Data Science for (not yet) scientists

Florian Huber

Nov 06, 2023

CONTENTS

I Introduction 3

1 Introduction: Data Science for Not-Yet-Scientists 5

2 What is Data Science? 7
- 2.1 Data is nothing new. So why now data science? . . . 8
- 2.2 A brief spotlight: the many facets of Data Science . . . 9

3 Data Science, Ethics, and Society 11
- 3.1 Let's look at some examples! . . . 11
- 3.2 Data Science comes with ethical considerations . . . 14

4 How to use this book (… if you ask us) 15
- 4.1 Skills you should have to work with this book . . . 15
- 4.2 Why Python? . . . 15
- 4.3 Open source libraries . . . 16
- 4.4 Setting Up Your Environment . . . 16

II Data Science Basics 19

5 Data and Data Types 21
- 5.1 History of Data . . . 21
- 5.2 Data Types . . . 21
- 5.3 Big Data . . . 23

6 Data - Information - Knowledge 25
- 6.1 From data to knowledge . . . 25

7 Data Science Workflow 27

III Data acquisition and first exploration 31

8 Data Acquisition & Preparation 33
- 8.1 Data Acquisition . . . 33
- 8.2 Data Cleaning . . . 34

9 First Data Exploration 37
- 9.1 Statistical Measures . . . 37
- 9.2 Statistical Measures and Distributions . . . 39
- 9.3 Statistical dispersion . . . 43

9.4 What can statistical measures do (and what not)? . . . 45
9.5 Comparing Distributions Visually . . . 46
9.6 Let's talk money . . . 49

IV In-depth Data Exploration 59

10 Correlation Analysis 61
10.1 Covariance, (Pearson) Correlation Coefficient . . . 61
10.2 Correlation Matrix . . . 62
10.3 What Does a Correlation Tell Us? . . . 67
10.4 Correlation vs. Causality . . . 67

11 Clustering 71
11.1 What is clustering? . . . 71
11.2 KMeans Clustering . . . 76
11.3 DBSCAN (Density-Based Spatial Clustering of Applications with Noise) . . . 82
11.4 Gaussian mixture models . . . 85
11.5 Hierarchical clustering (Ward) . . . 90
11.6 Comparison and Conclusion . . . 93

V Data Modeling 97

12 Dimensionality Reduction 99

13 Machine Learning 101

VI Working with text data 103

14 Introduction to Working with Text Data 105
14.1 1. Data and Structures as Strings . . . 105
14.2 2. Human Language in Text Form . . . 105
14.3 Basic Python string handling methods . . . 107
14.4 Regular Expressions (Regex) . . . 110

15 NLP - basic techniques to analyse text data 119
15.1 Example areas for the use of NLP techniques . . . 119
15.2 Python NLP libraries . . . 120
15.3 Tokenization, Stemming, Lemmatization . . . 121
15.4 NLTK . . . 121
15.5 NLP for languages other than English . . . 122
15.6 Applying SpaCy Models for Lemmatization . . . 123
15.7 Apply tokenization and lemmatization . . . 124
15.8 Mini-Exercise! . . . 126
15.9 Chapter Summary and Outlook . . . 126

16 Computing with Text: Counting words 127
16.1 Turning Text into Vectors . . . 127
16.2 One-Hot Encoding . . . 127
16.3 Term Frequency-Inverse Document Frequency (TF-IDF) . . . 128
16.4 Hands-on: Hotel Reviews . . . 129
16.5 Regression or Classification? . . . 136

17 Beyond Counting Individual Words: N-grams **147**
17.1 N-grams . . . 147
17.2 N-grams in TF-IDF Vectors . . . 148
17.3 TF-IDF with Bigrams: Growing Vectors and Managing High Dimensionality . . . 149
17.4 Find similar documents with tfidf . . . 161
17.5 Word Vectors: Word2Vec and Co . . . 163
17.6 Alternative short-cuts . . . 167

VII Look at the networks 171

18 Networks / Graph Theory **173**
18.1 Why Do We Need Graphs for Data Science? . . . 173
18.2 What is a Graph, What is a Network? . . . 173
18.3 Graphs in Python with `NetworkX` . . . 174
18.4 Not all graphs are connected . . . 179
18.5 Directed graphs . . . 180
18.6 Import network data . . . 182
18.7 Node importance . . . 183

19 Visualizing Graphs **185**
19.1 Add more dimensions to the visualization . . . 186
19.2 Force-directed / spring layout . . . 187
19.3 Visualization Tools . . . 189

20 Bottlenecks, Hubs, Communities **191**
20.1 Measuring the Structure of a Graph . . . 191
20.2 How important is a Node? . . . 191
20.3 What are the “most important” nodes? . . . 193
20.4 How important is an edge? . . . 194

VIII References 195

21 Acknowledgements **197**

22 Bibliography **199**

IX Source Code and Contributions 201

23 Source Code on GitHub **203**

24 Contribute **205**

Bibliography **207**

by Florian Huber

Düsseldorf University of Applied Sciences (HSD)
& Centre for Digitalization and Digitality (ZDD)

v0.5 2023-11-05

About me: I work as a professor for Data Science and Visual Analytics at the Düsseldorf University of Applied Sciences. This is also where I teach students the basics of data science, Python programming, machine learning, or where I give unsolicited advice on coffee, chocolate, and all other things that really matter in life.

Until I manage to either find or build a more suitable platform, you can also find me on Twitter/X: @me_datapoint.

Part I

Introduction

CHAPTER

ONE

INTRODUCTION: DATA SCIENCE FOR NOT-YET-SCIENTISTS

In today's world, data is generated at an unprecedented pace, and our ability to harness it is changing the way we live, work, and even think. Data science, the interdisciplinary field that blends statistics, computer science, and domain-specific knowledge, empowers us to extract insights from this vast ocean of data. As data science becomes increasingly essential across various industries and sectors, there is a growing need for skilled professionals who can make sense of data and transform it into actionable information. This book is designed to be your guide in this exciting and fast-evolving field.

There are numerous data science books, courses, and materials available, catering to different levels of expertise and backgrounds. However, many of these resources assume a strong foundation in computer science, math, or quantitative scientific disciplines. This is because until very recently such career shifts were the typical path to becoming a data scientist (also holds for the author of this book). But more and more universities or higher educational programs start to aim at the formation of a new generation of data scientists. Students which have no prior IT-related formation or scientific degrees. This book is for them! It fills the described gap by providing a comprehensive, hands-on introduction to data science for those who are just starting their journey or considering a career in this fascinating domain.

What sets this book apart from other resources is its focus on accessibility and practicality. We've designed it to be understandable to new undergraduate students, with only basic Python programming and math skills as requirements. You don't need to be an expert in computer science or have a strong background in statistics to grasp the concepts and techniques covered in this book.

Throughout the chapters, you'll find a wealth of hands-on examples and exercises that will help you develop a deep understanding of data science concepts and techniques. By working through these practical examples, you'll be able to apply your newly-acquired knowledge to real-world situations, making you better equipped to tackle data-driven challenges in your chosen field.

We firmly believe that becoming proficient in data science is within reach for anyone who possesses a combination of intellectual curiosity, a passion for learning, and a knack for logical puzzles or detective work. Additionally, a basic affinity for math and statistics goes a long way. Your academic or professional background doesn't define your potential in data science. Rather, your perseverance, your willingness to engage with complex problems, and your commitment to continuous learning will be your guiding stars.

This book represents our endeavor to dismantle any perceived barriers in your way, aiming to make data science comprehensible and accessible to a wide range of readers. We aim to empower aspiring data scientists to acquire and master the essential skills needed in an increasingly data-driven world.

So, are you ready to embark on your data science journey? Let's dive in and learn how to do the detective work of a data scientist to extract new knowledge from complex data.

CHAPTER

TWO

WHAT IS DATA SCIENCE?

Short answer, if we have to break it down to one sentence: **Data Science ≈ "Gaining and communicating insights from complex data through digital techniques"**.

Unlike mathematics, physics, or history, data science has just begun to become a discipline of its own. If you ask 10 practicing data scientists what data science is, you might get very different answers. Worse still, there's not even a consensus on whether data science is a field of its own, a technical approach, a mindset, or just a passing hype that will be named differently in five years. We will see a bit more about the different perspectives and origins of what data science is or isn't. But I want to start with the common basis. What is data science in general? I would answer that data science is essentially

> **the art of gaining and communicating insights from complex data through digital techniques**.

Many quantitative scientists could also argue that they often do exactly this. They aim to learn new things about the world from data. And the use of digital tools is also clearly no longer a significant difference. However, this does not argue against a field called "Data Science," but rather only says that many quantitative scientists nowadays are also to some extent data scientists. They even have to be if they want to keep up with the state of the art in their fields, as many research areas are currently undergoing rapid change due to the widespread adoption of new digital techniques such as machine learning approaches.

Beyond the short definition of data science mentioned, opinions on what data science exactly is converge a bit. Frequently this simply depends on the respective application area. Data science in consulting and business often means something different than data science in a more academic environment. However, in most cases, everyone can at least agree on a Venn diagram that is very often used in introductions in this - or slightly modified - form: Data science as the intersection of Digital Techniques (*digital tools/methods*), Statistics, and Domain Expertise.

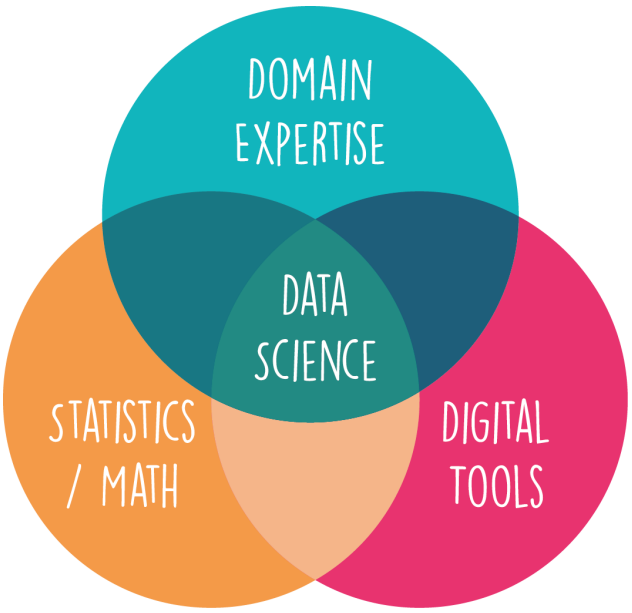


Figure 1. Venn diagram to indicate the intersection of fields for data science.

2.1 Data is nothing new. So why now data science?

Data has been a cornerstone of human understanding for millennia - from ancient civilizations keeping records of harvests and astronomy, to modern businesses tracking sales and performance. It's clear that data in itself is not a new concept. However, the emergence and ascendancy of data science as a discipline is a relatively recent phenomenon. So, why now?

The prominence of data science in today's world can be attributed to several concurrent developments:

(1) The exponential **increase in the volume of data generated**. Thanks to digitalization and the rise of the Internet, mobile devices, and IoT (Internet of Things), we are producing data at a previously unimaginable scale. This big data presents both a challenge and an opportunity - the challenge being how to handle and process this vast amount of information, and the opportunity being the valuable insights that can be gleaned from it.

This is accompanied by an increased recognition of the importance of data-driven decision-making across diverse sectors. Various industries, governments, and institutions have realized that leveraging the power of data can lead to increased efficiency, better decision-making, and provide a competitive advantage.

This existence (and appreciation) of larger and larger amounts of data can be seen as a substrate for the rise of data science, but it really needed a combination of several other developments to be able to properly work with such data (Fig. 2.1).

(2) The evolution and expansion of **statistical methodologies** have been a key driver. Statistics provide the foundational principles and techniques for analyzing data, making inferences, and predicting future trends. In the era of big data, classical and modern statistical techniques form the backbone of most analyses in data science. The relationship is actually so close, that in the 1990s statisticians like Jeff Wu even suggested renaming statistics to "Data Science" (despite all overlap, both terms still exist, and mean related but different things).

(3) The strides we've made in **data handling capabilities** have greatly facilitated the rise of data science. This obviously includes the drastic advancements in computational power and storage capabilities which made it possible to collect, store, and analyze these massive datasets. But this also includes many developments from computer science such as databases. Just a few decades ago, collecting, storing, and analyzing the vast amounts of data we deal with today would have been unimaginable, let alone impractical.

(4) There has been significant progress in the field of **algorithms** which also includes machine learning. It is algorithms, which are at the hear of nearly every tool that we use as data scientists for understanding and interpreting data. This can be optimization methods dating back more than 200 years (e.g., least square method) all the way to current deep learning approaches. These advancements have opened up new possibilities for predictive analytics, automation, and artificial intelligence.

(5) Lastly, the often-underestimated field of **data visualization** has seen revolutionary advancements. Effective data visualization makes complex data more comprehensible, accessible, and actionable. The development of powerful visualization tools enables us to present data in a visually compelling manner that fosters understanding and drives informed decisions.

So, while data is not new, the volume of data, our ability to process it, and the recognition of its value, are. These changes have given rise to the burgeoning field of data science, marking a new era in our relationship with data.

[1] Images of Thomas Bayes and Carl Friedrich Gauß taken from Wikipedia. For Thomas Bayes it is not even fully certain that the image actually of him.

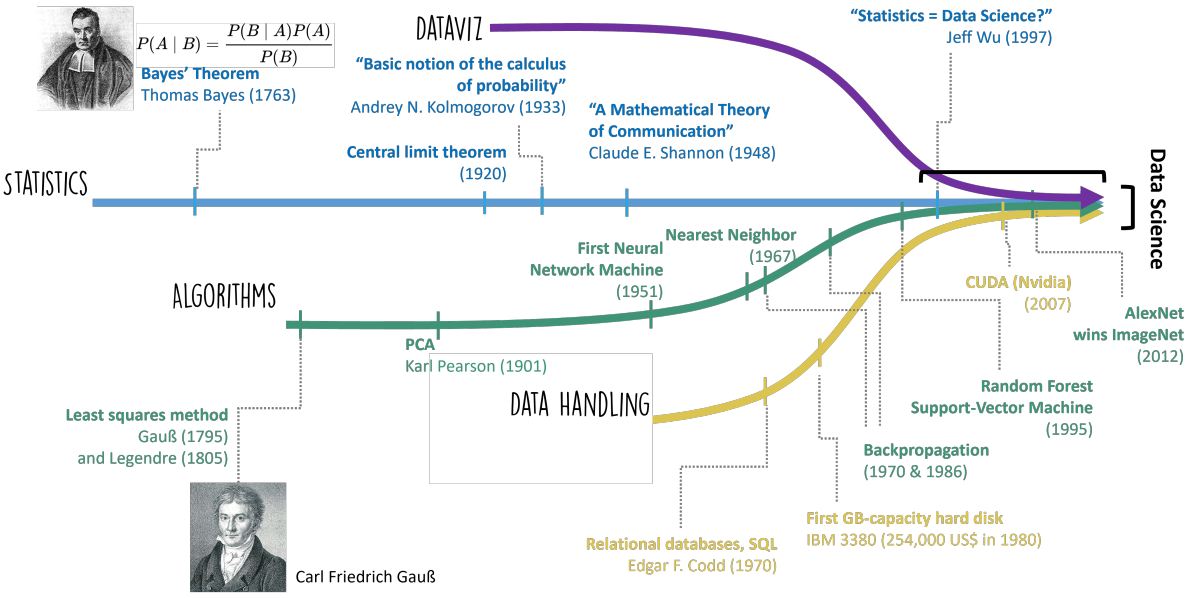


Fig. 2.1: The concurrent developments leading to Data Science[1].

2.2 A brief spotlight: the many facets of Data Science

Data science, by its very nature, stands at the bustling intersection of digital techniques, statistical methodologies, and domain expertise. It is a broad and incredibly diverse field with intricate links to many different sectors and disciplines. This diversity results in a wide variety of roles and responsibilities, each bringing unique skills and viewpoints to address an array of challenges and opportunities.

One of the key characteristics that makes data science so dynamic is its inherent multidisciplinarity. Data science **isn't just about dealing with numbers or coding**—it's about leveraging a suite of digital tools and statistical methods to draw insights from data, and applying these insights to a specific context or domain. A data scientist working in healthcare, for instance, might use different techniques and have a different focus than a data scientist working in retail or finance. The beauty of data science lies in this versatility—it is a field where myriad skills and disciplines converge and collaborate.

Given the breadth and depth of the field, being a successful data scientist often requires more than just technical skills. A natural curiosity to explore and understand data, openness to new ideas and methods, the eagerness to continuously learn and adapt, and most crucially, the ability to communicate and collaborate effectively are all vital attributes. After all, data science is a team sport. **No single person can master all facets of data science**; instead, it's about bringing diverse skills together, working with others, and learning from each other.

In the forthcoming pages, we aim to guide you through the multifaceted world of data science, shedding light on its various dimensions, the skills required, and the myriad ways in which data science can be applied. As you delve deeper into this exciting field, we hope to inspire you with the potential and the possibilities of data science, and prepare you for a journey of continuous learning and discovery. Welcome to the exciting world of data science!

CHAPTER

THREE

DATA SCIENCE, ETHICS, AND SOCIETY

In a world that is increasingly data-driven, the role of data science extends beyond pure analysis and prediction. The use of data and algorithms is intimately entwined with ethical considerations and societal implications. The power of data science to influence and shape our world is vast, and, as the Spider-Man quote goes: *"with great power comes great responsibility"* (see wikipedia).

Analyzing online user behavior, studying internal corporate processes, evaluating political campaigns, developing algorithms in sectors such as medicine and research, and communicating scientific content — all of these tasks can be performed using data science. However, each of these also brings its own ethical questions and societal implications.

Let's delve into each of these aspects in a bit more detail:

- **Analyzing Online User Behavior**: The analysis of online user behavior can offer significant insights into consumer preferences and habits, enabling businesses to tailor their offerings and improve their services. However, this needs to be balanced with privacy considerations and the need to obtain informed consent from users.
- **Studying Internal Corporate Processes**: Data science can be used to optimize corporate processes, increasing efficiency and productivity. But it's also essential to consider the implications for employees, such as changes in work patterns or privacy concerns.
- **Evaluating Political Campaigns**: The use of data science in political campaigns can help parties to understand the needs and preferences of the electorate. However, the misuse of personal data or the propagation of misinformation poses ethical dilemmas.
- **Developing Algorithms in Medicine and Research**: Algorithms can be powerful tools in medicine and research, enabling accurate diagnoses and efficient drug discovery. Nevertheless, considerations around data privacy, informed consent, and the potential for bias in algorithmic decision-making are critical.
- **Communicating Scientific Content**: Data visualization can help communicate complex scientific concepts effectively. Yet, it's crucial to present data accurately and transparently to avoid misinterpretation or misuse.

Therefore, the reach of data science is not limited to technology or business; it has immediate societal implications. But let's have a closer look at some specific examples to get a better idea of what this actually means.

3.1 Let's look at some examples!

3.1.1 Targeted Advertisement - good, bad, neutral?

Even before the digital revolution, targeted advertising was a commonly used strategy. Businesses would use basic demographic information, such as age, gender, and location, to tailor their messages to specific audiences. However, with the advent of the internet and the surge in digital data, the face of targeted advertising has transformed completely. Today's targeted ads are not just influenced by basic personal features but are significantly shaped by individual online behaviors, including search histories, website visits, and social media interactions Fig. 3.1. This evolution has ushered in a new era of personalized advertising, elevating its precision and relevance to unprecedented levels.

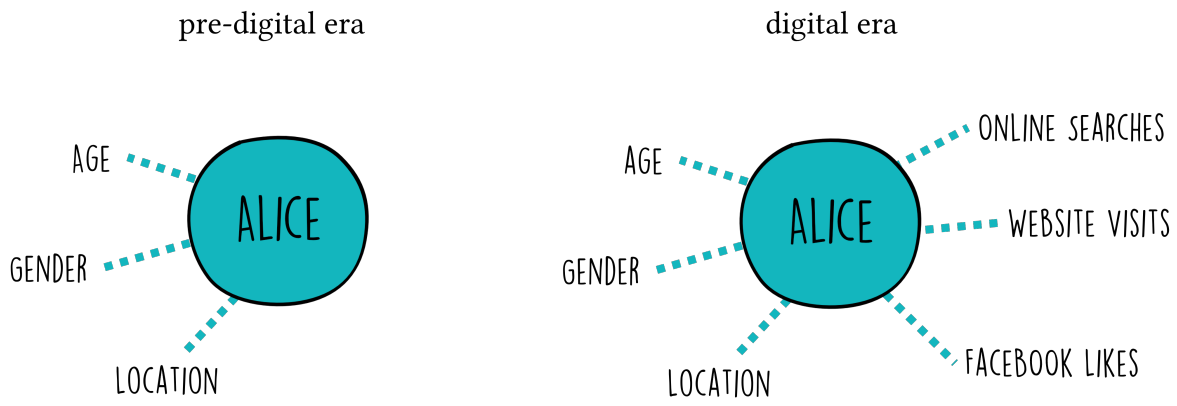


Fig. 3.1: Targeted advertisement can nowadays use much more data and information on a potential customer (here: Alice) when compared to pre-digital times.

You may already be feeling a touch of unease. Is it truly beneficial for companies or other organizations to have access to all this data? On the flip side, consider this: isn't it a **win/win situation**? Companies can reach a much more suitable audience, and individuals receive advertisements tailored to their profiles and interests. Take, for example, Alice (Fig. 3.2). Because of her age (neither too young nor too old), location (within 2 hours of the festival venue), and interests (many likes for Hip-Hop bands), she receives an advertisement for a Hip-Hop festival. This offer is a clear match for her. However, Alice wouldn't get a mail catalog for a package tour operator catering to a different customer profile—typically older and, let's say, less concerned about the carbon emissions of their tours. That's a plus for Alice. And a plus for both companies, isn't it?

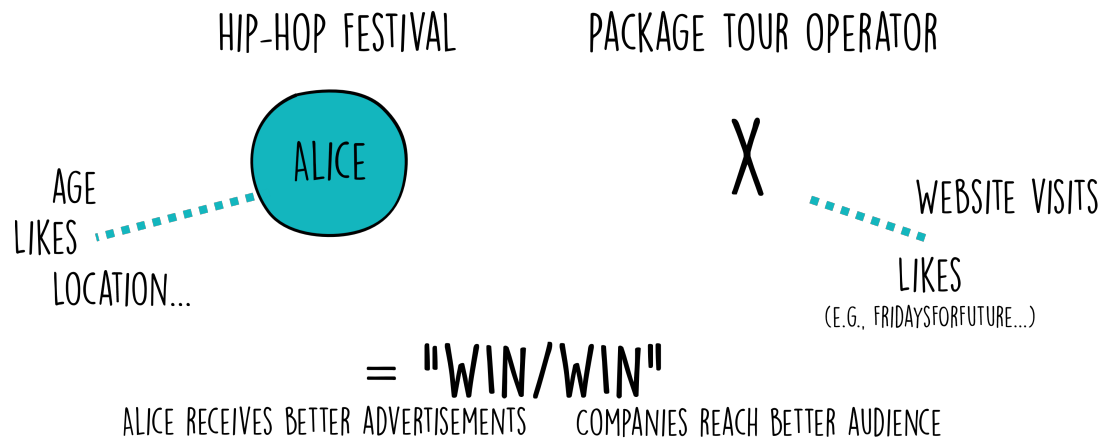


Fig. 3.2: In many cases target advertisement does little harm. It might even seem like a win/win situation at first.

However, things get a bit more complex when we consider the marketing of more sensitive products. Let's take the example of maternity wear. Suppose a company uses data science to detect early signs of pregnancy, like changes in online shopping behavior. They might then start sending ads for maternity clothes. This could potentially lead to uncomfortable or even harmful situations (Fig. 3.3).

For instance, what if the individual hasn't yet told their family or friends about the pregnancy?[1].Or, in a more distressing scenario, what if the pregnancy is unwanted, or complications arise? In such cases, receiving maternity ads could cause

[1] Such a case around a company called "target" appeared in 2012 in the New York Times and was often refereed to in debates around the power of "big data" in such a context. Worth noting maybe, that not everyone fully believed the story itself, e.g. this blog post.

emotional distress. And it's not limited to maternity products; this applies to any sensitive product or service.

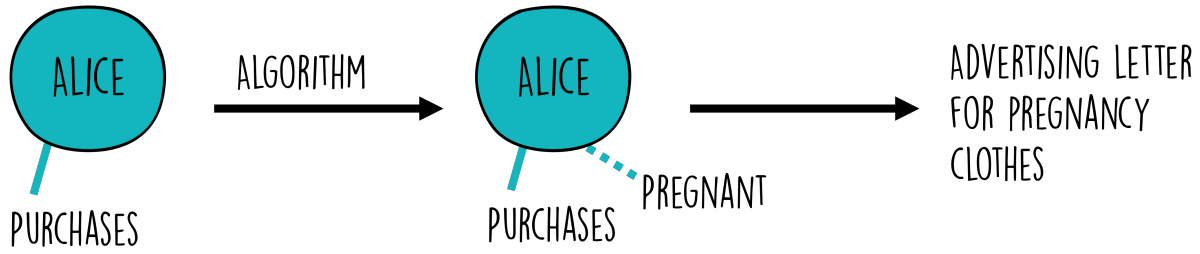


Fig. 3.3: But even advertisements can quickly turn into ethically much more complicated matters.

And what if the advertisement is not product in a classical sense, but a high-paying job? Consider a study using AdFisher, an automated tool designed to scrutinize the interplay between user behavior, Google ads, and the Ad Settings that Google provides for user control and transparency [Datta *et al.*, 2015]. During the course of the research, it was discovered that modifying the user profile to 'female' resulted in fewer advertisements for high-paying jobs compared to when it was set to 'male' (Fig. 3.4). This highlighted a potentially discriminatory consequence of algorithmic decision-making, underscoring the need for ethical considerations in data science. Yet, the complexity of the digital ad ecosystem, with its numerous players, rendered it challenging to definitively pin down the origin of this bias, illustrating the multifaceted nature of ethical issues in our field.

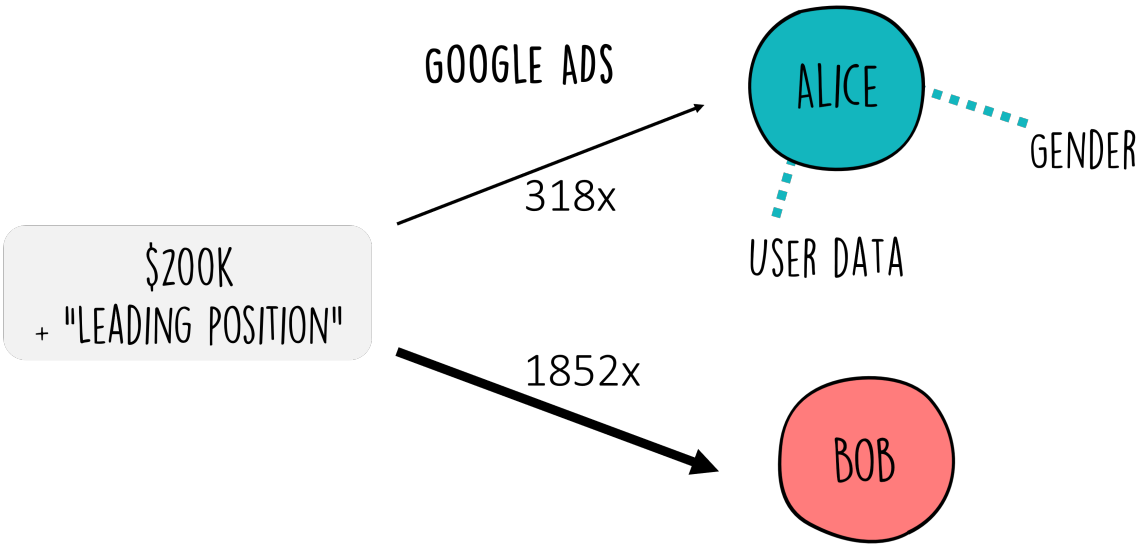


Fig. 3.4: Advertisements can have potentially large impacts… for instance if we talk about job advertisements.

A similar study examined Facebook's ad delivery process, revealing another layer of potential algorithmic discrimination [Ali *et al.*, 2019]. The research indicated that, even when advertisers aim for inclusivity in their targeting parameters, the optimization processes employed by Facebook could inadvertently skew ad delivery along gender and racial lines. This was particularly evident in advertisements for employment and housing opportunities. As such, these findings underline the importance of scrutinizing not just the algorithms behind data analytics but also the optimization mechanisms of digital platforms themselves, further emphasizing the role data scientists must play in promoting fairness and preventing discrimination.

Algorithms that preferentially select certain groups of people for job or housing ads, make it it fairly obvious that those applications have a strong the ethical dimension. But we can still go a few steps further to even more drastic examples. For this, let's move away from targeted advertisements. Data Science algorithms have also been used in the past to help with the actual selection of candidates for job vacancies, for instance Facebook [Reuters, 2018]. And it turned out that

due to several flaws and biases in the data (most developers are male), this algorithms had learned to also prefer male candidates over female candidates.

Algorithms are used for even more ethically critical applications. The software COMPAS for instance [Martin, 2019][Barocas and Selbst, 2016] is used to predict the risk for people that stand trial and hence influences if those people get released on bail or not.

Fig. 3.5: In several actual cases, algorithms were (and are) used for decisions with very severe personal consequences.

3.2 Data Science comes with ethical considerations

It is crucial to understand that, from an ethical standpoint, **data is not neutral** [D'Ignazio and Klein, 2020], and likewise, **algorithms are not neutral** [Martin, 2019][Stinson, 2022][Fazelpour and Danks, 2021]. As (future) data scientists, we must exercise the utmost care when working with data and algorithms. The examples provided should illustrate the importance of critically reflecting on the intentions behind our actions as data scientists. This reflection should encompass various perspectives to comprehend the "bigger picture". For instance, Shoshana Zuboff's "The Age of Surveillance Capitalism" [Zuboff, 2019] delves into how corporations like Google leverage users' personal data, or "behavioral surplus", to customize advertisements, essentially transforming users into products. Zuboff criticizes this practice, asserting that it undermines individuals' autonomy and capacity to influence their future. She proposes regulations and corporate accountability to ensure that technology benefits users rather than merely serving large corporations. Hence, the fact that we, as data scientists, *can* potentially extract maximum information or yield the "best" predictions from data does not mean we *should* do so.

A very important aspect in this context is the question of accountability, so who is in the end responsible for decisions taken by (or based on) an algorithm, see for instance [Martin, 2019]. Most data scientists are no trained ethicists and lawyers, and they also don't have to be. But I hope that the ethical aspects we briefly touch upon in this chapter make clear, that our role as data scientist is not finished when we have working code and a pretty looking plot in the end. It is not finished if we trained a machine-learning model that predicts with 99.9% accuracy (what does that mean anyway…). It is not finished when we successfully published our results or got a clap on our shoulder from the upper management. Our core job **includes** the ethical component, we have the duty to reflect on the potential consequences of our work. And this means, that we should not only learn how to apply the so many different data science methods, but we need to also learn enough about those method to judge why we use a certain method and what common pitfalls are that we need to test and take care of. Luckily, making this extra efforts will also technically make us better data scientists.

How do you feel now? Sounds like a heavy burden? Sure. But remember the wisdom of Spider-Man:

> **With great power comes great responsibility!**

CHAPTER

FOUR

HOW TO USE THIS BOOK (... IF YOU ASK US)

This textbook has been crafted with attention to detail to facilitate your journey into data science. However, the essence of its usefulness lies in how you engage with its content. Here's some advice to optimize your learning experience:

- **Progress at Your Own Speed**: Data science is a vast field, and everyone has their own pace. If you find certain concepts daunting, pause and seek additional resources. The internet is full of tutorials, videos, courses, and forums on practically all aspects that we cover in this book. The key is patience and persistence.
- **Learning by Doing**: Reading this book alone won't transform you into a data scientist. To truly learn data science, you need to actively engage in coding and working with real data sets. Observing code and nodding along saying, "yeah, I got it" is usually very away from having mastered it! We can't stress this enough - the most prevalent pitfall many students succumb to is passive learning! The true test of your understanding comes when you can independently apply the skills, starting from scratch, not merely copying, pasting, and executing someone else's code.
- **Lean on Tech, But Don't Rely on It**: While chatbots can serve as helpful data science assistants, they are not a substitute for your active learning. We firmly believe that the path to becoming a competent data scientist involves learning how to solve challenges on your own. Relying on a chatbot for answers is akin to copying solutions from another student or a forum. While this approach may help you complete your exercises in the short run, it won't contribute much to your long-term learning and skill development.

4.1 Skills you should have to work with this book

- **Python Programming Basics**: Understanding of variables, fundamental data types, loops, functions, and module imports are essential. If you're new to Python or need a refresher, consider resources like Python.org's Beginner's Guide, RealPython, or GeeksforGeeks Python Programming Language.
- At least **High School Math Skills**: Basic knowledge of statistics and algebra will serve you well, for instance to better understand the methods and their caveats.

4.2 Why Python?

Python, over the years, has emerged as the lingua franca of data science. But why?

1. **Versatile Libraries**: Python boasts an array of libraries such as NumPy, pandas, and scikit-learn, specifically tailored for data science tasks.
2. **Community Support**: The vast Python community ensures regular updates, myriad tutorials, and instant help on forums.
3. **Intuitive Syntax**: Python's clear and readable syntax makes it perfect for beginners.

4. **Flexibility**: Python seamlessly integrates with other languages, making it ideal for diverse tasks.
5. **Industry Adoption**: Major tech companies use Python, ensuring ample job opportunities.

While languages like R and tools like KNIME have their strengths, Python's holistic ecosystem and adaptability make it the preferred choice for many. All code examples in this book will be written in Python.

4.3 Open source libraries

Python's prowess in data science can be attributed to its incredible libraries. These libraries, backed by a robust open-source community, are the backbone of various data operations:

- **NumPy**: Fundamental package for numerical computations in Python. NumPy Documentation
- **pandas**: Offers data structures and operations for manipulating numerical tables and time series. pandas Documentation
- **SciPy**: Builds on NumPy and provides a large number of functions that operate on NumPy arrays and are useful for different types of scientific and engineering applications. SciPy Documentation
- **scikit-learn**: Simple and efficient tools for data mining and data analysis. scikit-learn Documentation

4.4 Setting Up Your Environment

4.4.1 Prerequisites

Before we begin, please ensure you have Anaconda or Miniconda installed on your machine. If you haven't already done so, you can download and install Anaconda from here. Anaconda is a free and open-source distribution of Python for scientific computing. It comes with conda, a package, dependency, and environment manager. Miniconda is a smaller, "minimal" version of Anaconda that includes only conda and Python.

4.4.2 Step 1: Clone the Repository

The first step is to clone the repository for this course. This can be done by using the `git clone` command:

```
git clone https://github.com/florian-huber/data_science_course.git
```

This command will create a new directory named `data_science_course` in your current directory, and download the entire repository into that directory.

4.4.3 Step 2: Navigate to the Repository

Next, navigate into the new `data_science_course` directory:

```
cd data_science_course
```

The `cd` command (which stands for "change directory") will move you into the directory that you specify after it.

4.4.4 Step 3: Create a New Environment

Once you're in the `data_science_course` directory, you can create a new conda environment for this course. This can be done using the `conda env create` command, along with the `-f` flag to specify the file defining the environment:

```
conda env create -f environment.yml
```

This command will create a new conda environment named `data_science_course`, based on the specifications in the `environment.yml` file. The environment will include Python 3.9, along with a number of other Python packages that are used in this course.

4.4.5 Step 4: Activate the New Environment

After the new environment has been created, you can activate it using the `conda activate` command:

```
conda activate data_science_course
```

Once the environment is activated, you can start working on the course materials. Remember, every time you want to work on the course, you should first activate the environment.

4.4.6 Step 5: Updating the Environment

In case of updates to the `environment.yml` file, you can update your conda environment using the `conda env update` command:

```
conda env update -f environment.yml --prune
```

The `--prune` option causes conda to remove any dependencies that are no longer required from the environment.

With these steps, you should have everything set up and ready to start diving into the material for this data science course. Happy data science coding!

Part II

Data Science Basics

CHAPTER

FIVE

DATA AND DATA TYPES

There are many different definitions for the term **data** [add refs]. Here's one of them, which will do for us to start with:

> Data ≈ values or findings obtained through observations, measurements, etc.

5.1 History of Data

A characteristic of data is that it must be measured or recorded. Hence, data is as old as the oldest human systems for recording data. One of the earliest remnants of such an endeavor is the Ishango bone, estimated to be around 20,000 years old [Pletser and Huylebrouck, 1999]. This artifact, displaying systematic notches grouped in blocks, resembles an ancient tally stick, hinting at our ancestral drive to count or record.

The very essence of data collection goes hand in hand with the birth of writing and symbolic representation. Surprisingly (to me at least), the oldest instances of these systems were not used to create elaborate letters, prose, or captivating stories. Instead, they were wielded for what may seem mundane today: collect and manage taxes and outstanding services. This fits to a saying that is linked to Benjamin Franklin: *"…in this world nothing can be said to be certain, except death and taxes"*.

While data has been collected and used for millennia, the term itself is relatively nascent. Originating from the Latin word *datum*, which means "given", the English adoption of *data* is believed to have been in the 1640s. The evolution of the term underscores our ever-growing understanding and reliance on the structured representation of knowledge.

5.2 Data Types

Before we dive deeper into how to acquire and process data, we first need to know some fundamental terms and their distinctions.

Structured vs. Unstructured Data

At the crux of data storage and management lie two primary categories: structured and unstructured data.

Structured data is organized into a defined schema or format. It's easy to search, manipulate, and analyze because of its systematic arrangement, often in rows and columns. Examples include relational databases and CSV files, but you can also just think of this as any data that you could meaningfully display in Microsoft Excel.

Conversely, *unstructured data* doesn't simply fit into a table. Or, we would say that this data has no specific form or model. As an important consequence, such data is typically harder to classify and analyze using traditional methods. Text documents, videos, and social media posts are common examples of unstructured data.

Feature vs. Data Points and the Dimensionality of Data

When we speak of data, especially in tables, we refer to *features* (or variables or attributes… this all depends on the domain or discipline people come from) and *data points*. Features are the distinct attributes or properties of the data set.

In tabular data, these often appear as columns. For instance, in a table cataloging books, features might include "Title", "Author", and "Publication Year".

On the other hand, *data points* are individual pieces of information, often represented as rows in tabular data. For example, each book listed in the aforementioned table would be a data point.

The concept of dimensionality arises from the number of features. A table with three features is 3-dimensional, while one with 15 features is 15-dimensional. Understanding dimensionality is crucial, especially in domains like machine learning, where high-dimensionality can lead to challenges such as the "curse of dimensionality".

Data vs. Metadata

While *data* represents the core information we aim to analyze or utilize, *metadata* is the information about this data. It describes the data's context, quality, condition, origin, and other characteristics. If data is a book, metadata is the blurb on the back, providing insights about its content, author, and publication details.

Categorical vs. Numerical Data

Data can often be pigeonholed into categorical or numerical types.

Categorical data is data that can be divided into specific categories but doesn't have any inherent order or hierarchy. For instance, the colors of a shirt (blue, red, green) are categorical.

Numerical data, as the name suggests, relates to numbers and can be further split into discrete (countable) and continuous (measurable) data. Age, height, and weight are all examples of numerical data.

Scale Types: An Overview

Delving deeper into data characterization, we encounter different scales that data can adhere to:

- *Nominal scale*: This is the most basic level, used for labeling variables without any quantitative value. It's the realm of categorical data, like gender or ethnicity.
- *Ordinal scale*: It denotes categories with a set order or rank. Though there's a hierarchy, the intervals between the ranks aren't uniformly defined. Examples include rating scales of "low", "medium", and "high".
- *Interval scale*: This is numerical data where the intervals between numbers are consistently uniform, but there isn't a true zero. Temperature, for example, is on an interval scale as 0°C doesn't signify a lack of temperature.
- *Ratio scale*: This scale boasts all the properties of the interval scale, but with a clear definition of zero. Length, weight, and age are examples of data on a ratio scale.

Scale	Level	Possible Operations	Typical Data Type	Example
Categorical	Nominal	Frequencies	string	[red, yellow, blue]
Categorical	Ordinal	Median, Quantiles	string	[good, average, bad]
Numerical	Interval	Means, Variances, Addition, Subtraction	int, float	Temperature in Celsius (5°C is **not** five times as warm as 1°C)
Numerical	Ratio	Means, Variances, Addition, Subtraction, Multiplication, Division	int, float	Lengths, Sizes, Weights, Monetary amounts …

5.3 Big Data

Working in data science there is really no way to avoid dealing with the challenges, the promises, or even the (many) definitions of **Big Data**. Since this is not our core concern in this book, I will simply stick to the very simple definition of saying

> **Big Data** ≈ Data that is too large, too complex, or too volatile to be evaluated using manual and traditional data processing methods.

Still, what does this mean? And why is there no sharp definition of which data is "big data" and which is not? Well, in essence, there is no sharp boundary between big and "not big" data. Big makes us first think of the sheer size of the data, or **volume**, say number of Giga-/Terra-/Peta-bytes. This is, however, far to simple. Astrophysicists collecting super-high-resolution telescope pictures of the sky would probably never consider a dataset that fits on a USB-stick to be *big* (high resolution images take a lot of disk space!). But librarians or historians might have a very different view on what is big and what is not. To get a better intuition for such volumes: 6.5 million English Wikipedia articles require only 20GB but equate to about 3000 encyclopedia volumes. This easily makes this *big data*. And yet, ten high-resolution movies require the same storage but don't form a *big* dataset.

Beyond the volume-related discussions, there are other factors that also contribute to whether or not we consider data as *big data*, which here means: things that further complicate the handling of the data. This can, for instance, be the **variety** of data, but also the **velocity** by which the data needs to be processed or analyzed. Together those terms are called the **"3V's"** (Volume, Variety, Velocity) that contribute to data being considered *big data*. Over the years, people started to add to this list, so that we now also have 4V's or 5V's … but I will leave this for you to research if you want to know more about these definitions.

If you still feel like you have no idea what big data actually means, feel free to just go ahead to the next chapters with a basic first intuition of:

> Anything that can be done with basic Excel methods **is not Big Data**.

CHAPTER

SIX

DATA - INFORMATION - KNOWLEDGE

So far in our exploration of data science, we've been mostly circling around *data*. However, as briefly touched upon in {doc}``./book/01_intro_data_science.md`` and contrary to what the term *data science* might suggest, data science isn't fundamentally about data. Think of it this way: if data is the raw material, then the ultimate product a data scientist crafts is *knowledge*. We sift through the vast sands of data to uncover gems of insight.

The "DIKW Pyramid" (Data - Information - Knowledge - Wisdom) beautifully visualizes the hierarchy that exists from raw data to profound knowledge (Fig. 6.1). While numerous interpretations of each tier exist, for our current exploration, let's break it down as:

- **Data**: The raw values or findings obtained through various means such as observations, measurements, surveys, etc. It represents the unprocessed reality, and -at least at this stage- it can be entirely incomprehensible or meaningless. Imagine a streaming camera in a forest meant to detect rare events of spotting certain wild animals. Most of the data recorded will show no such animals and will thus be entirely insignificant for our later analysis.
- **Information**: This is data in context. It has been processed, categorized, and given structure, making it more comprehensible. One way to put it is: "Information is data with potential significance for a user" [] . It's data with added value, transformed into a format that allows for a clearer interpretation.
- **Knowledge**: This goes beyond just having information; it's understanding the information. Knowledge encompasses facts, theories, and rules that have been distilled from the information and are characterized by a high degree of certainty and applicability.

6.1 From data to knowledge

Now, think of data science as detective work. A detective starts with clues (our data). These clues, in isolation, may seem random or unimportant. But as the detective starts piecing them together, a story unfolds (information). Finally, using their understanding of the world, the detective makes an informed guess about what actually happened (knowledge).

This hopefully gives you a bit of intuition of what we are after as data scientists. But it should also be noted, that there is no simple recipe of how to get from data to knowledge. In fact, this is what many scientists and philosophers were, and are, after: understanding how we can derive knowledge from data. There is no simple answer to this, and comparing many of the possible answers in details is far beyond the scope of this book. For the interested reader I would still like to briefly introduce a few concepts.

A classical, and often modified and extended, theory of how scientific knowledge progress works was proposed by Karl Popper [Popper, 2013]. He suggested that scientists propose hypotheses based on initial observations (data). These hypotheses are then rigorously tested, and if proven wrong, are modified or replaced by newer, more refined hypotheses. This iterative cycle continues, constantly refining our body of knowledge.

Another insightful approach is the "Inference to the Best Explanation" method [Bartelborth, 2017]. Here's how it works:

1. **Observation**: Start by observing phenomena. This is your raw data. (in the detectives picture: gather clues)
2. **Hypothesis Formulation**: Develop potential hypotheses that might explain the observed data.

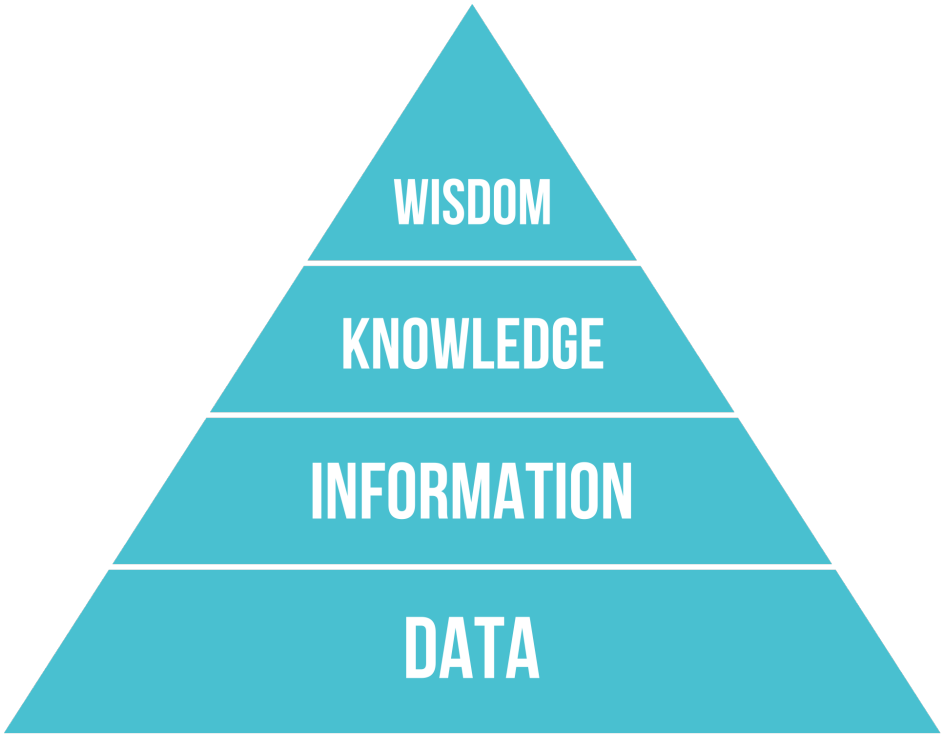


Fig. 6.1: The DIKW Pyramid. Source: wikipedia

3. **Refinement**: Some theories might not fit the facts. So, they're discarded.
4. **Comparison**: Which of our remaining theories explains the data best?
5. **Selection**: Choose the hypothesis (or hypotheses) that offers the most plausible and comprehensive explanation in line with other established knowledge.

While both Popper's approach and the inference to the best explanation emphasize rigorous testing and refinement, they underscore the essence of the scientific method – a systematic pursuit of knowledge. In the realm of data science, these principles guide us, ensuring that insights drawn from data are robust, valid, and, ultimately, enlightening.

In conclusion, data is the foundation upon which knowledge is constructed. But to elevate data to knowledge, a meticulous process of refinement, analysis, and understanding is imperative. There are many possible pitfalls (for instance: biases!). As data scientists, our mission is to navigate this intricate journey, transforming raw data into profound insights that inform, guide, and enlighten.

CHAPTER

SEVEN

DATA SCIENCE WORKFLOW

Data science can often feel like piecing together a jigsaw puzzle. But like any puzzle, there's a method to the madness, a sequence that can make the process smoother. That's where the idea of a data science workflow comes in. In our previous chapter, we used the analogies of detective work and the scientific method to explain data science. Similarly, every detective has a method to their investigation, and every scientist follows a systematic process in their research.

In data science, it's paramount to have a blueprint, a sequence of steps, that can guide the project from inception to completion. This blueprint is called a **data science workflow**. It serves as a roadmap that provides clarity on the stages involved in a data science project. Just as a detective might start with gathering evidence, then move to interviewing witnesses, and finally piece together the entire story, a data scientist moves through distinct phases to derive insights from data.

While the last chapter hinted that there's no universally accepted data science workflow, several models and frameworks have been proposed over the years [Weihs and Ickstadt, 2018][Donoho, 2017]. Each of these frameworks is tailored to different needs and contexts, and their structure and emphasis can vary significantly. For instance:

- Circular workflows, most commonly **CRISP-DM** [Wirth and Hipp, 2000][Weihs and Ickstadt, 2018] usually emphasize deployment. They often stem from an industry or software development background, where iterative development and frequent updates are the norms. These workflows recognize that the end of one project cycle can be the beginning of another, especially as new data comes in or as models need refining.
- Linear workflows, like **OSEMN** (Obtain, Scrub, Explore, Model, and iNterpret), tend to have a start and end. However, it's essential to understand that even in these linear models, iterations within steps are common [Mason and Wiggins, 2010].
- There are also **tree-like workflows**, where after a certain step, the process might branch out into multiple paths, reflecting the multifaceted nature of some data science projects [Guo, 2022].

In this book, we'll be introducing our tailored workflow called **ASEMIC** (Fig. 7.1). Inspired by OSEMN, this framework is designed to capture the essence of a typical data science project while allowing for flexibility. Important to note is that the data science typically is linked to overarching questions or tasks. Those will largely determine the many decisions we have to take in every phase of our workflow. The ASEMIC model comprises:

- **A**cquire data: This is where we obtain our raw data, the foundation of our entire project.
- **S**crub data: Data is rarely perfect. In this phase, we clean our data by handling missing values, outliers, and errors.
- **E**xplore data: Through visualization and statistical analysis, we delve deep into our data to understand its nuances.
- **M**odel data: Based on our understanding, we choose suitable algorithms and techniques to model our data.
- **I**nterpret results: Once we have our model results, we decipher what they mean in the context of our problem.
- **C**ommunicate results: Data science is as much about communication as computation. Here, we present our findings in a clear, compelling manner.

It's crucial to note that real-world data science is rarely a linear progression from one step to the next. It's an iterative process, filled with cycles (Fig. 7.2). After modeling, you might find that you need to go back to the data scrubbing

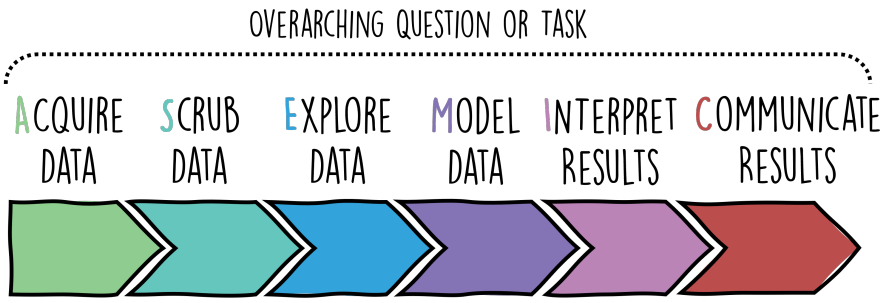


Fig. 7.1: Simplified sketch of an ASEMIC data science workflow.

phase, or after interpreting results, you might need to revisit the exploration phase. This non-linear, cyclical nature of data science is what makes it both challenging and fascinating.

Drawing parallels with the detective analogy: imagine a detective, after gathering all evidence (Acquire), ensures they're all pertinent and discards the irrelevant ones (Scrub). They then analyze the evidence, looking for patterns or connections (Explore), formulate a theory about the crime (Model), assess how well this theory fits with the facts (Interpret), and finally present their findings to their team or in court (Communicate). Just as a detective might revisit a crime scene or re-interview a witness based on new findings, a data scientist might loop back to earlier steps based on new insights or challenges.

In practice, each data science workflow is different an having a sketch (such as ASEMIC) in mind is mostly meant for the general orientation across the many different phases of such a project. But what can already be seen from such descriptions is that a data science process consists of a wide variety of different phases or steps, which ultimately also leads to the wide variety of techniques and tools that need to be used throughout such a process. The present book will aim to give a broad, general overview, but obviously cannot go into full depth for most of the introduced topics. Throughout the following sections you will hopefully gain a better understanding of what the different phases comprise, and what skills and expertise those require.

Speaking of expertise, let's rephrase something from Section 2.2 . It is very common in data science projects that they **cannot be carried out by one person alone!** If you were hoping to work all day on your own in front of a computer… better reconsider the idea of becoming a data scientist. In most cases you will need to team up with people that bring in other skills and expertise, and communicate the targets and results with people from very different backgrounds. In Fig. 7.3 I tried to illustrate that this often leads to very dynamic processes with strongly varying requirements for different phases or time points.

In the following sections, we'll use the ASEMIC workflow to navigate through a large variety of data science techniques from data handling to analysis, interpretation, and visualization.

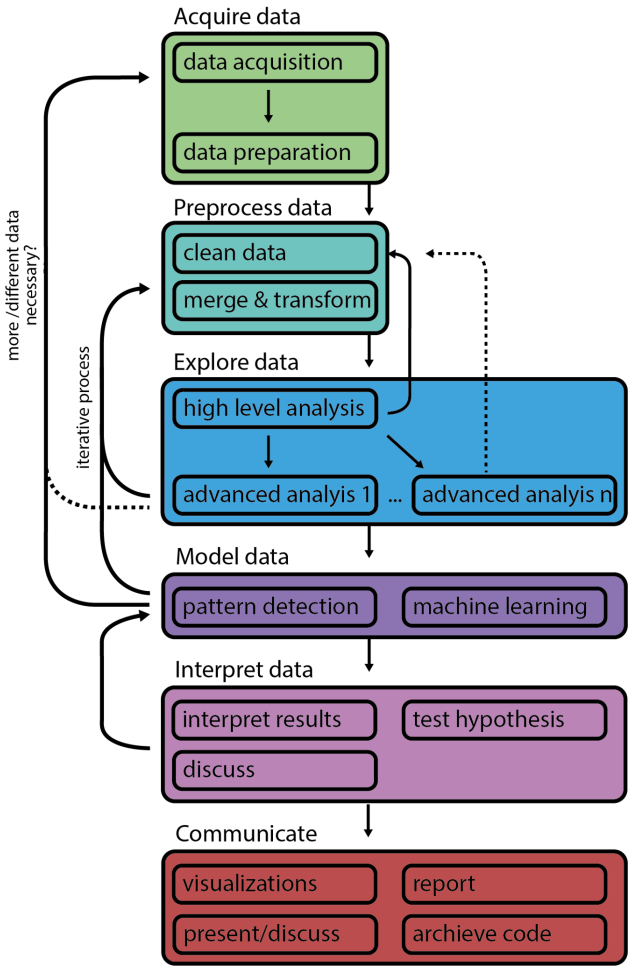


Fig. 7.2: More detailed sketch of the ASEMIC data science workflow, highlighting the interconnected nature of the different steps or phases.

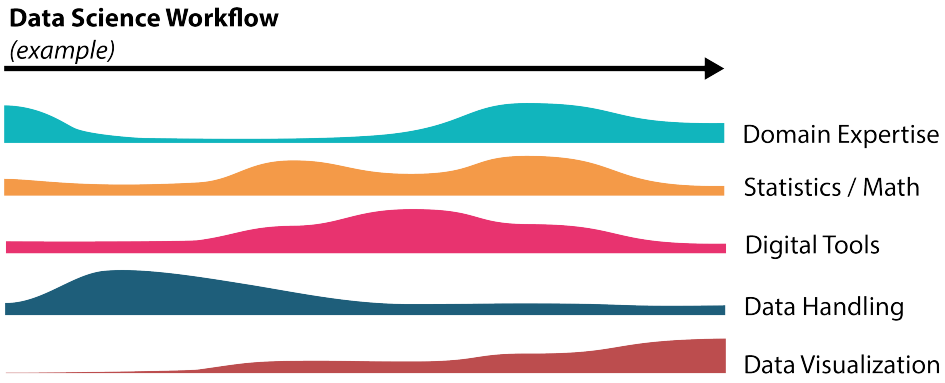


Fig. 7.3: Sketch of an example data science process to illustrate that different skills and expertise is required for different phases of a typical data science project.

