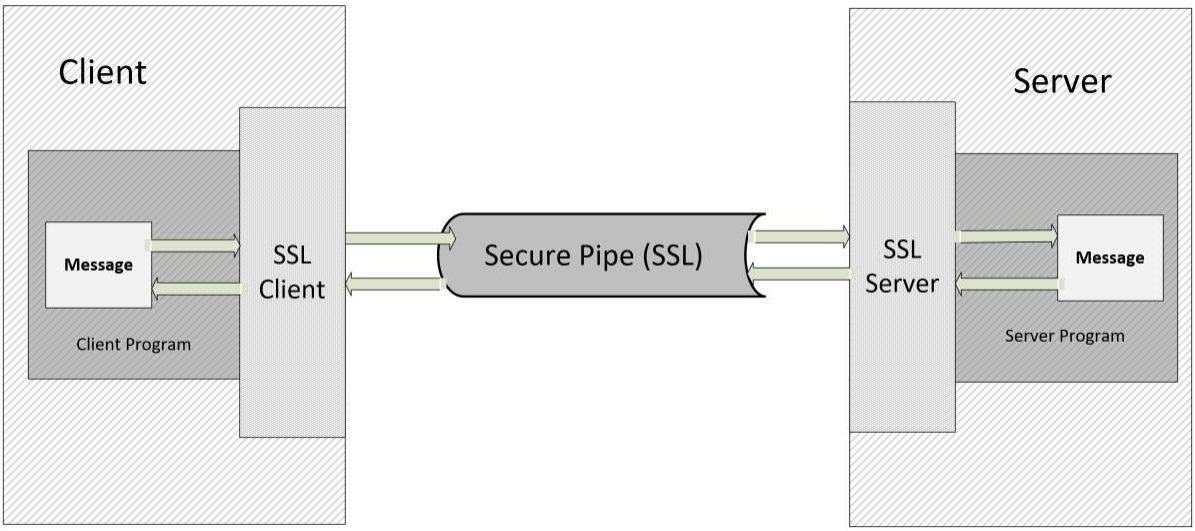


Figure 13.5: Secure Session between the client and the server

A TLS handshake establishes a secure connection between the participating nodes. The following are the steps that are involved in this process:

А. The client sends a `client hello` message to the server. The message also contains the following:
- The version of TLS that is used
- The list of cipher suites supported by the client
- A compression algorithm
- A random byte string, identified by `byte_client`

Б. The server sends a `server hello` message back to the client. The message also contains the following: A cipher suite selected by the server from the list provided by the client
- A session ID
- A random byte string, identified by `byte_server`
- A server digital certificate, identified by `cert_server` , containing the public key of the server

If the server requires a digital certificate for client authentication or a client certificate request, the client server request also includes the following:
- The distinguished names of the acceptable CAs

The types of certificates supported

В. The client verifies `cert_server` .

Г. The client generates a random byte string, identified by `byte_client2` , and encrypts it with the public key of the server provided through `cert_server` .

Д. The client generates a random byte string and identifies encrypts with its own private key.

Е. The server verifies the client certificate.

Ж. The client sends a `finished` message to the server, which is encrypted with a secret key.

З. To acknowledge this from the server side, the server sends a `finished` message to the client, which is encrypted with a secret key.

И. The server and client have now established a secure channel. They can now exchange messages that are symmetrically encrypted with the shared secret key. The entire methodology is shown as follows:

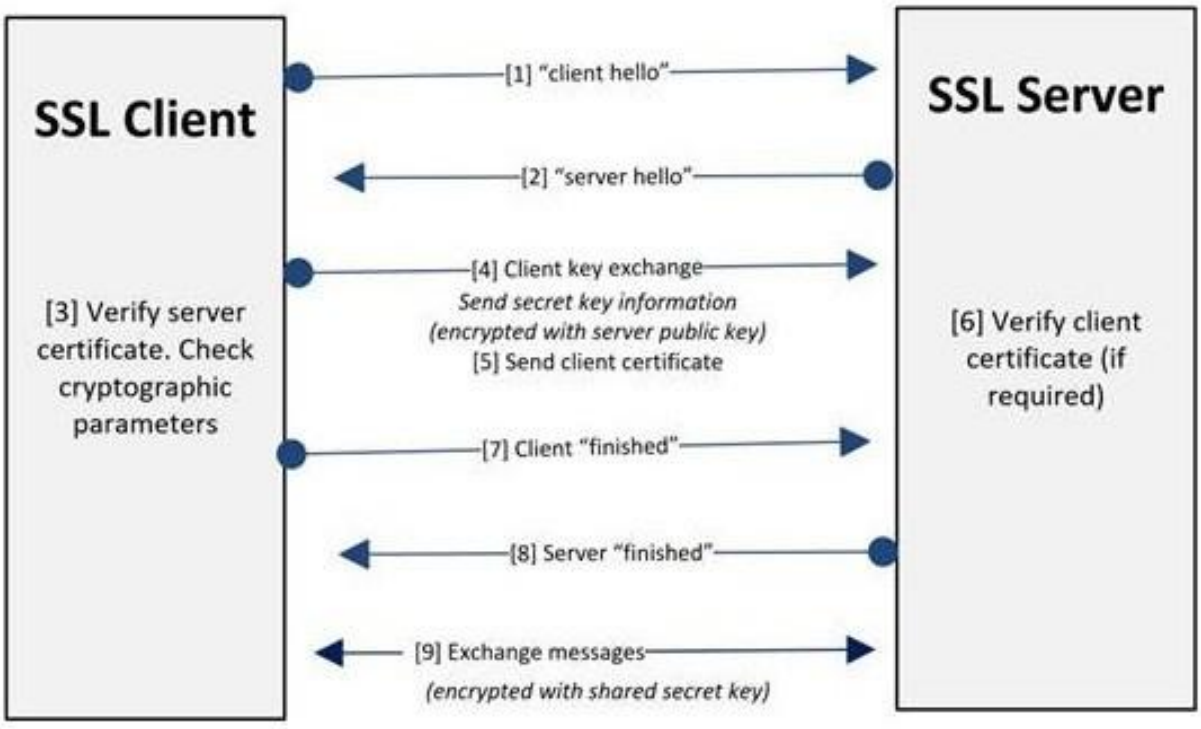


Figure 13.6: Secure Session between the client and the server

Now, let's discuss how we can use asymmetric encryption to create **Public Key Infrastructure** (**PKI**), which is created to meet one or more security goals for an organization.

Public Key Infrastructure

Asymmetric encryption is used to implement PKI. PKI is one of the most popular and reliable ways to manage encryption keys for an organization. All the participants trust a central trusting authority called a CA. CAs verify the identity of individuals and organizations and then issue them digital certificates (a digital certificate contains a copy of a person or organization's public key and its identity), verifying that the public key associated with that individual or organization actually belongs to that individual or organization.The way it works is that the CA asks a user to prove their identity, with different standards followed for individuals and organizations. This could involve simply verifying ownership of a domain name, or it could involve a more rigorous process that involves physical proof of identity, depending on the type of digital certificate that a user is trying to obtain. If the CA is satisfied that the user is indeed who they claim to be, the user then provides the CA with their public encryption key over a secure channel. The CA uses this information to create a digital certificate that contains information about the user's identity and their public key. This certificate is digitally signed by the CA. The user can then show their certificate to anyone who wants to verify their identity, without having to send it through a secure channel, as the certificate doesn't contain any sensitive information itself. The person receiving the certificate does not have to verify the user's identity directly. That person can simply verify that the certificate is valid by verifying the CA's digital signature, which validates that the public key contained in the certificate does, in fact, belong to the individual or organization named on the certificate.

> *The private key of the CA of an organization is the weakest link in the PKI chain of trust. If an impersonator gets hold of Microsoft's private key, for example, they can install malicious software onto millions of computers around the world by impersonating a Windows update.*

Blockchain and Cryptography

There is no doubt that in recent years there is so much excitement around blockchain and cryptocurrency. The blockchain is said to be one of the most secure technologies every invented. The excitement about blockchain started with Bitcoin and digital currencies. Although digital currencies were developed since 1980 but with the Bitcoin, it became mainstream. The rise of Bitcoin was due to the widespread availability of distributed systems. It has two important characteristics that made it a game-changer.It is decentralized by design. It uses a network of miners and a distributed algorithm called blockchain. Bitcoin is based on inherent incentives for miners to compete to add a block to the blockchain by attempting to answer a very complex computational puzzle. The winner miner is eligible to claim different bitcoins as a reward for his/her effort. Although the blockchain was developed for Bitcoin, it has found broader use and applications. Blockchain is based on a distributed consensus algorithm, using Distributed Ledger Technology (DLT). It has following characteristics:**Decentralization:** It is based distributed not centralized architecture. There is no central authority. Each node in a blockchain system is involved in maintaining the integrity of the DLT. There is a trusted consensus among all the participating nodes. In this distributed architecture, the transactions are stored on the nodes of the constituent nodes, forming a P2P network.**Chain-like formations:** All transactions of blockchain are accumulated in list of blocks. When several blocks are added, it created chain-like formation and is the reason of its name. **Immutability:** The data is secure, replicated and stored in immutable blocks. **Reliability**: A lineage or history is maintained for each transaction. Each transaction is verified and recorded using cryptographic techniques.Under the hood, blockchain transactions uses cryptographic hashes from each of the previous blocks in the chain. Hash functions are used to create on-way fingerprint has of an arbitrary chunk of data. A Merkle tree or hash tree is used to verify data stored, handled and transferred between different participating nodes. It uses SHA-2 for hashing. The diagram of a particular transaction is shown below.

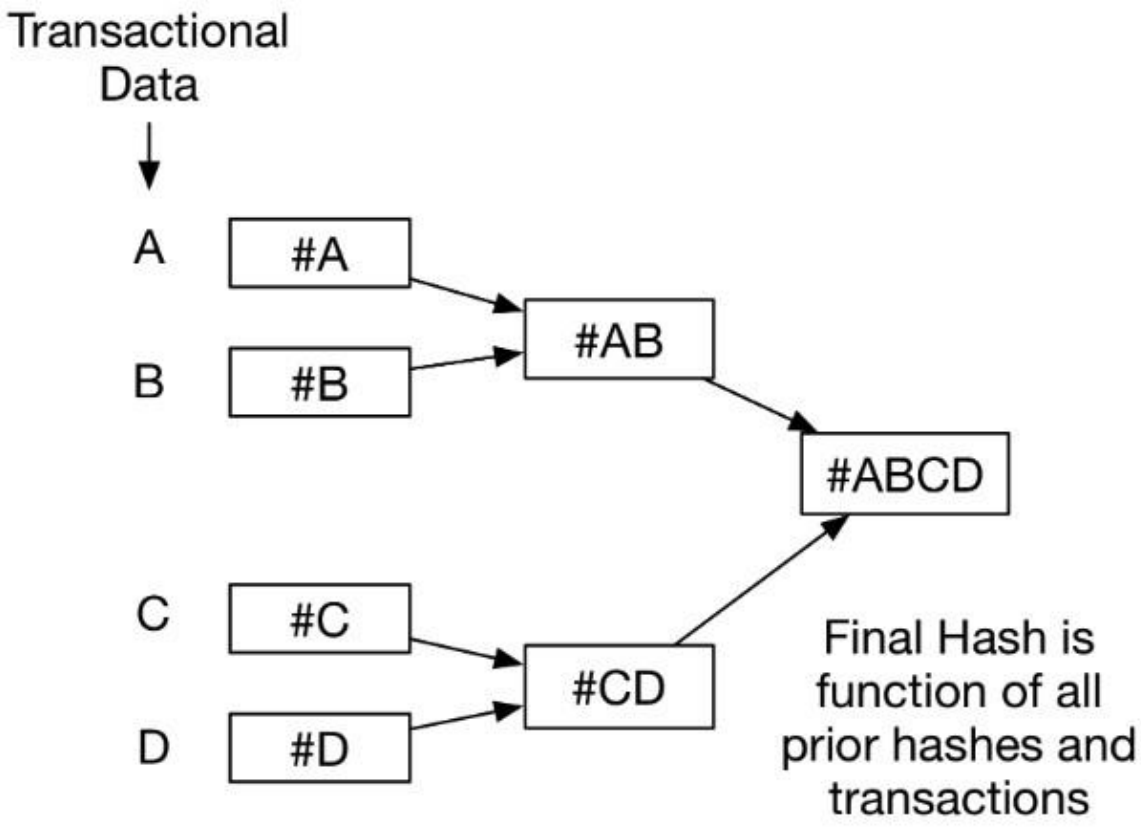


Figure 7: The Merkle Tree of Blockchain

Figure 7 summarizes the working of blockchain. It shows how transactions get converted into blocks, which are, in turn, converted into chains. On left hand side, four transactions, A, B, C and D are shown. Next, the Merkle root is created by applying hash function. The Merkle root can be considered as a data structure which forms part of the block header. As the transaction are immutable, the previously recorded transactions cannot be changed. Note that the hash value of the previous block header is also becomes part of block, thus incorporation transaction records. This creates chain like processing structures and is the reason of the name of Blockchain.Each of the Blockchain user is authenticated and authorized using cryptography eliminating the need of third party authentication and authorization. Digital signatures are used to secure transactions as well. The receiver of a transaction has a public key. Blockchain technology eliminates the involvement of third parties for transaction validation and relies on cryptographic proof for the same . Transactions are secured using digital signature. Each user has a unique private key and establishes her or his digital identity in the system

Example – Security Concerns When Deploying a Machine Learning Model

In *Chapter 6, Unsupervised Machine Learning Algorithms*, we looked at the CRISP-DM (Cross-Industry Standard Process for Data Mining) life cycle, which specifies the different phases of training and deploying a machine learning model. Once a model is trained and evaluated, the final phase is deployment. If it is a critical machine learning model, then we want to make sure that all of its security goals are met.Let's analyze the common challenges faced in deploying a model such as this and how we can address those challenges using the concepts discussed in this chapter. We will discuss strategies to protect our trained model against the following three challenges:

- **Man-in-the-Middle** (**MITM**) attacks
- Masquerading
- Data tempering

Let's look at them one by one.

