Artificial Intelligence and Big Data

Postgraduate Program in Central Banking (CEMFI)

Joël Marbet

January 19, 2026

Table of contents

About this Course

Artificial intelligence (AI) is transforming economics and finance, from credit risk assessment to economic forecasting and policy analysis. Recent breakthroughs in machine learning and generative AI have opened unprecedented opportunities for researchers and practitioners to extract insights from vast datasets and automate complex analytical tasks.

This course provides a hands-on introduction to the AI techniques driving these changes. While we explore the theoretical foundations necessary to understand how these methods work, our primary focus is on practical implementation. You will learn to build, and train machine learning models using Python, the dominant language in AI and data science. Through a combination of conceptual understanding and applied programming, you will gain the skills needed to harness AI for tackling real-world problems in economics, finance, and beyond.

Learning Objectives

By the end of this course, you will be able to:

- **Understand core AI concepts**: Grasp the fundamental principles behind supervised learning, natural language processing, and generative AI that drive modern applications in economics and finance
- **Implement machine learning models**: Build and train decision trees, neural networks, and other ML algorithms to solve prediction and classification problems using real-world datasets
- **Process and analyze text data**: Apply natural language processing techniques to extract insights from textual sources such as financial reports, news articles, and policy documents
- **Leverage generative AI**: Work with large language models (LLMs) programmatically through APIs, going beyond chatbot interactions to automate tasks and build AI-powered applications
- **Use Python for data science**: Write Python code using industry-standard libraries for data manipulation, visualization, and machine learning

Course Structure

This course is divided into four parts, each focusing on different aspects of AI and big data:

- **Part I: Foundations**
  1. Introduction to Artificial Intelligence and Big Data
  2. Introduction to Python
- **Part II: Supervised Machine Learning**
  3. Overview of Supervised Learning
  4. Decision Trees
  5. Neural Networks
  6. Practice Session I
- **Part III: Natural Language Processing**
  7. Overview of Natural Language Processing (NLP)
  8. Classical NLP Approaches
  9. Practice session II
- **Part IV: Generative AI**
  10. Overview of Generative AI
  11. Large Language Models (LLMs)
  12. Practice session III

Prerequisites

This course is designed to be **accessible to students with diverse backgrounds**. The **following will help** you get the most out of the course:

- **Statistics and probability**: A basic understanding of statistical concepts (mean, variance, distributions) and probability theory at the undergraduate level.
- **Mathematics**: Familiarity with linear algebra (vectors, matrices) and calculus (derivatives, gradients) will be helpful for understanding how machine learning algorithms work under the hood. However, we will try not to go too deeply into the mathematics. Instead, we focus on building intuition about how models work and on practical implementation.

The **following is explicitly not required** :

- **Programming experience**: Prior programming knowledge is helpful but not required. We introduce Python from the ground up in Part I, covering all necessary programming concepts for the course.
- **Machine learning background**: No prior experience with AI or machine learning is expected. We start with the fundamentals and build up to advanced topics.

Useful Resources

The course does not follow a particular textbook but has drawn material from several sources such as

- Hastie, Tibshirani, and Friedman (2009), "The Elements of Statistical Learning"
- Murphy (2012), "Machine Learning: A Probabilistic Perspective"
- Murphy (2022), "Probabilistic Machine Learning: An Introduction"
- Murphy (2023), "Probabilistic Machine Learning: Advanced Topics"
- Goodfellow, Bengio, and Courville (2016), "Deep Learning"
- Bishop (2006), "Pattern Recognition And Machine Learning"
- Nielsen (2019), "Neural Networks and Deep Learning"
- Sutton and Barto (2018), "Reinforcement Learning: An Introduction"

Note that all of these books are officially **available for free** in the form of PDFs or online versions (see the links in the references). However, you are not required to read them and, as a word of warning, the books go much deeper into the mathematical theory behind the machine learning techniques than we will in this course. Nevertheless, you may find them useful if you want to learn more about the subject.

Regarding **programming in Python**, McKinney (2022) "Python for Data Analysis" might serve as a good reference book. The book is **available for free** online and covers a lot of the material we will be using in this course. You can find it here: Python for Data Analysis.

Software Installation Notes

In this course, we will use Nuvolos to run all Python code, which provides a pre-configured environment with all the necessary packages. If you would like to set up a **local Python environment** on your computer instead, you can use the following guide. We use the Anaconda distribution, which simplifies package management and ensures everyone has a consistent development environment. For code editing and running Jupyter notebooks, we use Visual Studio Code (VS Code), a powerful and beginner-friendly code editor.

The following instructions will guide you through installing Python, creating a dedicated environment for this course, and setting up VS Code on your machine.

Anaconda Installation

The first step is to install the Anaconda distribution:

1. Download the Anaconda distribution from anaconda.com. Note: If you are using a M1 Mac (or newer), you have to choose the 64-Bit (Apple silicon) Graphical Installer. With an older Intel Mac, you can choose the 64-Bit (Intel chip) Graphical Installer. With Windows, you can choose the 64-Bit Graphical Installer (i.e., the only Windows option).

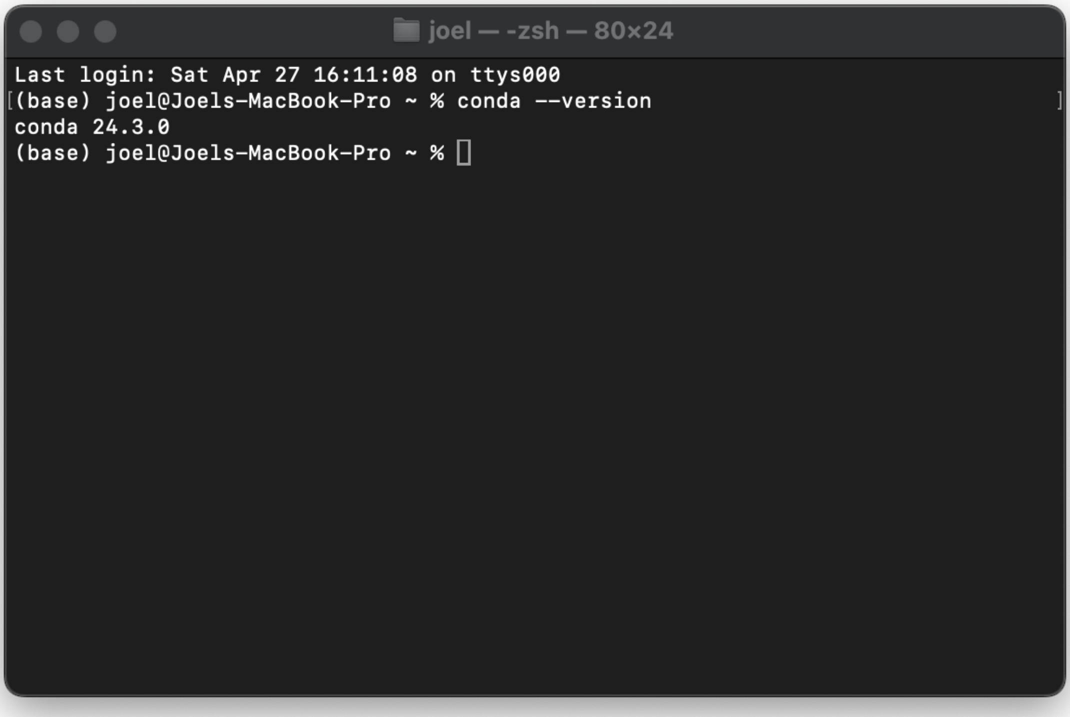
2. Open the installer that you have downloaded in the previous step and follow the on-screen instructions.

3. If it asks you to update Anaconda Navigator at the end, you can click `Yes` (to agree to the update), `Yes` (to quit Anaconda Navigator) and then `Update Now` (to actually start the update).

To **confirm that the installation was successful**, you can open a *terminal window* on macOS/Linux or an *Anaconda Prompt* if you are on Windows and run the following command:

```
conda --version
```

This should display the version of Conda that you have installed. If you see an error message, the installation was likely not successful and you should ask for advice from your peers or send me an email.



```
Last login: Sat Apr 27 16:11:08 on ttys000
[(base) joel@Joels-MacBook-Pro ~ % conda --version                              ]
conda 24.3.0
(base) joel@Joels-MacBook-Pro ~ % 
```

Figure 1: Terminal Output after Anaconda Installation

Creating a Conda Environment

Next, we want to create a new environment for this course that contains the correct Python version and all the Python packages we need. We can do this by creating a new Conda environment from the `environment.yml` provided on Moodle.

1. Open a *terminal window* on macOS/Linux or an *Anaconda Prompt* if you are on Windows.

2. There are two ways to create the Conda environment:

   Option A: Run the following command from the terminal or Anaconda Prompt:

   ```
   conda env create -f https://aibigdata.joelmarbet.com/environment.yml
   ```

   This downloads the `environment.yml` file automatically and creates the environment.

   Option B: Download the `environment.yml` file manually:

   i. Navigate to the folder where you have downloaded the `environment.yml` file. On macOS/Linux, you can do this by running the following command in the terminal:

      ```
      cd ~/Downloads
      ```

      which will navigate to the `Downloads` folder in your home directory.

      On Windows, you can do this by running the following command in the Anaconda Prompt:

      ```
      cd "%userprofile%/Downloads"
      ```

      which will navigate to the `Downloads` folder in your user profile.

      Note that if you use a different path that contains space you need to put the path in quotes, e.g., `cd "~/My Downloads"`.

   ii. Create a new Conda environment from the `environment.yml` file by running the following command in the terminal or Anaconda Prompt:

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

   Either option will create a new Conda environment called `ai-big-data-cemfi` with the correct Python version and all the Python packages we need for this course. Note that the installation might take a few minutes.

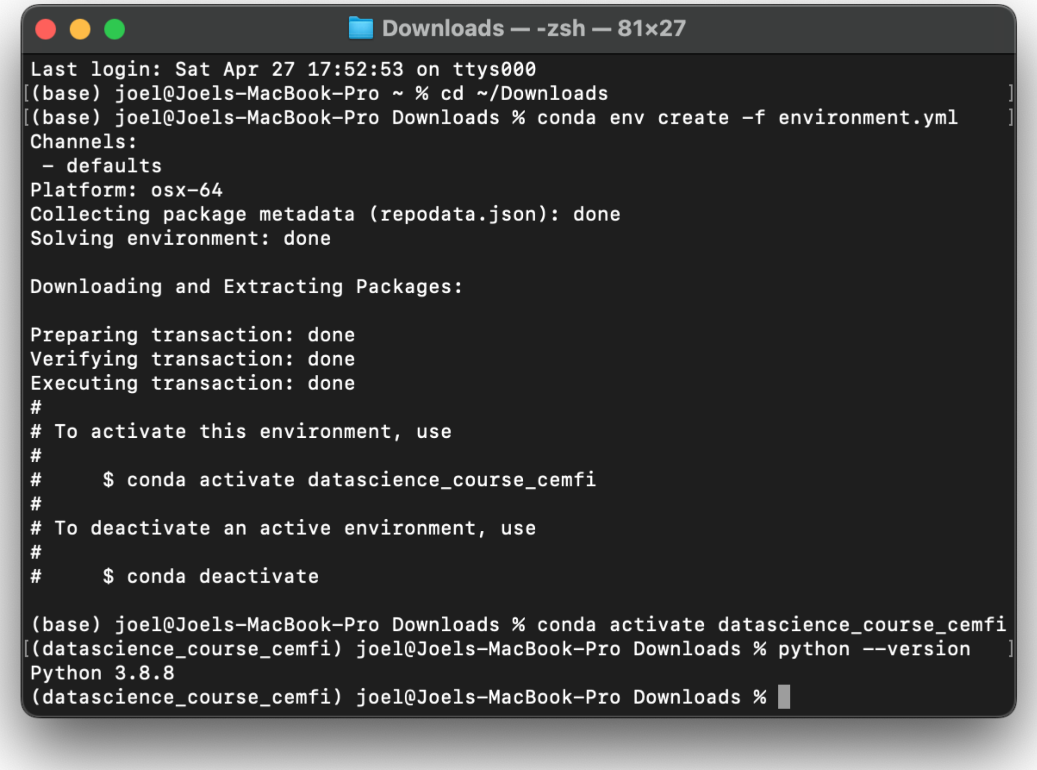
3. Activate the new Conda environment by running the following command in the terminal or Anaconda Prompt:

   ```
   conda activate ai-big-data-cemfi
   ```

To **confirm that the environment was created successfully**, you can run the following command in the terminal or Anaconda Prompt:

```
conda env list
```

This should display a list of all Conda environments on your machine, with an asterisk (`*`) next to the currently active environment. You should see `ai-big-data-cemfi` in the list.



```
Last login: Sat Apr 27 17:52:53 on ttys000
[(base) joel@Joels-MacBook-Pro ~ % cd ~/Downloads                               ]
[(base) joel@Joels-MacBook-Pro Downloads % conda env create -f environment.yml  ]
Channels:
 - defaults
Platform: osx-64
Collecting package metadata (repodata.json): done
Solving environment: done

Downloading and Extracting Packages:

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate datascience_course_cemfi
#
# To deactivate an active environment, use
#
#     $ conda deactivate

(base) joel@Joels-MacBook-Pro Downloads % conda activate datascience_course_cemfi
[(datascience_course_cemfi) joel@Joels-MacBook-Pro Downloads % python --version  ]
Python 3.8.8
(datascience_course_cemfi) joel@Joels-MacBook-Pro Downloads %
```

Figure 2: Terminal Output From Environment Creation

Resetting or Updating a Conda Environment

If you accidentally make changes to the environment and want to reset it to the original state, you can do this by navigating to the folder where you have downloaded `environment.yml` and then running the following command in the *terminal* or *Anaconda Prompt*:

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

Alternatively, you can also update the environment by running the following command in the *terminal* or *Anaconda Prompt*, which downloads the environment.yml file automatically from the course website:

```
conda env update --file https://aibigdata.joelmarbet.com/environment.yml
--prune
```

> This can also be used to update the environment if we add new packages to the `environment.yml` file.

Installing VS Code

The last step is to install the Visual Studio Code (VS Code) editor:

1. Download the Visual Studio Code editor from code.visualstudio.com.
2. Open the installer that you have downloaded in the previous step and follow the on-screen instructions.

We also need to install some VS Code extensions that will help us with Python programming and Jupyter notebooks:

1. Open VS Code.

2. Click on the `Extensions` icon on the left sidebar (or press `Cmd+Shift+X` on macOS or `Ctrl+Shift+X` on Windows).

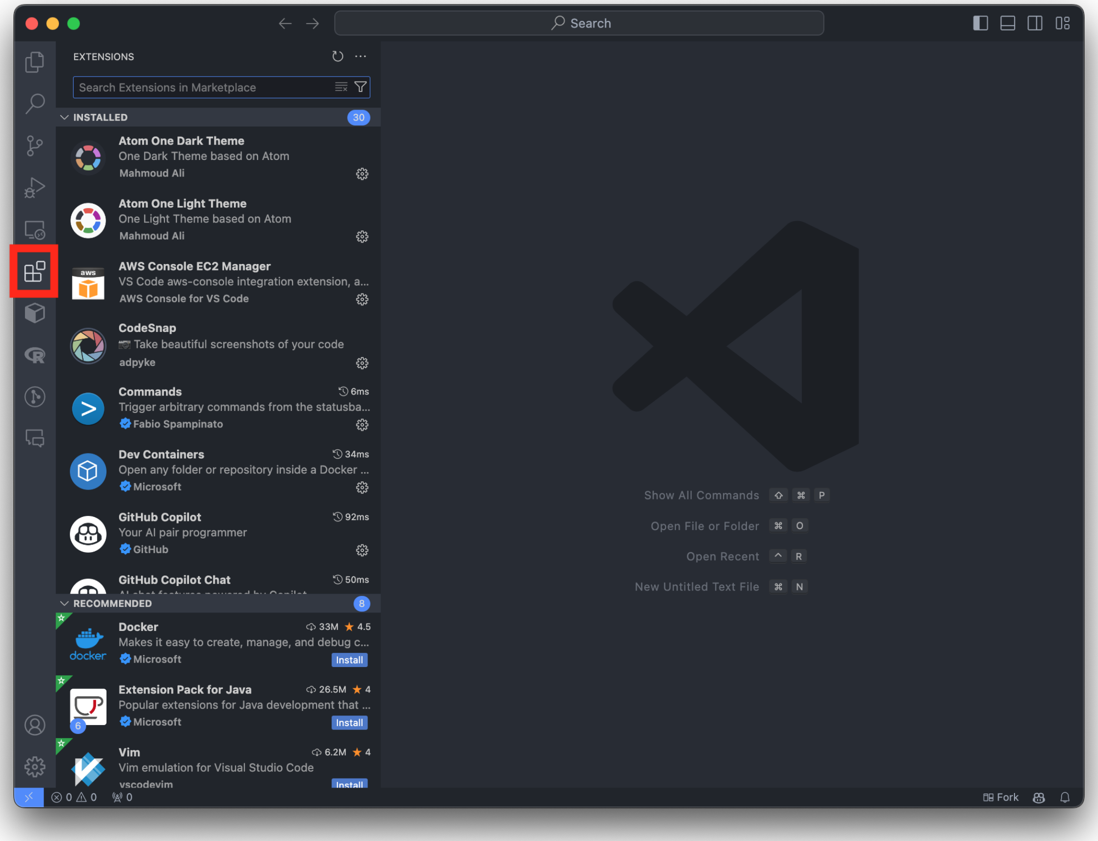


Figure 3: Installing Extensions in VSCode

3. Search for `Python` and click on the `Install` button for the extension that is provided by Microsoft.

4. Search for `Jupyter` and click on the `Install` button for the extension that is provided by Microsoft.

Testing the Installation

To test the installation, you can download a Juypter notebook from Moodle and open it in VS Code:

1. Open the Jupyter notebook in VS Code.

2. Click on `Select Kernel` in the top right corner of the notebook and choose the `ai-big-data-cemfi` kernel.

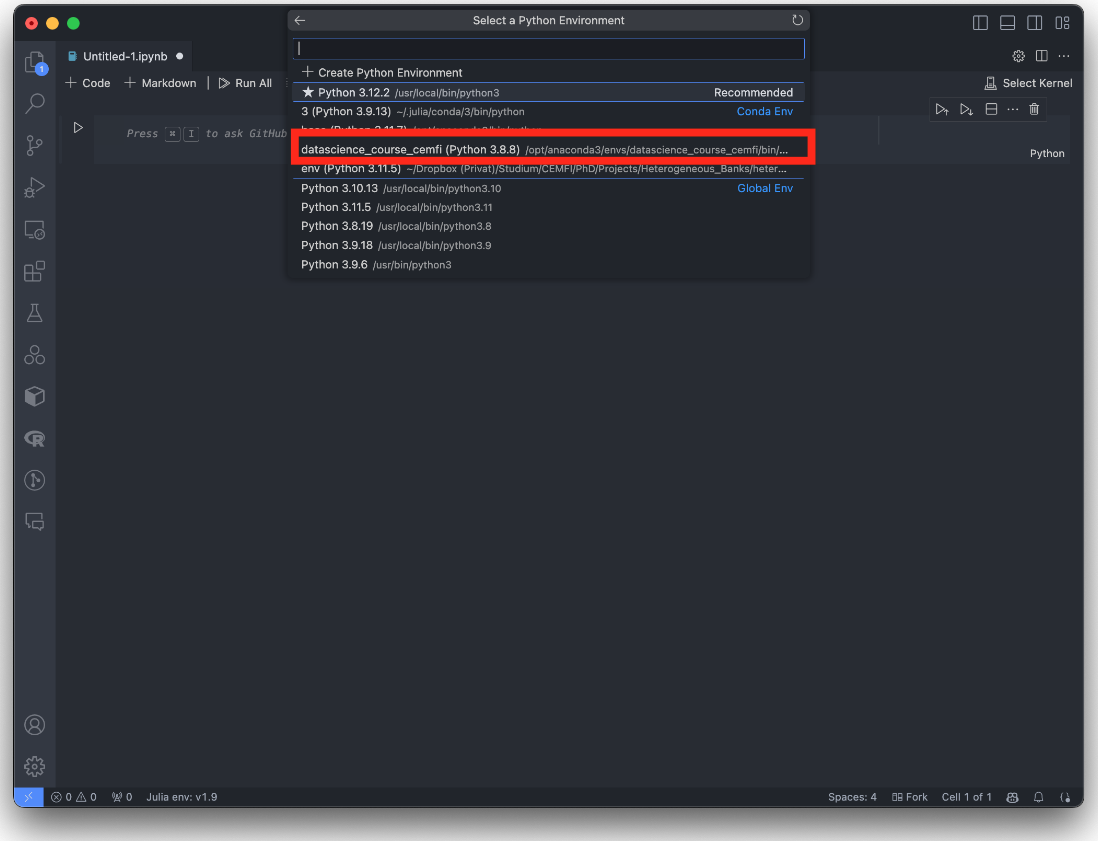

Select a Python Environment
+ Create Python Environment
★ Python 3.12.2 /usr/local/bin/python3 Recommended
3 (Python 3.9.13) ~/.julia/conda/3/bin/python Conda Env
datascience_course_cemfi (Python 3.8.8) /opt/anaconda3/envs/datascience_course_cemfi/bin/...
env (Python 3.11.5) ~/Dropbox (Privat)/Studium/CEMFI/PhD/Projects/Heterogeneous_Banks/heter...
Python 3.10.13 /usr/local/bin/python3.10 Global Env
Python 3.11.5 /usr/local/bin/python3.11
Python 3.8.19 /usr/local/bin/python3.8
Python 3.9.18 /usr/local/bin/python3.9
Python 3.9.6 /usr/bin/python3
Untitled-1.ipynb
+ Code
+ Markdown
Run All
Select Kernel
Python
Press ⌘ I to ask GitHub
Julia env: v1.9
Spaces: 4
Fork
Cell 1 of 1

Figure 4: VSCode Jupyter Kernel Selection

3. Run the first cell of the notebook by clicking on the `Execute Cell` button next to the cell on the left.

If you see the output of the cell (or a green check mark below the cell), the installation was successful.

💡 Running Jupyter Notebooks in the Browser

If you have issues running Jupyter notebooks in VSCode, you can also run them in the browser. To do this, you can open a *terminal window* on macOS/Linux or an *Anaconda Prompt* if you are on Windows and run the following command:

```
jupyter notebook
```

This will open a new tab in your default browser with the Jupyter notebook interface. You can then navigate to the folder where you have downloaded the course materials and open the notebooks from there.

Part I

Foundations

Chapter 1

Introduction to AI and Big Data

Artificial intelligence (AI) has seen remarkable progress in recent years, transforming from a specialized field into everyday technology that millions of people now use for writing, coding, and problem-solving. These advances have been fueled by **machine learning** (ML) methods with a **wide variety of applications**, including:

- Computer vision (e.g., image recognition, autonomous vehicles),
- Natural language processing (e.g., chatbots, translation, sentiment analysis),
- Speech recognition (e.g., voice assistants),
- Recommendation systems (e.g., Netflix, Amazon, Spotify),

and many more. These tools also have **many potential applications in economics and finance** and can be invaluable for extracting information from the ever-growing amounts of data available. As current (or future) Banco de España employees, you are in a unique position to work with large datasets that are often not available to the general public. This presents a unique **opportunity to apply these methods** to a wide range of problems.

While the field can be technical, **barriers to entry are not as high as they may seem**. Modern programming languages like Python, combined with powerful open-source libraries (e.g., scikit-learn, PyTorch), have made machine learning accessible to practitioners without requiring a deep background in mathematics or computer science. This course aims to provide you with the foundational knowledge and practical tools to apply machine learning methods to problems in economics and finance.

1.1 How is AI Relevant for You?

You might have heard of some well-known **advances in AI** from recent years:

- **Conversational AI**: ChatGPT, Claude, and Gemini respond to complex prompts and reason through multi-step problems

- **Image generation**: Midjourney, DALL-E 3, and Stable Diffusion create photorealistic images from text descriptions
- **Code assistants**: GitHub Copilot and Claude Code help programmers generate and edit code
- **Video generation**: Sora and Veo produce videos from text prompts

These are just a few examples of consumer-facing AI applications.

While these examples are impressive and can be very useful in various contexts. There is a wide range of potential applications that might be relevant for your work which go beyond these examples. For example, machine learning methods have been used **in practice** to

- **Predict** loan or firm defaults based on financial statements and alternative data,
- **Detect** fraud patterns in real-time transaction data,
- **Automate** document processing and information extraction from regulatory filings,
- **Monitor** news and social media for early warning signals, or
- **Forecast** macroeconomic indicators and stress test scenarios

to just name a few examples. Bank for International Settlements (2021) and Bank for International Settlements (2025) provide an overview of how machine learning methods have been used at central banks in recent years.

To give you a few more ideas from **academic research**, machine learning techniques have been used to, for example,

- Detect emotions in voices during press conferences after FOMC meetings (Gorodnichenko, Pham, and Talavera 2023),
- Identify Monetary Policy Shocks using Natural Language Processing (Aruoba and Drechsel 2022),
- Solve macroeconomic models with heterogeneous agents (Maliar, Maliar, and Winant 2021; Fernández-Villaverde, Hurtado, and Nuño 2023; Fernández-Villaverde et al. 2024; Kase, Melosi, and Rottner 2022), or
- Estimate structural models with the help of neural networks (Kaji, Manresa, and Pouliot 2023).

Varian (2014), Athey and Imbens (2019), Korinek (2023), and Dell (2025) provide further examples and discuss how these methods complement traditional econometrics.