Part III

Data acquisition and first exploration

CHAPTER

EIGHT

DATA ACQUISITION & PREPARATION

The foundational pillar of any data science project lies in the data itself. Sounds obvious, yet in practice many aspiring data scientists will still be surprised by **how much the quality and quantity of data matters for the success of a data science project**. Without pertinent and high-quality data, even the most sophisticated algorithms will fall short of producing meaningful results. This chapter aims at building a first intuition about the process of data acquisition and preparation, elucidating best practices to ensure your data science journey is built on solid ground.

8.1 Data Acquisition

Before plunging into data collection, it's imperative to ask: What's the problem at hand? What kind of data do we need to address it?

8.1.1 Key Considerations:

1. **Type of Data Needed**: What kind of information will help address the problem? Is it quantitative, qualitative, temporal, or spatial?
2. **Data Quality and Quantity**: Do the data offer enough breadth and depth to draw meaningful conclusions? Are there gaps or inconsistencies that might pose challenges?
3. **Data Sources**: Is the data readily available? If not, where can one procure it?
4. **Usage Restrictions**: Are there any commercial or privacy restrictions tied to the data?

8.1.2 Common Data Sources:

- **The Internet I**: A vast ocean of data. Public datasets are easily accessible and can be sourced from governmental agencies, NGOs, or platforms like Kaggle, Zenodo, and UCI.
- **The Internet II**: Web scraping is akin to a treasure hunt, extracting valuable data directly from web pages, given that you respect the legal and ethical boundaries.
- **Internal Corporate Data**: Picture a gold mine that a company sits upon; these are years' worth of data that can unearth invaluable insights if analyzed correctly.
- **Academic Data**: Scientifically collected and often meticulously maintained, these datasets can be found accompanying research papers.
- **Data Upon Request**: Sometimes, just asking can open doors. Organizations might share data if approached correctly and for a worthy cause.

- **Commercial Datasets**: There are instances when investing in a dataset can prove to be more cost-effective than collecting data from scratch.

In some data science projects we will start with an already prepared dataset, while in other projects the collection of data can be major (and time consuming!) part of the project itself. It can even happen that we need to reconsider some of our initial targets and strategies if we do not manage to collect the data as expected.

Once we do hold the data in our hands (well, rather on a disk or cloud drive), we further need to make sure that the data is technically, legally, or quality-wise suitable for what we plan to do with the data in the following phases.

Once the data is in your hands, several pivotal steps ensure its quality and relevance:

- **Data Format**: Ensuring that the data is in a readable and workable format is paramount. Do you need to develop a custom parser, or can off-the-shelf software be used for ingestion?
- **Source Credibility**: If you have ever tried to collect data yourself, you will know that not all data is reliable. The credibility of the data source can significantly impact the reliability of your findings.
- **Data Processing Knowledge**: A thorough understanding of how the data has been previously processed can save you from potential pitfalls. Valuable insights can often be gleaned from data processing logs or accompanying documentation.
- **Legal issues:** Having data is not the same as *being allowed to use it*. Things like copyrights or privacy issues might render the data unfit for our goals.

8.1.3 Remember: Crap in, Crap out!

Data is the bedrock of all subsequent data science operations. The notion that advanced algorithms can compensate for shoddy data is a common misconception. It's pivotal to invest ample time in sourcing and assessing data. The quality of data often becomes the deciding factor between the success and failure of a data science project.

8.1.4 Mini-Exercise: Hypothetical Scenario

Imagine you are tasked with evaluating the state of digitization in schools in your region.

Objective: Assess the current level of digital infrastructure and its utilization in schools.

Prompt: Spend a few minutes brainstorming:

1. What kind of data might you need?
2. Where could you potentially source this data?
3. Are there potential roadblocks or challenges in procuring or using this data?

8.2 Data Cleaning

Raw data, more often than not, is messy; it requires refinement to reveal its true value. This refinement process is what we term as "data cleaning". In this section, some of the most common data cleaning steps will be mentioned; steps which are fairly general across many different data types and sources. In actual practice, however, a lot of the more complex data cleaning requires a certain understanding - or *domain knowledge* - of the data. This is either acquired over time, when you work a lot with similar data and tasks; or it is distilled from a close collaboration with domain experts. Whenever you are not sure what the data in front of you actually means, and you hence are not sure what you may or may not do to the data, better invest time to build a deeper understanding for the data and/or ask someone with more experience on this type of data.

Here a few examples. If you work with audio data, but don't know that decibel is measured on a logarithmic scale, you might unintentionally run into serious trouble. Or, if you get colorful medical images but don't realize that the color scale is just a color scale applied to monochromatic images, your data handling and later conclusions might not make it look like you know what you are doing…

8.2.1 Dealing with Missing Values:

Missing data is a common issue many data scientists face. While the gaps can manifest in varied forms such as NaN, "999", "N/A", None, and "", a standardized approach often involves representing these uniformly as `NaN` (hint: Python libraries such as `Pandas` or `numpy` have there own NaN data types!).

8.2.2 Typical Challenges:

- **Duplicates**: Data can have repeated or conflicting entries.
- **Inconsistent Nomenclature**: Consistency in naming conventions can save hours of data wrangling later on. A classic example being different formats of names. "First Name Last Name" versus "Last Name, First Name".
- **Data Types**: Ensuring that numeric values aren't masquerading as strings can prevent potential analytical blunders (e.g., "12.5" instead of 12.5).
- **Decimal Delimiters**: Confusion between comma and dot can change data meaning, e.g., 12,010 becoming 12.01.

8.2.3 Further Cleaning Steps:

- **Unit Conversion**: Ensuring data is in consistent units.
- **Data Standardization**: This can be done via Min-Max scaling (often termed "normalization") or, frequently more effective, by ensuring data has a mean of 0 and a standard deviation of 1.
- **Non-linear Transformations**: Sometimes, linear thinking won't do. Transformations like logarithms can provide new perspectives on data.

In essence, data acquisition and preparation are the unsung heroes of a successful data science endeavor. By ensuring the foundation is robust and the raw materials are of top quality, you set the stage for analytical brilliance.

From my own experience: It is not uncommon that 80% of the work in a data science project go into the data acquisition and cleaning.

CHAPTER

NINE

FIRST DATA EXPLORATION

Different from what a linear sketch of a data science workflow might sometimes suggest, it is generally not at a single moment that we inspect our data. Instead, we inspect our data frequently, with varying focus and depths. In the previous sections we already saw that the collection and technical handling (e.g., importing) of the data already includes a certain degree of data inspections. Then, during data cleaning we had to have a closer look at that is in the data and what might be missing. Now, we want to take a next step, which often done by doing basic statistics and other quick data explorations. A good data exploration phase is like getting a quick snapshot of the data's characteristics, anomalies, patterns, and relationships. This foundational step often happens immediately after or even during the data acquisition and cleaning phases.

9.1 Statistical Measures

A comprehensive understanding of the data often starts with basic statistical measures. These measures provide concise summaries, allowing data scientists to grasp the main tendencies and dispersions in the dataset. Many of you will probably (or hopefully?) know many of the measures that are listed in this chapter and the core focus here will be to give a quick refresher. For those of you that feel lost when it comes to basic statistics it would make sense to invest time to gain a more solid foundation into those basics. There are many books, tutorials and courses on this matter, for instance also the w3schools website.

1. **Measures of Central Tendency**: These are quintessential in understanding where the 'center' of our data lies.
    - **Mean (Arithmetic Mean)**: The arithmetic mean is calculated as the sum of all values divided by the number of values. It's a common measure to determine the central value in a dataset. Mathematically, if X is a dataset with n values, the mean μ is calculated as:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$$

    - **Median**: The median is the middle value of an ordered dataset, separating the higher half from the lower half. If the dataset has an odd number of observations, the median is the middle number. If there's an even number, it's the average of the two middle numbers.
    - **Mode**: Represents the value that appears most frequently in a dataset. Datasets can have one mode, more than one mode, or no mode at all.
2. **Quantiles**: They segment the data into intervals which encompass equal probabilities.
    - **Median**: Also known as the 0.5-Quantile, representing the 50th percentile.
    - **Deciles**: Segments the data into 10 equal parts, marking every 10th percentile.
    - **Percentiles**: Denotes specific positions in a data set, which is divided into 100 equal parts.

3. **Measures of Spread**: While central tendency gives an overview of the data's central point, measures of spread describe how much the data tends to deviate from that point.

    - **Standard Deviation (STD)**: It quantifies the amount of variation in the dataset. A low standard deviation means that values are close to the mean, while a high standard deviation indicates that values are spread out over a wider range.

$$\sigma = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

    - **Interquartile Range**: Represents the range between the first (25th percentile) and third quartiles (75th percentile).
    - **Range**: It's the simplest measure of spread and is calculated as the difference between the maximum and minimum values in the dataset.
    - **Variance**: The average of the squared differences from the mean, often denoted as σ2σ2.

9.1.1 Exercise: Understanding Statistics in a Practical Scenario

Consider the 3rd semester of Media Informatics: The average age (mean) of the students = 22.7 years Standard Deviation (STD) = 2.1 years

Questions:

1. Given the mean and standard deviation, is it probable for there to be students in the 3rd semester of Media Informatics who are older than 40 years?
2. Based on the mean and standard deviation, what would be the estimated minimum and maximum age of the students?
3. Based on the mean, are most of the students above or below 22.7 years of age?

Below is a quick summary of the measures discussed:

Measures	Description	Nominal	Ordinal	Quantitative	Accounts for All Values	Sensitive to Outliers
Mode	Most frequent value	x	x	x		
Median	Central value with equal numbers of data points above and below		x	x		
Mean (Arithmetic Mean)	Average value			x	x	x
Geometric Mean	Average of growth rates; multiplicatively linked			x	x	x
Harmonic Mean	Average of fractions with a constant denominator or a special case of weighted arithmetic mean			x	x	x
Quantile / Quartile	Value below which a specified percentage of observations fall			x		
Minimum / Maximum	Smallest and largest values respectively		x	x		x

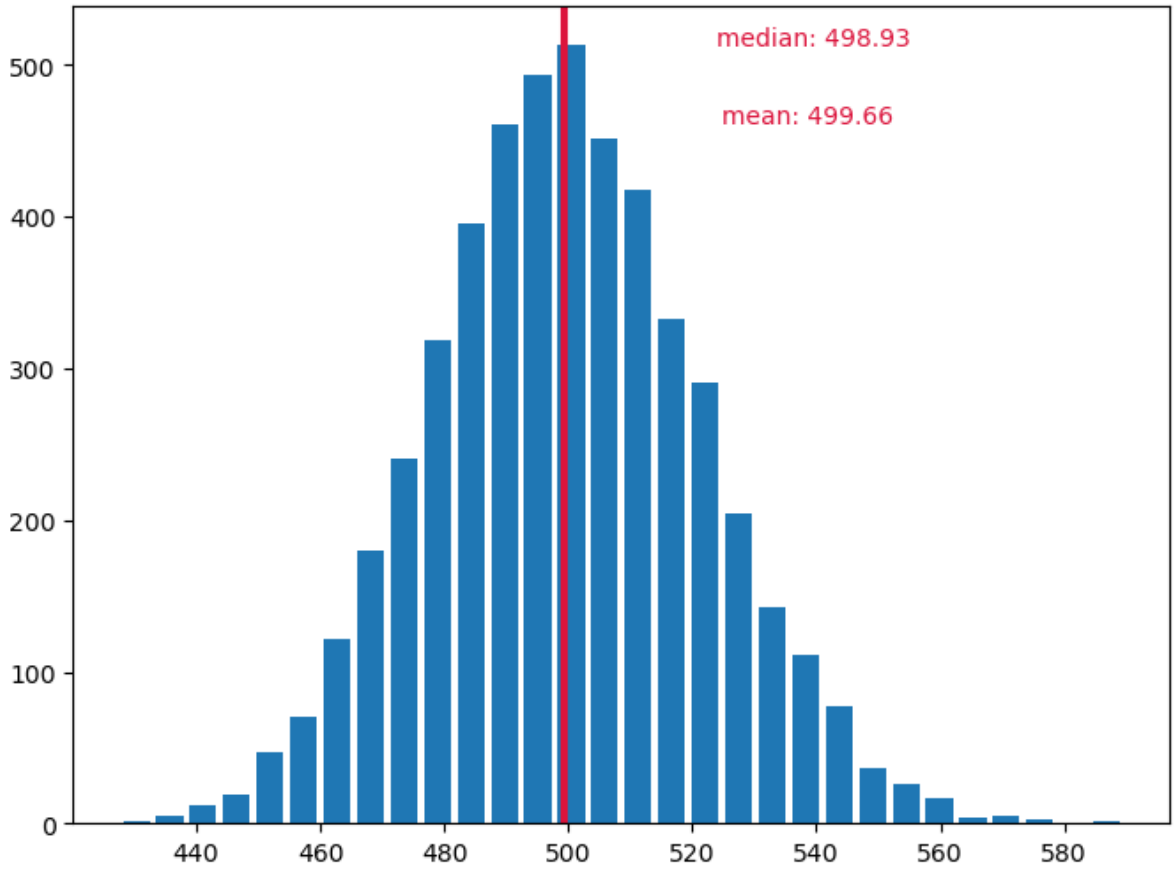
9.2 Statistical Measures and Distributions

Let us look at some distributions and their statistical measures. In the following Python code cells we will import the required libraries and define a simple plotting function for the task.

9.2.1 Symmetric Distribution

Symmetric distributions are those where values are distributed in a way that the shape on one side of the centerline mirrors the shape on the other. In other words, the left half of the distribution is a mirror image of the right half. One of the key properties of a symmetric distribution is that the mean and median will be the same, or very close.

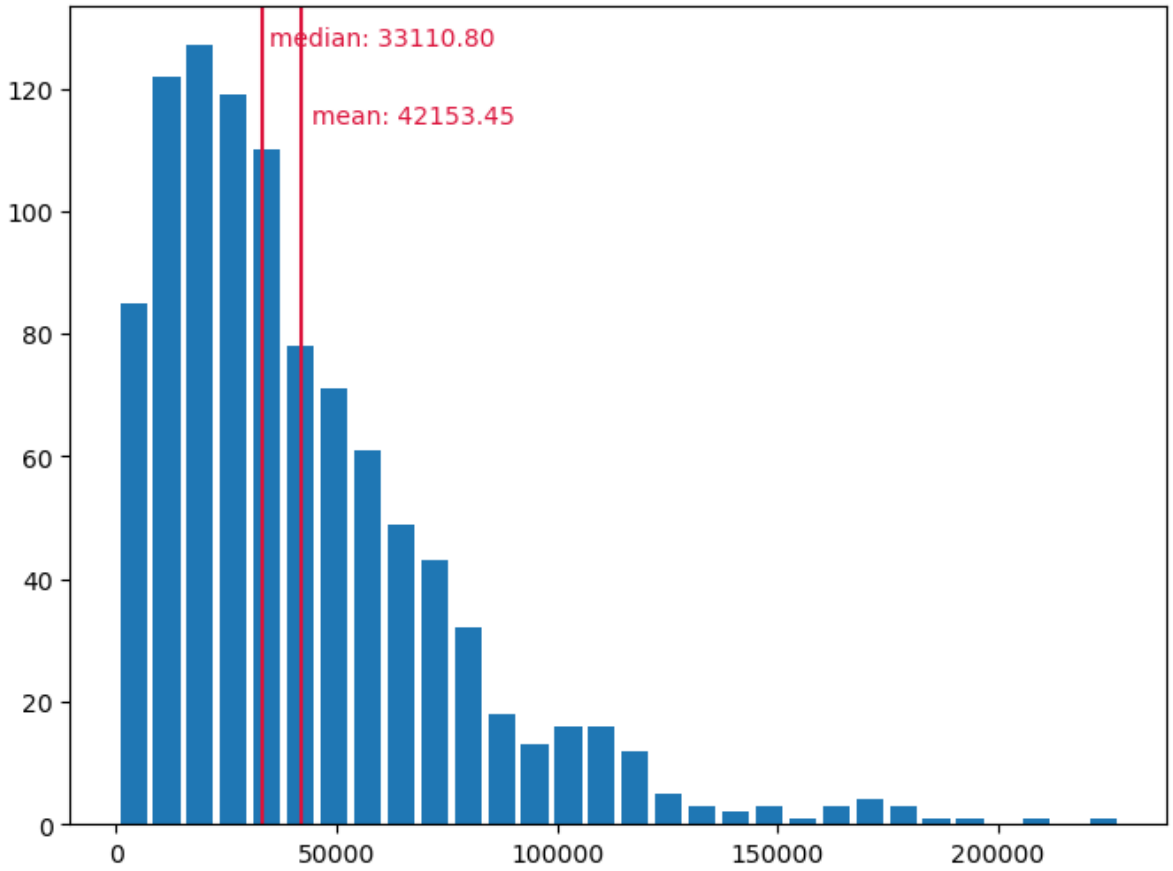
```
np.random.seed(0)
data = np.random.gamma(500, 1, 5000)
plot_dist(data, 30, ["mean", "median"])
```



9.2.2 Non-symmetric Distributions

In reality, most of the data we encounter tends to be non-symmetric. These distributions don't exhibit mirror-like symmetry around their center. It is important to note here, especially for non-symmetric distributions, the mean might not necessarily represent the "center" of the data.

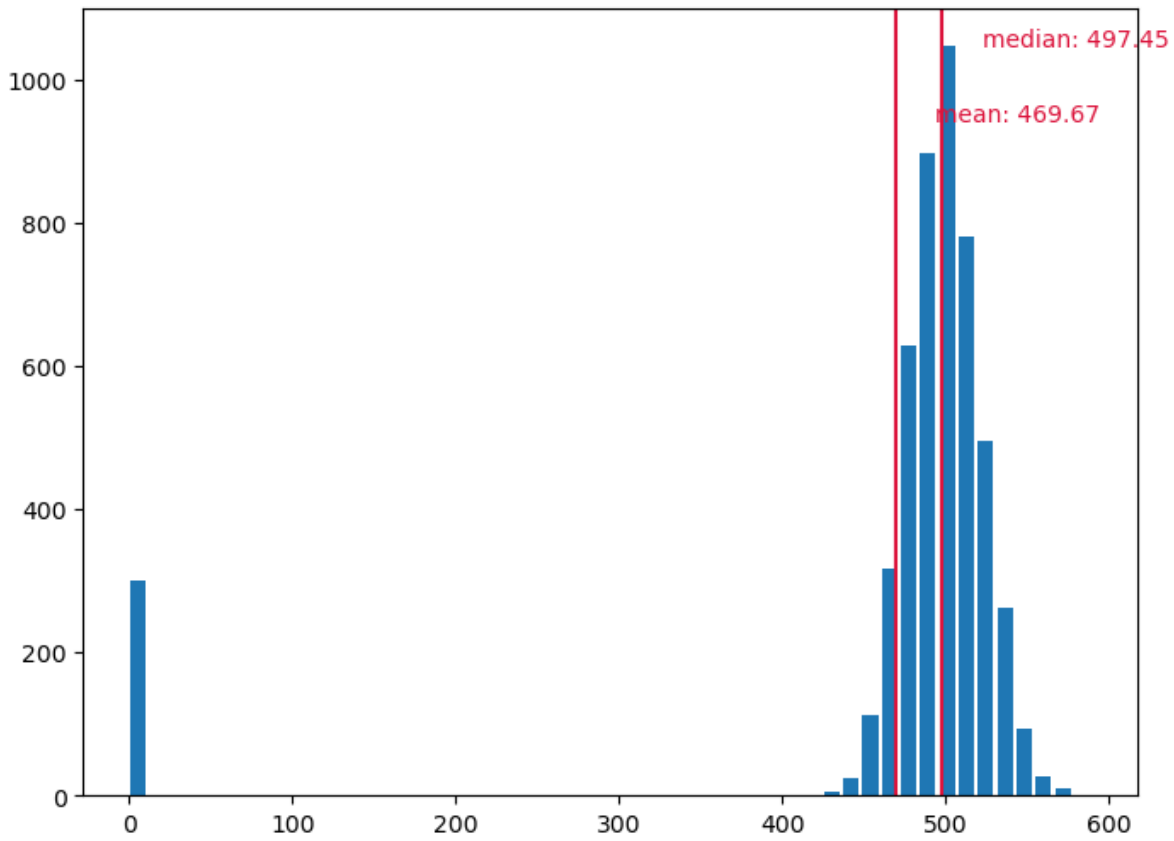
```
data = np.random.gamma(1.6, 26000, 1000)
plot_dist(data, 30, ["mean", "median"])
```



9.2.3 Distributions with Outliers

Outliers are values that stand apart from the bulk of the data. Their presence can distort our perceptions about the data and can notably skew our mean. It's essential to identify and manage outliers for better statistical interpretations.

```
np.random.seed(0)
data = np.random.gamma(500, 1, 5000)
data[:300] = 0
plot_dist(data, 50, ["mean", "median"])
```

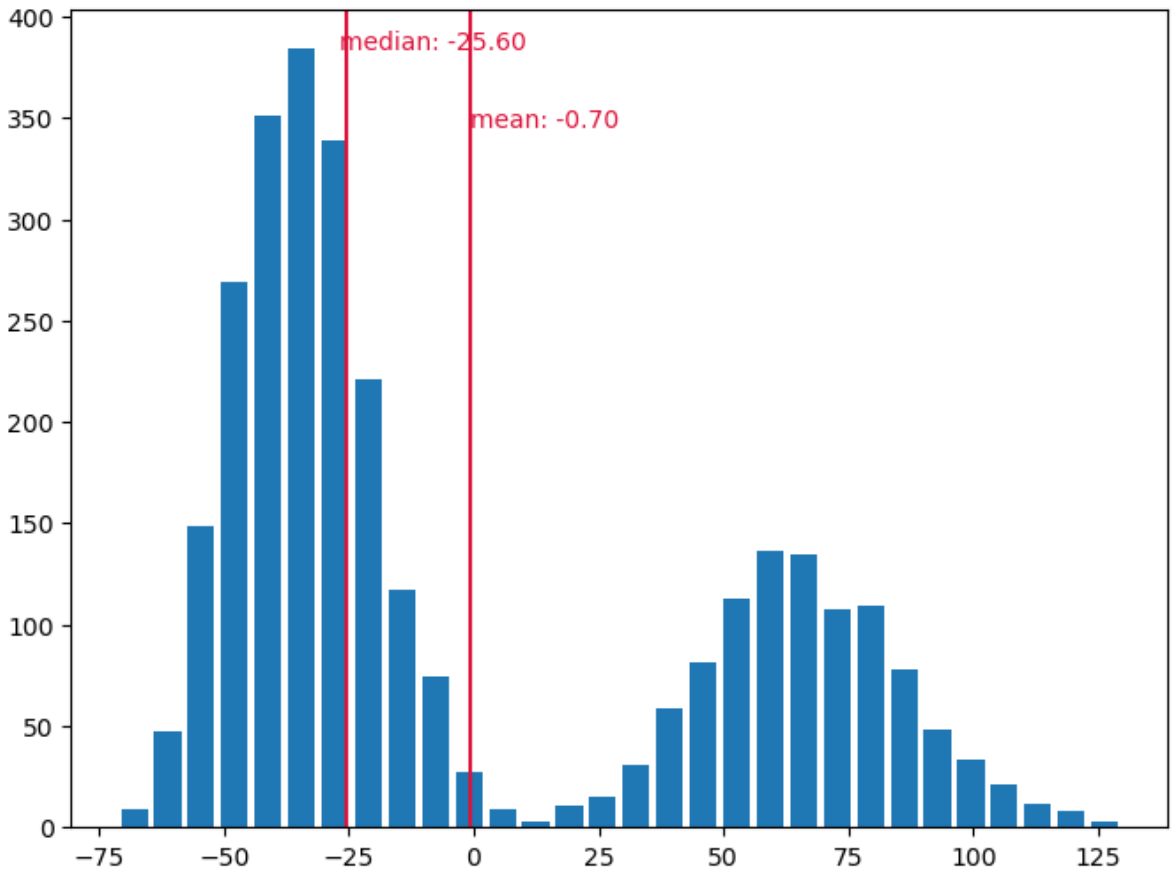


If you look at the distribution plot with the median and the mean: Which of the two measures is better in this case (and why)?

9.2.4 Mixed Distributions

At times, our dataset may not belong to a single type of distribution. Instead, it may be the result of a mix of two or more underlying distributions. This phenomenon is observed in mixed distributions. Recognizing and understanding the different underlying distributions can be crucial for analysis.

```
np.random.seed(0)
data1 = np.random.gamma(50, 2, 2000)
data2 = np.random.gamma(100, 2, 1000)
data = np.concatenate((data1, data2)) - 134

plot_dist(data, 30, ["mean", "median"])
```



Here again: If you look at the distribution plot with the median and the mean, which of the two measures is better in this case (and why)?

9.2.5 Statistical Measures

After observing the distributions, let's also calculate key statistical measures and see what insights they provide regarding the distribution's characteristics.

```
print(np.std(data))
print(np.min(data), np.max(data))
print(np.quantile(data, 0.5))
print(np.quantile(data, 0.9))
print(np.percentile(data, [25 ,75]))
```

```
50.09713754299179
-71.30263005114779 129.6311878602093
-25.604183317689994
76.81243703053997
[-39.37861049  52.79716049]
```

9.3 Statistical dispersion

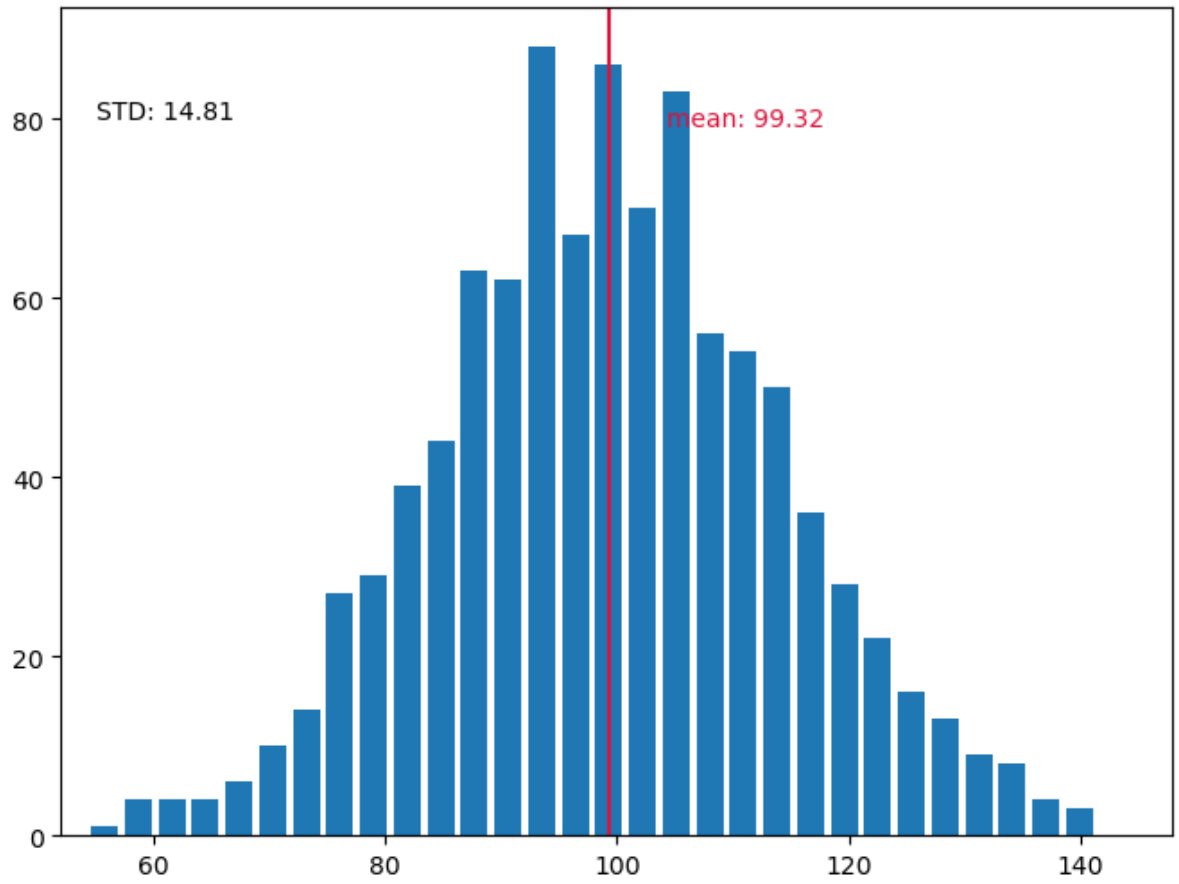
Statistical dispersion refers to the spread or variability of a dataset. It helps to understand the extent to which individual data points deviate from the mean. While two distributions might have similar centers (mean or median), their characteristics could differ considerably in terms of spread. For instance, one might be tightly clustered around the mean, whereas the other might be more spread out.

Dispersion can be illustrated with two distributions that have the same center (mean) but different standard deviations. In the following examples, both distributions are centered at 100, but they have standard deviations of 15 and 1.5, respectively. The standard deviation (often referred to as "STD") is a measure that tells us how spread out the numbers in a distribution are.

A higher standard deviation indicates that the data points tend to be farther from the mean, while a smaller standard deviation suggests that they are clustered closely around the mean.

```
np.random.seed(0)
data = np.random.normal(100, 15, 1000)

plot_dist(data, 30, ["mean"])
plt.text(55, 80, f"STD: {np.std(data):.2f}")
plt.xlim(52, 148)
```

```
(52.0, 148.0)
```

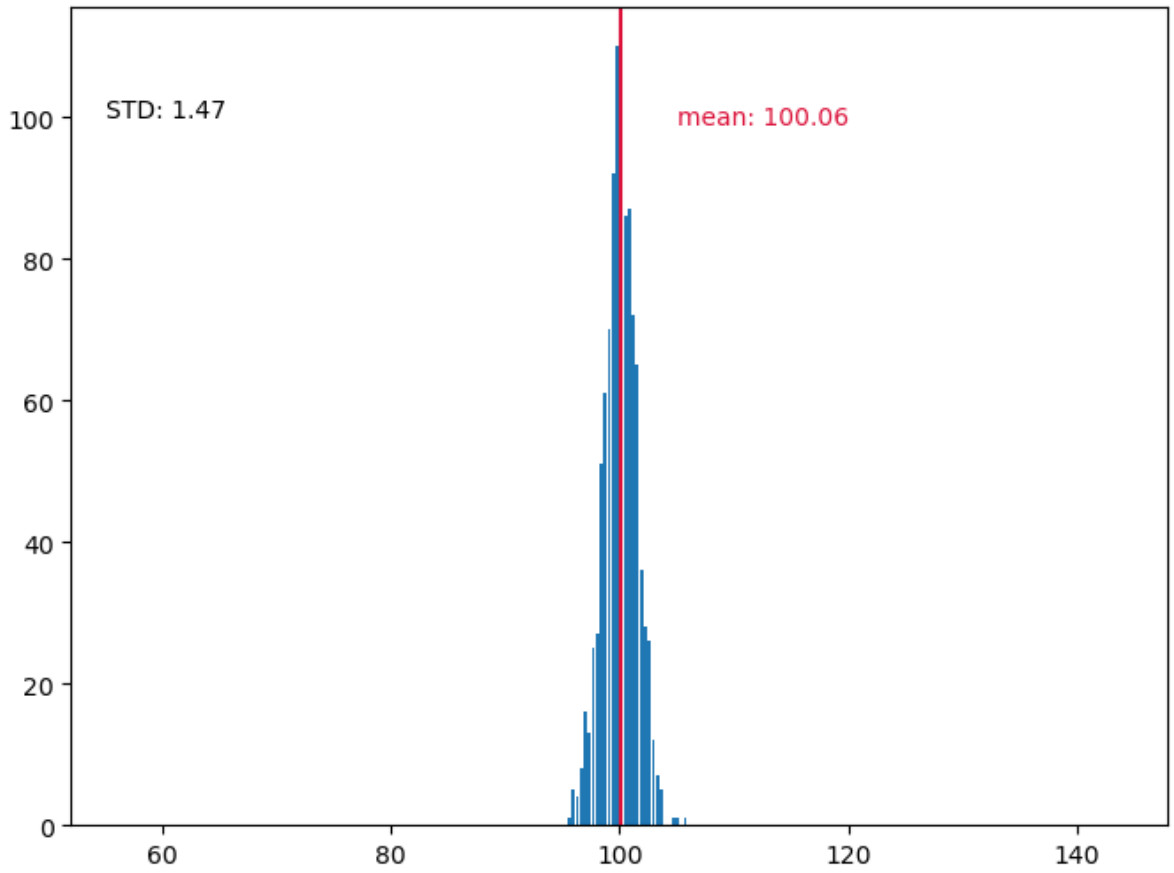


The plot above showcases a normal distribution with a mean of 100 and a standard deviation of 15. The data points are

spread relatively far from the mean, resulting in a wider bell shape.

```
np.random.seed(1)
data = np.random.normal(100, 1.5, 1000)

plot_dist(data, 30, ["mean"])
plt.text(55, 100, f"STD: {np.std(data):.2f}")
plt.xlim(52, 148)
```

```
(52.0, 148.0)
```



In contrast, the second plot demonstrates a distribution with the same mean of 100 but with a much smaller standard deviation of 1.5. This causes the data to be tightly clustered around the mean, making the bell shape much narrower.

By comparing the two plots, we observe the significance of the standard deviation in understanding the distribution's spread. While both distributions share the same center, their spreads are vastly different, and recognizing this difference is crucial in many statistical analyses and applications.

9.4 What can statistical measures do (and what not)?

9.4.1 First Data Exploration (using basic statistics)

In the code cell above we imported a dataset that consists of 14 different subsets ("A" to "P"). Each datapoint belongs into only one of those subsets and otherwise consists of only two numerical features: **x** and **y** which can think of as a 2D position.

```
data.head(3)
```

```
  dataset        x        y
0       A  55.3846  97.1795
1       A  51.5385  96.0256
2       A  46.1538  94.4872
```

We can use Pandas and its `groupby` method to quickly get several important statistical measures on all 14 subsets:

```
cols = [("x", "count"), ("x", "mean"), ("x", "std"),
        ("y", "count"), ("y", "mean"), ("y", "std")]
data.groupby('dataset').describe()[cols]
```

```
             x                               y
         count       mean        std  count       mean        std
dataset
A        142.0  54.263273  16.765142  142.0  47.832253  26.935403
B        142.0  54.266100  16.769825  142.0  47.834721  26.939743
C        142.0  54.261442  16.765898  142.0  47.830252  26.939876
D        142.0  54.269927  16.769959  142.0  47.836988  26.937684
E        142.0  54.260150  16.769958  142.0  47.839717  26.930002
F        142.0  54.267341  16.768959  142.0  47.839545  26.930275
G        142.0  54.268805  16.766704  142.0  47.835450  26.939998
H        142.0  54.260303  16.767735  142.0  47.839829  26.930192
I        142.0  54.267320  16.760013  142.0  47.837717  26.930036
J        142.0  54.268730  16.769239  142.0  47.830823  26.935727
K        142.0  54.265882  16.768853  142.0  47.831496  26.938608
L        142.0  54.267849  16.766759  142.0  47.835896  26.936105
M        142.0  54.266916  16.770000  142.0  47.831602  26.937902
```

9.4.2 Mini-Exercise: What do you think?

If you compare the mean and standard deviation (std) values for all 14 datasets (A to P), what do you expect them to look like? There is 142 datapoints in each subset. Will the datapoints of each subset be distributed similarly or not?

9.4.3 Solution

There is only one good way to find out and that is to inspect the data! Try, for instance, to run the following:

```
sb.relplot(x="x", y="y", col="dataset", kind="scatter", data=data, col_wrap=4)
```

I won't spoil it here right away, but what this dataset should illustrate once and for all is that basic statistical measures are a good starting point. But they often don't tell us enough about our data! By the way, this dataset is called the **datasaurus** [Matejka and Fitzmaurice, 2017].

9.5 Comparing Distributions Visually

When we're analyzing data, one of the most fundamental tasks is to understand the underlying distribution. Especially when dealing with multiple datasets, understanding and comparing their distributions can reveal patterns, outliers, and other crucial insights. As we just saw in the above exercise, visualization tools play an invaluable role in this exploration because they can tell us much more than a few statistical measures such as mean, median, or standard deviation alone. Here's a deeper look at some popular techniques for visual data representation:

1. **Box Plot**:
    - **Description**: A box plot (or whisker plot) displays a summary of a set of data values. It provides a visual summary of the minimum, first quartile, median, third quartile, and maximum of a dataset.
    - **Pros**:
        - Efficiently represents the data's spread and central tendency.
        - Useful for identifying outliers and comparing distributions across groups.
    - **Cons**:
        - Can be challenging to interpret for those unfamiliar with the plot's elements.
        - Doesn't represent the nuances in distributions; for instance, two very different distributions (like C and D) could have similar box plots.
2. **Strip Plot**:
    - **Description**: A scatter plot where one axis is categorical. It's useful for displaying all items in the dataset.
    - **Pros**:
        - Offers a sense of the density and distribution of data points.
        - Clearly shows individual data points.
    - **Cons**:
        - Can become cluttered and less interpretable with large datasets as points overlap.
        - Doesn't provide summary statistics about the data.
3. **Swarm Plot**:

- **Description**: Similar to a strip plot but data points are adjusted (or "swarmed") to avoid overlap, giving a clearer representation of the distribution of values.
- **Pros**:
  - Offers a clearer view of the distribution than strip plots for moderate datasets.
  - Good for visualizing the density of the data.
- **Cons**:
  - Not suitable for very large datasets as it can become cluttered and slow to render.
  - Like the strip plot, lacks summary statistics.

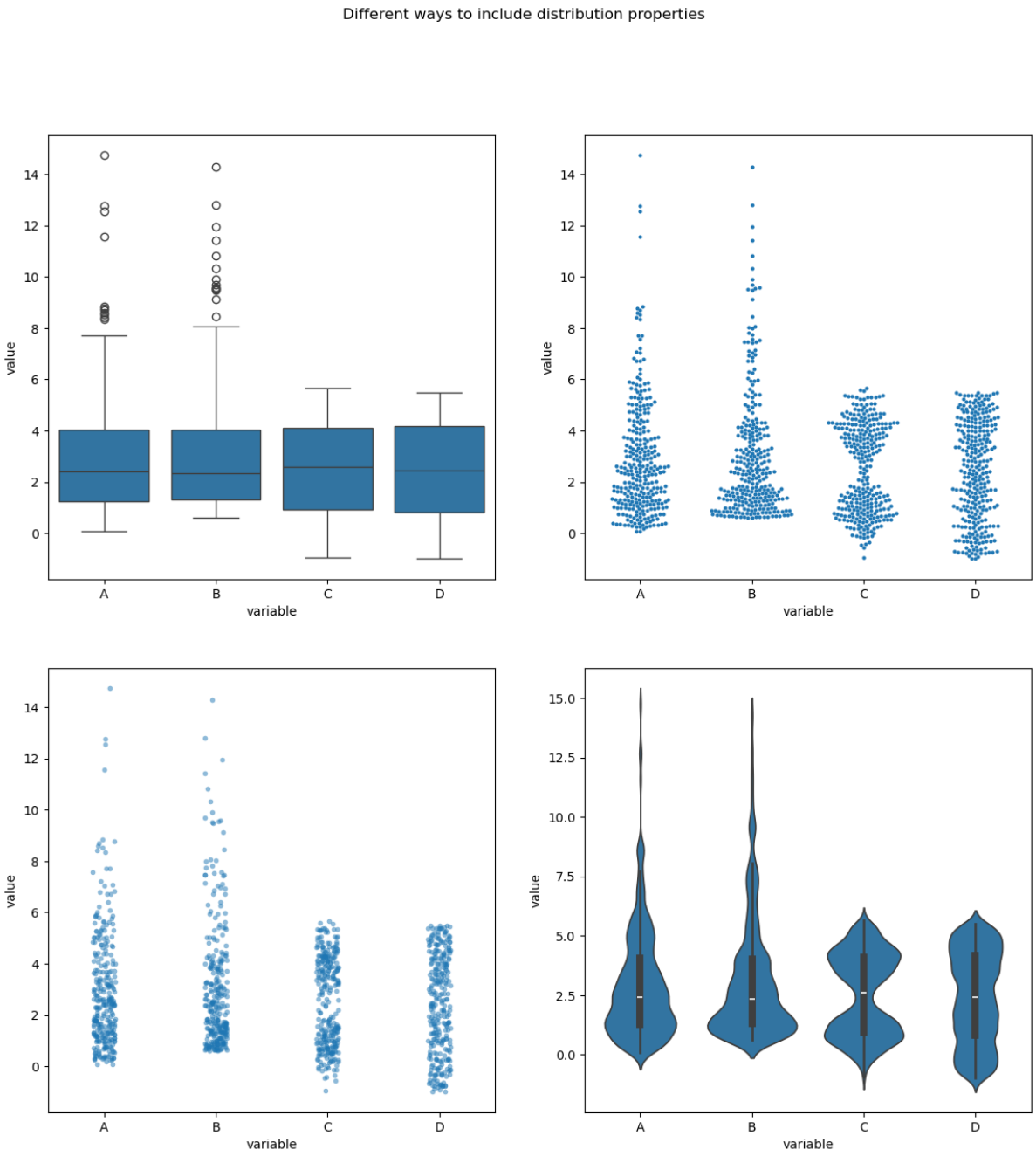
4. **Violin Plot**:
   - **Description**: Combines the benefits of both box plots and kernel density plots. It shows the full distribution of the data along with its summary statistics.
   - **Pros**:
     - Offers a detailed view of the data's distribution, including its density.
     - Combines the best of box plots (summary statistics) and density plots (distribution shape).
   - **Cons**:
     - Requires a degree of smoothing, which can be perceived as manipulating or altering the true nature of the data.
     - Can be harder for newcomers to interpret compared to simpler plots.

Using these visualization tools thoughtfully allows us to derive meaningful insights from our data, helping guide our analyses and decision-making.

```
fig, ax = plt.subplots(2, 2, figsize=(14, 14))

sb.boxplot(x="variable", y="value", data=pd.melt(datasets_test),
           ax=ax[0][0])

sb.stripplot(x="variable", y="value", data=pd.melt(datasets_test),
           ax=ax[1][0], alpha=0.5, size=4)

sb.swarmplot(x="variable", y="value", data=pd.melt(datasets_test),
           ax=ax[0][1], size=3)

sb.violinplot(x="variable", y="value", data=pd.melt(datasets_test),
           ax=ax[1][1], bw=.15)  # bw for "bandwidth" controls the degree of smoothing

fig.suptitle("Different ways to include distribution properties")
```

```
C:\Users\flori\AppData\Local\Temp\ipykernel_10992\2967438999.py:12: FutureWarning:

The `bw` parameter is deprecated in favor of `bw_method`/`bw_adjust`.
Setting `bw_method=0.15`, but please see docs for the new parameters
and update your code. This will become an error in seaborn v0.15.0.

  sb.violinplot(x="variable", y="value", data=pd.melt(datasets_test),
```

```
Text(0.5, 0.98, 'Different ways to include distribution properties')
```



By utilizing these visualization techniques, we can get a comprehensive understanding of our datasets' distributions and make more informed decisions during analysis.

9.6 Let's talk money

In real life we will come across distributions of all shapes, often non-symmetric. An even though it is considered polite in many cultures to not talk too much about our own wealth and incomes, we all probably know that income and wealth are unlikely to be narrow, symmetric distributions where most people have more or less the same, and earn more or less the same. I will leave the political debates up to you (but secretly do hope to spark some).

To me, things like wealth and income are good topics to learn about basic statistics. First, because we all know *something* about money and to some extend care about money (even if we say otherwise). And second, because it is full of striking example of skewed distributions. In fact, let me quote Thomas Piketty here [Piketty, 2014]:

> **Wealth is so concentrated that a large segment of society is virtually unaware of its existence.**

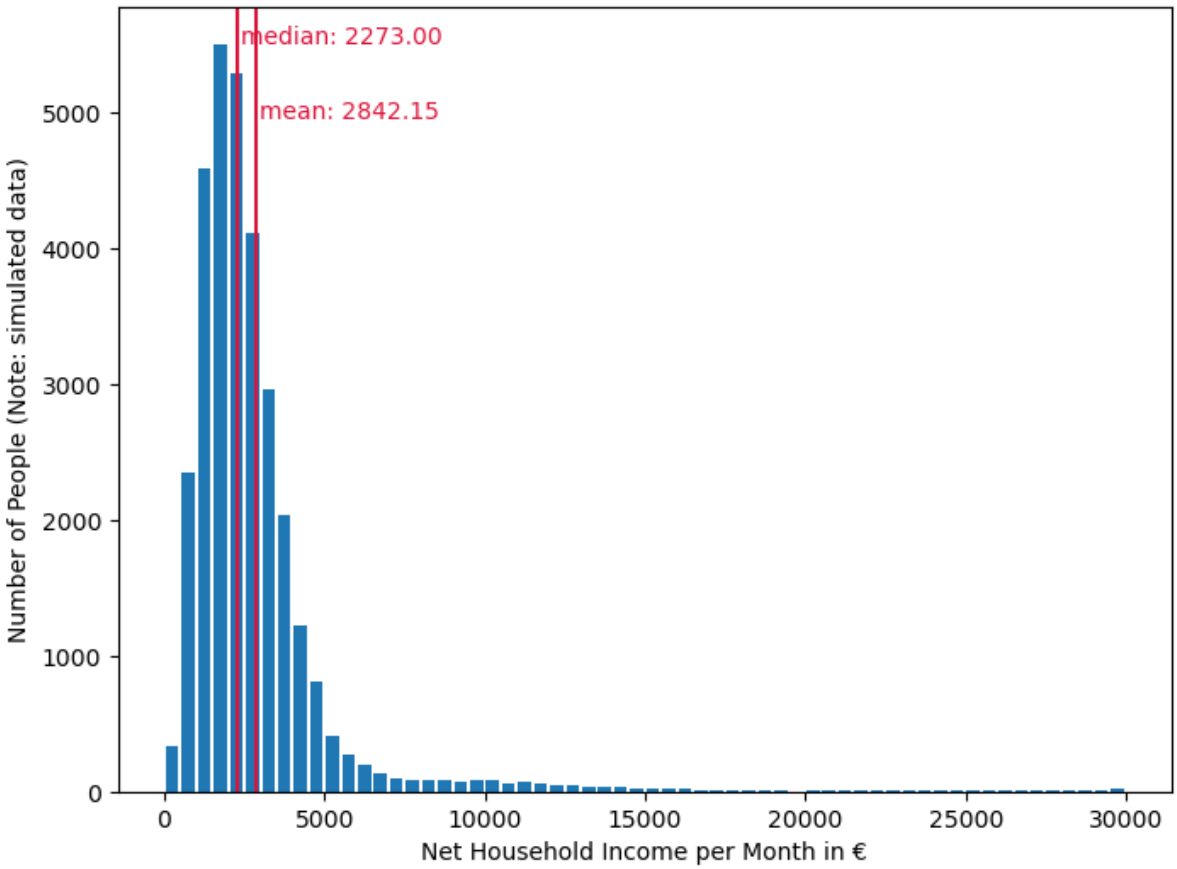
Real numbers on incomes in Germany, albeit already evaluated, can be found here.

For raw data requests, you can visit this site.

Let's dive into our fictional, but realistic, data:

```
# The visualization will depict income ranges and their frequencies.

plot_dist(income_data["income"], 60, ["mean", "median"])
plt.ylabel("Number of People (Note: simulated data)")
plt.xlabel("Net Household Income per Month in €")
```

```
Text(0.5, 0, 'Net Household Income per Month in €')
```



This data shows us the distribution of monthly household incomes, but we can dig deeper.

```
# Code to compute and display quantiles of the income data
income_data.quantile(q=np.arange(0, 1.1, 0.1))
```

```
      income
0.0      7.0
0.1   1069.9
0.2   1414.0
0.3   1705.0
0.4   1991.0
0.5   2273.0
0.6   2593.0
0.7   2989.0
0.8   3516.0
0.9   4483.1
1.0  29969.0
```

We can further dissect this data by dividing it into deciles:

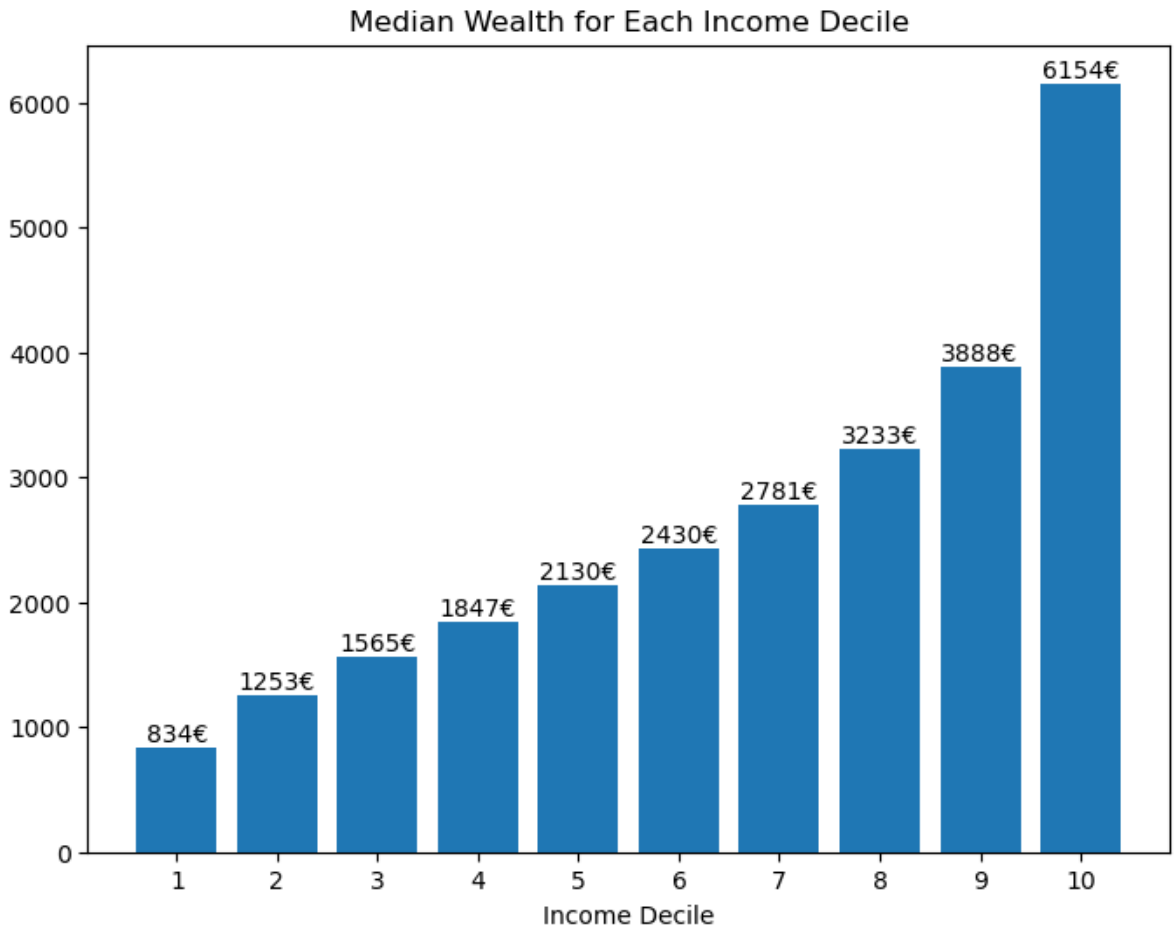
```
income_data["decile"] = pd.qcut(income_data['income'], q=10, labels=np.arange(10))
income_data.head()
```

```
   income decile
0    4912      9
1    2692      6
2    5118      9
3    1257      1
4    2083      4
```

Visualizing median income for each decile:

```
# Code to visualize the median income for each decile

fig, ax = plt.subplots(figsize=(8,6))
ax.bar(x=np.arange(10), height=income_data.groupby("decile").median()["income"])
for i, v in enumerate(income_data.groupby("decile").median()["income"]):
    ax.text(i, v + 50, f"{v:.0f}€", color='black', ha="center")

plt.xticks(ticks=np.arange(10), labels=np.arange(1,11), rotation=0)
plt.xlabel("Income Decile")
plt.title("Median Wealth for Each Income Decile")
plt.show()
```

```
C:\Users\flori\AppData\Local\Temp\ipykernel_10992\4219283847.py:4: FutureWarning:␣
 ↪The default of observed=False is deprecated and will be changed to True in a␣
 ↪future version of pandas. Pass observed=False to retain current behavior or␣
 ↪observed=True to adopt the future default and silence this warning.
  ax.bar(x=np.arange(10), height=income_data.groupby("decile").median()["income"])
C:\Users\flori\AppData\Local\Temp\ipykernel_10992\4219283847.py:5: FutureWarning:␣
 ↪The default of observed=False is deprecated and will be changed to True in a␣
 ↪future version of pandas. Pass observed=False to retain current behavior or␣
 ↪observed=True to adopt the future default and silence this warning.
  for i, v in enumerate(income_data.groupby("decile").median()["income"]):
```



Such a breakdown offers more insights into the income disparities.

```
# Code to determine and display the total income for each decile
income_sum = income_data.groupby("decile").sum()
income_sum["income"] *= 100/income_sum["income"].sum()
income_sum
```

```
C:\Users\flori\AppData\Local\Temp\ipykernel_10992\2203491340.py:2: FutureWarning:␣
 ↪The default of observed=False is deprecated and will be changed to True in a␣
 ↪future version of pandas. Pass observed=False to retain current behavior or␣
 ↪observed=True to adopt the future default and silence this warning.
  income_sum = income_data.groupby("decile").sum()
```

```
            income
decile
0         2.789230
1         4.418895
2         5.492008
3         6.492915
4         7.507340
5         8.548762
6         9.789732
7        11.394254
```

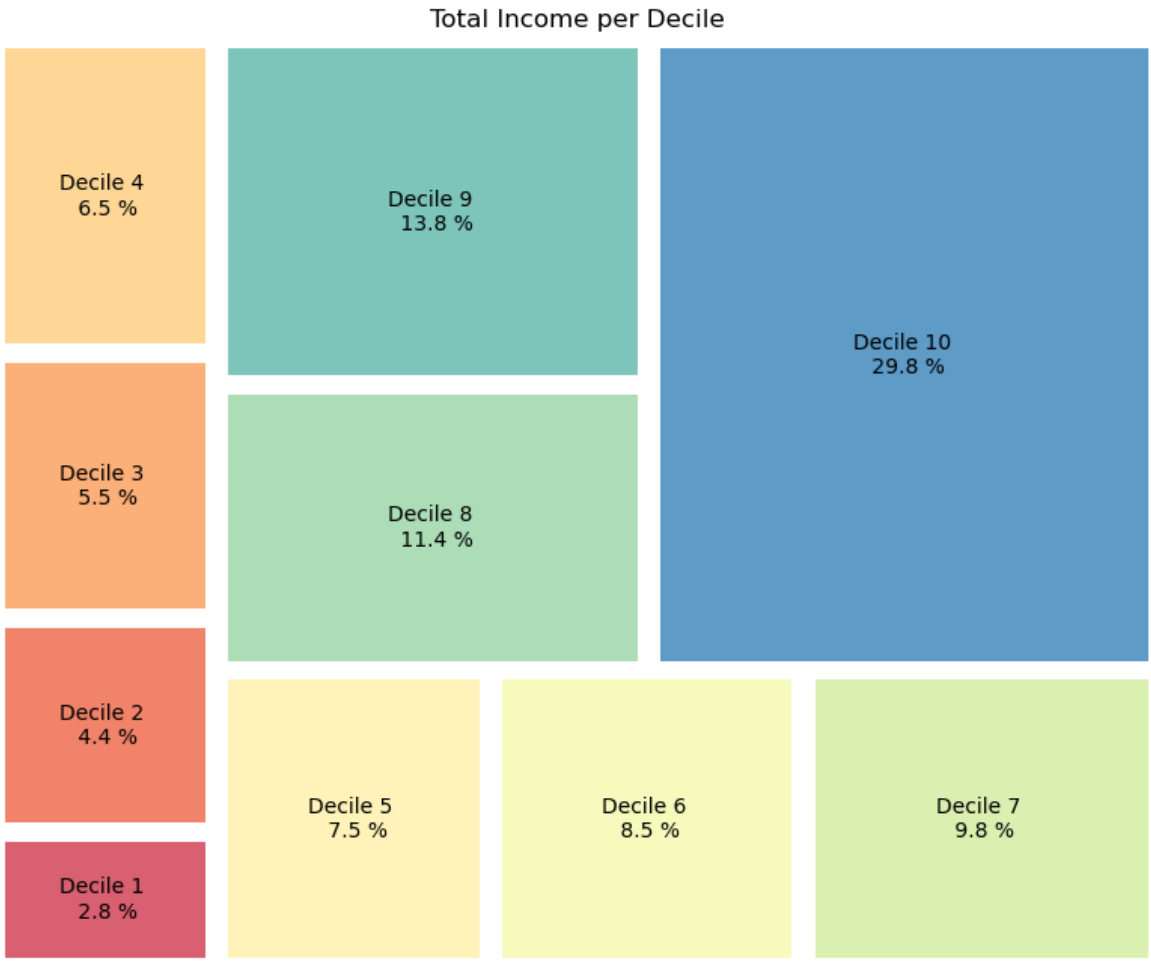
(continues on next page)

(continued from previous page)

```
8       13.790309
9       29.776555
```

This can be further visualized using a treemap:

```
# Code to visualize the income distribution across deciles using a treemap

# labels
labels = [f"Decile {i+1} \n {x:.1f} %" for i, x in enumerate(income_sum["income"])]

# plot
fig, ax = plt.subplots(figsize=(10,8))
squarify.plot(sizes=income_sum['income'], label=labels, alpha=.8,
              color=sb.color_palette("Spectral", len(income_sum)), pad=2)
plt.axis('off')
plt.title("Total Income per Decile")
plt.show()
```



9.6.1 Wealth

Typically, as the Piketty quote earlier on also said, wealth has a much more skewed distribution than income!