MITM attacks

One of the possible attacks that we would want to protect our model against is MITM attacks. A MITM attack occurs when an intruder tries to eavesdrop on a supposedly private communication to deploy a trained machine learning model.Let's try to understand MITM attacks sequentially using an example scenario.Let's assume that Bob and Alice want to exchange messages using PKI:

A. Bob is using {Pr_{Bob}, Pu_{Bob}} and Alice is using {Pr_{Alice}, Pu_{Alice}}. Bob has created a message, *M*Bob, and Alice has created a message, *M*Alice. They want to exchange these messages with each other in a secure way.
Б. Initially, they need to exchange their public keys to establish a secure connection with each other. This means that Bob uses *Pu*Alice to encrypt *M*Bob before sending the message to Alice.
B. Let's assume that we have an eavesdropper, *X*, who is using {Pr_X, Pu_X}. The attacker is able to intercept the public key exchanges between Bob and Alice and replace them with its own public certificate.
Г. Bob sends *M*Bob to Alice, encrypting it with *Pu*X instead of *Pu*Alice, wrongfully thinking that this is Alice's public certificate. Eavesdropper *X* intercepts the communication. It intercepts the *M*Bob message and decrypts it using *Pr*Bob.

This MITM attack is shown in the following diagram:

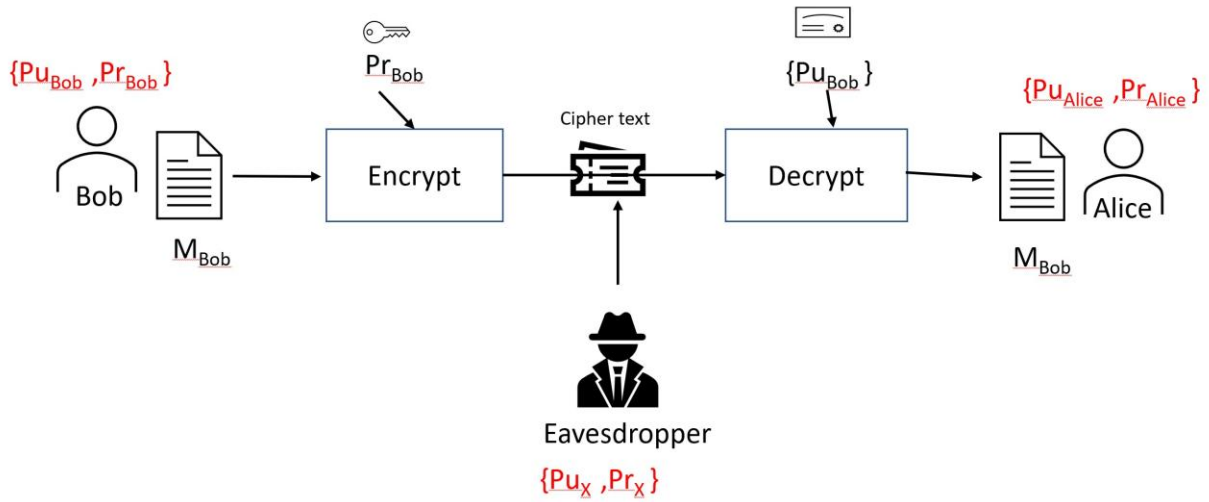


Figure 13.6: XNow, let's look at how we can prevent MITM attacks.

How to prevent MITM attacks

Let's explore how we can prevent MITM attacks by introducing a CA to the organization. Let's say the name of this CA is myTrustCA. The digital certificate has its public key, named PumyTrustCA, embedded in it. myTrustCA is responsible for signing the certificates for all of the people in the organization, including Alice and Bob. This means that both Bob and Alice have their certificates signed by myTrustCA. When signing their certificates, myTrustCA verifies that they are indeed who they claim to be.Now, with this new arrangement in place, let's revisit the sequential interaction between Bob and Alice:

A. Bob is using {Pr_{Bob}, Pu_{Bob}} and Alice is using {Pr_{Alice}, Pu_{Alice}}. Both of their public keys are embedded into their digital certificates, signed by myTrustCA. Bob has created a message, *M*Bob, and Alice has created a message, *M*Alice. They want to exchange these messages with each other in a secure way.
Б. They exchange their digital certificates, which contain their public keys. They will only accept the public keys if they are embedded in the certificates signed by the CA they trust. They need to exchange their public keys to establish a secure connection with each other. This means that Bob will use *Pu*Alice to encrypt *M*Bob before sending the message to Alice.
B. Let's assume that we have an eavesdropper, *X*, who is using {Pr_X, Pu_X}. The attacker is able to intercept the public key exchanges between Bob and Alice and replace them with its own public certificate, *Pu*X.
Г. Bob rejects *X*'s attempt, as the bad guy's digital certificate is not signed by the CA that Bob trusts. The secure handshake is aborted, the attempted attack is logged with a timestamp and all details, and a security exception is raised.

When deploying a trained machine learning model, instead of Alice, there is a deployment server. Bob only deploys the model after establishing a secure channel, using the previously mentioned steps.Let us see how we can implement this in Python.First let us import the packages that are needed.

```
from xmlrpc.client import SafeTransport, ServerProxy import
ssl
```

Now let us create a class that can verify the certirficate.

```
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
#     Create     the     client     proxy     s     =     ServerProxy('https://cloudanum.com:15000',
transport=VerifyCertSafeTransport('server_cert.pem
```

Let us look into other vulnerabilities that our deployed model can face.

Avoiding Masquerading

Attacker *X* pretends to be an authorized user, Bob, and gains access to sensitive data, which is the trained model, in this case. We need to protect the model against any unauthorized changes.One way of protecting our trained model against masquerading is by encrypting the model with an authorized user's private key. Once encrypted, anyone can read and utilize the model by decrypting it through the public key of the authorized user, which is found in their digital certificate. No one can make any unauthorized changes to the model.

Data and Model Encrpytion

Once the model is deployed, the real-time unlabeled data that is provided as input to the model can also be tampered with. The trained model is used for inference and provides a label to this data. To protect data against tampering, we need to protect the data at rest and in communication. To protect the data at rest, symmetric encryption can be used to encode it. To transfer the data, SSL/TLS-based secure channels can be established to provide a secure tunnel. This secure tunnel can be used to transfer the symmetric key and the data can be decrypted on the server before it is provided to the trained model.This is one of the more efficient and foolproof ways to protect data against tampering.Symmetric encryption can also be used to encrypt a model when it has been trained, before deploying it to a server. This will prevent any unauthorized access to the model before it is deployed.Let's see how we can encrypt a trained model at the source, using symmetric encryption with the help of the following steps, and then decrypt it at the destination before using it:

1. Let's first train a simple model using the Iris dataset:

```
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)
```

1. Now, let's define the names of the files that will store the model:

```
filename_source = 'myModel_source.sav'  filename_destination
= "myModel_destination.sav"  filename_sec =
"myModel_sec.sav"
```

A. Note that `filename_source` is the file that will store the trained unencrypted model at the source. `filename_destination` is the file that will store the trained unencrypted model at the destination, and `filename_sec` is the encrypted trained model.

Б. We will use `pickle` to store the trained model in a file: `from pickle import dump`

```
dump(model, open(filename_source, 'wb'))
```

1. Let's define a function named `write_key()` that will generate a symmetric key and store it in a file named `key.key` :

```
def write_key():
     key = Fernet.generate_key()      with
open("key.key", "wb") as key_file:
key_file.write(key)
```

1. Now, let's define a function named `load_key()` that can read the stored key from the `key.key` file:

```
def load_key():     return
open("key.key", "rb").read()
```

1. Next, let's define an `encrypt()` function that can encrypt and train the model, and store it in a file named `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)
```

1. We will use these functions to generate a symmetric key and store it in a file. Then, we will read this key and use it to store our trained model in a file named `filename_sec` :

```
write_key() encrypt(filename_source,load_key())
```

Now the model is encrypted. It will be transferred to the destination where it will be used for prediction:

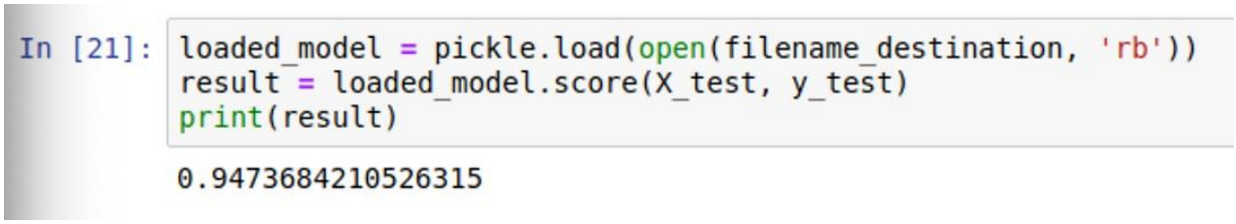
1. First, we will define a function named `decrypt()` that we can use to decrypt the model from `filename_sec` to `filename_destination` using the key stored in the `key.key` file:

```
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)
```

1. Now let's use this function to decrypt the model and store it in a file named `filename_destination` :