Before diving into specific techniques, let's clarify some terminology and understand where machine learning fits within the broader field of artificial intelligence.

1.2 Overview of Artificial Intelligence

Artificial intelligence (AI), machine learning (ML), and deep learning (DL) are often used interchangeably in the media. However, they describe more narrow subfields (Microsoft 2024):

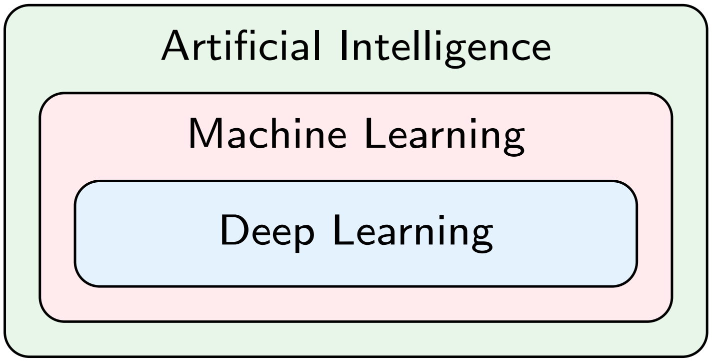


Figure 1.1: Artificial intelligence vs. Machine Learning vs. Deep Learning

- **Artificial Intelligence (AI)**: Any method allowing computers to imitate human behavior.
- **Machine Learning (ML)**: A subset of AI including methods that allow machines to improve at tasks with experience.
- **Deep Learning (DL)**: A subset of ML using neural networks with many layers allowing machines to learn how to perform tasks.

More recently, with the rise of **large language models (LLMs)** such as ChatGPT, several new terms have become popular:

- **Generative AI** refers to models that create new content—text, images, music, or video based on patterns learned from training data. ChatGPT and Midjourney are examples of generative AI.
- **Predictive AI** refers to models used to make predictions or classifications based on input data, such as predicting loan defaults or classifying images.
- **Agentic AI** refers to systems that can autonomously plan and execute multi-step tasks, use external tools, and take actions in the real world with minimal human oversight. Examples include AI coding assistants that can browse documentation, run tests, and edit files, or AI agents that can book travel or manage emails.

You may also encounter the term **Artificial General Intelligence (AGI)**, which refers to hypothetical systems capable of human-level reasoning across a wide range of tasks. Unlike current AI, which excels at specific tasks, AGI would generalize to novel problems without task-specific training. While AGI is a topic of active research and debate, no consensus exists on how close we are to achieving it or even how to define it precisely.[1] AGI and related concepts

[1] For more background, see Google Cloud's overview of AGI.

are beyond the scope of this course.

1.3 What is Machine Learning?

In this course, we will be mainly concerned with the subfield of artificial intelligence known as machine learning.

1.3.1 Definition

Murphy (2012) provides a simple definition of machine learning as

> [...] a **set of methods** that can **automatically detect patterns in data**, and then use the uncovered patterns to **predict future data**, or to **perform other kinds of decision making** under uncertainty [...]

Therefore, machine learning provides a range of methods for data analysis. In that sense, it is **similar to statistics or econometrics**.

A popular, albeit more technical, definition of ML is due to Mitchell (1997):

> A computer program is said to learn from experience E with respect to some class of tasks T, and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

In the context of this course, experience E is given by a dataset that we feed into a machine-learning algorithm, tasks T are usually some form of prediction that we would like to perform (e.g., loan default prediction), and the performance measure P is the measure assessing the accuracy of our predictions.

1.3.2 Relation of Machine Learning to Statistics and Econometrics

We have already mentioned that machine learning is similar to statistics and econometrics, in the sense that it provides a set of methods for data analysis. The **focus** of machine learning is more **on prediction rather than causality** meaning that in machine learning we are often interested in whether we can predict A given B rather than whether B truly causes A. For example, we could probably predict the sale of sunburn lotion on a day given the sales of ice cream on the previous day. However, this does not mean that ice cream sales cause sunburn lotion sales, it is just that the sunny weather on the first day causes both.

Varian (2014) provides another example showing the difference between prediction and causality:

> A classic example: there are often more police in precincts with high crime, but that does not imply that increasing the number of police in a precinct would increase crime. […] If our data were generated by policymakers who assigned police to areas with high crime, then the observed relationship between police and crime rates could be highly predictive for the *historical* data but not useful in predicting the causal impact of explicitly *assigning* additional police to a precinct.

Nevertheless, leaving problems aside where we are interested in causality, there is still a very large range of problems where we are interested in mere prediction, such as loan default prediction, or credit card fraud detection.

1.4 The Role of Big Data

The term **big data** refers to datasets that are too large or complex to be processed by traditional data processing methods. While there is no strict definition and the term is often used as a buzzword or marketing term, big data is often characterized by the **"three Vs"** (Laney 2001):

- **Volume**: The sheer amount of data generated and stored.
- **Velocity**: The speed at which new data is generated and needs to be processed.
- **Variety**: The different types of data (structured, unstructured, semi-structured) from various sources.

In the context of **banking and finance**, big data sources include:

- Transaction records (credit cards, payments, transfers)
- Customer interactions (call centers, online banking, mobile apps)
- Market data (stock prices, exchange rates, trading volumes)
- Alternative data (social media sentiment, satellite imagery, web scraping)
- Regulatory filings and reports

The combination of big data and machine learning creates a powerful synergy: **machine learning algorithms thrive on large datasets**, as more data generally leads to better pattern recognition and more accurate predictions. Conversely, traditional statistical methods often struggle with the high dimensionality and complexity of big data, making machine learning approaches increasingly attractive.

In this course, we will focus on machine learning methods that can, in principle, handle large datasets effectively. However, due to time constraints and the difficulty of the involved topics, we will put less emphasis on other important big data aspects such as distributed computing or data engineering. Varian (2014) provides a brief overview of big data and

1.5 History of Artificial Intelligence

Early contributions to the field reach back at least to McCulloch and Pitts (1943) and Rosenblatt (1958). They attempted to find mathematical representations of information processing in biological systems (Bishop 2006). As pointed out by Schmidhuber (2022), even earlier contributions in the form of linear regression[2] go back to work from Adrien-Marie Legendre and Johann Carl Friedrich Gauss around 1800, while some of the mathematical tools at the heart of today's AI models are even older than that. The **term "artificial intelligence"**, however, is much more recent and was **coined only in 1956** at a conference at Dartmouth College (Schmidhuber 2022).

1.5.1 Broad Developments in AI

Given the long history of AI, I will only show some broad developments in the field since the 1950s:

- **1950s-60s:** Early work on neural networks (e.g., perceptron by Rosenblatt (1958)) similar to what is used in deep learning today.
- **1970s-80s:** Development of expert systems (e.g., MYCIN). These are computer programs that mimic the decision-making abilities of a human expert in a specific domain. They use a set of rules and knowledge bases to make decisions and solve problems.
- **1990s-2000s:** Shift towards a more data-driven approach with statistical methods and machine learning (e.g., support vector machines, decision trees, etc.)
- **2010s:** Deep learning revolution (e.g., successful application of neural networks in many domains such as computer vision, natural language processing, etc.)
- **2020s:** Rise of large language models (e.g., GPT-3, ChatGPT) and generative AI (e.g., DALL-E, Midjourney)

1.5.2 Why has AI Become So Popular Recently?

The field has **grown substantially mainly in recent years** due to

- Advances in **computational power** of personal computers
- Increased availability of large datasets → **"big data"**
- Improvements in **algorithms**

Figure 1.2 shows how the **number of transistors on a microchip** has increased over time. This increase in computational power has allowed for more complex machine learning algorithms to be run on personal computers. This trend is often referred to as **Moore's Law** which states that the number of transistors on a microchip doubles approximately every two years. While

[2] As we will see this can be thought of as a basic form of an artificial neural network.

the trend has slowed down in recent years, the increase in computational power has been a key driver for the recent advances in AI.

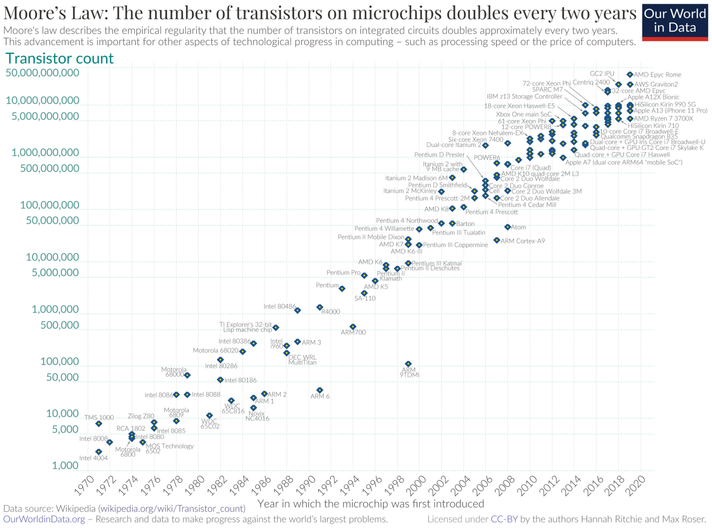


Figure 1.2: Moore's Law (Source: OurWorldinData.org)

Figure 1.3 shows how the **size of popular datasets in machine learning** has increased over time. Larger datasets allow machine learning algorithms to learn more complex patterns in the data which often leads to better performance. Note that the figure is in log scale meaning that for each tick on the y-axis, the dataset size increases by a factor of 10. Unfortunately, the figure ends in 2015 and dataset sizes have continued to grow since then. For example, **GPT-3** (released in 2020) was trained on a dataset with approximately **300 billion tokens** (words or word pieces) (Brown et al. 2020). Even more recent models will have been trained on even larger datasets. Furthermore, as discussed in the section on big data, thanks to **digitalization** and the rise of the **internet**, some companies now have access to **vast amounts of data** on user behavior (e.g., Amazon has data on purchases made by its customers, Netflix has data on what movies users watch, etc.), which can be used to train machine learning models.

The **need for large data sets still limits the applicability** to certain fields. For example, in macroeconomic forecasting, we usually only have quarterly data for 40-50 years. Conventional time series methods (e.g., ARIMA) often still tend to perform better than ML methods (e.g., neural networks). However, in this particular case, with the advent of pretrained time series models, e.g., Chronos, this might change in the (near) future.

On the algorithmic side, there have been many **improvements in optimization algorithm and model architectures** (e.g., development of transformers) that have allowed for more efficient training of machine learning models. These improvements have also contributed to the recent advances in AI. Furthermore, a strong community effort has gone into open-sourcing machine learning frameworks (e.g., PyTorch) and pre-trained models (e.g., BERT, GPT) which has lowered the barrier to entry for many practitioners.

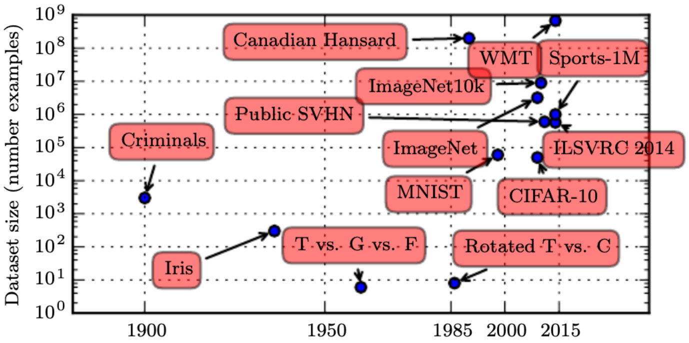


Figure 1.3: Data set sizes over time (Source: Goodfellow, Bengio, and Courville (2016))

1.6 Types of Learning

Machine learning methods are **commonly distinguished based on** the **tasks that we would like to perform**, and the **data that we have access to for learning** how to perform said task. ML methods are commonly categorized into

- **Supervised Learning**: Learn function $y = f(x)$ from data that you observe for x and y
- **Unsupervised Learning**: “Make sense” of observed data x
- **Reinforcement Learning**: Learn how to interact with the environment

Key for the distinction between supervised and unsupervised learning is whether we have access to **labeled data** (i.e., data where we observe both input features x and output labels y) or not, as the following example illustrates.

Example: Fraud Detection

Suppose you work in a bank’s fraud detection department and want to identify fraudulent credit card transactions. The approach you take depends on what data you have available:

- **Supervised Learning:** If you have historical transaction data where each transaction is labeled as “fraudulent” or “legitimate” (e.g., from past investigations or customer reports), you can train a model to learn the patterns that distinguish fraud from normal transactions. The model learns a function that maps transaction features (amount, location, time, merchant type, etc.) to a fraud prediction. Once trained, the model can classify new, unseen transactions.
- **Unsupervised Learning:** If you don’t have labels, perhaps because fraud is rare and hard to identify, or you’re looking for new types of fraud that haven’t been seen before, you can use unsupervised learning to find unusual patterns. For example, a clustering algorithm might group transactions by similarity and flag transactions that don’t fit well into any cluster as potential anomalies worth investigating.

Types of Learning in Practice

Machine learning models might **combine different types of learning**. In the context of the previous example of fraud detection, one might use unsupervised methods to detect anomalies and flag suspicious transactions, followed by human review that generates labels, which then feed into supervised models for more accurate classification.

A related approach is **semi-supervised learning**, which uses a small amount of labeled data together with a large amount of unlabeled data. This is particularly useful when labeling is expensive or time-consuming—for example, having experts manually classify thousands of regulatory documents. The model learns patterns from the abundant unlabeled data while using the limited labels to guide its understanding.

Another example of combining different learning methods is provided by large language models (LLMs) such as ChatGPT. LLMs are typically trained using a combination of self-supervised learning (a form of unsupervised learning), supervised learning, and reinforcement learning.

The focus of this course will be on supervised learning. Nevertheless, let's have a closer look at the three types of learning.

1.6.1 Supervised Learning

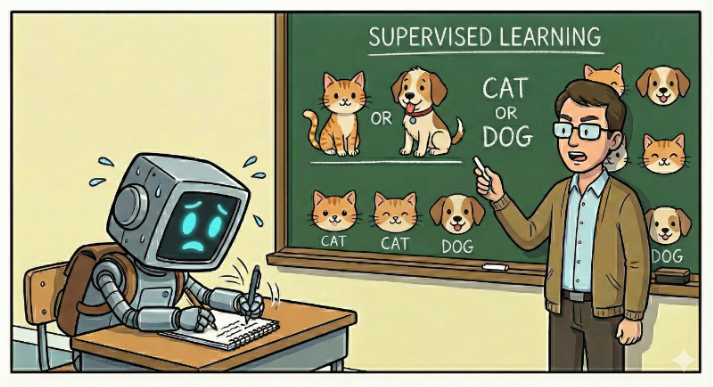


Figure 1.4: Supervised Learning

Supervised learning is probably the **most common** form of machine learning. In supervised learning, we have a **training dataset** consisting of input-output pairs (x_n, y_n) for $n = 1, \ldots, N$. The goal is to learn a function f that maps inputs x to outputs y.

The type of **function f might be incredibly complex**, e.g.

- From images of cats and dogs x to a classification of the image y (→ Figure 1.5)
- From text input x to some coherent text response y (→ ChatGPT)

- From text input x to a generated image y (→ Midjourney)
- From bank loan application form x to a loan decision y

Regarding **terminology**, note that sometimes

- Inputs x are called features, predictors, or covariates,
- Outputs y are called labels, targets, or responses.

Based on the **type of output**, we can distinguish between

- **Classification**: Output y is in a set of mutually exclusive labels (i.e., classes), i.e. $\mathcal{Y} = \{1, 2, 3, \dots, C\}$
- **Regression**: Output y is a real-valued quantity, i.e. $y \in \mathbb{R}$

Let's have a closer look at some examples of classification and regression tasks.

Classification

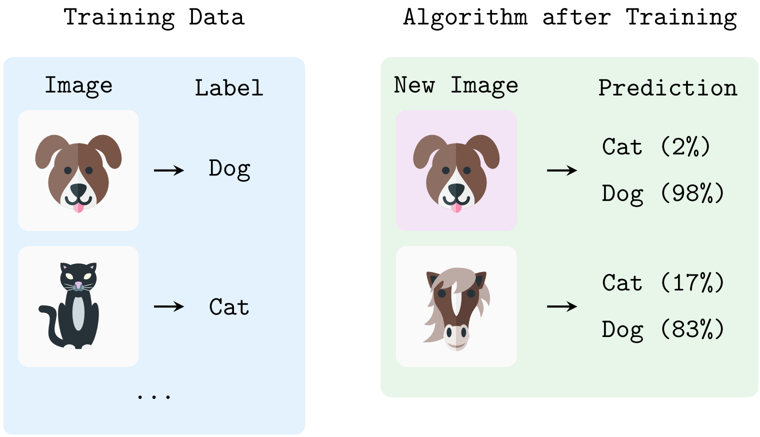
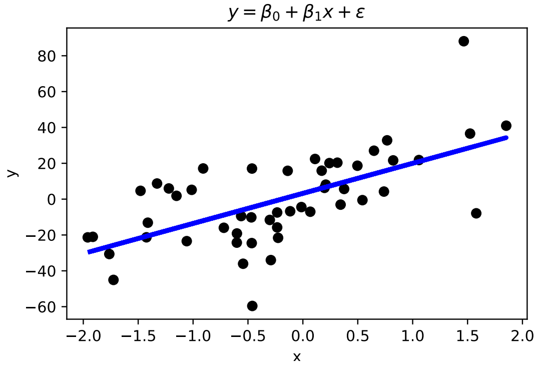


Figure 1.5: Training a machine learning algorithm to classify images of cats and dogs

Figure 1.5 shows an example of a **binary classification task**. The algorithm is trained on a dataset of images of cats and dogs. The goal is to predict the label (i.e., "cat" or "dog") of a new image (new in the sense that the images were not part of the training dataset). After training, the algorithm can predict the label of new images with a certain degree of accuracy. However, if you give the algorithm an image of, e.g., a horse it might mistakenly predict that it is a dog because the algorithm has never seen an image like that before and because it has been trained only for binary classification (it only knows two kinds of classes, "cats" and "dogs"). In this example, x would be an image in the training dataset and y would be the label of that image.

Extending the training dataset to also include images of horses with a corresponding label would turn the tasks into **multiclass classification**.

Regression



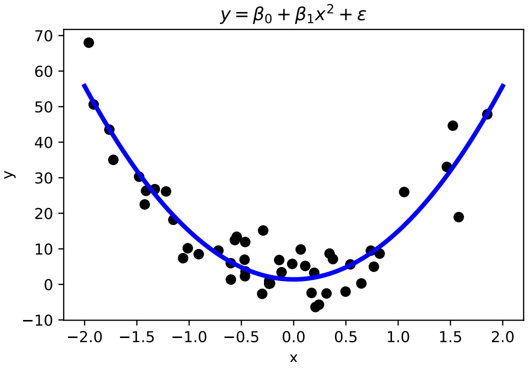


Figure 1.6: Linear and Polynomial Regression

In **regression tasks**, the variable that we want to predict is continuous. Linear and polynomial regression in Figure 1.6 are a form of supervised learning. Thus, you are already familiar with some basic ML techniques from the statistics and econometrics courses.

Another common way to solve regression tasks is to use **neural networks**, which can learn **highly non-linear relationships**. In contrast to, for example, polynomial regression, neural networks can learn these relationships **without the need to specify the functional form** (i.e., whether it is quadratic as in Figure 1.6) of the relationship. This makes them very flexible and powerful tools. We will have a look at neural networks later on.

1.6.2 Unsupervised Learning

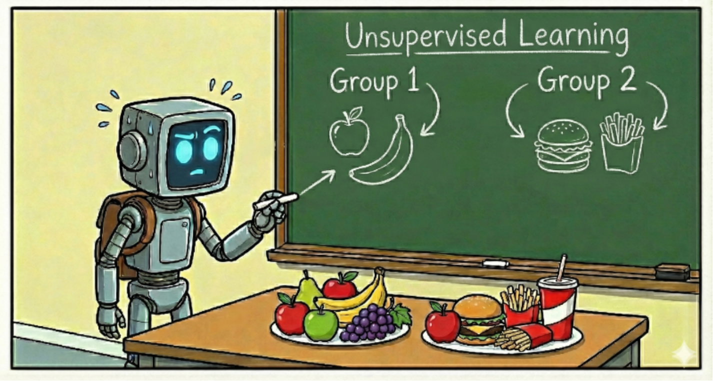


Figure 1.7: Unsupervised Learning

An issue with supervised learning is that we need labeled data which is often not available. **Unsupervised learning** is used to **explore data** and to **find patterns** that are not immediately obvious. For example, unsupervised learning could be used to find groups of customers with similar purchasing behavior in a dataset of customer transactions. Therefore, the task is to learn some structure in the data x. Note that we only have features in the dataset and no labels, i.e., the training dataset consists of N data points x_n.

Unsupervised learning tasks could be, for example,

- Finding **clusters** in the data, i.e. finding data points that are "similar" (→ clustering)
- Finding **latent factors** that capture the "essence" of the data (→ dimensionality reduction)

Let's have a look at some examples of clustering and dimensionality reduction.

Clustering

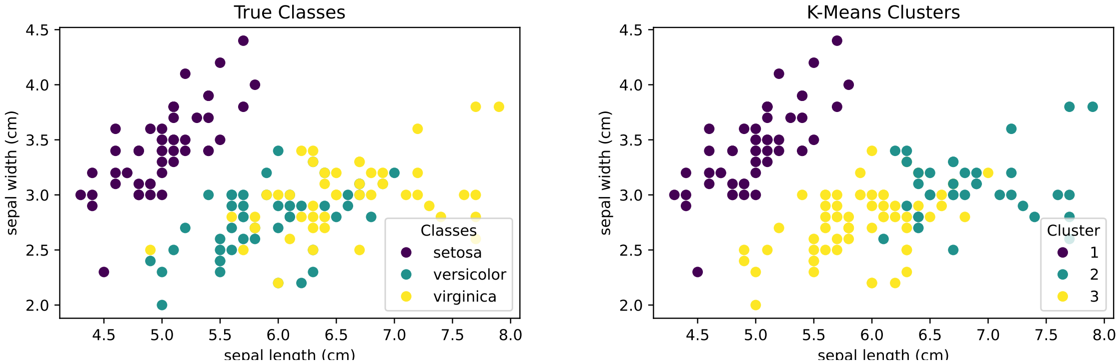


Figure 1.8: Clusters in data on iris flowers (left-hand side: true classes, right-hand side: k-means clusters)

Clustering is a form of unsupervised learning where the goal is to group data points into so-called clusters based on their similarity. We want to find clusters in the data such that observations within a cluster are more similar to each other than to observations in other clusters.

Figure 1.8 shows an example of a **clustering task**. The dataset consists of measurements of sepal (and petal) length and width of three species of iris flowers. The goal is to find clusters based on just the similarity in sepal and petal lengths and widths without relying on information about the actual iris flower species. The left-hand panel of Figure 1.8, shows the actual classification of the iris flowers. The right-hand side shows the result of a k-means clustering algorithm that groups the data points into three clusters.

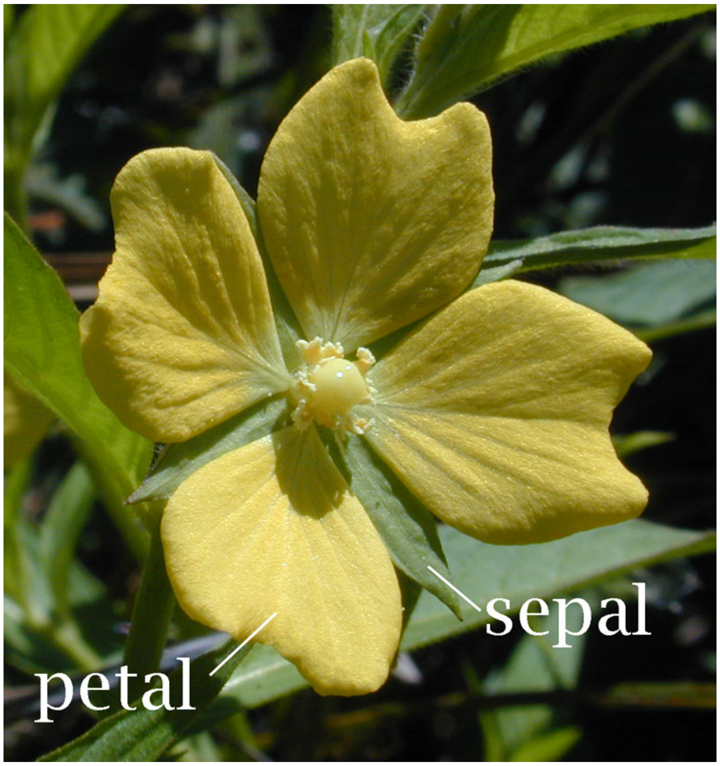


Figure 1.9: Petal vs Sepal (Source: Wikimedia)

Dimensionality Reduction

Suppose you observe data on house prices and many variables describing each house. You might observe, e.g., property size, number of rooms, room sizes, proximity to the closest supermarket, and hundreds of variables more. A ML algorithm (e.g., principal component analysis or autoencoders) could find the **unobserved factors that determine house prices**. These factors sometimes (but not always) have an interpretation. For example, a factor driving house prices could be *amenities*. This factor could summarize variables such as proximity to the closest supermarket, number of nearby restaurants, etc. Ultimately, **hundreds of explanatory variables** in the data set might be **represented by a small number of factors**.