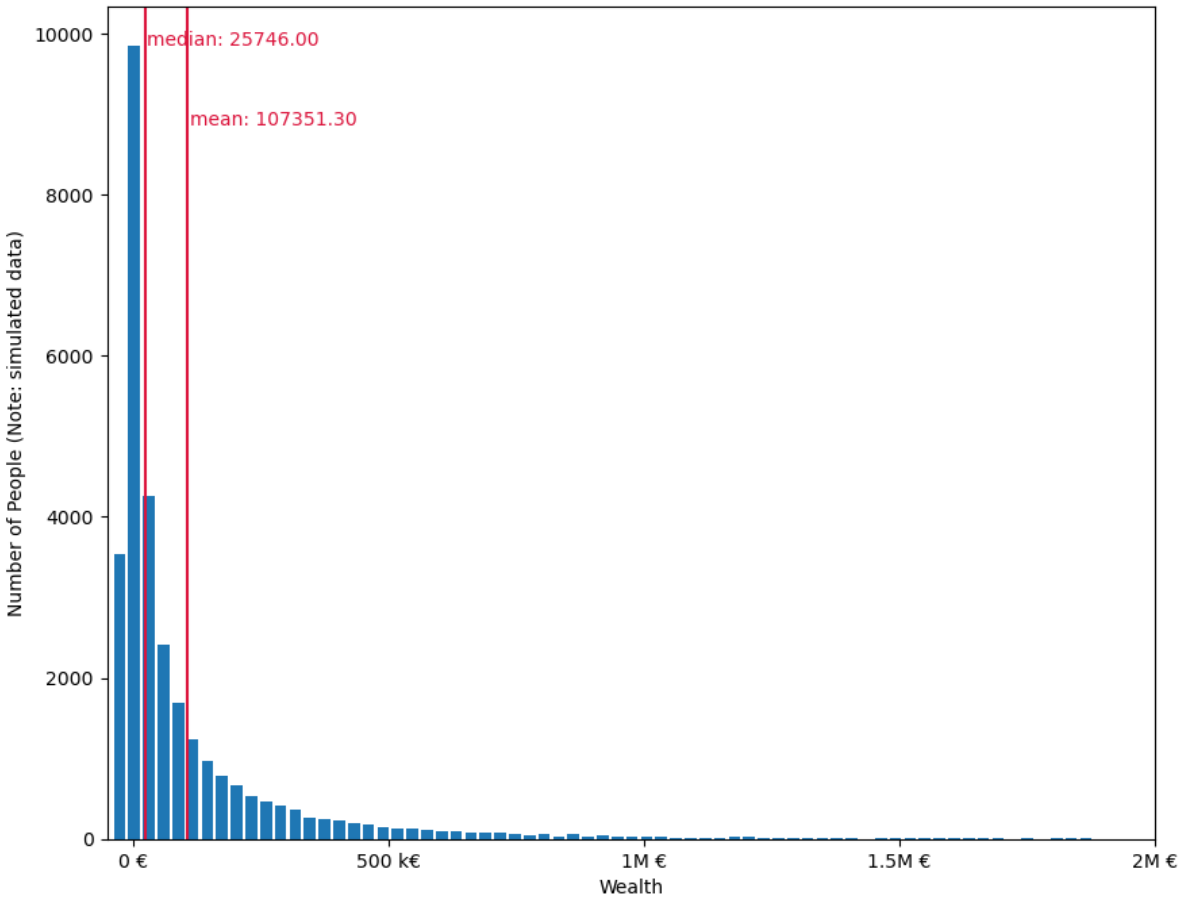
Here, we will again work with generated, but realistic, data.

```
wealth_data.describe()
```

```
             wealth
count  3.000000e+04
mean   1.073513e+05
std    2.156466e+05
min   -3.999200e+04
25%    9.970000e+02
50%    2.574600e+04
75%    1.188800e+05
max    2.826167e+06
```

Such wealth distributions, particularly when skewed, might be challenging to interpret directly. Different visual representations can offer distinct perspectives:

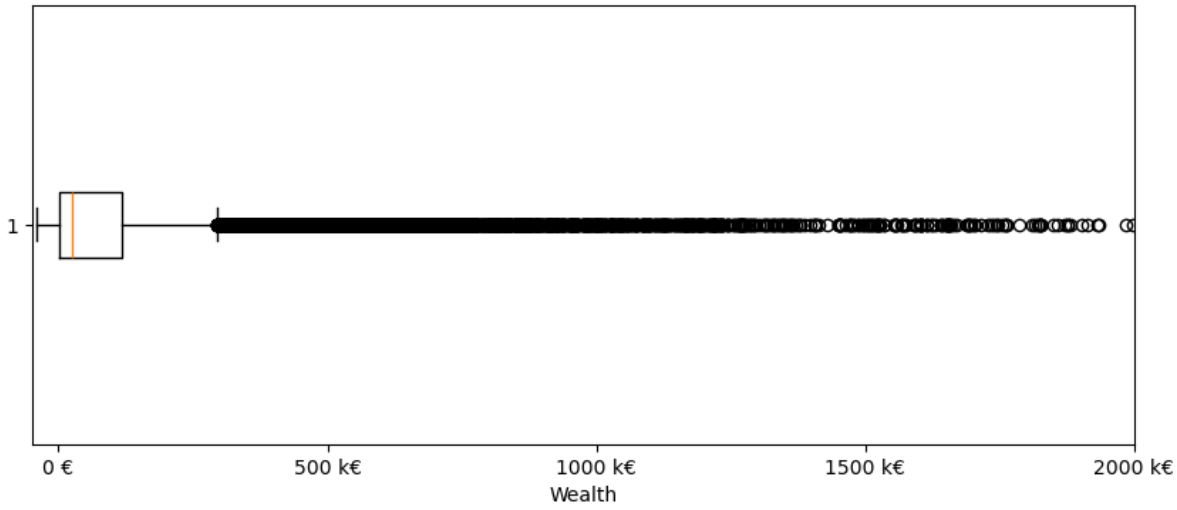
```
([<matplotlib.axis.XTick at 0x2022c7548e0>,
  <matplotlib.axis.XTick at 0x2022c7548b0>,
  <matplotlib.axis.XTick at 0x2022c754100>,
  <matplotlib.axis.XTick at 0x2022ca653a0>,
  <matplotlib.axis.XTick at 0x2022ca65c40>],
 [Text(0.0, 0, '0 €'),
  Text(500000.0, 0, '500 k€'),
  Text(1000000.0, 0, '1M €'),
  Text(1500000.0, 0, '1.5M €'),
  Text(2000000.0, 0, '2M €')])
```



This plot is technically OK. But is has a number of disadvantages. It is visually very centered on the left side due to the strong asymmetry in the wealth distribution. This also leads to the fact, that the *long tail*, that is all the fortunes above > 500k€ are practically impossible to read in this plot. We can try different plot types.

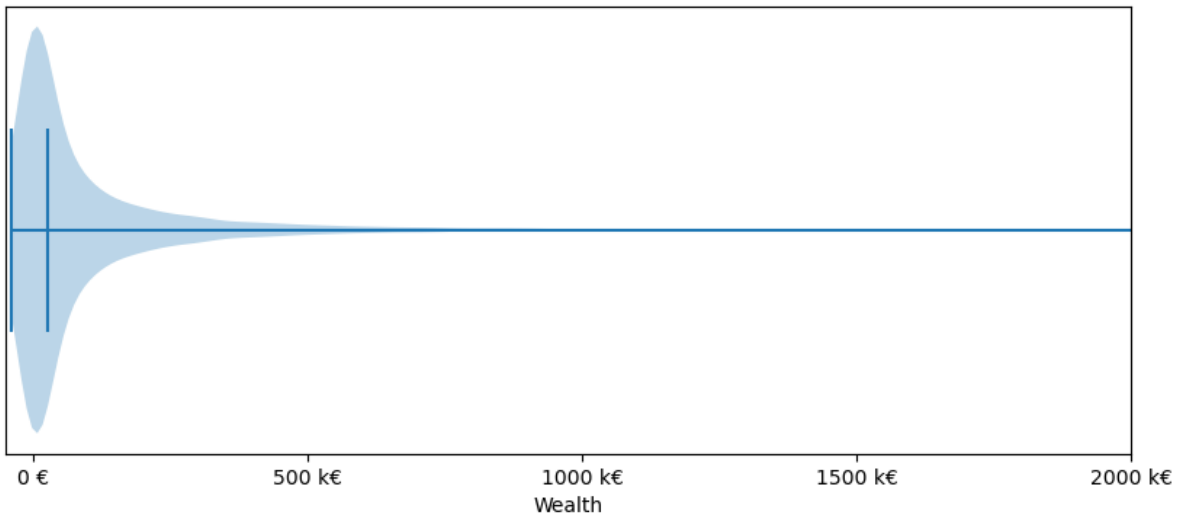
We will quickly see, that a **boxplot** is not making things better here:

```
([<matplotlib.axis.XTick at 0x2022ca9ebb0>,
  <matplotlib.axis.XTick at 0x2022ca9eb80>,
  <matplotlib.axis.XTick at 0x2022ca65d90>,
  <matplotlib.axis.XTick at 0x20227f37f10>,
  <matplotlib.axis.XTick at 0x2022896ac40>],
 [Text(0.0, 0, '0 €'),
  Text(500000.0, 0, '500 k€'),
  Text(1000000.0, 0, '1000 k€'),
  Text(1500000.0, 0, '1500 k€'),
  Text(2000000.0, 0, '2000 k€')])
```

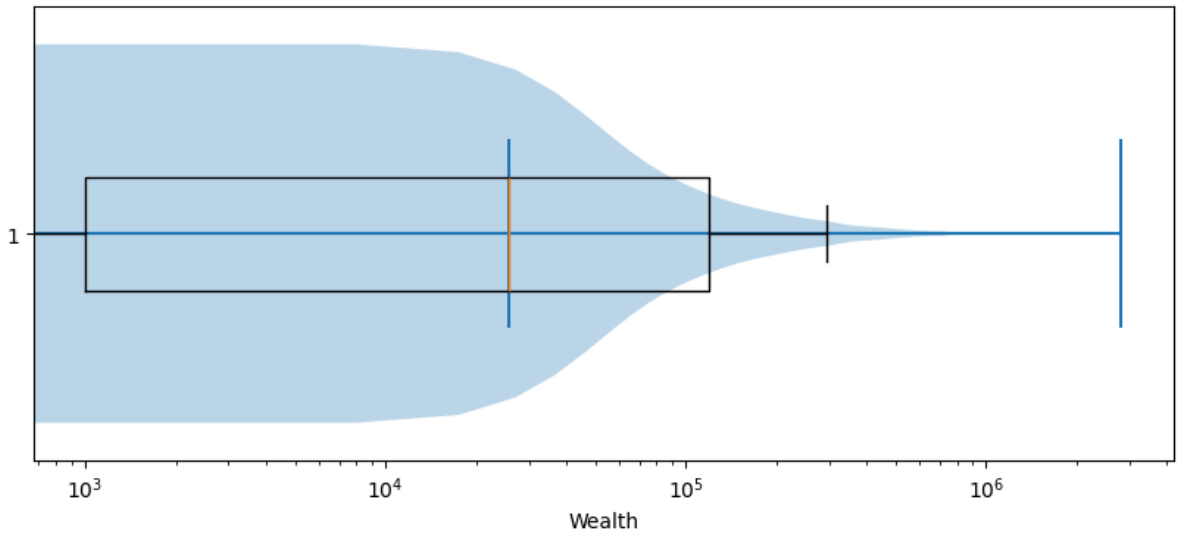


A **violin plot** is not as ugly as the boxplot in this case, but it also doesn't solve our problem.

```
([<matplotlib.axis.XTick at 0x2022c6a4f70>,
  <matplotlib.axis.XTick at 0x2022c6a4f40>,
  <matplotlib.axis.XTick at 0x2022c6a41f0>,
  <matplotlib.axis.XTick at 0x2022c689e50>,
  <matplotlib.axis.XTick at 0x2022c689a90>],
 [Text(0.0, 0, '0 €'),
  Text(500000.0, 0, '500 k€'),
  Text(1000000.0, 0, '1000 k€'),
  Text(1500000.0, 0, '1500 k€'),
  Text(2000000.0, 0, '2000 k€')])
```



Sometimes it helps to switch from a linear to a logarithmic scale. Not in this case though:

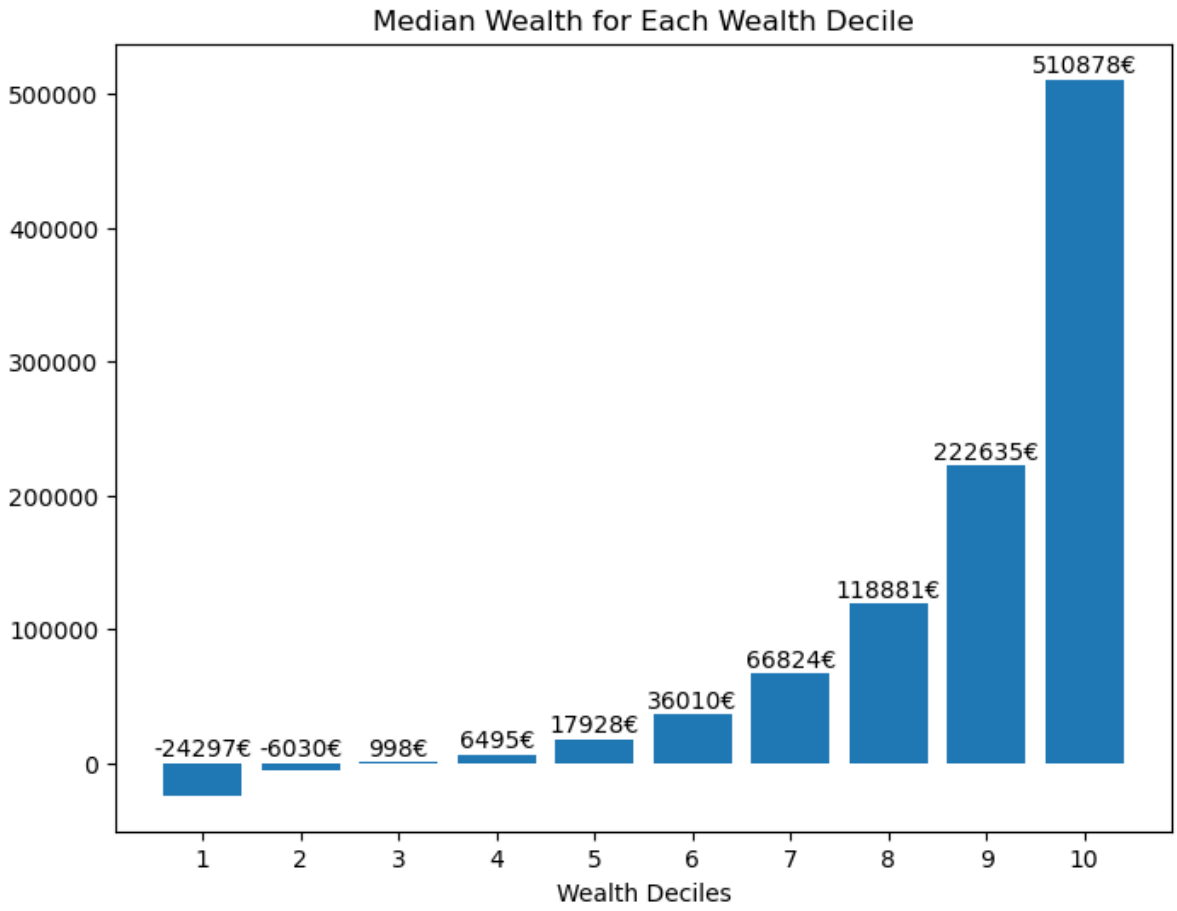


9.6.2 Exploring Wealth Distribution: A Different Approach

Analyzing wealth distribution can be quite abstract due to the vast range of values we might encounter. Even switching to non-linear scales is not overly satisfying here. Partly also, because capital can also be <= 0 (which, unfortunately, it often is).

One effective way to dissect this vastness is by segmenting it into more digestible chunks – like deciles.

```
C:\Users\flori\AppData\Local\Temp\ipykernel_10992\996220485.py:5: FutureWarning:␣
 ↪The default of observed=False is deprecated and will be changed to True in a␣
 ↪future version of pandas. Pass observed=False to retain current behavior or␣
 ↪observed=True to adopt the future default and silence this warning.
  ax.bar(x=np.arange(10), height=wealth_data.groupby("decile").median()["wealth"])
C:\Users\flori\AppData\Local\Temp\ipykernel_10992\996220485.py:8: FutureWarning:␣
 ↪The default of observed=False is deprecated and will be changed to True in a␣
 ↪future version of pandas. Pass observed=False to retain current behavior or␣
 ↪observed=True to adopt the future default and silence this warning.
  for i, v in enumerate(wealth_data.groupby("decile").median()["wealth"]):
```

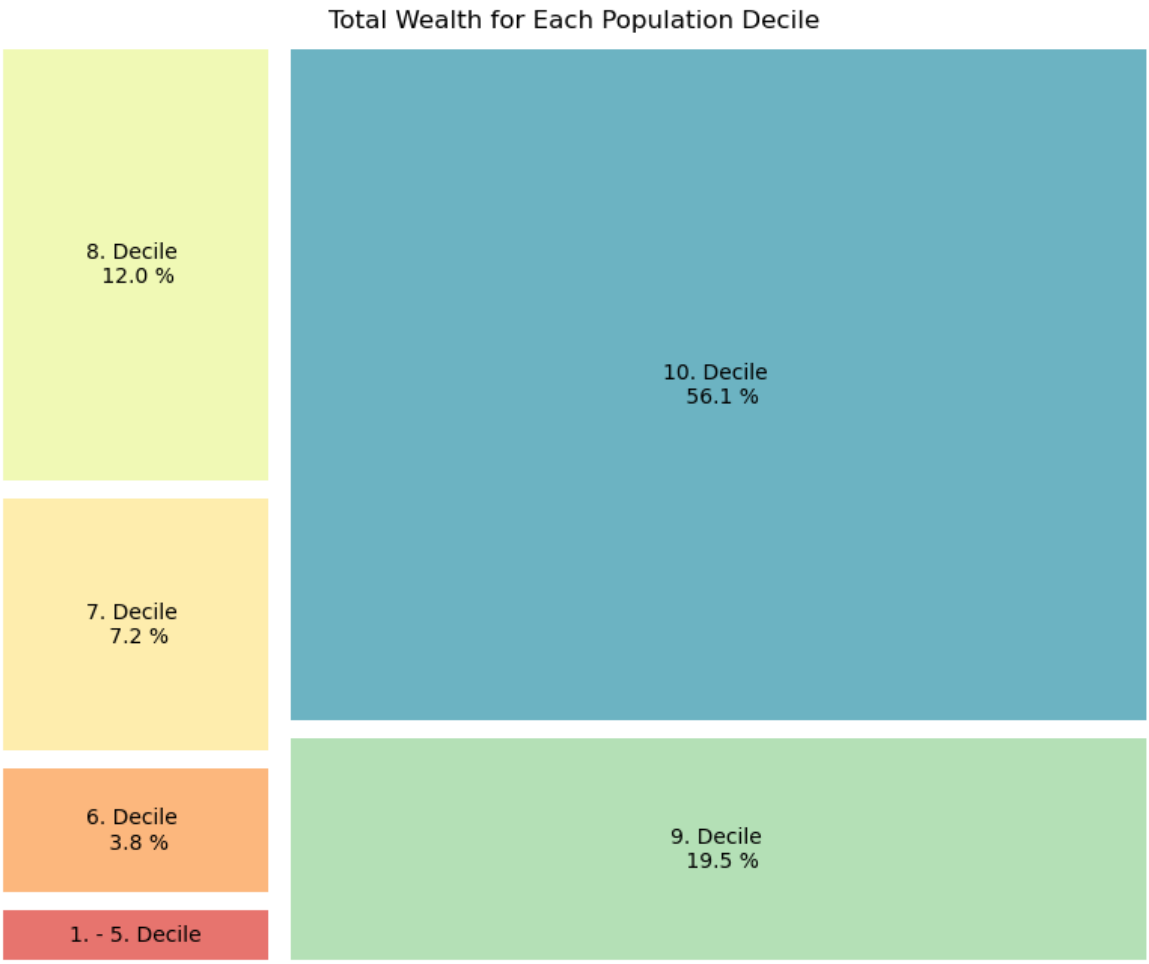


But if we wish to amplify the impact of these insights visually, we can take a more graphical approach.

Consider the following data which provides a breakdown of wealth percentages across certain segments of the population:

```
C:\Users\flori\AppData\Local\Temp\ipykernel_10992\2253224710.py:1: FutureWarning:␣
↪The default of observed=False is deprecated and will be changed to True in a␣
↪future version of pandas. Pass observed=False to retain current behavior or␣
↪observed=True to adopt the future default and silence this warning.
  wealth_sum = wealth_data.groupby("decile").sum()
```

```
           wealth
decile
0       -2.337298
1       -0.576260
2        0.113087
3        0.625263
4        1.681564
5        3.404859
6        6.310448
7       11.313193
8       21.260922
9       58.204221
```



This visualization provides an illustrative look at how wealth is dispersed across different segments of society. Through such visual aids, abstract numbers transform into tangible insights. And I bet they help a lot in sparking discussions on the distribution of wealth…

Part IV

In-depth Data Exploration

CHAPTER

TEN

CORRELATION ANALYSIS

In general terms, **correlation** refers to the relationship between two or more attributes, conditions, or functions (derived from the Latin *correlation* = mutual relationship).

In statistics, this concept is utilized more specifically as a "measure of association" that describes the strength and possibly the direction of dependence between two statistical variables.

Thus, we are looking for a metric that can numerically describe whether a correlation exists and, if so, how pronounced it is.

10.1 Covariance, (Pearson) Correlation Coefficient

A first step towards measuring correlation is through variance. This measures the spread of data points. Instead of variance, we first look at **covariance** (Cov), which is the average product of deviation.

$$Cov(X,Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x_i})(y_i - \bar{y_i})$$

The covariance already gives a good indication of what we understand by correlation. A significant disadvantage, however, is that the covariance is not normalized. Depending on the range of values, the covariance can become arbitrarily large (or small).

The most famous normalized correlation coefficient is the **Pearson Correlation Coefficient** (or often just: Correlation Coefficient). This is nothing more than the covariance divided by the product of the square roots of the respective variances (Var)

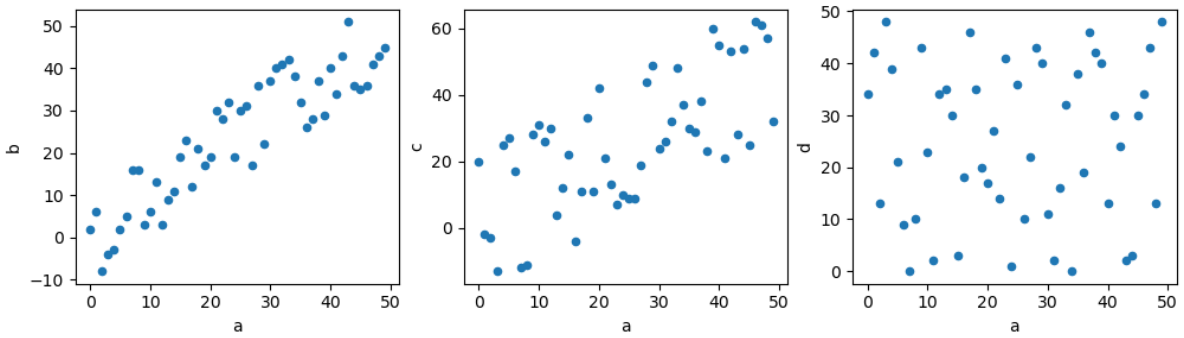
$$Corr(X,Y) = \frac{Cov(X,Y)}{\sqrt{Var(X)}\sqrt{Var(Y)}}$$

Now let's have a look at how correlations look like and how they can be quantified.

```
corr_data.head(3)
```

```
   a  b   c   d
0  0  2  20  34
1  1  6  -2  42
2  2 -8  -3  13
```

We created some toy data with four features (a, b, c, d). One way to look for interesting relationships between different features is simply to plot one variable (or feature) against another:



What would you say, which features are correlated, and which are not?

Here, different variables are plotted against each other, a-b, a-c, and a-d. I guess most will agree that the first and second plot show a certain degree of correlation. First, between a and b, and in the second panel between a and c. The third panel shows two features which apparently are entirely unrelated in their behavior. Another thing we can see in those examples is that we should be able to measure different degrees or strengths of a correlation.

We can clearly see the correlation in a<->b. The Pearson Correlation Coefficient measures this type of correlation very well. For this example, the values are $Corr(a, b) = 0.90$, $Corr(a, c) = 0.68$, and $Corr(a, d) = 0.02$. We would thus see a strong correlation between a and b and a moderate correlation between a and c. However, there is no noticeable correlation between a and d.

The correlation coefficient allows us to quickly identify significant correlations in the data. For datasets with many variables (or features), we can easily determine the correlations for all combinations of the variables, resulting in what is called a correlation matrix.

10.2 Correlation Matrix

The correlation matrix for data with the variables a, b, and c would thus be a matrix that contains the Pearson Correlation Coefficients for all possible combinations, i.e.:

	a	b	c
a	$Corr(a, a)$	$Corr(b, a)$	$Corr(c, a)$
b	$Corr(a, b)$	$Corr(b, b)$	$Corr(c, b)$
c	$Corr(a, c)$	$Corr(b, c)$	$Corr(c, c)$

The diagonal is trivial and always equals 1. The actual points of interest are the particularly high and low values off the diagonal, as they indicate strong correlations or anti-correlations (with a negative sign).

Using `Pandas` we can very easily compute the Pearson correlations between all possible pairs of numerical features in a dataset:

```
corr_data.corr()
```

```
          a         b         c         d
a  1.000000  0.900512  0.675015  0.022422
b  0.900512  1.000000  0.533063 -0.124933
c  0.675015  0.533063  1.000000  0.105887
d  0.022422 -0.124933  0.105887  1.000000
```

More specific example

In the following we still look at generated data, but data which is somewhat realistic. We want to see if body measures show any interesting correlations, say between shoe size and height of people.

```
                 time    sex  height  shoe_size  age
0  04.10.2016 17:58:51  woman   160.0       40.0   56
1  04.10.2016 17:58:59  woman   171.0       39.0   24
2  04.10.2016 18:00:15  woman   174.0       39.0   35
3  04.10.2016 18:01:17  woman   176.0       40.0   47
4  04.10.2016 18:01:22    man   195.0       46.0   24
```

Let's first do some standard inspection of the data.

```
data.describe()
```

```
           height  shoe_size         age
count  100.000000  100.00000  101.000000
mean   165.233800   39.77500   43.227723
std     39.817544    5.55613   15.786628
min      1.630000   35.00000   17.000000
25%    163.000000   38.00000   30.000000
50%    168.500000   39.00000   42.000000
75%    174.250000   40.00000   56.000000
max    364.000000   88.00000   70.000000
```

Basic cleaning or processing

In the table above, some things seem weird. The minimum height is 1.63 and the maximum height is 364.0. Probably some fantasy figures, one apparently also with shoe size 88. This are typical issues we can quickly discover in a first inspection and then decide what to do about it. Here, we will simply decide to only take data within more or less realistic boundaries.

```
mask = (data["shoe_size"] < 50) & (data["height"] > 100) & (data["height"] < 250)
data = data[mask]
```

We can then move on to the correlations:

```
data.corr()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[9], line 1
----> 1 data.corr()

File ~\anaconda3\envs\data_science\lib\site-packages\pandas\core\frame.py:10707,␣
 ↪in DataFrame.corr(self, method, min_periods, numeric_only)
  10705 cols = data.columns
  10706 idx = cols.copy()
> 10707 mat = data.to_numpy(dtype=float, na_value=np.nan, copy=False)
  10709 if method == "pearson":
  10710     correl = libalgos.nancorr(mat, minp=min_periods)

File ~\anaconda3\envs\data_science\lib\site-packages\pandas\core\frame.py:1892, in␣
 ↪DataFrame.to_numpy(self, dtype, copy, na_value)
```

(continues on next page)

(continued from previous page)

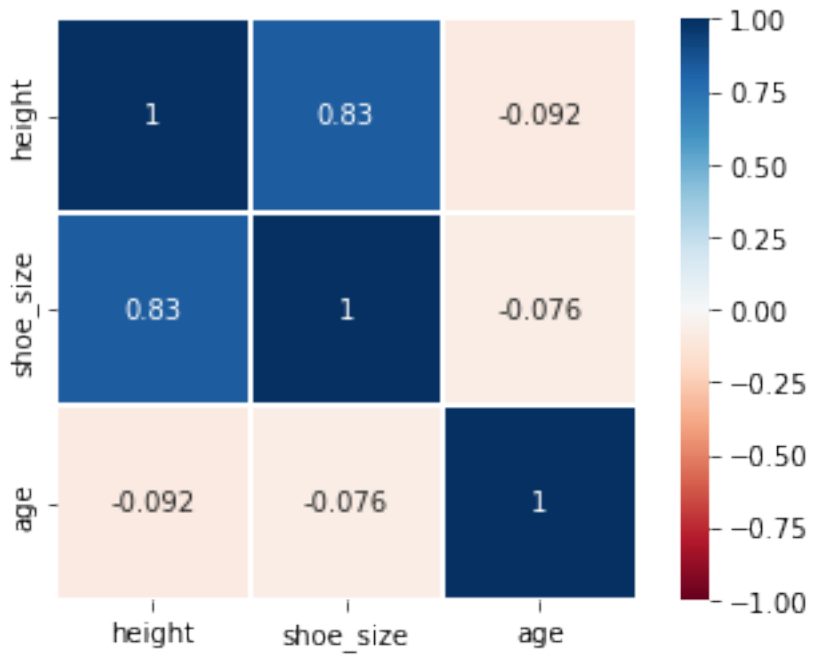
```
   1890 if dtype is not None:
   1891     dtype = np.dtype(dtype)
-> 1892 result = self._mgr.as_array(dtype=dtype, copy=copy, na_value=na_value)
   1893 if result.dtype is not dtype:
   1894     result = np.array(result, dtype=dtype, copy=False)

File ~\anaconda3\envs\data_science\lib\site-packages\pandas\core\internals\
 ↪managers.py:1656, in BlockManager.as_array(self, dtype, copy, na_value)
   1654         arr.flags.writeable = False
   1655 else:
-> 1656     arr = self._interleave(dtype=dtype, na_value=na_value)
   1657     # The underlying data was copied within _interleave, so no need
   1658     # to further copy if copy=True or setting na_value
   1660 if na_value is lib.no_default:

File ~\anaconda3\envs\data_science\lib\site-packages\pandas\core\internals\
 ↪managers.py:1715, in BlockManager._interleave(self, dtype, na_value)
   1713     else:
   1714         arr = blk.get_values(dtype)
-> 1715     result[rl.indexer] = arr
   1716     itemmask[rl.indexer] = 1
   1718 if not itemmask.all():

ValueError: could not convert string to float: '04.10.2016 17:58:51'
```

Question: What does that actually mean?

Instead of as a matrix with values, the correlation matrix is often also graphically represented, especially for larger datasets, to easily spot particularly high and low coefficients.

```
fig, ax = plt.subplots(figsize=(8, 8))
sb.heatmap(data.corr(), vmin=-1, vmax=1,
           square=True, lw=2,
           annot=True, cmap="RdBu",
           ax=ax,
          )
```

```
<AxesSubplot:>
```



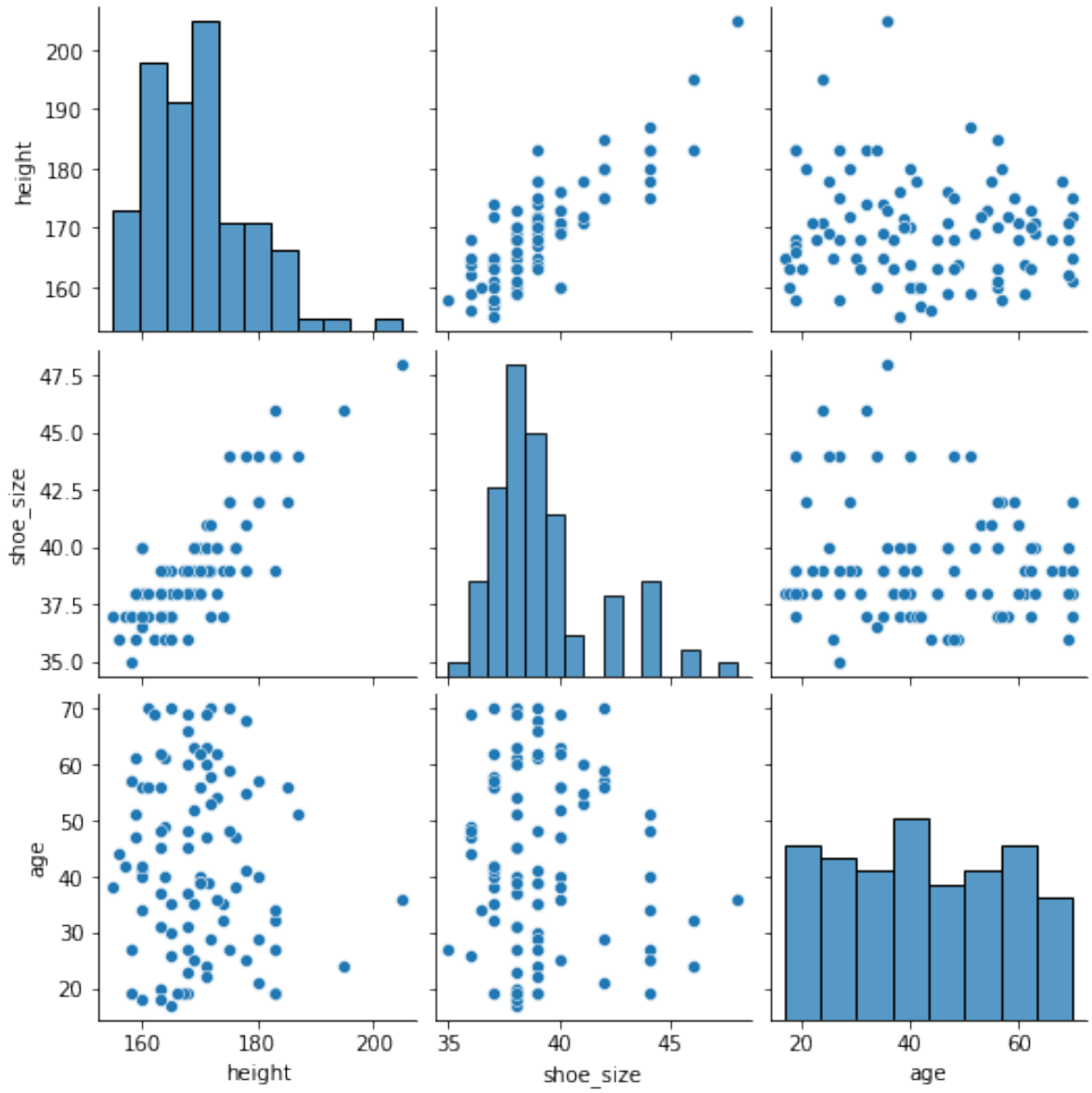
While high (and low) correlation coefficients often indicate actual correlations in the data, there are also a number of issues/limitations with this approach:

- It only detects linear relationships. A low correlation doesn't mean there isn't a significant relationship!
- The value of the slope has no influence (though the sign does).

Just like before, with statistical metrics vs. actual distributions, the correlation coefficient sometimes proves to be too *rough*, and we might want to look more closely at the data. If there aren't too many variables, this can be done with the `pairplot()` function from seaborn:

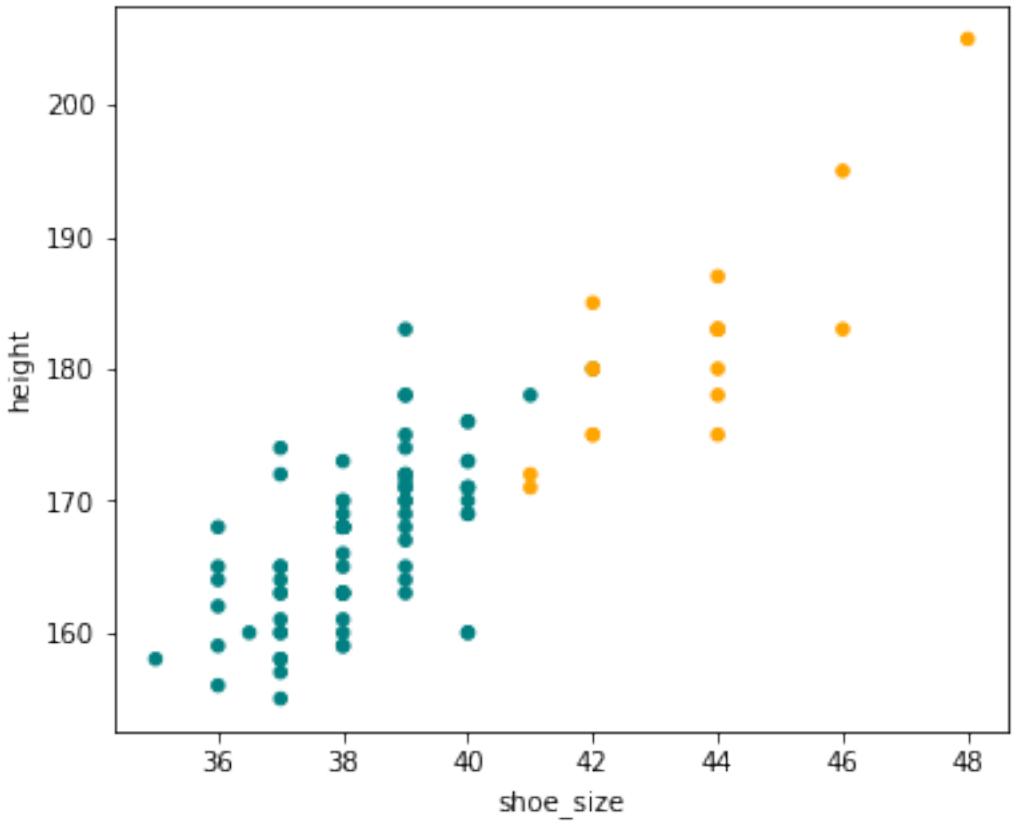
```
sb.pairplot(data)
```

```
<seaborn.axisgrid.PairGrid at 0x264ba9ec760>
```



It is of course also easily possible to plot only the parts we are most interested in, for instance:

```
<AxesSubplot:xlabel='shoe_size', ylabel='height'>
```



10.3 What Does a Correlation Tell Us?

If we discover correlations in our data, for example, through high or low Pearson Correlation Coefficients, what can we do with it?

Although high or low coefficients are often the first thing we look for, a value of 1.0 or -1.0 isn't necessarily a cause for celebration, even though these values describe a maximum correlation.

The reason is simple. If two variables are perfectly correlated, it means one can perfectly describe the other. They do not provide more information than just one of them alone. Moreover, in practice, a correlation of 1.0 (or -1.0) is more likely a result of both variables describing the same thing. Typical examples would be temperature given in both Celsius and Kelvin or the circumference and diameter of wheels. In such cases, correlations mainly tell us which variables can be discarded..

10.4 Correlation vs. Causality

Searching for correlations within data is often a pivotal step in the data science process. But why? What can we do with a high correlation (that isn't too high, i.e., equal to 1.0)?

The significance of a discovered correlation entirely depends on the problem posed or the task at hand. In extreme scenarios, two possible outcomes arise:

1. **We've Achieved Our Goal!** The correlation is all we need. For instance, situations where a connection merely needs to be identified, not necessarily understood. Suppose in an online store we discover that 78% of customers

who purchased item X also bought item Y (and this conclusion is supported by a reliable data foundation). This insight would suffice to make relevant suggestions to customers or even reorder items X and Y in tandem.

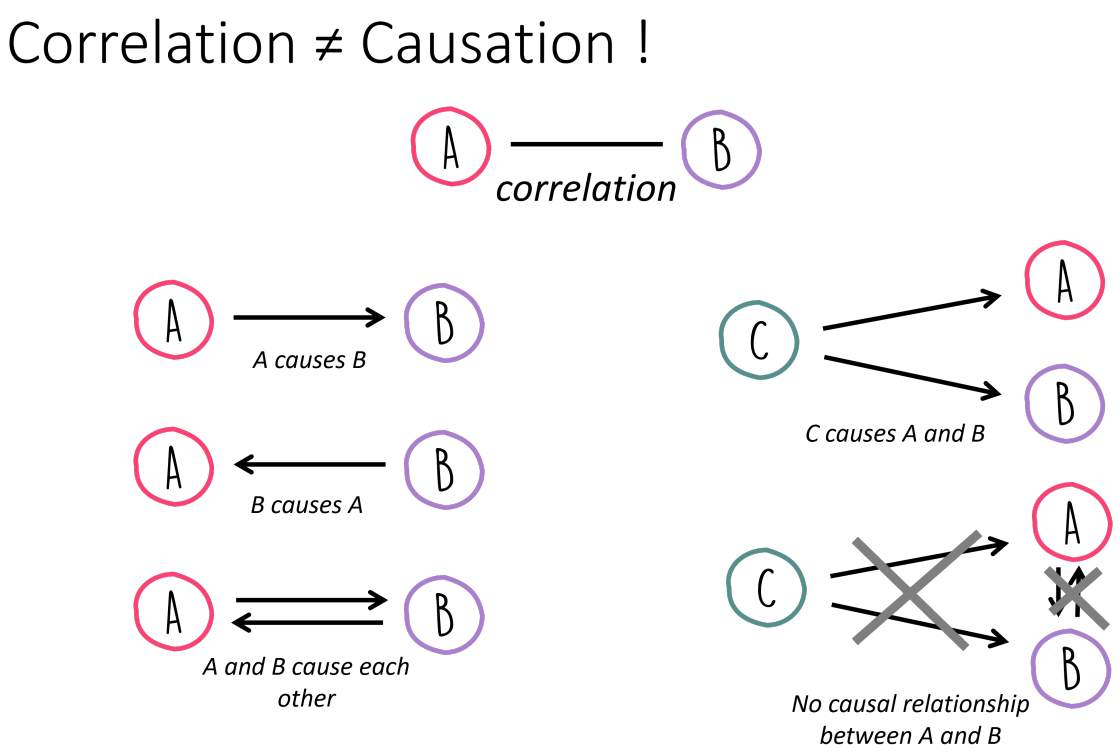
2. **It's Just the Beginning!** In this case, the correlation is merely an initial hint in the search for deeper relationships. For example, determining if customers purchase item Y **because** they bought item X. Or, more broadly, if a correlation exists between A and B, whether **A is the cause of B**. This leads us to the concept of causality…

Causality means that if A is the cause for effect B, then B is produced by A. Now, the question arises, how does this differ from correlation?

In daily life, correlation and causality are often confused. This mix-up occurs because it's inherently human to *suspect* causal relationships behind observed correlations. Examples of such observations include:

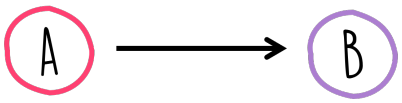
- "I ate fruits from tree B and then fell ill."
- "I took medicine X, and an hour later, I felt better."

In both instances, we're essentially observing only a correlation, yet we quickly infer a causal link. Without additional information or experiments, a correlation serves at best as a hint for a possible causal relationship. When A correlates with B, this can potentially be explained by various causal relationships:

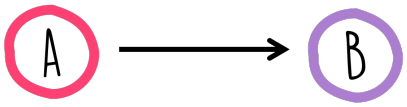


Things become a tad more complex, as there isn't one definitive path from identifying a correlation to proving a causal relationship! Often, targeted experiments can help validate or refute hypotheses about causal links. For instance, if eliminating A should result in the absence of B, assuming the causality $A \rightarrow B$ holds true. However, this only applies if no other underlying cause could "step in" and also trigger B.

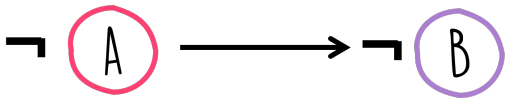
hypothesis:



experiment: more A leads to more B



counterfactual dependency: „not A“ implies „not B“



CHAPTER

ELEVEN

CLUSTERING

Clustering is a technique that groups together data points based on their similarities. In light of the data science workflow discussed before `ch_workflow` clustering can be used as a data exploration tool as well as to model our data. In this section, we will explore different clustering algorithms, including KMeans and DBSCAN, and apply them to synthetic datasets. We will discuss how clustering algorithms work and what their different advantages and limitations are.

11.1 What is clustering?

Clustering refers to the process of searching for a "natural" or useful divide of data points into a number of groups or: clusters. There is a commonly shared intuition that a "good" cluster is one where all datapoints have high similarity or share some important properties. In same cases, this can indeed be simple in the sense that most people would intuitively agree on a suitable division into clusters, such as illustrated in Fig. 11.1. At least, I haven't yet had a student that gave an answer that was different from "three" when asked for the number of groups in this dataset. In practice, however, those question will not be answered by any of my students, nor by me. We will use a range of different **clustering algorithms** to do the clustering for us.

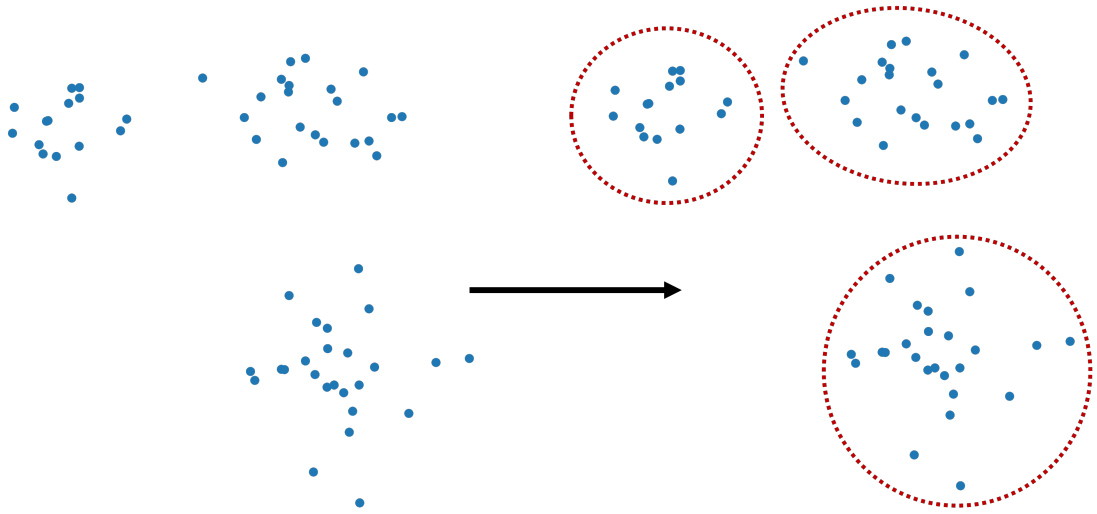


Fig. 11.1: Clustering refers to dividing data points into seperate groups or: clusters.

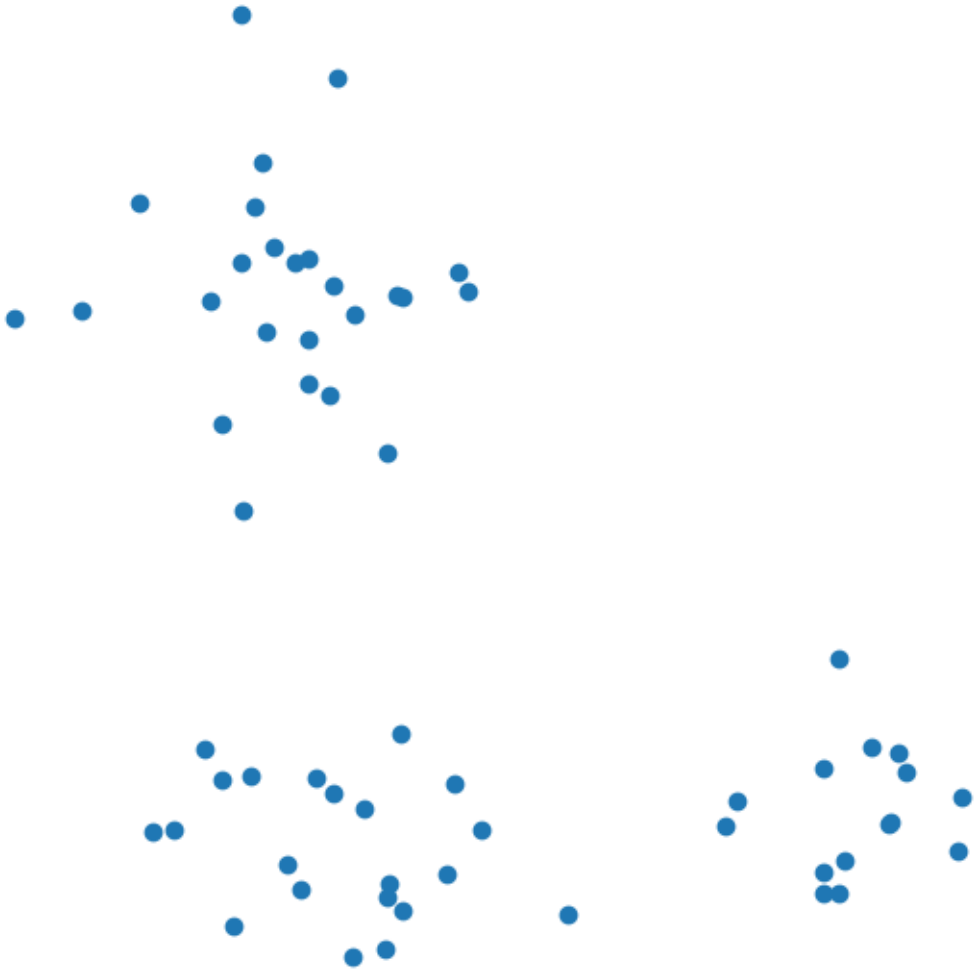
Next, we create such a synthetic dataset with three clusters. This dataset consists of three groups of points, each generated from a different Gaussian distribution. Each group of points has its own mean and standard deviation along the x and y axes.

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

# optional, only to avoid KMeans warning on Windows (too few chunks compared to␣
↪threads)
import os
os.environ["OMP_NUM_THREADS"] = "2"
```

```
clusters = [(15, 0.5, 0.5, 1.5, -0.7),
            (21, 0.8, 0.5, -2.1, -1.2),
            (25, 0.8, 0.7, -2.6, 2.3)]

np.random.seed(42)
data = np.zeros((0, 2))
for (n_points, x_scale, y_scale, x_offset, y_offset) in clusters:
    xpts = np.random.randn(n_points) * x_scale + x_offset
    ypts = np.random.randn(n_points) * y_scale + y_offset
    data = np.vstack((data, np.vstack((xpts, ypts)).T))
```

We visualize the dataset using pyplot. The dataset contains three visually distinct clusters.