```
decrypt(filename_sec,load_key())
```

1. Now let's use this unencrypted file to load the model and use it for predictions:



Note that we have used symmetric encryption to encode the model. The same technique can be used to encrypt data as well, if needed.

Summary

In this chapter, we learned about cryptographic algorithms. We started by identifying the security goals of a problem. We then discussed various cryptographic techniques and also looked at the details of the PKI infrastructure. Finally, we looked at the different ways of protecting a trained machine learning model against common attacks. Now, you should be able to understand the fundamentals of security algorithms used to protect modern IT infrastructures.In the next chapter, we will look at designing large-scale algorithms. We will study the challenges and trade-offs involved in designing and selecting large algorithms. We will also look at the use of a GPU and clusters to solve complex problems.

OceanofPDF.com

14 Large-Scale Algorithms

Join our book's Discord space

https://packt.link/40Algos



Large-scale algorithms are designed to solve gigantic complex problems. The characterizing feature of large-scale algorithms is their need to have more than one execution engine due to the scale of their data and processing requirements. This chapter starts by discussing what types of algorithms are best suited to be run in parallel. Then, it discusses the issues related to parallelizing algorithms. Next, it presents the **Compute Unified Device Architecture** (**CUDA**) architecture and discusses how a single **graphics processing unit** (**GPU**) or an array of GPUs can be used to accelerate algorithms. It also discusses what changes need to be made to

the algorithm to effectively utilize the power of the GPU. Finally, this chapter discusses cluster computing and discusses how Apache Spark creates **Resilient Distributed Datasets** (**RDDs**) to create an extremely fast parallel implementation of standard algorithms.By the end of this chapter, you will be able to understand the basic strategies related to the design of large-scale algorithms.The following topics are covered in this chapter:

- Introduction to large-scale algorithms
- The design of parallel algorithms
- Algorithms to utilize the GPU
- Understanding algorithms utilizing cluster computing
- How to use the GPU to run large-scale algorithms
- How to use the power of clusters to run large-scale algorithms

Let's start with the introduction.

Introduction to large-scale algorithms

Human beings like to be challenged. For centuries, various human innovations have allowed us to solve very complex problems in different ways. From predicting the next target area of a locust attack to calculating the largest prime number, the methods to provide answers for complex problems around us kept on evolving. With the advent of the computer, we found a powerful new way to solve complex algorithms.

Defining a well-designed, large-scale algorithm

AI well-designed, large-scale algorithm has the following two characteristics:

- It is designed to handle a huge amount of data and processing requirements using an available pool of resources optimally.
- It is scalable. As the problem becomes more complex, it can handle the complexity simply by provisioning more resources.

One of the most practical ways of implementing large-scale algorithms is by using the divide and conquer strategy, that is, divide the larger problem into smaller problems that can be solved independently of each other.

Terminology

Let's look into some of the terminology that can be used to quantify the quality of large-scale algorithms.

Latency

Latency is the end-to-end time taken to perform a single computation. If *Compute1* represents a single computation that starts at *t1* and ends at *t2*, then we can say the following:*Latency = t2-t1*

Throughput

In the context of parallel cloud computing, throughput is the number of single computations that can be performed simultaneously. For example, if, at *t1*, we can perform four simultaneous computations, *C1*, *C2*, *C3*, and C_4, then the throughput is four.

The Data Locality Principle

Traditionally, in parallel and distributed processing the data locality principle is pivotal in deciding the optimal resource allocation. It fundamentally suggests that the movement of data should be discouraged in distributed infrastructures. Whenever possible, instead of moving data, it should be processed locally on the node where it resides, otherwise, it reduces the benefit of the parallelization and horizontal scaling.Meanwhile, the advances in networking bandwidth have made data locality principle less relevant. Network bandwidth can be quantified by network bisection bandwidth which is the bandwidth between two equal parts of a network. It is important when in distributed computing with resources that are physically distributed. If we draw a line somewhere

in-between two set of resources in a distributed network, bisectional bandwidth is the rate of communication at which servers on one side of the line can communicate with servers on the other side as shown in Figure 14.1. For distributed computing to work efficiently, this is the most important parameter to consider. If we do not have enough network bisection bandwidth, the benefits gained by the availability of multiple execution engines in distributed computing will be overshadowed by slow communication links.

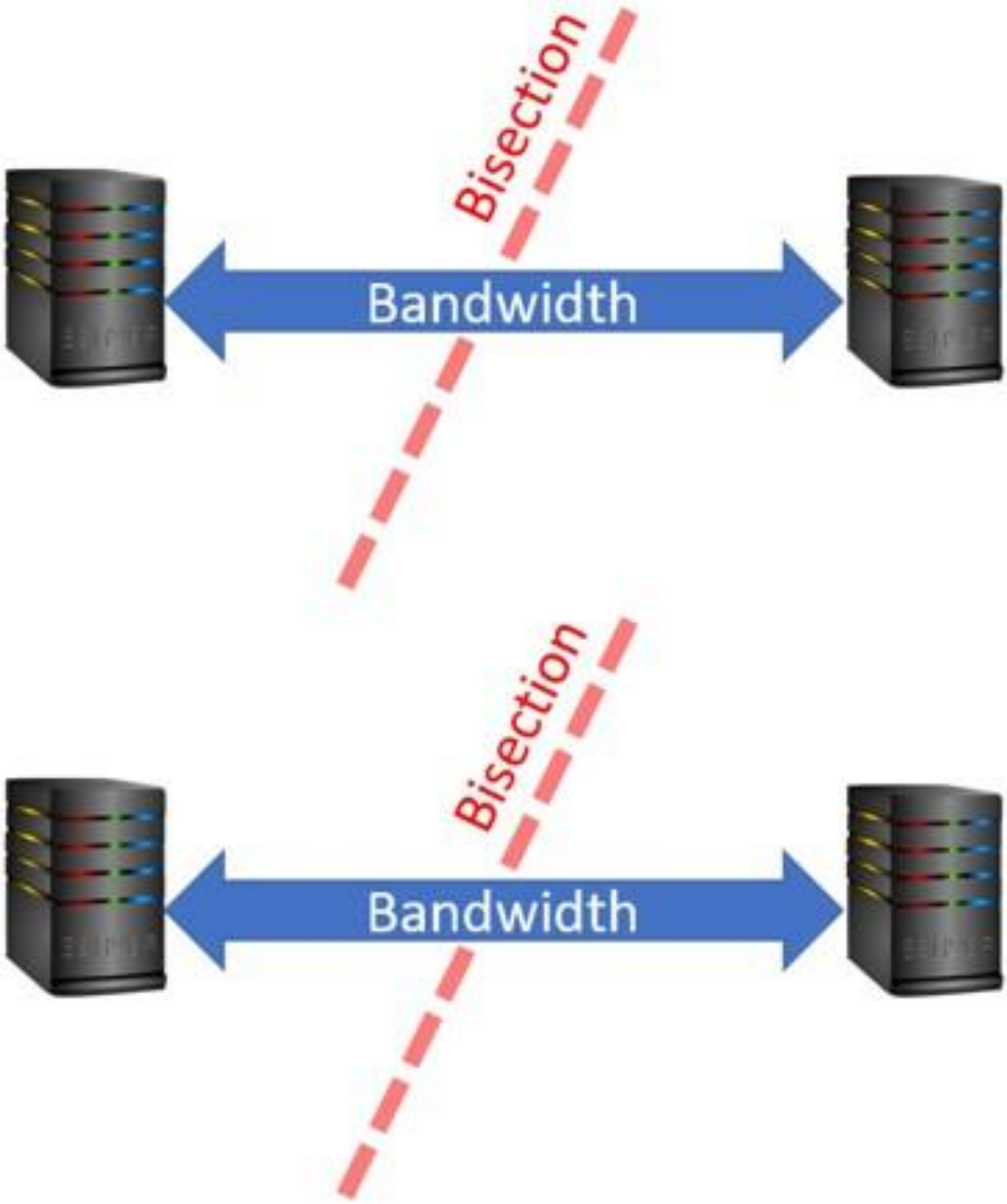


Figure 14.1: Bisection Bandwidth

High bisectional bandwidth enables us to process the data where it is without copying it. These days major cloud computing provides offer Google's network great bisection bandwidth. For example, within a Google

data center the bisection bandwidth is as high as 1 PB/s of bandwidth. The bisection bandwidth of other major cloud vendors is in this range as well.With petabit bisectional bandwidth the new options and design patterns for efficiently storing and processing big data emerge. The communication is so fast that data-locality principle that was always respected in the older Hadoop based Big Data architectures is no more relevant. It no longer makes sense to transfer files and store them locally. Instead, it makes sense to use the data from where it is stored.

Scaling Out

Horizontal scaling is used to divide a complex problem into smaller problems that can be allocated to more than one computing resources. For an efficient solution we are looking to implement the following two characteristics in our infrastructure:

A. Elasticity: Ability of the infrastructure to provisioning more or less resources according to the computing demands.
Б. Load Balancing: Utilizing all available resources in optimal way to optimize performance, availability, and reliability.

Let us look into them one by one:

Elasticity

Ability of the infrastructure to add or remove resources as the requirements changes is called elasticity. Autoscaling in Cloud computing is one way of achieving elasticity. Autoscaling is shown in Figure 14.2. Based on the need, the server group can adjust the number of nodes in the server group.

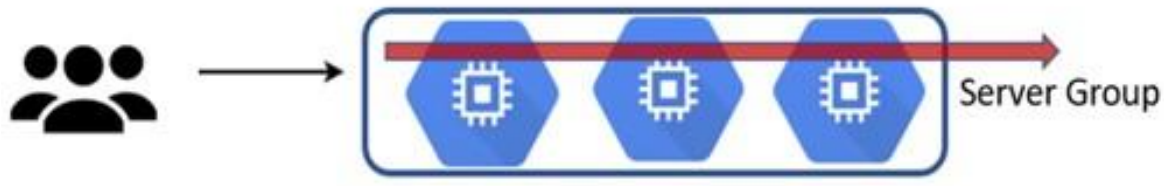


Figure 14.2: Server Autoscaling

*The three cloud computing giants, Google, Amazon, and Microsoft can provide highly elastic infrastructures. Due to the gigantic size of their shared resource pools, there are very few companies that have the potential to match the elasticity of infrastructure of these three companies.*If the infrastructure is elastic, it can create a scalable solution to the problem.

Load Balancing

Load Balancing is the mechanism of efficiently distributing workload among the computing resources available. There are different mechanisms that can be used to distribute the load. The simplest algorithm is the round robin, which is the workload is distributed equally to the available computing resources in a cyclic mechanism. This is shown in Figure 14.3.

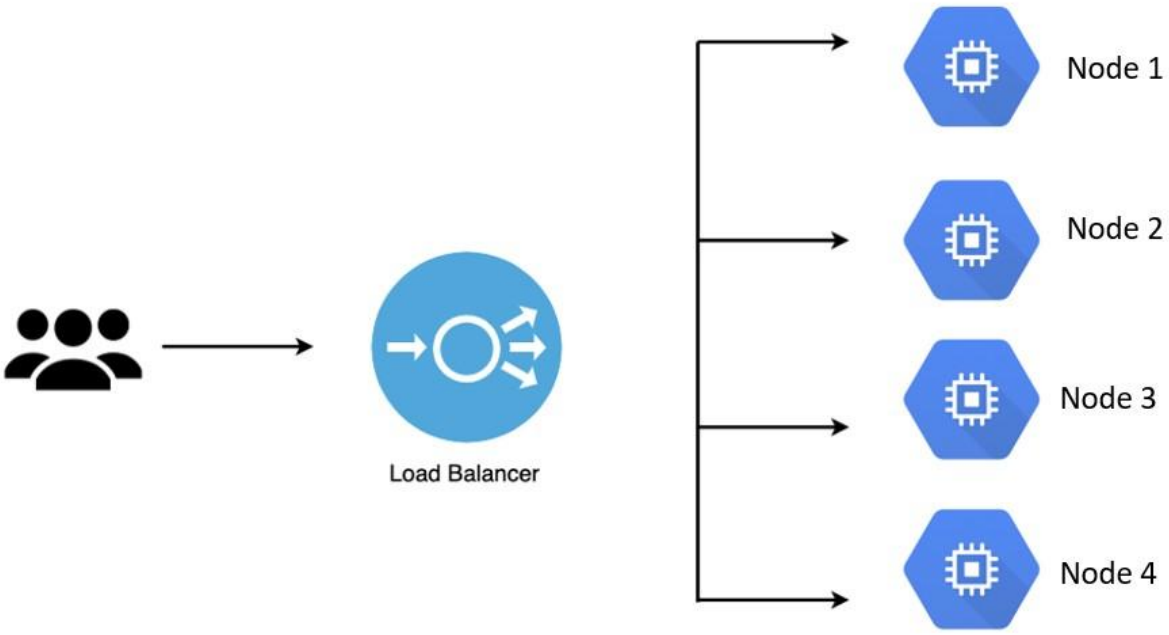


Figure 14.3: Load Balancing

There are two ways we can make load balancing more intelligent.By Adding Load Distribution Rules: (instead of doing it in simple cyclic manner) By Adding Server Autoscaling: This is shown in Figure 14.4.

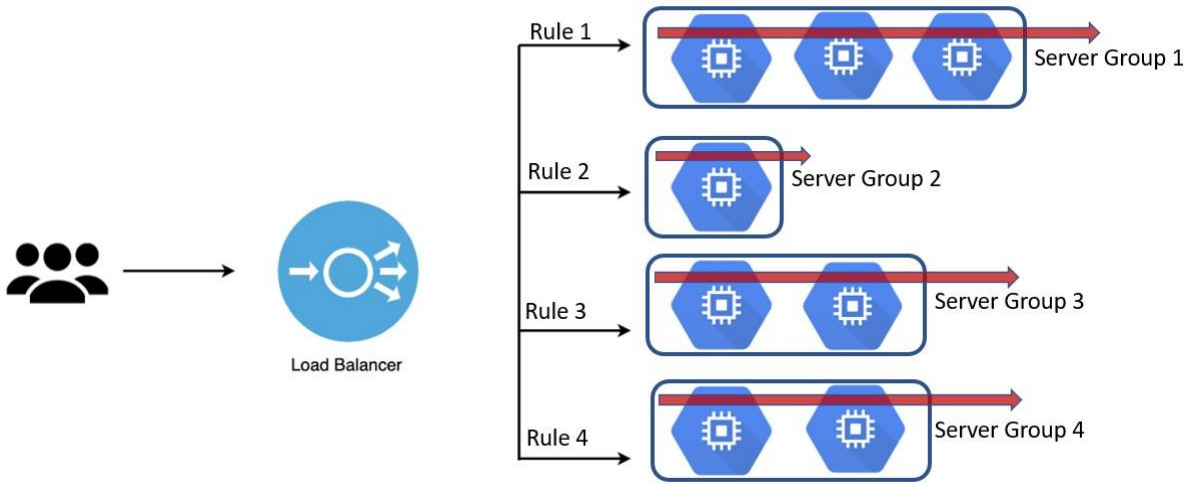


Figure 14.4: Intelligent Load Balancing with Rules and Server Autoscaling

The design of parallel algorithms

It is important to note that parallel algorithms are not a silver bullet. Even the best designed parallel architectures may not give the performance that we may expect. One law that is widely used to design parallel algorithms is Amdahl's law.

Amdahl's law

Gene Amdahl was one of the first people who studied parallel processing in the 1960s. He proposed Amdahl's law, which is still applicable today and can become a basis to understand the various trade-offs involved when designing a parallel computing solution. Amdahl's law can be explained as follows:It is based on the concept that in any computing process, not all of the processes can be executed in parallel. There will be a sequential portion of the process that cannot be parallelized.Let's look at a particular example. Assume that we want to read a large number of files stored on a computer and want to train a machine learning model using the data found in these files.This whole process is called P. It is obvious that P can be divided into the following two subprocesses:

- *P1*: Scan the files in the directory, create a list of filenames that matches the input file, and pass it on.

- *P2*: Read the files, create the data processing pipeline, process the files, and train the model.

Conducting sequential process analysis

The time to run *P* is represented by *Tseq(P)*. The times to run *P1* and *P2* are represented by *Tseq(P1)* and *Tseq(P2)*. It is obvious that, when running on a single node, we can observe two things:

- *P2* cannot start running before P1 is complete. This is represented by *P1 -- > P2*
- *Tseq(P) = Tseq(P1) + Tseq(P2)*