1.6.3 Reinforcement Learning

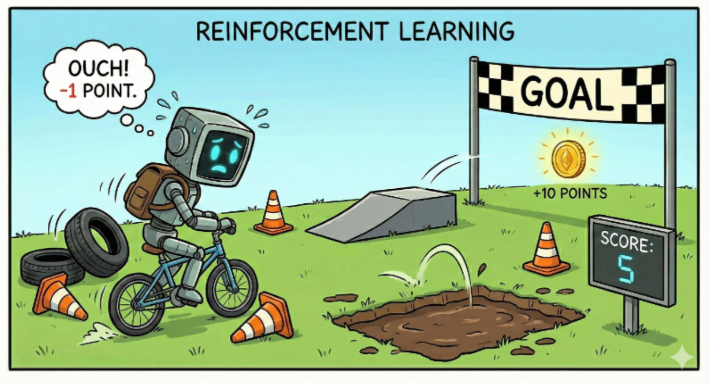


Figure 1.10: Reinforcement Learning

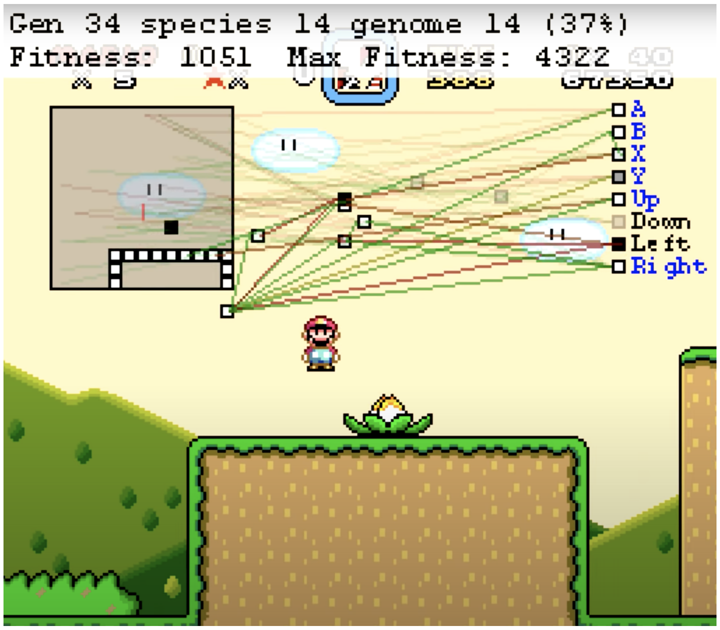
In **reinforcement learning**, an agent learns how to interact with its environment. The agent receives feedback in the form of rewards or penalties for its actions. The goal is to learn a policy that maximizes the total reward.

For example, a machine could learn to play chess using reinforcement learning

- **Input** x would be the current position (i.e., the position of pieces on the board)
- **Action** a would be the next move to make given the position
- One also needs to define a **reward** (e.g., winning the game at the end)
- Goal is then to find $a = \pi(x)$ to maximize the reward

This is also the principle behind **AlphaZero** that learned how to play Go and chess.

Another example is **MarI/O** which learned how to play Super Mario World. The algorithm learns to play the game by receiving feedback in the form of rewards (e.g., points for collecting coins, penalties for dying) and then improves in playing the game by “an advanced form of trial and error”.



Gen 34 species 14 genome 14 (37%)

Fitness: 1051 Max Fitness: 4322

Figure 1.11: MarI/O playing Super Mario World (Source: YouTube)

In this course, we will focus on supervised learning. However, we will look at some unsupervised learning techniques if time allows. Reinforcement learning is going beyond the scope of this course and will not be covered.

Mini-Exercise

Are the following tasks examples of supervised, unsupervised, or reinforcement learning?

1. Predicting the price of a house based on its size and location (given a dataset of house prices and features).
2. Finding groups of customers with similar purchasing behavior (given a dataset of customer transactions and customer characteristics).
3. Detecting fraudulent credit card transactions (given a dataset of *unlabeled* credit card transactions).
4. Detecting fraudulent credit card transactions (given a dataset of *labeled* credit card transactions).
5. Recognizing handwritten digits in the MNIST dataset (see next section).
6. Grouping news articles by topic based only on their content (without knowing the topics in advance).
7. Predicting whether a customer will cancel their subscription next month, given

historical data on customer behavior and cancellations.

8. Classifying emails as spam or not spam, using a dataset where each email is labeled as spam or not.
9. Training a robot to navigate a maze by receiving rewards for reaching the exit and penalties for hitting walls.
10. Identifying unusual trading patterns in financial markets to flag potential market manipulation (given a dataset of trades with no labels indicating manipulation).
11. Automatically categorizing incoming regulatory documents by topic (given a corpus of documents that have already been manually categorized).
12. Training an agent to set interest rates in a simulated economy, receiving rewards based on how well it stabilizes inflation and output over time.
13. Extracting the sentiment (hawkish vs. dovish) from central bank press releases (given a dataset of statements labeled by economists as hawkish or dovish).
14. Reducing hundreds of macroeconomic indicators to a smaller set of latent factors that capture the state of the economy.
15. Improving a chatbot's responses by having users rate each reply as helpful or unhelpful after their conversation.

Solution

1. **Supervised learning (regression):** We have labeled data (house prices) and want to predict a continuous value.
2. **Unsupervised learning (clustering):** No labels; we're finding structure in the data based on similarity.
3. **Unsupervised learning (anomaly detection):** Without labels, we can only identify unusual patterns that deviate from normal behavior.
4. **Supervised learning (classification):** With fraud/legitimate labels, we can train a classifier.
5. **Supervised learning (classification):** MNIST includes digit labels (0–9) for each image.
6. **Unsupervised learning (clustering/topic modeling):** No predefined topics; we discover them from the data.
7. **Supervised learning (classification):** Historical cancellation data provides labels (churned/retained).
8. **Supervised learning (classification):** Each email is labeled as spam or not spam.
9. **Reinforcement learning:** The robot learns from rewards and penalties through interaction with its environment.
10. **Unsupervised learning (anomaly detection):** No labels; we identify outliers that deviate from normal trading patterns.
11. **Supervised learning (classification):** We have labeled categories and want to assign new documents to them.

12. **Reinforcement learning:** The agent learns a policy through sequential interaction with the simulated economy, receiving rewards based on outcomes.
13. **Supervised learning (classification):** Labeled sentiment data allows us to train a classifier.
14. **Unsupervised learning (dimensionality reduction):** No labels; we're finding latent structure (e.g., via PCA or factor models).
15. **Reinforcement learning (RLHF):** The chatbot learns to generate better responses based on human feedback signals. This approach, known as Reinforcement Learning from Human Feedback (RLHF), is how models like ChatGPT are fine-tuned after initial training.

1.7 Popular Practice Datasets

There are many publicly available datasets that you can use to learn how to implement machine learning methods. Here are some well-known platforms with a large collection of datasets

- Kaggle,
- HuggingFace, and
- OpenML.

Another good source for practice datasets is the collection of datasets provided by scikit-learn. These datasets can be easily loaded into Python from the scikit-learn package. Furthermore, Murphy (2022) provides an overview of some well-known datasets that are often used in machine learning. For example, MNIST is a dataset of handwritten digits (see Figure 1.12) that is often used to test machine learning algorithms. The dataset consists of 60,000 training images and 10,000 test images. Each image is a 28x28 pixel image of a handwritten digit. The goal is to predict the digit in the image.

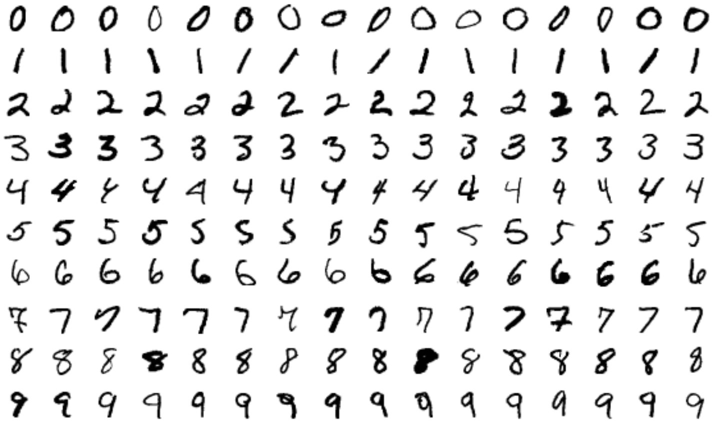


Figure 1.12: MNIST (Source: Wikimedia)

Chapter 2

Programming in Python

This section provides a brief introduction to programming in Python, covering the basics of the language, essential libraries for data analysis, and best practices for coding. The goal is to equip you with the skills needed to work with Python effectively in the context of artificial intelligence and big data.

Python has become the de facto standard for AI and data science due to its simplicity, readability, and rich ecosystem of specialized libraries. Throughout the course, we will use Python for various tasks, including data manipulation, visualization, statistical analysis, and implementing machine learning algorithms. By the end of this section, you should be comfortable with Python's core concepts and ready to tackle basic real-world AI challenges.

Note that programming is a skill that cannot be mastered overnight. It requires practice and continuous learning. I encourage you to experiment with the code examples provided in this section and to work through the exercises. Don't worry if things don't click immediately; programming fluency develops through repetition and problem-solving.

> **Note**
>
> The material in this section draws from the material developed by Alba Miñano-Mañero and extended by Jesús Villota Miranda, which they kindly prepared for another data science course that I taught at CEMFI.

2.1 Overview of Python

2.1.1 What is Python?

Python is a high-level, interpreted programming language created by Guido van Rossum and first released in 1991. It emphasizes code readability and simplicity through its clean syntax and use of significant whitespace, making it an ideal language for both beginners and experienced programmers. Python is a general-purpose language that excels across diverse domains—from automation to scientific computing and artificial intelligence. Its extensive standard library and vast ecosystem of third-party packages enable rapid development and prototyping. Today, Python is one of the most popular programming languages worldwide and has become the lingua franca of data science and machine learning, largely due to powerful libraries like NumPy, Pandas, scikit-learn, TensorFlow, and PyTorch.

2.1.2 Why Python and Not Other Languages?

While languages like R, Julia, and MATLAB, or sometimes even lower-level languages like C++ are used in data science and AI, Python offers distinct advantages for this course:

- **Unified ecosystem**: Python handles the entire data science workflow—from data collection and cleaning to modeling and deployment—within a single language, avoiding the friction of switching between tools.
- **Industry adoption**: Major tech companies and research institutions have standardized on Python for AI/ML work, making it a very marketable skill for practitioners.
- **Library maturity**: The ecosystem offers battle-tested libraries (NumPy, Pandas, scikit-learn) alongside cutting-edge deep learning frameworks (PyTorch, TensorFlow), providing both stability and innovation.
- **Gentle learning curve**: Readable syntax allows you to focus on concepts rather than wrestling with language complexity—particularly valuable when learning AI and big data techniques.
- **Community support**: With the largest community in data science, you'll find abundant tutorials, Stack Overflow answers, and open-source projects to learn from.

That said, other languages have their strengths: R, for example, excels in statistical analysis and visualization and Julia offers superior performance for numerical computing. Python strikes the best balance for our purposes: **accessible enough for newcomers yet powerful enough for production systems**.

2.1.3 Installation and Setup

For this course, we will primarily use Nuvolos, a cloud-based platform that provides a pre-configured Python environment with all necessary libraries and a VS Code interface. This eliminates installation headaches and ensures everyone has an identical setup. You can access Nuvolos through the link in the sidebar.

However, learning to set up Python locally is a valuable skill for future projects. If you wish to work on your own machine, here are the general steps to install Python and the required packages:

1. **Install Python via Anaconda/Miniconda**: Anaconda is a distribution that bundles Python with common data science packages. Miniconda is a lighter version that installs only Python and the conda package manager, allowing you to install packages as needed.
2. **Create a virtual environment**: Virtual environments isolate project dependencies, preventing version conflicts between projects. Use `conda env create -f https://ai-bigdata.joelmarbet.com/environment.yml` to create the environment required for this course.
3. **Install additional packages**: If required, you can install additional packages individually (e.g., `conda install numpy`).
4. **Set up VSCode**: Install Visual Studio Code and add the Python and Jupyter extensions. VSCode provides an excellent development experience with features like code completion, debugging, and integrated notebook support.

Detailed installation instructions on how to install the environment used in this course are available in the "Notes for Local Installation" PDF linked in the sidebar. For troubleshooting or platform-specific issues, consult the documentation or reach out after class or by email.

2.2 Development Environment

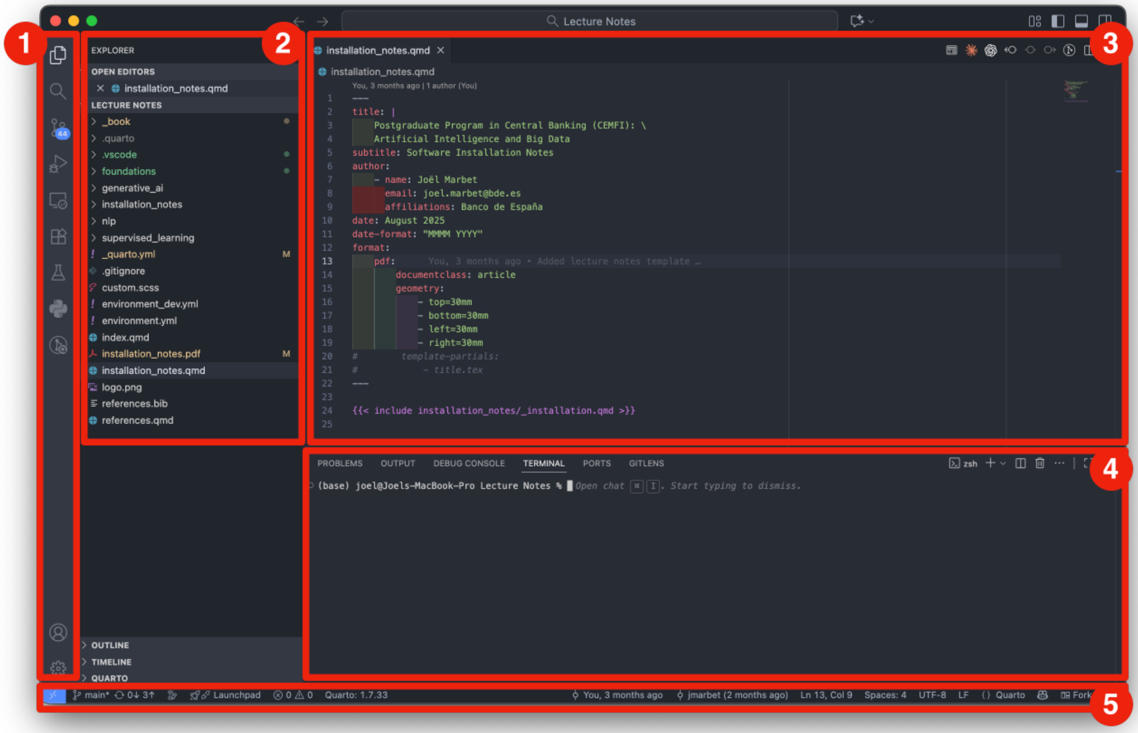
A good development environment significantly improves your productivity and learning experience. This section covers the main tools you'll encounter in this course.

2.2.1 Visual Studio Code (VSCode)

VSCode is a free, lightweight, yet powerful code editor that has become a developer favorite for many different programming languages. It combines the simplicity of a text editor with features traditionally found in full IDEs.

Figure 2.1 shows the main components of the VSCode interface:

1. **Activity Bar**: Located on the far left, it provides quick access to different views like Explorer, Search, Source Control, Extensions, and more.
2. **Side Bar**: Displays different views depending on the selected activity (e.g., file explorer, search results).
3. **Editor Area**: The central area where you write and edit your code files.
4. **Panel**: The bottom area that can display output, terminal, problems, and debug information.
5. **Status Bar**: Located at the bottom, it shows information about the current file, such as line number, encoding, and selected Python interpreter.



```
---
title: |
    Postgraduate Program in Central Banking (CEMFI): \
    Artificial Intelligence and Big Data
subtitle: Software Installation Notes
author:
    - name: Joël Marbet
      email: joel.marbet@bde.es
      affiliations: Banco de España
date: August 2025
date-format: "MMMM YYYY"
format:
    pdf:      You, 3 months ago • Added lecture notes template …
        documentclass: article
        geometry:
            - top=30mm
            - bottom=30mm
            - left=30mm
            - right=30mm
#       template-partials:
#           - title.tex
---

{{< include installation_notes/_installation.qmd >}}
```

```
(base) joel@Joels-MacBook-Pro Lecture Notes % ▌Open chat ⌘ I. Start typing to dismiss.
```

Figure 2.1: VSCode - Overview

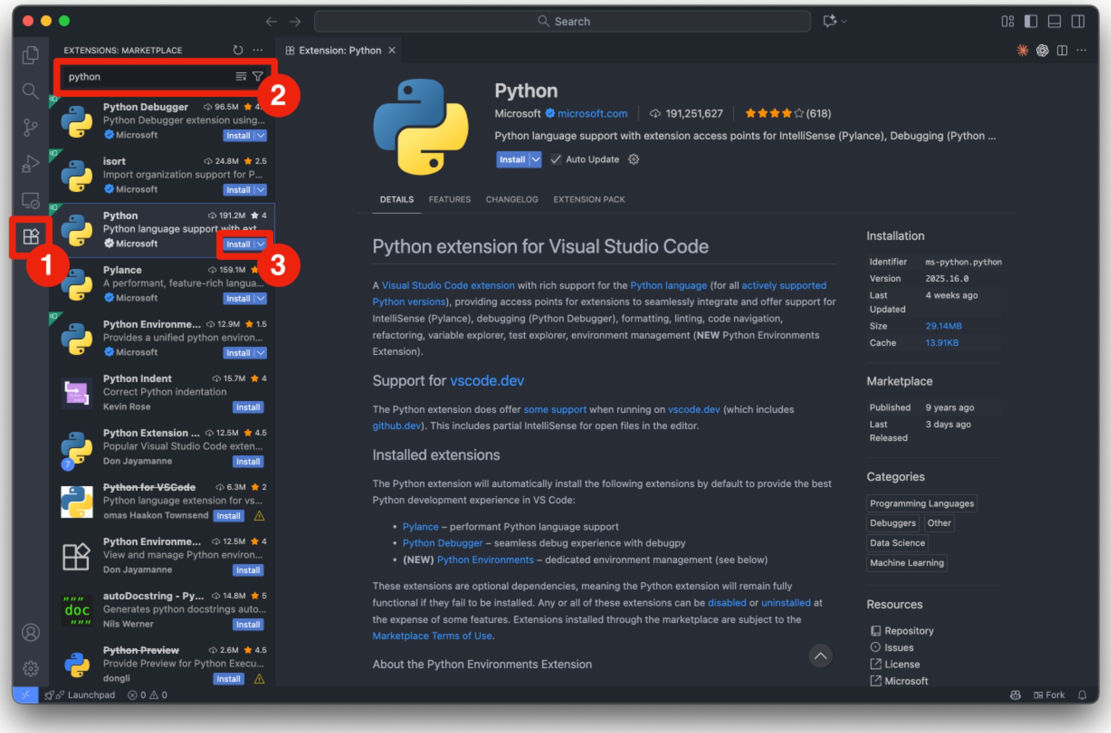
Note that not all elements are always visible; for example, the Panel is hidden by default and can be toggled as needed.

For Python development, you'll want to install the following extensions in VSCode:

- **Python Extension**: Provides IntelliSense (code completion), linting, debugging, and code navigation. It automatically detects your Python installations and allows you to select interpreters.
- **Jupyter Extension**: Enables you to create, edit, and run Jupyter notebooks directly within VSCode, eliminating the need to switch to a browser.

You can install extensions by clicking on the Extensions icon in the left sidebar and searching for them by name.

2.2.1.1 Install Python Extension


Search
EXTENSIONS: MARKETPLACE
python
1
2
3
Python Debugger 96.5M
Python Debugger extension using...
Microsoft
Install
isort 24.8M 2.5
Import organization support for P...
Microsoft
Install
Python 191.2M 4
Python language support with ext...
Microsoft
Install
Pylance 159.1M
A performant, feature-rich langua...
Microsoft
Install
Python Environme... 12.9M 1.5
Provides a unified python environ...
Microsoft
Install
Python Indent 15.7M 4
Correct Python indentation
Kevin Rose
Install
Python Extension ... 12.5M 4.5
Popular Visual Studio Code exten...
Don Jayamanne
Install
~~Python for VSCode~~ 6.3M 2
Python language extension for vs...
omas Haakon Townsend
Install
Python Environme... 12.5M 4
View and manage Python environ...
Don Jayamanne
Install
autoDocstring - Py... 14.8M 5
Generates python docstrings auto...
Nils Werner
Install
~~Python Preview~~ 2.6M 4.5
Provide Preview for Python Execu...
dongli
Install
Launchpad
0 0
Extension: Python
Fork
DETAILS
FEATURES
CHANGELOG
EXTENSION PACK
Install
Auto Update

Python

Microsoft microsoft.com | 191,251,627 | ★★★★☆ (618)

Python language support with extension access points for IntelliSense (Pylance), Debugging (Python ...

☑ Auto Update

Python extension for Visual Studio Code

A Visual Studio Code extension with rich support for the Python language (for all actively supported Python versions), providing access points for extensions to seamlessly integrate and offer support for IntelliSense (Pylance), debugging (Python Debugger), formatting, linting, code navigation, refactoring, variable explorer, test explorer, environment management (**NEW** Python Environments Extension).

Support for vscode.dev

The Python extension does offer some support when running on vscode.dev (which includes github.dev). This includes partial IntelliSense for open files in the editor.

Installed extensions

The Python extension will automatically install the following extensions by default to provide the best Python development experience in VS Code:

- Pylance – performant Python language support
- Python Debugger – seamless debug experience with debugpy
- **(NEW)** Python Environments – dedicated environment management (see below)

These extensions are optional dependencies, meaning the Python extension will remain fully functional if they fail to be installed. Any or all of these extensions can be disabled or uninstalled at the expense of some features. Extensions installed through the marketplace are subject to the Marketplace Terms of Use.

About the Python Environments Extension

Installation

Identifier	ms-python.python
Version	2025.16.0
Last Updated	4 weeks ago
Size	29.14MB
Cache	13.91KB

Marketplace

Published	9 years ago
Last Released	3 days ago

Categories

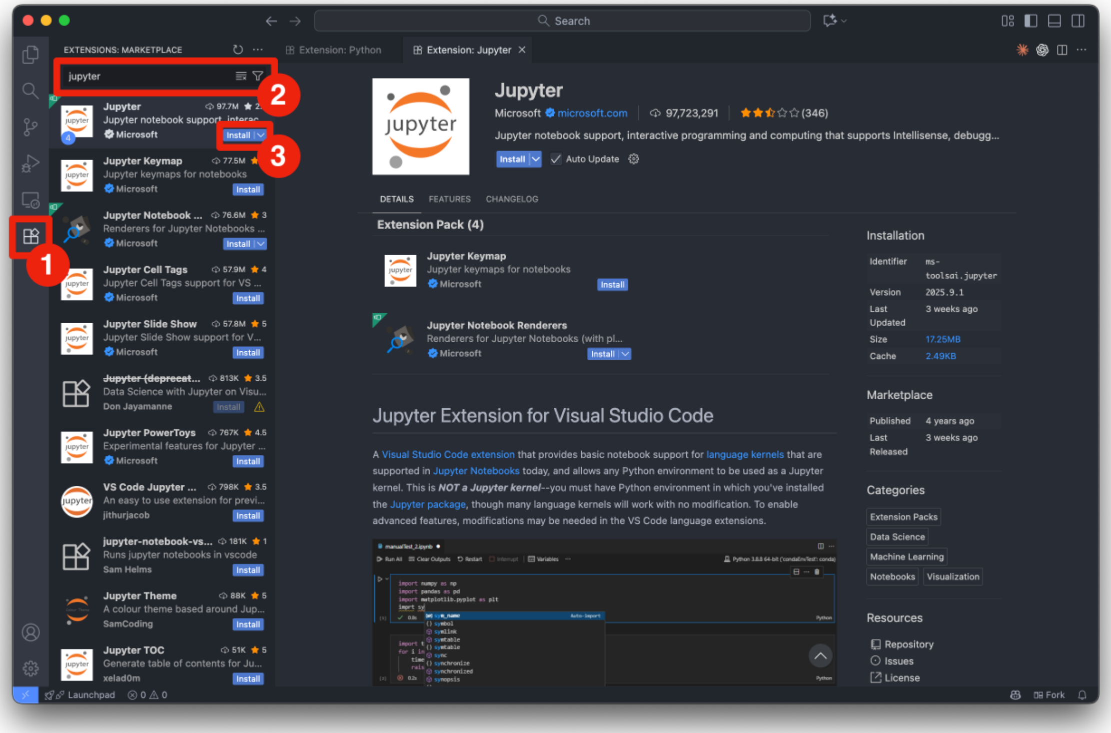
Programming Languages
Debuggers
Other
Data Science
Machine Learning

Resources

Repository
Issues
License
Microsoft

Figure 2.2: VSCode - Install Python Extension

2.2.1.2 Install Jupyter Extension



Jupyter

Microsoft microsoft.com | 97,723,291 | (346)

Jupyter notebook support, interactive programming and computing that supports Intellisense, debugg...

Install | ✓ Auto Update

Extension Pack (4)

Jupyter Keymap
Jupyter keymaps for notebooks
Microsoft

Jupyter Notebook Renderers
Renderers for Jupyter Notebooks (with pl...
Microsoft

Jupyter Extension for Visual Studio Code

A Visual Studio Code extension that provides basic notebook support for language kernels that are supported in Jupyter Notebooks today, and allows any Python environment to be used as a Jupyter kernel. This is ***NOT a Jupyter kernel***--you must have Python environment in which you've installed the Jupyter package, though many language kernels will work with no modification. To enable advanced features, modifications may be needed in the VS Code language extensions.