```
fig, ax = plt.subplots(figsize=(7, 7))

ax.scatter(data[:,0], data[:,1])
plt.axis('off')
plt.show()
```



Assigning the three clusters in this case does not exactly feel like rocket science. We could all do this manually with a pen in no time. So, why bother to use - let alone understand or develop - algorithms for this task?

There are two reasons for why the shown figure makes the task look easy. First, the points are distributed in a way that immediately suggests clear boundarys or center positions. And everyone knows that not all cases will look that simple. More importantly, however, is the fact that in most cases our datapoints will be nothing we can simply "look" at, or draw lines around, because the data won't be 2-dimensional. Instead datpoints might have any number of features, which is exactly what I mean here with *dimensions*. And in 4D, 5D, 6D, 10D, 100D, there is no chance to manually assign clusters to datapoints.

In addition, there are even more reasons to rather search for a good algorithm than to assign clusters manually. Algorithms allow to cluster much larger datasets. And, in some cases, they will give consistent, reproducible results, i.e., they will assign each element the same cluster whenever we run the algorithm again. We will later see, that this is not true for all clustering algorithms.

11.1.1 Why do we want to cluster data?

Clustering data serves as a fundamental technique in the data scientist's toolkit for a myriad of reasons beyond the mere convenience of handling multidimensional data. One significant motive is the discovery of intrinsic patterns and structures within the data that are not apparent through simple observation, especially when dealing with high-dimensional datasets. These hidden patterns can unveil relationships, categories, and subclasses within the data that can lead to valuable insights and, ultimately, inform decision-making processes.

Moreover, clustering enables us to summarize a vast amount of data through the formation of representative groups or clusters. This is not only efficient for data compression but also crucial for simplifying complex data for further analysis, visualization, and reporting. When we group similar data points together, we can study these groups' characteristics to understand the underlying population better. For instance, in customer segmentation, clustering helps in identifying groups with common behaviors, leading to targeted marketing strategies.

Another compelling reason to employ clustering is anomaly detection. By understanding what is 'normal' for a given cluster, it becomes easier to spot outliers or anomalies that may signify errors, fraud, or new, previously undocumented phenomena. This is particularly relevant in fields such as cybersecurity, where clustering can help in identifying unusual patterns that may indicate security breaches.

Lastly, clustering can serve as a pre-processing step for other algorithms. By organizing data into clusters, we can run other algorithms more efficiently within each cluster, reducing computational costs and improving algorithm performance. This layered approach can be seen in complex tasks such as image and speech recognition, where initial clustering can significantly streamline subsequent analysis.

In summary, clustering is not just a convenience but a powerful method for uncovering and leveraging the rich, multi-dimensional structures in data, which in turn can lead to more effective and informed decision-making across diverse domains.

Hard and Soft Clustering

In most cases, **clustering** refers to the process of assigning at most **one** cluster to each data point. It is hence a binary decision that needs to be made, either manually, or -and that is what we are interested in here- in an automated way. It is also possible to allow clusters to overlap such that elements can belong to more than one cluster. This is then called *soft clustering* as displayed in Fig. 11.2. Most commonly, and in this introductory book as well, the term clustering primarily refers to *hard clustering*.

11.1.2 Mini-exercise

Team up with your neighbor (if in a class), or try on your own. Looking at the data displayed in Fig. 11.1, how would you write an algorithm to automatically assign each datapoint a cluster it belongs to?

11.1.3 What did you come up with?

When I ask this to students, I typically get many different answers. Including many suggestions that are very likely to work for the displayed example. One way to go is to look at distances between points and somehow define how far points are allowed to be before they no longer fall into the same cluster. Other approaches might include fitting lines, borders, shapes into the scatter datapoints. But there are many more options one could come up with. It is therefore no big surprise, that many different clustering algorithms exist.

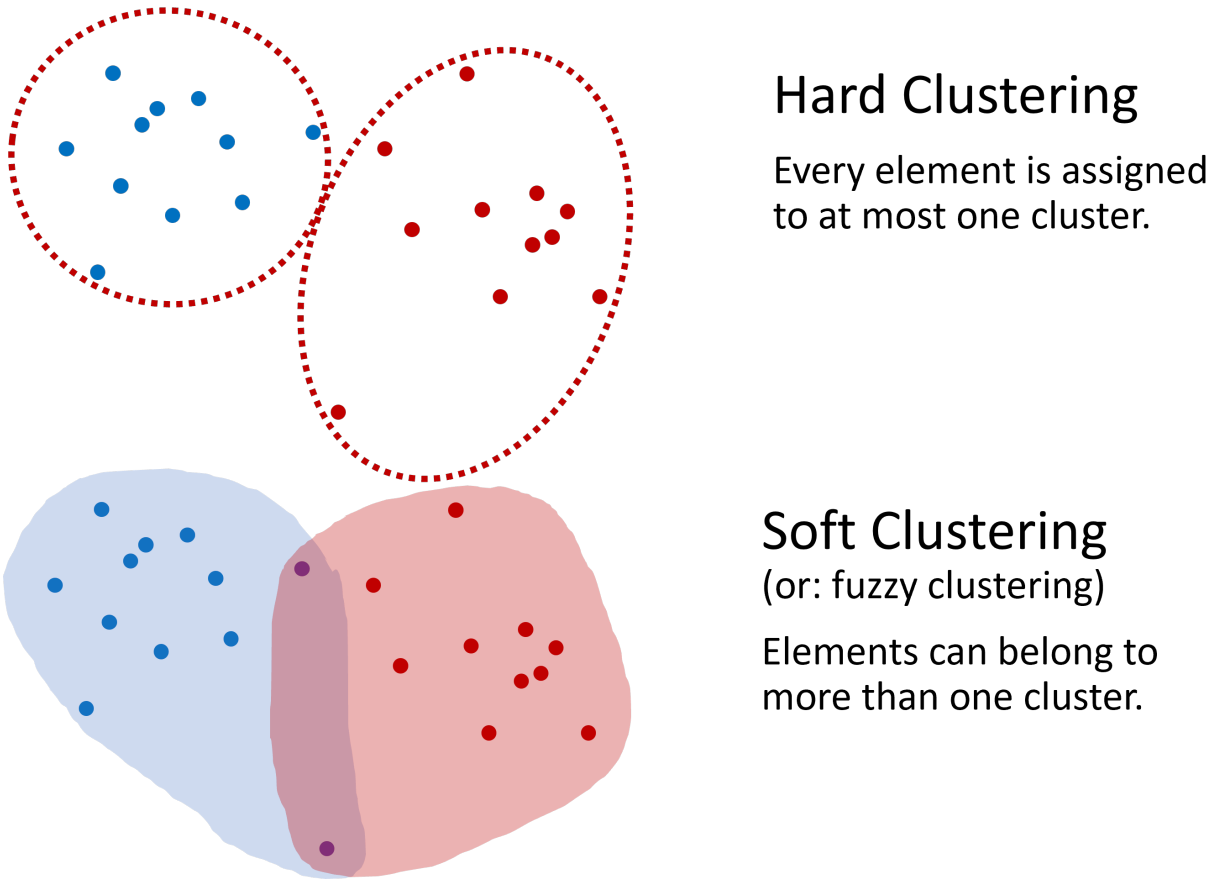


Fig. 11.2: Two conceptually differnt types of clustering are *hard clustering* in which elements can only be part of one cluster (or none), and *soft clustering* which allows elements to be in more than one cluster. Unless specified otherwise, clustering usually refers to *hard clustering*.

11.2 KMeans Clustering

The K-means algorithm is often likened to the task of organizing similar items into buckets. To understand how it works in an accessible way, let's envision we are sorting fruit based on sweetness and size. Each fruit will be a data point with sweetness and size as its two features.

Here's the step-by-step process:

1. **Initialization**: First, we decide on the number of buckets (clusters) we want – let's say three for our fruit example. We randomly pick three fruits and declare each as the 'representative' or centroid of a bucket.
2. **Assignment**: Each piece of fruit is then assigned to the closest bucket's representative based on its sweetness and size. 'Closest' here means the representative that has the most similar sweetness and size to the fruit we're trying to sort.
3. **Update Centroids**: After all fruits are assigned, each bucket's representative is updated to be the 'average' fruit of that bucket, considering the sweetness and size of all fruits in the bucket. This new 'average' fruit becomes the new centroid.
4. **Repeat Assignment and Update**: Steps 2 and 3 are repeated. Fruits may switch buckets if they're closer to a different centroid after the update. New centroids are calculated after the reassignments.
5. **Convergence**: This process of assignment and updating continues until things settle down – that is, the fruits no longer switch buckets (clusters), and the centroids no longer change. This is called convergence.
6. **Result**: At the end, we have our fruits sorted into buckets where each fruit is surrounded by others with similar sweetness and size. Each bucket represents a cluster.

In a more technical context, the 'sweetness' and 'size' are analogs for the features of the dataset, and the 'average' fruit represents the mean of the cluster's points in feature space.

The algorithm repeatedly performs two straightforward steps: assigning points to the nearest centroid and then updating those centroids. This simplicity, however, comes with the drawbacks mentioned, such as having to pre-specify the number of clusters and its sensitivity to the initial placement of centroids, among others.

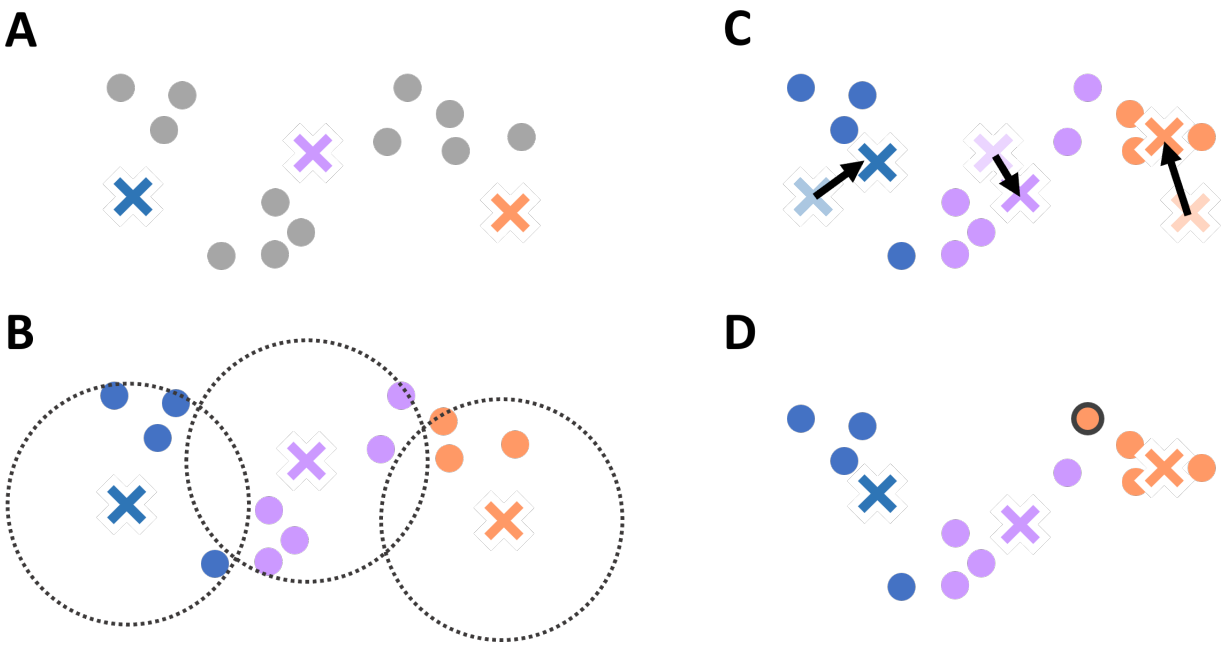


Fig. 11.3: Sketch of k-means algorithm…

Pros:

- Simple and easy to understand
- Efficient in terms of computational complexity
- Works well with isotropic clusters

Cons:

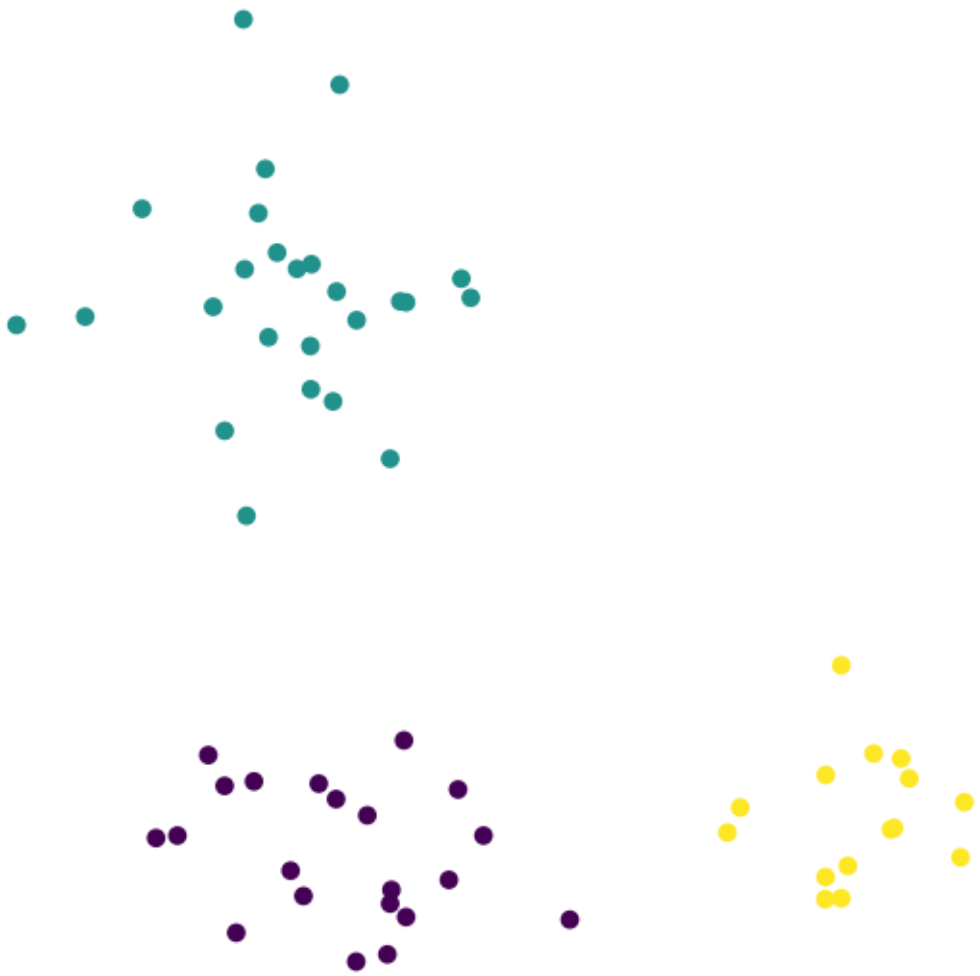
- Requires the user to specify the number of clusters
- Assumes equal-sized clusters
- Sensitive to initial conditions and may converge to a local minimum
- Does not consider outliers

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3, random_state=0).fit(data)
```

```
C:\Users\flori\anaconda3\envs\data_science\lib\site-packages\sklearn\cluster\_
↪kmeans.py:1416: FutureWarning: The default value of `n_init` will change from 10␣
↪to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\flori\anaconda3\envs\data_science\lib\site-packages\sklearn\cluster\_
↪kmeans.py:1440: UserWarning: KMeans is known to have a memory leak on Windows␣
↪with MKL, when there are less chunks than available threads. You can avoid it by␣
↪setting the environment variable OMP_NUM_THREADS=1.
  warnings.warn(
```

We visualize the resulting clustering. Each cluster is shown with a different color.

```
fig, ax = plt.subplots(figsize=(7, 7))

ax.scatter(data[:,0], data[:,1], c=kmeans.labels_)

plt.axis("off")
#plt.savefig("example_clustering_00_clusters.png", dpi=300)
plt.show()
```



11.2.1 Very often, clusters are not that easily distinguishable!

In the following code blocks we create a new synthetic dataset with five clusters, and visualize it. This is already one step more "realistic" in the sense that most commonly we do not deal with ideally distinctive clusters an shown in the examples above. Very often is is not so clear where a cluster begins and where one ends, nor how many clusters we actually have.

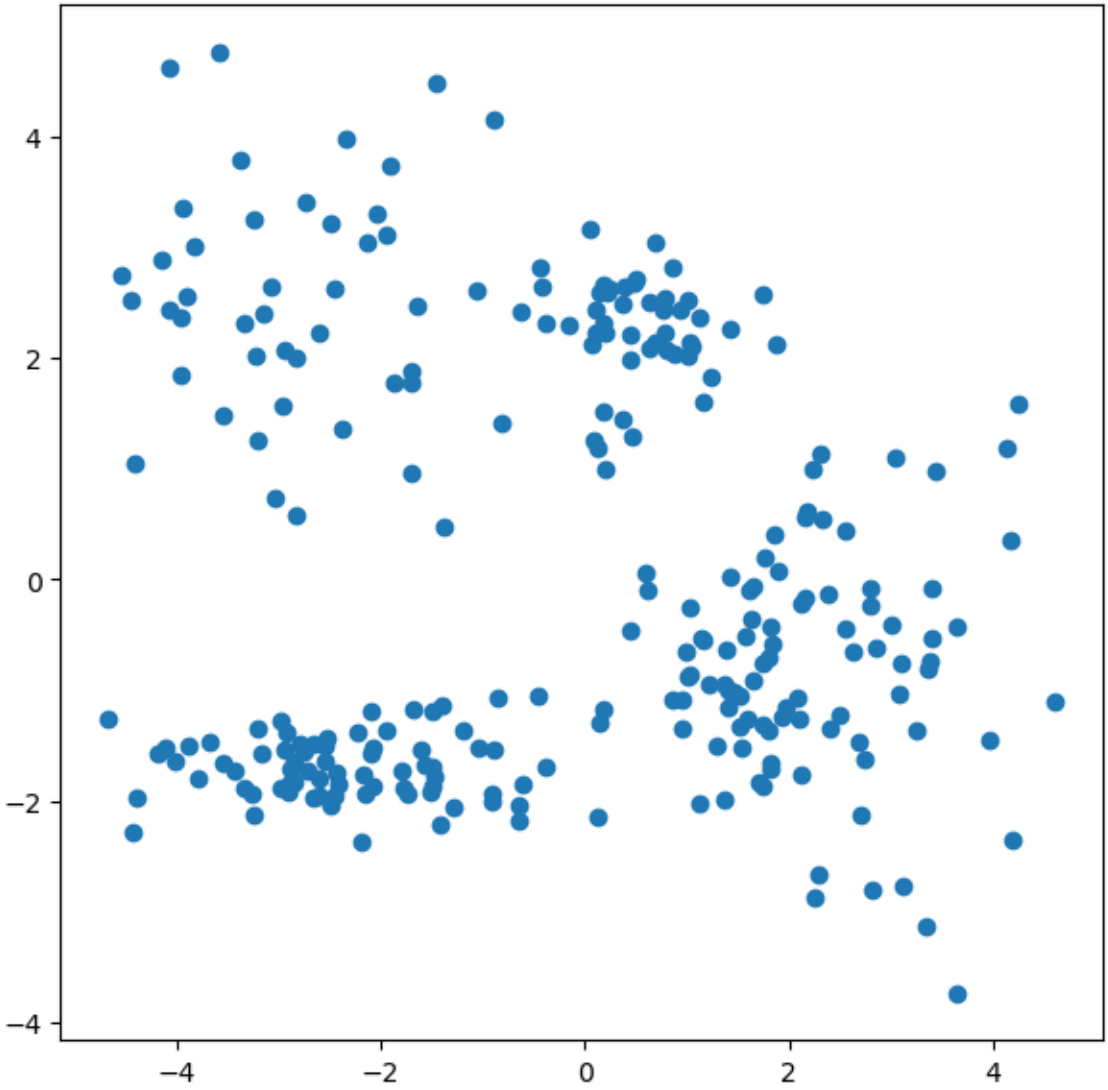
```
clusters = [(61, 1, 1.5, 2.5, -0.7),
            (37, 0.8, 0.5, 1.1, -1.2),
            (49, 1.1, 1.2, -2.6, 2.3),
            (44, 0.45, 0.4, 0.5, 2.3),
            (70, 0.9, 0.3, -2.5, -1.7)]

np.random.seed(1)
data = np.zeros((0, 2))
for (n_points, x_scale, y_scale, x_offset, y_offset) in clusters:
    xpts = np.random.randn(n_points) * x_scale + x_offset
    ypts = np.random.randn(n_points) * y_scale + y_offset
    data = np.vstack((data, np.vstack((xpts, ypts)).T))
```

```
data.shape
```

```
(261, 2)
```

```
fig, ax = plt.subplots(figsize=(7, 7))

ax.scatter(data[:,0], data[:,1])
# plt.savefig("example_clustering_01.png", dpi=300, bbox_inches="tight")
```

```
<matplotlib.collections.PathCollection at 0x19a3df3efd0>
```




```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=5, random_state=0).fit(data)
kmeans.labels_
```

```
C:\Users\flori\anaconda3\envs\data_science\lib\site-packages\sklearn\cluster\_
↪kmeans.py:1416: FutureWarning: The default value of `n_init` will change from 10␣
↪to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
```

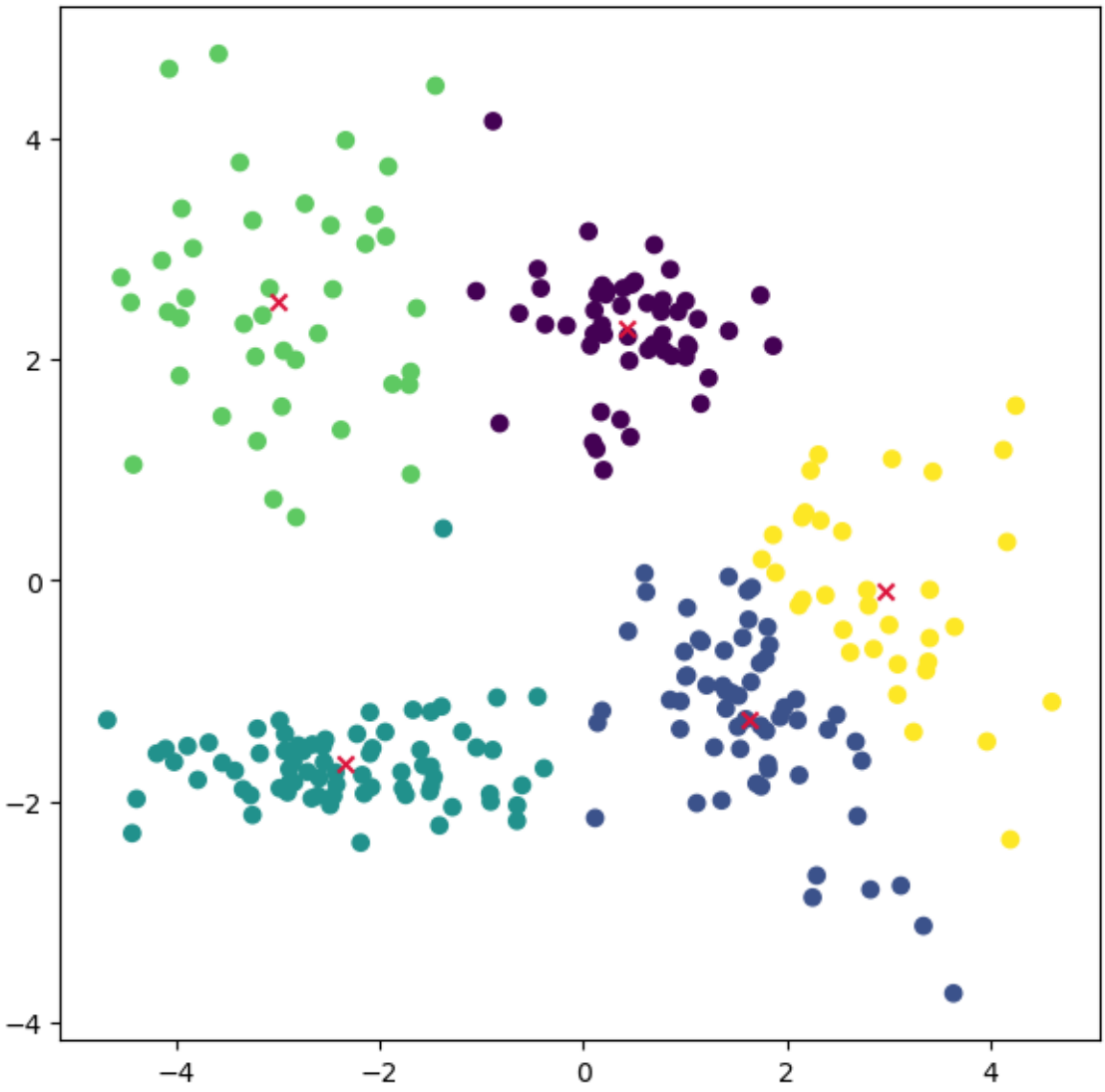
```
array([4, 4, 1, 1, 4, 0, 4, 0, 1, 1, 4, 1, 4, 4, 1, 1, 4, 1, 4, 4, 1, 4,
       4, 4, 4, 1, 4, 1, 4, 4, 1, 1, 1, 1, 1, 1, 1, 1, 4, 4, 4, 1, 4, 4,
       4, 4, 1, 4, 4, 1, 4, 4, 1, 4, 1, 4, 1, 4, 4, 4, 1, 1, 1, 2, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 2, 1, 4, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 2, 3, 3, 3, 3,
       3, 0, 3, 3, 2, 3, 3, 3, 3, 3, 0, 3, 0, 3, 0, 3, 3, 0, 3, 3, 3, 3,
       3, 3, 3, 0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

```
kmeans.cluster_centers_
```

```
array([[ 0.42175481,  2.27101038],
       [ 1.6215401 , -1.2506384 ],
       [-2.34187934, -1.65931661],
       [-2.99127157,  2.52658441],
       [ 2.96607164, -0.10142692]])
```

```
fig, ax = plt.subplots(figsize=(7, 7))

ax.scatter(data[:,0], data[:,1], c=kmeans.labels_)
ax.scatter(kmeans.cluster_centers_[:, 0],
           kmeans.cluster_centers_[:, 1], marker="x", color="crimson")

#plt.savefig("example_clustering_01_kmeans.png", dpi=300, bbox_inches="tight")
```

```
<matplotlib.collections.PathCollection at 0x19a3e274970>
```




```
kmeans.predict([[-2, 0], [4, 3]])
```

```
array([2, 4])
```

```
kmeans.cluster_centers_
```

```
array([[ 0.42175481,  2.27101038],
       [ 1.6215401 , -1.2506384 ],
       [-2.34187934, -1.65931661],
       [-2.99127157,  2.52658441],
       [ 2.96607164, -0.10142692]])
```

11.3 DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise. It's like an adventurous hike where we group together hikers based on how closely they walk together and how often they stop to rest. To explain DBSCAN in an accessible way, let's consider our hikers as data points in a park (our dataset).

Here's how DBSCAN works:

1. **Starting Point (Core Points)**: We start by picking a hiker and checking around them to see how many other hikers are within a certain distance, which we'll call 'epsilon'. If there are enough hikers (meeting a minimum number, `min_samples`) within this distance, we consider this a 'core' group, much like a core point in DBSCAN.
2. **Forming Groups (Cluster Expansion)**: From our core hiker, we extend the group by asking each new hiker in the epsilon area to invite other hikers within their own epsilon area. If these hikers also have enough companions within their reach, the group grows – this is how DBSCAN expands clusters.
3. **Connecting Groups (Density Reachability)**: Sometimes, a hiker might not have enough people within their epsilon distance to form their own core group but is close enough to be part of the existing group. These hikers act as bridges and help in joining different core groups, forming a larger cluster.
4. **Identifying Lone Hikers (Noise Detection)**: Not everyone wants to walk in groups. Hikers who don't have enough nearby companions and aren't close enough to a group are considered 'noise'. In data terms, these are outliers.
5. **Adapting to Terrain (Handling Different Densities)**: Just as groups of hikers might spread out in open fields and bunch up in narrow trails, DBSCAN tries to adapt to areas of different point densities. This can be tricky because the same 'epsilon' and `min_samples` might not work for both sparse and dense areas, which is a challenge for DBSCAN.
6. **Exploring the Entire Park (Full Dataset)**: The process continues until all hikers are either grouped or labeled as noise. Unlike K-means, we don't need to know how many groups (clusters) we want to form in advance – the hikers (data points) naturally form groups based on their proximity and the number of companions they have.

The DBSCAN algorithm's capacity to identify outliers and form clusters of arbitrary shapes makes it versatile and powerful, particularly for complex datasets where patterns aren't obvious. However, choosing the right epsilon and `min_samples` is crucial, as these parameters will greatly influence the clustering outcome.

Pros:

- Does not require the user to specify the number of clusters
- Can identify clusters with arbitrary shapes
- Robust to noise
- Can identify outliers (as "noise")

Cons:

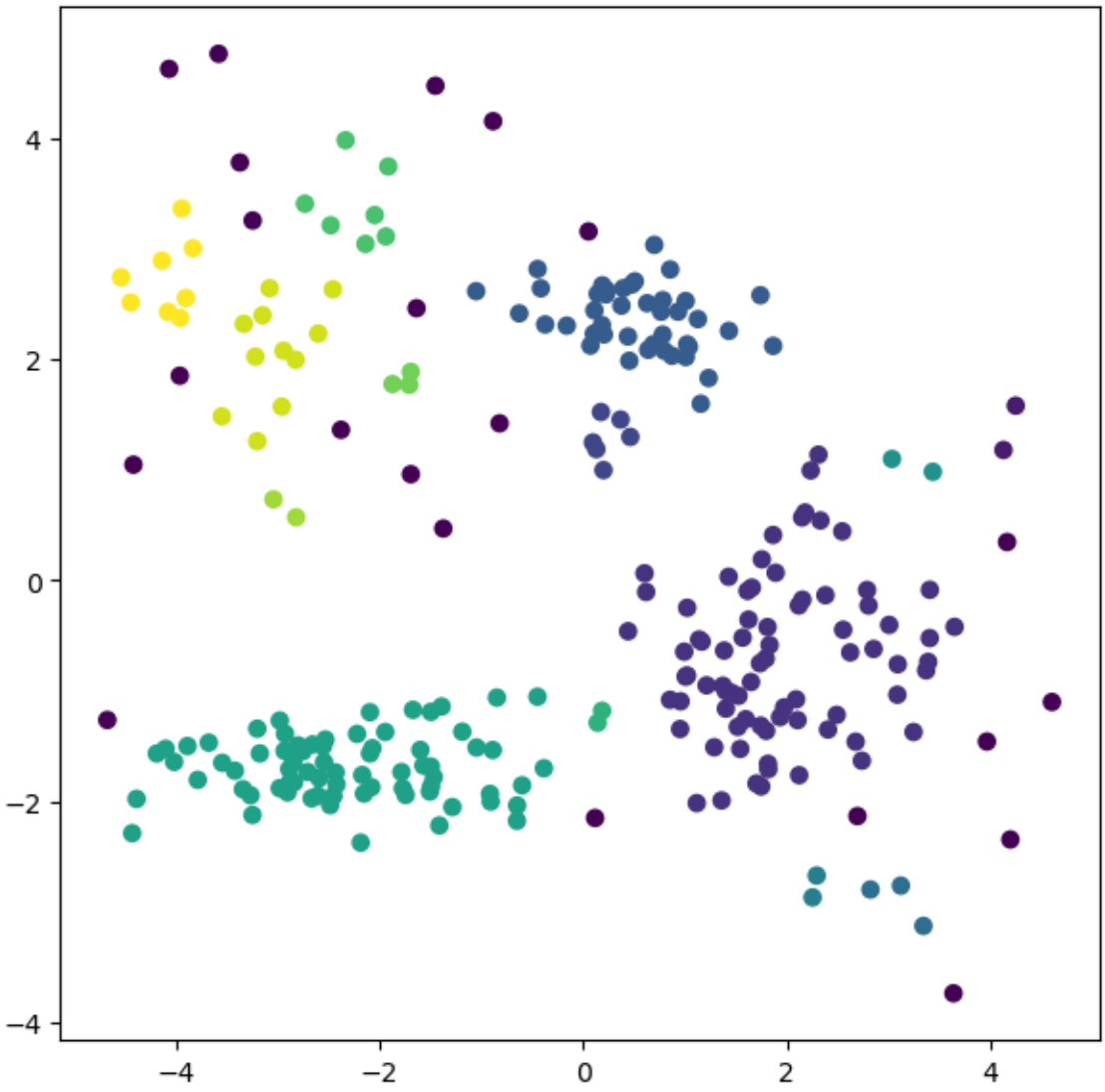
- May struggle with clusters of varying densities
- The choice of hyperparameters (epsilon and min_samples) can significantly affect the results

```
from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps=0.5, min_samples=2).fit(data)
```

```
dbscan.labels_
```

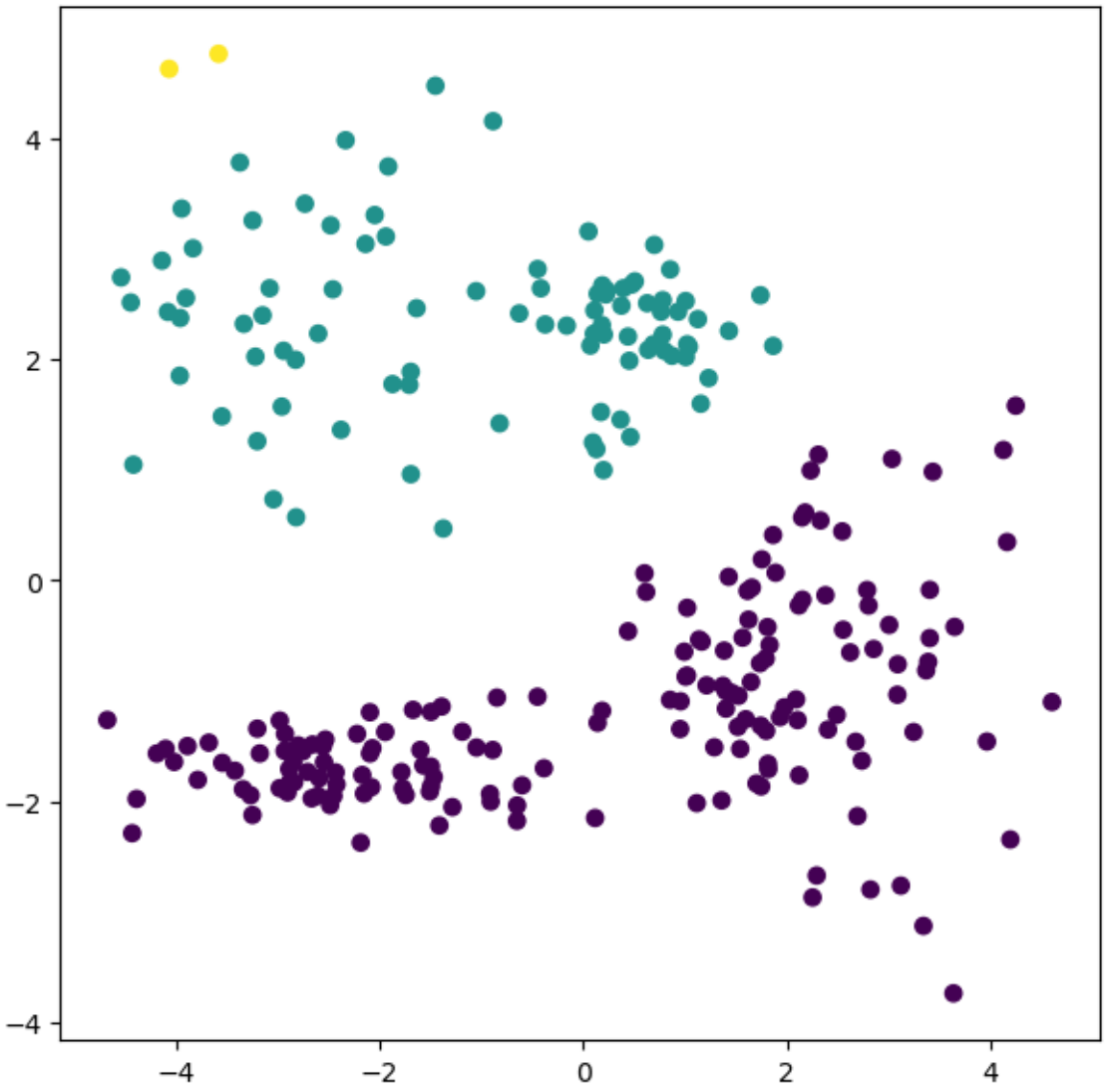
```
array([ 0,  1,  1,  1,  1,  2,  0,  3,  4,  5, -1,  1,  1,  1, -1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  6,  1,  1,  1,  1,
        1,  1,  1,  1, -1,  1,  1,  1,  1, -1,  1,  1, -1, -1,  1,  4,  1,
        1,  1,  1,  5,  1,  4,  6,  1,  1,  1,  1,  1,  7,  1,  1,  1,  8,
        1,  1,  1, -1,  1,  1,  1,  1,  8,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  7,  1,  1,  1,  1,  1,  1,  1,  9,  9, 10, -1,
       11, 10, 12,  7, -1, 13, -1, -1, 13,  3, 12, 13, -1, -1, -1, -1,  9,
       12,  3, 12, -1,  9, -1,  9, 13,  2, 12, 11, -1, 12,  9, -1, 10,  3,
        9, 13, 13, 12, 13, -1, 12, 12, 12, 13, 12,  2,  2,  3,  3,  3,  3,
        3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,
        3,  2,  3,  3,  3,  3,  2,  3,  3,  3,  3,  3,  3,  3,  3, -1,  3,
        3,  3,  3,  3,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,
        7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,
        7,  7, -1,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,
        7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,
        7,  7,  7,  7,  7,  7], dtype=int64)
```

```
fig, ax = plt.subplots(figsize=(7, 7))

ax.scatter(data[:,0], data[:,1], c=dbscan.labels_)

plt.savefig("example_clustering_01_dbscan_eps_05.png", dpi=300,
            bbox_inches="tight")
```




```
dbscan = DBSCAN(eps=1, min_samples=2).fit(data)

fig, ax = plt.subplots(figsize=(7, 7))

ax.scatter(data[:,0], data[:,1], c=dbscan.labels_)

# plt.savefig("example_clustering_01_dbscan_eps_1.png", dpi=300, bbox_inches="tight")
```

```
<matplotlib.collections.PathCollection at 0x19a3e483580>
```



11.4 Gaussian mixture models

Gaussian Mixture Models (GMM) can be thought of as a sophisticated extension of the K-means clustering technique, with added flexibility. If we were to visualize data clustering as a process of sorting different types of coins based on their size and weight, K-means would sort them into predefined buckets, while GMM would be like having scales and calipers that tell us not just which bucket a coin most likely belongs to, but also provide a measure of our certainty about it.

Here's a breakdown of how GMM works:

1. **Assumptions**: GMM starts with the assumption that our data points are generated from a mixture of several Gaussian distributions. Each Gaussian, also known as a normal distribution, represents a cluster. It's like assuming that coins come from different countries, each with its unique size and weight characteristics.
2. **Expectation-Maximization (EM) Steps**:
    - **Expectation (E-step)**: Here, the algorithm assesses each data point and estimates the probabilities of it belonging to each of the Gaussian distributions (clusters). In our coin analogy, this is like guessing the origin

of a coin based on its size and weight, but instead of being certain, we assign probabilities to the likelihood of it coming from each country.

- **Maximization (M-step)**: Based on these probabilities, GMM updates the parameters of the Gaussian distributions—namely, the means (which are like the centroid in K-means), the variances (which tell us how spread out each cluster is), and the mixture weights (which tell us how large or small each cluster is compared to others). This is akin to adjusting our scale and calipers based on all the coins we've measured to better fit the actual data.

3. **Iterative Process**: These two steps are repeated iteratively, with each pass refining the parameters and improving the model's accuracy in representing the underlying clusters.
4. **Soft Clustering**: Unlike K-means, which assigns each point to a single cluster, GMM provides a probability distribution over the clusters for each point, indicating its degree of membership in every cluster. This soft assignment is particularly useful when we're not sure about the boundaries of clusters, much like a coin that's in between typical sizes and weights for known designs.
5. **Final Model**: After several iterations, when the changes in the parameters become negligible, the algorithm has hopefully converged to the best fit for our data, and we have a final model that tells us not only where the clusters are, but also how certain we can be about each data point's membership.

Pros:

- More flexible than K-means and can model clusters with different shapes, sizes, and orientations
- Provides a soft clustering, assigning probabilities of each point belonging to each cluster

Cons:

- Requires the user to specify the number of clusters
- Computationally more expensive than K-means
- Requires a high enough number of datapoints to fit the gaussians

```
from sklearn.mixture import GaussianMixture

gm = GaussianMixture(n_components=5, random_state=0).fit(data)
```

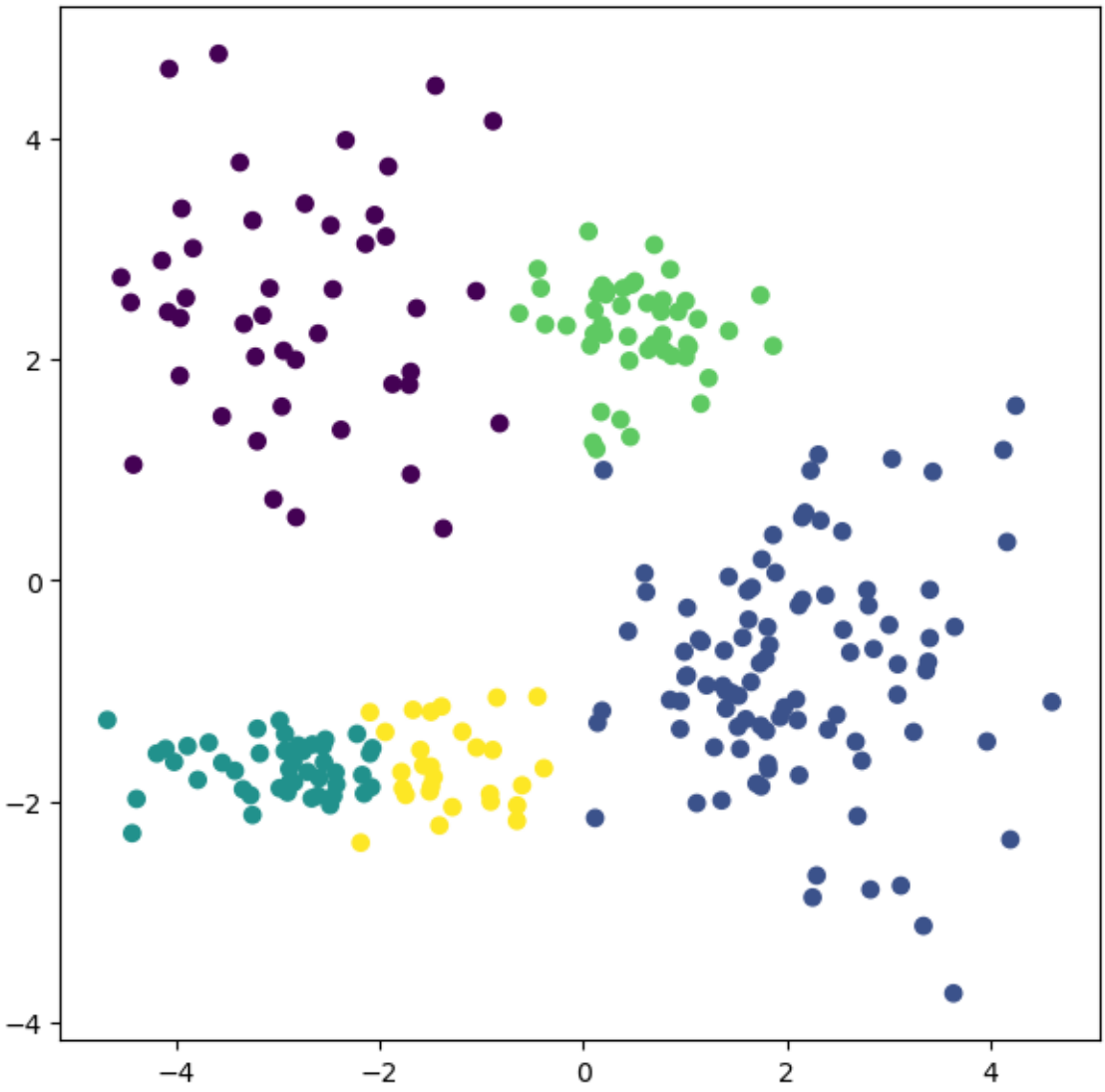
```
gm.means_
```

```
array([[-2.80121071,  2.49623595],
       [ 2.08237568, -0.79412545],
       [-2.96321278, -1.69907782],
       [ 0.49887671,  2.26838731],
       [-1.43864991, -1.67923165]])
```

```
labels = gm.predict(data)
```

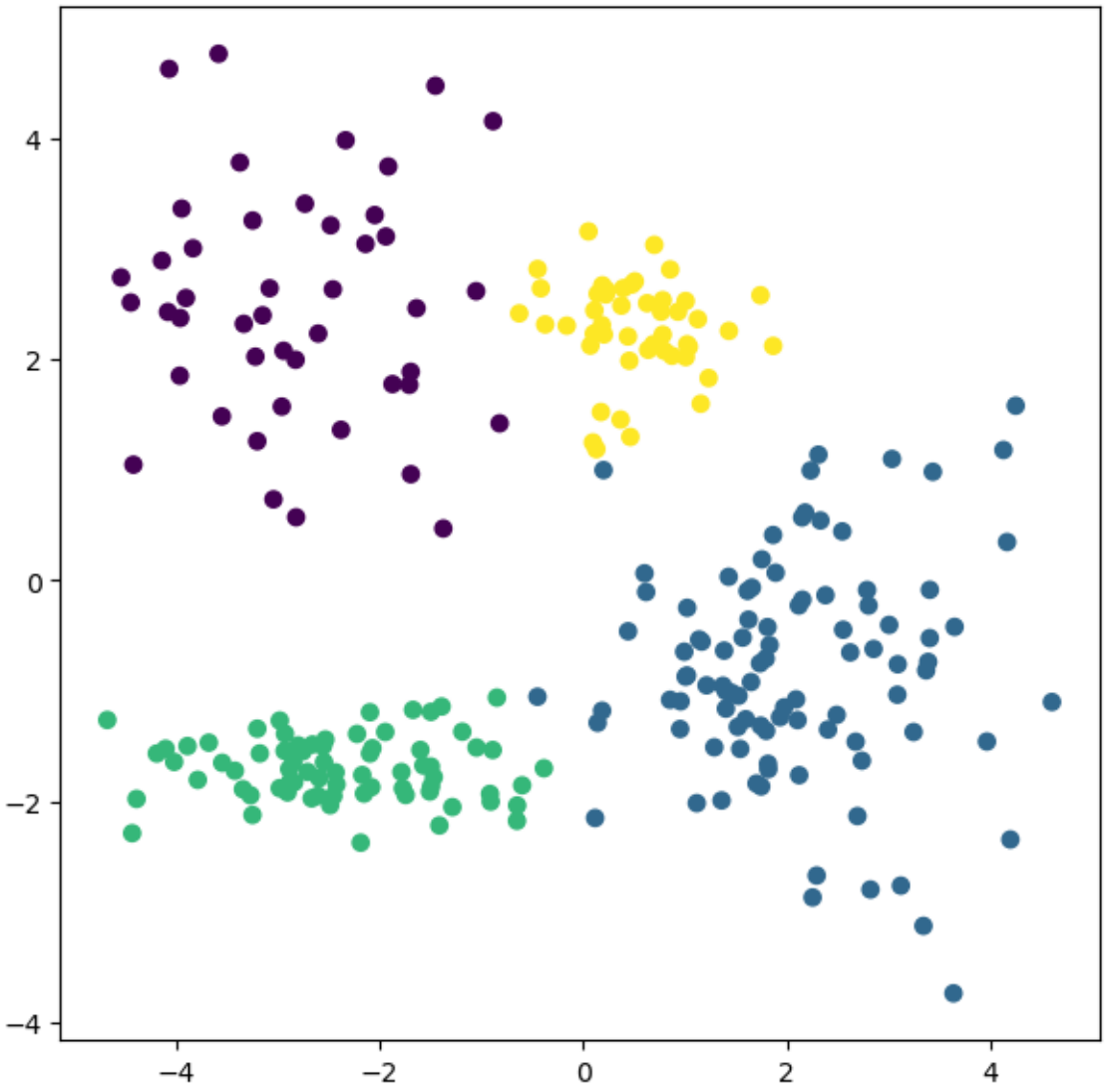
```
fig, ax = plt.subplots(figsize=(7, 7))

ax.scatter(data[:,0], data[:,1], c=labels)
# plt.savefig("example_clustering_01_gaussian_mixture.png", dpi=300, bbox_inches=
↪"tight")
```

```
<matplotlib.collections.PathCollection at 0x19a3df7cee0>
```




```
gm = GaussianMixture(n_components=4, random_state=0).fit(data)
fig, ax = plt.subplots(figsize=(7, 7))

ax.scatter(data[:,0], data[:,1], c=gm.predict(data))
# plt.savefig("example_clustering_01_gaussian_mixture_c4.png", dpi=300, bbox_inches=
↪"tight")
```

```
<matplotlib.collections.PathCollection at 0x19a3ed3eca0>
```

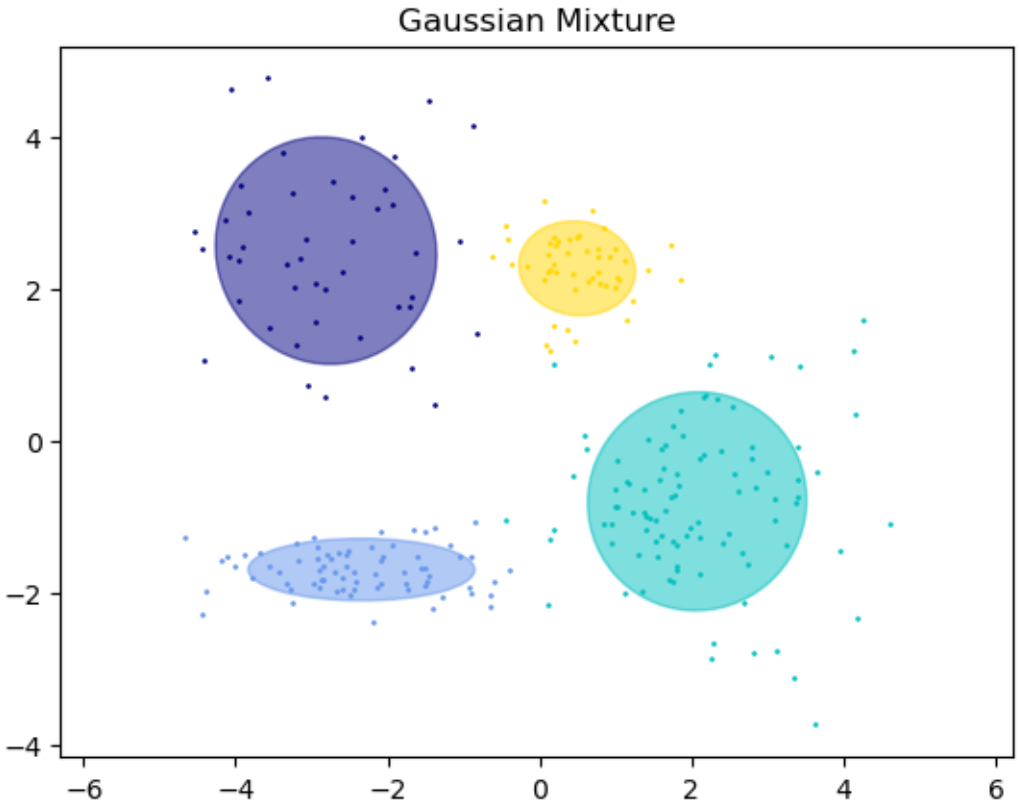



```
# https://scikit-learn.org/stable/auto_examples/mixture/plot_gmm.html
import itertools
from scipy import linalg
import matplotlib as mpl

color_iter = itertools.cycle(["navy", "c", "cornflowerblue", "gold", "darkorange"])

def plot_results(X, Y_, means, covariances, index, title):
    splot = plt.subplot(1, 1, 1 + index)
    for i, (mean, covar, color) in enumerate(zip(means, covariances, color_iter)):
        v, w = linalg.eigh(covar)
        v = 2.0 * np.sqrt(2.0) * np.sqrt(v)
        u = w[0] / linalg.norm(w[0])
        # as the DP will not use every component it has access to
        # unless it needs it, we shouldn't plot the redundant
        # components.
        if not np.any(Y_ == i):
            continue
```

(continues on next page)

(continued from previous page)

```
        plt.scatter(X[Y_ == i, 0], X[Y_ == i, 1], 0.8, color=color)

        # Plot an ellipse to show the Gaussian component
        angle = np.arctan(u[1] / u[0])
        angle = 180.0 * angle / np.pi  # convert to degrees
        ell = mpl.patches.Ellipse(mean, v[0], v[1],
                                  angle=180.0 + angle, color=color)
        ell.set_clip_box(splot.bbox)
        ell.set_alpha(0.5)
        splot.add_artist(ell)

    #plt.xlim(-9.0, 5.0)
    #plt.ylim(-3.0, 6.0)
    plt.axis("equal")
    plt.title(title)

plot_results(data, gm.predict(data),
             gm.means_, gm.covariances_, 0, "Gaussian Mixture")
```



11.5 Hierarchical clustering (Ward)

Hierarchical clustering, particularly the Ward method, operates much like a family tree of data points, showing the relationships between different clusters from the ground up. To visualize hierarchical clustering in an everyday context, imagine a scenario where we're trying to organize a large collection of books on a set of shelves.

Here's the process outlined in a more accessible manner:

1. **Starting with Every Book (Initialization)**: Initially, each book is considered its own 'cluster'. This is the bottom-up approach where every single item starts in its own group.
2. **Creating Shelves (Agglomeration)**: We begin to organize the books by placing the most similar ones together on a shelf. In the context of hierarchical clustering, similarity is usually based on the distance between data points. The Ward method specifically looks to minimize the variance within a cluster, which is like trying to put books of similar sizes or genres together to make the shelf look neat (minimize variance).
3. **Building Sections (Building the Hierarchy)**: Each time we place a pair of books together, we're building a small section of our shelf. As we continue this process, we start to see not just individual books or pairs but groups and subgroups forming, each representing a branch in our tree of book categories.
4. **Forming a Library (Creating the Dendrogram)**: We don't just stop once we've created a few shelves; we continue until all books are grouped in a way that shows their relationships, from the individual books up to the entire genre sections. In clustering terms, this is akin to building a dendrogram—a tree-like diagram that records the sequences of merges or splits.
5. **Deciding on Genres (Determining the Number of Clusters)**: The beauty of hierarchical clustering is that we don't need to decide how many genres (clusters) we want at the start. Once our dendrogram is complete, we can cut it at the level that makes sense to us, which might be broad genres or more specific sub-genres, depending on our needs.

The pros of this method are that it doesn't require us to predefine the number of clusters, and it gives us a visual representation of the data's hierarchy, which can be incredibly insightful. However, this method can be demanding in terms of computation, especially with large datasets—it's like trying to organize a library by hand. Furthermore, because it uses an agglomerative approach, decisions made early on cannot be undone without starting over, which can sometimes lead to less optimal clustering. Also, while we don't need to specify the number of clusters upfront, we still need a criterion to decide where to 'cut' the dendrogram to define our clusters.

Pros:

- Does not require the user to specify the number of clusters beforehand (can be chosen by analyzing the dendrogram)
- Provides a hierarchical representation of the data, which can be useful for understanding the relationships between clusters

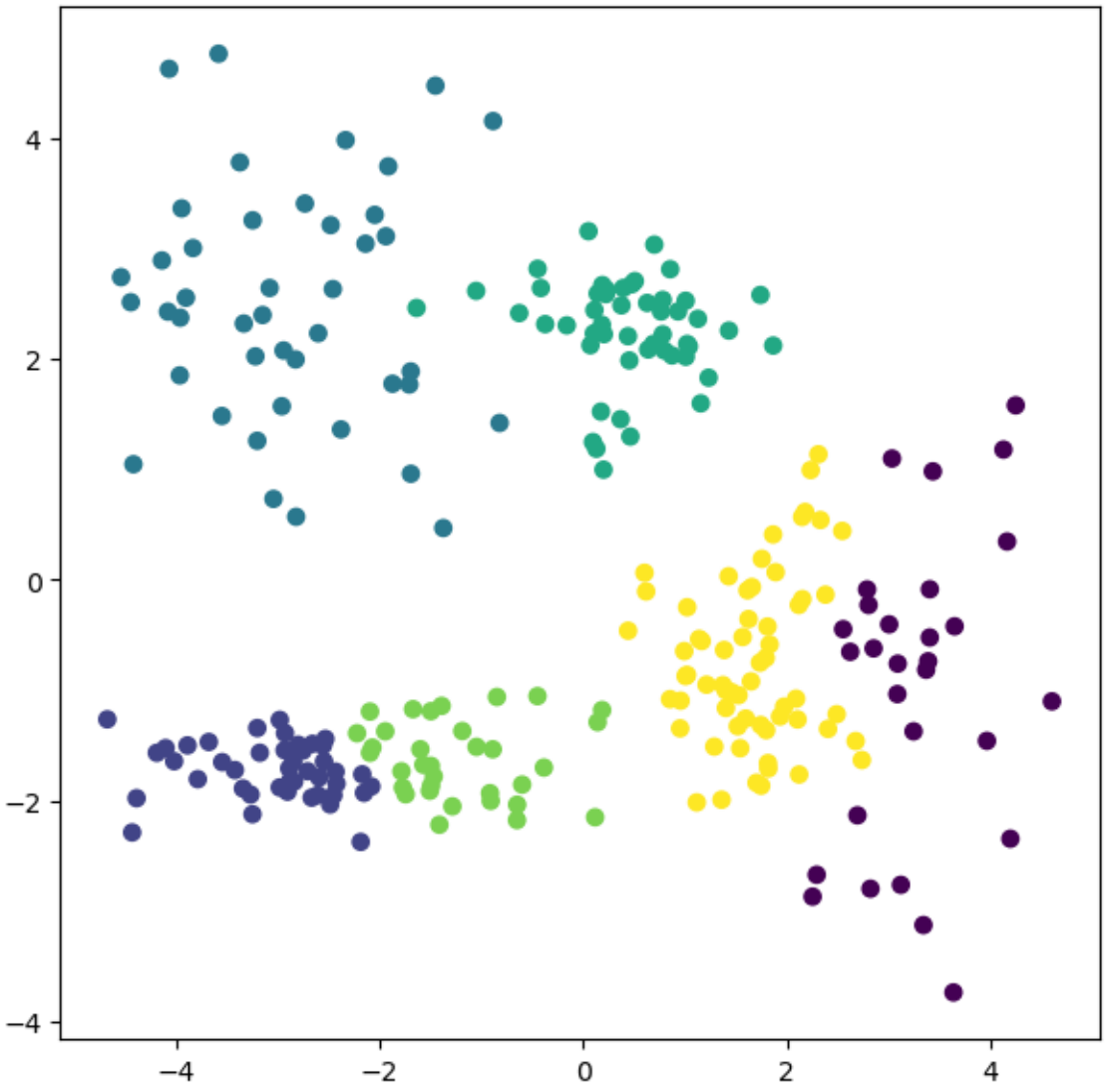
Cons:

- Computationally more expensive than K-means and DBSCAN, especially for large datasets
- Assumes that the distance between clusters can be represented by the within-cluster variance
- Requires number of clusters OR distance threshold

```
from sklearn.cluster import AgglomerativeClustering

ward = AgglomerativeClustering(linkage="ward",
                               n_clusters=None,
                               distance_threshold=10.0).fit(data)
```