Let's assume that P overall takes 10 minutes to run on a single node. Out of these 10 minutes, P1 takes 2 minutes to run and P2 takes 8 minutes to run on a single node. This is shown in the following diagram:



Figure 14.5: Time taken to run P1 and P2 (for example, feel free to change)

Now the important thing to note is that *P1* is sequential in nature. We cannot make it faster by making it parallel. On the other hand, *P2* can easily be split into parallel subtasks that can run in parallel. So, we can make it run faster by running it in parallel.*The major benefit of using cloud computing is the availability of a large pool of resources and many of them are used in parallel. The plan to use these resources for a given problem is called an execution plan. Amdahl's law is used comprehensively to identify the bottlenecks for a given problem and a pool of resources.*

Conducting parallel execution analysis

If we want to use more than one node to speed up *P*, it will only affect *P2* by a factor of *s>1*:

$$T_{par}(P) = T_{seq}(P1) + \frac{1}{s} T_{seq}(P2)$$

The speedup of the process P can be easily calculated as follows:

$$S(P) = \frac{T_{seq}(P)}{T_{par}(P)}$$

The ratio of the parallelizable portion of a process to its total is represented by *b* and is calculated as follows:

$$b = \frac{T_{seq}(P2)}{T_{seq}(P)}$$

For example, in the preceding scenario, *b = 8/10 = 0.8.*Simplifying these equations will give us Amdahl's law:

$$S(P) = \frac{1}{1 - b + \frac{b}{s}}$$

Here, we have the following:

- *P* is the overall process.
- *b* is the ratio of the parallelizable portion of *P*. *s* is the
- speedup achieved in the parallelizable portion of *P*.

Let's assume that we plan to run the process P on three parallel nodes:

- *P1* is the sequential portion and cannot be reduced by using parallel nodes. It will remain at 2 seconds.
- *P2* now takes 3 seconds instead of 9 seconds.

So, the total time taken by process *P* is reduced to 5 seconds, as shown in the following diagram:

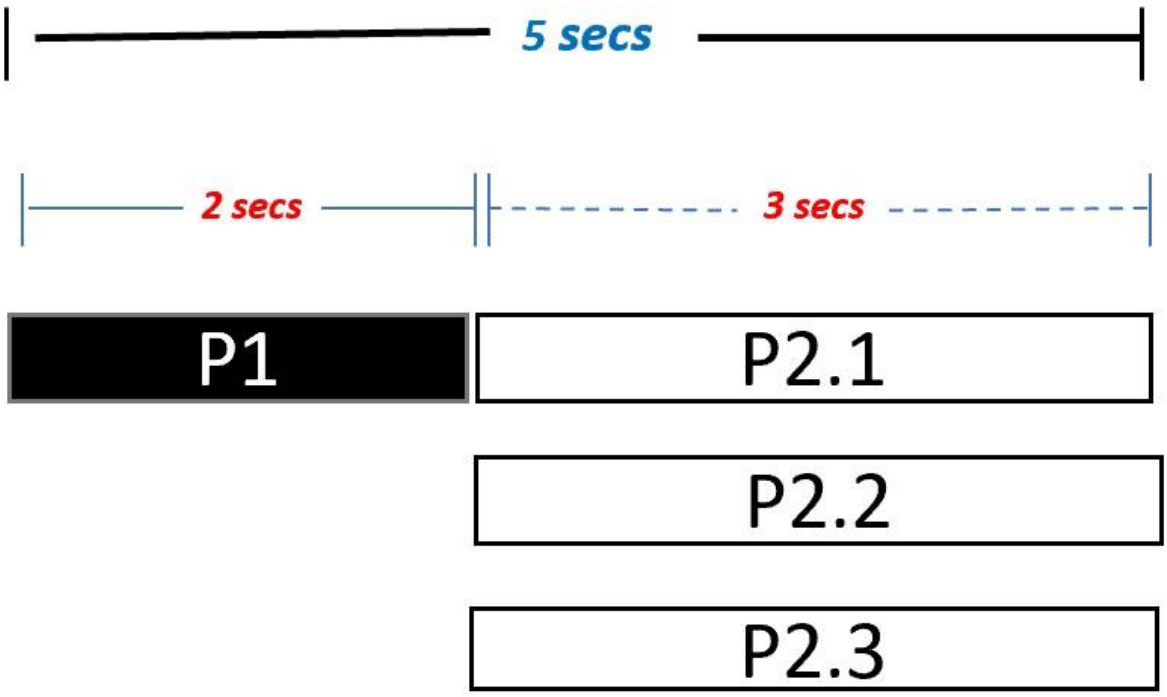


Figure 14.6: Reduced timespan of P (feel free to change)

In the preceding example, we can calculate the following:

- *np* = the number of processors = 3 *b* =
- the parallel portion = 9/11 = 81.81% *s* =
- the speedup = 3

Now, let's look at a typical graph that explains Amdahl's law:

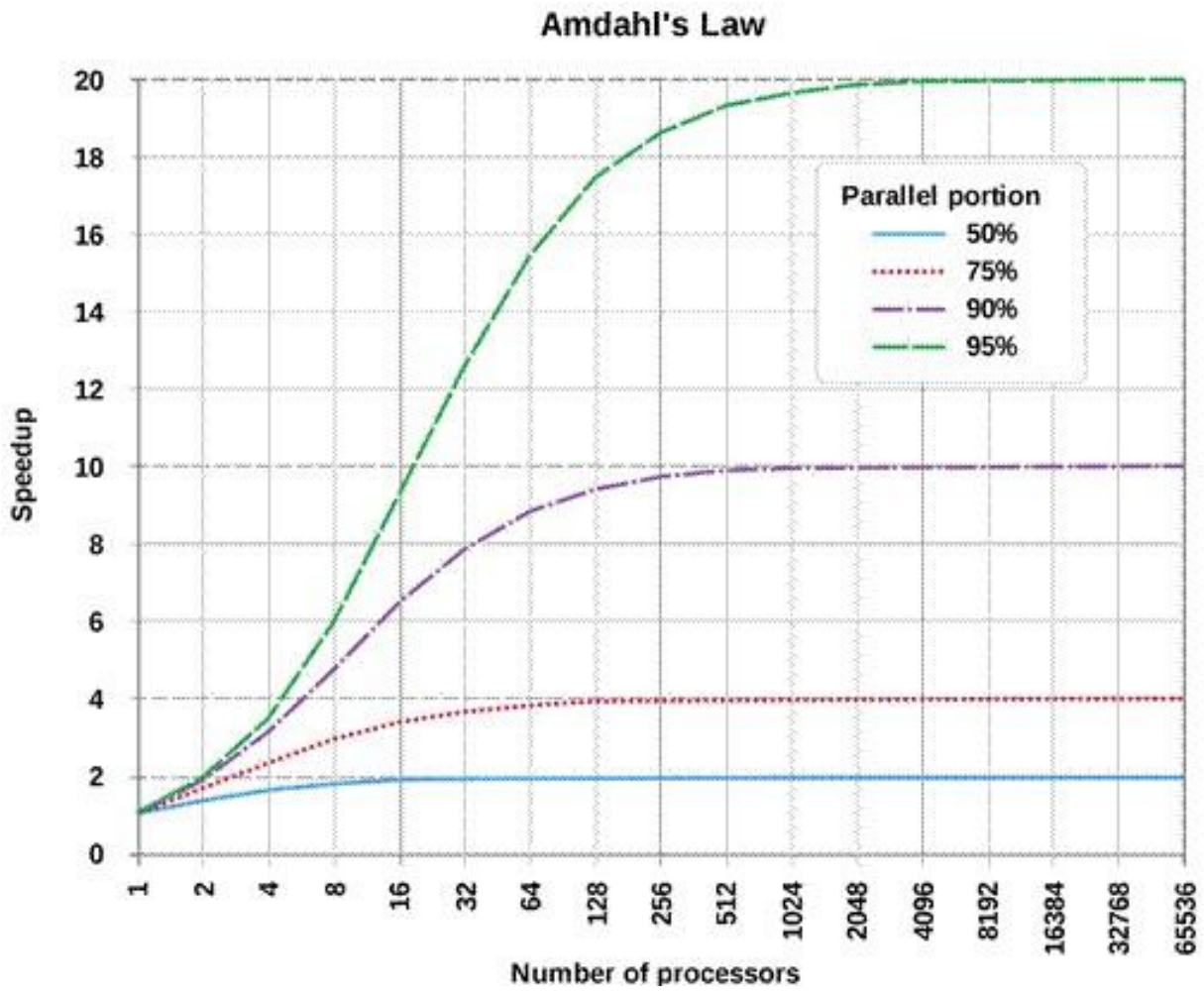


Figure 14.7: A graph Representing Amdahl's Law

In the preceding diagram, we draw the graph between *s* and *np* for different values of *b*.

Airflow- A Practical Application

Use Airflow to author workflows as Directed Acyclic Graphs (DAGs) of tasks. The Airflow scheduler executes your tasks on an array of workers while following the specified dependencies. Rich command line utilities make performing complex surgeries on DAGs a snap. The rich user interface makes it easy to visualize pipelines running in production, monitor progress, and troubleshoot issues when needed.DAGs are designed to be run many times, and multiple runs of them can happen in parallel. DAGs

are parameterized, always including an interval they are “running for” (the data interval), but with other optional parameters as well.Tasks have dependencies declared on each other. You’ll see this in a DAG either using the >> and << operators:

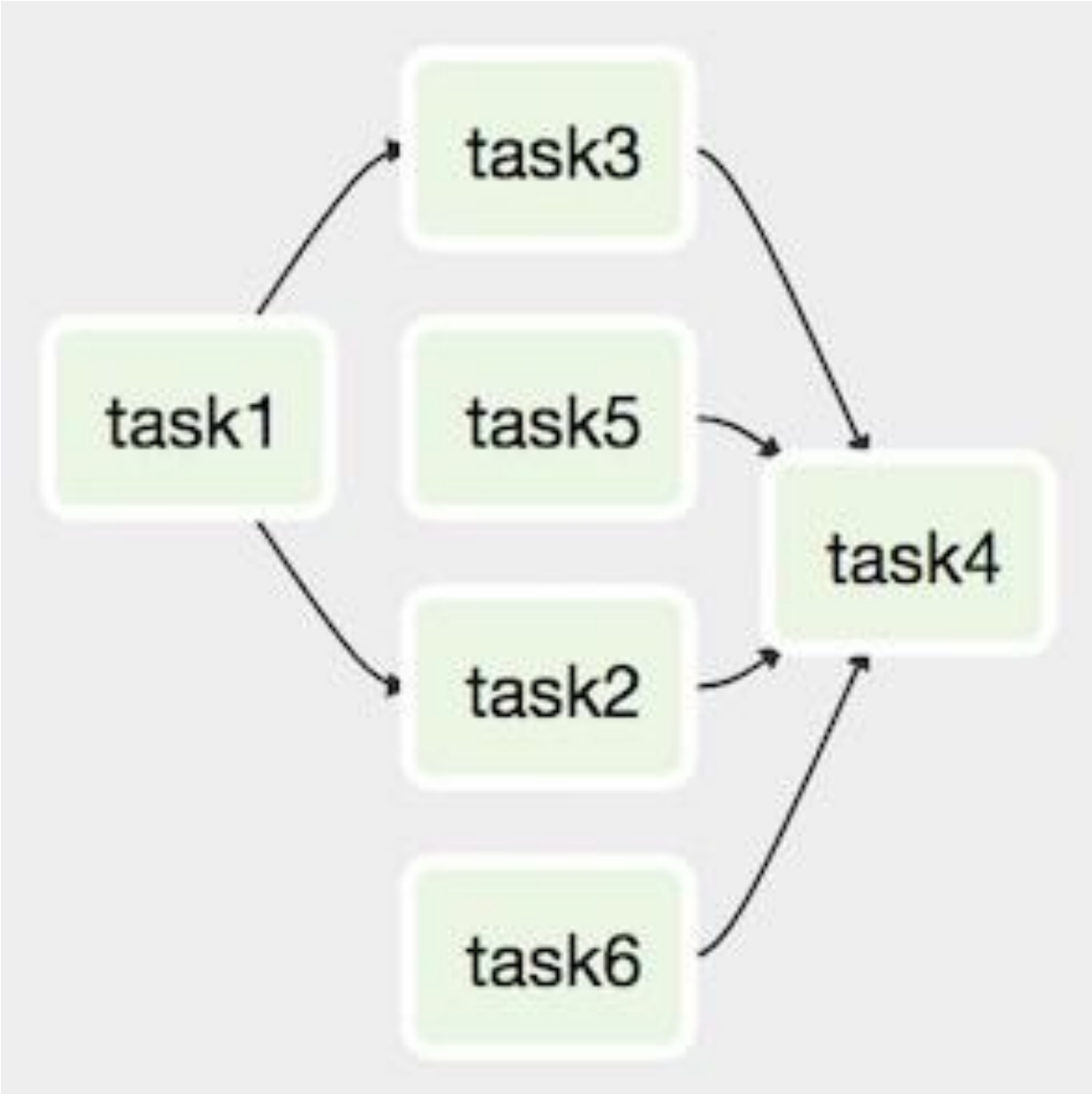


Figure 14.8: Representation of a DAG

Understanding task granularity

When we parallelize an algorithm, a larger job is divided into multiple parallel tasks. It is not always straightforward to find the optimal number of parallel tasks into which a job should be divided. If there are too few parallel tasks, we will not get much benefit from parallel computing. If there are too many tasks, then it will generate too much overhead. This is a challenge also termed as task granularity.

Enabling concurrent processing in Python

The easiest way to enable parallel processing in Python is by cloning a current process that will start a new concurrent process called the **child process**.*Python programmers, although not biologists, have created their own process of cloning. Just like a cloned sheep, the clone copy is the exact copy of the original process.*

Strategizing multi-resource processing

Initially, large-scale algorithms were used to run on huge machines called **supercomputers**. These supercomputers shared the same memory space. The resources were all local—physically placed in the same machine. It means that the communications between the various processors were very fast and they were able to share the same variable through the common memory space. As the systems evolved and the need to run large-scale algorithms grew, the supercomputers evolved into **Distributed Shared Memory** (**DSM**) where each processing node used to own a portion of the physical memory. Eventually, clusters were developed, which are loosely coupled and rely on message passing among processing nodes. For largescale algorithms, we need to find more than one execution engines running in parallel to solve a complex problem:

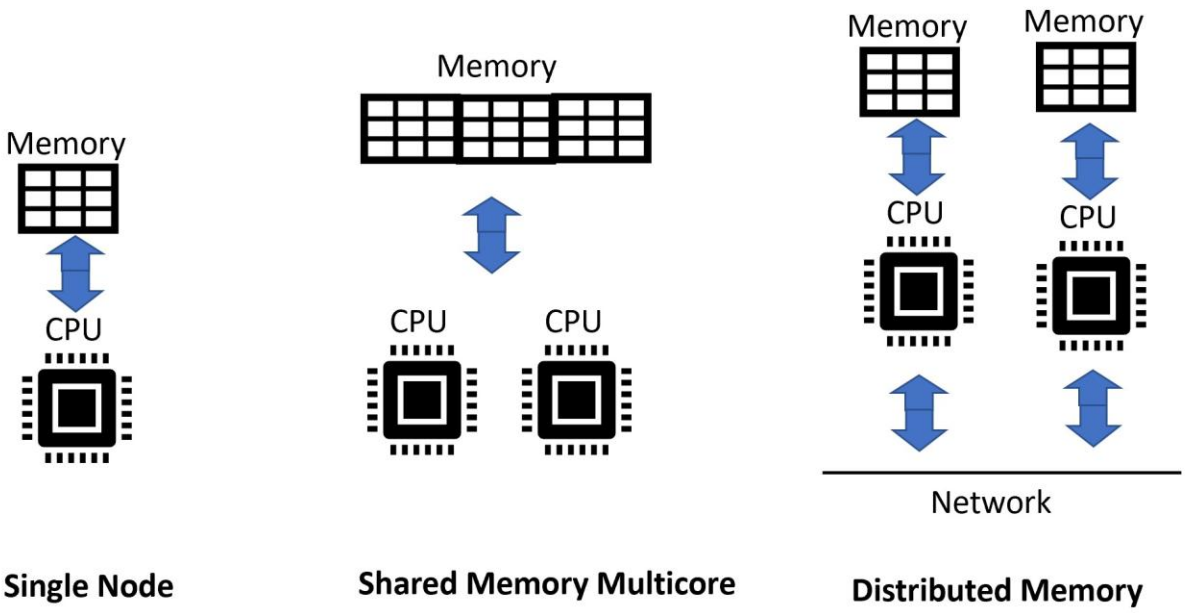


Figure 14.9:

There are three strategies to have more than one execution engine:

- **Look within**: Exploit the resources already on the computer. Use the hundreds of cores of the GPU to run a large-scale algorithm. • **Look outside**: Use distributed computing to find more computing resources that can be collectively used to solve the large-scale problem at hand.
- **Hybrid strategy**: Use distributed computing and, on each of the nodes, use the GPU or an array of GPUs to expedite the running of the algorithm.

Introducing CUDA

GPUs have been designed originally for graphic processing. They have been designed to suit the needs of optimization dealing with the multimedia data of a typical computer. To do so, they have developed certain characteristics that differentiate them from CPUs. For example, they have thousands of cores as compared to the limited number of CPU cores. Their clock speed is much slower than a CPU. A GPU has its own DRAM. For example, Nvidia's RTX 2080 has 8GB of RAM. Note that GPUs are specialized processing devices and do not have general processing unit

features, including interrupts or means of addressing devices, for example, a keyboard and mouse. Here is the architecture of GPUs:

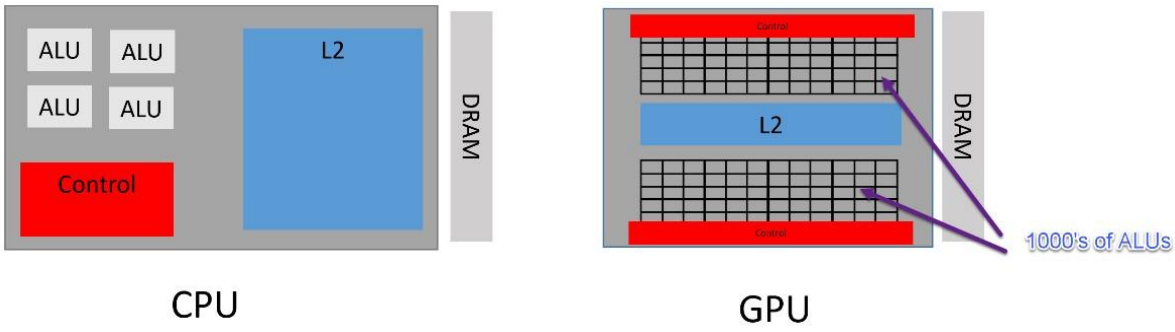


Figure 14.10: Architecture of GPUs

Soon after GPUs became mainstream, data scientists started exploring GPUs for their potential to efficiently perform parallel operations. As a typical GPU has thousands of ALUs, it has the potential to spawn 1,000s concurrent processes. This makes GPUs the architecture optimized for dataparallel computation. Hence, algorithms that can perform parallel computations are best suited for GPUs. For example, an object search in a video is known to be at least 20 times faster in GPUs, compared to CPUs. Graph algorithms, which were discussed in *Chapter 5* , *Graph Algorithms*, are known to run much faster on GPUs than on CPUs.To realize the dreams of data scientists to fully utilize GPUs for algorithms, in 2007, Nvidia created an open source framework called CUDA, which stands for Compute
Unified Device Architecture. CUDA abstracts the working of the CPU and GPU as a host and device respectively. The host, that is, the CPU, is responsible for calling the device, which is the GPU. The CUDA architecture has various layers of abstractions that can be presented as follows:

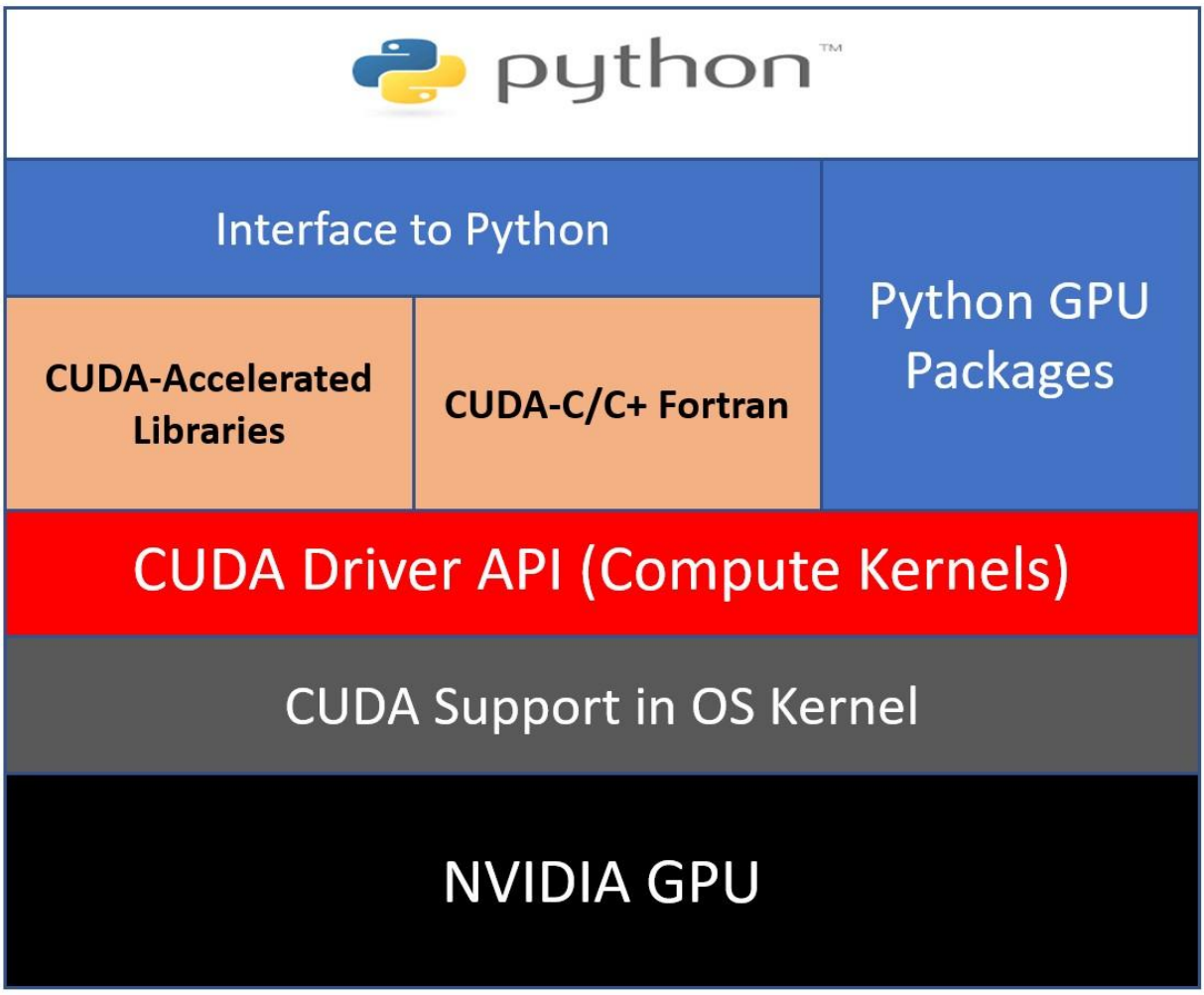


Figure 14.11: Architecture of a CUDA

Note that CUDA runs on top of Nvidia's GPUs. It needs support in the OS kernel. CUDA initially started with support in the Linux kernel. More recently, Windows is now fully supported. Then, we have the CUDA Driver API that acts as a bridge between the programming language API and the CUDA driver. On the top layer, we have support for C, C+, and Python.

Designing parallel algorithms on CUDA

Let's look deeper into how the GPU accelerates certain processing operations. As we know, CPUs are designed for the sequential execution of data that results in significant running time for certain classes of applications. Let's look into the example of processing an image of a size of

1,920 x 1,200. It can be calculated that there are 2,204,000 pixels to process. Sequential processing means that it will take a long time to process them on a traditional CPU. Modern GPUs such as Nvidia's Tesla are capable of spawning this unbelievable amount of 2,204,000 parallel threads to process the pixels. For most multimedia applications, the pixels can be processed independently of each other and will achieve a significant speedup. If we map each pixel with a thread, they can all be processed in O(1) constant time.But image processing is not the only application where we can use data parallelism to speed up the process. Data parallelism can be used in preparing data for machine learning libraries. In fact, the GPU can massively reduce the execution time of parallelizable algorithms, which include the following:

- Mining money for bitcoins
- Large-scale simulations
- DNA analysis
- Video and photos analysis

GPUs are not designed for **Single Program, Multiple Data** (**SPMD**). For example, if we want to calculate the hash for a block of data, it is a single program that cannot run in parallel. GPUs will perform slower in such scenarios.*The code that we want to run on the GPU is marked with special CUDA keywords called* **kernels***. These kernels are used to mark the functions that we intend to run on GPUs for parallel processing. Based on the kernels, the GPU compiler separates which code needs to run on the GPU and the CPU.*

Using GPUs for data processing in Python

GPUs are great for data processing in a multidimensional data structure. These data structures are inherently parallelizable. Let's see how we can use the GPU for multidimensional data processing in Python:

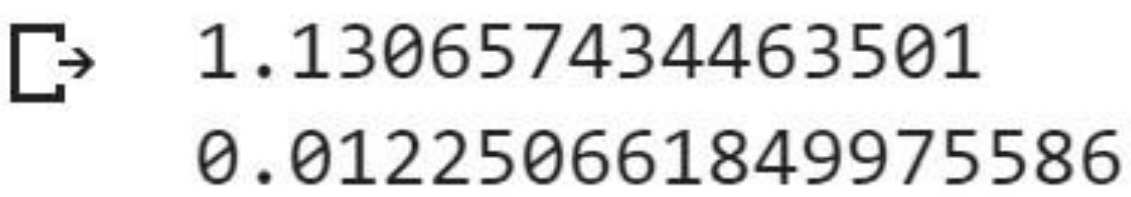
1. First, let's import the Python packages that are needed:

```
import numpy as np
import cupy as cp import
time
```

We will be using a multidimensional array in NumPy, which is a traditional Python package that uses the CPU.Then, we create a multidimensional array using a CuPy array, which uses the GPU. Then, we will compare the timings:

```
### Running at CPU using Numpy
start_time = time.time() myvar_cpu
= np.ones((800,800,800)) end_time
= time.time() print(end_time -
start_time) ### Running at GPU
using 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)
```

If we will run this code, it will generate the following output:



Note that it took around 1.13 seconds to create this array in NumPy and around 0.012 in CuPy, which makes the initialization of this array 92 times faster in the GPU.

Cluster computing

Cluster computing is one of the ways of implementing parallel processing for large-scale algorithms. In cluster computing, we have multiple nodes connected via a very high-speed network. Large-scale algorithms are submitted as jobs. Each job is divided into various tasks and each task is

run on a separate node.Apache Spark is one of the most popular ways of implementing cluster computing. In Apache Spark, the data is converted into distributed fault-tolerant datasets, which are called **Resilient Distributed Datasets** (**RDDs**). RDDs are the core Apache Spark abstraction. They are immutable collections of elements that can be operated in parallel. They are split into partitions and are distributed across the nodes, as shown here:

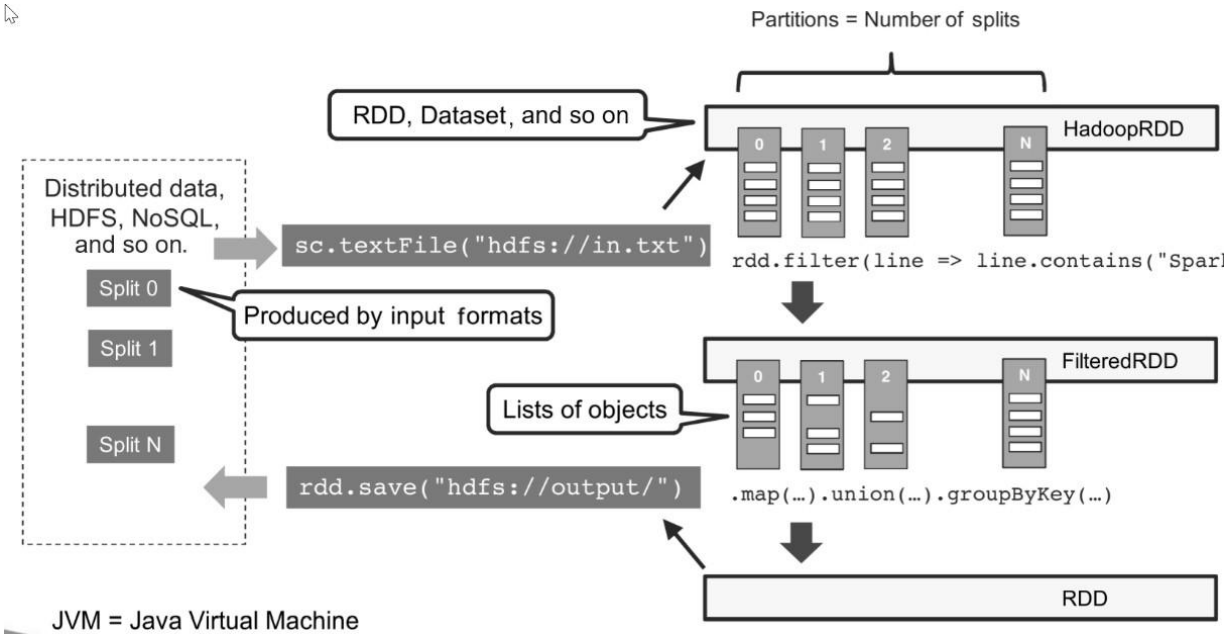


Figure 14.12:

Through this parallel data structure, we can run algorithms in parallel.

Implementing data processing in Apache Spark

Let's see how we can create an RDD in Apache Spark and run distributed processing on it across the cluster:

1. For this, first, we need to create a new Spark session, as follows:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('cloudanum').getOrCreate()
```