Installation

Identifier	ms-toolsai.jupyter
Version	2025.9.1
Last Updated	3 weeks ago
Size	17.25MB
Cache	2.49KB

Marketplace

Published	4 years ago
Last Released	3 weeks ago

Figure 2.3: VSCode - Install Jupyter Extension

We will primarily use VSCode within Nuvolos for this course, but you can also set it up locally following the installation instructions provided earlier. Note that the version on Nuvolos has an additional menu button at the top left, which provides access to menus to open files, settings, and other options. In the local version of VSCode, these options are available in the standard menu bar at the top of the window/screen.

2.2.2 Jupyter Notebooks

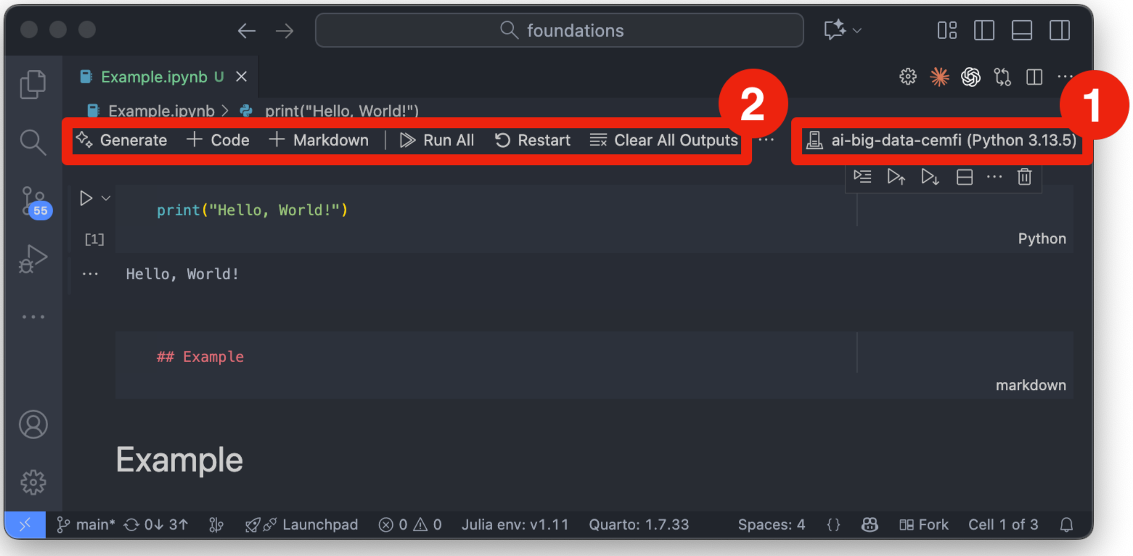
The main way we will interact with Python code in this course is through Jupyter Notebooks. Jupyter Notebooks, as well as the popular Jupyter Lab, are all part of the **Jupyter Project** which revolves around the provision of tools and standards for interactive computing across different computing languages (**Ju**lia, **Pyth**on, **R**).

Jupyter Notebooks are interactive documents that combine live code, visualizations, and explanatory text. They're ideal for exploratory data analysis and prototyping. They allow you to write and execute code in small chunks (cells), see immediate outputs, and document your thought process alongside the code. While Jupyter Notebooks are excellent for exploration and learning, they may not be the best choice for production code or large projects due to challenges with version control and code organization. However, they remain a popular tool in data science and AI for their interactivity and ease of use. We will execute Jupyter Notebooks within VSCode instead of the more traditional browser-based interface. The reason for this choice is to provide a unified development environment where you can seamlessly switch between writing notebooks and scripts, debugging code, and managing files. Furthermore, VSCode integrates well with recent AI-assisted coding tools, which can enhance your productivity.

Figure 2.4 shows an example of a Jupyter notebook opened in VSCode. To work with Jupyter notebooks in VSCode, follow these steps:

1. **Open Notebook**: Open a notebook file (file extension: `.ipynb`) or create a new one from the menu ("File" -> "New File" and then select "Jupyter Notebook").
2. **Choose Kernel**: Ensure that the kernel (Python interpreter) is set correctly. In this course, you should always select `ai-big-data-cemfi` as the kernel. You can change the kernel by clicking on the current kernel name (or "Select Kernel") in the top-right corner of the notebook interface (denoted by number 1 in Figure 2.4). Then, click on "Python Environments" and select `ai-big-data-cemfi` from the list.

If you have done this correctly, you should see `ai-big-data-cemfi` displayed as the selected kernel as shown in Figure 2.4.



ai-big-data-cemfi (Python 3.13.5)

```
print("Hello, World!")
```

```
Hello, World!
```

```
## Example
```

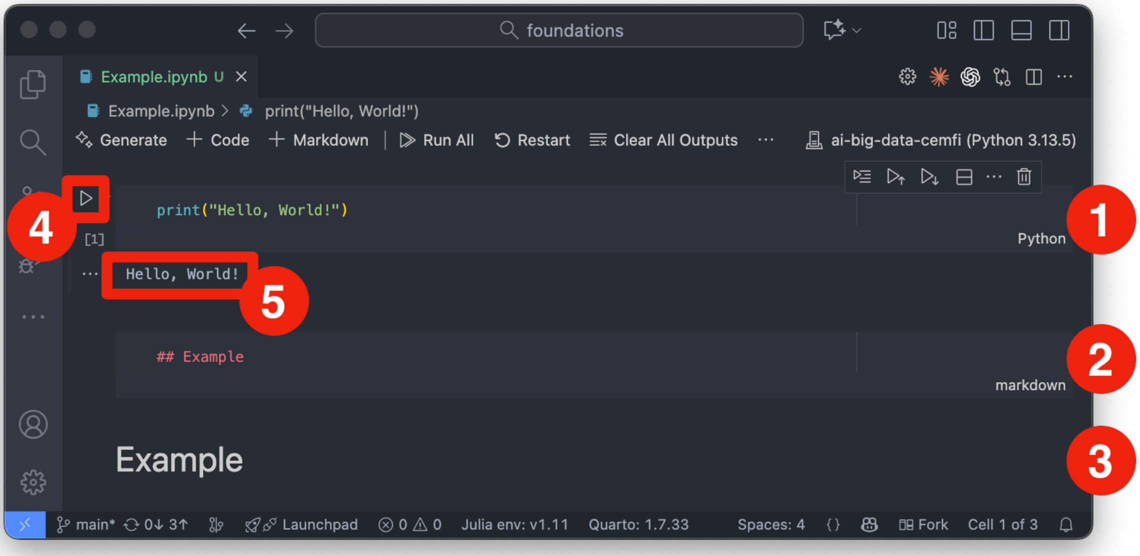
Example

Figure 2.4: VSCode - Jupyter Initial Setup

A Jupyter notebook consists of a sequence of cells, which can be of two main types:

- **Code Cells**: These cells contain executable code. You can run them individually, and the output (results, plots, error messages) will be displayed directly below the cell.
- **Markdown Cells**: These cells allow you to write formatted text using Markdown syntax. You can include headings, lists, links, images, and even LaTeX equations for mathematical notation.

These cells can be created from the toolbar at the top of the notebook interface (denoted by number 2 in Figure 2.4) or from the + button appearing under cells when hovering over them. From the toolbar you can also run cells, stop execution, restart the kernel, and perform other notebook-related actions. Cells can also be executed by selecting them and pressing `Shift-Enter` or by clicking the "Play" button in the toolbar (denoted by number 4 in Figure 2.5). Once you run a cell, the output will appear directly below it (denoted by number 5 in Figure 2.5). Markdown cells can be edited by double-clicking on them, and you can switch between code and markdown cell types using the dropdown menu in the toolbar. Number 2 and 3 in Figure 2.5 show markdown cells that are being edited and rendered, respectively. Number 1 in Figure 2.5 shows a code cell. Note that code cells have "Python" written in the bottom right corner to indicate the language being used.



```
print("Hello, World!")
```

```
Hello, World!
```

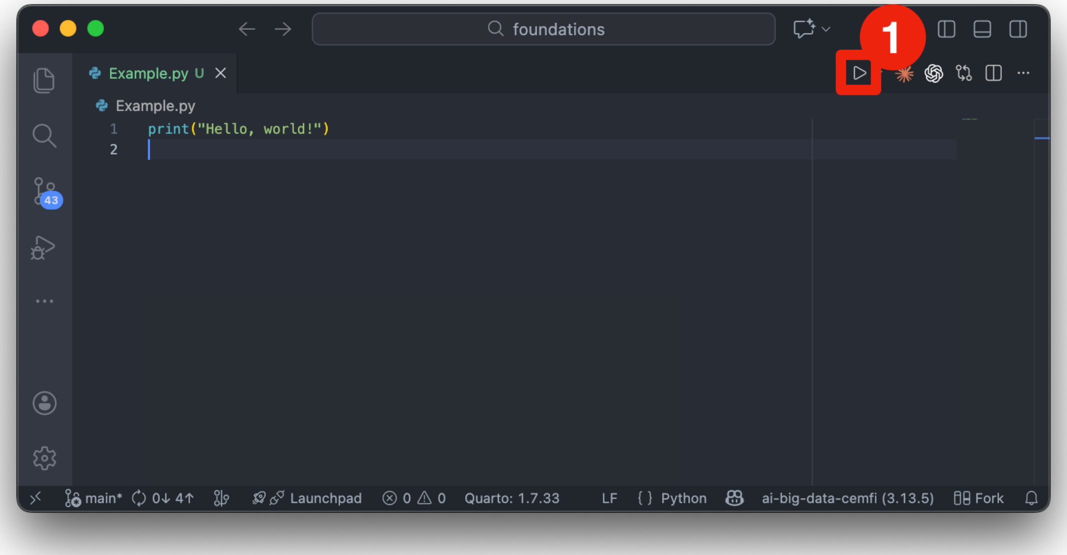
```
## Example
```

Example

Figure 2.5: VSCode - Jupyter Cells

2.2.3 Notebooks vs. Scripts

Another way to write and run Python code is through scripts. Scripts are plain text files with a `.py` extension that contain Python code. They are executed as a whole, either from the command line or within an IDE like VSCode. They are better suited for larger projects, production code, and automation tasks.



```python
print("Hello, world!")
```

Figure 2.6: VSCode - Python Script

Figure 2.6 shows an example of a Python script opened in VSCode. You can run the entire script by right-clicking anywhere in the editor and selecting “Run Python File in Terminal” or by clicking the “Play” button (denoted by number 1 in Figure 2.6). The output will appear in the integrated terminal at the bottom of the VSCode window.

When to use notebooks:

- Exploratory data analysis and visualization
- Step-by-step tutorials and documentation
- Quick prototyping and experimentation
- Presenting results with integrated plots and explanations

When to use scripts:

- Production code and automation
- Code that will be imported as modules

- Version control such as Git (notebooks can be harder to diff)
- Long-running processes without intermediate outputs

Best Practices:

- Keep notebooks focused on a single topic or analysis
- Use descriptive cell outputs and markdown for documentation
- Restart kernel and run all cells before sharing to ensure reproducibility

We will primarily use Jupyter notebooks for in-class exercises and exploratory tasks, but I will provide some Python scripts as examples. Understanding both formats is important for effective Python programming.

2.2.4 Google Colab

Google Colab is a free cloud-based Jupyter notebook environment that requires no setup and provides free access to GPUs. It's particularly useful for:

- Working on machines without Python installed
- Experimenting with deep learning models that require GPUs
- Collaborating with others in real-time (similar to Google Docs)
- Accessing more computational resources than your local machine provides

Limitations:

- Sessions timeout after periods of inactivity
- Files are stored in Google Drive or must be re-uploaded each session
- Not suitable for long-running jobs or production workflows
- Might not work in certain restricted corporate environments

If you have trouble installing the environments locally, Google Colab can be a good alternative. To use Colab, simply navigate to colab.research.google.com. There you can create a new notebook or upload an existing one. I will provide links to the notebooks used in this course that you can open directly in Colab if needed. However, Nuvolos is the preferred environment for this course and will give you the best experience.

2.3 Python Fundamentals

Python is an **interpreted language**. By this we mean that the Python interpreter will run a program by executing the source code line-by-line without the need for compilation into machine code beforehand. Furthermore, Python is an **Object-Oriented Programming (OOP)** language. Everything we define in our code exists within the interpreter as a Python object, meaning it has associated **attributes** (data) and **methods** (functions that operate on that data). We will see these concepts in more detail later.

First, let's have a look at the basics of any programming language. All programs consist of the following

- Variables,
- Functions,
- Loops, and
- Conditionals.

2.3.1 Variables

Variables are basic elements of any programming language. They

- store information,
- can be manipulated by the program, and
- can be of different types, e.g. integers, floating point numbers (floats), strings (sequences of characters), or booleans (true or false)

2.3.1.1 Creating Variables

Python is **dynamically typed**, meaning you don't need to declare variable types explicitly. The interpreter infers the type based on the assigned value. For example, the following code creates a variable `x` and assigns it the integer value `100`. The `type()` function is then used to check the type of the variable.

```python
x = 100
type(x)
```

```
<class 'int'>
```

The Python interpreter output `int`, indicating that `x` is of type integer.

As the example above shows, you can create a variable by simply assigning a value to it using the equals sign (`=`). What happens under the hood is that Python creates an object in memory to store the value `100` and then creates a reference (the variable name `x`) that points to that object. When you later use the variable `x` in your code, Python retrieves the value from the memory location that `x` references. For example, we can then do computations with `x`:

```python
y = x + 50
print(y)
```

```
150
```

Python retrieved the value of `x` (which is `100`), added `50` to it, and assigned the result to the new variable `y`.

Note that you can reassign variables to new values or even different types. For example, you can change the value of `x` simply by assigning a new value to it

```python
x = 200
print(x)
```

```
200
```

Note that now `x` points to a new object in memory with the value `200`. The previous object with the value `100` will be automatically cleaned up by Python's garbage collector if there are no other references to it. This might not seem important now, but there are some implications of this behavior when working with mutable objects, which we will cover later.

2.3.1.2 Naming Variables

The process of naming variables is an important aspect of programming. Good variable names enhance code readability and maintainability, making it easier for others (and yourself) to understand the purpose of each variable.

For example, consider the following two variable names

```python
a = 25
number_of_students = 25
```

The first variable name, `a`, is vague and does not convey any information about what it represents. In contrast, `number_of_students` is descriptive and clearly indicates that the variable holds the count of students. This makes the code more understandable, especially in larger programs where many variables are used.

Python imposes certain rules on how variable names can be constructed:

1. They must start with a letter (a-z, A-Z) or an underscore (_).
2. They can only contain letters, numbers (0-9), and underscores.
3. They cannot be the same as Python's reserved keywords (e.g., `if`, `else`, `while`, `for`, etc.). `help(keywords)` will show which words are reserved.
4. Variable names are case-sensitive, meaning that `Variable`, `variable`, and `VARIABLE` would be considered different variables.

In addition to these rules, good practices for naming variables include to

- Use meaningful and descriptive names that convey the purpose of the variable
- Use lowercase letters and separate words with underscores (`snake_case`) for better readability (some programmers use `camelCase`, but snake_case is preferred in Python)
- Avoid using single-letter names except for loop counters or very short-lived variables
- Avoid using built-in function names because that will overwrite the function (i.e., if we write `type` we will no longer be able to use `type` to access the `type` of variables)
- Be consistent with naming conventions throughout your codebase
- While you can use names in any language, English is generally preferred so that anyone can follow the code

The following code snippet lists all reserved keywords in Python that cannot be used as variable names

```python
import keyword

for kw in keyword.kwlist:
    print(kw)
```

```
False
None
True
and
as
assert
async
await
break
class
continue
def
del
elif
else
except
finally
for
from
global
if
import
in
is
lambda
```

```
nonlocal
not
or
pass
raise
return
try
while
with
yield
```

Make sure you don't use any of these words as variable names in your code.

2.3.1.3 Basic Data Types

Python has several built-in data types that are commonly used:

- **Integers (`int`)**: Whole numbers, e.g., `42`, `-7`
- **Floating-point numbers (`float`)**: Numbers with decimal points, e.g., `3.14`, `-0.001`
- **Complex numbers (`complex`)**: Numbers with real and imaginary parts, e.g., `2 + 3j`
- **Strings (`str`)**: Sequences of characters enclosed in single or double quotes, e.g., `'Hello, World!'`, `"Python"`
- **Booleans (`bool`)**: Logical values representing `True` or `False`

Since Python is dynamically typed, the creation of variables of these types is straightforward, as shown in the following examples:

```python
this_is_int = 5
type(this_is_int)
```

```
<class 'int'>
```

```python
this_is_float = 3.14
type(this_is_float)
```

```
<class 'float'>
```

```python
this_is_complex = 2 + 3j
type(this_is_complex)
```

```
<class 'complex'>
```

```
this_is_str = "Hello, Python!"
type(this_is_str)
```

```
<class 'str'>
```

```
this_is_bool = True
type(this_is_bool)
```

```
<class 'bool'>
```

Note that boolean values are special in the sense that they are equivalent to integers: `True` is equivalent to `1` and `False` is equivalent to `0`. This means you can perform arithmetic operations with boolean values, and they will behave like integers in those contexts.

There is another data type called `NoneType`, which you might encounter. It represents the absence of a value and is created using the `None` keyword.

```
this_is_none = None
type(this_is_none)
```

```
<class 'NoneType'>
```

You can also create more complex data types, which we will cover in the section on data structures.

2.3.1.4 Basic Operations

A key element of programming is manipulating the variables you create. Python supports various basic operations for different data types, including arithmetic operations for numbers, string operations for text, and boolean operations for logical values.

Arithmetic Operations: You can perform arithmetic operations on integers and floats using operators like `+`, `-`, `*`, `/`, `//` (floor division), `%` (modulus), and `**` (exponentiation).

```
a = 10
b = 3
```

```
sum_result = a + b # Addition
print(sum_result)
```

```
13
```

```
diff_result = a - b # Subtraction
print(diff_result)
```

```
7
```

```
prod_result = a * b # Multiplication
print(prod_result)
```

```
30
```

```
div_result = a / b # Division
print(div_result)
```

```
3.3333333333333335
```

```
floor_div_result = a // b # Floor Division
print(floor_div_result)
```

```
3
```

```
mod_result = a % b # Modulus
print(mod_result)
```

```
1
```

```
exp_result = a ** b # Exponentiation
print(exp_result)
```

```
1000
```

String Operations: Strings can be concatenated using the + operator and repeated using the * operator.

```
str1 = "Hello, "
str2 = "World!"
concat_str = str1 + str2 # Concatenation
print(concat_str)
```

```
Hello, World!
```

Sometimes, you may want to repeat a string multiple times

```
repeat_str = str1 * 3 # Repetition
print(repeat_str)
```

```
Hello, Hello, Hello,
```

Another useful operation is string interpolation, which allows you to embed variables within strings. This can be done using f-strings (formatted string literals) by prefixing the string with `f` and including expressions inside curly braces `{}`.

```
name = "Alba"
age = 30
intro_str = f"Her name is {name} and she is {age} years old."
print(intro_str)
```

```
Her name is Alba and she is 30 years old.
```

Boolean Operations: You can use logical operators like `and`, `or`, and `not` to combine or negate boolean values.

```
bool1 = True
bool2 = False
and_result = bool1 and bool2 # Logical AND
print(and_result)
```

```
False
```

```
or_result = bool1 or bool2 # Logical OR
print(or_result)
```

```
True
```

```
not_result = not bool1 # Logical NOT
print(not_result)
```

```
False
```

To compare values, you can use comparison operators like `==` (equal to), `!=` (not equal to), `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to).

```
a = 10
b = 20
```

```
eq_result = (a == b) # Equal to
print(eq_result)
```

```
False
```

```
neq_result = (a != b) # Not equal to
print(neq_result)
```

```
True
```

```
lt_result = (a < b) # Less than
print(lt_result)
```

```
True
```

```
gt_result = (a > b) # Greater than
print(gt_result)
```

```
False
```

```
le_result = (a <= b) # Less than or equal to
print(le_result)
```

```
True
```

```
ge_result = (a >= b) # Greater than or equal to
print(ge_result)
```

```
False
```

Note that the result of comparison operations is always a boolean value (`True` or `False`). This will be useful when we discuss conditional statements later.

⚠ Warning

Be careful not to confuse the assignment operator `=` with the equality comparison operator `==`. The single equals sign `=` assigns a value to a variable, while the double equals sign `==`

> checks if two values are equal and returns a boolean result.

We can also combine multiple comparison operations using logical operators. For example, to check if a number is within a certain range, we can use the `and` operator

```python
num = 15
is_in_range = (num > 10) and (num < 20)
print(is_in_range)
```

```
True
```

This checks if `num` is greater than `10` and less than `20`, returning `True` if both conditions are met. Of course, we can also use `or` to check if at least one condition is met or `not` to negate a condition.

2.3.2 Functions

Functions are reusable blocks of code that perform a specific task. They help organize code, improve readability, and allow for code reuse. In Python, you define a function using the `def` keyword, followed by the function name and parentheses containing any parameters. For example, here is a simple function that takes two arguments, performs a calculation, and returns the result

```python
def function_name(arg1, arg2):
  r3 = arg1 + arg2
  return r3
```

Note that the indentation (whitespace at the beginning of a line) is crucial in Python, as it defines the scope of the function. The code block inside the function must be indented consistently. In the example above, two spaces are used for indentation, but tabs or four spaces are also common conventions. VSCode will automatically convert tabs to spaces based on your settings and the convention used in the file.

Suppose we want to create a function that greets a user by their name. We can define such a function as follows

```python
def greet(name):
  greeting = f"Hello, {name}!"
  return greeting
```

You can then call the function by passing the required argument

```
message = greet("Alba")
print(message)
```

```
Hello, Alba!
```

We could also define the function without a return value and simply print the greeting directly

```
def greet_print(name):
  print(f"Hello, {name}!")
```

You can call this function in the same way

```
greet_print("Alba")
```

```
Hello, Alba!
```

We can also define functions with multiple outputs by returning a tuple of values. For example, here is a function that takes two numbers and returns both their sum and product

```
def sum_and_product(x, y):
  sum_result = x + y
  product_result = x * y
  return sum_result, product_result
```

You can call this function and unpack the returned values into separate variables

```
s, p = sum_and_product(5, 10)
print(f"Sum: {s}, Product: {p}")
```

```
Sum: 15, Product: 50
```

or you can capture the returned tuple in a single variable

```
result = sum_and_product(5, 10)
print(f"Result: {result}")
```

```
Result: (15, 50)
```

You can define functions with multiple `return` statements to handle different conditions. For example, here is a function that checks if a number is positive, negative, or zero and returns an appropriate message

```
def check_number(num):
  if num > 0:
    return "Positive"
  elif num < 0:
    return "Negative"
  else:
    return "Zero"
```

You can call this function with different numbers to see the results

```
print(check_number(10))   # Output: Positive
```

```
Positive
```

```
print(check_number(-5))   # Output: Negative
```

```
Negative
```

```
print(check_number(0))    # Output: Zero
```

```
Zero
```

When you pass a variable to a function, the function receives a local copy of that value. Modifying this copy inside the function does not affect the original variable outside. However, if you need to modify a variable defined outside the function (a global variable), you must explicitly declare it using the global keyword. The difference between local and global variables is also called the scope of a variable. The following example illustrates the difference

```
global_var = 10

def edit_input(input_var):

    # Access the input variable
    print("Input you gave me", input_var)

    input_var = input_var + 5  # This modifies the local copy of input_var and not global_var
    print("Inside the function - modified input_var:", input_var)

    return input_var  # Return the modified value
```

```
def edit_global(input_var):

    global global_var # Make global_var accessible inside the function

    # Access the input variable
    print("Input you gave me", input_var)

    global_var = global_var + input_var  # This modifies the global variable
    print("Inside the function - modified global_var:", global_var)

    return None

# Call the function
edit_input(global_var)
```

```
Input you gave me 10
Inside the function - modified input_var: 15
15
```

```
print("Outside the function - global_var:", global_var)
```

```
Outside the function - global_var: 10
```

```
# Call the function
edit_global(global_var)
```

```
Input you gave me 10
Inside the function - modified global_var: 20
```

```
print("Outside the function - global_var:", global_var)
```

```
Outside the function - global_var: 20
```

Oftentimes it is better to avoid global variables if possible, as they can lead to code that is harder to understand and maintain. Instead, prefer passing variables as arguments to functions and returning results. For example, if you would like to modify the value of global_var, you could simply assign the returned value of the function to it

```
global_var = edit_input(global_var)
```

```
Input you gave me 20
Inside the function - modified input_var: 25
```

```
print("Outside the function - global_var:", global_var)
```

```
Outside the function - global_var: 25
```

Functions can also have **default arguments**, which are used if no value is provided when the function is called. For example, here is a function that greets a user with a default name if none is provided

```
def greet_with_default(name="Guest"):
  print(f"Hello, {name}!")

greet_with_default()
```

```
Hello, Guest!
```

```
greet_with_default("Jesus")
```

```
Hello, Jesus!
```

We used the same function, once without providing an argument (so it uses the default value “Guest”) and once with a specific name (“Jesus”).