```
fig, ax = plt.subplots(figsize=(7, 7))

ax.scatter(data[:,0], data[:,1], c=ward.labels_)
# plt.savefig("example_clustering_01_hierarchical_ward.png", dpi=300, bbox_inches=
↪"tight")
```

```
<matplotlib.collections.PathCollection at 0x19a3eec4910>
```




```
from sklearn.cluster import AgglomerativeClustering

ward = AgglomerativeClustering(linkage="ward",
                               n_clusters=None,
                               distance_threshold=2.0).fit(data)
```

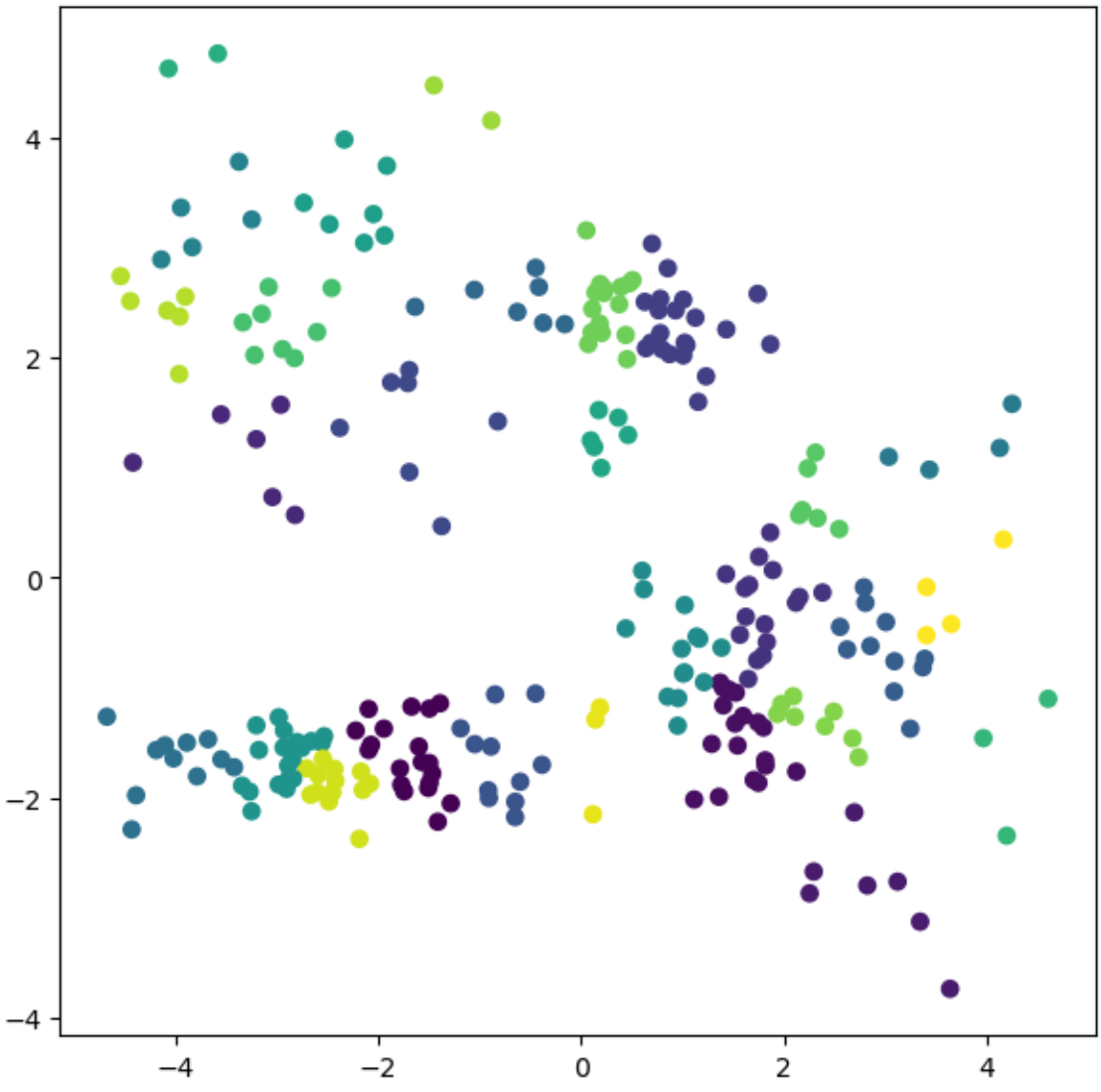
```
gm = GaussianMixture(n_components=4, random_state=0).fit(data)
fig, ax = plt.subplots(figsize=(7, 7))
```

(continues on next page)

(continued from previous page)

```
ax.scatter(data[:,0], data[:,1], c=ward.labels_)
#plt.savefig("example_clustering_01_hierarchical_ward_dist2.png", dpi=300, bbox_
↪inches="tight")
```

```
<matplotlib.collections.PathCollection at 0x19a3eccde80>
```



11.6 Comparison and Conclusion

In this section, we have explored four different clustering algorithms: K-means, DBSCAN, Gaussian Mixture Model (GMM) and hierarchical clustering. Each of these algorithms has its own advantages and disadvantages.

K-means is a simple and easy-to-understand algorithm that works well with isotropic clusters. However, it requires the user to specify the number of clusters beforehand and assumes equal-sized clusters. K-means is also sensitive to initial conditions and may converge to a local minimum.

DBSCAN does not require the user to specify the number of clusters and can identify clusters with arbitrary shapes. It is also robust to noise. However, DBSCAN may struggle with clusters of varying densities, and the choice of hyperparameters (epsilon and min_samples) can significantly affect the results.

The **Gaussian Mixture Model** (GMM) is a generative probabilistic model that assumes each cluster is generated from a Gaussian distribution. GMM is more flexible than K-means and can model clusters with different shapes, sizes, and orientations. However, GMM requires the user to specify the number of clusters and is computationally more expensive.

Hierarchical Clustering builds a hierarchy of clusters, either agglomerative (bottom-up) or divisive (top-down), with the Ward method being an example that minimizes the within-cluster variance. This method doesn't require specifying the number of clusters, offering a visual dendrogram to aid in cluster selection, though it is computationally expensive for large datasets.

In conclusion, the choice of clustering algorithm depends on the data, the problem, and the desired characteristics of the clustering solution. It is often helpful to try multiple algorithms and compare their results to select the best method for a particular problem.

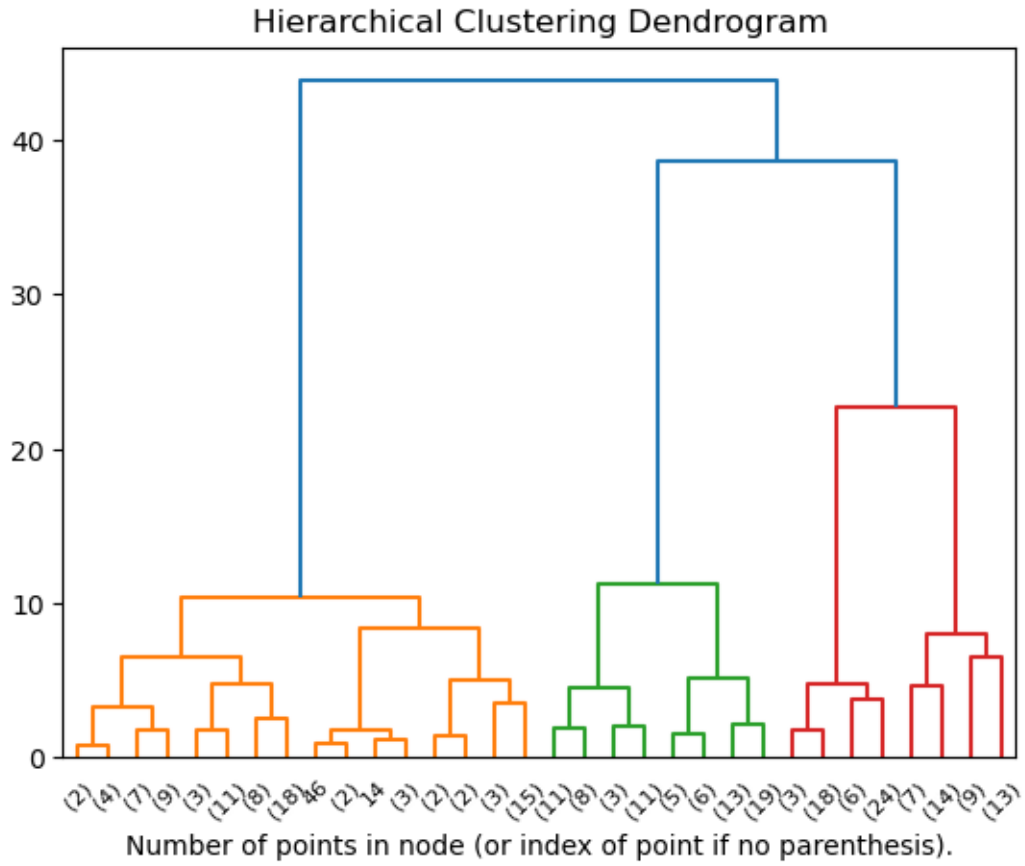
```
from scipy.cluster.hierarchy import dendrogram
from sklearn.datasets import load_iris
from sklearn.cluster import AgglomerativeClustering


def plot_dendrogram(model, **kwargs):
    # Create linkage matrix and then plot the dendrogram

    # create the counts of samples under each node
    counts = np.zeros(model.children_.shape[0])
    n_samples = len(model.labels_)
    for i, merge in enumerate(model.children_):
        current_count = 0
        for child_idx in merge:
            if child_idx < n_samples:
                current_count += 1  # leaf node
            else:
                current_count += counts[child_idx - n_samples]
        counts[i] = current_count

    linkage_matrix = np.column_stack(
        [model.children_, model.distances_, counts]
    ).astype(float)

    # Plot the corresponding dendrogram
    dendrogram(linkage_matrix, **kwargs)
```

```
plt.title("Hierarchical Clustering Dendrogram")
# plot the top three levels of the dendrogram
plot_dendrogram(ward, truncate_mode="level", p=4)
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("Number of points in node (or index of point if no parenthesis).")
plt.show()
```

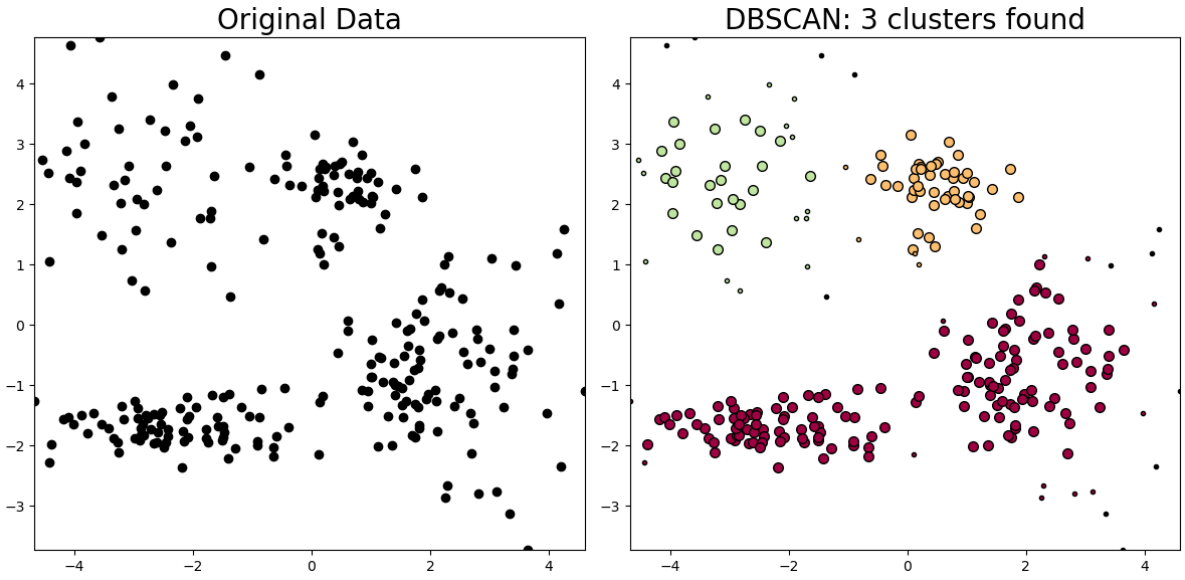



```
max_distance = 1
db = DBSCAN(eps=max_distance, min_samples=10).fit(data)
# Extract a mask of core cluster members
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
# Extract labels (-1 is used for outliers)
labels = db.labels_
n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
unique_labels = set(labels)

# Plot up the results!
min_x = np.min(data[:, 0])
max_x = np.max(data[:, 0])
min_y = np.min(data[:, 1])
max_y = np.max(data[:, 1])

fig = plt.figure(figsize=(12,6))
plt.subplot(121)
plt.plot(data[:,0], data[:,1], 'ko')
plt.xlim(min_x, max_x)
plt.ylim(min_y, max_y)
```

(continues on next page)

(continued from previous page)

```
plt.title('Original Data', fontsize = 20)

plt.subplot(122)
# The following is just a fancy way of plotting core, edge and outliers
# Credit to: http://scikit-learn.org/stable/auto_examples/cluster/plot_dbscan.html
 ↪#sphx-glr-auto-examples-cluster-plot-dbscan-py
colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = data[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=7)

    xy = data[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=3)
plt.xlim(min_x, max_x)
plt.ylim(min_y, max_y)
plt.title('DBSCAN: %d clusters found' % n_clusters, fontsize = 20)
fig.tight_layout()
plt.subplots_adjust(left=0.03, right=0.98, top=0.9, bottom=0.05)
```



Part V

Data Modeling

CHAPTER

TWELVE

DIMENSIONALITY REDUCTION

```
# Material yet to be cleaned and prepared...
```

CHAPTER

THIRTEEN

MACHINE LEARNING

```
# Material yet to be cleaned and prepared...
```

Part VI

Working with text data

CHAPTER

FOURTEEN

INTRODUCTION TO WORKING WITH TEXT DATA

Data Science is a versatile field that integrates techniques from different domains to extract meaningful information from data. Often, the data we work with is numerical, perhaps tabular and structured. However, a significant portion of the data available today is unstructured and in the form of text. This chapter focuses on why and how we work with such text data.

Text data can be found in various forms and domains. It's ubiquitous in today's digital world and holds immense value when it comes to generating insights and making informed decisions. In general, we can categorize the applications of text data in data science into two broad categories:

14.1 1. Data and Structures as Strings

String data forms a large part of the information we handle in our day-to-day lives and in scientific research. It can include:

- **File and Folder names**: These are often encoded as text and carry essential information about the content they represent. Proper management and manipulation of these names can significantly enhance our productivity and efficiency.
- **Categorical Data**: Many real-world datasets contain categorical features such as Pokémon type, credit card usage, eye color, etc. These categories are typically represented as text data.
- **Names and Designations**: In various fields like social sciences, human resources, and even technical domains, names and designations (for people, products, companies, etc.) are prevalent and crucial.

14.2 2. Human Language in Text Form

Much of our communication today happens via written text. Such data is rich and diverse, and its analysis can provide a wealth of insights. It includes:

- **Social Media, Emails, Chats**: A substantial amount of information is exchanged through social media posts, emails, and chat conversations. Analyzing this data can help understand user behavior, sentiments, trends, and more.
- **News and Press Releases, Print Media**: These sources provide a constant stream of new information about the world. Text analysis on such data can reveal patterns, detect events, and even predict future occurrences.
- **The Internet**: The Internet is predominantly text, from information-dense Wikipedia articles to YouTube video tags and titles. Web scraping and text mining can help extract valuable insights from this vast reservoir of data.
- **Protocols and Transcripts**: Transcriptions of meetings, court proceedings, and more provide a formal record of events. Analyzing these can reveal decision patterns, biases, and other organizational dynamics.

- **Legislative Texts, Manuals**: Legal texts and manuals contain structured and vital information. Text analysis can aid in understanding, summarizing, and even automating parts of these documents.

In the following sections of this chapter, we will delve into how we can effectively manage and process these text data to extract valuable insights. This includes techniques from simple string manipulation to advanced natural language processing. Understanding text data and how to work with it is an invaluable skill in the modern data science landscape.

But let us start with the most fundamental basics: string handling! Here a simple short text to work with (the text was generated by ChatGPT with its unique sense of humor…):

```
my_text = """Once upon a time, in a far-off land of data, lived a strange yet
curious character known as Stringman. Stringman wasn't an ordinary
resident of this land. He had a unique ability to transform himself
into different forms.\n
One day, String-man decided to visit the 'Tuple Twins'. As he was
journeying, he had to pass through the eerie 'List Forest'.
Suddenly, a wild 'IndexError' appeared. It was well-known that
'IndexErrors' were not fond of anyone from the 'String' family.
String man, being clever, quickly transformed into 'string-man'
and tricked the 'IndexError' saying, "You must be mistaken, I am
not a Stringman, but a mere hyphenated man."\n
Fooled by his disguise, the 'IndexError' let him pass. String-man
then continued his journey, relishing his victory over the 'IndexError'
and looking forward to meeting the 'Tuple Twins'.
"""

print(my_text)
```

```
Once upon a time, in a far-off land of data, lived a strange yet
curious character known as Stringman. Stringman wasn't an ordinary
resident of this land. He had a unique ability to transform himself
into different forms.

One day, String-man decided to visit the 'Tuple Twins'. As he was
journeying, he had to pass through the eerie 'List Forest'.
Suddenly, a wild 'IndexError' appeared. It was well-known that
'IndexErrors' were not fond of anyone from the 'String' family.
String man, being clever, quickly transformed into 'string-man'
and tricked the 'IndexError' saying, "You must be mistaken, I am
not a Stringman, but a mere hyphenated man."

Fooled by his disguise, the 'IndexError' let him pass. String-man
then continued his journey, relishing his victory over the 'IndexError'
and looking forward to meeting the 'Tuple Twins'.
```

14.3 Basic Python string handling methods

The Python string data type already comes with many basic string handling methods. Here, more as a repetition, some of the most common ones:

14.3.1 Python String Methods

Method	Description
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
index()	Searches the string for a specified value and returns the position of where it was found
islower()	Returns True if all characters in the string are lower case
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
lower()	Converts a string into lower case
strip()	Returns a trimmed version of the string
lstrip()	Returns a left trim version of the string
replace()	Returns a string where a specified value is replaced with a specified value
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
upper()	Converts a string into upper case

This is not a full introduction to basic string handling with Python. For more information, you can easily find plenty of material online, for instance the w3schools on Python strings. Here, we will simply go through a few common examples as a refresher.

```
my_text.lower()
```

```
'once upon a time, in a far-off land of data, lived a strange yet\ncurious␣
↪character known as stringman. stringman wasn\'t an ordinary\nresident of this␣
↪land. he had a unique ability to transform himself\ninto different forms.\n\none␣
↪day, string-man decided to visit the \'tuple twins\'. as he was\njourneying, he␣
↪had to pass through the eerie \'list forest\'.\nsuddenly, a wild \'indexerror\'␣
↪appeared. it was well-known that\n\'indexerrors\' were not fond of anyone from␣
↪the \'string\' family. \nstring man, being clever, quickly transformed into \
↪'string-man\'\nand tricked the \'indexerror\' saying, "you must be mistaken, i␣
↪am\nnot a stringman, but a mere hyphenated man."\n\nfooled by his disguise, the \
↪'indexerror\' let him pass. string-man\nthen continued his journey, relishing␣
↪his victory over the \'indexerror\'\nand looking forward to meeting the \'tuple␣
↪twins\'.\n'
```

One of the first things to note when working with text data in Python is the presence of special character sequences like `\'` and `\n`. You might have noticed these appearing in our story about Stringman when we printed the lowercased text using `my_text.lower()`. These are known as escape sequences.

The `\'` sequence is used to include literal single quotes (`'`) in our text string. This is necessary because Python interprets single quotes as marking the start or end of a string. Therefore, to use a single quote as part of the string itself (for

instance, in contractions like “don’t” or to denote possession as in “Python’s”), we ‘escape’ it using a backslash (`\`) before the quote.

On the other hand, `\n` is a newline character that is used to start a new line. Python interprets this sequence as a single character that moves the cursor to the next line. This is why we see the text broken into multiple lines when we print the content of `my_text`.

Understanding and being able to handle such escape sequences is an essential part of working with text data, as they can affect the processing and analysis of the text.

Modify strings

A very common method used to modify strings in Python is via the `replace()` methods. This is well suited to replace specific characters or sequences of characters.

```
s = "We don't always like all characters and sequences we have."
s.replace("i", "!").replace("e", "3")
```

```
"W3 don't always l!k3 all charact3rs and s3qu3nc3s w3 hav3."
```

```
print("abc".replace("ab", "cc").replace("c", "x"))
```

a) ccx
b) ab
c) xxx
d) abx

14.3.2 Mini Quiz!

Make a guess: What would the following code return?

Count words

Counting words is a very common task. It is even central to many data science methods. Python string methods include methods like `count()` which, like their name says, can do some of this counting for us. But we will quickly see, that often this is not good enough.

```
my_text.count("no")
```

```
4
```

Careful. This will not count all words “no”, but every single occurrence of the letter sequence `no` in our string, which would also include words like “none” or “nothing” etc.

```
my_text.count("stringman"), my_text.count("string-man")
```

```
(0, 1)
```

```
my_text.lower().count("stringman"), my_text.lower().count("string-man")
```

```
(3, 3)
```

14.3.3 Tokenize

```
words = my_text.lower().split(" ")  # still many wrong words in there
print(words[:40])
```

```
['once', 'upon', 'a', 'time,', 'in', 'a', 'far-off', 'land', 'of', 'data,', 'lived
 ↪', 'a', 'strange', 'yet\ncurious', 'character', 'known', 'as', 'stringman.',
 ↪'stringman', "wasn't", 'an', 'ordinary\nresident', 'of', 'this', 'land.', 'he',
 ↪'had', 'a', 'unique', 'ability', 'to', 'transform', 'himself\ninto', 'different',
 ↪ 'forms.\n\none', 'day,', 'string-man', 'decided', 'to', 'visit']
```

```
words = my_text.lower().replace(".", "").replace(",", "").replace("\n", " ").split("
 ↪")
print(words[:40])
```

```
['once', 'upon', 'a', 'time', 'in', 'a', 'far-off', 'land', 'of', 'data', 'lived',
 ↪'a', 'strange', 'yet', 'curious', 'character', 'known', 'as', 'stringman',
 ↪'stringman', "wasn't", 'an', 'ordinary', 'resident', 'of', 'this', 'land', 'he',
 ↪'had', 'a', 'unique', 'ability', 'to', 'transform', 'himself', 'into', 'different
 ↪', 'forms', '', 'one']
```

14.3.4 Very common problem: different variations of a word

```
words.count("stringman"), words.count("string-man")
```

```
(3, 2)
```

```
variations = ["string-man", "stringman", "string man"]

# Solution 1
translations = {"string-man": "stringman",
                "string man": "stringman",
                "stringman": "stringman"}

print([translations[w] for w in variations])
```

```
['stringman', 'stringman', 'stringman']
```

```
# Solution 2 - real bad

print([w.replace("-", "") for w in variations])
```

```
['stringman', 'stringman', 'string man']
```

```
my_text2 = "Das Wichtigste steht nicht in meinem Tweet sondern in \
den Hashtags #openscience #openaccess #research"

words = my_text2.lower().split(" ")
words = [w.strip(".,!? ") for w in words]
words
print(words)
```

```
['das', 'wichtigste', 'steht', 'nicht', 'in', 'meinem', 'tweet', 'sondern', 'in',
 ↪'den', 'hashtags', '#openscience', '#openaccess', '#research']
```

```
for w in words:
    if w.startswith("#"):
        print(f"Hashtag: {w}")
```

```
Hashtag: #openscience
Hashtag: #openaccess
Hashtag: #research
```

14.4 Regular Expressions (Regex)

Regular Expressions, often shortened to "regex," are a powerful tool for working with text data. They're a sequence of characters forming a search pattern, primarily used for pattern matching with strings or string manipulation. In the world of Data Science, regular expressions find widespread use in text processing tasks.

14.4.1 What are Regular Expressions?

At their core, regular expressions are a means to describe patterns within strings. They offer a flexible and concise way to identify strings of text such as particular words, patterns of characters, or a combination of these. This can be as simple as searching for a specific word or as complicated as extracting all email addresses from a text.

14.4.2 Uses of Regular Expressions

Regular expressions are used for several text processing tasks:

- **Validation**: They can check if the input data follows a certain format, such as an email address or a telephone number.
- **Search**: You can use regex to locate specific strings or substrings within a larger piece of text.
- **Substitution**: They can be used to replace certain patterns in a string.
- **Splitting**: Regular expressions can define the delimiter to split a larger string into a list of smaller substrings.
- **Data Extraction**: They are often used to scrape web data, where specific patterns need to be extracted from HTML code.

14.4.3 Pros and Cons of Regular Expressions

Pros

- **Versatile**: Regular expressions can handle a multitude of string matching problems with concise expressions.
- **Portable**: The principles of regular expressions can be applied across many programming languages, command-line tools (like `grep`, `sed`, `awk`), databases, etc.
- **Powerful**: Regular expressions can match complex patterns and perform sophisticated string manipulations.

Cons

- **Complexity**: Regular expressions can become complex and hard to understand and maintain, especially for intricate patterns.
- **Readability**: A complicated regular expression can be quite cryptic, and lack of standardization in regex flavors can cause compatibility issues.
- **Efficiency**: Depending on the complexity, regular expressions might not be the most efficient way to perform string operations, especially for very large text data.

14.4.4 Using Regular Expressions in Python

Python's built-in `re` module allows us to use regular expressions (see documentation). The module provides several functions, including `match()`, `search()`, `findall()`, `split()`, `sub()`, and others, each designed to manipulate strings in different ways.

```
import re

search = re.search(r"string[- ]?man", my_text.lower())
```

```
search
```

```
<re.Match object; span=(92, 101), match='stringman'>
```

```
search.group(1)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[16], line 1
----> 1 search.group(1)

IndexError: no such group
```

```
results = re.findall("string[- ]?man", my_text.lower())
```

```
results
```

```
['string-man', 'stringman']
```

```
r"\nmno"
```

```
'\\nmno'
```

```
"\nmno"
```

```
'\nmno'
```

```
print(r"\nmno")
print("\nmno")
```

```
\nmno

mno
```

14.4.5 Text, Numbers, Classes

- `.` any character except “new line” (“\n”)
- `\d` a digit, that is 1, 2, … 9
- `\D` no digit, i.e., any character except 1, 2, … 9
- `\w` a word character, which includes lower + upper case letters, digits, and underscore
- `\W` no word character
- `\s` a space or tab character
- `\S` no space or tab character, almost all characters
- `[aAbBcC]` any of the contained characters
- `[^aAbBcC]` NONE of the contained characters

14.4.6 Positions in the String

- `^` beginning of the string
- `$` end of the string
- `\b` word boundary

14.4.7 Repetitions

- `?` zero or once, i.e., the expression is optional
- `+` at least once
- `*` any number of times (including none)
- `{n}` exactly n times
- `{min,max}` at least min times and at most max times
- `{min,}` at least min times
- `{,max}` at most max times

14.4.8 Characters

- `[A-Z]` ABCDEFGHIJKLMNOPQRSTUVWXYZ
- `[a-z]` abcdefghijklmnopqrstuvwxyz
- `[0-9]` (or `\d`) 0123456789
- `\w` Word = At least one letter, a digit, or underscore (short for `[a-zA-Z0-9_]`).
- `\W` No word. Short for `[^a-zA-Z0-9_]`
- `\d` Digit (short for `[0-9]`).
- `\D` No digit (short for `[^0-9]`).

14.4.9 Groups and Branches

- `|` is an `or` –> `A|B` therefore looks for matches with A or B.
- `[]` contains a set of characters. `[abc]` would therefore expect an a, b, or c.
- `(...)` denotes a group. This will be given separately later, e.g., when we execute `re.findall()` or `re.search()`.
- `(?:...)` is a "non-capturing group", i.e., a group that is not given separately later.
- `(?=...)`, `(?!...)` "Positive Lookahead", "Negative Lookahead" describe patterns that must follow. However, these are not read out with.
- `(?<=...)`, `(?<!...)` "Positive Lookbehind", "Negative Lookbehind" describe patterns that must precede. However, these are not read out with.

```
# Beispiele

re.findall(r"\.\w{2,3}", "dot.com oder dot.de")  #Achtung: nicht r".w..."
```

```
['.com', '.de']
```

```
re.findall(r"\.\w{2,3}$", "dot.com oder dot.de")
```

```
['.de']
```

```
nerd_string = "500GB for 100$ is not so great. \
I want 1000.0 Gigabytes for less than 99.90 $"
```

```
regex = r"[0-9\.]+\s*(GB|[Gg]igabytes)"
re.findall(regex, nerd_string)
```

```
['GB', 'Gigabytes']
```

```
regex = r"([0-9\.]+)\s*(GB|[Gg]igabytes)"
re.findall(regex, nerd_string)
```

```
[('500', 'GB'), ('1000.0', 'Gigabytes')]
```

```
# Artikel und Nomen finden
#regex = r"(der|die|das)\s([A-Z][a-z]*)"
#regex = r"(?:der|die|das)\s[A-Z][a-z]*"
regex = r"(?<=der|die|das)\s([A-Z][a-z]*)"

artikel = """
FRANKFURT/MAIN dpa | Der Frankfurter Oberbürgermeister Peter Feldmann (SPD) muss sich␣
 ↪im Zusammenhang mit der Awo-Affäre vor Gericht verantworten. Eine entsprechende␣
 ↪Anklage der Frankfurter Staatsanwaltschaft wegen des Verdachts der Vorteilsannahme␣
 ↪wurde zugelassen, wie das Landgericht Frankfurt am Montag mitteilte. Termine für␣
 ↪den Prozess standen zunächst noch nicht fest.

Die Ermittlungsbehörde hatte im März Anklage wegen eines hinreichenden Tatverdachts␣
 ↪der Vorteilsannahme erhoben. Feldmanns Frau habe als Leiterin einer Awo-Kita „ohne␣
 ↪sachlichen Grund“ ein übertarifliches Gehalt bezogen, hieß es.

Zudem habe die Awo laut Staatsanwaltschaft Feldmann im Wahlkampf 2018 durch␣
 ↪Einwerbung von Spenden unterstützt. Im Gegenzug habe er die Interessen der Awo␣
 ↪Frankfurt „wohlwollend berücksichtigen“ wollen.

Feldmann steht auch wegen einiger Ausrutscher unter Druck - unter anderem wegen eines␣
 ↪Videos, das zeigt, wie Feldmann auf dem Flug zum Europa-League-Finale von Eintracht␣
 ↪Frankfurt nach Sevilla von Flugbegleiterinnen spricht, „die mich hormonell am␣
 ↪Anfang erst mal außer Gefecht gesetzt haben“. Feldmann entschuldigte sich dafür,␣
 ↪lehnte einen Rücktritt aber ab.
"""
search = re.findall(regex, artikel)
search
```

```
['Awo',
 'Frankfurter',
 'Vorteilsannahme',
 'Landgericht',
 'Vorteilsannahme',
 'Awo',
 'Interessen',
 'Awo']
```

```
import time

def smallest_chatbot_in_the_world():
    print("Hi. What's up?")
    user_input = input(">>> ")

    # mini chat-bot
    regex_problem = r"(problem|issue[s]?) with ([a-zA-Z\s]*\b)"
    problem_mentioned = re.findall(regex_problem, user_input)

    if len(problem_mentioned) > 0:
        problem = problem_mentioned[0][1]
        problem = re.sub(r"\b(my|our)\b", "your", problem)
        print("Come on! ... *MIMIMI* ... ")
        time.sleep(1)
        print(f"Stop complaining about {problem}!!")
    else:
        print("Yeah, I know what you mean.")
    time.sleep(2)
    print("Let's try something better next time... gotta go.")
```

```
smallest_chatbot_in_the_world()
```

```
Hi. What's up?
>>> Hi! I have a real problem with my data science course!
Come on! ... *MIMIMI* ...
Stop complaining about your data science course!!
Let's try something better next time... gotta go.
```

```
smallest_chatbot_in_the_world()
```

```
Hi. What's up?
>>> I have serious issues with our world today.
Come on! ... *MIMIMI* ...
Stop complaining about your world today!!
Let's try something better next time... gotta go.
```

```
#user_input = "Hi there. I have a problem with my broken bike."
#user_input = "Hello... I have a real problem!"
#user_input = "Hello... I have a real problem with all the violence!"

regex_problem = r"(problem|issue) with ([a-zA-Z\s]*\b)"
problem_mentioned = re.findall(regex_problem, user_input)

# mini chat-bot
print(">>", user_input)
if len(problem_mentioned) > 0:
    problem = problem_mentioned[0][1]
    problem = re.sub(r"\b(my|our)\b", "your", problem)
    print(f"What exactly is wrong with {problem}?")
else:
    print("Yeah, I know what you mean.")
```

```
>> Hi! I have a real problem with my data science course!
What exactly is wrong with your data science course?
```

```
import random

reflections = {
    "am": "are",
    "was": "were",
    "i": "you",
    "i'd": "you would",
    "i've": "you have",
    "i'll": "you will",
    "my": "your",
    "are": "am",
    "you've": "I have",
    "you'll": "I will",
    "your": "my",
    "yours": "mine",
    "you": "me",
    "me": "you"
}

psychobabble = [
    [r'I need (.*)',
     ["Why do you need {0}?",
      "Would it really help you to get {0}?",
      "Are you sure you need {0}?"]],

    [r'Why don\'?t you ([^\?]*)\??',
     ["Do you really think I don't {0}?",
      "Perhaps eventually I will {0}.",
      "Do you really want me to {0}?"]],

    [r'Why can\'?t I ([^\?]*)\??',
     ["Do you think you should be able to {0}?",
      "If you could {0}, what would you do?",
      "I don't know -- why can't you {0}?",
      "Have you really tried?"]],

    [r'I can\'?t (.*)',
     ["How do you know you can't {0}?",
      "Perhaps you could {0} if you tried.",
      "What would it take for you to {0}?"]],

    [r'I am (.*)',
     ["Did you come to me because you are {0}?",
      "How long have you been {0}?",
      "How do you feel about being {0}?"]],

    [r'(.*)\?',
     ["Why do you ask that?",
      "Please consider whether you can answer your own question.",
      "Perhaps the answer lies within yourself?",
      "Why don't you tell me?"]],
]
```

(continues on next page)

(continued from previous page)

```
def reflect(fragment):                          #These have to be here...
            tokens = fragment.lower().split()
            for i, token in enumerate(tokens):
                if token in reflections:
                    tokens[i] = reflections[token]
            return ' '.join(tokens)

def eliza_answer(user_input):
    for pattern, responses in psychobabble:
        match = re.search(pattern, str(user_input))
        if match:                                       #ELIZA Responses
            rspns = random.choice(responses)
            return rspns.format(*[reflect(g) for g in match.groups()])
    else:                                               #ChatterBot Responses
        response = "..."
        return response
```

```
user_input = "I am so ashamed of who I was"
print(">>", user_input)
print(eliza_answer(user_input))

user_input = "Its just that I can't forget about it"
print(">>", user_input)
print(eliza_answer(user_input))

user_input = "How should I go on from here?"
print(">>", user_input)
print(eliza_answer(user_input))
```

```
>> I am so ashamed of who I was
How long have you been so ashamed of who you were?
>> Its just that I can't forget about it
How do you know you can't forget about it?
>> How should I go on from here?
Please consider whether you can answer your own question.
```

Some eliza-like example code https://github.com/graylu21/ELIZA-ChatterBot/blob/master/ELIZAChatterBot.py

14.4.10 Developing Regex solutions

Luckily, there are nice tools to help develop regular expressions more interactively. It remains a game of often tricky puzzles, but it often helps a lot to use those tools. For instance:

- https://regex101.com/
- https://regexr.com/

```
dummy.dummy@something.com
this goes to all people@all
This-is-my-mail@my-mail.home.com
```

CHAPTER

FIFTEEN

NLP - BASIC TECHNIQUES TO ANALYSE TEXT DATA

Natural Language Processing (NLP) is a branch of artificial intelligence that deals with the interaction between computers and humans using natural language. It allows computers to read, understand, interpret, and derive meaning from human languages in a valuable and structured manner. NLP stands at the intersection of computer science, artificial intelligence, and computational linguistics, aiming to build computational models of human language understanding for the development of various practical applications.

The importance of NLP in data science cannot be overstated. As we generate vast amounts of data in textual form each day, from social media posts and product reviews to emails and support tickets, NLP provides the tools and techniques necessary to make sense of this data. By transforming unstructured text into structured data, NLP allows us to analyze and extract insights from human language, providing valuable context to support decision-making processes.

NLP is a very large and active field. Here we will only cover a few basics to make the first steps towards applying modern NLP techniques in data science workflows.

15.1 Example areas for the use of NLP techniques

Just to get an idea of how broad as well as how relevant NLP is, here some of the most common applications:

1. **Text Classification**: This NLP application is tasked with assigning predefined categories or tags to text. With its automatic text analysis capabilities, it streamlines the process of organizing and categorizing text, proving essential for tasks such as spam detection, content filtering, and topic labeling.
2. **Sentiment Analysis**: employs NLP to identify and quantify subjective information within a text. This technique measures the sentiment or emotional tone behind words, for instance to help businesses understand customer sentiments towards products, services, or brand topics.
3. **Summarization and Topic Modeling**: These techniques involve distilling large volumes of text into concise summaries or extracting the main topics from a document or a collection of documents. By automatically identifying key points and themes, these applications can make a large corpus of text more accessible and digestible.
4. **Spell Checking**: This is a commonly used application that proofreads text for spelling errors. By comparing words against a dictionary of correctly spelled words, spell checking tools can suggest corrections, thus enhancing the clarity and credibility of the written text.
5. **Machine Translation**: This complex NLP task involves translating text from one language to another. With the ability to handle vast amounts of information, machine translation systems can greatly expedite multilingual communication and overcome language barriers. This technology is, for instance, used to handle large volumes of information that would be impractical to translate manually.
6. **Chatbots**: NLP plays a pivotal role in the functioning of chatbots, enabling them to understand human language, interpret user queries, and respond in a conversational manner. By leveraging NLP, chatbots can deliver customer service, provide information, and even entertain, all in real time. As every reader will know, this field has seen several breakthroughs in recent years.

15.2 Python NLP libraries

There are several Python libraries that are popularly used for modern Natural Language Processing (NLP) tasks. Here are a few of the most commonly used ones:

1. **NLTK (Natural Language Toolkit)**: This is a widely used library for symbolic and statistical NLP. It provides easy-to-use interfaces to over 50 corpora and lexical resources, such as WordNet. NLTK also includes text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning. It's excellent for teaching and working with the basics of NLP.
2. **SpaCy**: This library is known for its advanced NLP capabilities and efficient performance. SpaCy is designed to handle large volumes of text, and its features include named entity recognition, part-of-speech tagging, dependency parsing, and sentence segmentation. Its flexibility and speed make it ideal for production-grade NLP tasks.
3. **Gensim**: This is a robust open-source vector space modeling and topic modeling toolkit. Gensim is designed to handle large text collections using data streaming and incremental algorithms, which is different from most other scientific software packages that only target batch and in-memory processing. It's especially good for tasks that involve topic modeling and document similarity analysis.
4. **TextBlob**: This library simplifies text processing tasks by providing a consistent API for diving into common NLP tasks such as part-of-speech tagging, noun phrase extraction, sentiment analysis, and more. TextBlob is very beginner-friendly and is an excellent choice for basic NLP tasks and for people getting started with NLP in Python.
5. **Transformers (by Hugging Face)**: This library is based on the transformer architecture (like BERT, GPT-2, RoBERTa, XLM, etc) and has pre-trained models for many NLP tasks. It offers simple, yet powerful, APIs for performing tasks such as text classification, named entity recognition, translation, summarization, and more. It is a go-to library for state-of-the-art NLP (but clearly beyond the NLP basics that we will cover here).

Here, we will work with **SpaCy** and **NLTK**.

```
#!pip install nltk
#!pip install spacy
```

```
import os
from matplotlib import pyplot as plt

# NLP related libraries to work with text data
import nltk
import spacy
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[2], line 5
      2 from matplotlib import pyplot as plt
      4 # NLP related libraries to work with text data
----> 5 import nltk
      6 import spacy

ModuleNotFoundError: No module named 'nltk'
```

15.3 Tokenization, Stemming, Lemmatization

Before we dive into any complex operations with text data, we must first prepare it by dividing it into smaller, more manageable units. Similar to how we break down a paragraph into sentences and sentences into words when reading, we apply the same concept in Natural Language Processing (NLP) - but in a slightly different manner. This process is called tokenization.

1. **Tokenization**: This is the first step in many NLP pipelines. Tokenization is the process of breaking up the original raw text into smaller pieces, known as tokens. These tokens help us understand the context in which they're used and draw meaning from them. Tokens are often words, but they can also be phrases, sentences, or other units, depending on the level of detail needed. For example, the sentence "This is an example sentence" would be tokenized into ['This', 'is', 'an', 'example', 'sentence'].

Following tokenization, the resulting tokens often need to be normalized, a process that refines these tokens to improve their usefulness in further analysis.

2. **Token Normalization**: Token normalization is a process that includes converting all text to the same case (usually lower), removing punctuation, and similar tasks. The process involves two major techniques: stemming and lemmatization.

    - **Stemming**: Stemming is the method of reducing inflected or derived words to their base or root form. For example, "running", "runs", and "ran" are all variations of the word "run", so stemming reduces them all to "run". However, stemming can sometimes be too crude, often cutting off the end of words in a way that leaves a base that isn't a real word. For example, "argument" might be stemmed to "argu."
    - **Lemmatization**: Lemmatization, on the other hand, is a more sophisticated process. While it has the same goal as stemming—to reduce a word to its base form—it uses a detailed lexicon and morphological analysis of words to achieve this. For example, lemmatization correctly identifies that the lemma for "better" is "good". While it is more accurate than stemming, it is also more computationally intensive.

Tokenization, followed by token normalization, forms the initial preprocessing steps for most NLP tasks. They transform raw text data into a more digestible and analyzable format, preparing the ground for more advanced NLP techniques.

```
#nltk.download('wordnet')
#nltk.download('omw-1.4')
#nltk.download('punkt')
```

15.4 NLTK

Here, we demonstrate the NLTK processes of tokenization, stemming, and lemmatization.

15.4.1 Tokenization

Tokenization is the process of splitting a large paragraph or text into sentences or words. These sentences or words are known as tokens. This is a crucial step in NLP as we often deal with words in text data.

In this block of code, we import NLTK and define a string of text. The text contains a list of words with different grammatical forms. Using NLTK's `TreebankWordTokenizer`, we break down the text into individual words, or "tokens". The output is a list of these tokens.

```
text = "feet cats wolves talking talked?"

tokenizer = nltk.tokenize.TreebankWordTokenizer()
```

(continues on next page)

(continued from previous page)

```
tokens = tokenizer.tokenize(text)
print(tokens)
```

```
['feet', 'cats', 'wolves', 'talking', 'talked', '?']
```

15.4.2 Stemming

Stemming is a process of reducing inflected (or sometimes derived) words to their word stem or root form—generally a written word form. The stem need not be identical to the morphological root of the word.

Here, we create a `PorterStemmer` object and use it to find the root stem of each word in our list of tokens. The result is a list of these stems. You'll notice that the stems aren't always valid words (like 'wolv' for 'wolves'), as stemming operates on a rule-based approach without understanding the context.

```
stemmer = nltk.stem.PorterStemmer()
print([stemmer.stem(w) for w in tokens])
```

```
['feet', 'cat', 'wolv', 'talk', 'talk', '?']
```

15.4.3 Lemmatization

Lemmatization is the process of reducing inflected words to their word base or dictionary form. It's similar to stemming but is more accurate as it takes the context and meaning of the word into consideration.

Instead of the `PorterStemmer`, we use NLTK's `WordNetLemmatizer` to find the dictionary base form (or lemma) of each word. This results in a list of lemmas. As you can see, lemmatization provides a more accurate root form ('wolf' for 'wolves') as compared to stemming.

```
stemmer = nltk.stem.WordNetLemmatizer()
print([stemmer.lemmatize(w) for w in tokens])
```

```
['foot', 'cat', 'wolf', 'talking', 'talked', '?']
```

15.5 NLP for languages other than English

Natural Language Processing (NLP) is a truly global discipline, extending its reach to languages far beyond just English.

However, it's worth noting that the effectiveness and ease of applying NLP techniques may vary across languages. For instance, languages with complex morphology like Finnish or Turkish, or those with little word delimitation like Chinese, can present unique challenges. Furthermore, resources and pre-trained models, especially those for machine learning, are more readily available for some languages, particularly English, than for others.

15.5.1 Let's try some German

```
text = "Füsse Katzen Wölfe sprechen gesprochen?"  # Not an actual German sentence.␣
↪Only some words for illustrative purposes.

tokenizer = nltk.tokenize.TreebankWordTokenizer()
tokens = tokenizer.tokenize(text)
tokens

stemmer = nltk.stem.SnowballStemmer("german")
print([stemmer.stem(token) for token in tokens])
```

```
['fuss', 'katz', 'wolf', 'sprech', 'gesproch', '?']
```

15.6 Applying SpaCy Models for Lemmatization

SpaCy is a highly versatile and efficient Python library for Natural Language Processing (NLP). It offers comprehensive and advanced functionalities, outperforming NLTK in terms of efficiency and speed. You can find extensive details in SpaCy's official documentation.

Having familiarized ourselves with the concept of lemmatization, let's now explore its practical application using SpaCy.

Initially, you need to ensure that SpaCy and the relevant language models are installed in your environment. In the case of English, `en_core_web_sm` is a suitable model, whereas for German, `de_core_news_sm` can be utilized. SpaCy offers a variety of models for different languages which you can explore on the SpaCy models page.

Installation of SpaCy and downloading of language models can be performed via the following terminal commands:

```
pip install spacy
python -m spacy download en_core_web_sm
python -m spacy download de_core_news_sm
```

Download the required language models first:

```
# Uncomment these lines to download the models
#!python -m spacy download en_core_web_sm
#!python -m spacy download de_core_news_sm
```

Now that the models are installed, we can load the desired one:

```
#nlp = spacy.load('de_core_news_lg')  # large german language model
nlp = spacy.load('de_core_news_sm')  # small german lanugage model
```

Let's now define a text and pass it through the loaded model:

```
text = "Füsse Katzen Wölfe sprechen gesprochen?"
doc = nlp(text)  # create NLP object
```

```
doc
```

```
Füsse Katzen Wölfe sprechen gesprochen?
```

15.6.1 Tokenization

By passing the text through the loaded NLP model, SpaCy already performs tokenization and a host of other operations under the hood:

```
[token.text for token in doc]
```

```
['Füsse', 'Katzen', 'Wölfe', 'sprechen', 'gesprochen', '?']
```

15.6.2 Lemmatization

Unlike **NLTK**, SpaCy has not option for **stemming**. But it provides many different language models (for many different languages) that allow for good **lemmatization**.

```
[token.lemma_ for token in doc]
```

```
['Fuß', 'Katze', 'Wölfe', 'sprechen', 'sprechen', '--']
```

Each word in the text is replaced with its base form or lemma, taking into account its usage in the sentence. This helps in text normalization, a critical step in text preprocessing for NLP tasks.

15.7 Apply tokenization and lemmatization

“War of the worlds” von H.G. Wells

Im Folgenden arbeiten wir mit dem Text des Buches “War of the Worlds” von H.G. Wells (frei verfügbar über das Gutenberg Projekt, die Datei steht auf Moodle).

```
# Define the filename and open the file
filename = "../datasets/wells_war_of_the_worlds.txt"
with open(filename, "r", encoding="utf-8") as file:
    text = file.read()

# Perform some basic cleaning: replace newline characters with spaces
text = text.replace("\n", " ")
```

```
# How many characters?
len(text)
```

```
338168
```

```
# Have a look at the first part of the text
text[:1000]
```

```
'The Project Gutenberg eBook of The War of the Worlds, by H. G. Wells  This eBook␣
↪is for the use of anyone anywhere in the United States and most other parts of␣
↪the world at no cost and with almost no restrictions whatsoever. You may copy it,
↪ give it away or re-use it under the terms of the Project Gutenberg License␣
↪included with this eBook or online at www.gutenberg.org. If you are not located␣
↪in the United States, you will have to check the laws of the country where you␣
↪are located before using this eBook.  Title: The War of the Worlds  Author: H. G.
↪ Wells  Release Date: July 1992 [eBook #36] [Most recently updated: November 27,␣
↪2021]  Language: English   *** START OF THE PROJECT GUTENBERG EBOOK THE WAR OF␣
↪THE WORLDS ***  cover      The War of the Worlds  by H. G. Wells        ‘But who␣
↪shall dwell in these worlds if they be inhabited?     . . . Are we or they Lords␣
↪of the World? . . . And     how are all things made for man?’                  ␣
```

(continues on next page)

(continued from previous page)

```
# Load the English language model
nlp = spacy.load('en_core_web_sm')

# Create an NLP object by processing the text
doc = nlp(text)
```

```
# Tokenization: split the text into individual tokens (words)
tokens = [token.text for token in doc]
print(tokens[:20])
```

```
['The', 'Project', 'Gutenberg', 'eBook', 'of', 'The', 'War', 'of', 'the', 'Worlds',
 ↪ ',', 'by', 'H.', 'G.', 'Wells', ' ', 'This', 'eBook', 'is', 'for']
```

Now that we have all tokens of our book, we can obviously count the number of tokens (which is not the number of words!). But we can also look at how many different tokens there are by using the Python `set()` function.

```
# Print the total number of tokens and the number of unique tokens
print(f"Total tokens: {len(tokens)}")
print(f"Unique tokens: {len(set(tokens))}")
```

```
Total tokens: 71440
Unique tokens: 7292
```

Let us now do the same, but with **lemmatization**.

```
# Lemmatization: reduce each token to its base or root form
lemmas = [token.lemma_ for token in doc]
print(lemmas[:40])
```

```
['the', 'Project', 'Gutenberg', 'eBook', 'of', 'the', 'War', 'of', 'the', 'Worlds',
 ↪ ',', 'by', 'H.', 'G.', 'Wells', ' ', 'this', 'eBook', 'be', 'for', 'the', 'use',
 ↪ 'of', 'anyone', 'anywhere', 'in', 'the', 'United', 'States', 'and', 'most',
 ↪'other', 'part', 'of', 'the', 'world', 'at', 'no', 'cost', 'and']
```

We can also select tokens more specifically by using one of many attributes or methods from SpaCy (see documentation).

For instance:

- `.is_punct` returns `True` if a token is a punctuation.
- `.is_alpha` returns `True`if a token contains alphabetic characters
- `.is_stop` returns `True` if word belongs to a so called "stop list" (less important words, we will come to this later)

Since we here only want to count words:

```
lemmas = [token.lemma_ for token in doc if token.is_alpha]
print(lemmas[:40])
```

```
['the', 'Project', 'Gutenberg', 'eBook', 'of', 'the', 'War', 'of', 'the', 'Worlds',
 ↪ 'by', 'Wells', 'this', 'eBook', 'be', 'for', 'the', 'use', 'of', 'anyone',
 ↪'anywhere', 'in', 'the', 'United', 'States', 'and', 'most', 'other', 'part', 'of
 ↪', 'the', 'world', 'at', 'no', 'cost', 'and', 'with', 'almost', 'no',
 ↪'restriction']
```

```
# Print the total number of lemmas and the number of unique lemmas
print(f"Total lemmas: {len(lemmas)}")
print(f"Unique lemmas: {len(set(lemmas))}")
```

```
Total lemmas: 60629
Unique lemmas: 5589
```

By doing this, we are effectively shrinking the size of the dataset we are working with, while still retaining the essential meaning. It's worth noting that we also removed "stop words" - common words such as "and", "the", "a" - during lemmatization, which usually do not contain important information and are often removed in NLP.

In the following steps, we could now investigate which words are the most common ones, we could identify named entities (such as people or places) or use this text data to train a machine learning model (like a text classifier or a sentiment analysis model).

15.8 Mini-Exercise!

Why do we get more tokens than lemmas? Have a look at both and find the answer!

15.9 Chapter Summary and Outlook

Throughout this chapter, we delved into the world of Natural Language Processing (NLP), exploring several key techniques for handling and processing text data effectively:

- **Cleaning:** This is often the first step in processing text data, involving tasks like removing URLs, Emojis, and special characters, or replacing unwanted line breaks (`"\n"`).
- **Tokenization:** This involves breaking down text into smaller parts called tokens. Tokens can be as small as individual words or can even correspond to sentences or paragraphs, depending on the level of analysis required.
- **Stemming:** Words can appear in different forms depending on gender, number, person, tense, and so on. Stemming involves reducing these words to their root or stem form. For example, the word "finding" could be stemmed to "find". This process is heuristic and sometimes may lead to non-meaningful stems.
- **Lemmatization:** Similar to stemming, lemmatization aims to reduce words to their base form, but with a more sophisticated approach that takes vocabulary and morphological analysis into account. Lemmatization ensures that only the inflectional endings are removed, thus isolating the canonical form of a word known as a lemma. For example, "found" would be lemmatized to "find".
- **Other Operations:** These could include removing numbers, punctuation marks, symbols, and stop words (commonly used words like "and", "the", "a", etc.), as well as converting text to lowercase for uniformity.

We've also discussed the application of these concepts using powerful Python libraries like NLTK and SpaCy, which provide intuitive and efficient tools for dealing with NLP tasks.

CHAPTER

SIXTEEN

COMPUTING WITH TEXT: COUNTING WORDS

16.1 Turning Text into Vectors

As we progress in the world of Data Science, we often find ourselves needing to transform our data into a format that's more suitable for computation and analysis. When dealing with textual data, this means converting our words into numeric representations, often in the form of vectors. But why vectors, you may ask? Machine Learning algorithms, at their core, work with numerical data. They're able to identify patterns, trends and relationships within numeric data much more effectively. Numerical vectors are a very suitable input for machine learning models. In addition, such vectors allow to directly use methods from algebra, for instance to computed distances or similarities between data points.

16.1.1 Distance/similarity measures

Depending on the type of numerical vector there are many suitable methods to compute vector-vector similarities (or inverse: distances). For real-numbered (float) vectors we can for instance simply use the **Euclidean distance** which you will know from Pythagoras. Or a **Cosine similarity** based on the angle between two vectors. For binary vectors there are other measures such as the **Jaccard similarity**.

16.2 One-Hot Encoding

The first step towards this transformation is to use a method known as **One-Hot Encoding**. In this process, each word in the sentence is represented as a vector in an N-dimensional space where N is the number of unique words in the sentence. Each word is represented as a vector of length N, with all elements being 0, except for one element which is 1, indicating the presence of the word.

For instance, consider the sentences: "I like cats" and "I like dogs". In a one-hot encoding scheme, these sentences would be represented by the vectors:

- I: [1, 0, 0, 0]
- like: [0, 1, 0, 0]
- cats: [0, 0, 1, 0]
- dogs: [0, 0, 0, 1]

With this encoding, we can calculate the Euclidean distance between these sentence vectors, providing a way to compare sentences. However, this method has limitations. The vectors can become extremely large for sentences with many unique words, and all words are treated as orthogonal, meaning the distance between any two different words is always the same, which is often not what we want.

For instance, consider the sentences:
"That is the one and only way to happiness."

and
"That is the one and only way to die."

Despite having only one word difference, the Euclidean distance between these sentences would be very small since only one word differs. The distance between "happiness" and "die" is the same as between "That" and "This", even though the first obviously has a much larger impact on the meaning of the sentence.

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
# That is the one and only way to happiness
vector1 = np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 0])
# That is the one and only way to die
vector2 = np.array([1, 1, 1, 1, 1, 1, 1, 1, 0, 1])
# This is the one and only way to happiness
vector3 = np.array([0, 1, 1, 1, 1, 1, 1, 1, 1, 1])


distance = np.sqrt(np.sum(vector1 ** 2 + vector2 ** 2))
print(f"Distance between vector1 and vector2: {distance:.3f}.")

distance = np.sqrt(np.sum(vector1 ** 2 + vector3 ** 2))
print(f"Distance between vector1 and vector3: {distance:.3f}.")
```

```
Distance between vector1 and vector2: 4.243.
Distance between vector1 and vector3: 4.243.
```

16.3 Term Frequency-Inverse Document Frequency (TF-IDF)

To resolve these issues, we use a more sophisticated method called **Term Frequency-Inverse Document Frequency (TF-IDF)**. This measure helps us to evaluate how important a word is to a document within a corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus.