1. Once we have created a Spark session, we use a CSV file for the source of the RDD. Then, we will run the following function—it will create an RDD that is abstracted as a DataFrame called df. The ability to abstract an RDD as a DataFrame was added in Spark 2.0 and this

makes it easier to process the data: `df = spark.read.csv('taxi2.csv',inferSchema=True,header=True)` Let's look into the columns of the DataFrame:

```
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']
```

1. Next, we can create a temporary table from the DataFrame, as

follows: `df.createOrReplaceTempView("main")`

1. Once the temporary table is created, we can run SQL statements to process the data:

```
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|
+------------+-----+------------------+-----------------+
```

An important point to note is that although it looks like a regular DataFrame, it is just a high-level data structure. Under the hood, it is the RDD that spreads data across the cluster. Similarly, when we run SQL functions, under the hood, they are converted into parallel transformer and reducers and they fully use the power of the cluster to process the code.

The hybrid strategy

Increasingly, cloud computing is becoming more and more popular to run large-scale algorithms. This provides us with the opportunity to combine *look outside* and *look within* strategies. This can be done by provisioning one or more GPUs in multiple virtual machines, as shown in the following screenshot:

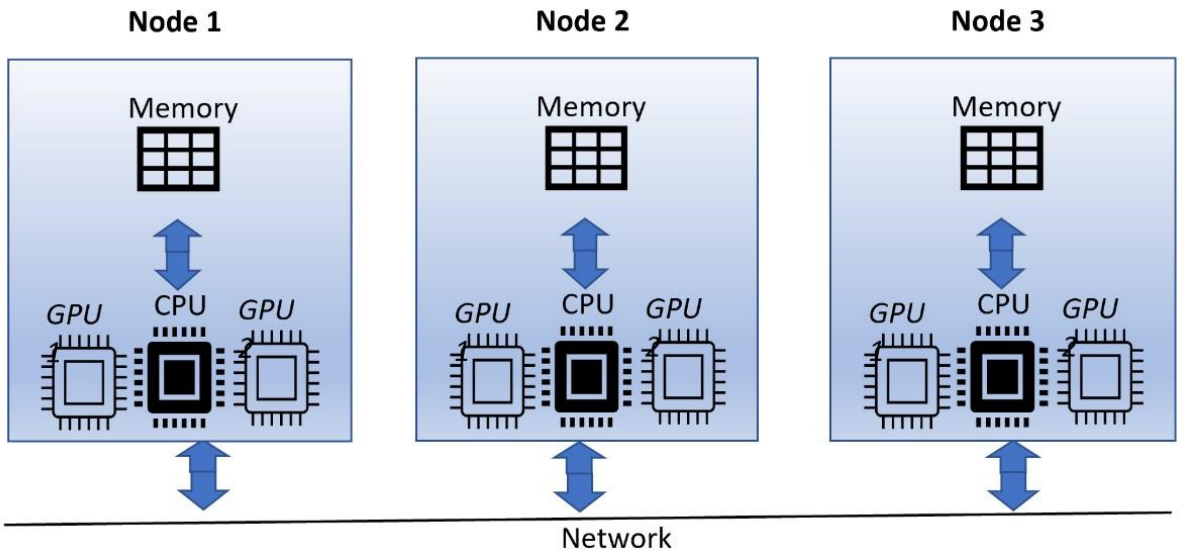


Figure 14.13: GPUs in multiple virtual machines

Making the best use of hybrid architecture is a non-trivial task. This is approached by first dividing the data into multiple partitions. Computeintensive tasks that require less data are parallelized within each node at the
GPUs.

Summary

In this chapter, we looked into the design of parallel algorithms and the design issues of large-scale algorithms. We looked into using parallel computing and GPUs to implement large-scale algorithms. We also looked into how we can use Spark clusters to implement large-scale algorithms.In this chapter, we learned about issues related to large-scale algorithms. We looked into the issues related to parallelizing algorithms and the potential bottlenecks created in the process.In the next chapter, we will look at some practical aspects of implementing algorithms.

OceanofPDF.com

15 Practical Considerations

Join our book's Discord space

https://packt.link/40Algos



There are a bunch of algorithms presented in this book that can be used to solve real-world problems. This chapter is about some practical considerations of the algorithms presented in this book.This chapter is organized as follows. We will start with an introduction. Then, we will present the issues around the of the explainability of an algorithm, which is the degree to which the internal mechanics of an algorithm can be explained in understandable terms. Then, we will present the ethics of using an algorithm and the possibility of creating biases when implementing them. Next, the techniques for handling NP-hard problems are discussed. Finally, we will investigate factors that should be considered before choosing an algorithm.By the end of this chapter, you will have learned about the practical considerations that are important to keep in mind when using algorithms to solve real-world problems.In this chapter, we will cover the following topics:

- Introducing practical considerations
- The explainability of an algorithm
- Understanding ethics and algorithms
- Reducing bias in models
- Tackling NP-hard problems When to use algorithms

Let's start with the introduction.

Introducing practical considerations

In addition to designing, developing, and testing an algorithm, in many cases, it is important to consider certain practical aspects of starting to rely on a machine to solve a real-world problem. For certain algorithms, we may need to consider ways to reliably

incorporate new important information that is expected to keep changing even after we have deployed our algorithm. We need to carefully consider that whether incorporating this new information will change the quality of our well-tested algorithm in any way? If so, how does our design going to handle it?