We can also use **keyword arguments** to call functions. This allows us to specify the names of the parameters when calling the function, making it clear what each argument represents. For example

```
def introduce(name, age):
  print(f"My name is {name} and I am {age} years old.")

introduce(name="Alba", age=30)
```

```
My name is Alba and I am 30 years old.
```

We can even change the order of the arguments when using keyword arguments, as shown above. You can also mix positional and keyword arguments, but positional arguments must come before keyword arguments.

```
introduce("Alba", age=30) # This works
```

```
My name is Alba and I am 30 years old.
```

```
#introduce(age=30, "Alba") # This will raise a SyntaxError
```

Positional arguments must be provided in the correct order, starting from the first parameter defined in the function. If you try to provide them in the wrong order, Python will raise a `TypeError`. For example, the following code will raise an error because the first argument is expected to be `name`, but we intended to provide an integer for `age`.

```
#introduce(30, name="Alba") # This will raise a TypeError
```

Finally, note that the function needs to be defined before it is called in the code. If you try to call a function before its definition, Python will raise a `NameError` indicating that the function is not defined.

```
#test_function()  # This will raise a NameError

def test_function():
  print("This is a test function.")
```

But the following will work correctly

```
def test_function():
  print("This is a test function.")

test_function()  # This will work correctly
```

```
This is a test function.
```

For this reason, function definitions are often placed at the beginning of a script or notebook cell, before any calls to those functions.

2.3.3 Conditional statements

Conditional statements allow you to control the flow of your program based on certain conditions. In Python, you can use `if`, `elif`, and `else` statements to execute different blocks of code depending on whether a condition is true or false. We have already seen an example of this in the `check_number` function above.

In the following example, the `do_something()` function will only be executed if `condition` evaluates to `True`, while `do_some_other_thing()` will always be executed.

```
if condition:
  do_something()

do_some_other_thing()
```

It is important to note that Python uses indentation to define the scope of code blocks. The code inside the `if` statement must be indented consistently to indicate that it belongs to that block.

```
a = 10

if a > 5:
  print("a is greater than 5")
  print("This line is also part of the if block")
```

```
a is greater than 5
This line is also part of the if block
```

```
print("This line is outside the if block")
```

```
This line is outside the if block
```

You can also nest `if` statements within each other to create more complex conditions. For example

```
a = 10

if a > 5:
  if a < 15:
    print("a is between 5 and 15")
  else:
    print("a is greater than or equal to 15")
else:
  print("a is less than or equal to 5")
```

```
a is between 5 and 15
```

Here, we first check if `a` is greater than `5`. If that condition is true, we then check if `a` is less than `15`. Depending on the outcome of these checks, different messages will be printed. Compared to the previous example, we also used an `else` statement to handle the case where `a` is not less than `15`.

We can also use `elif` (short for "else if") to check multiple conditions in a more concise way. For example

```
a = 10

if a < 5:
  print("a is less than 5")
elif a < 15:
  print("a is between 5 and 15")
else:
  print("a is greater than or equal to 15")
```

```
a is between 5 and 15
```

To reach the `elif` block, the first `if` condition must evaluate to `False`. If it evaluates to `True`, the code inside that block will be executed, and the rest of the conditions will be skipped. If none of the conditions are met, the code inside the `else` block will be executed.

Note that if statements can also be written in a single line using a ternary conditional operator. For example

```
a = 10
result = "a is greater than 5" if a > 5 else "a is less than or equal to 5"
print(result)
```

```
a is greater than 5
```

The above code assigns a different string to the variable `result` based on the condition `a > 5`. If the condition is true, it assigns "a is greater than 5"; otherwise, it assigns "a is less than or equal to 5".

2.3.4 Loops

Loops allow you to execute a block of code multiple times, which is useful for iterating over collections of data or performing repetitive tasks. In Python, there are two main types of loops: `for` loops and `while` loops.

`while` loops repeatedly execute a block of code as long as a specified condition is true. For example

```
count = 0
while count < 5:
  print("Count is", count)
  count += 1  # Increment count by 1
```

```
Count is 0
Count is 1
Count is 2
Count is 3
Count is 4
```

```
print("Final count is", count)
```

```
Final count is 5
```

In this example, the loop will continue to run as long as `count` is less than `5`. Inside the loop, we print the current value of `count` and then increment it by `1`. Once `count` reaches `5`, the condition becomes false, and the loop exits. Note that `count += 1` is a shorthand for `count = count + 1`.

`for` loops are used to iterate over a sequence (like a list, tuple, or string) or other iterable objects. We will see examples of such objects in the section on data structures. For the moment, let's look at a simple example of a `for` loop that iterates over a list of numbers

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
  print("Number is", num)
```

```
Number is 1
Number is 2
Number is 3
Number is 4
Number is 5
```

or alternatively, we can use the `range()` function to generate a sequence of numbers to iterate over

```
for i in range(5):  # Generates numbers from 0 to 4
  print("i is:", i)
```

```
i is: 0
i is: 1
i is: 2
i is: 3
i is: 4
```

We can use the function `range` also to get a sequence of number to loop over. It follows the syntax `range(start, stop, step)`

- **Start**
  - Where the sequence starts: it includes the start value (first number will always be `start`)
  - Optional
  - Defaults to 0 (unless otherwise specified)
- **Stop**:
  - Where the sequence ends: it does not include the stop value (last number will always be `stop-step`)
  - Required field
- **Step**:
  - Step size of the sequence, i.e., how much we increase the value at each iteration: `start + step`, `start + 2*step`, `start + 3*step`, …
  - Optional
  - Defaults to 1 (unless otherwise specified)

```
for i in range(2, 10, 2):  # Generates even numbers from 2 to 8
  print("i is", i)
```

```
i is 2
i is 4
i is 6
i is 8
```

But as mentioned before, `for` loops can iterate over any iterable object, not just sequences of numbers. For example, we can iterate over the characters in a string

```
for letter in "Cemfi":
    print(letter)
```

```
C
e
m
f
i
```

or over a list of strings

```python
months_of_year = ["January", "February", "March", "April", "May", "June",
 ↪  "July", "August", "September", "October", "November", "December"]

# Loop through the months and add some summer vibes
for month in months_of_year:
    if month == "June":
        print(f"Get ready to enjoy the summer break, it's {month}!")
    elif month == "July" or month =="August":
        print(f"{month} is perfect to find reasons to escape from Madrid")
         ↪
    else:
        print(f"Winter is coming")
```

```
Winter is coming
Winter is coming
Winter is coming
Winter is coming
Winter is coming
Get ready to enjoy the summer break, it's June!
July is perfect to find reasons to escape from Madrid
August is perfect to find reasons to escape from Madrid
Winter is coming
Winter is coming
Winter is coming
Winter is coming
```

Where we combined loops with conditional statements to print different messages based on the current month.

Note that you can use the `break` statement to exit a loop prematurely when a certain condition is met, and the `continue` statement to skip the current iteration and move to the next one.

```python
for i in range(10):
  if i == 5:
    break  # Exit the loop when i is 5
  print("i is", i)
```

```
i is 0
i is 1
i is 2
i is 3
i is 4
```

```
for i in range(10):
  if i % 2 == 0:
    continue  # Skip even numbers
  print("i is", i)
```

```
i is 1
i is 3
i is 5
i is 7
i is 9
```

You can also create nested loops, where one loop is placed inside another loop. This is useful for iterating over multi-dimensional data structures or performing more complex tasks.

```
for i in range(3):
  for j in range(2):
    print(f"i: {i}, j: {j}")
```

```
i: 0, j: 0
i: 0, j: 1
i: 1, j: 0
i: 1, j: 1
i: 2, j: 0
i: 2, j: 1
```

`enumerate()` is a built-in function that adds a counter to an iterable and returns it as an enumerate object. This is particularly useful when you need both the index and the value of items in a loop.

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(f"Index: {index}, Fruit: {fruit}")
```

```
Index: 0, Fruit: apple
Index: 1, Fruit: banana
Index: 2, Fruit: cherry
```

2.3.5 Exercises

Now that we have covered the basics of Python programming, it's time to practice what we've learned. Here are some exercises to help you reinforce your understanding of variables, data types, functions, conditionals, and loops.

1. Create two variables, `a` and `b`, and assign them the values `10` and `20`, respectively. Write a function that takes these two variables as input and returns their product and their difference.
2. Write a function called `is_even` that takes a number as input and returns `True` if the number is even and `False` otherwise. Try calling the function with different numbers to test it.
3. Write a loop that computes the result of the sum $\sum_{i=1}^{10} i^2$ and prints the result.
4. Write a loop to compute the product of all odd numbers between `1` and `20`. Print the final result. *Hint:* You could reuse the `is_even` function you defined earlier.
5. Compute the sum of all numbers between `1` and `1000` that are divisible by `3` or `5`. Print the final result.

2.4 Data Structures

The fundamental data types we have seen so far are useful for storing single values. However, in practice, we often need to work with collections of data. Python provides several built-in **collection types** to handle such cases. The most commonly used data structures in Python are

- **Lists**: Ordered, mutable collections of items
- **Tuples**: Ordered, immutable collections of items
- **Dictionaries**: Ordered (Unordered prior to Python 3.7), mutable collections of key-value pairs
- **Sets**: Unordered collections of unique items
- **Ranges**: Immutable sequences of numbers, often used for iteration

We will explore each of these types in more detail below.

2.4.1 Lists

We have already seen lists in some of the previous examples. A list is an ordered collection of items that can be of different types. Lists are mutable, meaning you can change their contents after creation. You can create a list by enclosing items in square brackets `[]`, separated by commas.

```
my_list = [1, 2.5, "Hello", True]
print(my_list)
```

```
[1, 2.5, 'Hello', True]
```

We can access individual elements in a list using their index, which starts at `0` for the first element. For example

```
first_element = my_list[0]
print("First element:", first_element)
```

```
First element: 1
```

You can also access elements from the end of the list using negative indices, where `-1` refers to the last element, `-2` to the second last, and so on.

```
last_element = my_list[-1]
print("Last element:", last_element)
```

```
Last element: True
```

Multiple elements can be accessed using slicing, which allows you to specify a range of indices. The syntax for slicing is `list[start:stop]`, where `start` is the index of the first element to include, and `stop` is the index of the first element to exclude.

```
sub_list = my_list[1:3]  # Elements at index 1 and 2
print("Sub-list:", sub_list)
```

```
Sub-list: [2.5, 'Hello']
```

Since lists are mutable, you can modify their contents. For example, you can change the value of an element at a specific index.

```
my_list[2] = "World"
print("After modification:", my_list)
```

```
After modification: [1, 2.5, 'World', True]
```

To add elements to a list, we can use the `append()` method to add an item to the end of the list or the `insert()` method to add an item at a specific index, or `extend()` to add multiple items at once.

```python
my_list.append("New Item")
print("After appending:", my_list)
```

```
After appending: [1, 2.5, 'World', True, 'New Item']
```

```python
my_list.insert(1, "Inserted Item")
print("After inserting:", my_list)
```

```
After inserting: [1, 'Inserted Item', 2.5, 'World', True, 'New Item']
```

```python
my_list.extend([3, 4, 5])
print("After extending:", my_list)
```

```
After extending: [1, 'Inserted Item', 2.5, 'World', True, 'New Item', 3, 4,
5]
```

Note how these methods modify the original list in place and return `None`, so you should not write `my_list = my_list.append(...)`.

There are also options to remove items from a list. You can use the `remove()` method to remove the first occurrence of a specific value, the `pop()` method to remove an item at a specific index (or the last item if no index is provided), or the `clear()` method to remove all items from the list.

```python
my_list.remove("World")
print("After removing 'World':", my_list)
```

```
After removing 'World': [1, 'Inserted Item', 2.5, True, 'New Item', 3, 4, 5]
```

```python
popped_item = my_list.pop(2)  # Remove item at index 2
print("After popping index 2:", my_list)
```

```
After popping index 2: [1, 'Inserted Item', True, 'New Item', 3, 4, 5]
```

```python
print("Popped item:", popped_item)
```

```
Popped item: 2.5
```

```python
my_list.clear()
print("After clearing:", my_list)
```

```
After clearing: []
```

There is a convenient way to create lists using **list comprehensions**. List comprehensions provide a concise way to create lists based on existing iterables. The syntax is `[expression for item in iterable if condition]`, where `expression` is the value to be added to the list, `item` is the variable representing each element in the iterable, and `condition` is an optional filter. For example, here is how to create a list of squares of even numbers from `0` to `9`.

```
squares_of_even = [x**2 for x in range(10) if x % 2 == 0]
print("Squares of even numbers:", squares_of_even)
```

```
Squares of even numbers: [0, 4, 16, 36, 64]
```

Let's break down the list comprehension above:

- `x**2`: This is the expression that defines what each element in the new list will be. In this case, it's the square of `x`.
- `for x in range(10)`: This part iterates over the numbers from `0` to `9`.
- `if x % 2 == 0`: This is a condition that filters the numbers, including only even numbers in the new list. It uses the modulus operator `%` to check if `x` is divisible by `2`. If a number is divisible by `2`, the remainder is `0`, indicating that it is even.

2.4.2 Tuples

Tuples are similar to lists in that they are ordered collections of items. However, tuples are immutable, meaning that once they are created, their contents cannot be changed. You can create a tuple by enclosing items in parentheses `()`, separated by commas.

```
my_tuple = (1, 2.5, "Hello", True)
print(my_tuple)
```

```
(1, 2.5, 'Hello', True)
```

You can access elements in a tuple using indexing and slicing, just like with lists.

```
first_element = my_tuple[0]
print("First element:", first_element)
```

```
First element: 1
```

```
second_element = my_tuple[1]
print("Second element:", second_element)
```

```
Second element: 2.5
```

```
last_element = my_tuple[-1]
print("Last element:", last_element)
```

```
Last element: True
```

```
sub_tuple = my_tuple[1:3]  # Elements at index 1 and 2
print("Sub-tuple:", sub_tuple)
```

```
Sub-tuple: (2.5, 'Hello')
```

Note that we have seen tuples before when we defined functions that return multiple values. In such cases, Python automatically packs the returned values into a tuple, which can then be unpacked into separate variables.

```
def get_coordinates():
  x = 10
  y = 20
  return x, y  # Returns a tuple (10, 20)

x_coord, y_coord = get_coordinates()  # Unpacks the tuple into separate
 ↪  variables
print("X coordinate:", x_coord)
```

```
X coordinate: 10
```

```
print("Y coordinate:", y_coord)
```

```
Y coordinate: 20
```

Note that tuples are faster than lists for certain operations due to their immutability, making them a good choice for storing data that should not change. If you need to be able to modify the contents, use a list instead. For example, the following code will raise an error because we are trying to change an element of a tuple

```
#my_tuple[1] = 3.0  # This will raise a TypeError
```

While tuples are immutable, you can concatenate two tuples to create a new tuple

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
combined_tuple = tuple1 + tuple2
print("Combined tuple:", combined_tuple)
```

```
Combined tuple: (1, 2, 3, 4, 5, 6)
```

or you can repeat a tuple multiple times

```
repeated_tuple = tuple1 * 3
print("Repeated tuple:", repeated_tuple)
```

```
Repeated tuple: (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Unpacking can also be used with tuples. For example, you can unpack the elements of a tuple into separate variables

```
my_tuple = (10, 20, 30)
a, b, c = my_tuple
print("a:", a)
```

```
a: 10
```

```
print("b:", b)
```

```
b: 20
```

```
print("c:", c)
```

```
c: 30
```

If you don't want to unpack all elements, you can use the asterisk (*) operator to capture the remaining elements in a list

```
my_tuple = (10, 20, 30, 40, 50)
a, b, *rest = my_tuple
print("a:", a)
```

```
a: 10
```

```
print("b:", b)
```

```
b: 20
```

```
print("rest:", rest)
```

```
rest: [30, 40, 50]
```

It is also common to use _ (underscore) as a variable name for values that you want to ignore during unpacking

```
my_tuple = (10, 20, 30)
a, _, c = my_tuple  # Ignore the second element
print("a:", a)
```

```
a: 10
```

```
print("c:", c)
```

```
c: 30
```

2.4.3 Dictionaries

Dictionaries are ordered (unordered prior to Python 3.7) collections of key-value pairs. Each key is unique and is used to access its corresponding value. Dictionaries are mutable, meaning you can change their contents after creation. The keys in a dictionary must be unique and immutable (e.g., strings, numbers, or tuples), while the values can be of any data type and can be duplicated. You can create a dictionary by enclosing key-value pairs in curly braces `{}`, with each key and value separated by a colon `:` and pairs separated by commas.

```
my_dict = {"name": "Alba", "age": 30, "is_student": False}
print(my_dict)
```

```
{'name': 'Alba', 'age': 30, 'is_student': False}
```

You can access values in a dictionary using their keys. For example

```
name = my_dict["name"]
print("Name:", name)
```

```
Name: Alba
```

You can also add new key-value pairs or update existing ones

```
my_dict["city"] = "Madrid"  # Add a new key-value pair
print("After adding city:", my_dict)
```

```
After adding city: {'name': 'Alba', 'age': 30, 'is_student': False, 'city':
'Madrid'}
```

Alternatively, you can use the `update()` method to add or update multiple key-value pairs at once

```
my_dict.update({"age": 31, "country": "Spain"})
print("After updating age and adding country:", my_dict)
```

```
After updating age and adding country: {'name': 'Alba', 'age': 31,
'is_student': False, 'city': 'Madrid', 'country': 'Spain'}
```

Note that if you use a key that already exists in the dictionary, the corresponding value will be updated. This applies whether you use the assignment syntax or the `update()` method.

The keys and values can be accessed using the `keys()` and `values()` methods, respectively. You can also use the `items()` method to get key-value pairs as tuples.

```
keys = my_dict.keys()
print("Keys:", keys)
```

```
Keys: dict_keys(['name', 'age', 'is_student', 'city', 'country'])
```

```
values = my_dict.values()
print("Values:", values)
```

```
Values: dict_values(['Alba', 31, False, 'Madrid', 'Spain'])
```

```
items = my_dict.items()
print("Items:", items)
```

```
Items: dict_items([('name', 'Alba'), ('age', 31), ('is_student', False),
('city', 'Madrid'), ('country', 'Spain')])
```

The latter is particularly useful for iterating over both keys and values in a loop.

We can remove key-value pairs from a dictionary using the `del` statement or the `pop()` method.

```
del my_dict["is_student"]
print("After deleting is_student:", my_dict)
```

```
After deleting is_student: {'name': 'Alba', 'age': 31, 'city': 'Madrid',
'country': 'Spain'}
```

```
age = my_dict.pop("age")
print("After popping age:", my_dict)
```

```
After popping age: {'name': 'Alba', 'city': 'Madrid', 'country': 'Spain'}
```

```
print("Popped age:", age)
```

```
Popped age: 31
```

2.4.4 Sets

Sets are unordered collections of unique items. They are mutable, meaning you can change their contents after creation. Sets are useful for storing items when the order does not matter and duplicates are not allowed. You can create a set by enclosing items in curly braces `{}`, separated by commas.

```
my_set = {1, 2, 3, 4, 5}
print("Set:", my_set)
```

```
Set: {1, 2, 3, 4, 5}
```

You can also create a set from an iterable, such as a list, using the `set()` constructor.

```
my_list = [1, 2, 2, 3, 4, 4, 5]
my_set_from_list = set(my_list)
print("Set from list:", my_set_from_list)
```

```
Set from list: {1, 2, 3, 4, 5}
```

You can add items to a set using the `add()` method and remove items using the `remove()` or `discard()` methods.

```python
my_set.add(6)
print("After adding 6:", my_set)
```

```
After adding 6: {1, 2, 3, 4, 5, 6}
```

```python
my_set.remove(3)
print("After removing 3:", my_set)
```

```
After removing 3: {1, 2, 4, 5, 6}
```

```python
my_set.discard(10)  # Does not raise an error if 10 is not in the set
print("After discarding 10:", my_set)
```

```
After discarding 10: {1, 2, 4, 5, 6}
```

There is also a `frozenset` type, which is an immutable version of a set. Once created, the contents of a frozenset cannot be changed. You can create a frozenset using the `frozenset()` constructor.

```python
my_frozenset = frozenset([1, 2, 3, 4, 5])
print("Frozenset:", my_frozenset)
```

```
Frozenset: frozenset({1, 2, 3, 4, 5})
```

Sets are particularly useful for performing mathematical set operations such as union, intersection, difference, and symmetric difference. For example

```python
set_a = {1, 2, 3, 4}
set_b = {3, 4, 5, 6}
union_set = set_a.union(set_b)
print("Union:", union_set)
```

```
Union: {1, 2, 3, 4, 5, 6}
```

```python
intersection_set = set_a.intersection(set_b)
print("Intersection:", intersection_set)
```

```
Intersection: {3, 4}
```

```
difference_set = set_a.difference(set_b)
print("Difference (A - B):", difference_set)
```

```
Difference (A - B): {1, 2}
```

```
symmetric_difference_set = set_a.symmetric_difference(set_b)
print("Symmetric Difference:", symmetric_difference_set)
```

```
Symmetric Difference: {1, 2, 5, 6}
```

More compactly, you can use operators for these operations

```
union_set = set_a | set_b
intersection_set = set_a & set_b
difference_set = set_a - set_b
symmetric_difference_set = set_a ^ set_b
```

2.4.5 Ranges

Ranges are immutable sequences of numbers, commonly used for iteration in loops. You can create a range using the `range()` function, which generates a sequence of numbers based on the specified start, stop, and step values. The syntax is `range(start, stop, step)`, where `start` is the first number in the sequence (inclusive), `stop` is the end of the sequence (exclusive), and `step` is the increment between each number.

```
my_range = range(0, 10, 2)  # Generates numbers from 0 to 8 with a step of 2
print("Range:", list(my_range)) # Convert to list for display
```

```
Range: [0, 2, 4, 6, 8]
```

You can also create a range with just the `stop` value, in which case the sequence starts from `0` and increments by `1` by default.

```
my_range_default = range(5)  # Generates numbers from 0 to 4
print("Range with default start and step:", list(my_range_default))
```

```
Range with default start and step: [0, 1, 2, 3, 4]
```

You have seen earlier how to use ranges in `for` loops to iterate over a sequence of numbers. Ranges are memory efficient because they generate numbers on-the-fly and do not store the entire sequence in memory, making them suitable for large sequences.

2.4.6 Mutable vs. Immutable Objects

In the examples up to now you have already seen that data types can be classified as either **mutable** or **immutable** based on whether their values can be changed after they are created.

- **Mutable objects**: These objects can be modified after they are created. Examples of mutable data types in Python include lists, dictionaries, and sets. When you modify a mutable object, you are changing the object itself, and any other references to that object will reflect the changes.
- **Immutable objects**: These objects cannot be modified after they are created. Examples of immutable data types in Python include integers, floats, strings, and tuples. When you attempt to modify an immutable object, you are actually creating a new object with the modified value, leaving the original object unchanged.

An important implication of mutability is what happens when you assign one variable to another. For mutable objects, both variables will reference the same object in memory, so changes made through one variable will affect the other. For immutable objects, each variable will reference its own separate object.

```
# Mutable example with lists
list1 = [1, 2, 3]
list2 = list1  # Both variables reference the same list
list2.append(4)  # Modify list2
print("list1:", list1)  # list1 is also affected
```

```
list1: [1, 2, 3, 4]
```

```
print("list2:", list2)
```

```
list2: [1, 2, 3, 4]
```

```
# Immutable example with strings
str1 = "Hello"
str2 = str1  # Both variables reference the same string
str2 += ", World!"  # Modify str2 (creates a new string)
print("str1:", str1)  # str1 remains unchanged
```

```
str1: Hello
```

```
print("str2:", str2)
```

```
str2: Hello, World!
```

The concept of mutability is important to understand when working with data structures and functions in Python, as it can affect how data is passed and modified within your code. When passing mutable objects to functions, changes made to the object within the function will affect the original object outside the function.

```python
def modify_list(input_list):
    input_list.append(100)  # Modifies the original list

my_list = [1, 2, 3]
modify_list(my_list)
print(my_list)  # my_list is changed
```

```
[1, 2, 3, 100]
```

In contrast, passing immutable objects to functions will not affect the original object.

```python
def modify_int(input_int):
    input_int += 10  # Creates a new integer

my_int = 5
modify_int(my_int)
print(my_int)  # my_int remains unchanged
```

```
5
```

Therefore, it is crucial to be aware of the mutability of the data types you are working with to avoid unintended side effects in your code.

2.4.7 Exercises

Now that we have covered the basics of data structures in Python, it's time to practice what we've learned. Here are some exercises to help you reinforce your understanding of lists, tuples, dictionaries, sets, and ranges.

1. Create a list of the first 10 square numbers (i.e., 1, 4, 9, ..., 100) using a list comprehension. Print the resulting list.
2. Create a tuple containing the names of the days of the week. Access and print the name of the third day.
3. Create a dictionary that maps the names of three countries to their respective capitals. Access and print the capital of one of the countries.

4. Create a set containing the unique vowels in the word “programming”. Print the resulting set.
5. Create a range of numbers from 1 to 20 with a step of 3. Use a for loop to iterate over the range and print each number.

2.5 Object-Oriented Programming (OOP) Basics

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around “objects” - which combine data (attributes) and functions (methods) that operate on that data. Think of objects as self-contained units that represent real-world entities or concepts. In Python, everything is an object, including basic data types like integers and strings. Therefore, we have been using OOP concepts all along without being explicit about it.

2.5.1 Classes and Objects

A **class** is like a blueprint or template for creating objects. An **object** is a specific instance created from that class. For example, if “Car” is a class, then “my_toyota” and “your_honda” would be objects (instances) of that class.