The TF-IDF weight is composed of two terms:

- The Term Frequency (TF), which is the frequency of a word in a document. It is computed as:

$$TF(t) = \frac{\text{Number of times term t appears in a document}}{\text{Total number of terms in the document}}$$

- The Inverse Document Frequency (IDF), computed as the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears.

$$IDF(t) = \log(\frac{\text{Total number of documents}}{\text{Number of documents with term t in it}})$$

- The TF-IDF weight is the product of these quantities:

$$\text{TF–IDF} = TF \times IDF$$

This results in a vector space model where each word is assigned a TF-IDF score, and a document is represented as a vector of these scores. We can use libraries like `scikit-learn` to compute these scores easily.

Finally, we can use these TF-IDF vectors to train Machine Learning models. For example, we can train a logistic regression model for sentiment analysis. By turning our words into vectors, we've opened up a new world of possibilities for analysis and model training, pushing the boundaries of what we can achieve with Natural Language Processing.

16.4 Hands-on: Hotel Reviews

Let's work with some actual data!

We here will work with a dataset of about 20,000 hotel reviews from tripadvisor see original dataset on zenodo or see dataset on kaggle. The goal will be to train a machine learning model to predict rating based on a review text.

16.4.1 Data import and exploration

We will import the data using `Pandas`. We will then explore the data a bit. But since this dataset was already prepared to work for machine learning tasks, there is complicated data processing steps that we have to do at this point.

```
File ..\datasets\tripadvisor_hotel_reviews.csv already exists. Skipping download.
```

```
filename = "../datasets/tripadvisor_hotel_reviews.csv"
data = pd.read_csv(filename)
data.head()
```

```
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
Cell In[4], line 2
      1 filename = "../datasets/tripadvisor_hotel_reviews.csv"
----> 2 data = pd.read_csv(filename)
      3 data.head()

File ~\anaconda3\envs\data_science\lib\site-packages\pandas\io\parsers\readers.
 ↪py:948, in read_csv(filepath_or_buffer, sep, delimiter, header, names, index_col,
 ↪ usecols, dtype, engine, converters, true_values, false_values, skipinitialspace,
 ↪ skiprows, skipfooter, nrows, na_values, keep_default_na, na_filter, verbose,␣
 ↪skip_blank_lines, parse_dates, infer_datetime_format, keep_date_col, date_parser,
 ↪ date_format, dayfirst, cache_dates, iterator, chunksize, compression, thousands,
 ↪ decimal, lineterminator, quotechar, quoting, doublequote, escapechar, comment,␣
 ↪encoding, encoding_errors, dialect, on_bad_lines, delim_whitespace, low_memory,␣
 ↪memory_map, float_precision, storage_options, dtype_backend)
    935 kwds_defaults = _refine_defaults_read(
    936     dialect,
    937     delimiter,
   (...)
    944     dtype_backend=dtype_backend,
    945 )
    946 kwds.update(kwds_defaults)
--> 948 return _read(filepath_or_buffer, kwds)

File ~\anaconda3\envs\data_science\lib\site-packages\pandas\io\parsers\readers.
 ↪py:611, in _read(filepath_or_buffer, kwds)
    608 _validate_names(kwds.get("names", None))
    610 # Create the parser.
--> 611 parser = TextFileReader(filepath_or_buffer, **kwds)
```

(continues on next page)

(continued from previous page)

```
    613 if chunksize or iterator:
    614     return parser

File ~\anaconda3\envs\data_science\lib\site-packages\pandas\io\parsers\readers.
↪py:1448, in TextFileReader.__init__(self, f, engine, **kwds)
   1445     self.options["has_index_names"] = kwds["has_index_names"]
   1447 self.handles: IOHandles | None = None
-> 1448 self._engine = self._make_engine(f, self.engine)

File ~\anaconda3\envs\data_science\lib\site-packages\pandas\io\parsers\readers.
↪py:1723, in TextFileReader._make_engine(self, f, engine)
   1720     raise ValueError(msg)
   1722 try:
-> 1723     return mapping[engine](f, **self.options)
   1724 except Exception:
   1725     if self.handles is not None:

File ~\anaconda3\envs\data_science\lib\site-packages\pandas\io\parsers\c_parser_
↪wrapper.py:93, in CParserWrapper.__init__(self, src, **kwds)
     90 if kwds["dtype_backend"] == "pyarrow":
     91     # Fail here loudly instead of in cython after reading
     92     import_optional_dependency("pyarrow")
---> 93 self._reader = parsers.TextReader(src, **kwds)
     95 self.unnamed_cols = self._reader.unnamed_cols
     97 # error: Cannot determine type of 'names'

File parsers.pyx:579, in pandas._libs.parsers.TextReader.__cinit__()

File parsers.pyx:668, in pandas._libs.parsers.TextReader._get_header()

File parsers.pyx:879, in pandas._libs.parsers.TextReader._tokenize_rows()

File parsers.pyx:890, in pandas._libs.parsers.TextReader._check_tokenize_status()

File parsers.pyx:2050, in pandas._libs.parsers.raise_parser_error()

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc7 in position 10968:␣
↪invalid continuation byte
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20491 entries, 0 to 20490
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   Review  20491 non-null  object
 1   Rating  20491 non-null  int64 
dtypes: int64(1), object(1)
memory usage: 320.3+ KB
```

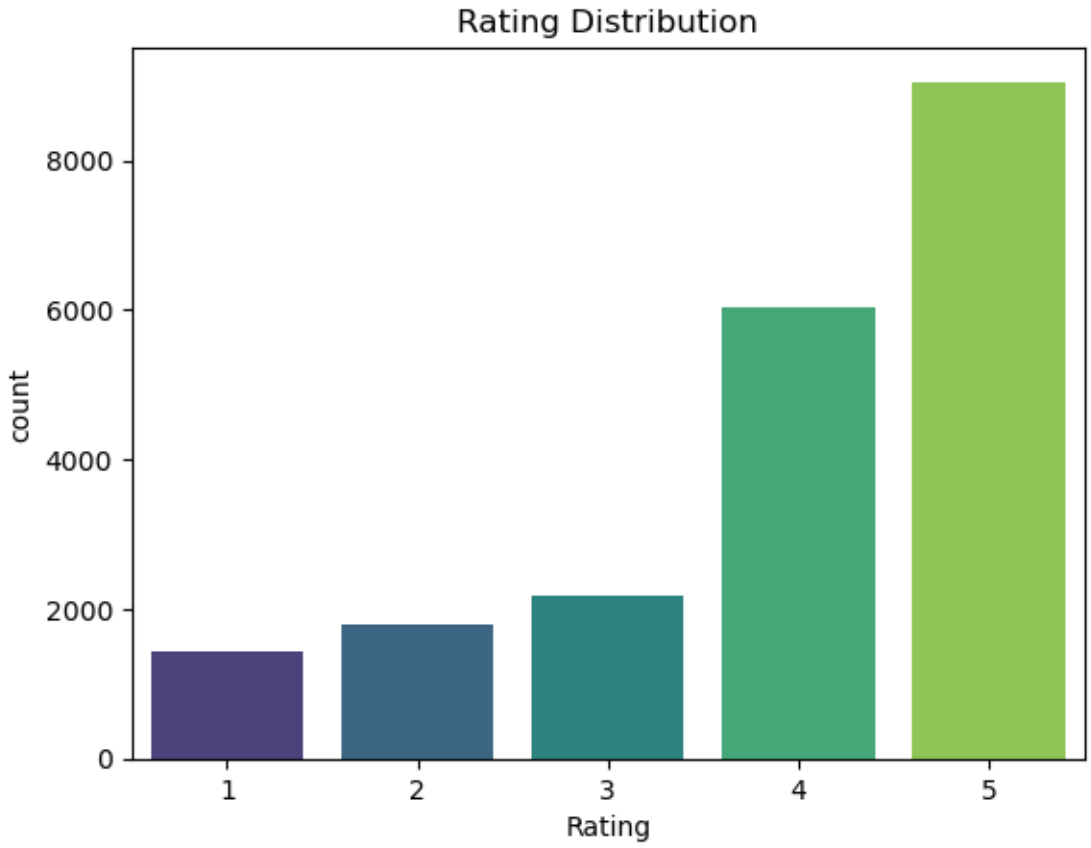
```
sb.countplot(data=data,
             x='Rating',
             palette='viridis'
```

(continues on next page)

(continued from previous page)

```
            ).set_title('Rating Distribution')
```

```
Text(0.5, 1.0, 'Rating Distribution')
```



Here we see a first potential issue, a **bias of the ratings**. The dataset contains far more positive reviews (>= 4) than neutral or negative ones. This is a common problem in machine learning and there are strategies to deal with this (e.g. *oversampling* or using *class weights*), but for now we will simply ignore it.

16.4.2 Train/Test Split

Let us first split the data into a training and test set (see chapters on machine learning!).

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    data['Review'], data['Rating'], test_size=0.2, random_state=0)

print(f"Train dataset size: {X_train.shape}")
print(f"Test dataset size: {X_test.shape}")
```

```
Train dataset size: (16392,)
Test dataset size: (4099,)
```

```
tfidf = TfidfVectorizer()
tfidf_vectors = tfidf.fit_transform(X_train)
tfidf_vectors.shape
```

```
(16392, 46816)
```

This is a pretty big array (or matrix). So, how does such a tfidf vector look like, for instance the tfidf-vector for the first review?

```
# This is the review
data.iloc[0, 0]
```

```
'nice hotel expensive parking got good deal stay hotel anniversary, arrived late␣
↪evening took advice previous reviews did valet parking, check quick easy, little␣
↪disappointed non-existent view room room clean nice size, bed comfortable woke␣
↪stiff neck high pillows, not soundproof like heard music room night morning loud␣
↪bangs doors opening closing hear people talking hallway, maybe just noisy␣
↪neighbors, aveda bath products nice, did not goldfish stay nice touch taken␣
↪advantage staying longer, location great walking distance shopping, overall nice␣
↪experience having pay 40 parking night,  '
```

```
# And this is the tfidf-vector of this review
tfidf_vectors[0, :]
```

```
<1x46816 sparse matrix of type '<class 'numpy.float64'>'
        with 248 stored elements in Compressed Sparse Row format>
```

What does `sparse matrix` mean?

We have seen before, that such sentence (or text) vector can quickly become very long. Here each vector as 47073 positions, one for each word. But most of these words do not occur in a particular sentence, so most positions in a tfidf-vector are simply `0`.

To save all those numbers would be very vastefull memorywise. So, scikit-learn here uses so called `sparse matrices` which only store the data for non-zero positions. We can look at the data with the `.data` attribute. Or we can convert it to a dense vector (that includes all zeros) by using `.toarray()`.

16.4.3 Reduce tfidf vector size

So far we simply used the `TfidfVectorizer` without setting any parameters, which means that it will simply use default values. In our case, however, we end up with rather large tfidf vectors. In particular given the relatively small dataset (small for NLP standards) and the rather short reviews.

In many cases, and here as well, we also deal with text data of strongly varying quality. This can lead to many words in the `tfidf model` which are not meaningfull enough. We can have a look at the words which were included:

```
# Have a look at the last 100 words in the tfidf model
tfidf.get_feature_names_out()[-100:]
```

```
array(['zoned', 'zones', 'zonethis', 'zoning', 'zoo', 'zoogarten',
       'zooji', 'zoological', 'zoologicher', 'zoologische',
       'zoologischer', 'zoologisher', 'zoom', 'zooming', 'zooms', 'zoomy',
       'zoran', 'zorbas', 'zucca', 'zuid', 'zum', 'zumo', 'zurich',
       'zvago', 'zyrtec', 'zytec', 'zz', 'zzzt', 'zzzzt', 'zzzzzs',
       'zzzzzzzzz', 'µlack', 'ààmake', 'ààneither', 'ààthe', 'ààthere',
       'âge', 'âme', 'ânes', 'âre', 'âs', 'ä__ç', 'ä__ç_é_', 'ä_ëù_ez',
       'äarely', 'äbec', 'äcialy', 'äciate', 'äciated', 'äcor', 'äe',
       'äed', 'äellement', 'äes', 'ägal', 'älanie', 'älie', 'ändo',
       'änial', 'äon', 'äpublique', 'ära', 'ärico', 'äring', 'äro', 'äs',
       'ätence', 'ätro', 'äut', 'åeach', 'åreakfast', 'æn', 'ærom',
       'æsimas', 'èery', 'èou', 'é__', 'é_ç', 'é_ç_a', 'é_çål', 'éenny',
       'êtyle', 'î_', 'î__ere', 'î__here', 'î__hese', 'öreat', 'ù_', 'ùn',
       'ùoom', 'ûan', 'ü_e', 'üefinitely', 'üescribed', 'üifficult',
       'üust', 'üâjili', 'üã', 'üè__', 'üècia'], dtype=object)
```

Many of those words make no sense! It includes rare strings and typos.

In such cases it is generelly advisable to reduce the number of words in our tfidf vector by removing words which are either occuring very rarely (only a few in the whole dataset) or which occur very often (in a large part of the documents). The first means our machine learning models can anyway not really learn a lot from it because it hardly occurs during training. The second means that such words probably have very little *discriminative power* because they are so common.

Both things can easily be done with the `TfidfVectorizer` by setting the `min_df` (minimum document frequency) and the `max_df` (maximum document frequency) values. We can use integers which will be interpreted as absolute numbers, for instance `min_df=10` means the word must occur at least 10 times. But we can also use floats which will be interpreted as fractions, so `max_df=0.25` meant that words which occur in more than 25% of all documents are not taken into our tfidf vectors.

```
tfidf = TfidfVectorizer(min_df=10, max_df=0.25)
tfidf_vectors = tfidf.fit_transform(X_train)
tfidf_vectors.shape
```

```
(16392, 8533)
```

```
# Again have a look at the last 100 words in the tfidf model
tfidf.get_feature_names_out()[-100:]
```

```
array(['workers', 'working', 'workmen', 'workout', 'works', 'world',
       'worlds', 'worldwide', 'worn', 'worried', 'worries', 'worry',
       'worrying', 'worse', 'worst', 'worth', 'worthwhile', 'worthy',
       'wotif', 'would', 'wouldn__ç_é_', 'wouldnt', 'wound', 'wow',
       'wrap', 'wrapped', 'wraps', 'wreck', 'wrist', 'wristband', 'write',
       'writing', 'written', 'wrong', 'wrote', 'wrought', 'wtc', 'www',
       'wyndham', 'wynyard', 'xmas', 'ya', 'yahoo', 'yard', 'yards',
       'yea', 'yeah', 'year', 'yearly', 'years', 'yell', 'yelled',
       'yelling', 'yellow', 'yen', 'yep', 'yes', 'yesterday', 'yet',
       'yikes', 'yo', 'yoga', 'yoghurt', 'yoghurts', 'yogurt', 'yogurts',
       'york', 'yorker', 'yorkers', 'you', 'you__ç_éèe', 'you__ç_éêl',
       'you__ç_éö', 'young', 'younger', 'youngest', 'your', 'youre',
       'yourself', 'youth', 'yr', 'yrs', 'yuan', 'yuck', 'yuk', 'yum',
       'yummy', 'yunque', 'zealand', 'zen', 'zero', 'zip', 'zocalo',
       'zona', 'zone', 'zones', 'zoo', 'äcor', 'äes', 'äs'], dtype=object)
```

This already looks much better! The data still contains some weird words, but we will leave those for now.

```
tfidf_vectors[0, :].data
```

```
array([0.06700227, 0.04814239, 0.0737314 , 0.14154248, 0.10954503,
       0.07379485, 0.05589574, 0.02538999, 0.03330127, 0.07433131,
       0.07433131, 0.04219366, 0.04137184, 0.05397631, 0.03138902,
       0.04151596, 0.06958298, 0.02706299, 0.06886759, 0.03304963,
       0.04081672, 0.05737137, 0.05624327, 0.05360418, 0.06619347,
       0.02755513, 0.05002418, 0.05606788, 0.05404594, 0.04785075,
       0.04915887, 0.03642892, 0.03748446, 0.03243513, 0.07721433,
       0.30833586, 0.04426299, 0.04034077, 0.03785657, 0.03519163,
       0.06277531, 0.02390963, 0.06435793, 0.05350299, 0.06728831,
       0.0692184 , 0.05539765, 0.06313989, 0.04911765, 0.05564336,
       0.06758343, 0.07433131, 0.06259825, 0.0370815 , 0.1068746 ,
       0.04739551, 0.03907159, 0.03288319, 0.04419008, 0.04091831,
       0.04135398, 0.04753323, 0.04492   , 0.0333464 , 0.04303091,
       0.0957015 , 0.07615933, 0.07316787, 0.06758343, 0.05250387,
       0.04423862, 0.04423862, 0.0652141 , 0.05165567, 0.13853985,
       0.03150579, 0.02665046, 0.23164298, 0.05275866, 0.09108527,
       0.13444755, 0.09382108, 0.10954503, 0.06700227, 0.05539765,
       0.04749861, 0.0377591 , 0.04781493, 0.03839869, 0.06351934,
       0.0266909 , 0.04271578, 0.03033907, 0.04078311, 0.04557067,
       0.03830825, 0.11754202, 0.0503942 , 0.09477833, 0.04329094,
       0.05767617, 0.10984981, 0.06160136, 0.05454852, 0.0298762 ,
       0.03355992, 0.03233317, 0.07775643, 0.07971517, 0.02936134,
       0.02768133, 0.03363738, 0.05462259, 0.1457154 , 0.04588483,
       0.07751199, 0.03236025, 0.04746412, 0.03132208, 0.05440213,
       0.04421432, 0.05688459, 0.05053684, 0.04746412, 0.05949569,
       0.05274376, 0.07336576, 0.07263656, 0.03881012, 0.07509548,
       0.04574072, 0.06135586, 0.06454185, 0.0334297 , 0.05994018,
       0.03294796, 0.03825697, 0.04636163, 0.05809876, 0.02853945,
       0.14633573, 0.05937092, 0.04446024, 0.06802072, 0.09465483,
       0.03802962, 0.03411622, 0.03610889, 0.03863536, 0.06470694,
       0.09758859, 0.03315243, 0.03636611, 0.06028169, 0.06642336,
       0.03533992, 0.09977828, 0.08373493, 0.0373908 , 0.02631188,
       0.0526832 , 0.02898129, 0.04008825, 0.04060035, 0.06351934,
       0.02896245, 0.04887409, 0.02379561, 0.13081138, 0.03259367,
       0.05053684, 0.0942867 , 0.07433131, 0.04409375, 0.07497263,
       0.03235347, 0.04421432, 0.0504415 , 0.04993394, 0.06218894,
       0.03934063, 0.04173641, 0.05988039, 0.03919806, 0.05102878,
       0.06799905, 0.03377057, 0.03209952, 0.04109014, 0.02481554,
       0.07081109, 0.08461189, 0.02768953, 0.03248309, 0.05929106,
       0.04591393, 0.03610889, 0.03639747, 0.0959347 , 0.03734433,
       0.06432815, 0.12320272, 0.05118158, 0.06857963, 0.06271706,
       0.07497263, 0.02682822, 0.12083881, 0.05087844, 0.16477471,
       0.03443276, 0.14240732, 0.13344956, 0.06313989, 0.05398083])
```

```
tfidf_vectors[0, :].indices
```

```
array([2894,  864,  991, 6611, 6697,  871, 4969, 7835, 3743, 2802, 2268,
       6922, 7978, 6620, 5274, 5551, 1555, 8271, 8384, 3622, 1706, 7239,
       4021, 8229, 2381, 7254, 7650, 5902,  336, 5069, 2957, 5398, 1479,
       4456, 3270, 2976, 2477,  301, 4967, 1333, 6204, 6070, 4966,  302,
       1712,  778, 3356, 8206, 7235, 3432, 4619, 8203, 4668, 3875, 6431,
       1450, 7299, 2907, 2926,  473, 6766, 2966,  149,  121, 3454, 8364,
       6375, 8105, 5406, 2297, 1376, 1040, 5720, 6552, 6816, 2916, 8328,
```

(continues on next page)

(continued from previous page)

```
       2384, 8287, 5188, 4808, 2546, 5075,  847, 7547, 3358, 4786, 5623,
       6858, 8320, 3170, 3469, 2413, 5074, 5651,    0,    3, 7664, 6484,
       1624, 3792, 6825, 1807, 8246, 8041, 7753, 8162, 3746, 3457, 4979,
       6266, 2817, 3634, 5681, 7899, 5682, 3674, 8463, 5077, 3914, 4740,
       2044, 6343, 5892,  725, 6687, 6286, 6604, 4155, 6285, 6603, 2547,
       1904,  656, 7395, 8430, 1453, 7455, 3145, 4410, 5695, 6024, 7961,
        258, 8398, 1434, 1242, 4327, 8383, 5197, 3576, 1295, 1670, 5744,
       3745, 4296,  249, 7877, 2820,  963,  255, 5281, 7863, 3957, 7139,
       6786, 2110,  855, 2891, 7888, 8441, 4394,  903, 6977, 4927, 4832,
       3237, 4754, 4810, 7607, 8434, 2553, 7338, 7454, 6112, 6916, 6477,
       3861, 8215,  892,  304, 5954, 6550, 3763,  899, 5931, 5136, 8150,
        604, 7756, 5295, 5096, 5142, 1211, 4533, 2769, 4546,  662, 1448,
       2259, 6005, 2046, 8145, 5973, 1503])
```

```
example_vector = pd.DataFrame({
    "word": tfidf.get_feature_names_out()[tfidf_vectors[0, :].indices],
    "tfidf": tfidf_vectors[0, :].data
})
example_vector
```

```
              word     tfidf
0          fajardo  0.067002
1              bay  0.048142
2    bioluminescent  0.073731
3             seas  0.141542
4            seven  0.109545
..             ...       ...
210           rate  0.034433
211        culebra  0.142407
212        vieques  0.133450
213     rainforest  0.063140
214          check  0.053981

[215 rows x 2 columns]
```

```
# we don't really want to look at the full vector (let's do 20)
print(tfidf_vectors[0, :].toarray()[0, :20])
print(tfidf_vectors[0, :].toarray().shape)
```

```
[0.03345672 0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.        ]
(1, 46816)
```

16.5 Regression or Classification?

When we're dealing with machine learning tasks involving labels or targets, these tasks usually fall under two main categories: Regression and Classification.

As we already discussed in the machine learning part of the course, **regression** generally refers to predicting a continuous number while **classification** refers to the prediction of specific, discret values.

If we want to predict if a mail is spam or not, or if we want to predict if an image displays a dog, a cat, or a parrot, we have a classification task.

Here we want to predict ratings between 0 (bad) and 5 (very good), so what kind of task is this? What do you think?

Actually, this case is a bit of a gray zone.

The ratings are discrete numbers: 1, 2, 3, 4, or 5. But the ratings are also **ordered**, so that predicting a 2 where the true label is a 1 is not that bad! When we train a model on a classification task, then it usually doesn't matter which wrong class was predicted, so predicting a 2 where it's actually 1 is as bad as predicting a 4 [^loss-functions]. So, that makes *regression* to appear as the natural choice…

However, in practice, training a model on a classification task typically works notably better in such situations where we only have very few possible ordered values (see also [Gupta *et al.*, 2010]).

Nevertheless, we will start with a **linear regression** model and than compare it to a **logistic regression** model.

```
y_train.head()
```

```
4115     4
16210    5
4075     4
8215     3
6499     4
Name: Rating, dtype: int64
```

16.5.1 Linear Regression model

- This model is fast to train an can handle the size of this data fairly well.
- It will output floats instead of the actual labels 1 to 5.

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(tfidf_vectors, y_train)
```

```
LinearRegression()
```

Evaluate the model

For this we will first convert the test data using the exact same tfidf model as for the train data. This is very important, if you compute a new tfidf model using fit or fit_transform, this approach won't work!

```
tfidf_vectors_test = tfidf.transform(X_test)

predictions = model.predict(tfidf_vectors_test)
```

```
np.round(predictions[:20], 1)
```

```
array([1.9, 1.8, 5.7, 6.1, 3.7, 4.7, 3.5, 1.6, 5.2, 3.7, 4. , 4. , 5.3,
       4.5, 4.3, 5.3, 5.7, 3. , 4.5, 4.4])
```

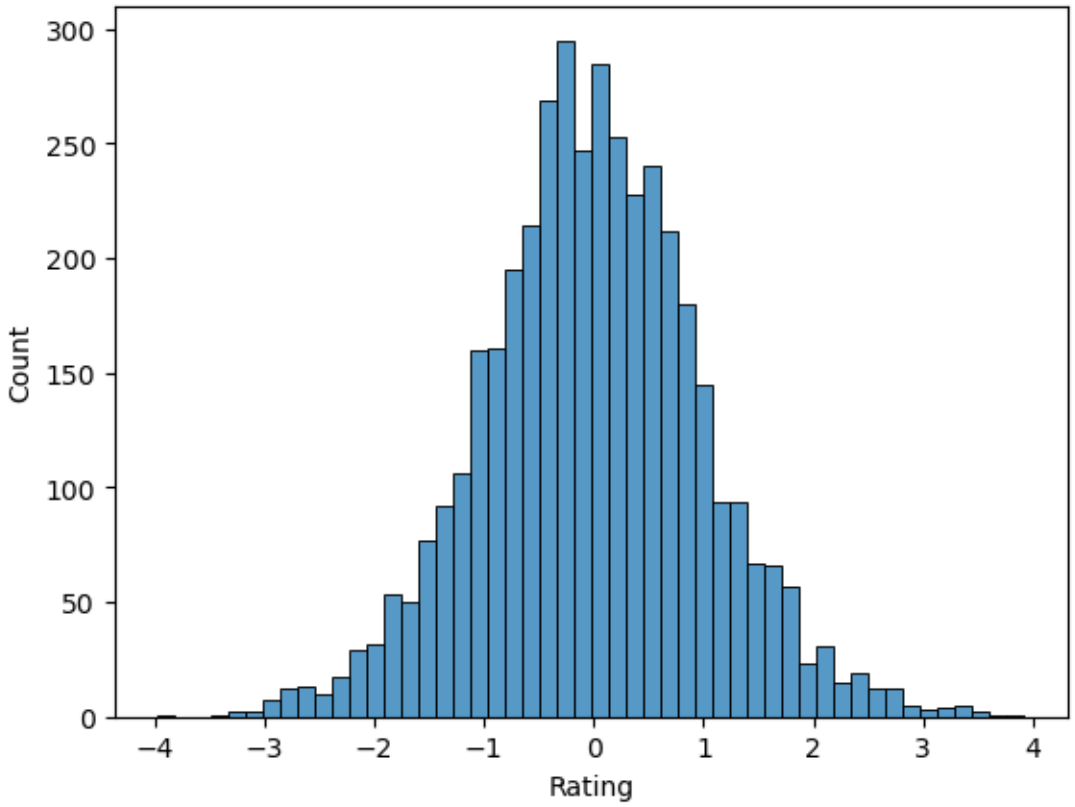
```
y_test[:20].values
```

```
array([2, 2, 4, 5, 3, 5, 4, 3, 4, 5, 4, 4, 4, 4, 5, 5, 5, 4, 5, 5],
      dtype=int64)
```

How good are the predictions?

```
sb.histplot(predictions - y_test)
```

```
<Axes: xlabel='Rating', ylabel='Count'>
```



```
# mean absolute error:
np.abs(predictions - y_test).mean()
```

```
0.7872191832942967
```

A mean absolute error of 0.8 for rating from 1 to 5 is not spectacular, but it isn't bad either. There is one very particular problem due to the linear regression model though:

```
predictions.min(), predictions.max()
```

```
(-1.8635146070281188, 7.274303749464632)
```

Prediction of impossible values

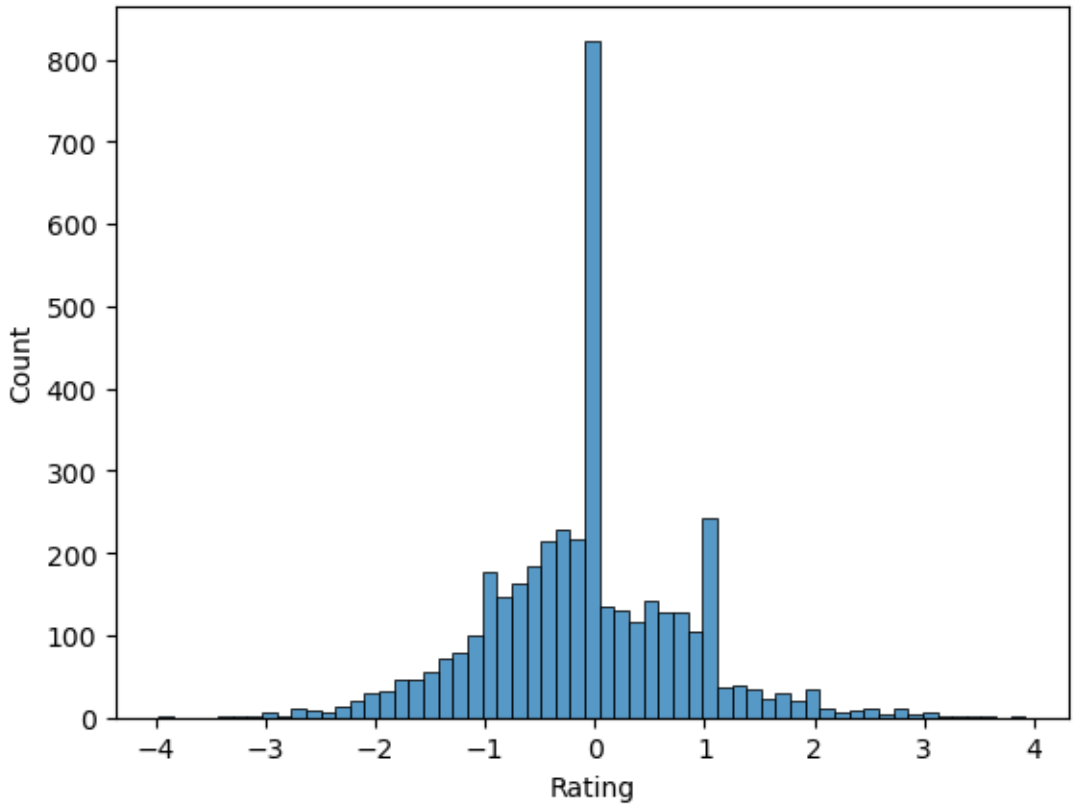
The linear regression model has no bounds and could in principle output any float value. Here we got impossible negative values and ratings > 5. This could of course be fixed by simply adding an additional line of code that sets all values < 1 to 1 and all values > 5 to 5.

```
predictions[predictions < 1] = 1
predictions[predictions > 5] = 5

print(f"Mean absolute error after clipping the predictions: {np.abs(predictions - y_
↪test).mean()}")
```

```
Mean absolute error after clipping the predictions: 0.6740064461149007
```

```
sb.histplot(predictions - y_test)
```

```
<Axes: xlabel='Rating', ylabel='Count'>
```



Interestingly, this clipping step improved the quality of the predictions notably which can both be seen in the mean absolute error (MAE) and the error distribution.

16.5.2 Logistic Regression model

- hint: in this case don't worry if you get warnings about reaching the iterations limit

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(tfidf_vectors, y_train)
```

```
C:\Users\flori\anaconda3\envs\ai_smart_health\lib\site-packages\sklearn\linear_
 ↪model\_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

```
LogisticRegression()
```

```
tfidf_vectors_test = tfidf.transform(X_test)

predictions = model.predict(tfidf_vectors_test)
```

```
np.round(predictions[:20], 1)
```

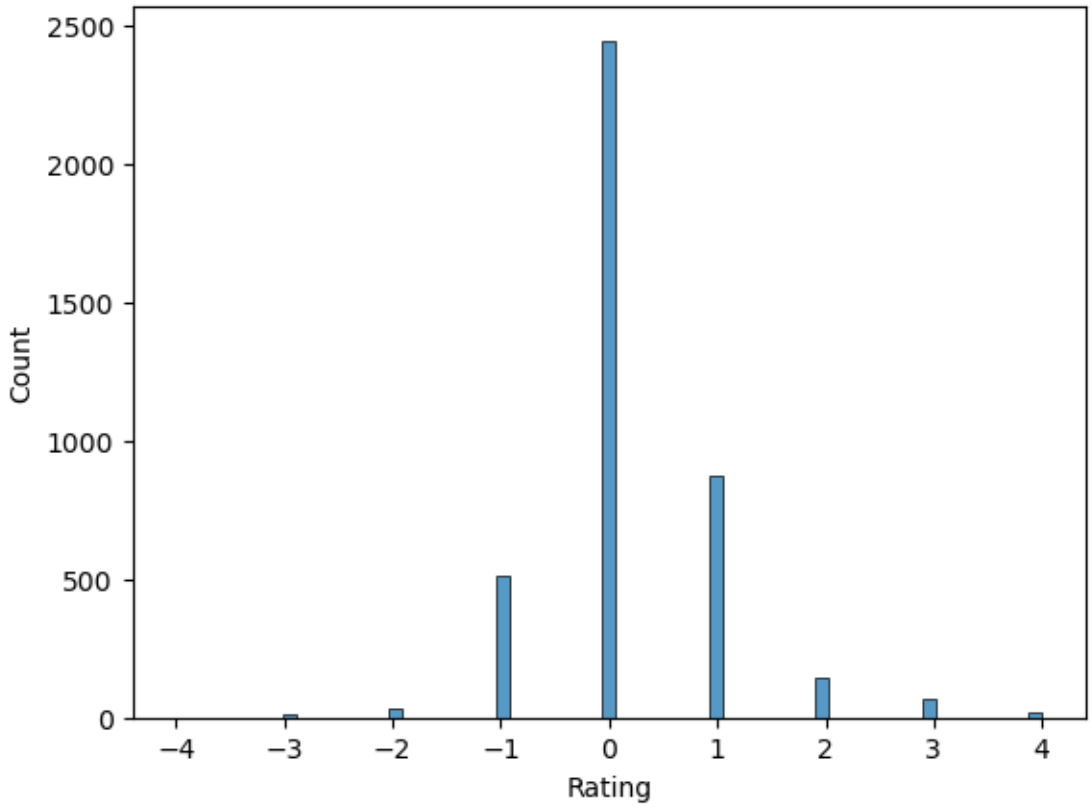
```
array([2, 4, 5, 5, 4, 5, 5, 2, 5, 5, 4, 4, 4, 5, 5, 5, 5, 4, 4, 5],
      dtype=int64)
```

```
y_test[:20].values
```

```
array([2, 2, 4, 5, 3, 5, 4, 3, 4, 5, 4, 4, 4, 4, 5, 5, 5, 4, 5, 5],
      dtype=int64)
```

```
sb.histplot(predictions - y_test)
```

```
<Axes: xlabel='Rating', ylabel='Count'>
```




```
# mean absolute error:
np.abs(predictions - y_test).mean()
```

```
0.4981702854354721
```

16.5.3 Back to: regression vs. classification

For a classification task it usually does not make sense to compute mean absolute errors. But here it does. And it allows us to compare the performance of the linear regression vs. the logistic regression model. And, as spoilert above, the classification task does indeed result in a better predictions!

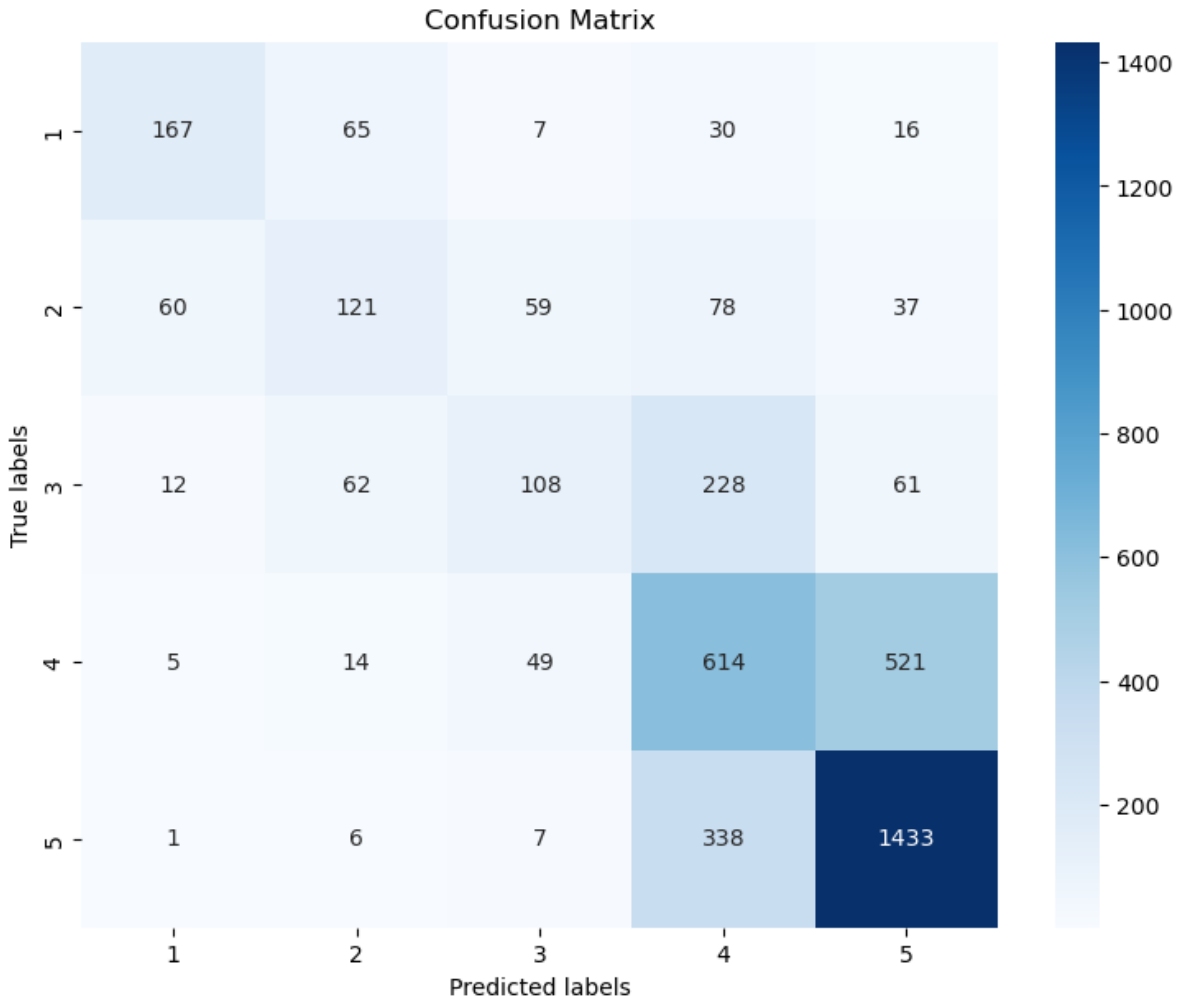
```
from sklearn.metrics import confusion_matrix, classification_report

print(confusion_matrix(y_test, predictions))
print(classification_report(y_test, predictions))
```

```
[[ 167   65    7   30   16]
 [  60  121   59   78   37]
 [  12   62  108  228   61]
 [   5   14   49  614  521]
 [   1    6    7  338 1433]]
              precision    recall  f1-score   support

           1       0.68      0.59      0.63       285
           2       0.45      0.34      0.39       355
           3       0.47      0.23      0.31       471
           4       0.48      0.51      0.49      1203
           5       0.69      0.80      0.74      1785

    accuracy                           0.60      4099
   macro avg       0.55      0.49      0.51      4099
weighted avg       0.58      0.60      0.58      4099
```

16.5.4 Confusion matrix

The confusion matrix can tell us a lot about where the model works well and where it fails. Often is is more accessible if the matrix is plotted, for instance using seaborns `heatmap`.

```
cm = confusion_matrix(y_test, predictions, labels=model.classes_)

# Plotting the confusion matrix with a heatmap
plt.figure(figsize=(9,7))
sb.heatmap(cm, annot=True, fmt='d',
           cmap='Blues',
           xticklabels=model.classes_,
           yticklabels=model.classes_)
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
plt.show()
```



16.5.5 Convert your prediction tasks

Sometimes it can help to change the actual prediction task to make it clearer, and thereby simpler, for a machine learning model to learn. In the present case we could for instance say that we convert that ratings 1 to 5 into only three categories: bad (1 or 2), neutral (3), or good (4 or 5).

This is of course less nuanced, but it can make it an easier classification task.

```
# Change the rating to Good - Neutral - Bad
def convert_rating(score):
    if score > 3:
        return 'good'
    if score == 3:
        return 'neutral'
    return 'bad'

data["rating_simplified"] = data['Rating'].apply(convert_rating)
data.head()
```

```
                                              Review  Rating rating_simplified
0  nice hotel expensive parking got good deal sta...       4              good
1  ok nothing special charge diamond member hilto...       2               bad
2  nice rooms not 4* experience hotel monaco seat...       3           neutral
3  unique, great stay, wonderful time hotel monac...       5              good
4  great stay great stay, went seahawk game aweso...       5              good
```

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    data['Review'],
    data["rating_simplified"],
    test_size=0.2,
    random_state=0)

print(f"Train dataset size: {X_train.shape}")
print(f"Test dataset size: {X_test.shape}")
```

```
Train dataset size: (16392,)
Test dataset size: (4099,)
```

```
tfidf = TfidfVectorizer()
tfidf_vectors = tfidf.fit_transform(X_train)
tfidf_vectors.shape
```

```
(16392, 46816)
```

```
model = LogisticRegression()
model.fit(tfidf_vectors, y_train)
```

```
C:\Users\flori\anaconda3\envs\ai_smart_health\lib\site-packages\sklearn\linear_
 ↪model\_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

```
LogisticRegression()
```

```
tfidf_vectors_test = tfidf.transform(X_test)

predictions = model.predict(tfidf_vectors_test)
```

```
predictions[:20]
```

```
array(['bad', 'bad', 'good', 'good', 'good', 'good', 'good', 'bad',
```

(continues on next page)

(continued from previous page)

```
       'good', 'good', 'good', 'good', 'good', 'good', 'good', 'good',
       'good', 'good', 'good', 'good'], dtype=object)
```

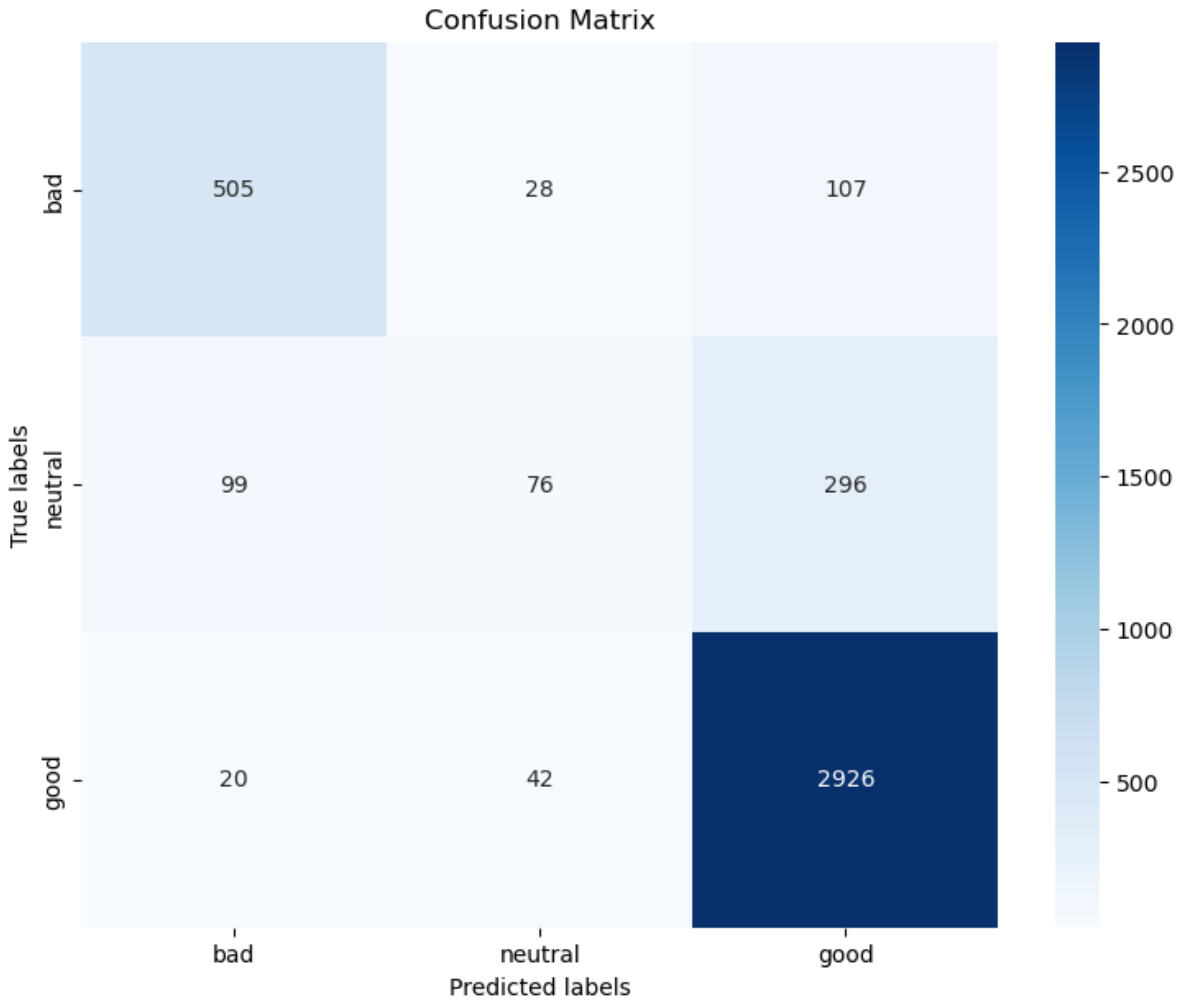
```
y_test[:20].values
```

```
array(['bad', 'bad', 'good', 'good', 'neutral', 'good', 'good', 'neutral',
       'good', 'good', 'good', 'good', 'good', 'good', 'good', 'good',
       'good', 'good', 'good', 'good'], dtype=object)
```

```
from sklearn.metrics import confusion_matrix, classification_report

print(confusion_matrix(y_test, predictions))
print(classification_report(y_test, predictions))
```

```
[[ 505  107   28]
 [  20 2926   42]
 [  99  296   76]]
              precision    recall  f1-score   support

         bad       0.81      0.79      0.80       640
        good       0.88      0.98      0.93      2988
     neutral       0.52      0.16      0.25       471

    accuracy                           0.86      4099
   macro avg       0.74      0.64      0.66      4099
weighted avg       0.83      0.86      0.83      4099
```

```
labels = ["bad", "neutral", "good"]
cm = confusion_matrix(y_test, predictions, labels=labels)

# Plotting the confusion matrix with a heatmap
plt.figure(figsize=(9,7))
sb.heatmap(cm, annot=True, fmt='d',
           cmap='Blues',
           xticklabels=labels,
           yticklabels=labels)
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
plt.show()
```



16.5.6 Question to you:

Judge yourself. Is this a good result? Could this be done better (and if so, how?) ?

CHAPTER

SEVENTEEN

BEYOND COUNTING INDIVIDUAL WORDS: N-GRAMS

So far in our journey through text data processing, we've dealt with counting individual words. While this approach, often referred to as a "bag of words" model, can provide a basic level of understanding and can be useful for certain tasks, it often falls short in capturing the true complexity and richness of language. This is mainly because it treats each word independently and ignores the context and order of words, which are fundamental to human language comprehension.

For example, consider the two phrases
"The movie is good, but the actor was bad."
and
"The movie is bad, but the actor was good."

If we simply count individual words, both phrases are identical because they contain the exact same words! However, their meanings are diametrically opposed. The order of words and the context in which they are used are important.

17.1 N-grams

N-grams are continuous sequences of n items in a given sample of text or speech. In the context of text analysis, an item can be a character, a syllable, or a word, although words are the most commonly used items. The integer *n* in "n-gram" refers to the number of items in the sequence, so a bigram (or 2-gram) is a sequence of two words, a trigram (3-gram) is a sequence of three words, and so on.

To illustrate, consider the two sentences above. With 3-grams we could also get the pieces "movie is good", "movie is bad", "actor was bad", and "actor was good". Bigrams (or 2-grams) would not catch those differences. But they can also be very helpful in cases such as "don't like" vs "do like".

Now we will see how we can make use of such n-grams.

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
from sklearn.feature_extraction.text import TfidfVectorizer
```

17.2 N-grams in TF-IDF Vectors

When creating TF-IDF vectors, we can incorporate the concept of n-grams. The scikit-learn `TfidfVectorizer` provides the `ngram_range` parameter that allows us to specify the range of n-grams to include in the feature vectors.

17.2.1 Dataset - Madrid Restaurant Reviews

We will now use a large, text-based dataset containing more than 176.000 restaurant reviews from Madrid (see dataset on zenodo). The dataset (about 142MB) can be downloaded via the following code block.

```
File ..\datasets\madrid_reviews.csv already exists. Skipping download.
```

```
filename = "../datasets/madrid_reviews.csv"
data = pd.read_csv(filename)
data = data.drop(["Unnamed: 0"], axis=1)
data.head()
```

```
   parse_count       restaurant_name  rating_review    sample  \
0            1          Sushi_Yakuza              4  Positive
1           11  Azotea_Forus_Barcelo              1  Negative
2           12   Level_Veggie_Bistro              5  Positive
3           13  Sto_Globo_Sushi_Room              5  Positive
4           14  Azotea_Forus_Barcelo              5  Positive

          review_id                 title_review  \
0  review_731778139            Good sushi option
1  review_766657436  Light up your table at night
2  review_749493592                    Delicious
3  review_772422246            Loved this place
4  review_761855600     Amazing terrace in madrid

                                    review_preview  \
0  The menu of Yakuza is a bit of a lottery, some...
1  Check your bill when you cancel just in case y...
2  I had the yuca profiteroles and the veggie bur...
3  A friend recommended this place as one of the ...
4  Amazing terrace in madrid - great atmosphere a...

                                       review_full                date  \
0  The menu of Yakuza is a bit of a lottery, some...   December 10, 2019
1  Check your bill when you cancel just in case y...     August 23, 2020
2  I had the yuca profiteroles and the veggie bur...       March 6, 2020
3  A friend recommended this place as one of the ...  September 29, 2020
4  Amazing terrace in madrid - great atmosphere a...       July 27, 2020

     city                                    url_restaurant author_id
0  Madrid  https://www.tripadvisor.com/Restaurant_Review-...     UID_0
1  Madrid  https://www.tripadvisor.com/Restaurant_Review-...     UID_1
2  Madrid  https://www.tripadvisor.com/Restaurant_Review-...     UID_2
3  Madrid  https://www.tripadvisor.com/Restaurant_Review-...     UID_3
4  Madrid  https://www.tripadvisor.com/Restaurant_Review-...     UID_4
```

```
data.shape
```

```
(176848, 12)
```

```
data.review_full[2]
```

```
'I had the yuca profiteroles and the veggie burger, by recommendation of the␣
↪server. It was absolutely delicious and the service was outstanding. I will␣
↪definitely be back again with friends !'
```

```
from sklearn.feature_extraction.text import TfidfVectorizer

# considers both unigrams and bigrams
vectorizer = TfidfVectorizer(ngram_range=(1, 2))
tfidf_vectors = vectorizer.fit_transform(data.review_full)
tfidf_vectors.shape
```

```
(176848, 1496963)
```

This way, we are not just considering the frequency of individual words, but also the frequency of sequences of words, which often capture more meaning than the individual words alone.

But: this creates even bigger tfidf vectors!
Now we end up with vectors of about 1 million entries (even though most will be zero most of the time).

17.3 TF-IDF with Bigrams: Growing Vectors and Managing High Dimensionality

The power of n-grams comes at a cost. Specifically, as we increase the size of our n-grams, the dimensionality of our feature vectors also increases. In the case of bigrams, for every pair of words that occur together in our text corpus, we add a new dimension to our feature space. This can quickly lead to an explosion of features. For instance, a modest vocabulary of 1,000 words leads to a potential of 1,000,000 (1,000 x 1,000) bigrams.

This high dimensionality can lead to two issues:

1. **Sparsity:** Most documents in the corpus will not contain most of the possible bigrams, leading to a feature matrix where most values are zero, i.e., a sparse matrix.
2. **Computational resources:** The computational requirement for storing and processing these feature vectors can become significant, especially for large text corpora.

Several techniques can help manage this high-dimensionality problem:

- **Feature selection:** We can limit the number of bigrams we include in our feature vector. This could be done based on the frequency of the bigrams. For example, we could choose to include only those bigrams that occur more than a certain number of times in the corpus.
- **Dimensionality reduction:** Techniques such as Principal Component Analysis (PCA) or Truncated Singular Value Decomposition (TruncatedSVD) can be used to reduce the dimensionality of the feature space, while preserving as much of the variance in the data as possible.
- **Using Hashing Vectorizer:** Scikit-learn provides a `HashingVectorizer` that uses a hash function to map the features to indices in the feature vector. This approach has a constant memory footprint and does not require to keep a vocabulary dictionary in memory, which makes it suitable for large text corpora.

It's important to weigh the trade-offs between capturing more context using n-grams and managing the resulting high dimensionality.

Let us here use the simplest way to reduce the tfidf vector size: a more restrictive feature selection!

```
vectorizer = TfidfVectorizer(min_df=10, max_df=0.2,
                             ngram_range=(1, 2))
tfidf_vectors = vectorizer.fit_transform(data.review_full)
tfidf_vectors.shape
```

```
(176848, 119341)
```

This looks much better! Maybe we can even include 3-grams?

```
vectorizer = TfidfVectorizer(min_df=5, max_df=0.25,
                             ngram_range=(1, 3))
tfidf_vectors = vectorizer.fit_transform(data.review_full)
tfidf_vectors.shape
```

This looks OK, at least size-wise. The reason why this doesn't explode in terms of vector size is that the `min_df` parameter also counts for 2-grams, 3-grams etc. This here means that only the 3-grams which occur at least `min_df`-times will be kept.

Now we should check which ngrams the tfidf model finally included.