Expecting the Unexpected

Most solutions of real-world problems developed using algorithms are based on some assumptions. These assumptions may unexpectedly change after the model has been deployed. For some algorithms use assumptions that may be affected by changing global geo-political situation. For example, consider a trained model that is predict the financial profit for an international company with offices all over the world. An unexpected disruptive event like a war or spread of a sudden deadly virus may result in fundamentally changing the assumptions for this model t the quality of predictions. For such use-cases, the advice is to “expect the unexpected” and strategize for surprises. For certain datadriven models the surprise may come from changes in the regulatory polices after the solution has been deployed. *When we are using algorithms to solve a real-world problem, we are, in a way, relying on machines for problem solving. Even the most sophisticated algorithms are based on simplification and assumptions and cannot handle surprises. We are still not even close to fully handing over critical decision making to our own designed algorithms.*For example, Google's designed recommendation engine algorithms have recently faced the European Union's regulatory restrictions due to privacy concerns. These algorithms may be some of the most advanced in their field. But if banned, these algorithms may turn out to be useless as they cannot be used to solve the problems they were supposed to tackle. But, the truth of the matter is that, unfortunately, the practical considerations of an algorithm are still after-thoughts that are not usually considered at the initial design phase. For many use cases, once an algorithm is deployed and the short-term excitement of providing the solution is over, the practical aspects and implications of using an algorithm will be discovered over time and will define the success or failure of the project.Let's look into a practical example where not paying attention to the practical consideration failed a high-profile project designed by one of the best IT companies in the world.

The sad story of an AI Twitter Bot

Let's present the classical example of Tay, which was presented as the first-ever AI Twitter bot created by Microsoft in 2016. Being operated by an AI algorithm, Tay was supposed to learn from the environment and keep on improving itself. Unfortunately, after living in cyberspace for a couple of days, Tay started learning from the racism and rudeness of ongoing tweets. It soon started writing offensive tweets of its own. Although it exhibited intelligence and quickly learned how to create customized tweets based on real-time events, as designed, at the same time, it seriously offended people. Microsoft took it

offline and tried to re-tool it, but that did not work. Microsoft had to eventually kill the project. That was the sad end of an ambitious project.Note that although the intelligence built into it by Microsoft was impressive, the company ignored the practical implications of deploying a self-learning Twitter bot. The NLP and machine learning algorithms may have been the best in the class but due to the obvious shortcomings, it was practically a useless project. Today, Tay has become a textbook example of a failure due to ignoring the practical implications of allowing algorithms to learn on the fly. The lessons learned by the failure of Tay definitely influenced the AI projects of later years. Data scientists also started paying more attention to the transparency of algorithms. That brings us to the next topic, which explores the need and ways to make algorithms transparent.

The explainability of an algorithm

First, let us differentiate between a black box and a white box algorithm.

- A black box algorithm is one whose logic of is not interpretable by humans either due to its complexity or due to its logic being represented in a convoluted manner. A
- white box algorithm is one whose logic is visible and understandable for a human.

In other words, explainability helps the human brain to understand why an algorithm is giving specific results. The degree of explainability is the measure to which a particular algorithm is understandable for the human brain. Many classes of algorithms, especially those related to machine learning, are classified as black box. If the algorithms are used for critical decision-making, it may be important to understand the reasons behind the results generated by the algorithm. Converting black box algorithms into white box ones also provides better insights into the inner workings of the model. For example, an explainable algorithm will guide doctors as to which features were actually used to classify patients as sick or not. If the doctor has any doubts about the results, they can go back and double-check those particular features for accuracy.

Machine learning algorithms and explainability

The explainability of an algorithm has special importance for machine learning algorithms. In many applications of machine learning, users are asked to trust a model to help them to make decisions. Explainability provides transparency when needed in such use cases.Let's look deeper at a specific example. Let's assume that we want to use machine learning to predict the prices of houses in the Boston area based on their characteristics. Let's also assume that local city regulations will allow us to use machine learning algorithms only if we can provide detailed information for the justification of any predictions whenever needed. This information is needed for audit purposes to make sure that certain segments of the housing market are not artificially manipulated. Making our trained model

explainable will provide this additional information.Let's look into different options that are available for implementing the explainability of our trained model.

Presenting strategies for explainability

For machine learning, there are fundamentally two strategies to provide explainability to algorithms:

- **A global explainability strategy:** This is to provide the details of the formulation of a model as a whole.
- **A local explainability strategy:** This is to provide the rationale for one or more individual predictions made by our trained model.

For global explainability, we have techniques such as Testing with Concept Activation Vectors (TCAV), which is used for providing explainability for image classification models. TCAV depends on calculating directional derivatives to quantify the degree of the relationship between a user-defined concept and the classification of pictures. For example, it will quantify how sensitive a prediction of classifying a person as male is to the presence of facial hair in the picture. There are other global explainability strategies such as partial dependence plots and calculating the permutation importance, which can help to explain the formulations in our trained model. Both global and local explainability strategies can either be model-specific or model-agnostic. Model-specific strategies apply to certain types of models, whereas model-agnostic strategies can be applied to a wide variety of models.The following diagram summarizes the different strategies available for machine learning explainability:

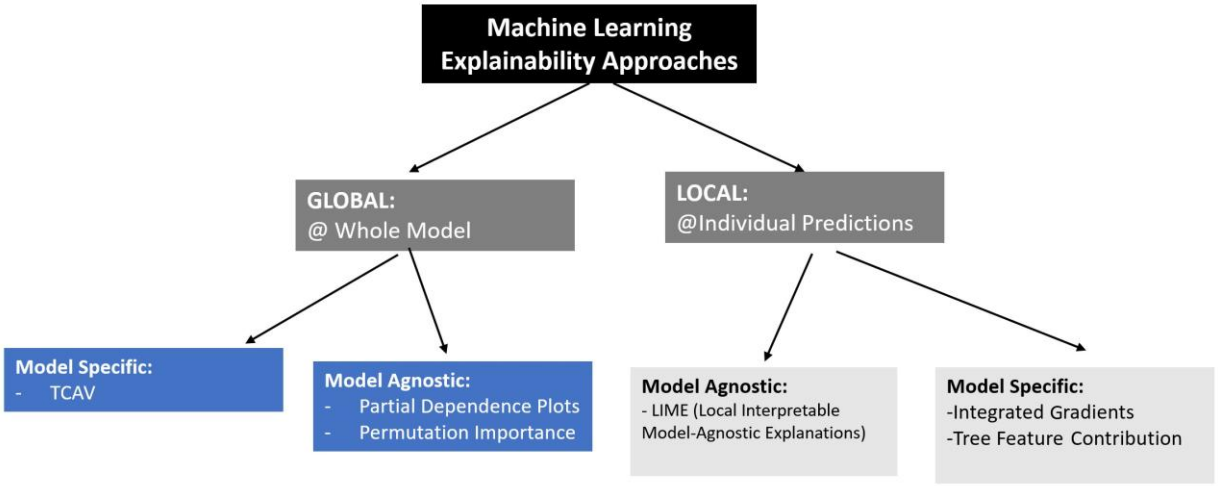


Figure 15.1:

Now, let's look at how we can implement explainability using one of these strategies.

Implementing explainability

Local Interpretable Model-Agnostic Explanations (LIME) is a model-agnostic approach that can explain individual predictions made by a trained model. Being model-agnostic, it can explain the predictions of most types of trained machine learning models.LIME explains decisions by inducing small changes to the input for each instance. It can gather the effects on the local decision boundary for that instance. It iterates over the loop to provide details for each variable. Looking at the output, we can see which variable has the most influence on that instance.Let's see how we can use LIME to make the individual predictions of our house price model explainable:

1. If you have never used LIME before, you need to install the package using `pip` :

```
!pip install lime
```

1. Then, let's import the Python packages that we need:

```
import sklearn as sk import
numpy as np
from lime.lime_tabular import LimeTabularExplainer as ex
```

1. We will train a model that can predict housing prices in a particular city. For that we will first import the dataset that is stored in the `housing.pkl` file. Then, we will explore the features it has:

```
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')
```

Based on these features we need to predict the price of a home.

1. Now, let's train the model. We will be using a random forest regressor to train the model. First we divide the data into testing and training partitions and then we using it to train the model:

```
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)
```

1. Next, let us identify the category columns:

```
 cat_col = [i for i, col in enumerate(housing.data.T)
                          if np.unique(col).size < 10]
```

1. Now, let's instantiate the LIME explainer with the required configuration parameters. Note that we are specifying that our label is `'price'` , representing the prices of houses in Boston:

```
myexplainer = ex(X_train,
    feature_names=housing.feature_names,
class_names=['price'],
categorical_features=cat_col,     mode='regression')
```

1. Let us try to look into the details of predictions. For that first let us import the pyplot as the plotter from matplotlib

```
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()
```

1. As the LIME explainer works on individual predictions, we need to choose the predictions we want to analyze. We have asked the explainer for its justification of the predictions indexed as `1` and `35` :

```
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()
```

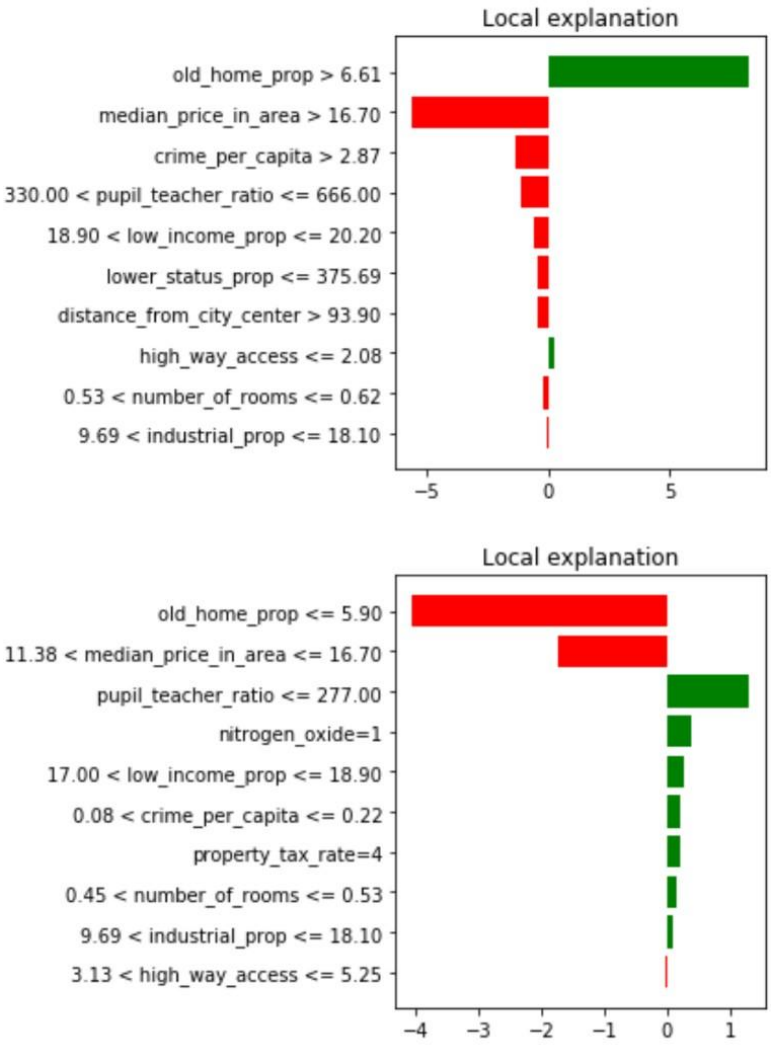


Figure 15.2:

Let's try to analyze the preceding explanation by LIME, which tells us the following:

- **The list of features used in the individual predictions**: They are indicated on the *y*axis in the preceding screenshot.
- **The relative importance of the features in determining the decision**: The larger the bar line, the greater the importance is. The value of the number is on the *x*-axis. **The**
- **positive or negative influence of each of the input features on the label**: Red bars show a negative influence and green bars show the positive influence of a particular feature.

Understanding ethics and algorithms

The formulation of patterns through algorithms may directly or indirectly result in unethical decision making. While designing algorithms, it is difficult to foresee the full scope of the potential ethical implications, especially for large-scale algorithms, where more than one user may be involved in the design. This makes it even more difficult to analyze the effects of human subjectivity.*More and more companies are making the ethical analysis of an algorithm part of its design. But the truth is that the problems may not become apparent until we find a problematic use case.*

Problems with learning algorithms

Algorithms capable of fine-tuning themselves according to changing data patterns are called learning algorithms. They are in learning mode in real time, but this real-time learning capability may have ethical implications. This creates the possibility that their learning could result in decisions that may have problems from an ethical point of view. As they are created to be in a continuous evolutionary phase, it is almost impossible to continuously perform an ethical analysis of them.*As the complexity of algorithms grows, it is becoming more and more difficult to fully understand their longterm implications for individuals and groups within society.*

Understanding ethical considerations

Algorithmic solutions are mathematical formulations without hearts. It is the responsibility of the people responsible for developing algorithms to ensure that they conform to ethical sensitivities around the problem we are trying to solve. These ethical considerations of algorithms depend on the type of the algorithm.For example, let's look into the following algorithms and their ethical considerations. Some examples of powerful algorithms for which careful ethical considerations are needed are as follows:

- Classification algorithms, when used on society, determine how individuals and groups are shaped and managed.
- Algorithms, when used in recommendation engines, can match resumes to job seekers, both individuals and groups.
- Data mining algorithms are used to mine information from users and are provided to decision-makers and governments.
- Machine learning algorithms are starting to be used by governments to grant or deny visas to applicants.