Here’s a simple example of defining a class and creating objects from it:

```
# Define a class
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited ${amount}. New balance: ${self.balance}")

    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds!")
        else:
            self.balance -= amount
            print(f"Withdrew ${amount}. New balance: ${self.balance}")

# Create objects (instances)
account1 = BankAccount("Alba", 1000)
account2 = BankAccount("Jesus", 500)
```

```
# Use methods
account1.deposit(200)
```

```
Deposited $200. New balance: $1200
```

```
account1.withdraw(300)
```

```
Withdrew $300. New balance: $900
```

```
# Check balances (accessing attributes)
print(f"{account1.owner}'s balance: ${account1.balance}")
```

```
Alba's balance: $900
```

```
print(f"{account2.owner}'s balance: ${account2.balance}")
```

```
Jesus's balance: $500
```

The `__init__` method is a special method called a **constructor** that runs automatically when you create a new object. The `self` parameter refers to the instance itself and is used to access its attributes and methods.

2.5.2 Attributes and Methods

Attributes are variables that belong to an object and store its data. **Methods** are functions that belong to an object and define its behavior.

```
class Student:
    def __init__(self, name, student_id):
        self.name = name                # attribute
        self.student_id = student_id  # attribute
        self.courses = []             # attribute

    def enroll(self, course):         # method
        self.courses.append(course)
        print(f"{self.name} enrolled in {course}")

    def get_courses(self):            # method
        return self.courses
```

```
# Create and use a student object
student = Student("Alba", "S12345")
student.enroll("Artificial Intelligence and Big Data")
```

```
Alba enrolled in Artificial Intelligence and Big Data
```

```
student.enroll("Python Programming")
```

```
Alba enrolled in Python Programming
```

```
print(f"{student.name}'s courses: {student.get_courses()}")
```

```
Alba's courses: ['Artificial Intelligence and Big Data', 'Python
Programming']
```

2.5.3 Inheritance

Inheritance is a fundamental OOP concept where a new class (called a **child** or **subclass**) can be based on an existing class (called a **parent** or **superclass**). The child class inherits all the attributes and methods of the parent class and can add new ones or modify existing behavior.

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound")

    def sleep(self):
        print(f"{self.name} is sleeping... Zzz")

# Child class inherits from Animal
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Call the parent's __init__
        self.breed = breed      # Add a new attribute

    def speak(self):            # Override the parent's method
        print(f"{self.name} barks!")
    def fetch(self):            # Add a new method
        print(f"{self.name} fetches the ball")

# Create objects
generic_animal = Animal("Generic")
my_dog = Dog("Buddy", "Labrador")

# Method inheritance: Dog inherits sleep() from Animal without modification
my_dog.sleep()
```

```
Buddy is sleeping... Zzz
```

```
# Method overriding: Dog has its own version of speak()
generic_animal.speak()
```

```
Generic makes a sound
```

```
my_dog.speak()
```

```
Buddy barks!
```

```
# New method: fetch() is only available in Dog
my_dog.fetch()
```

```
Buddy fetches the ball
```

```
print(f"{my_dog.name} is a {my_dog.breed}")
```

```
Buddy is a Labrador
```

This example demonstrates three key aspects of inheritance:

- **Method inheritance**: The `Dog` class automatically gets the `sleep()` method from `Animal` without any additional code. When we call `my_dog.sleep()`, it uses the parent's implementation.
- **Method overriding**: The `Dog` class defines its own `speak()` method, which replaces the parent's version. When we call `my_dog.speak()`, it prints "barks!" instead of "makes a sound".
- **Method extension**: The `Dog` class adds a new `fetch()` method that doesn't exist in `Animal`.

The `super()` function is used to call methods from the parent class. In the example above, `super().__init__(name)` calls the `Animal` class's constructor to initialize the `name` attribute before adding the `breed` attribute specific to dogs.

While we won't create complex inheritance hierarchies in this course, understanding this concept helps when working with libraries like scikit-learn. For example, when you use a model like `LinearRegression`, it inherits from base classes that provide common methods like `fit()`, `predict()`, and `score()`. This is why all scikit-learn models share a consistent interface—they all inherit from the same base classes.

```
# Preview: scikit-learn models use inheritance
# All estimators inherit common methods from base classes
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor

# Both models have the same interface because they inherit from the same base
 ↪  class
lr = LinearRegression()
dt = DecisionTreeRegressor()

# Both have fit(), predict(), score() methods inherited from base classes
print("LinearRegression methods:", [m for m in dir(lr) if not
 ↪  m.startswith('_')][:5])
```

```
LinearRegression methods: ['copy_X', 'fit', 'fit_intercept',
'get_metadata_routing', 'get_params']
```

```
print("DecisionTreeRegressor methods:", [m for m in dir(dt) if not
 ↪  m.startswith('_')][:5])
```

```
DecisionTreeRegressor methods: ['apply', 'ccp_alpha', 'class_weight',
'cost_complexity_pruning_path', 'criterion']
```

2.5.4 Why Use OOP?

OOP helps organize complex programs by grouping related data and functionality together. This makes code:

- **More intuitive**: Objects model real-world entities
- **Easier to maintain**: Changes to one class don't affect unrelated code
- **Reusable**: Classes can be used in multiple parts of your program

In data science, you'll often work with objects like DataFrames (from pandas), models (from scikit-learn), or plots (from matplotlib), even if you don't create your own classes frequently.

```
# Example: You're already using OOP when working with lists!
my_list = [1, 2, 3]         # my_list is an object of class 'list'
my_list.append(4)           # append is a method
my_list.sort()              # sort is a method
print(len(my_list))         # len works with the object's internal data
```

```
4
```

For **this course**, understanding how to use objects and their methods is more important than creating complex class hierarchies. Most of the time, you'll be **using classes created by others** (like pandas DataFrames or scikit-learn models) **rather than writing your own**.

2.5.5 Exercises

Now that we have covered the basics of object-oriented programming in Python, here are some exercises to help reinforce your understanding of classes, objects, attributes, methods, and inheritance.

1. Create a `Rectangle` class with `width` and `height` attributes. Add methods `area()` that returns the area and `perimeter()` that returns the perimeter. Create a rectangle object and test both methods.
2. Create a `Counter` class with a `count` attribute that starts at 0. Add methods `increment()` to increase the count by 1, `decrement()` to decrease it by 1, and `reset()` to set it back to 0. Test your class by creating a counter and calling its methods.
3. Create a `Vehicle` parent class with attributes `brand` and `year`, and a method `info()` that prints vehicle information. Then create a `Car` child class that adds a `num_doors` attribute and overrides the `info()` method to also display the number of doors.

2.6 Essential Packages

In this section, we will introduce some of the most essential packages in Python for data science and scientific computing. These packages provide powerful tools and functionalities that make it easier to work with data, perform numerical computations, and create visualizations.

ℹ Modules vs. Packages

A module, in Python, is a program that can be imported into interactive mode or other programs for use. A Python package typically comprises multiple modules. Physically, a package is a directory containing modules and possibly subdirectories, each potentially containing further modules. Conceptually, a package links all modules together using the package name for reference.

2.6.1 Scientific Computing: NumPy

NumPy (**Num**erical **Py**thon) is one of the most common packages used in Python. In fact, numerous computational packages that offer scientific capabilities utilize NumPy's array objects as a standard interface for data exchange. That's why understanding NumPy arrays and array-based computing principles is crucial.

NumPy offers a vast array of efficient methods for creating and manipulating numerical data arrays. Unlike Python lists, which can accommodate various data types within a single list, NumPy arrays require homogeneity among their elements for efficient mathematical operations. Utilizing NumPy arrays provides advantages such as faster execution and reduced memory consumption compared to Python lists. With NumPy, data storage is optimized through the specification of data types, enhancing code optimization.

ℹ Note

Documentation for this package is available at https://numpy.org/doc/stable/.

To use NumPy in your code, you typically import it with the alias `np`

```
import numpy as np
```

2.6.1.1 Creating NumPy Arrays

Arrays serve as a fundamental data structure within the NumPy. They represent a grid of values containing information on raw data, element location, and interpretation. Elements share a common data type, known as the array dtype.

One method of initializing NumPy arrays involves using Python lists, with nested lists employed for two- or higher-dimensional data structures.

```
a = np.array([1, 2, 3, 4, 5, 6])
print("1D array:", a)
```

```
1D array: [1 2 3 4 5 6]
```

We can access the elements through indexing.

```
a[0]
```

```
np.int64(1)
```

Arrays are N-Dimensional (that's why sometimes we refer to them as ndarray). That means that NumPy arrays will encompass vector (1-Dimensions), Matrices (2D) or tensors (3D and higher). We can get all the information of the array by checking its attributes. To create a 2D array, we can use nested lists:

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
```

Mathematically, we can think of this as a matrix with 2 rows and 4 columns, i.e.,

$$a = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

We can check its attributes to get more information about the array:

```
print('Dimensions/axes:', a.ndim)
```

```
Dimensions/axes: 2
```

```
print('Shape (size of array in each dimension):', a.shape)
```

```
Shape (size of array in each dimension): (2, 4)
```

```
print('Size (total number of elements):', a.size)
```

```
Size (total number of elements): 8
```

```
print('Number of bytes:', a.nbytes)
```

```
Number of bytes: 64
```

```
print('Data type:', a.dtype)
```

```
Data type: int64
```

```
print('Item size (in bytes):', a.itemsize)
```

```
Item size (in bytes): 8
```

We have already seen how to access elements in a 1D array. For 2D arrays, we can use two indices: the first for the row and the second for the column.

```
element = a[0, 2]  # Access the element in the first row and third column
print("Element at (0, 2):", element)
```

```
Element at (0, 2): 3
```

We can also use slicing to access subarrays. For example, to get the first two rows and the first three columns:

```
subarray = a[0:2, 0:3]
print("Subarray:\n", subarray)
```

```
Subarray:
 [[1 2 3]
 [5 6 7]]
```

We don't need to specify both indices all the time. For example, to get the first row, we can do

```
first_row = a[0, :]
print("First row:", first_row)
```

```
First row: [1 2 3 4]
```

or to get the second column

```
second_column = a[:, 1]
print("Second column:", second_column)
```

```
Second column: [2 6]
```

We can initialize arrays using different commands depending on our aim. For instance, the most straightforward case would be to pass a list to `np.array()` to create one:

```
arr1 = np.array([5,6,7])
arr1
```

```
array([5, 6, 7])
```

However, sometimes we are more ambiguous and have no information on what our array contains. We just need to be able to initialize an array so that later on, our code, can update it. For this, we typically create arrays of the desired dimensions and fill them with zeros (`np.zeros()`), ones (`np.ones()`), with a given value (`np.full()`) or without initializing (`np.empty()`).

> Tip
>
> When working with large data, `np.empty()` can be faster and more efficient. Also, large arrays can take up most of your memory and, in those cases, carefully establishing the `dtype()` can help to manage memory more efficiently (i.e., choose 8 bits over 64 bits.)

```python
np.zeros(4)
```

```
array([0., 0., 0., 0.])
```

```python
np.ones((2,3))
```

```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

To create higher-dimensional arrays, we can pass a tuple representing the shape of the array:

```python
np.ones((3,2,1))
```

```
array([[[1.],
        [1.]],

       [[1.],
        [1.]],

       [[1.],
        [1.]]])
```

This created a 3D array with 3 layers of matrices with 2 rows and 1 column.

We can use `np.full()` to create an array of constant values that we specify in the `fill_value` option.

```python
np.full((2,2) , fill_value= 4)
```

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

`np.empty()` creates an array without initializing its values. The values in the array will be whatever is already present in the allocated memory, which can be random and unpredictable.

```
np.empty(2)
```

```
array([0., 1.])
```

With `np.linspace()`, we can create arrays with evenly spaced values over a specified range. The syntax is `np.linspace(start, stop, num)`, where `start` is the starting value, `stop` is the ending value, and `num` is the number of evenly spaced values to generate.

```
np.linspace(0, 1, 5)  # Generates 5 evenly spaced values between 0 and 1
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

`np.arange()` is another useful function to create arrays with evenly spaced values, similar to the built-in `range()` function but returning a NumPy array. The syntax is `np.arange(start, stop, step)`, where `start` is the starting value, `stop` is the ending value (exclusive), and `step` is the increment between each value.

```
np.arange(0, 10, 2)  # Generates values from 0 to 8 with a step of 2
```

```
array([0, 2, 4, 6, 8])
```

Note that both `np.linspace()` and `np.arange()` can be used to create sequences of numbers, but they differ in how you specify the spacing and the number of elements. In general, use `np.linspace()` when you want a specific number of evenly spaced values over a range, and use `np.arange()` when you want to specify the step size between values.

Sometimes, you might also need to create identity matrices, which are square matrices with ones on the diagonal and zeros elsewhere. You can use `np.eye()` to create an identity matrix of a specified size.

```
np.eye(3)  # Creates a 3x3 identity matrix
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Or you might want to create diagonal matrices with specific values on the diagonal. You can use `np.diag()` for this purpose.

```python
np.diag([1, 2, 3])  # Creates a diagonal matrix with 1, 2, 3 on the diagonal
```

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

Finally, to create random arrays, NumPy provides several functions in the `np.random` module. For example, you can create an array of random floats between 0 and 1 using `np.random.rand()`, or an array of random integers within a specified range using `np.random.randint()`, or a normal distribution using `np.random.randn()`.

```python
np.random.rand(2, 3)  # Creates a 2x3 array of random floats between 0 and 1
```

```
array([[0.13998123, 0.62860598, 0.18765128],
       [0.43531749, 0.53105854, 0.10783084]])
```

```python
np.random.randint(0, 10, size=(2, 3))  # Creates a 2x3 array of random integers between 0 and 9
```

```
array([[2, 9, 2],
       [6, 0, 6]])
```

```python
np.random.randn(2, 3)  # Creates a 2x3 array of random floats from a standard normal distribution
```

```
array([[-0.537313  , -0.49899483,  0.75224564],
       [ 1.48123443,  0.225615  ,  0.40000993]])
```

Random Seed

When generating random numbers, it's often useful to set a random seed using `np.random.seed()`. This ensures that the sequence of random numbers generated is reproducible, meaning that you will get the same random numbers each time you run your code with the same seed. This is particularly important for debugging and sharing results.

2.6.1.2 Managing Array Data

Arrays accept common operations like sorting, concatenating and finding unique elements.

For instance, using the `sort()` method we can sort elements within an array.

```
arr1 = np.array((10,2,5,3,50,0))
np.sort(arr1)
```

```
array([ 0,  2,  3,  5, 10, 50])
```

In multidimensional arrays, we can sort the elements of a given dimension by specifying the axis along which to sort. When axis=0, the operation collapses along the first dimension (rows in a 2D array), giving one result per column. When axis=1, it collapses along the second dimension (columns in a 2D array), giving one result per row.

```
mat1 = np.array([[1,2,3],[8,1,5]])
mat1
```

```
array([[1, 2, 3],
       [8, 1, 5]])
```

```
mat1.sort(axis=1)  # Sort along columns
mat1
```

```
array([[1, 2, 3],
       [1, 5, 8]])
```

Using `concatenate` we can join the elements of two arrays along an existing axis.

```
arr1 = np.array((1,2,3))
arr2 = np.array((6,7,8))
np.concatenate((arr1,arr2))
```

```
array([1, 2, 3, 6, 7, 8])
```

Instead, if we want to concatenate along a new axis, we use `vstack()` and `hstack()`

```
np.vstack((arr1,arr2))  # Vertical stack
```

```
array([[1, 2, 3],
       [6, 7, 8]])
```

```
np.hstack((arr1,arr2))  # Horizontal stack
```

```
array([1, 2, 3, 6, 7, 8])
```

It is also possible to reshape arrays. For instance, let's reshape the concatenation of `arr1` and `arr2` to 3 rows and 2 columns

```
arr_c = np.concatenate((arr1,arr2))
arr_c.reshape((3,2))
```

```
array([[1, 2],
       [3, 6],
       [7, 8]])
```

We can also perform aggregation functions over all elements, like finding the minimum, maximum, means, sum of elements and much more.

```
print(arr1.min())
```

```
1
```

```
print(arr1.sum())
```

```
6
```

```
print(arr1.max())
```

```
3
```

```
print(arr1.mean())
```

```
2.0
```

This can also be done over a specific axis in multidimensional arrays. For example, let's create a 2D array and find the sum across rows and columns

```
mat2 = np.array([[1,2,3],[4,5,6]])
print(mat2.sum(axis=0))  # Sum along rows
```

```
[5 7 9]
```

```
print(mat2.sum(axis=1))  # Sum along columns
```

```
[ 6 15]
```

It is also possible to get only the unique elements of an array or to count how many elements are repeated.

```
arr1 = np.array((1,2,3,3,1,1,5,6,7,8,11,11))
print(np.unique(arr1))
```

```
[ 1  2  3  5  6  7  8 11]
```

```
unq, count = np.unique(arr1, return_counts=True)
print("Unique elements:", unq)
```

```
Unique elements: [ 1  2  3  5  6  7  8 11]
```

```
print("Counts:", count)
```

```
Counts: [3 1 2 1 1 1 1 2]
```

Using `where()`, we can find the indices of elements that satisfy a given condition.

```
arr1 = np.array((10,15,20,25,30,35,40))
indices = np.where(arr1 > 25)
print("Indices of elements greater than 25:", indices)
```

```
Indices of elements greater than 25: (array([4, 5, 6]),)
```

We can also use boolean indexing to filter elements based on a condition.

```
filtered_elements = arr1[arr1 > 25]
print("Elements greater than 25:", filtered_elements)
```

```
Elements greater than 25: [30 35 40]
```

And we can replace elements that meet a condition using `np.where()`

```
new_arr = np.where(arr1 > 25, -1, arr1)  # Replace elements greater than 25
 ↪  with -1
print("Array after replacement:", new_arr)
```

```
Array after replacement: [10 15 20 25 -1 -1 -1]
```

2.6.1.3 Array Operations

NumPy arrays support common operations as addition, subtraction and multiplication. These operations are performed element-wise, meaning that they are applied to each corresponding element in the arrays.

```
A = np.array(((1,2,3),
              (4,5,6)))
B = np.array(((10,20,30),
              (40,50,60)))
```

Element-wise addition, subtraction and multiplication can be performed with `+`, `-` and `*`.

```
A + B
```

```
array([[11, 22, 33],
       [44, 55, 66]])
```

```
B - A
```

```
array([[ 9, 18, 27],
       [36, 45, 54]])
```

```
A * B
```

```
array([[ 10,  40,  90],
       [160, 250, 360]])
```

To multiply (`*`) or divide (`/`) all elements by an scalar, we just specify the scalar.

```
A * 10
```

```
array([[10, 20, 30],
       [40, 50, 60]])
```

```
B / 10
```

```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

Note that NumPy automatically **broadcasts** the scalar to all elements of the array.

Broadcasting

Broadcasting is a powerful mechanism in NumPy that allows operations to be performed on arrays of different shapes. When performing operations between arrays of different shapes, NumPy automatically expands the smaller array along the dimensions of the larger array so that they have compatible shapes. This process is called broadcasting. For example, consider adding a 1D array to a 2D array. NumPy will “broadcast” the 1D array across the rows of the 2D array to perform the addition.

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])
B = np.array([10, 20, 30])  # 1D array
C = A + B  # B is broadcasted across the rows of A
print(C)
```

```
[[11 22 33]
 [14 25 36]]
```

Comparing NumPy arrays is also possible using operators as `==`, `!=`, and the like. Comparisons will result in an array of booleans indicating if the condition is met for a given element.

```
arr1 = np.array(((1,2,3),(4,5,6)))
arr2 = np.array(((1,5,3),(7,2,6)))
arr1==arr2
```

```
array([[ True, False,  True],
       [False, False,  True]])
```

Recall that we use double equals `==` for comparison, while a single equals `=` is used for assignment.

Note that element-wise multiplication is different from matrix multiplication. Matrix multiplication is achieved with either `@` or `matmul()`.

```
np.matmul(arr1,arr2.T) # Note the transpose of arr2 to match dimensions
```

```
array([[20, 29],
       [47, 74]])
```

```
arr1 @ arr2.T  # Note the transpose of arr2 to match dimensions
```

```
array([[20, 29],
       [47, 74]])
```

2.6.1.4 Exercises

1. Create a 1D array with all integer elements from 1 to 10 (both included). No hard-coding allowed!
2. From the array you created in 1, create one that contains all odd elements and one with all even elements.
3. Create a new array that replaces all elements in 1 that are odd by -1.
4. Create a 3-by-3 matrix filled with 'True' values (i.e., booleans).
5. Suppose you have array `a=np.array(['a','b','c','d','e','f','g'])` and `b = np.array(['g','h','c','a','e','w','g'])`. Find all elements that are equal. Can you get the position where the elements of both arrays match?
6. Write a function that takes a element an array and prints elements that are divisible by a given number. Try it creating an array from 1 to 20 and printing divisibles by 3.

7. Consider two matrices, A and B, both of size 100x100, filled with random integer values between 1 and 10. Implement a function to perform element-wise multiplication of these matrices using nested loops. Implement the same operation using Numpy's vectorized multiplication. Repeat again with matrices of size 1000x1000, 10000x10000 and compare the execution time. Which one is faster?

2.6.2 Data Management: Pandas

Pandas is an open source library providing high-performance, easy-to-use data structures and data analysis tools. Pandas is particularly suited to the analysis of *tabular* data, i.e. data that can go into a table. In other words, if you can imagine the data in an Excel spreadsheet, then Pandas is the tool for the job.

- A fast and efficient DataFrame object for data manipulation with indexing
- Tools for reading and writing data: CSV, Excel, SQL
- Intelligent data alignment and integrated handling of missing data
- Flexible reshaping and pivoting of data sets
- Intelligent label-based slicing, indexing, and subsetting of large data sets
- High performance aggregating, merging, joining or transforming data
- Hierarchical indexing provides an intuitive way of working with high-dimensional data
- Time series-functionality: date-based indexing, frequency conversion, moving windows, date shifting and lagging

> ℹ Note
>
> Documentation for this package is available at https://pandas.pydata.org/docs/.

To use Pandas, you typically import it with the alias `pd`

```
import pandas as pd
```

We will also import NumPy as it is often used alongside Pandas for numerical operations.