```
vectorizer.get_feature_names_out()[-100:]
```

```
array(['yr old', 'yrs', 'yrs old', 'yuca', 'yucca', 'yuck', 'yuk', 'yum',
       'yum and', 'yum great', 'yum soup', 'yum the', 'yum we', 'yum yum',
       'yumm', 'yummi', 'yummie', 'yummiest', 'yumminess', 'yummm',
       'yummmm', 'yummy', 'yummy all', 'yummy and', 'yummy as',
       'yummy but', 'yummy dishes', 'yummy food', 'yummy good',
       'yummy great', 'yummy if', 'yummy it', 'yummy my', 'yummy pizza',
       'yummy sangria', 'yummy tapas', 'yummy the', 'yummy they',
       'yummy too', 'yummy very', 'yummy we', 'yummy will', 'yummy would',
       'yummy yummy', 'yup', 'yuzu', 'zalacain', 'zalacain is',
       'zamburinas', 'zamburiñas', 'zara', 'zaragoza', 'zarzuela',
       'zealand', 'zen', 'zen market', 'zenith', 'zenith eggs', 'zerain',
       'zero', 'zero stars', 'zest', 'zesty', 'zing', 'zingy', 'zombie',
       'zone', 'zone of', 'zones', 'zoo', 'zucca', 'zucchini',
       'zucchini and', 'zucchini flowers', 'zucchini salad',
       'zucchini with', 'zumo', 'zumo de', 'zurbano', 'área', 'ôven',
       'único', '□□□', '□□', '□□□□', '□□', '□□□', '□□□□', '□□□□□', '□□',
       '□□□□', '□□□', '□□', '□□□', '□□□', '□□', '□□', '□□', '□□□□', '□□'],
      dtype=object)
```

```
vectorizer.get_feature_names_out()[-1]
```

```
'□□'
```

Well, that does not always immediately look like very good word combinations. We do see a lot of 2-grams and 3-grams. Most combinations of 2 or 3 words, however, seem to be grammatically wrong.

Why is that?

The reason is that our selection criteria (using `min_df` and `max_df`) removed a lot of very common words so that **yes it does** becomes **yes does**.

But we can leave it to the machine learning algorithms now to make more sense of it.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    data.review_full, data.rating_review, test_size=0.2, random_state=0)

print(f"Train dataset size: {X_train.shape}")
print(f"Test dataset size: {X_test.shape}")
```

```
Train dataset size: (141478,)
Test dataset size: (35370,)
```

```
vectorizer = TfidfVectorizer(min_df=10, max_df=0.2,
                             max_features=10000,
                             #ngram_range=(1, 2)
                            )
tfidf_vectors = vectorizer.fit_transform(X_train)
tfidf_vectors.shape
```

```
(141478, 13241)
```

```
vectorizer.get_feature_names_out()[-100:]
```

```
array(['yakisoba', 'yakitori', 'yakitoro', 'yam', 'yamil', 'yang', 'yard',
       'yards', 'yate', 'yay', 'yeah', 'year', 'yearly', 'years', 'yell',
       'yelled', 'yelling', 'yellow', 'yelp', 'yep', 'yerbabuena', 'yes',
       'yesterday', 'yet', 'yo', 'yoghurt', 'yogurt', 'yogurts', 'yolk',
       'yolks', 'york', 'yorker', 'young', 'younger', 'youngest',
       'youngish', 'youngsters', 'your', 'youre', 'yours', 'yourself',
       'yourselves', 'youth', 'youthful', 'youtube', 'yoy', 'yr', 'yrs',
       'yuca', 'yucca', 'yuck', 'yuk', 'yum', 'yumm', 'yummie',
       'yummiest', 'yumminess', 'yummm', 'yummmm', 'yummy', 'yup', 'yuzu',
       'zalacain', 'zamburinas', 'zamburiñas', 'zara', 'zaragoza',
       'zarzuela', 'zealand', 'zen', 'zenith', 'zerain', 'zero', 'zest',
       'zesty', 'zing', 'zingy', 'zombie', 'zone', 'zones', 'zoo',
       'zucca', 'zucchini', 'zumo', 'área', 'único', '□□□', '□□', '□□□□',
       '□□□', '□□□□', '□□□□□', '□□', '□□□□', '□□□', '□□', '□□', '□□',
       '□□□□', '□□'], dtype=object)
```

This time we will start right away with a classification model:

17.3.1 Logistic Regression model

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(max_iter=300)  # don't worry it also works without setting␣
↪max_iter
model.fit(tfidf_vectors, y_train)
```

```
C:\Users\flori\anaconda3\envs\ai_smart_health\lib\site-packages\sklearn\linear_
↪model\_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

```
LogisticRegression(max_iter=300)
```

```
tfidf_vectors_test = vectorizer.transform(X_test)
predictions = model.predict(tfidf_vectors_test)
```

```
np.round(predictions[:20], 1)
```

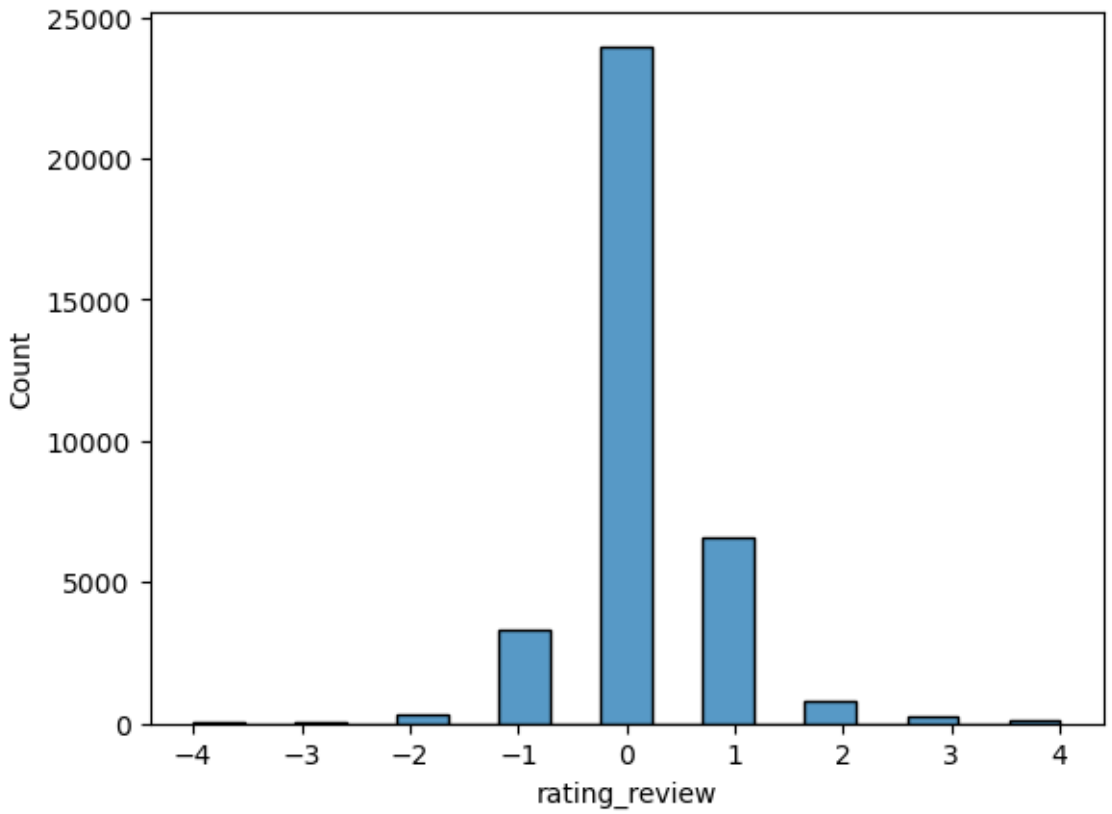
```
array([4, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 3, 5, 4, 5, 5],
      dtype=int64)
```

```
y_test[:20].values
```

```
array([5, 5, 4, 5, 3, 5, 5, 5, 5, 4, 5, 5, 4, 5, 5, 3, 5, 5, 5, 5],
      dtype=int64)
```

```
sb.histplot(predictions - y_test)
plt.xlabel("prediction error")
```

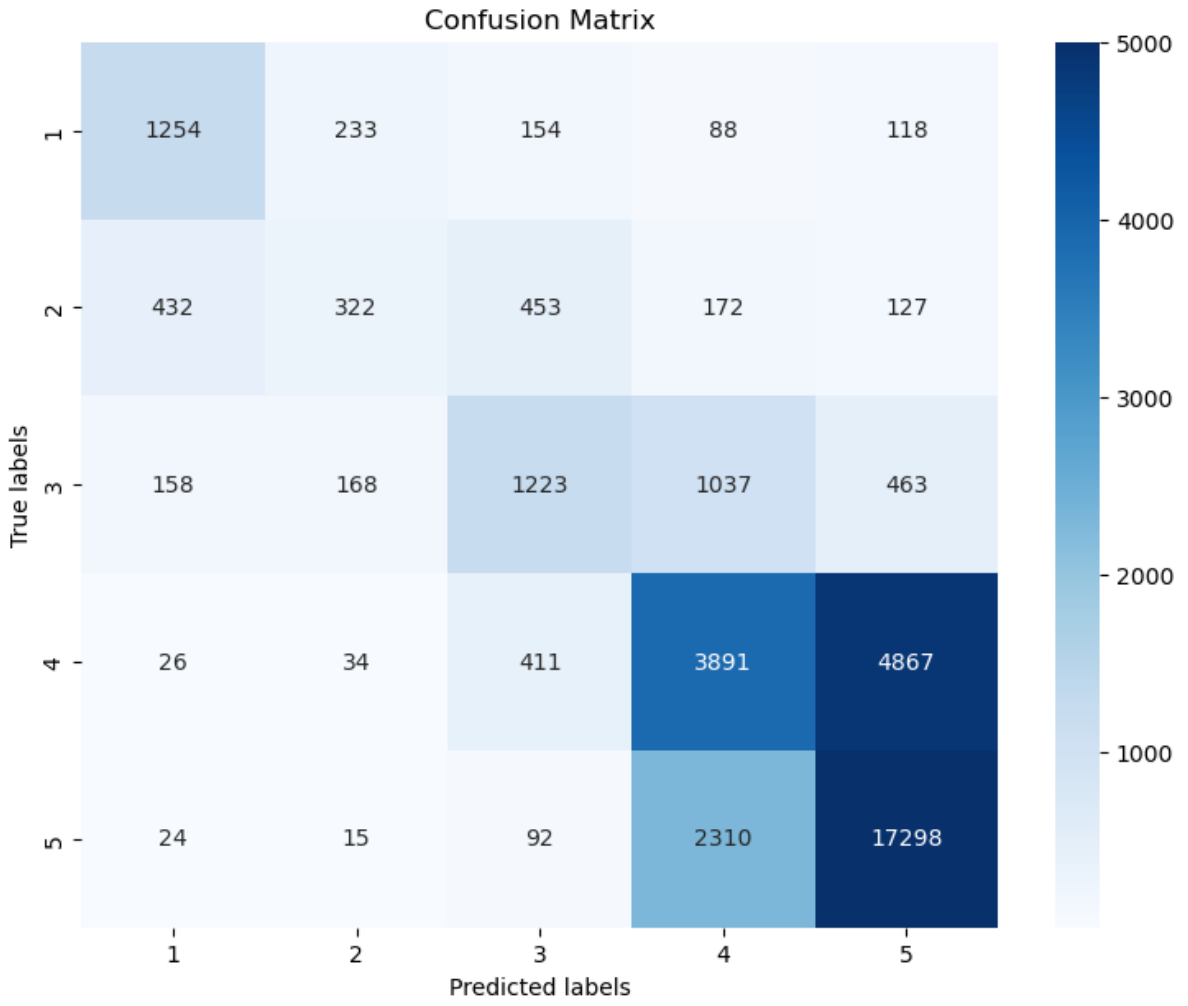
```
<Axes: xlabel='rating_review', ylabel='Count'>
```




```
# mean absolute error:
np.abs(predictions - y_test).mean()
```

```
0.3786542267458298
```

```
from sklearn.metrics import confusion_matrix, classification_report

cm = confusion_matrix(y_test, predictions, labels=model.classes_)

# Plotting the confusion matrix with a heatmap
plt.figure(figsize=(9,7))
sb.heatmap(cm, annot=True, fmt='d',
           cmap='Blues',
           xticklabels=model.classes_,
           yticklabels=model.classes_,
           vmax=5000
          )
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
plt.show()
```




```
vectorizer = TfidfVectorizer(min_df=10, max_df=0.2,
                             max_features=10000,
                             ngram_range=(1, 3))
tfidf_vectors = vectorizer.fit_transform(X_train)
tfidf_vectors.shape
```

```
(141478, 10000)
```

```
vectorizer.get_feature_names_out()[-100:]
```

```
array(['you are not', 'you ask', 'you ask for', 'you can', 'you can also',
       'you can buy', 'you can choose', 'you can eat', 'you can enjoy',
       'you can find', 'you can get', 'you can go', 'you can have',
       'you can order', 'you can see', 'you can sit', 'you can try',
       'you cannot', 'you choose', 'you come', 'you could', 'you do',
       'you do not', 'you don', 'you don want', 'you eat', 'you enjoy',
       'you enter', 'you expect', 'you feel', 'you feel like', 'you find',
       'you for', 'you get', 'you get the', 'you go', 'you go to',
       'you have', 'you have the', 'you have to', 'you in', 'you just',
```

(continues on next page)

(continued from previous page)

```
       'you know', 'you like', 'you ll', 'you ll be', 'you ll find',
       'you love', 'you make', 'you may', 'you might', 'you must',
       'you must try', 'you need', 'you need to', 'you order', 'you pay',
       'you pay for', 'you re', 'you re in', 'you re looking',
       'you re not', 'you really', 'you see', 'you should', 'you sit',
       'you the', 'you to', 'you try', 'you ve', 'you visit', 'you walk',
       'you want', 'you want to', 'you were', 'you will', 'you will be',
       'you will find', 'you will have', 'you will not', 'you with',
       'you won', 'you won be', 'you won regret', 'you would',
       'you would expect', 'young', 'your', 'your food', 'your meal',
       'your money', 'your mouth', 'your own', 'your place', 'your table',
       'your time', 'your way', 'yourself', 'yum', 'yummy'], dtype=object)
```

Question!

Why did we now get a notably smaller tfidf vector length?

```
tfidf_vectors[0, :].data
```

```
array([0.1387891 , 0.13512541, 0.14732539, 0.11708958, 0.14946676,
       0.11511072, 0.1444836 , 0.13665878, 0.14979188, 0.10268665,
       0.08515603, 0.12004832, 0.10720641, 0.11965181, 0.12394399,
       0.092254  , 0.14979188, 0.11356895, 0.10914472, 0.14331753,
       0.09882652, 0.10212461, 0.09038474, 0.09779526, 0.09394805,
       0.14525597, 0.12540266, 0.13598151, 0.12847323, 0.14875365,
       0.11584212, 0.11375017, 0.11152386, 0.12644517, 0.1377188 ,
       0.07982853, 0.05498834, 0.13392387, 0.11242216, 0.14367861,
       0.1499564 , 0.14313935, 0.1202977 , 0.07495173, 0.14612707,
       0.11327957, 0.13124683, 0.056673  , 0.05969443, 0.1270542 ,
       0.08700208, 0.14599061, 0.07204574, 0.1137082 , 0.08188261,
       0.10429517, 0.08306113, 0.07947322, 0.07596308, 0.12319629,
       0.09291383, 0.09143116, 0.10948931, 0.178938  , 0.12462447,
       0.09796248, 0.10013418, 0.11864844, 0.11132398, 0.08634848,
       0.09937276, 0.09949771, 0.08454477, 0.05570801, 0.09212747])
```

```
tfidf_vectors[0, :].indices
```

```
array([8292, 3801, 7590, 9155, 9260, 4170, 9254, 3032, 4240, 9844, 6542,
       6429, 8291, 7868, 6832, 3800, 9342, 3792, 2735, 4848, 7587, 5121,
       2055, 9154, 1465,  938, 6612, 9259, 3320, 1125, 5068, 9866,  704,
       8424, 7817, 9058, 4141, 6406, 9252, 5394, 1433, 5585, 7626, 3029,
       4452, 1015, 9813, 5396, 4223, 4715, 6426, 5932, 2410, 9826, 6831,
       9340, 2511, 4845, 2052, 4822, 4333, 6610, 1111, 9843, 8443, 8423,
       8523, 4263,  212, 2208, 6403, 5392, 5583, 3019, 4449])
```

```
example_vector = pd.DataFrame({
    "word": vectorizer.get_feature_names_out()[tfidf_vectors[0, :].indices],
    "tfidf": tfidf_vectors[0, :].data
})
example_vector
```

```
                  word     tfidf
0          to say that  0.138789
1         in spain and  0.135125
2   the most expensive  0.147325
3           we did not  0.117090
4       we should have  0.149467
..                 ...       ...
70                 sat  0.099373
71                once  0.099498
72             outside  0.084545
73                from  0.055708
74              looked  0.092127

[75 rows x 2 columns]
```

This time we will start right away with a classification model:

17.3.2 Logistic Regression model

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(max_iter=300)  # don't worry it also works without setting␣
↪max_iter
model.fit(tfidf_vectors, y_train)
```

```
C:\Users\flori\anaconda3\envs\ai_smart_health\lib\site-packages\sklearn\linear_
↪model\_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

```
LogisticRegression(max_iter=300)
```

```
tfidf_vectors_test = vectorizer.transform(X_test)
predictions = model.predict(tfidf_vectors_test)
```

```
np.round(predictions[:20], 1)
```

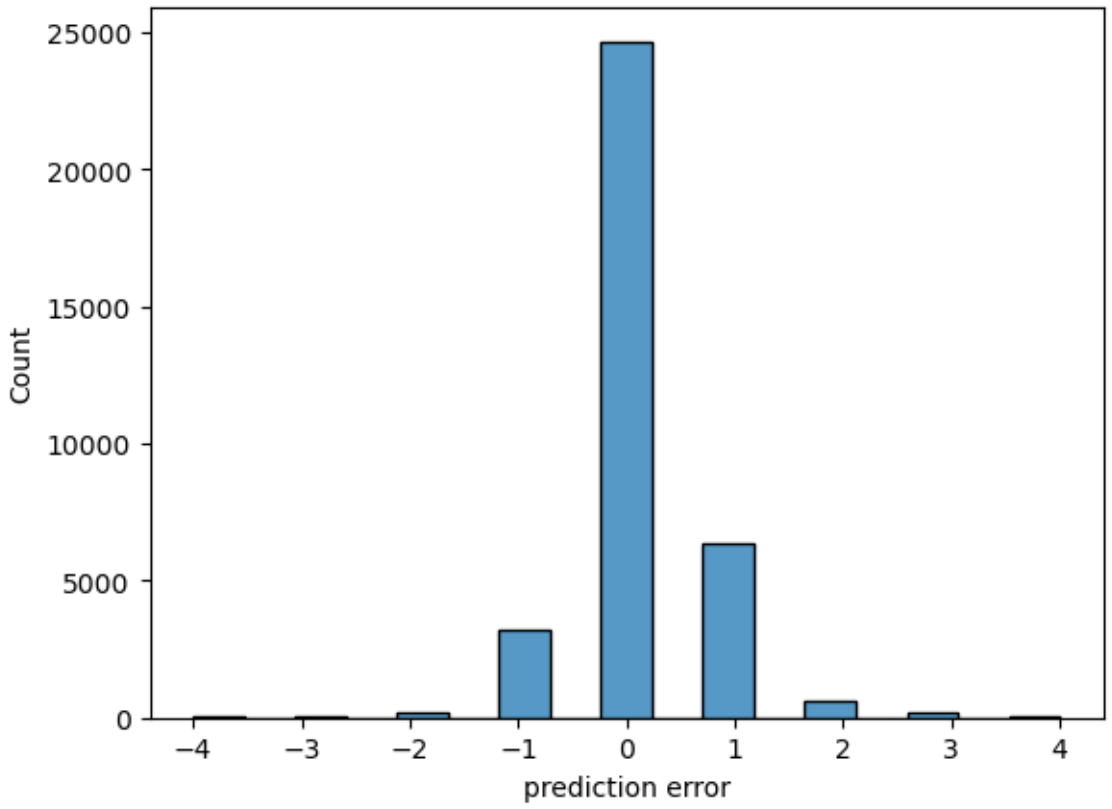
```
array([4, 5, 4, 5, 4, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 3, 5, 5, 5, 5],
      dtype=int64)
```

```
y_test[:20].values
```

```
array([5, 5, 4, 5, 3, 5, 5, 5, 5, 4, 5, 5, 4, 5, 5, 3, 5, 5, 5, 5],
      dtype=int64)
```

```
sb.histplot(predictions - y_test)
plt.xlabel("prediction error")
```

```
Text(0.5, 0, 'prediction error')
```




```
# mean absolute error:
np.abs(predictions - y_test).mean()
```

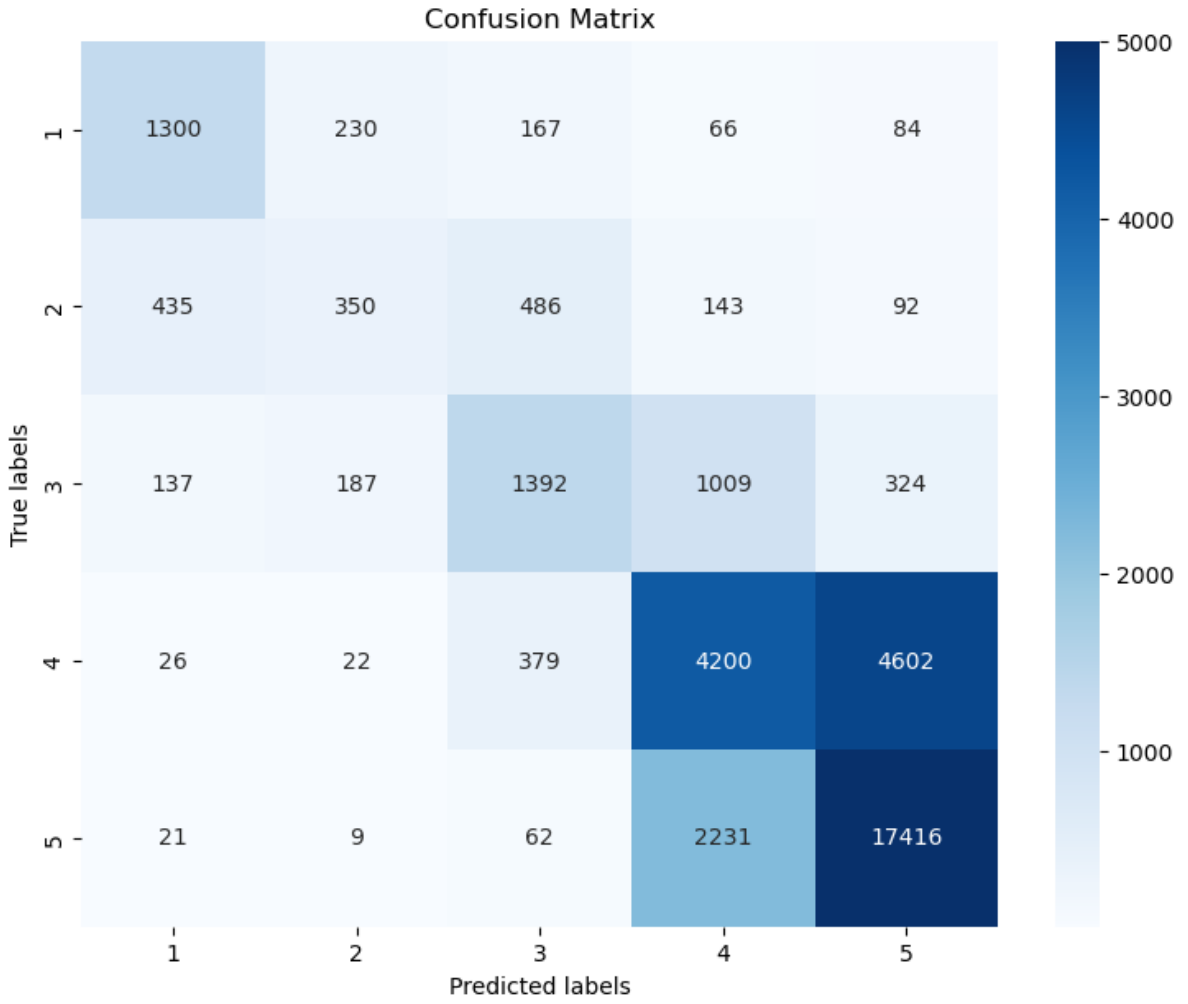
```
0.34684761096974837
```

```
from sklearn.metrics import confusion_matrix, classification_report

cm = confusion_matrix(y_test, predictions, labels=model.classes_)

# Plotting the confusion matrix with a heatmap
plt.figure(figsize=(9,7))
sb.heatmap(cm, annot=True, fmt='d',
           cmap='Blues',
           xticklabels=model.classes_,
           yticklabels=model.classes_,
           vmax=5000
          )
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



17.3.3 Did the 2-grams and 3-grams help?

Well, the prediction accuracy only got slightly better. So, it seems to have *some* effect, but nothing spectacular. However, this is not a general finding and might look very differently for other datasets or problems.

We can now also look at the ngrams that have the largest impact on the model predictions:

```
ngrams = pd.DataFrame({"ngram": vectorizer.get_feature_names_out(),
                       "weight": model.coef_[0]
                      })
ngrams.sort_values("weight")
```

```
          ngram    weight
2008  delicious -5.679471
2477  excellent -5.174349
8694  very good -4.929491
```

(continues on next page)

(continued from previous page)

```
7185      tasty -3.620315
8954   was good -3.569174
...          ...       ...
3619   horrible  5.512549
1015      avoid  5.905717
6344       rude  5.934622
9826      worst  6.540459
7215   terrible  7.033302

[10000 rows x 2 columns]
```

Here, too, we find only very few 2-grams in the top-20 and bottom-20 lists. Most of the times, the model still seems to judge the reviews based on individual words.

```
ngrams.sort_values("weight").head(20)
```

```
          ngram    weight
2008   delicious -5.679471
2477   excellent -5.174349
8694   very good -4.929491
7185       tasty -3.620315
8954    was good -3.569174
2987    friendly -3.554020
1272         bit -3.392447
248      amazing -3.362312
7327    the best -3.300674
4991        nice -3.179749
967   atmosphere -2.883584
5066     not bad -2.815060
1215        best -2.774438
4385      little -2.698092
8959   was great -2.671645
5714     perfect -2.630462
4486       loved -2.577662
6117  reasonable -2.553251
2574   fantastic -2.466228
4492      lovely -2.462175
```

```
from sklearn.metrics import confusion_matrix, classification_report

print(confusion_matrix(y_test, predictions))
print(classification_report(y_test, predictions))
```

```
[[ 1300   230   167    66    84]
 [  435   350   486   143    92]
 [  137   187  1392  1009   324]
 [   26    22   379  4200  4602]
 [   21     9    62  2231 17416]]
              precision    recall  f1-score   support

           1       0.68      0.70      0.69      1847
           2       0.44      0.23      0.30      1506
           3       0.56      0.46      0.50      3049
           4       0.55      0.46      0.50      9229
```

(continues on next page)

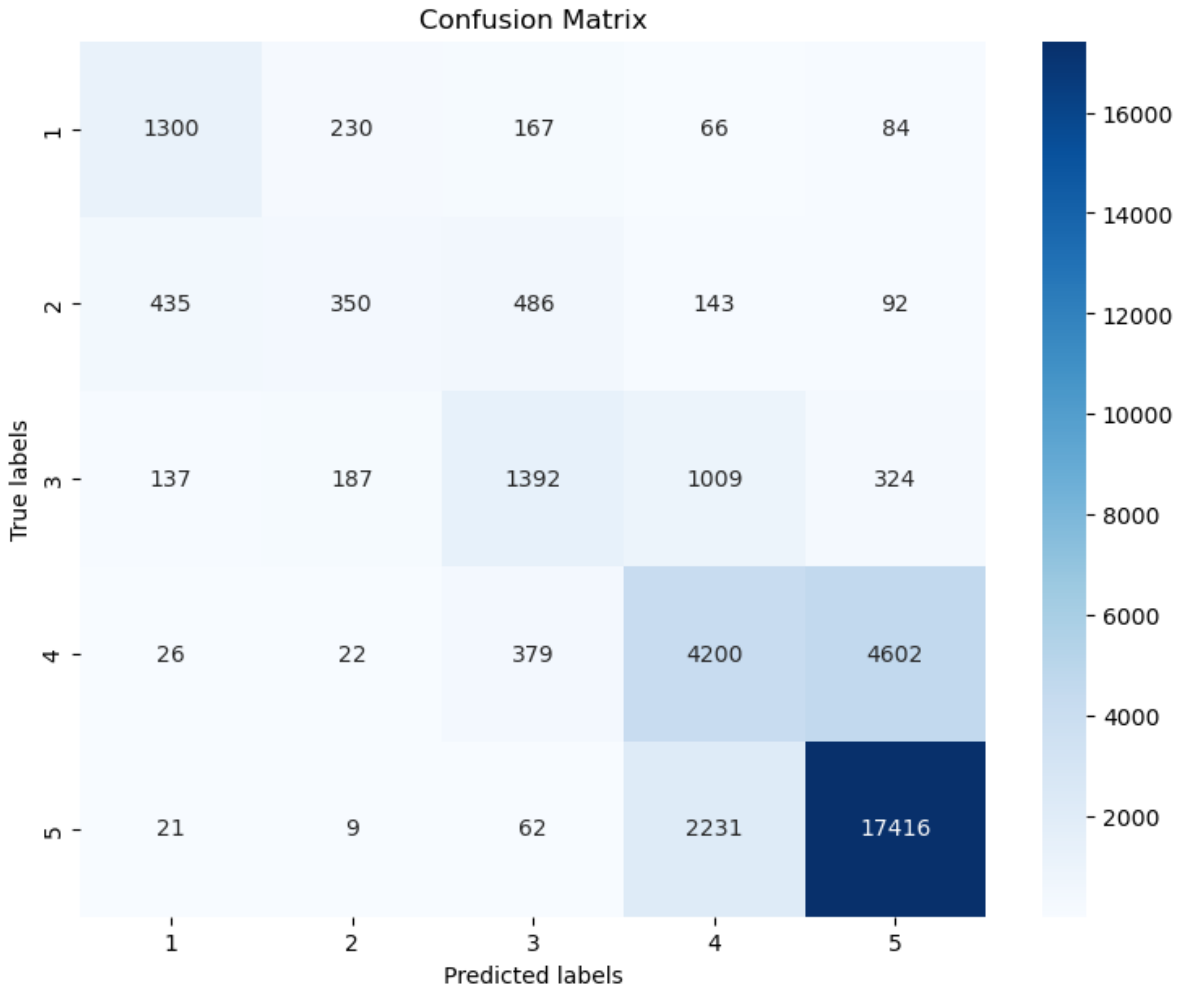
(continued from previous page)

```
           5       0.77      0.88      0.82     19739

    accuracy                           0.70     35370
   macro avg       0.60      0.55      0.56     35370
weighted avg       0.68      0.70      0.68     35370
```

17.3.4 Confusion matrix

The confusion matrix can tell us a lot about where the model works well and where it fails. Often is is more accessible if the matrix is plotted, for instance using seaborns `heatmap`.

```
cm = confusion_matrix(y_test, predictions, labels=model.classes_)

# Plotting the confusion matrix with a heatmap
plt.figure(figsize=(9,7))
sb.heatmap(cm, annot=True, fmt='d',
           cmap='Blues',
           xticklabels=model.classes_,
           yticklabels=model.classes_)
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
plt.show()
```



17.4 Find similar documents with tfidf

So far, we used the tfidf-vectors as feature vectors to train machine learning models. As we just saw, this works very well to predict review rating or to classify documents as positive/negative (=sentiment analysis).

But there is more we can do with tfidf vectors. Why not use the vectors to compute distances or similarities? This way, we can search for the most similar documents in a corpus!

```
vectorizer = TfidfVectorizer(min_df=10, max_df=0.2,
                             max_features=50000,
                             ngram_range=(1, 3))
tfidf_vectors = vectorizer.fit_transform(X_train)
tfidf_vectors.shape
```

```
(141478, 50000)
```

```
tfidf_vectors.shape
```

```
(141478, 50000)
```

```
X_train.shape
```

```
(141478,)
```

17.4.1 Compare one vector to all other vectors

Even though we here deal with very large vectors, computing similarities or angles between these vectors is compuationally very efficient. This means, we can simply compare a the tfidf vector of a given text to all > 140,000 documents in virtually no time!

In order for this to work, however, we should not rely on for-loops. Those are inherently slow in Python. We rather use optimized functions for this such as from `sklear.metrics.pairwise`.

```
from sklearn.metrics.pairwise import cosine_similarity

review_id = -11#-9#-2
query_vector = tfidf_vectors[review_id, :]

cosine_similarities = cosine_similarity(query_vector, tfidf_vectors).flatten()
cosine_similarities.shape
```

```
(141478,)
```

```
np.sort(cosine_similarities)[::-1]
```

```
array([1.        , 0.36692149, 0.31812401, ..., 0.        , 0.        ,
       0.        ])
```

```
np.argsort(cosine_similarities)[::-1]
```

```
array([141467,  94876,  91939, ...,  95683,  43090, 104856], dtype=int64)
```

```
top5_idx = np.argsort(cosine_similarities)[::-1][1:6]
top5_idx
```

```
array([ 94876,  91939, 103516,  47460,  51098], dtype=int64)
```

Let us now look at the results of our search by displaying the top-5 most similar documents (according to the cosine score on the tfidf-vectors). This usually doesn't work perfectly, but it does work to quite some extent. Try it out yourself and have a look at what documents this finds for you!

```
print("\n****Original document:****")
print(X_train.iloc[review_id])

for i in top5_idx:
```

(continues on next page)

(continued from previous page)

```
    print(f"\n----Document with similarity {cosine_similarities[i]:.3f}:----")
    print(X_train.iloc[i])
```

```
****Original document:****
The food was great, the staff were friendly. The place was not crowded. The␣
 ↪location was very convenient, while hidden from the hustle and bustle.

----Document with similarity 0.367:----
Takes you away from the hustle and bustle of the city ! Lovely setting.Recommend␣
 ↪the sangria cocktails.

----Document with similarity 0.318:----
Cute little spot away from the hustle and bustle. We enjoyed the most delicious␣
 ↪little cake (Tigre chocolate) and hot chocolate. Friendly service as well!

----Document with similarity 0.309:----
Stopped off for a quick drink, great little place away from the hustle and bustle.␣
 ↪Tasty complimentary tortilla, didn't have a meal, but food looked great Would␣
 ↪come back.

----Document with similarity 0.307:----
This taberna is nestled away from the hustle and bustle, but close to Palacio Real.
 ↪ Very authentic and local. Go hungry, cocido madrileño is not for the faint of␣
 ↪heart.

----Document with similarity 0.298:----
Perfect dinner spot in La Latina, a little ways away from the hustle and bustle of␣
 ↪this barrio. The service was above par and the food delicious. Will definitely␣
 ↪be back!
```

17.5 Word Vectors: Word2Vec and Co

Tfidf vectors are a rather basic, but still often used, technique. Arguably, this is because they are based on relatively simple statistics and easy to compute. They typically do a good job in weighing words according to their importance in a larger corpus and allow us to ignore words with low *distriminative power* (for instance so-called *stopwords* such as "a", "the", "that", …).

With **n-grams** we can even go one step further and also count sentence pieces longer than one word, which allows to also take some grammar or negations into account. The price, however, is that we have to restrict the number of n-grams to avoid exploding vector sizes.

While TF-IDF vectors and n-grams serve as powerful techniques to represent and manipulate text data, they have limitations. These methods essentially treat words as individual, isolated units, devoid of any context or relation to other words. In other words, they lack the ability to capture the semantic meanings of words and the linguistic context in which they are used.

Take these two sentences as an example:

(1) *The customer likes cake with a cappuccino.*
(2) *The client loves to have a cookie and a coffee.*

We will immediately indentify that both sentences speak of very similar things. But if you look at the words in both sentences you will realize that only *"The"* and *"a"* are found in both. And, as we have seen in the tfidf-part, such words tell

very little about the sentence content. All other words, however, only occur in one or the other sentence. Tfidf-vectors would compute a zero similarity here.

This is where we come to **word vectors**. Word vectors, also known as **word embeddings**, are mathematical representations of words in a high-dimensional space where the semantic similarity between words corresponds to the geometric distance in the embedding space. Simply put, similar words are close together, and dissimilar words are farther apart. If done well, this should show that *"cookie"* and *"cake"* are not the same word, but mean something very related.

The most prominent example of such a technique is Word2Vec [Mikolov *et al.*, 2013][Mikolov *et al.*, 2013].

```
#!pip install gensim
```

```
import nltk

tokenizer = nltk.tokenize.TreebankWordTokenizer()
stemmer = nltk.stem.WordNetLemmatizer()

def process_document(doc):
    """Convert document to lemmas."""
    tokens = tokenizer.tokenize(doc)
    tokens = [x.strip(".,;:!? ") for x in tokens]
    return [stemmer.lemmatize(w) for w in tokens]
```

```
from tqdm.notebook import tqdm

sentences = [process_document(doc) for doc in tqdm(X_train.values)]
```

```
  0%|          | 0/141478 [00:00<?, ?it/s]
```

```
len(sentences)
```

```
141478
```

We will now train our own Word2Vec model using `Gensim`, see also documentation.

```
from gensim.models import Word2Vec

# Assume 'sentences' is a list of lists of tokenized sentences
model = Word2Vec(sentences,
                 vector_size=200,
                 window=5,
                 min_count=2,
                 workers=4)
```

```
model.save("word2vec_madrid_reviews.model")
```

```
vector = model.wv['delicious']  # get numpy vector of a word
```

```
vector
```

```
array([-2.03538680e+00,  2.52757788e-01, -6.78372085e-01, -1.26871383e+00,
       -1.08166724e-01,  8.22216213e-01,  3.38415295e-01,  7.98148632e-01,
       -9.87190962e-01, -9.55492854e-01,  4.07470644e-01,  1.39118874e+00,
        1.67936873e+00, -1.62421167e-01, -1.18595815e+00,  2.57541704e+00,
       -5.80342531e-01,  1.33421934e+00,  2.37748718e+00, -1.43586814e+00,
       -1.04479599e+00,  6.39563650e-02, -8.97780657e-01,  6.41878784e-01,
       -1.74520206e+00,  4.68130469e-01,  8.02812040e-01, -9.29517210e-01,
       -7.74732411e-01, -8.87739718e-01, -5.06934702e-01, -3.21538508e-01,
       -4.80596572e-02, -1.68594670e+00, -2.41771966e-01, -4.61565822e-01,
       -2.03054279e-01,  9.59063053e-01, -4.65506434e-01,  1.38559628e+00,
       -4.67663795e-01, -5.63467026e-01, -1.44417775e+00, -1.22851729e+00,
        1.17849803e+00, -1.26880860e+00, -5.07019520e-01,  3.69102329e-01,
        2.92213416e+00, -4.93991584e-01, -6.48129582e-01,  1.70414066e+00,
       -1.23612189e+00, -1.72287583e+00,  1.15597653e+00,  1.44868660e+00,
       -1.22250819e+00,  2.37475419e+00,  6.53695226e-01,  2.05163908e+00,
       -3.08574414e+00,  1.66374397e+00, -2.80261766e-02,  6.86524808e-01,
        1.53102946e+00,  6.65094972e-01, -5.36559187e-02, -7.84830332e-01,
        7.35198379e-01,  1.32039881e+00,  7.09394872e-01, -1.21944356e+00,
        8.04880083e-01, -2.76637882e-01, -6.61925673e-01, -1.25887072e+00,
        1.19793057e+00,  4.21494067e-01,  7.66302586e-01,  9.96498585e-01,
        5.08949719e-02,  1.45889115e+00,  5.01284957e-01, -7.05735385e-01,
        1.15695286e+00, -2.63115740e+00,  6.68963552e-01, -1.25631428e+00,
        8.86810184e-01,  6.45234525e-01,  1.62897801e+00,  6.84916005e-02,
        1.24690199e+00, -4.46591347e-01, -1.15185416e+00,  5.72938859e-01,
        1.25214541e+00, -1.03020048e+00,  1.85466862e+00,  2.71878541e-01,
        6.15206361e-01,  1.77544081e+00,  1.54688907e+00, -3.51346731e+00,
        2.00846910e+00,  2.17208838e+00, -5.17885029e-01, -5.46077728e-01,
       -2.82651377e+00, -8.21791515e-02,  2.36416310e-01, -6.74597442e-01,
       -8.51826787e-01, -3.59785527e-01, -1.56974423e+00,  9.27218020e-01,
       -2.02717376e+00,  1.95397782e+00,  7.02663004e-01, -1.02236593e+00,
        6.56838715e-01,  6.73914135e-01, -3.00290734e-01, -9.34027076e-01,
       -8.00638616e-01, -1.32641673e-01, -7.23512411e-01, -1.51655793e+00,
       -8.64614844e-01,  7.01261044e-01,  1.04395604e+00,  1.31523907e+00,
        8.43236089e-01, -2.57687283e+00, -1.49496460e+00, -4.15781051e-01,
       -5.65135837e-01,  2.03486538e+00,  1.17266095e+00, -2.50355184e-01,
        1.85218894e+00,  5.07799566e-01,  1.23914981e+00, -4.94150430e-01,
        2.27533412e+00, -2.61036470e-03, -3.87329668e-01, -1.11296475e-01,
       -1.18795252e+00,  3.70148182e-01,  1.59401250e+00, -1.83947122e+00,
       -2.73712277e+00,  4.71669245e+00,  1.95886984e-01, -4.84028697e-01,
        9.64134216e-01,  2.53249586e-01, -2.59700298e+00, -2.38229573e-01,
       -1.69923174e+00,  3.20591122e-01, -2.14936686e+00, -5.11169016e-01,
        7.22345829e-01,  1.10016990e+00,  2.35557246e+00, -7.25124359e-01,
       -1.31868494e+00, -6.81534111e-01, -4.91266608e-01,  1.32315159e+00,
        9.05143797e-01,  9.32789594e-02, -6.45776391e-01, -1.09242570e+00,
       -3.32741797e-01, -2.71128982e-01, -1.77400219e+00,  1.17381942e+00,
       -3.83675754e-01, -1.02986455e+00, -2.15794826e+00, -9.86504018e-01,
        1.52942610e+00,  7.36541212e-01, -3.30162406e+00,  1.44280720e+00,
       -1.08328497e+00,  5.31407893e-02, -1.49774420e+00, -2.06774545e+00,
       -1.72131769e-02, -1.64670253e+00,  1.24993122e+00, -2.00452232e+00,
        5.38596213e-01, -4.72304463e-01,  1.02590454e+00,  7.42713094e-01],
      dtype=float32)
```

```
model.wv.most_similar('delicious', topn=10)
```

```
[('tasty', 0.8175682425498962),
```

(continues on next page)

(continued from previous page)

```
 ('delicious.', 0.7884759306907654),
 ('yummy', 0.7805776000022888),
 ('amazing', 0.7275819778442383),
 ('fantastic', 0.7188233137130737),
 ('divine', 0.7097207903862),
 ('exquisite', 0.6883265376091003),
 ('superb', 0.6864668130874634),
 ('incredible', 0.6731298565864563),
 ('phenomenal', 0.6625100374221802)]
```

```
model.wv.most_similar('pizza', topn=10)
```

```
[('Pizza', 0.687938392162323),
 ('hamburger', 0.6807577610015869),
 ('burger', 0.6770654916763306),
 ('pasta', 0.6611385345458984),
 ('carbonara', 0.6239833831787109),
 ('burrito', 0.589520275592804),
 ('pizza.', 0.5812183022499084),
 ('margarita', 0.5620388388633728),
 ('crust', 0.5615034699440002),
 ('calzone', 0.550714910030365)]
```

```
model.wv.most_similar('horrible', topn=10)
```

```
[('terrible', 0.8658505082130432),
 ('awful', 0.8121690154075623),
 ('bad', 0.7098363041877747),
 ('appalling', 0.6666131615638733),
 ('disgusting', 0.652198851108551),
 ('poor', 0.6406951546669006),
 ('shocking', 0.6216946840286255),
 ('okay', 0.6188369989395142),
 ('dreadful', 0.6177916526794434),
 ('Terrible', 0.6082724928855896)]
```

```
model.wv.most_similar('friendly', topn=10)
```

```
[('friendly.', 0.764923095703125),
 ('courteous', 0.7515475153923035),
 ('polite', 0.7410220503807068),
 ('welcoming', 0.7141723036766052),
 ('cordial', 0.7002096176147461),
 ('attentive', 0.6988195180892944),
 ('personable', 0.6342913508415222),
 ('gracious', 0.6272123456001282),
 ('professional', 0.6221848726272583),
 ('freindly', 0.6203247308731079)]
```

```
model.wv.most_similar('chocolate', topn=10)
```

```
[('brownie', 0.7869287729263306),
 ('cheesecake', 0.7738447189331055),
 ('carrot', 0.7625924348831177),
 ('caramel', 0.7542667388916016),
 ('crepe', 0.742887020111084),
 ('flan', 0.7408572435379028),
 ('custard', 0.7393072843551636),
 ('tiramisu', 0.7378472685813904),
 ('icecream', 0.7375630736351013),
 ('tart', 0.7370063662528992)]
```

```
model.wv.most_similar('coffee', topn=10)
```

```
[('tea', 0.7526535987854004),
 ('croissant', 0.7297593355178833),
 ('cappuccino', 0.7134137153625488),
 ('coffee.', 0.6472688913345337),
 ('churros', 0.6273057460784912),
 ('coffe', 0.6237097382545471),
 ('juice', 0.6207398176193237),
 ('breakfast', 0.6145669221878052),
 ('latte', 0.6066522002220154),
 ('espresso', 0.5939613580703735)]
```

17.6 Alternative short-cuts

Training your own Word2Vec model is fun and sometimes also really helpful. Here it is quite OK for instance, because we have a relatively big text corpus (> 140,000 documents) with a clear general topic focus on restaurants and food.

Often, however, you simply may want to use a model that covers a language more broadly. Instead of training your own model on a much much bigger corpus, we can simply use a model that was trained already, see for instance here on the Gensim website.

Another way is to use **SpaCy**. Its larger language models already contain word embeddings!

```
# Comment out and run the following to first download a large english model
#!python -m spacy download en_core_web_lg
```

```
import spacy

nlp = spacy.load("en_core_web_lg")
```

As we have seen before, SpaCy converts the text into tokens, but also does much more. We can look at different attributes of the tokens to extract the computed information. For instance:

- `.text`: The original token text.
- `has_vector`: Does the token have a vector representation?
- `.vector_norm`: The L2 norm of the token's vector (the square root of the sum of the values squared)
- `.is_oov`: Out-of-vocabulary

```
tokens = nlp("dog cat banana afskfsd")
for token in tokens:
    print(token.text, token.has_vector, token.vector_norm, token.is_oov)
```

```
dog True 75.254234 False
cat True 63.188496 False
banana True 31.620354 False
afskfsd False 0.0 True
```

```
tokens[0].vector
```

```
array([ 1.2330e+00,  4.2963e+00, -7.9738e+00, -1.0121e+01,  1.8207e+00,
        1.4098e+00, -4.5180e+00, -5.2261e+00, -2.9157e-01,  9.5234e-01,
        6.9880e+00,  5.0637e+00, -5.5726e-03,  3.3395e+00,  6.4596e+00,
       -6.3742e+00,  3.9045e-02, -3.9855e+00,  1.2085e+00, -1.3186e+00,
       -4.8886e+00,  3.7066e+00, -2.8281e+00, -3.5447e+00,  7.6888e-01,
        1.5016e+00, -4.3632e+00,  8.6480e+00, -5.9286e+00, -1.3055e+00,
        8.3870e-01,  9.0137e-01, -1.7843e+00, -1.0148e+00,  2.7300e+00,
       -6.9039e+00,  8.0413e-01,  7.4880e+00,  6.1078e+00, -4.2130e+00,
       -1.5384e-01, -5.4995e+00,  1.0896e+01,  3.9278e+00, -1.3601e-01,
        7.7732e-02,  3.2218e+00, -5.8777e+00,  6.1359e-01, -2.4287e+00,
        6.2820e+00,  1.3461e+01,  4.3236e+00,  2.4266e+00, -2.6512e+00,
        1.1577e+00,  5.0848e+00, -1.7058e+00,  3.3824e+00,  3.2850e+00,
        1.0969e+00, -8.3711e+00, -1.5554e+00,  2.0296e+00, -2.6796e+00,
       -6.9195e+00, -2.3386e+00, -1.9916e+00, -3.0450e+00,  2.4890e+00,
        7.3247e+00,  1.3364e+00,  2.3828e-01,  8.4388e-02,  3.1480e+00,
       -1.1128e+00, -3.5598e+00, -1.2115e-01, -2.0357e+00, -3.2731e+00,
       -7.7205e+00,  4.0948e+00, -2.0732e+00,  2.0833e+00, -2.2803e+00,
       -4.9850e+00,  9.7667e+00,  6.1779e+00, -1.0352e+01, -2.2268e+00,
        2.5765e+00, -5.7440e+00,  5.5564e+00, -5.2735e+00,  3.0004e+00,
       -4.2512e+00, -1.5682e+00,  2.2698e+00,  1.0491e+00, -9.0486e+00,
        4.2936e+00,  1.8709e+00,  5.1985e+00, -1.3153e+00,  6.5224e+00,
        4.0113e-01, -1.2583e+01,  3.6534e+00, -2.0961e+00,  1.0022e+00,
       -1.7873e+00, -4.2555e+00,  7.7471e+00,  1.0173e+00,  3.1626e+00,
        2.3558e+00,  3.3589e-01, -4.4178e+00,  5.0584e+00, -2.4118e+00,
       -2.7445e+00,  3.4170e+00, -1.1574e+01, -2.6568e+00, -3.6933e+00,
       -2.0398e+00,  5.0976e+00,  6.5249e+00,  3.3573e+00,  9.5334e-01,
       -9.4430e-01, -9.4395e+00,  2.7867e+00, -1.7549e+00,  1.7287e+00,
        3.4942e+00, -1.6883e+00, -3.5771e+00, -1.9013e+00,  2.2239e+00,
       -5.4335e+00, -6.5724e+00, -6.7228e-01, -1.9748e+00, -3.1080e+00,
       -1.8570e+00,  9.9496e-01,  8.9135e-01, -4.4254e+00,  3.3125e-01,
        5.8815e+00,  1.9384e+00,  5.7294e-01, -2.8830e+00,  3.8087e+00,
       -1.3095e+00,  5.9208e+00,  3.3620e+00,  3.3571e+00, -3.8807e-01,
        9.0022e-01, -5.5742e+00, -4.2939e+00,  1.4992e+00, -4.7080e+00,
       -2.9402e+00, -1.2259e+00,  3.0980e-01,  1.8858e+00, -1.9867e+00,
       -2.3554e-01, -5.4535e-01, -2.1387e-01,  2.4797e+00,  5.9710e+00,
       -7.1249e+00,  1.6257e+00, -1.5241e+00,  7.5974e-01,  1.4312e+00,
        2.3641e+00, -3.5566e+00,  9.2066e-01,  4.4934e-01, -1.3233e+00,
        3.1733e+00, -4.7059e+00, -1.2090e+01, -3.9241e-01, -6.8457e-01,
       -3.6789e+00,  6.6279e+00, -2.9937e+00, -3.8361e+00,  1.3868e+00,
       -4.9002e+00, -2.4299e+00,  6.4312e+00,  2.5056e+00, -4.5080e+00,
       -5.1278e+00, -1.5585e+00, -3.0226e+00, -8.6811e-01, -1.1538e+00,
       -1.0022e+00, -9.1651e-01, -4.7810e-01, -1.6084e+00, -2.7307e+00,
        3.7080e+00,  7.7423e-01, -1.1085e+00, -6.8755e-01, -8.2901e+00,
        3.2405e+00, -1.6108e-01, -6.2837e-01, -5.5960e+00, -4.4865e+00,
```

(continues on next page)

(continued from previous page)

```
        4.0115e-01, -3.7063e+00, -2.1704e+00,  4.0789e+00, -1.7973e+00,
        8.9538e+00,  8.9421e-01, -4.8128e+00,  4.5367e+00, -3.2579e-01,
       -5.2344e+00, -3.9766e+00, -2.1979e+00,  3.5699e+00,  1.4982e+00,
        6.0972e+00, -1.9704e+00,  4.6522e+00, -3.7734e-01,  3.9101e-02,
        2.5361e+00, -1.8096e+00,  8.7035e+00, -8.6372e+00, -3.5257e+00,
        3.1034e+00,  3.2635e+00,  4.5437e+00, -5.7290e+00, -2.9141e-01,
       -2.0011e+00,  8.5328e+00, -4.5064e+00, -4.8276e+00, -1.1786e+01,
        3.5607e-01, -5.7115e+00,  6.3122e+00, -3.6650e+00,  3.3597e-01,
        2.5017e+00, -3.5025e+00, -3.7891e+00, -3.1343e+00, -1.4429e+00,
       -6.9119e+00, -2.6114e+00, -5.9757e-01,  3.7847e-01,  6.3187e+00,
        2.8965e+00, -2.5397e+00,  1.8022e+00,  3.5486e+00,  4.4721e+00,
       -4.8481e+00, -3.6252e+00,  4.0969e+00, -2.0081e+00, -2.0122e-01,
        2.5244e+00, -6.8817e-01,  6.7184e-01, -7.0466e+00,  1.6641e+00,
       -2.2308e+00, -3.8960e+00,  6.1320e+00, -8.0335e+00, -1.7130e+00,
        2.5688e+00, -5.2547e+00,  6.9845e+00,  2.7835e-01, -6.4554e+00,
       -2.1327e+00, -5.6515e+00,  1.1174e+01, -8.0568e+00,  5.7985e+00],
      dtype=float32)
```

```
nlp1 = nlp(X_train[0])
nlp2 = nlp(X_train[1])
```

```
nlp1.similarity(nlp1)
```

```
1.0
```

```
nlp1.similarity(nlp2)
```

```
0.732720161085499
```

```
nlp1
```

```
The menu of Yakuza is a bit of a lottery, some plates are really good (like most␣
↪of the sushi rolls) and instead some others are terrible ( the pizza sushi and␣
↪most of the fried starters). Taking this in consideration, it´s a great option␣
↪if you feel like sushi and can avoid ordering from the rest of the menu. We even␣
↪ordered for delivery more than once and the packaging they use is great.
```

```
nlp2
```

```
Check your bill when you cancel just in case you get an extra charge surprise on␣
↪it   There are no more words to describe my experience when I was there. Sad and␣
↪poor experience!! Never return! Never again
```

17.6.1 Limitations and Extensions

While Word2Vec is a powerful tool, it's not without limitations. The main issue with Word2Vec (and similar models that derive their semantics based on the local usage context) is that they assign one vector per word. This becomes a problem for words with multiple meanings based on their context (homonyms and polysemes). To tackle such limitations, extensions like FastText and advanced methods such as GloVe (Global Vectors for Word Representation) and transformers like BERT (Bidirectional Encoder Representations from Transformers) have been proposed.

17.6.2 FastText

FastText, also developed by Facebook, extends Word2Vec by treating each word as composed of character n-grams. So the vector for a word is made of the sum of these character n-grams. This allows the embeddings to capture the meaning of shorter words and suffixes/prefixes and understand new words once the character n-grams are learned.

17.6.3 GloVe

GloVe, developed by Stanford, is another method to create word embeddings. While Word2Vec is a predictive model — a model that predicts context given a word, GloVe is a count-based model. It leverages matrix factorization techniques on the word-word co-occurrence matrix.

17.6.4 Transformers: BERT & GPT

BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pretrained Transformer) models ushered in the era of transformers that not only consider local context but also take the entire sentence context into account to create word embeddings.

In conclusion, while TF-IDF and n-grams offer a solid start, word embeddings and transformers take the representation of words to the next level. By considering context and semantic meaning, they offer a more complete and robust method to work with text

Part VII

Look at the networks

CHAPTER

EIGHTEEN

NETWORKS / GRAPH THEORY

Graph theory, or network analysis, is a crucial part of the data science toolkit. It offers a perspective that significantly deviates from the classic tabular or relational data models, enabling us to model, understand, and navigate complex systems of interaction and connection.

18.1 Why Do We Need Graphs for Data Science?

Consider the variety of real-world systems that can be understood as networks or graphs:

- **Information Technology Networks:** These include the internet (a vast network of computers and servers), telephone networks, and local computer networks. Each node represents a device or server, and each edge is a communication link.
- **Transportation Networks:** The systems that connect locations, like road networks, railway networks, waterways, and even networks for utilities like water, gas, and electricity. Here, nodes could represent stations, intersections, or locations, and edges the routes or pathways between them.
- **Social Networks:** And this does not just refer to social media! It could include networks of friendships, relationships, professional contacts, memberships in clubs or organizations, and more. Nodes represent individuals or entities, and edges represent relationships or interactions between them.
- **Biological Networks:** Within biology, networks can represent protein interactions, gene regulation, neural connections, food webs, and much more. Nodes in these networks could be anything from individual organisms to specific genes, with edges indicating different types of biological interaction.

The beauty of graph theory is its generalizability. The same mathematical principles and techniques can be used to explore, analyze, and draw conclusions from all these diverse networks.

18.2 What is a Graph, What is a Network?

To delve deeper, we need to understand some key terms and components. Networks or graphs consist of two different types of elements: nodes and edges.

- **Nodes (or vertices):** These represent the entities within our system or dataset. Depending on the context, a node could represent a person, a computer, a protein, an intersection, or any other entity.
- **Edges (or links, connections):** These represent relationships or interactions between the entities. An edge could represent a physical connection (like a cable, road, rail, or pipe), a social or professional relationship, a communication channel, a biological interaction, and more.

Edges can be either directed or undirected, and weighted or unweighted. Undirected edges indicate a mutual or two-way interaction, while directed edges indicate a one-way interaction. Weighted edges carry additional information about the strength or magnitude of the interaction, while unweighted edges simply indicate the presence of an interaction.

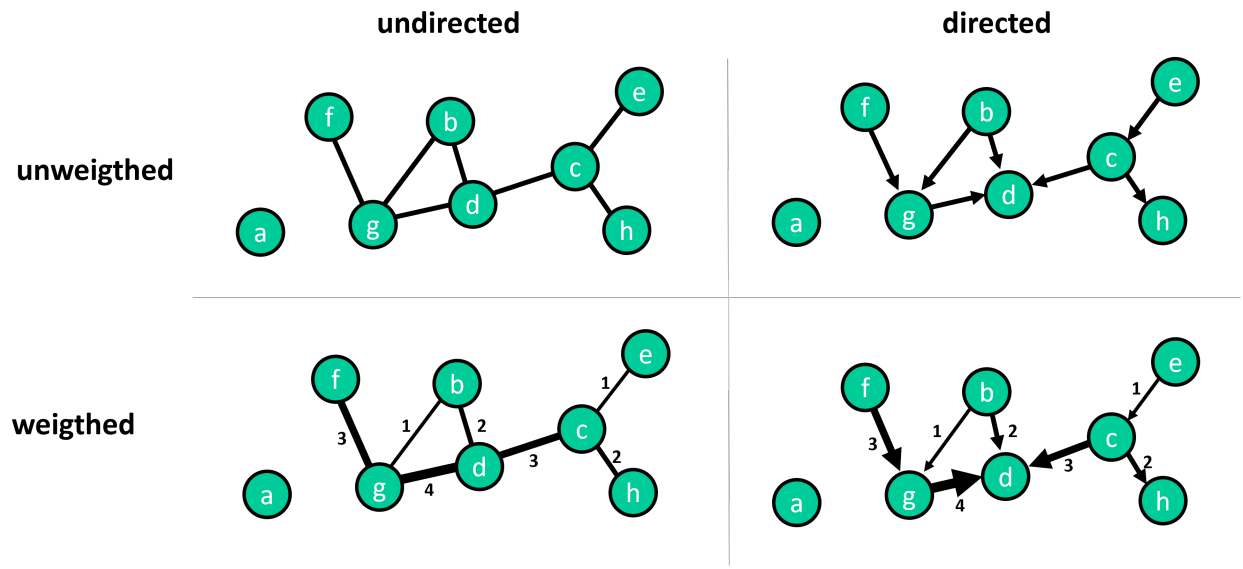


Fig. 18.1: Graphs come in different types or flavors. Key distinctions to make are whether or not links are directed or undirected. And whether or not link are weighted.

In the simplest form of a graph, we have unweighted and undirected edges, but real-world graphs often feature directed and/or weighted edges.

Finally, a network or graph has a count N of nodes and a count L of edges. The degree k of a node is the number of links or connections it has. For directed graphs, we distinguish between in-degree and out-degree, that is, the number of incoming and outgoing links.

The standard library in Python for working with graphs is `networkx`. Comprehensive documentation is available at this link: https://networkx.org/documentation/latest/_downloads/networkx_reference.pdf.

18.3 Graphs in Python with `NetworkX`

If if comes to working with networks/graphs, the goto Python library is `NetworkX`, which also comes with an extensive documentation. In addition, `NetworkX` also provides own tutorial material.

Here, we will do a quick introduction to NetworkX which we will then use for all graph-related parts in this course.

18.3.1 Installation

NetworkX can simply be installed using `pip install networkx`, or from a Jupyter notebook, use `!pip install networkx`.

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

# Import networkx
import networkx as nx
nx.__version__
```

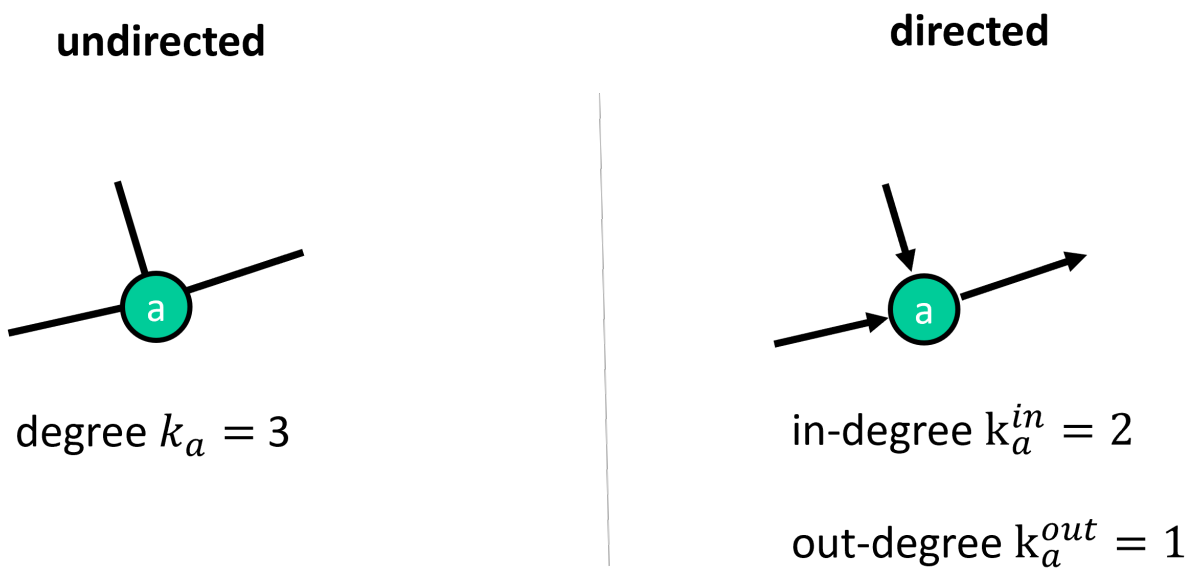


Fig. 18.2: The degree describes the number of links of a particular node. For directed graphs we distinguish in-degree and out-degree.

```
'3.1'
```

18.3.2 Create a graph object

We start by creating a graph object for an undirected graph using `nx.Graph()`. To initiate a directed graph we can use `nx.DiGraph()` … but that will come a bit later.