Hence, the ethical consideration, of algorithms will depend on the use case and the entities they directly or indirectly affect. Careful analysis is needed from an ethical point of view before starting to use an algorithm for critical decision-making. In the upcoming

sections, we shall see the factors that we should keep in mind while performing a careful analysis of algorithms.

Inconclusive evidence

The data that is used to train a machine learning algorithm may not have conclusive evidence. For example, in clinical trials, the effectiveness of a drug may not be proven due to the limited available evidence. Similarly, there may be limited inconclusive evidence that a certain postal code in a certain city is more likely to be involved in fraud. We should be careful when we are judging our decision-making based on the mathematical patterns found through algorithms using this limited data.*Decisions that are based on inconclusive evidence are prone to lead to unjustified actions.*

Traceability

The disconnection between the training phase and the testing phase in machine learning algorithms means that if there is some harm caused by an algorithm, it is very hard to trace and debug. Also, when a problem is found in an algorithm, it is difficult to actually determine the people who were affected by it.

Misguided evidence

Algorithms are data-driven formulations. The Garbage-in, Garbage-out (GIGO) principle means that results from algorithms can only be as reliable as the data on which they are based. If there are biases in the data, they will be reflected in the algorithms as well.

Unfair outcomes

The use of algorithms may result in harming vulnerable communities and groups that are already at a disadvantage.Additionally, the use of algorithms to distribute research funding has been proven on more than one occasion to be biased toward the male population. Algorithms used for granting immigration are sometimes unintentionally biased toward vulnerable population groups.Despite using high-quality data and complex mathematical formulations, if the result is an unfair outcome, the whole effort may bring more harm than benefit.

Reducing bias in models

In the current world, there are known, well-documented general biases based on gender, race, and sexual orientation. It means that the data we collect is expected to exhibit those biases unless we are dealing with an environment where an effort has been made to

remove these biases before collecting the data.All bias in algorithms is, directly or indirectly, due to human bias. Human bias can be reflected either in data used by the algorithm or in the formulation of the algorithm itself. For a typical machine learning project following the **CRISP-DM** (short for **Cross-Industry Standard Process**) lifecycle, which was explained in *Chapter 5*, *Graph Algorithms*, the bias looks like the following:

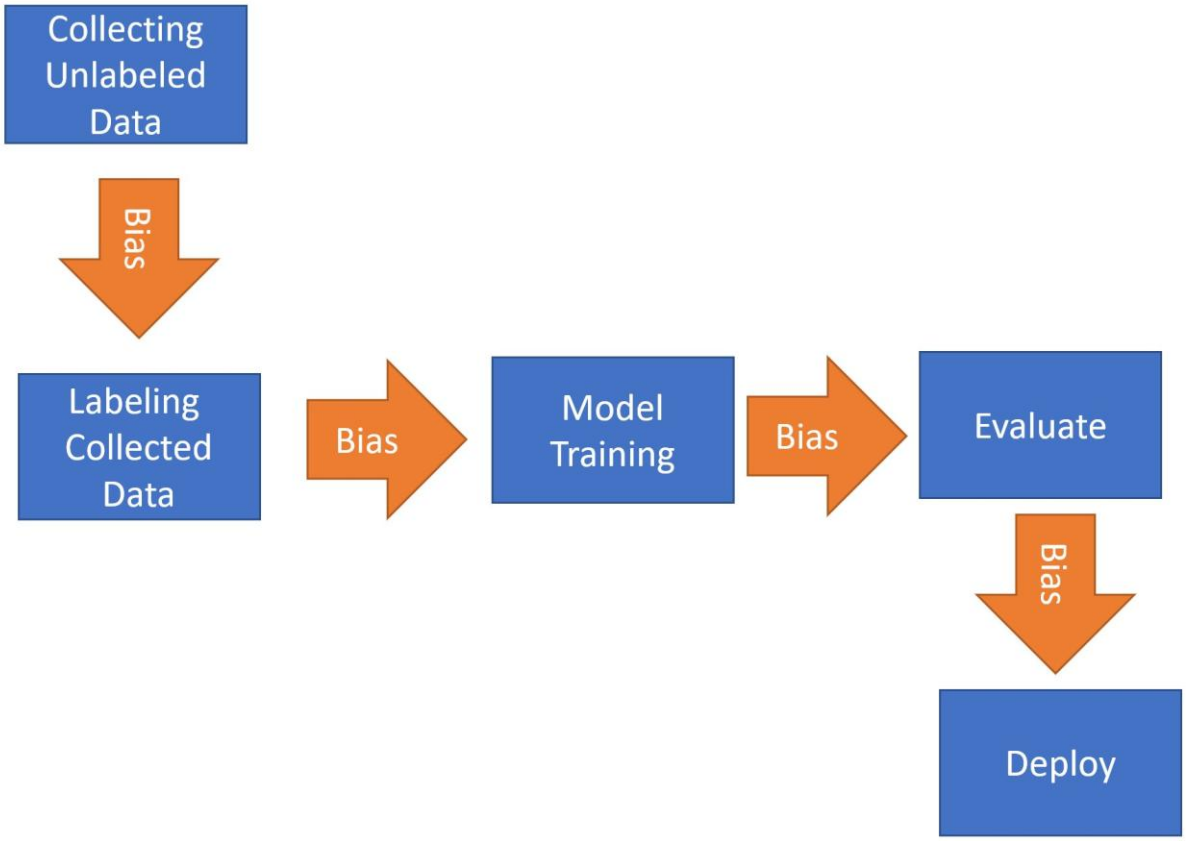


Figure 5.3: X

The trickiest part of reducing bias is to first identify and locate unconscious bias.

Tackling NP-hard problems

NP-hard problems were extensively discussed in *Chapter 4*, *Designing Algorithms*. Some NP-hard problems are important and we need to design algorithms to solve them.If finding the solution for an NP-hard problem seems out of reach due to its complexity or the limitations of the available resources, we can take one of these approaches:

- Simplifying the problem
- Customizing a well-known solution to a similar problem

- Using a probabilistic method

Let's look into them one by one.

Simplifying the problem

We can simplify the problem based on certain assumptions. The solved problem still gives a solution that is not perfect but is still insightful and useful. For this to work, the chosen assumptions should be as non- restrictive as possible.

Example

The relationship between features and labels in regression problems is seldom perfectly linear. But it may be linear within our usual operating range. Approximating the relationship as linear greatly simplifies the algorithm and is extensively used. But this also introduces some approximations affecting the accuracy of the algorithm. The trade-off between approximations and accuracy should be carefully studied and the right balance suitable for the stakeholders should be chosen.

Customizing a well-known solution to a similar problem

If a solution to a similar problem is known, that solution can be used as a starting point. It can be customized to solve the problem we were looking for. The concept of Transfer Learning (TL) in machine learning is based on this principle. The idea is to use the inference of already pre-trained models as the starting point for training the algorithm.

Example

Let's assume that we want to train a binary classifier that can differentiate between Apple and Windows laptops based on real-time video feed using computer vision during corporate training. From the video feed, the first phase of model development would be to detect different objects and identify which of the objects are laptops. Once done, we can move to the second phase of formulating rules that can differentiate between Apple and Windows laptops.Now, there are already well-trained, well-tested open source models that can deal with the first phase of this model training. Why not use them as a starting point and use the inference toward the second phase, which is to differentiate between Windows and Apple laptops? This will give us a jump start and the solution will be less error-prone as phase 1 is already well tested.

Using a probabilistic method

We use a probabilistic method to get a reasonably good solution that is workable, but not optimal. When we used decision tree algorithms in *Chapter 7*, *Traditional Supervised Learning Algorithms*, to solve the given problem, the solution was based on the probabilistic method. We did not prove that it is an optimal solution, but it was a reasonably good solution to give us a useful answer to the problem that we were trying to solve within the constraints defined in the requirements.

Example

Many machine learning algorithms start from a random solution and then iteratively improve the solution. The final solution might be efficient but we cannot prove that it is the best one. This method is used in complex problems to solve them in a reasonable timeframe. That is why, for many machine learning algorithms, the only way to get repeatable results is to use the same sequence of random numbers by using the same seed.

When to use algorithms

Algorithms are like tools in a practitioner's toolbox. First, we need to understand which tool is the best one to use under the given circumstances. Sometimes, we need to ask ourselves whether we even have a solution for the problem we are trying to solve and when the right time to deploy our solution is. We need to determine whether the use of an algorithm can provide a solution to a real problem that is actually useful, rather than the alternative. We need to analyze the effect of using the algorithm in terms of three aspects:

- **Cost**: Can use justify the cost related to the effort of implementing the algorithm?
- **Time**: Does our solution make the overall process more efficient than simpler alternatives?
- **Accuracy**: Does our solution produce more accurate results than simpler alternatives?

To choose the right algorithm, we need to find the answers to the following questions:

- Can we simplify the problem by making assumptions?
- How will we evaluate our algorithm? What are the key metrics?
- How will it be deployed and used?
- Does it need to be explainable?
- Do we understand the three important non-functional requirements—security, performance, and availability?

Is there any expected deadline?

A practical example – black swan events

Algorithms input data, process and formulate it, and solve a problem. What if the data gathered is about an extremely impactful and very rare event? How can we use algorithms with the data generated by that event and the events that may have led to that Big Bang? Let's look into this aspect in this section.Such extremely rare events were represented by the black swan events metaphor by Nassim Taleb in his book, Fooled by Randomness, in 2001.*Before black swans were first discovered in the wild, for centuries, they were used to represent something that cannot happen. After their discovery, the term remained popular but there was a change in what it represents. It now represents something so rare that it cannot be predicted*.Taleb provided these four criteria to classify an event as a black swan event.

Four criteria to classify an event as a black swan event

It is a bit tricky to decide whether a rare event should be classified as a black swan event or not. In general, in order to be classified as a black swan, it should meet the following four criteria.

A. First, once the event has happened, for observers it must be a mind-blowing surprise, for example, dropping the atom bomb on Hiroshima.
Б. The event should be a blockbuster—a disruptor and a major one, such as the outbreak of the Spanish Flu.
В. Once the event has happened and the dust has settled, data scientists who were part of the observer group should realize that actually it was not that much of a surprise. Observers never paid attention to some important clues. Had they the capacity and initiative, the black swan event could have been predicted. For example, the Spanish Flu outbreak had some leads that were known to be ignored before it became a global outbreak. Also, the Manhattan Project was run for years before the atomic bomb was actually dropped on Hiroshima. People in the observer group just could not connect the dots.
Г. When it happened, while the observers of the black swan event got the surprise of their lifetime, there may be some people for whom it may not have been a surprise at all. For example, for scientists working for years to develop the atomic bomb, the use of atomic power was never a surprise but an expected event.

Applying algorithms to black swan events

There are majors aspects of black swan events that are related to algorithms:

- There are many sophisticated forecasting algorithms available. But if we hope to use standard forecasting techniques to predict a black swan event as a precaution, it will not work. Using such forecasting algorithms will only offer false security.
- Once the black swan event has happened, predicting its exact implications on broader areas of society including the economy, the public, and governmental issues is not usually possible. First, being a rare event, we do not have the right data to feed to the algorithms, and we do not have a grasp of the correlation and interactions between broader areas of society that we may have never explored and understood.
- An important thing to note is that black swan events are not random events. We just did not have the capacity to pay attention to the complex events that eventually led to these events. This is an area where algorithms can play an important role. We should make sure that, in the future, we have a strategy to predict and detect these minor events, which combined over time to generate the black swan event.

*The COVID-19 outbreak of early 2020 is the best example of a black swan event of our times.*The preceding example shows how important it is to first consider and understand the details of the problem we are trying to solve and then come up with the areas where we can contribute toward a solution by implementing an algorithm-based solution. Without a comprehensive analysis, as presented earlier, the use of algorithms may only solve part of a complex problem, falling short of expectations.

Summary

In this chapter, we learned about the practical aspects that should be considered while designing algorithms. We looked into the concept of algorithmic explainability and the various ways we can provide it at different levels. We also looked into the potential ethical issues in algorithms. Finally, we described which factors to consider while choosing an algorithm.Algorithms are engines in the new automated world that we are witnessing today. It is important to learn about, experiment with, and understand the implications of using algorithms. Understanding their strengths and limitations and the ethical implications of using algorithms will go a long way in making this world a better place to live in. And this book is an effort to achieve this important goal in this ever-changing and evolving world.

OceanofPDF.com