```
import numpy as np
```

Pandas builds on two main data structures: `Series` and `DataFrames`. `Series` represent 1D arrays while `DataFrames` are 2D labeled arrays. The easiest way to think about both structures is to conceptualize `DataFrames` as containers of lower dimension data. That is, `DataFrames` columns are composed of `Series`, and each of the elements of a `Series` (i.e., the rows of the `DataFrame`) are individual scalar (numbers or strings) values. In plain words, `Series` are columns made of scalar elements and `DataFrames` are collections of `Series` that get an assigned label. All pandas data structures are value-mutable (i.e., we can change the values of elements and replace `DataFrames`) but some are not always size-mutable. The length of a Series cannot be changed, but, for example, columns can be inserted into a DataFrame.

2.6.2.1 Pandas Series

A `Series` is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index. A `Series` can be created from a list, dictionary, or scalar value using the `pd.Series()` constructor. To create a `Series` from a list, you can do the following:

```
data = [10, 20, 30, 40, 50]
series = pd.Series(data)
```

If you want to specify custom index labels, you can pass a list of labels to the `index` parameter:

```
data = [10, 20, 30, 40, 50]
labels = ['a', 'b', 'c', 'd', 'e']
series = pd.Series(data, index=labels)
```

You can additionally assign a name to the `Series` using the `name` parameter:

```
data = [10, 20, 30, 40, 50]
labels = ['a', 'b', 'c', 'd', 'e']
series = pd.Series(data, index=labels, name='My Series')
```

These functions work the same way when creating a `Series` from a NumPy array. When creating a `Series` from a dictionary, the keys of the dictionary become the index labels, and the values become the data:

```
data = {'a': 10, 'b': 20, 'c': 30}
series = pd.Series(data)
```

You can access elements in a `Series` using their index labels or integer positions. For example, to access the element with label 'b':

```
value = series['b']
print("Value at index 'b':", value)
```

```
Value at index 'b': 20
```

If you want to access elements by their integer position, you can use the `iloc` attribute:

```
value = series.iloc[1]  # Access the second element (index 1)
print("Value at position 1:", value)
```

```
Value at position 1: 20
```

Note that both label-based and positional indexing can be used interchangeably in many cases.

`.loc` is used for label-based indexing, which means you access elements by their index labels:

Syntax	Description	Example	Result
`series.loc[label]`	Single label access	`s.loc['b']`	Value at index 'b'
`series.loc[label_list]`	Multiple labels	`s.loc[['a', 'c']]`	Series with values at 'a' and 'c'
`series.loc[start:end]`	Slice by labels (inclusive)	`s.loc['a':'c']`	Series from 'a' to 'c' (inclusive)
`series.loc[condition]`	Boolean indexing	`s.loc[s > 5]`	Values where condition is True

`.iloc` is used for positional indexing, which means you access elements by their integer position in the Series:

Syntax	Description	Example	Result
`series.iloc[position]`	Single position access	`s.iloc[1]`	Value at position 1
`series.iloc[position_list]`	Multiple positions	`s.iloc[[0, 2]]`	Series with values at positions 0 and 2
`series.iloc[start:end]`	Slice by positions (exclusive end)	`s.iloc[1:3]`	Series from position 1 to 2
`series.iloc[negative_pos]`	Negative indexing	`s.iloc[-1]`	Value at last position

Key Differences:

1. Indexing method:
   - `.loc` uses the actual index labels (strings, dates, etc.)
   - `.iloc` uses integer positions (0, 1, 2, …)
2. Slicing behavior:
   - `.loc` slicing is inclusive of both endpoints
   - `.iloc` slicing is exclusive of the end position

You can retrieve all index labels and values of a `Series` using the `index` and `values` attributes, respectively:

```
index_labels = series.index
print("Index labels:", index_labels)
```

```
Index labels: Index(['a', 'b', 'c'], dtype='object')
```

```
values = series.values
print("Values:", values)
```

```
Values: [10 20 30]
```

You can perform various operations on `Series`, such as arithmetic operations, aggregation functions, and filtering. For example, to add a scalar value to all elements in the `Series`:

```
new_series = series + 5
print("Series after adding 5:\n", new_series)
```

```
Series after adding 5:
 a    15
b    25
c    35
dtype: int64
```

You can also filter the `Series` based on a condition:

```
filtered_series = series[series > 20]
print("Filtered Series (values > 20):\n", filtered_series)
```

```
Filtered Series (values > 20):
 c    30
dtype: int64
```

They work and behave similarly to NumPy arrays in many ways but with additional functionality for handling missing data and labeled data.

2.6.2.2 Pandas DataFrames

Pandas Series are great for one-dimensional data, but in data science, we often work with two-dimensional data tables. This is where Pandas DataFrames come into play. A DataFrame is a two-dimensional labeled data structure with columns of potentially different types. You can think of it as a spreadsheet or SQL table, or a dictionary of Series objects.

2.6.2.2.1 Creating DataFrames

You can create a DataFrame from various data sources, such as dictionaries, lists of lists, or NumPy arrays. Here's an example of creating a DataFrame from a dictionary:

```
data = {
    'Name': ['Alba', 'Jesus', 'Yang'],
    'Age': [30, 25, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)
df = df.set_index('Name')  # Set 'Name' as the index
print("DataFrame:\n", df)
```

```
DataFrame:
        Age          City
Name
```

```
Alba     30     New York
Jesus    25  Los Angeles
Yang     35      Chicago
```

You can also create a DataFrame from a list of lists:

```
# Creating a DataFrame from a list of lists
pd.DataFrame(
    data=[
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ],
    index=["R1", "R2", "R3"],
    columns=["C1", "C2", "C3"]
)
```

```
    C1  C2  C3
R1   1   2   3
R2   4   5   6
R3   7   8   9
```

There are several more ways to create DataFrames, including from CSV files, Excel files, SQL databases, and more. Most of the time, you'll be loading data from external sources rather than creating DataFrames from scratch.

Indexing works similarly to Series, but now you have both row and column labels to consider. Here are some common ways to index and select data in a DataFrame:

Method	Description
`df[column_label]` or `df.column_label` or `df.loc[:, column_label]`	Access a single column by label (returns a Series)
`df[[col1, col2]]`	Access multiple columns by label (returns a DataFrame)
`df.loc[row_labels, column_labels]`	Access rows and columns by **label** (names)
`df.iloc[row_positions, column_positions]`	Access rows and columns by **position** (integers)
`df[boolean_condition]`	Filter rows based on a boolean condition

Consider the following DataFrame

```
df = pd.DataFrame(
    data={
        "area":             ["USA", "Eurozone", "Japan", "UK", "Canada",
         ↪  "Australia"],
        "year":             [2024, 2024, 2024, 2024, 2024, 2024],
        "gdp_growth":       [2.1, 1.3, 0.7, 1.5, 1.8, 2.0],       # in percent
        "inflation":        [3.2, 2.5, 1.0, 2.8, 2.2, 2.6],       # in percent
        "policy_rate":      [5.25, 4.00, -0.10, 5.00, 4.75, 4.35], # in percent
        "unemployment":     [3.8, 6.5, 2.6, 4.2, 5.1, 4.0],       # in percent
        "fx_usd":           [1.00, 1.09, 143.5, 0.79, 1.36, 1.51] # USD per
         ↪  unit of local currency
    },
    index=["A", "B", "C", "D", "E", "F"]
)
df
```

```
        area  year  gdp_growth  inflation  policy_rate  unemployment  fx_usd
A        USA  2024         2.1        3.2         5.25           3.8    1.00
B   Eurozone  2024         1.3        2.5         4.00           6.5    1.09
C      Japan  2024         0.7        1.0        -0.10           2.6  143.50
D         UK  2024         1.5        2.8         5.00           4.2    0.79
E     Canada  2024         1.8        2.2         4.75           5.1    1.36
F  Australia  2024         2.0        2.6         4.35           4.0    1.51
```

First, we will set the “areas” column as the index of the DataFrame. This will allow us to access rows by area name. We can do this using the `set_index()` method.

```
df = df.set_index("area")
```

We could also do it in-place (modifying the original DataFrame directly)

```
df.set_index("area", inplace=True)
```

2.6.2.2.2 Inspecting DataFrames

You can inspect the first few rows of a DataFrame using the `head()` method and the last few rows using the `tail()` method. By default, both methods display 5 rows, but you can specify a different number as an argument.

```
df.head()  # First 5 rows
```

```
          year  gdp_growth  inflation  policy_rate  unemployment  fx_usd
area
USA       2024         2.1        3.2         5.25           3.8    1.00
Eurozone  2024         1.3        2.5         4.00           6.5    1.09
Japan     2024         0.7        1.0        -0.10           2.6  143.50
UK        2024         1.5        2.8         5.00           4.2    0.79
Canada    2024         1.8        2.2         4.75           5.1    1.36
```

```
df.tail(3)  # Last 3 rows
```

```
           year  gdp_growth  inflation  policy_rate  unemployment  fx_usd
area
UK         2024         1.5        2.8         5.00           4.2    0.79
Canada     2024         1.8        2.2         4.75           5.1    1.36
Australia  2024         2.0        2.6         4.35           4.0    1.51
```

You can get a summary of the DataFrame using the `info()` method, which provides information about the index, columns, data types, and memory usage.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 6 entries, USA to Australia
Data columns (total 6 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   year          6 non-null      int64  
 1   gdp_growth    6 non-null      float64
 2   inflation     6 non-null      float64
 3   policy_rate   6 non-null      float64
 4   unemployment  6 non-null      float64
 5   fx_usd        6 non-null      float64
dtypes: float64(5), int64(1)
memory usage: 336.0+ bytes
```

You can get basic statistical details of the DataFrame using the `describe()` method, which provides measures like mean, standard deviation, min, max, and quartiles for numerical columns.

```
df.describe()
```

```
         year  gdp_growth  inflation  policy_rate  unemployment      fx_usd
count     6.0    6.000000   6.000000     6.000000      6.000000    6.000000
mean   2024.0    1.566667   2.383333     3.875000      4.366667   24.875000
std       0.0    0.520256   0.754763     1.998187      1.318585   58.114711
min    2024.0    0.700000   1.000000    -0.100000      2.600000    0.790000
25%    2024.0    1.350000   2.275000     4.087500      3.850000    1.022500
50%    2024.0    1.650000   2.550000     4.550000      4.100000    1.225000
75%    2024.0    1.950000   2.750000     4.937500      4.875000    1.472500
max    2024.0    2.100000   3.200000     5.250000      6.500000  143.500000
```

2.6.2.2.3 Indexing and Selecting DataFrames

We can get a single column as a Series using python's getitem syntax on the DataFrame object.

```
df['inflation'] # returns a series
```

```
area
USA          3.2
Eurozone     2.5
Japan        1.0
UK           2.8
Canada       2.2
Australia    2.6
Name: inflation, dtype: float64
```

```
type(df['inflation'])
```

```
<class 'pandas.core.series.Series'>
```

…or using attribute syntax.

```
df.inflation  # returns a series
```

```
area
USA          3.2
Eurozone     2.5
Japan        1.0
UK           2.8
Canada       2.2
Australia    2.6
Name: inflation, dtype: float64
```

If we use a list of column names, we get a DataFrame back

```
df[['inflation']]  # returns a DataFrame
```

```
           inflation
area
USA              3.2
Eurozone         2.5
Japan            1.0
UK               2.8
Canada           2.2
Australia        2.6
```

```
type(df[['inflation']])
```

```
<class 'pandas.core.frame.DataFrame'>
```

This is useful for selecting multiple columns at once.

```
df[['inflation', 'unemployment']]  # returns a dataframe with selected
 ↪  columns
```

```
           inflation  unemployment
area
USA              3.2           3.8
Eurozone         2.5           6.5
Japan            1.0           2.6
UK               2.8           4.2
Canada           2.2           5.1
Australia        2.6           4.0
```

We can use `.loc` to select rows and columns by label, and `.iloc` to select rows and columns by position.

- `.loc` uses labels (names) for both rows and columns. The syntax is `df.loc[rows, columns]`. Both can be single labels, lists, or slices. Slices with `.loc` are **inclusive** of the end.
- `.iloc` uses integer positions (like Python lists). The syntax is `df.iloc[rows, columns]`. Slices with `.iloc` are **exclusive** of the end (like standard Python slicing).

Suppose `df` looks like this:

-	name	age	city
0	Alice	23	Madrid
1	Bob	34	London
2	Carol	29	Berlin

- `df['age']` or `df.age` -> Series with ages.
- `df[['name', 'city']]` -> DataFrame with just name and city columns.
- `df.loc[1, 'city']` -> `'London'` (row label 1, column 'city').
- `df.loc[0:1, ['name', 'age']]` -> Rows 0 to 1, columns 'name' and 'age' (inclusive).
- `df.iloc[0:2, 1:3]` -> Rows 0 to 1, columns 1 and 2 (note that row 2 and column 3 are not included).
- `df[df['age'] > 25]` -> Rows where age is greater than 25.

As indicated above, both `.loc` and `.iloc` can take single labels/positions, lists of labels/positions, or slices. Here are some additional tips:

- Use `:` to select all rows or columns:
  - `df.loc[:, 'age']` (all rows, 'age' column).
  - `df.iloc[1, :]` (row 1, all columns).
- Remember: `.loc` is label-based and **inclusive**; `.iloc` is position-based and **exclusive**.

```
df.loc["UK","gdp_growth"] # get the value in row "UK" and column "gdp_growth"
```

```
np.float64(1.5)
```

```
df.iloc[3,1] # get the value in row 3 and column 1 (recall: python uses
 ↪  zero-based index)
```

```
np.float64(1.5)
```

You can also get subsets of rows and columns using slices or lists

```
df.loc["USA":"UK",["policy_rate", "fx_usd"]] # Subset rows from "USA" to "UK"
 ↪  and columns "policy_rate" and "fx_usd"
```

```
          policy_rate  fx_usd
area
USA              5.25    1.00
Eurozone         4.00    1.09
Japan           -0.10  143.50
UK               5.00    0.79
```

We can filter rows based on a boolean condition.

```
df[df['unemployment'] > 5.0]  # returns a dataframe with rows where
↪  unemployment is greater than 5.0
```

```
          year  gdp_growth  inflation  policy_rate  unemployment  fx_usd
area
Eurozone  2024         1.3        2.5         4.00           6.5    1.09
Canada    2024         1.8        2.2         4.75           5.1    1.36
```

To filter rows in a DataFrame based on multiple conditions, you can use logical operators:

Operator	Symbol	Meaning	General Pattern
AND	`&`	All conditions must be true	`df[(condition1) & (condition2)]`
OR	`|`	At least one condition must be true	`df[(condition1) | (condition2)]`
NOT	`~`	Negates a condition (condition is false)	`df[~(condition)]`

You can combine these operators to build more complex filters as needed. For example

`df[(condition1 & condition2) | (~condition3 & condition4)]`

To reduce the likelihood of mistakes, always enclose each condition in parentheses to ensure correct evaluation.

The following example filters the DataFrame to include only rows where the `fx_usd` is less than 1.0 **and** the `inflation` is greater than 2.0:

```
df[(df['fx_usd'] < 1.0) & (df["inflation"] > 2.0)]
```

```
      year  gdp_growth  inflation  policy_rate  unemployment  fx_usd
area
UK    2024         1.5        2.8          5.0           4.2    0.79
```

An alternative to boolean indexing is the `query()` method, which allows you to filter rows using a string expression. This can be more readable, especially for complex conditions:

```
df.query("fx_usd < 1.0 and inflation > 2.0")
```

```
      year  gdp_growth  inflation  policy_rate  unemployment  fx_usd
area
UK    2024         1.5        2.8          5.0           4.2    0.79
```

The `query()` method supports standard comparison operators (`<`, `>`, `==`, `!=`, `<=`, `>=`) and logical operators (`and`, `or`, `not`). You can also reference variables from the local environment using the `@` prefix:

```
threshold = 2.0
df.query("inflation > @threshold")
```

```
           year  gdp_growth  inflation  policy_rate  unemployment  fx_usd
area
USA        2024         2.1        3.2         5.25           3.8    1.00
Eurozone   2024         1.3        2.5         4.00           6.5    1.09
UK         2024         1.5        2.8         5.00           4.2    0.79
Canada     2024         1.8        2.2         4.75           5.1    1.36
Australia  2024         2.0        2.6         4.35           4.0    1.51
```

2.6.2.2.4 DataFrame Operations

There are many operations you can perform on DataFrames. Here are some common ones:

Adding Columns:

Method	Code Pattern (Abstraction)	Notes
Direct assign	`df[new_col] = values`	Adds or overwrites a column
`assign()`	`df.assign(new_col=values)`	Adds a new column (returns a new DataFrame)
`insert()`	`df.insert(loc, new_col, values)`	Adds at specific position
Multiple cols	`df[[col1, col2]] = values`	Assign multiple columns at once

Adding Rows:

Method	Code Pattern (Abstraction)	Notes
`loc`	`df.loc[new_label] = values`	Adds or overwrites a row by index label
`iloc`	`df.iloc[position] = values`	Overwrites a row at a specific integer position (does not add a new row)
`concat()`	`df = pd.concat([df, new_rows_df])`	Adds one or more new rows from another DataFrame

For example, to add a new column that approximates real GDP growth (i.e., nominal GDP growth minus inflation):

```
df["real_gdp_growth"] = df.gdp_growth - df.inflation  # Create a new column
 ↪  as the difference between gdp_growth and inflation
df["avg_weather"] = [20.5, 18.0, 15.0, 12.5, 10.0, 22.0]  # Add a new column
 ↪  with average weather data
df
```

```
           year  gdp_growth  inflation  ...  fx_usd  real_gdp_growth
           avg_weather
area                                    ...
USA        2024         2.1        3.2  ...    1.00             -1.1
20.5
Eurozone   2024         1.3        2.5  ...    1.09             -1.2
18.0
Japan      2024         0.7        1.0  ...  143.50             -0.3
15.0
UK         2024         1.5        2.8  ...    0.79             -1.3
12.5
Canada     2024         1.8        2.2  ...    1.36             -0.4
10.0
Australia  2024         2.0        2.6  ...    1.51             -0.6
22.0

[6 rows x 8 columns]
```

Using `assign()`, we can do the same without modifying the original DataFrame (note that `assign()` returns a new DataFrame):

```
df = df.drop(columns=["real_gdp_growth"])  # Remove previously added column
df_new = df.assign(real_gdp_growth=df.gdp_growth - df.inflation)
df_new
```

```
           year  gdp_growth  inflation  ...  fx_usd  avg_weather
           real_gdp_growth
area                                    ...
USA        2024         2.1        3.2  ...    1.00         20.5
-1.1
Eurozone   2024         1.3        2.5  ...    1.09         18.0
-1.2
Japan      2024         0.7        1.0  ...  143.50         15.0
-0.3
```

```
UK          2024          1.5         2.8  ...    0.79          12.5
-1.3
Canada      2024          1.8         2.2  ...    1.36          10.0
-0.4
Australia   2024          2.0         2.6  ...    1.51          22.0
-0.6

[6 rows x 8 columns]
```

Using `insert()`, we can add a new column at a specific position. For example, to insert a `gdp_per_capita` column as the second column (index 1):

```
df.insert(
    loc=1,  # Insert at the second position (0-based index)
    column='gdp_per_capita',  # Name of the new column
    value=[60000, np.nan, 40000, np.nan, 55000, 70000]  # Values for the new
 ↪  column
)
df
```

```
           year  gdp_per_capita  gdp_growth  ...  unemployment  fx_usd
           avg_weather
area                                         ...
USA        2024         60000.0         2.1  ...           3.8    1.00
20.5
Eurozone   2024             NaN         1.3  ...           6.5    1.09
18.0
Japan      2024         40000.0         0.7  ...           2.6  143.50
15.0
UK         2024             NaN         1.5  ...           4.2    0.79
12.5
Canada     2024         55000.0         1.8  ...           5.1    1.36
10.0
Australia  2024         70000.0         2.0  ...           4.0    1.51
22.0

[6 rows x 8 columns]
```

Deleting data:

What to Remove	Method/Option	Code Pattern (Abstraction)	Notes
Columns by label	`drop()`	`df.drop([col1, col2, ...], axis=1)`	Returns new DataFrame
Columns by label (in-place)	`drop()`	`df.drop([col1, col2, ...], axis=1, inplace=True)`	Modifies original DataFrame
Columns by position	`drop()`	`df.drop(df.columns[[pos1, pos2, ...]], axis=1)`	Use integer positions
Columns with missing values	`dropna()`	`df.dropna(axis=1)`	Removes columns with any missing
Rows by label	`drop()`	`df.drop([row1, row2, ...], axis=0)`	Returns new DataFrame
Rows by label (in-place)	`drop()`	`df.drop([row1, row2, ...], axis=0, inplace=True)`	Modifies original DataFrame
Rows by position	`drop()`	`df.drop(df.index[[pos1, pos2, ...]], axis=0)`	Use integer positions
Rows with missing values	`dropna()`	`df.dropna(axis=0)`	Removes rows with any missing
Duplicate rows	`drop_duplicates()`	`df.drop_duplicates()`	Removes duplicate rows

For example, to remove the `avg_weather` column we just added

```
df.drop("avg_weather", axis=1)
```

```
           year  gdp_per_capita  gdp_growth  ...  policy_rate  unemployment
fx_usd
area                                         ...
USA        2024         60000.0         2.1  ...         5.25           3.8
1.00
Eurozone   2024             NaN         1.3  ...         4.00           6.5
1.09
Japan      2024         40000.0         0.7  ...        -0.10           2.6
143.50
UK         2024             NaN         1.5  ...         5.00           4.2
0.79
Canada     2024         55000.0         1.8  ...         4.75           5.1
1.36
Australia  2024         70000.0         2.0  ...         4.35           4.0
1.51
```

```
[6 rows x 7 columns]
```

We can also drop columns with NaN values

```
df.dropna(axis=1)  # Drops columns with any NaN values
```

```
           year  gdp_growth  inflation  ...  unemployment  fx_usd
           avg_weather
area                                    ...
USA        2024         2.1        3.2  ...           3.8    1.00
20.5
Eurozone   2024         1.3        2.5  ...           6.5    1.09
18.0
Japan      2024         0.7        1.0  ...           2.6  143.50
15.0
UK         2024         1.5        2.8  ...           4.2    0.79
12.5
Canada     2024         1.8        2.2  ...           5.1    1.36
10.0
Australia  2024         2.0        2.6  ...           4.0    1.51
22.0

[6 rows x 7 columns]
```

Or fill it up with default "fallback" data:

```
df.fillna(df.gdp_per_capita.median())  # Fills NaN values with the median of
↪  the gdp_per_capita column
```

```
           year  gdp_per_capita  gdp_growth  ...  unemployment  fx_usd
           avg_weather
area                                         ...
USA        2024         60000.0         2.1  ...           3.8    1.00
20.5
Eurozone   2024         57500.0         1.3  ...           6.5    1.09
18.0
Japan      2024         40000.0         0.7  ...           2.6  143.50
15.0
UK         2024         57500.0         1.5  ...           4.2    0.79
12.5
Canada     2024         55000.0         1.8  ...           5.1    1.36
10.0
```

```
Australia  2024          70000.0          2.0  ...             4.0    1.51
22.0

[6 rows x 8 columns]
```

Note that both `drop()` and `fillna()` return a new DataFrame by default. Thus, when we access `df` again, we will see that it still contains the `avg_weather` column and any NaN values.

```
df  # Original DataFrame remains unchanged
```

```
           year  gdp_per_capita  gdp_growth  ...  unemployment  fx_usd
           avg_weather
area                                         ...
USA        2024          60000.0          2.1  ...             3.8    1.00
20.5
Eurozone   2024              NaN          1.3  ...             6.5    1.09
18.0
Japan      2024          40000.0          0.7  ...             2.6  143.50
15.0
UK         2024              NaN          1.5  ...             4.2    0.79
12.5
Canada     2024          55000.0          1.8  ...             5.1    1.36
10.0
Australia  2024          70000.0          2.0  ...             4.0    1.51
22.0

[6 rows x 8 columns]
```

We can also sort the entries in dataframes, e.g. alphabetically by index or numerically by column values

What to Sort	Method/Option	Code Pattern (Abstraction)	Notes
By column(s)	`sort_values()`	`df.sort_values(by=col)`	Sort by one column (ascending by default)
By multiple columns	`sort_values()`	`df.sort_values(by=[col1, col2])`	Sort by several columns (priority order)
By column(s), descending	`sort_values()`	`df.sort_values(by=col, ascending=False)`	Sort in descending order
By multiple columns, custom order	`sort_values()`	`df.sort_values(by=[col1, col2], ascending=[True, False])`	Custom order for each column

What to Sort	Method/Option	Code Pattern (Abstraction)	Notes
By index	`sort_index()`	`df.sort_index()`	Sort by row index (ascending by default)
By index, descending	`sort_index()`	`df.sort_index(ascending=False)`	Sort index in descending order
By columns (column labels)	`sort_index()`	`df.sort_index(axis=1)`	Sort columns by their labels
By columns, descending	`sort_index()`	`df.sort_index(axis=1, ascending=False)`	Sort columns in descending order

For example, to sort the DataFrame by `inflation` in descending order

```
df.sort_values(by='inflation', ascending=False)
```

```
           year  gdp_per_capita  gdp_growth  ...  unemployment  fx_usd
           avg_weather
area                                         ...
USA        2024         60000.0         2.1  ...           3.8    1.00
20.5
UK         2024             NaN         1.5  ...           4.2    0.79
12.5
Australia  2024         70000.0         2.0  ...           4.0    1.51
22.0
Eurozone   2024             NaN         1.3  ...           6.5    1.09
18.0
Canada     2024         55000.0         1.8  ...           5.1    1.36
10.0
Japan      2024         40000.0         0.7  ...           2.6  143.50
15.0

[6 rows x 8 columns]
```

To sort by multiple columns, e.g., first by `year` (ascending) and then by `gdp_growth` (descending):

```
df.sort_values(by=['year', 'gdp_growth'], ascending=[True, False])
```

```
           year  gdp_per_capita  gdp_growth  ...  unemployment  fx_usd
           avg_weather
```

```
area                                         ...
USA        2024          60000.0          2.1  ...          3.8    1.00
20.5
Australia  2024          70000.0          2.0  ...          4.0    1.51
22.0
Canada     2024          55000.0          1.8  ...          5.1    1.36
10.0
UK         2024              NaN          1.5  ...          4.2    0.79
12.5
Eurozone   2024              NaN          1.3  ...          6.5    1.09
18.0
Japan      2024          40000.0          0.7  ...          2.6  143.50
15.0

[6 rows x 8 columns]
```

We can also sort by index

```
df.sort_index()
```

```
           year  gdp_per_capita  gdp_growth  ...  unemployment  fx_usd
           avg_weather
area                                         ...
Australia  2024          70000.0          2.0  ...          4.0    1.51
22.0
Canada     2024          55000.0          1.8  ...          5.1    1.36
10.0
Eurozone   2024              NaN          1.3  ...          6.5    1.09
18.0
Japan      2024          40000.0          0.7  ...          2.6  143.50
15.0
UK         2024              NaN          1.5  ...          4.2    0.79
12.5
USA        2024          60000.0          2.1  ...          3.8    1.00
20.5

[6 rows x 8 columns]
```

or column names

```
df.sort_index(axis=1)
```

```
           avg_weather  fx_usd  gdp_growth  ...  policy_rate  unemployment
           year
area                                        ...
USA               20.5    1.00         2.1  ...         5.25           3.8
2024
Eurozone          18.0    1.09         1.3  ...         4.00           6.5
2024
Japan             15.0  143.50         0.7  ...        -0.10           2.6
2024
UK                12.5    0.79         1.5  ...         5.00           4.2
2024
Canada            10.0    1.36         1.8  ...         4.75           5.1
2024
Australia         22.0    1.51         2.0  ...         4.35           4.0
2024

[6 rows x 8 columns]
```

Pandas supports a wide range of methods for merging different datasets. These are described extensively in the documentation. Here we just give a few examples.