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

```
print(G)  # this is a graph object, but has no nodes or edges
```

```
Graph with 0 nodes and 0 edges
```

18.3.3 Add nodes and edges

What we initialized above is an empty graph. We can now add nodes and edges to the graph.

Nodes can be added:

- individually with `.add_node(...)`
- many nodes at once with `.add_nodes_from(...)`

Edges can be added:

- individually with `.add_edge(...)`
- many nodes at once with `.add_edges_from(...)`

```
# add a single node
G.add_node("Alice")

# add all nodes from a list
G.add_nodes_from(["Peter", "Sally", "Klaus", "Leonora"])
```

```
# get the nodes
G.nodes
```

```
NodeView(('Alice', 'Peter', 'Sally', 'Klaus', 'Leonora'))
```

```
# add single edge
G.add_edge("Alice", "Peter")

# add edges from a list
G.add_edges_from([("Peter", "Klaus"),
                  ("Klaus", "Leonora"),
                  ("Klaus", "Sally"),
                  ("Sally", "Leonora"),
                  ("Leonora", "Peter"),
                 ])
```

```
# get the edges
G.edges
```

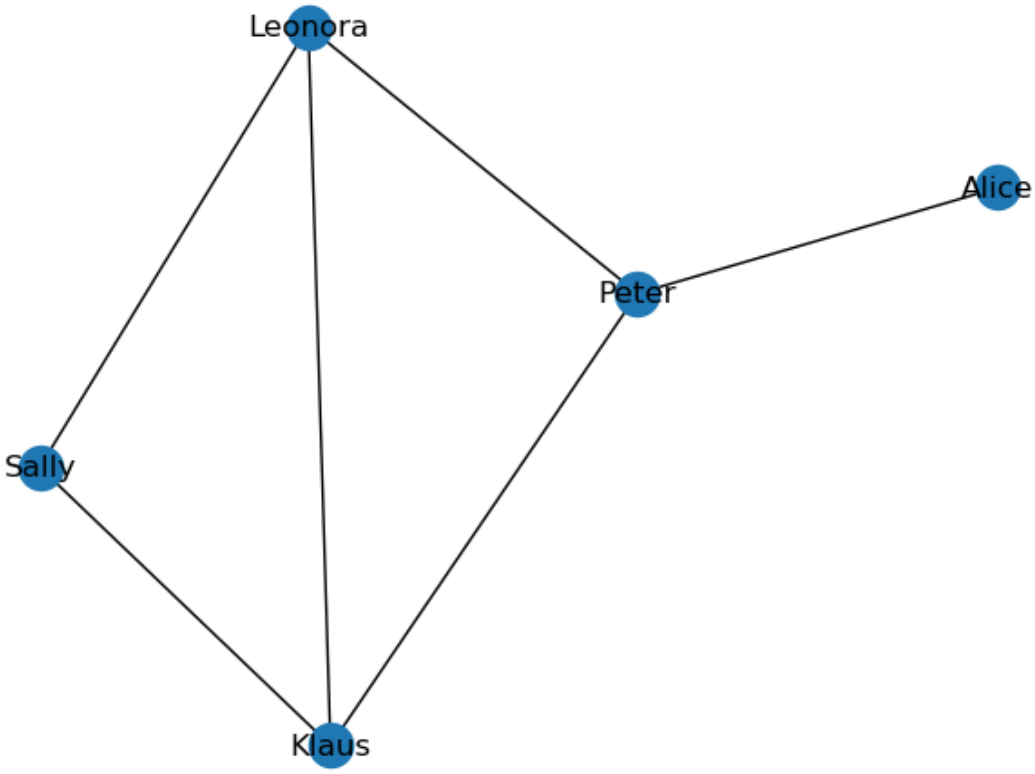
```
EdgeView([('Alice', 'Peter'), ('Peter', 'Klaus'), ('Peter', 'Leonora'), ('Sally',
 ↪'Klaus'), ('Sally', 'Leonora'), ('Klaus', 'Leonora')])
```

18.3.4 Visualize the graph with NetworkX

To make it clear from the start, NetworkX is great for a lot of things. And it also allows you to quickly visualize a graph. But this is mostly to get a quick impression of the data and not so much to produce high-quality visuals. In particular to display larger networks, we can better use specialized software for network visualizations which we will cover in the next chapter.

For now, however, having a quick look at our graph is always better than simply reading lists of strings (or tuples of strings). So, let's see what we just created!

```
nx.draw(G, with_labels=True)
```



We see that some nodes have more connections (up to three in this example), while others have fewer connections, such as the node "Alice" with only one edge to another node ("Peter"). The number of edges (or links/connections/relationships) of an individual node is called **degree**.

With NetworkX we can quickly get the degrees of all nodes in a graph via the attribute `.degree`.

```
G.degree
```

```
DegreeView({'Alice': 1, 'Peter': 3, 'Sally': 2, 'Klaus': 3, 'Leonora': 3})
```

18.3.5 Graph Metrics: Edge Count, Density, and Average Degree

To analyze networks, it's crucial to understand a few fundamental properties or metrics. These metrics can offer us insights into the structure and characteristics of the network, helping us interpret its overall behavior.

Some typical questions we might want to ask is:

- Has our graph many connections or rather few connections? (but then: what is many and what is few?)
- How many links do our nodes have (on average)?

Maximum Possible Number of Edges

One way to better judge if our graph has few or many edges is to first compute how many edges it *could have*. The **maximum possible number of edges**, denoted as L_{max}, in a network is given when every node is connected to every other node. In an undirected graph, this results in a complete graph where every pair of distinct vertices is connected by a unique edge. For a network with N nodes, the maximum possible number of edges is:

$$Lmax = \frac{N(N-1)}{2}$$

The division by 2 ensures that we are not double-counting edges since in an undirected graph, an edge between nodes A and B is the same as the edge between nodes B and A.

Density

The density d of a graph provides a measure of how many edges the graph has in relation to the maximum possible number of edges (L_{max}). It is defined as the ratio of the actual number of edges L to L_{max}:

$$d = L/L_{max} = \frac{2L}{N(N-1)}$$

The density can range between 0 and 1. A density close to 1 indicates that the graph is dense and most of the nodes are connected to each other, whereas a density close to 0 indicates that the graph is sparse with few edges.

Average Degree and Density

The average degree $\langle k \rangle$ of a graph is the mean number of edges per node. It's calculated by summing all node degrees and dividing by the total number of nodes N:

$$\langle k \rangle = \frac{\sum_i k_i}{N}$$

We can also express the density of a graph in terms of the average degree. This provides a sense of how interconnected the nodes of a graph are on average, relative to the maximum possible degree ($N-1$):

$$d = \frac{\langle k \rangle}{N-1}$$

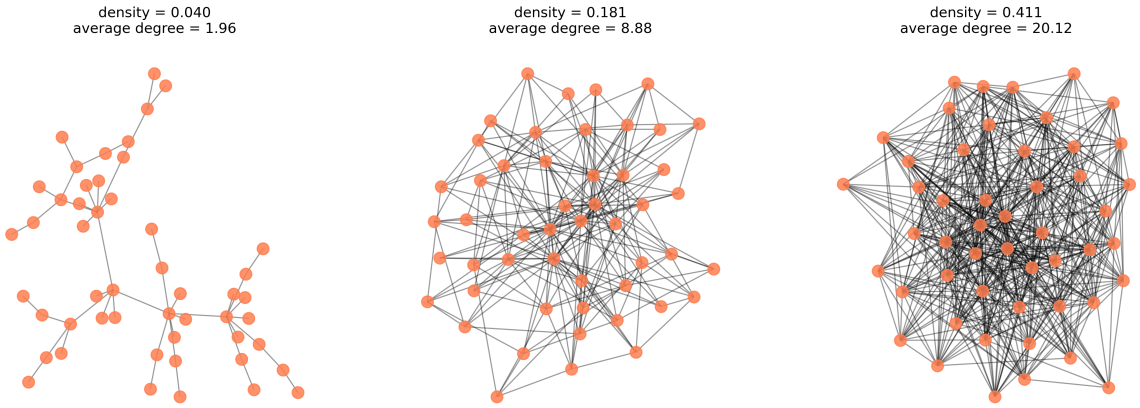
In the next sections, we will see how these metrics can be useful for understanding real-world networks and their properties. We'll also demonstrate how to calculate these metrics using the `networkx` Python library.

```
print(f"Number of nodes = {G.number_of_nodes()}")
print(f"Number of edges = {G.number_of_edges()}")
print(f"Maximum possible number of edges = {G.number_of_nodes() * (G.number_of_
 ↪nodes() - 1) / 2}")
print(f"Density = {nx.density(G)}")

# Code test
assert nx.density(G) == 0.6, "6 out of 10 possible edges --> 0.6"
```

```
Number of nodes = 5
Number of edges = 6
Maximum possible number of edges = 10.0
Density = 0.6
```

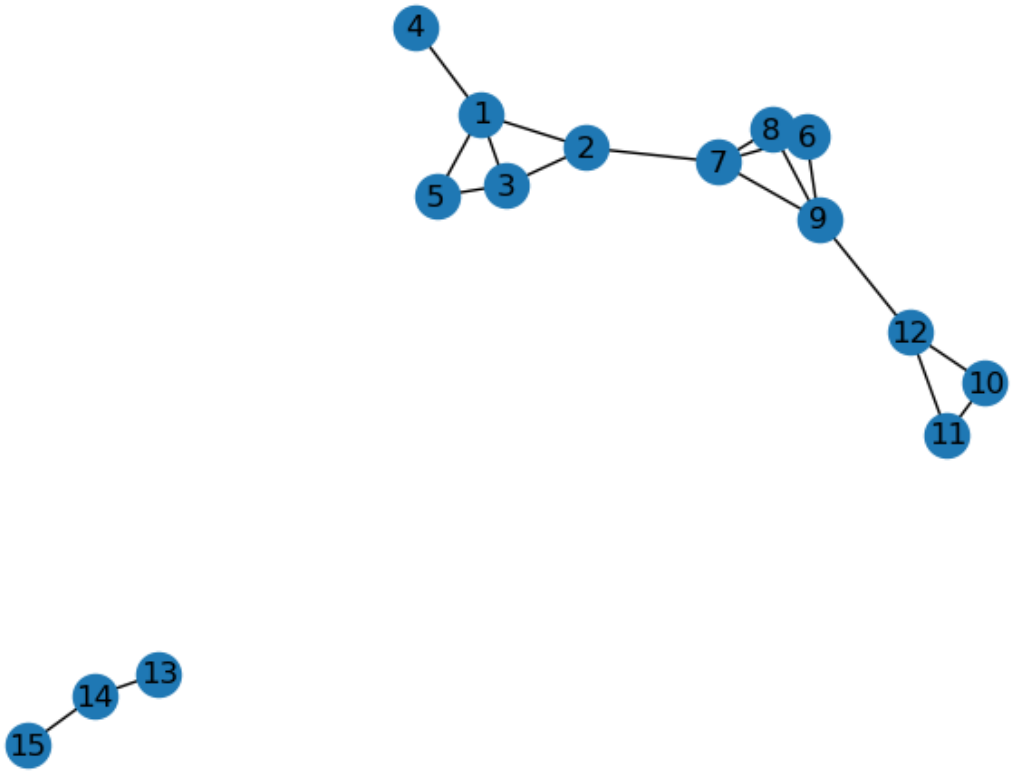
```
print(f"Average degree = {2 * G.number_of_edges() / G.number_of_nodes()}")
# or: print(f"Average degree = {np.mean([x[1] for x in G.degree])}")
```

```
Average degree = 2.4
```



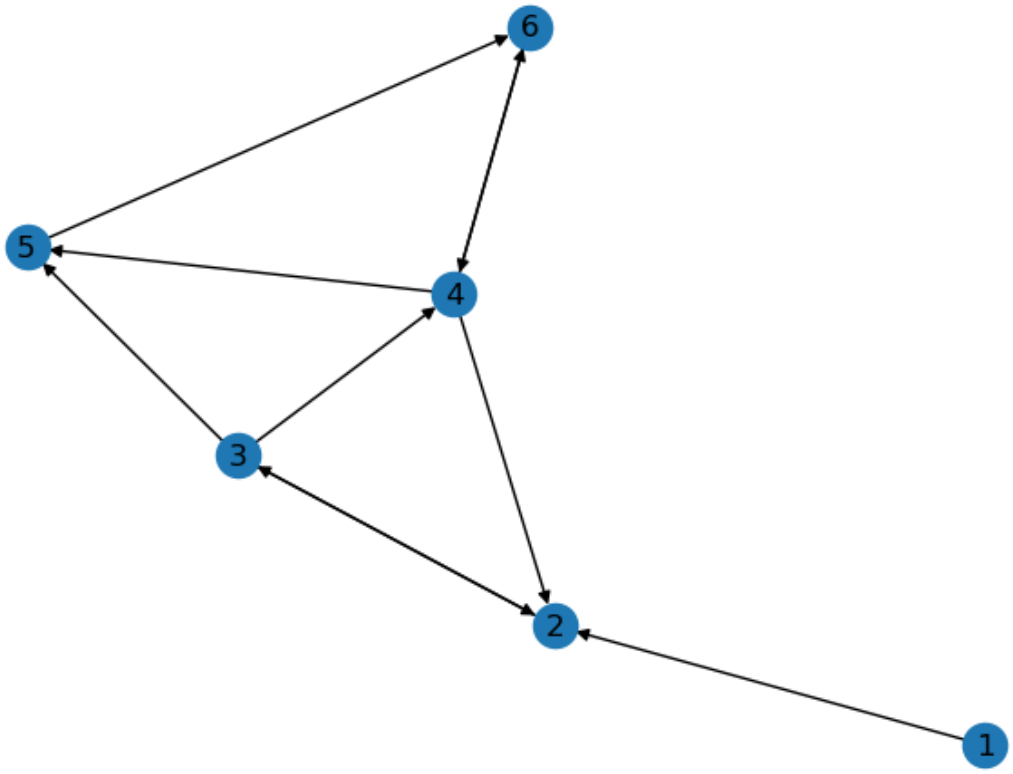
18.4 Not all graphs are connected

```
G = nx.Graph()
G.add_edges_from([[1,2], [2,3], [1,3], [1,4], [1,5], [3,5],
                  [6,7], [7,8], [8,9], [6,9], [6,8], [7,9],
                  [10,11], [11,12], [10,12], [9,12], [2,7],
                  [13, 14], [14, 15]])
nx.draw(G, with_labels=True)
```

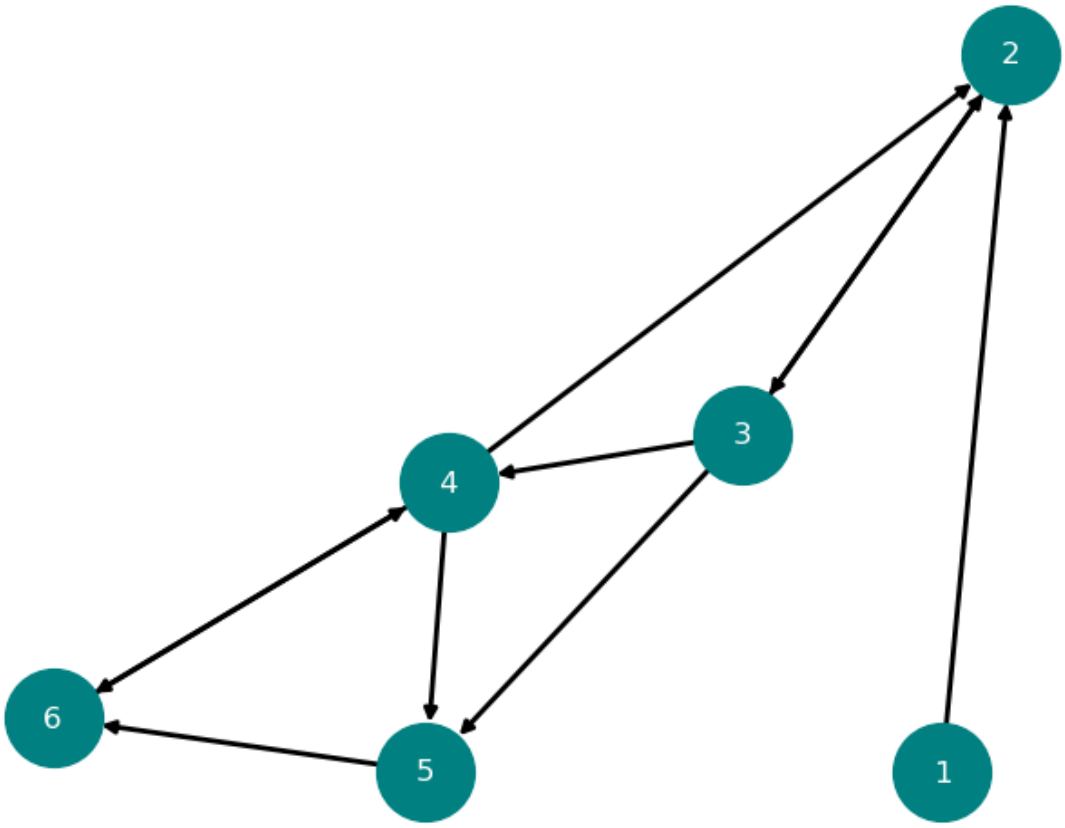


18.5 Directed graphs

```
D = nx.DiGraph()
D.add_edges_from([
    (1,2), (2,3),
    (3,2), (3,4),
    (3,5), (4,2),
    (4,5), (4,6),
    (5,6), (6,4),
])
nx.draw(D, with_labels=True)
```




```
nx.draw(D, with_labels=True,
        node_color='teal',
        node_size=1500,
        font_color='white',
        width=2)
```



18.6 Import network data

Hier schauen wir uns Flugverbindungen in den USA an `openflights_usa.graphml`

18.6.1 EXERCISE 1

Is there a direct flight between Indianapolis and Fairbanks, Alaska (FAI)? A direct flight is one with no intermediate stops.

18.6.2 EXERCISE 2

If I wanted to fly from Indianapolis to Fairbanks, Alaska what would be an itinerary with the fewest number of flights?

18.6.3 EXERCISE 3

Is it possible to travel from any airport in the US to any other airport in the US, possibly using connecting flights? In other words, does there exist a path in the network between every possible pair of airports?

18.7 Node importance

CHAPTER

NINETEEN

VISUALIZING GRAPHS

As previously described, graphs essentially consist of only two components, nodes and edges. Both nodes and edges may further possess additional attributes. However, these attributes usually do not include specific position or location data, with the exception of geographical data.

This means that a graph is defined by the nodes and their connections. The exact position of the nodes, both absolutely in space and relative to each other, is not defined! That's why the following four depictions can all be considered equivalent, as the network structure (topology) is identical in all four cases.

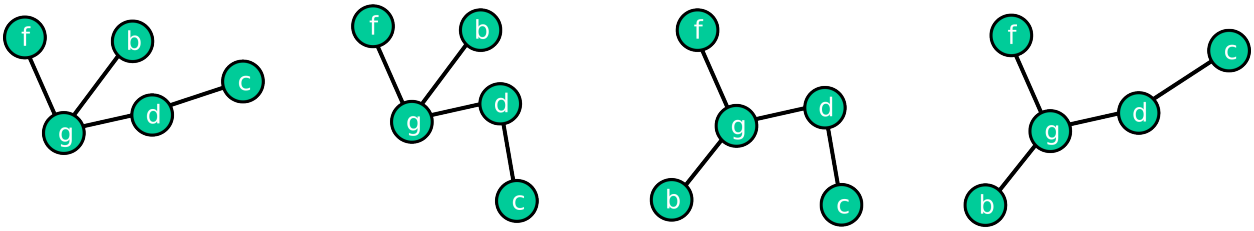
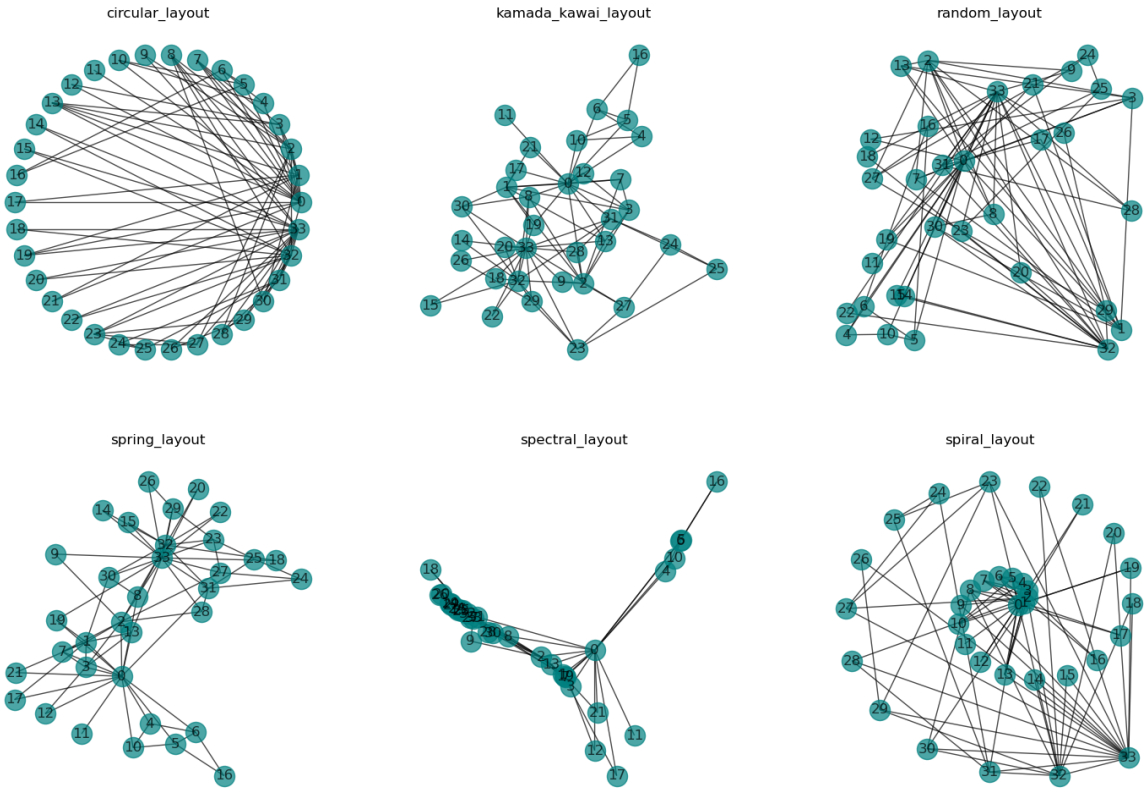


Fig. 19.1: Graphs are based on nodes and their connections (links/edges). But in nearly all cases, this does not fully determine the position of the nodes.

When we need to visualize a graph, we (or an algorithm) must decide how to arrange the nodes. There are numerous possibilities for this, and the following images show six such network layouts that are included in networkx.

```
nx.layout.__all__
```

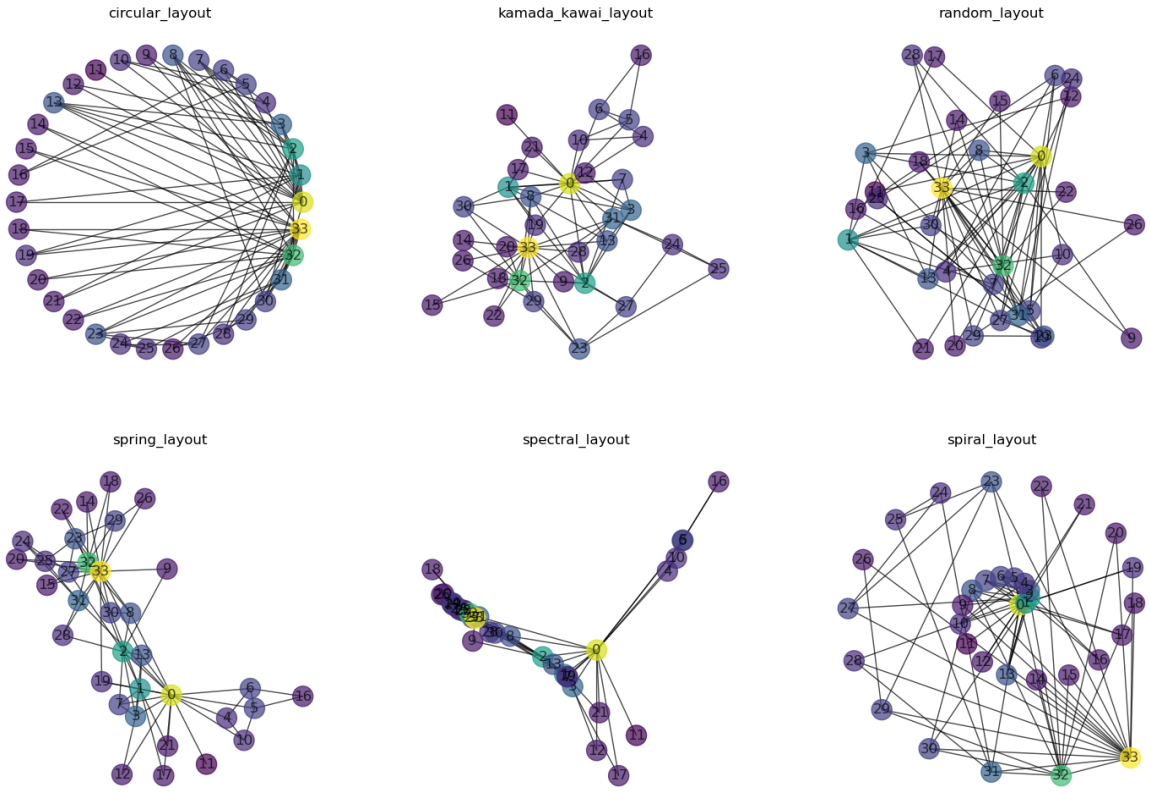
```
['bipartite_layout',
 'circular_layout',
 'kamada_kawai_layout',
 'random_layout',
 'rescale_layout',
 'rescale_layout_dict',
 'shell_layout',
 'spring_layout',
 'spectral_layout',
 'planar_layout',
 'fruchterman_reingold_layout',
 'spiral_layout',
 'multipartite_layout',
 'arf_layout']
```



19.1 Add more dimensions to the visualization

Inspecting graphs visually is often a great way to get an intuition of the data. Besides node and edge positions, we can add additional information (or attributes) through node labels, node color, but also node size and shape.

In the following we will color each node by its degree, i.e., the number of connections each node has.



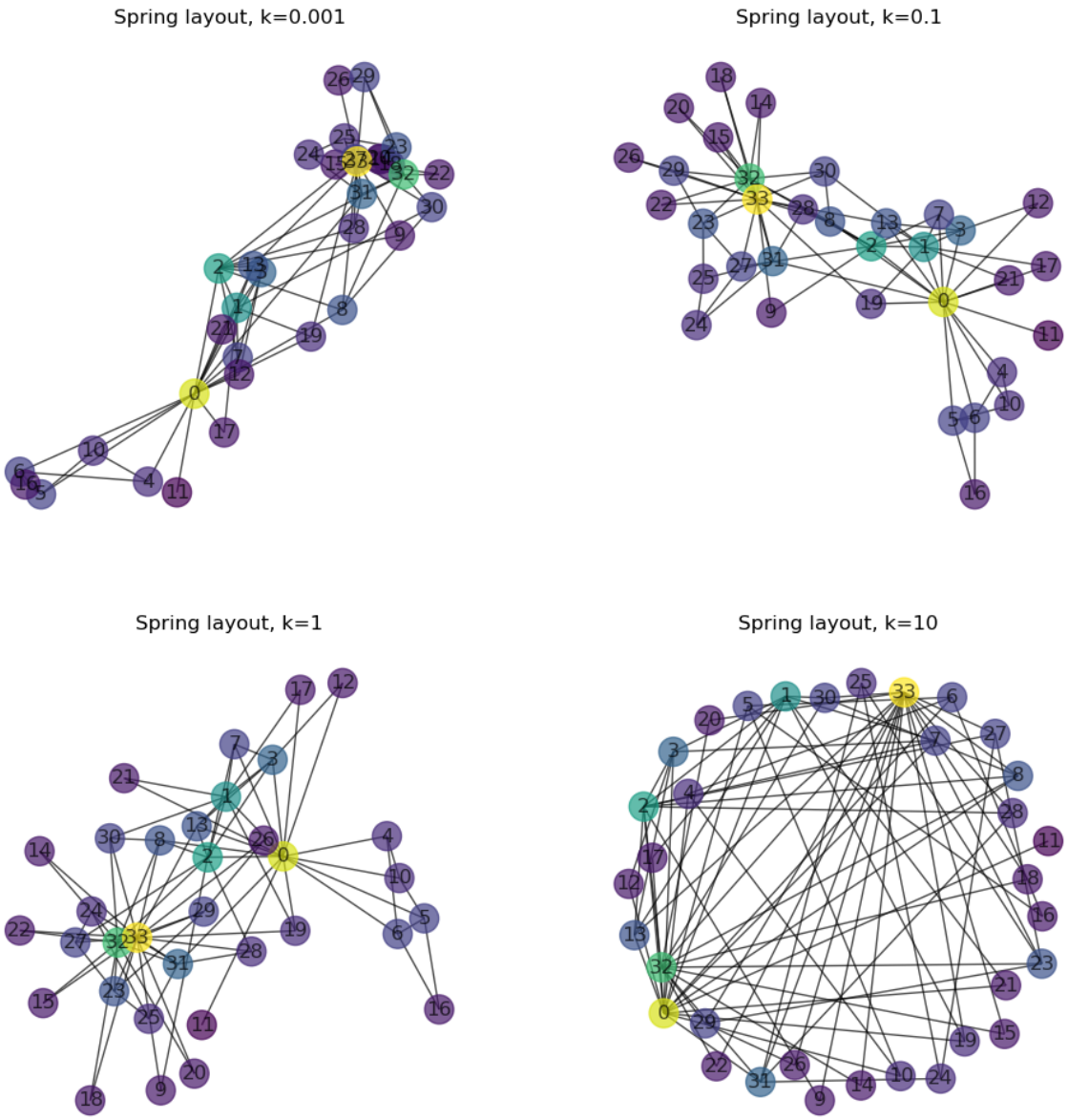
Different layouts have different strengths and weaknesses. A circular arrangement (**circular layout**) can, for instance, be useful for quickly identifying which nodes have the most connections (and where these lead). Layouts such as random layout or other fixed forms like spirals etc. (as seen above: **spiral layout**) are much less commonly used in practice.

The most commonly used layout is a so-called "force-directed" layout (in networkx the **spring-layout**, but also the **kamada-kawai-layout** fall in this category).

19.2 Force-directed / spring layout

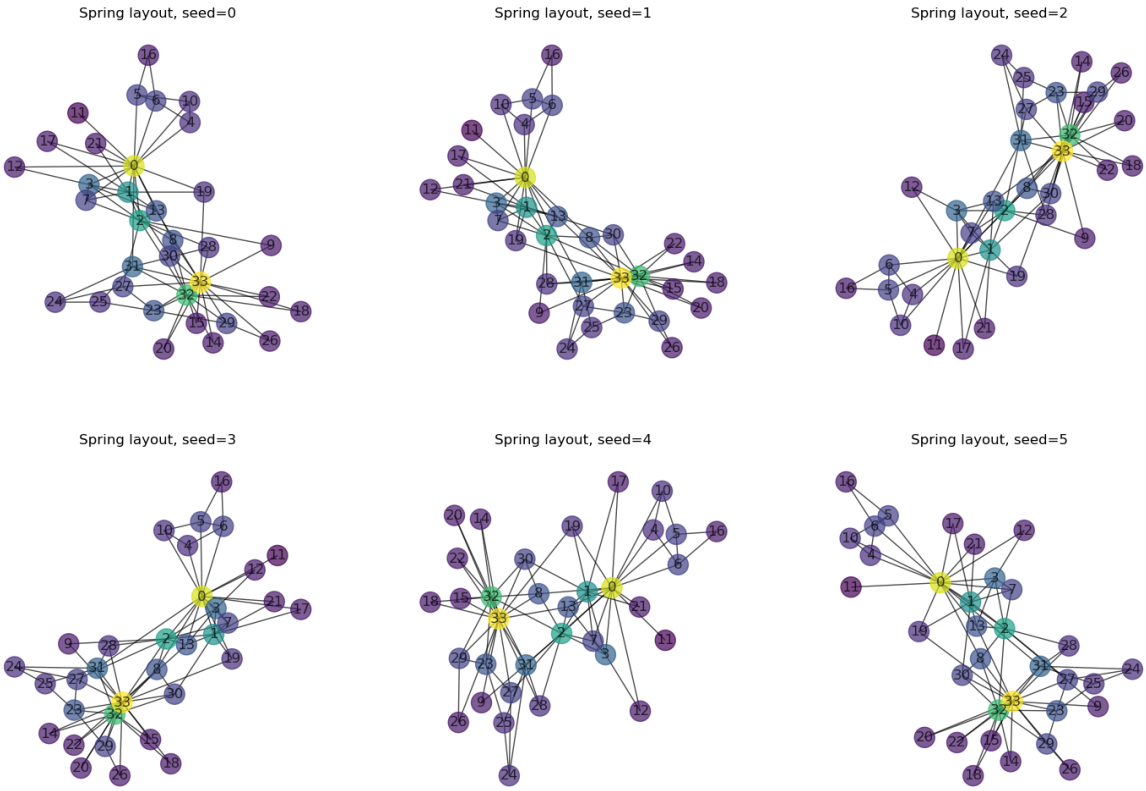
In the so-called force-directed layout, links are interpreted as mechanical springs. In a sort of physics simulation, the nodes (connected by the mechanical springs) are then gradually moved depending on the prevailing forces (springs and attractive forces).

Depending on the tool, various parameters can be set. Central here is the "strength" and/or "length" of the springs, often expressed as the physical spring constant k. This can lead to very different results:



19.2.1 Visual Randomness

Moreover, even with a set layout type and parameters, there are significant visual variations that result from random starting positions of the nodes. If we apply the same layout to the same graph multiple times, the results all look different (unless the seed is fixed).



19.3 Visualization Tools

While Python, along with `networkx`, can be used for graph visualization, it's typically used only for initial exploration or for very small graphs or subgraphs. For more elaborate visualizations, there are several quite useful and freely available tools:

19.3.1 Cytoscape (https://cytoscape.org/)

Cytoscape provides a wide range of settings for layout and graphical features, making it a flexible tool for graph visualization.

Pros:

- Many layout and graphical settings.
- Good export functionality (image, graph, HTML).
- Good data import functions.

Cons:

- No live rendering.
- Possibly more quickly overwhelmed than Gephi and Graphia?

19.3.2 Gephi (https://gephi.org/)

Gephi is recognized for its live rendering and ability to handle relatively large graphs. It also offers numerous statistical tools for in-depth graph analysis:

Pros:

- Live rendering.
- Can handle (relatively) large graphs reasonably well.
- Many statistical functions.

Cons:

- Graphically offers fewer options than Cytoscape.

19.3.3 Graphia (https://graphia.app/)

Graphia is a quick tool that provides a 3D option for graph visualization. However, it's not as flexible as the other tools in terms of layout and graphical settings:

Pros:

- Fast performance.
- 3D option available.

Cons:

- Fewer customization options (for both layout and graphical display).
- Image export is not very flexible.

CHAPTER

TWENTY

BOTTLENECKS, HUBS, COMMUNITIES

20.1 Measuring the Structure of a Graph

Graphs come in a multitude of different forms or structures. There are graphs with few connections per node (low average degree, e.g., high-school dating dataset) and those with many connections per node (e.g., social networks).

Moreover, there is a whole range of metrics that can tell us something about the nature of a graph. For instance, the average path length is an indicator of how far apart elements typically are. Networks with many cross-connections and hubs/centers usually have a small average path length (often referred to as "Small World" in some contexts).

The *Clustering Coefficient* indicates how well nodes are interconnected, i.e., how often neighbor nodes are also neighbors of each other ("A friend of a friend is a friend…").

Average path length

$$\langle \ell \rangle = \frac{\sum_{i,j} \ell_{ij}}{N(N-1)}$$

where ℓ_{ij} is the shortest path from i to j.

Clustering coefficient:

$$C(i) = \frac{\tau(i)}{\tau_{max}(i)} = \frac{2\tau(i)}{k_i(k_i-1)}$$

$$C = \frac{\sum_{i:k>1} C(i)}{N_{k>1}}$$

20.2 How important is a Node?

Often, we are interested in the importance of individual nodes. Which elements are essential for the network? What are potential bottlenecks? Who / What is best connected?

For this, there are a number of measuring methods, two of the most well-known are *closeness centrality* and *betweenness centrality*. The naming of these two terms is not ideal, at least we saw that if often is confusing to students at first. But let's try to understand both measures to learn what we can use them for.

20.2.1 Closeness Centrality

The *closeness centrality* measures the average distance from a node to all other nodes. High closeness centrality means a small average distance (since the reciprocal is used in the formula), i.e., the nodes with the highest values are the "central" points of the network. It's fastest to get anywhere from here.

Closeness centrality (for nodes):

$$g_i = \frac{N-1}{\sum_{i \neq j} \ell_{ij}}$$

with ℓ_{ij} being the shortest path from i to j.

It is important to note, that the closeness centrality does not necessarily reflect what we intuitively mean by the "importance" of a node for the rest of the network. If you live in a tiny village that happens to be close to a major trafic hub, such as an international airport, this node (your village) would get a high closeness centrality because many other nodes are within short reach. The really important node, however, is the trafic hub itself an not your village (hope this doesn't hurt any feelings…).

20.2.2 Betweenness Centrality

Betweenness centrality considers how many of all shortest paths pass through a certain node. Or in other words, how many shortest paths would be eliminated if a certain node were removed. Bottlenecks in the network, for instance, are characterized by high *betweenness centrality*. But also central hubs that facilitate many paths in the network will rank high.

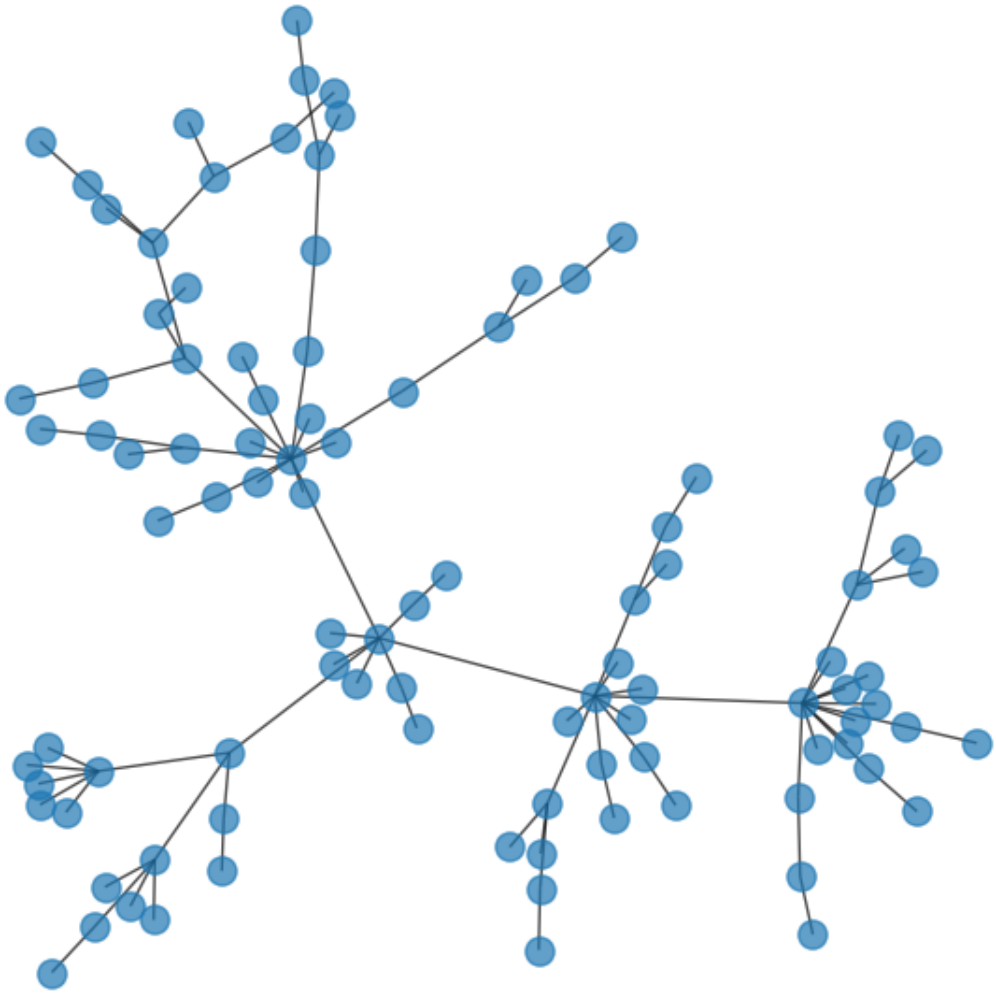
Betweenness centrality (for nodes)

$$b_i = \sum_{h \neq j \neq i} \frac{\sigma(j, h|i)}{\sigma(j, h)}$$

With $\sigma(j, h)$ the number of shortest paths from node j to h and $\sigma(j, h|i)$ the number of shortest paths from node j to h that pass through node i.

```
# generate a random network
G = nx.powerlaw_cluster_graph(100, 1, 0.5, seed=0)
pos = nx.spring_layout(G, k=0.05, seed=0)
```

```
fig, ax = plt.subplots(figsize=(8, 8))
nx.draw(G,
        pos=pos,
        node_size=120,
        alpha=0.7,
        ax=ax,
       )
```



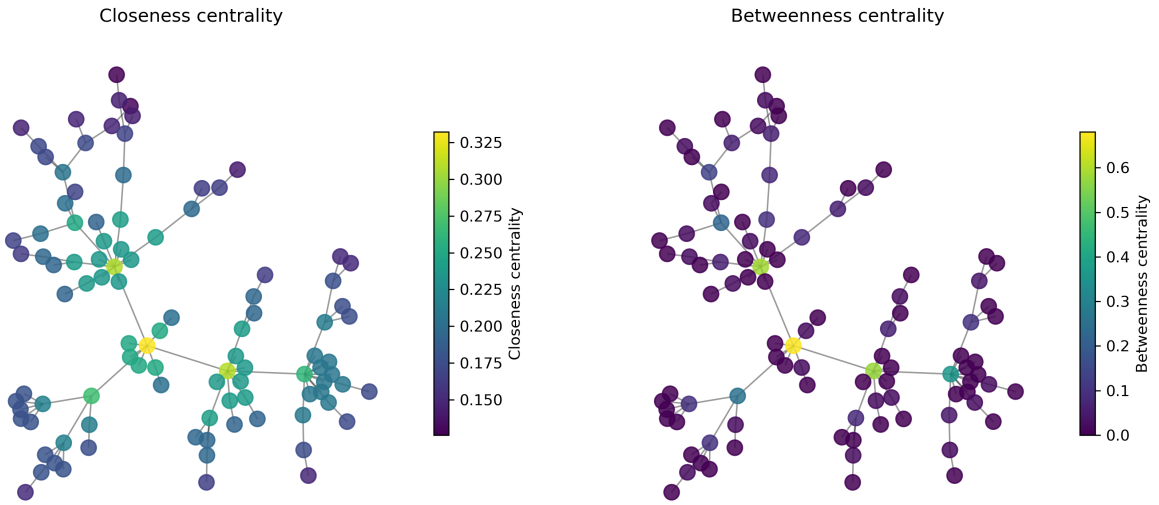
20.3 What are the “most important” nodes?

Graphs often represent complex data containing a lot of information about relationships, flows, routes, etc. …

We are often interested in measuring the “importance” of nodes. And importance can mean a lot of different things, depending on what my data is and what my precise question and interest is.

closeness centrality

betweenness centrality



20.4 How important is an edge?

Betweenness centrality can be calculated not only for nodes but also for edges. Analog to the case for nodes you can think of this as a question to how often a certain link is used when traversing the graph between two nodes.

Betweenness centrality (for edges)

$$b_e = \sum_{i,j} \frac{\sigma(i,j|e)}{\sigma(i,j)}$$

With $\sigma(i,j)$ the number of shortest paths from node i to j and $\sigma(i,j|e)$ the number of shortest paths from node i to j that pass through edge e.

Part VIII

References

CHAPTER

TWENTYONE

ACKNOWLEDGEMENTS

Many thanks to the DAISY (Data Science, Artificial Intelligence, Intelligent Systems) students which were the first students this course was taught to. Many aspects of the course from conception to typos have been revised due to their reactions, their comments and feedback.

This course has also been taught to master media-informatics students. They, too, greatly helped to test and evaluate, to improve and revise the presented course material and topics.

Special thanks go to David Sokalski who teaches this course with me, evaluates the live coding materials and helps with feedback and great suggestions.

This book would not have been possible without the Netherlands eScience Center, an outstanding place to work on research software, but also an highly idealistic bunch of people dreaming of a better academic world. For some reason, they believed that I could become a good research software engineer and a data scientists despite my very poor programming skills at the time I applied for that job. In the three years I worked there I learned more than ever before allowing me to succeed with a career switch towards data science.

Precisely that career switch finally brought me to the Düsseldorf University of Applied Sciences (HSD) and the Centre for Digitalization and Digitality (ZDD). Unlike many other places, they fully embraced truly interdisciplinary approaches, which in my case meant they accepted my application even though it did not exactly match a linear academic profile, nor a formal training in informatics.

CHAPTER

TWENTYTWO

BIBLIOGRAPHY

{bibliography}

Part IX

Source Code and Contributions

CHAPTER

TWENTYTHREE

SOURCE CODE ON GITHUB

The full material to produce this book via Jupyter Book [Community, 2020], expect for the used datasets which we only include as links, can be found on the GitHub repository.

CHAPTER

TWENTYFOUR

CONTRIBUTE

We welcome feedback, comments and contributions to this book. For mistakes, issues etc. please on an `issues` on GitHub, or comment to an existing one. Thanks!

BIBLIOGRAPHY

[1] Amit Datta, Michael Carl Tschantz, and Anupam Datta. Automated experiments on ad privacy settings: a tale of opacity, choice, and discrimination. 2015. arXiv:1408.6491.

[2] Muhammad Ali, Piotr Sapiezynski, Miranda Bogen, Aleksandra Korolova, Alan Mislove, and Aaron Rieke. Discrimination through optimization. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–30, nov 2019. URL: https://doi.org/10.1145%2F3359301, doi:10.1145/3359301.

[3] Reuters. Amazon scraps secret AI recruiting tool that showed bias against women. October 2018. URL: https://www.reuters.com/article/us-amazon-com-jobs-automation-insight-idUSKCN1MK08G (visited on 2023-06-05).

[4] Kirsten Martin. Ethical Implications and Accountability of Algorithms. *Journal of Business Ethics*, 160(4):835–850, December 2019. URL: https://doi.org/10.1007/s10551-018-3921-3 (visited on 2023-06-09), doi:10.1007/s10551-018-3921-3.

[5] Solon Barocas and Andrew D. Selbst. Big Data's Disparate Impact. *California Law Review*, 104(3):671–732, 2016. Publisher: California Law Review, Inc. URL: https://www.jstor.org/stable/24758720 (visited on 2023-06-09).

[6] Catherine D'Ignazio and Lauren F. Klein. *Data feminism*. <Strong> ideas series. The MIT Press, Cambridge, Massachusetts ; London, England, 2020. ISBN 978-0-262-04400-4.

[7] C. Stinson. Algorithms are not neutral. *AI Ethics*, 2:763–770, 2022. doi:10.1007/s43681-022-00136-w.

[8] Sina Fazelpour and David Danks. Algorithmic bias: Senses, sources, solutions. *Philosophy Compass*, 16(8):e12760, 2021. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/phc3.12760. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/phc3.12760 (visited on 2023-06-09), doi:10.1111/phc3.12760.

[9] Shoshana Zuboff. *The Age of Surveillance Capitalism: The Fight for a Human Future at the New Frontier of Power*. Profile Books, 1st edition, 2019. ISBN 9781781256848.

[10] Vladimir Pletser and Dirk Huylebrouck. The Ishango Artefact: the Missing Base 12 Link. *Forma*, 1999.

[11] Karl Popper. *Karl Popper: Logik der Forschung*. Akademie Verlag, July 2013. ISBN 978-3-05-006378-2. URL: https://www.degruyter.com/document/doi/10.1524/9783050063782/html (visited on 2023-09-04), doi:10.1524/9783050063782.

[12] Thomas Bartelborth. *Die erkenntnistheoretischen Grundlagen induktiven Schließens*. Universität Leipzig, 2017. URL: https://nbn-resolving.org/urn:nbn:de:bsz:15-qucosa-220168.

[13] Claus Weihs and Katja Ickstadt. Data Science: the impact of statistics. *International Journal of Data Science and Analytics*, 6(3):189–194, November 2018. URL: https://doi.org/10.1007/s41060-018-0102-5 (visited on 2023-09-05), doi:10.1007/s41060-018-0102-5.

[14] David Donoho. 50 Years of Data Science. *Journal of Computational and Graphical Statistics*, 26(4):745–766, October 2017. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/10618600.2017.1384734. URL: https://doi.org/10.1080/10618600.2017.1384734 (visited on 2023-09-05), doi:10.1080/10618600.2017.1384734.

[15] Rüdiger Wirth and Jochen Hipp. Crisp-dm: towards a standard process model for data mining. In *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, volume 1, 29–39. Manchester, 2000.

[16] Hilary Mason and Chris Wiggins. Dataists » A Taxonomy of Data Science. 2010. URL: http://www.dataists.com/2010/09/a-taxonomy-of-data-science/ (visited on 2023-09-05).

[17] Philip Guo. Data Science Workflow: Overview and Challenges. 2022. URL: https://cacm.acm.org/blogs/blog-cacm/169199-data-science-workflow-overview-and-challenges/fulltext (visited on 2023-09-05).

[18] Justin Matejka and George Fitzmaurice. Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, 1290–1294. New York, NY, USA, May 2017. Association for Computing Machinery. URL: https://doi.org/10.1145/3025453.3025912 (visited on 2023-09-07), doi:10.1145/3025453.3025912.

[19] Thomas Piketty. *Capital in the Twenty-First Century:*. Belknap Press, Cambridge, MA, April 2014. ISBN 978-0-674-43000-6.

[20] Narendra Kumar Gupta, Giuseppe Di Fabbrizio, and Patrick Haffner. Capturing the stars: predicting ratings for service and product reviews. In *HLT-NAACL 2010*. 2010.

[21] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. October 2013. arXiv:1310.4546 [cs, stat]. URL: http://arxiv.org/abs/1310.4546 (visited on 2023-06-12), doi:10.48550/arXiv.1310.4546.

[22] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. September 2013. arXiv:1301.3781 [cs]. URL: http://arxiv.org/abs/1301.3781 (visited on 2023-06-12), doi:10.48550/arXiv.1301.3781.

[23] Executable Books Community. Jupyter Book. February 2020. URL: https://zenodo.org/record/4539666 (visited on 2023-06-06), doi:10.5281/zenodo.4539666.