Method	Function	Description	Key Parameters	Use Case
Inner Join	`pd.merge(df1, df2, how='inner')`	Returns only rows with matching keys in both dataframes	`on`, `left_on`, `right_on`	When you only want records that exist in both datasets
Left Join	`pd.merge(df1, df2, how='left')`	Returns all rows from left dataframe, matching rows from right	`on`, `left_on`, `right_on`	Keep all records from primary dataset, add matching info
Right Join	`pd.merge(df1, df2, how='right')`	Returns all rows from right dataframe, matching rows from left	`on`, `left_on`, `right_on`	Keep all records from secondary dataset
Outer Join	`pd.merge(df1, df2, how='outer')`	Returns all rows from both dataframes	`on`, `left_on`, `right_on`	When you want all records from both datasets

Method	Function	Description	Key Parameters	Use Case
Cross Join	`pd.merge(df1, df2, how='cross')`	Cartesian product of both dataframes	None required	Create all possible combinations
Concat Vertical	`pd.concat([df1, df2])`	Stacks dataframes vertically (rows)	`axis=0`, `ignore_index`	Combine datasets with same columns
Concat Horizontal	`pd.concat([df1, df2], axis=1)`	Joins dataframes horizontally (columns)	`axis=1`, `join`	Combine datasets with same index
Join Method	`df1.join(df2)`	Left join based on index	`how`, `lsuffix`, `rsuffix`	Quick join on index when columns don't overlap

```
df_trade = pd.DataFrame({
    "area": ["USA", "Eurozone", "Japan", "China", "India", "Brazil"],
    "exports_bn": [1650, 2200, 705, 3360, 323, 281],
    "imports_bn": [2407, 2000, 641, 2601, 507, 219],
    "trade_balance": [-757, 200, 64, 759, -184, 62]
}).set_index("area")
df_trade
```

```
          exports_bn  imports_bn  trade_balance
area
USA             1650        2407           -757
Eurozone        2200        2000            200
Japan            705         641             64
China           3360        2601            759
India            323         507           -184
Brazil           281         219             62
```

```
inner_result = pd.merge(df, df_trade, how='inner', left_index=True,
↪   right_index=True)
inner_result
```

```
          year  gdp_per_capita  ...  imports_bn  trade_balance
area                            ...
USA       2024         60000.0  ...        2407           -757
Eurozone  2024             NaN  ...        2000            200
Japan     2024         40000.0  ...         641             64
```

```
[3 rows x 11 columns]
```

```
left_result = pd.merge(df, df_trade, how='left', left_index=True,
↪  right_index=True)
left_result
```

```
           year  gdp_per_capita  ...  imports_bn  trade_balance
area                             ...
USA        2024         60000.0  ...      2407.0         -757.0
Eurozone   2024             NaN  ...      2000.0          200.0
Japan      2024         40000.0  ...       641.0           64.0
UK         2024             NaN  ...         NaN            NaN
Canada     2024         55000.0  ...         NaN            NaN
Australia  2024         70000.0  ...         NaN            NaN

[6 rows x 11 columns]
```

```
right_result = pd.merge(df, df_trade, how='right', left_index=True,
↪  right_index=True)
right_result
```

```
            year  gdp_per_capita  ...  imports_bn  trade_balance
area                              ...
USA       2024.0         60000.0  ...        2407           -757
Eurozone  2024.0             NaN  ...        2000            200
Japan     2024.0         40000.0  ...         641             64
China        NaN             NaN  ...        2601            759
India        NaN             NaN  ...         507           -184
Brazil       NaN             NaN  ...         219             62

[6 rows x 11 columns]
```

```
outer_result = pd.merge(df, df_trade, how='outer', left_index=True,
↪  right_index=True)
outer_result
```

```
             year  gdp_per_capita  ...  imports_bn  trade_balance
area                               ...
Australia  2024.0         70000.0  ...         NaN            NaN
Brazil        NaN             NaN  ...       219.0           62.0
Canada     2024.0         55000.0  ...         NaN            NaN
```

```
China          NaN             NaN  ...      2601.0          759.0
Eurozone    2024.0             NaN  ...      2000.0          200.0
India          NaN             NaN  ...       507.0         -184.0
Japan       2024.0         40000.0  ...       641.0           64.0
UK          2024.0             NaN  ...         NaN            NaN
USA         2024.0         60000.0  ...      2407.0         -757.0

[9 rows x 11 columns]
```

```
pd.concat([df, df_trade], axis=1).sort_index()  # Concatenate along columns
```

```
              year  gdp_per_capita  ...  imports_bn  trade_balance
area                                ...
Australia   2024.0         70000.0  ...         NaN            NaN
Brazil         NaN             NaN  ...       219.0           62.0
Canada      2024.0         55000.0  ...         NaN            NaN
China          NaN             NaN  ...      2601.0          759.0
Eurozone    2024.0             NaN  ...      2000.0          200.0
India          NaN             NaN  ...       507.0         -184.0
Japan       2024.0         40000.0  ...       641.0           64.0
UK          2024.0             NaN  ...         NaN            NaN
USA         2024.0         60000.0  ...      2407.0         -757.0

[9 rows x 11 columns]
```

```
pd.concat([df, df_trade], axis=0)  # Concatenate along rows
```

```
              year  gdp_per_capita  ...  imports_bn  trade_balance
area                                ...
USA         2024.0         60000.0  ...         NaN            NaN
Eurozone    2024.0             NaN  ...         NaN            NaN
Japan       2024.0         40000.0  ...         NaN            NaN
UK          2024.0             NaN  ...         NaN            NaN
Canada      2024.0         55000.0  ...         NaN            NaN
Australia   2024.0         70000.0  ...         NaN            NaN
USA            NaN             NaN  ...      2407.0         -757.0
Eurozone       NaN             NaN  ...      2000.0          200.0
Japan          NaN             NaN  ...       641.0           64.0
China          NaN             NaN  ...      2601.0          759.0
India          NaN             NaN  ...       507.0         -184.0
Brazil         NaN             NaN  ...       219.0           62.0

[12 rows x 11 columns]
```

Sometimes it can be useful to apply a function to all values of a column/row. For instance, we might be interested in normalised inflation. We can do this using the `apply()` method. This method applies a function to each element in the Series or DataFrame.

```
df.inflation.apply(lambda x: (x - df.inflation.mean()) / df.inflation.std()) # Standardize the inflation column
```

```
area
USA          1.082018
Eurozone     0.154574
Japan       -1.832806
UK           0.552050
Canada      -0.242902
Australia    0.287066
Name: inflation, dtype: float64
```

Sometimes it is necessary to rename columns or indices in a DataFrame. There are several ways to do this, depending on whether you want to rename all columns, specific columns, or apply a function to transform the names.

Method	Syntax	Use Case	Example
Direct Assignment	`df.columns = [list]`	Replace all column names at once	`df.columns = ['A', 'B', 'C']`
rename() with Dictionary	`df.rename(columns={dict})`	Rename specific columns selectively	`df.rename(columns={'old': 'new'})`
rename() with inplace	`df.rename(columns={dict}, inplace=True)`	Modify original DataFrame directly	`df.rename(columns={'old': 'new'}, inplace=True)`
rename() with Function	`df.rename(columns=function)`	Apply transformation to all columns	`df.rename(columns=str.upper)`
String Methods	`df.columns.str.method()`	Apply string operations to column names	`df.columns = df.columns.str.replace('_', ' ')`
Lambda Function	`df.rename(columns=lambda x: expression)`	Custom transformations on column names	`df.rename(columns=lambda x: x.replace('old', 'new'))`

Key Parameters

Parameter	Description	Default	Example
`columns`	Dictionary or function for column mapping	`None`	`{'old_name': 'new_name'}`
`inplace`	Modify DataFrame in place vs. return copy	`False`	`inplace=True`
`errors`	How to handle missing keys	`'ignore'`	`errors='raise'`

```
df1 = df.copy()  # Create a copy of the DataFrame

df1 = df1.rename(columns={
    "gdp_growth": "gdp_growth_(%)",
    "gdp_per_capita": "gdp_per_capita_($)",
    "inflation": "inflation_rate_(%)",
    "policy_rate": "policy_rate_(%)",
    "unemployment": "unemployment_rate_(%)",
    "fx_usd": "fx_rate_($/X)",
    "avg_weather": "avg_weather_(°C)",
    })  # Rename columns
df1
```

```
           year  gdp_per_capita_($)  ...  fx_rate_($/X)  avg_weather_(°C)
area                                 ...
USA        2024             60000.0  ...           1.00              20.5
Eurozone   2024                 NaN  ...           1.09              18.0
Japan      2024             40000.0  ...         143.50              15.0
UK         2024                 NaN  ...           0.79              12.5
Canada     2024             55000.0  ...           1.36              10.0
Australia  2024             70000.0  ...           1.51              22.0

[6 rows x 8 columns]
```

We can also work directly with column names

```
df1.columns = df.columns.str.replace('_', ' ')
df1
```

```
           year  gdp per capita  gdp growth  ...  unemployment  fx usd  avg
weather
area                                         ...
USA        2024         60000.0         2.1  ...           3.8    1.00
20.5
```

```
Eurozone   2024            NaN         1.3  ...          6.5    1.09
18.0
Japan      2024        40000.0         0.7  ...          2.6  143.50
15.0
UK         2024            NaN         1.5  ...          4.2    0.79
12.5
Canada     2024        55000.0         1.8  ...          5.1    1.36
10.0
Australia  2024        70000.0         2.0  ...          4.0    1.51
22.0

[6 rows x 8 columns]
```

or the row names

```
# Capitalize the first letter of each area name
df1.index = df.index.str.upper() # Convert all area names to uppercase
df1.columns = df.columns.str.capitalize()  # Capitalize the first letter of
↪  each column name
df1
```

```
           Year  Gdp_per_capita  Gdp_growth  ...  Unemployment  Fx_usd
           Avg_weather
area                                         ...
USA        2024        60000.0         2.1  ...          3.8    1.00
20.5
EUROZONE   2024            NaN         1.3  ...          6.5    1.09
18.0
JAPAN      2024        40000.0         0.7  ...          2.6  143.50
15.0
UK         2024            NaN         1.5  ...          4.2    0.79
12.5
CANADA     2024        55000.0         1.8  ...          5.1    1.36
10.0
AUSTRALIA  2024        70000.0         2.0  ...          4.0    1.51
22.0

[6 rows x 8 columns]
```

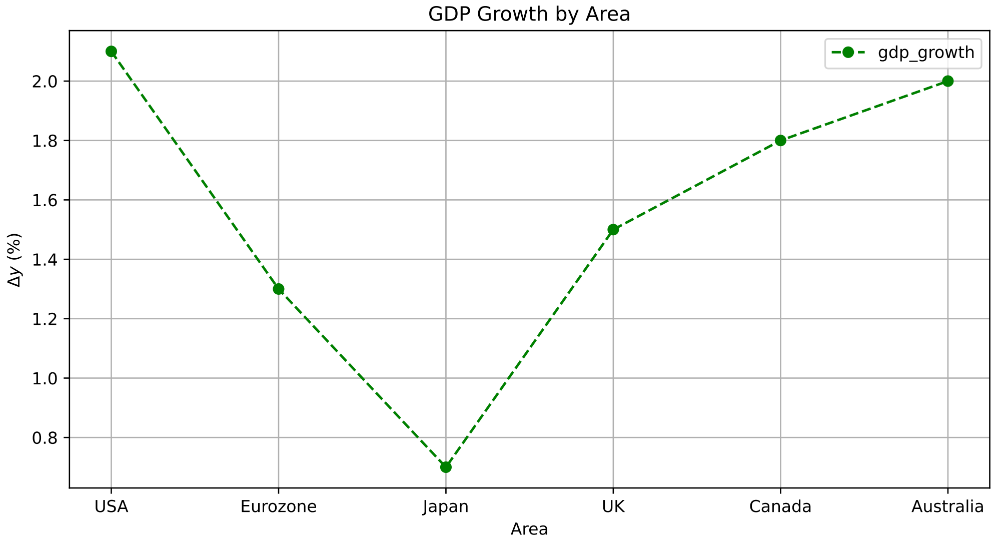
2.6.2.3 Data Visualization with Pandas

DataFrames have all kinds of useful plotting built in. Note that by default `matplotlib` is used as the backend for plotting in Pandas. However, Pandas imports matplotlib for you in the background and you don't have to do it yourself.

You can create various types of plots directly from DataFrames and Series using the `plot()` method. Here are some examples:

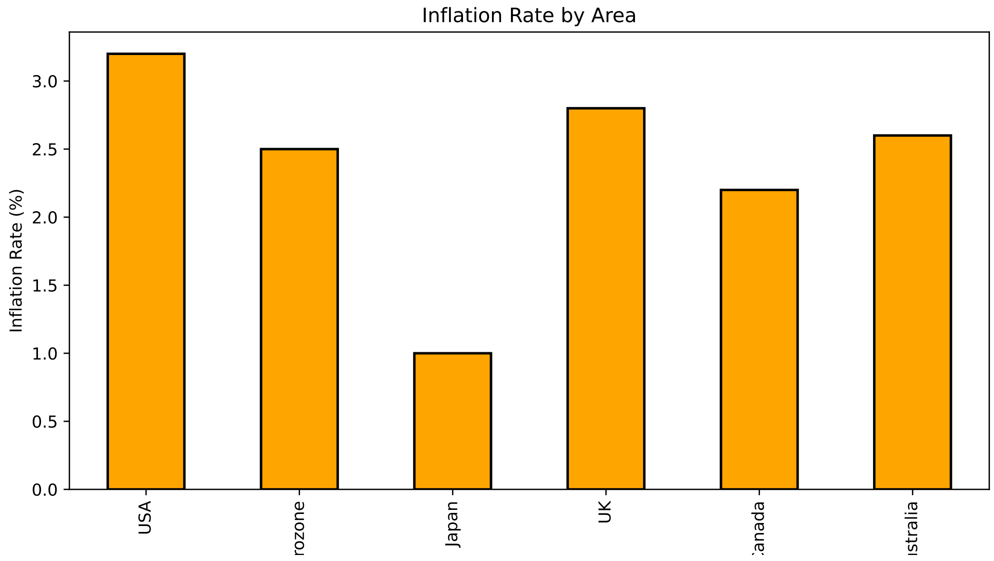
```
df.gdp_growth.plot(
  kind='line',
  title='GDP Growth by Area',
  ylabel=r'$ \Delta y$ (%)',
  xlabel='Area',
  grid=True,
  figsize=(10, 5),
  legend=True,
  color='green',
  marker='o',
  linestyle='--'
)
```

```
<Axes: title={'center': 'GDP Growth by Area'}, xlabel='Area', ylabel='$
\\Delta y$ (%)'>
```



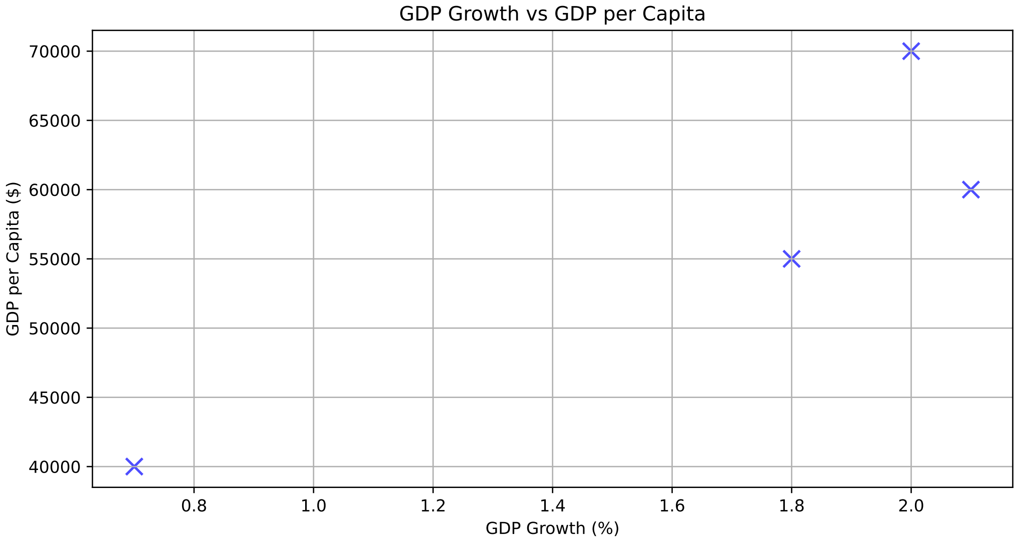
```
df.inflation.plot(
  kind='bar',
  title='Inflation Rate by Area',
  ylabel='Inflation Rate (%)',
  xlabel='Area',
  color="orange",
  grid=False,
  figsize=(10, 5),
  legend=False,
  edgecolor='black',
  linewidth=1.5
)
```

```
<Axes: title={'center': 'Inflation Rate by Area'}, xlabel='Area',
ylabel='Inflation Rate (%)'>
```



```
df.plot(
  kind="scatter",
  x="gdp_growth",
  y="gdp_per_capita",
  title="GDP Growth vs GDP per Capita",
  xlabel="GDP Growth (%)",
  ylabel="GDP per Capita ($)",
  grid=True,
  figsize=(10, 5),
  color="blue",
  marker="x",
  s=100,  # Size of the markers
  alpha=0.7,  # Transparency of the markers
  linewidth=1.5 # Edge width of the markers
)
```

```
<Axes: title={'center': 'GDP Growth vs GDP per Capita'}, xlabel='GDP Growth
(%)', ylabel='GDP per Capita ($)'>
```



2.6.2.4 Importing and Exporting Data

We have seen how to create DataFrames from scratch. However, in practice, we often need to load data from external files or databases. Pandas provides a variety of functions to read and write data in different formats. Data can be imported from CSV, Excel, and more. To read a CSV file into a DataFrame, you can use the `pd.read_csv()` function.

```
file_csv ='./data.csv'
data = pd.read_csv(file_csv)
```

To read an Excel file, you can use the `pd.read_excel()` function.

```
file_excel = './data.xlsx'
data = pd.read_excel(file_excel, sheet_name='Sheet1')
```

To write a DataFrame to a CSV file, you can use the `to_csv()` method.

```
df.to_csv('output.csv', index=False)
```

To write a DataFrame to an Excel file, you can use the `to_excel()` method.

```
df.to_excel('output.xlsx', sheet_name='Sheet1', index=False)
```

We will cover these and other data I/O methods in more detail in later sections of the course.

2.6.2.5 Grouping and Aggregating Data

One of the most powerful features of Pandas is the ability to group data by one or more columns and then apply aggregate functions to each group. This is done using the `groupby()` method, which splits the data into groups based on some criteria, applies a function to each group, and then combines the results.

```
# Create a sample DataFrame with multiple years
df_multi_year = pd.DataFrame({
    "area": ["USA", "USA", "Eurozone", "Eurozone", "Japan", "Japan"],
    "year": [2023, 2024, 2023, 2024, 2023, 2024],
    "gdp_growth": [2.5, 2.1, 0.9, 1.3, 1.2, 0.7],
    "inflation": [4.1, 3.2, 5.4, 2.5, 3.3, 1.0]
})
df_multi_year
```

```
       area  year  gdp_growth  inflation
0       USA  2023         2.5        4.1
1       USA  2024         2.1        3.2
2  Eurozone  2023         0.9        5.4
3  Eurozone  2024         1.3        2.5
4     Japan  2023         1.2        3.3
5     Japan  2024         0.7        1.0
```

To calculate the average GDP growth and inflation for each area across all years:

```
df_multi_year.groupby("area").mean()
```

```
            year  gdp_growth  inflation
area
Eurozone  2023.5        1.10       3.95
Japan     2023.5        0.95       2.15
USA       2023.5        2.30       3.65
```

You can also apply multiple aggregation functions at once using `agg()`:

```
df_multi_year.groupby("area").agg({
    "gdp_growth": ["mean", "std"],
    "inflation": ["min", "max"]
})
```

```
         gdp_growth           inflation
               mean        std       min  max
area
Eurozone       1.10   0.282843       2.5  5.4
Japan          0.95   0.353553       1.0  3.3
USA            2.30   0.282843       3.2  4.1
```

Grouping by multiple columns is also possible:

```
# Group by both area and whether gdp_growth is above 1%
df_multi_year["high_growth"] = df_multi_year["gdp_growth"] > 1.0
df_multi_year.groupby(["area", "high_growth"])["inflation"].mean()
```

```
area      high_growth
Eurozone  False          5.40
          True           2.50
Japan     False          1.00
          True           3.30
USA       True           3.65
Name: inflation, dtype: float64
```

The `groupby()` method is essential for data analysis tasks like computing summary statistics by category, creating pivot tables, and preparing data for visualization.

Scaling Beyond Pandas: PySpark

While Pandas excels at handling data that fits in memory, real-world big data applications often involve datasets too large for a single machine. PySpark is the Python API for Apache Spark, a distributed computing framework that can process massive datasets across clusters of computers. PySpark DataFrames offer a similar interface to Pandas but distribute computations across many machines. For the purposes of this course, we will focus on Pandas, but it's worth noting that many concepts learned here can be transferred to PySpark when working with big data.

2.6.3 Visualization: Matplotlib & Seaborn

Matplotlib is Python's primary library for creating static, animated, and interactive visualizations.

The library is built around two core components:

Figure: The top-level container that holds all plot elements. A figure can contain one or more axes.

Axes: The plotting area where data is displayed. Each axes object includes an x-axis and y-axis (plus z-axis for 3D plots) and provides methods for plotting data points.

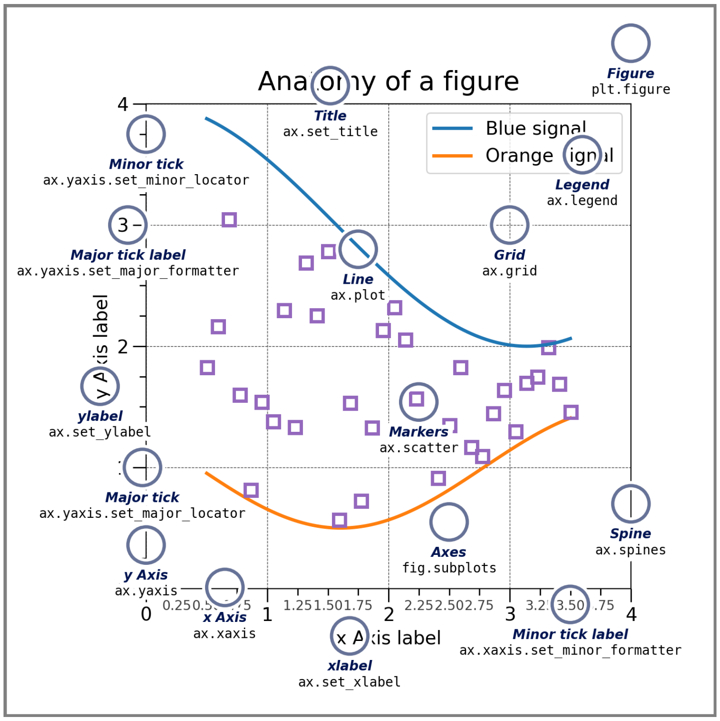


Figure 2.7: Matplotlib Figure and Axes (Source: matplotlib.org)

ℹ Note

Documentation for these packages is available at https://matplotlib.org/stable/ and https://seaborn.pydata.org/api.html.

We can import Matplotlib as follows

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

Seaborn is built on top of Matplotlib and provides a high-level interface for drawing attractive and informative statistical graphics. We can import Seaborn as follows

```
import seaborn as sns
```

For some examples, we won't need seaborn, but we are importing it here because it has some built-in datasets that we can use for visualization. Let's load one of these datasets:

```
# Load the 'tips' dataset from seaborn
df = sns.load_dataset('tips')
df.head()
```

```
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

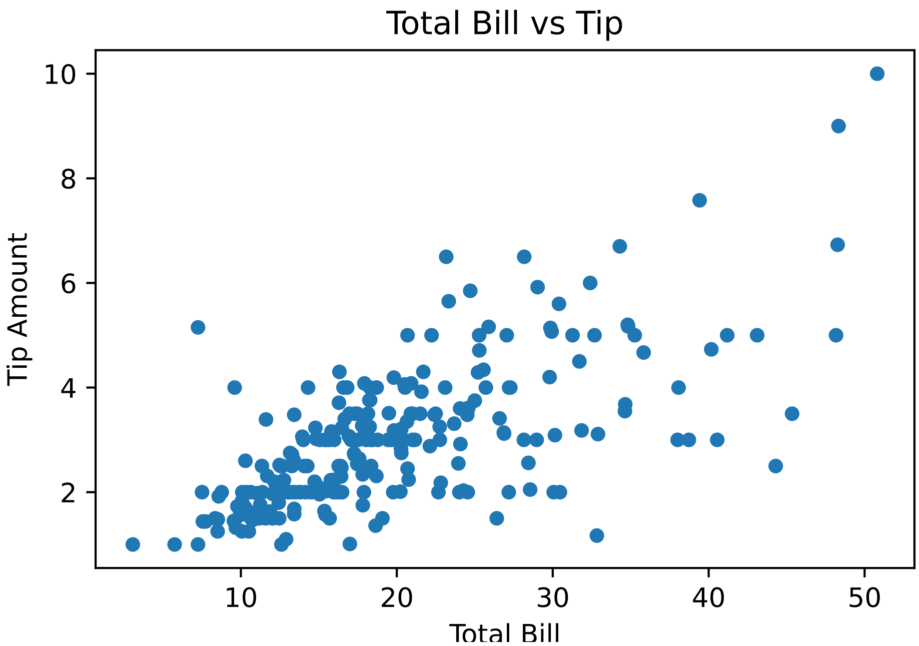
We have loaded a dataset that contains information about tips received by waitstaff in a restaurant, including total bill amount, tip amount, gender of the payer, whether they are a smoker, day of the week, time of day, and size of the party.

We have already seen how to create simple plots using Pandas. For example, we can create a scatter plot of total bill vs. tip using Pandas' built-in plotting capabilities (which uses Matplotlib under the hood)

```
df.plot.scatter(x='total_bill', y='tip', title='Total Bill vs Tip',
 ↪  xlabel='Total Bill', ylabel='Tip Amount')
```

```
<Axes: title={'center': 'Total Bill vs Tip'}, xlabel='Total Bill',
ylabel='Tip Amount'>
```

```
plt.show()
```



Oftentimes, this is enough for making a quick plot. We can use Matplotlib directly

```
plt.figure(figsize=(8, 6))
```

```
<Figure size 800x600 with 0 Axes>
```

```
plt.scatter(df['total_bill'], df['tip'], color='blue')
```

```
<matplotlib.collections.PathCollection object at 0x16e33fc50>
```

```
plt.title('Total Bill vs Tip')
```

```
Text(0.5, 1.0, 'Total Bill vs Tip')
```

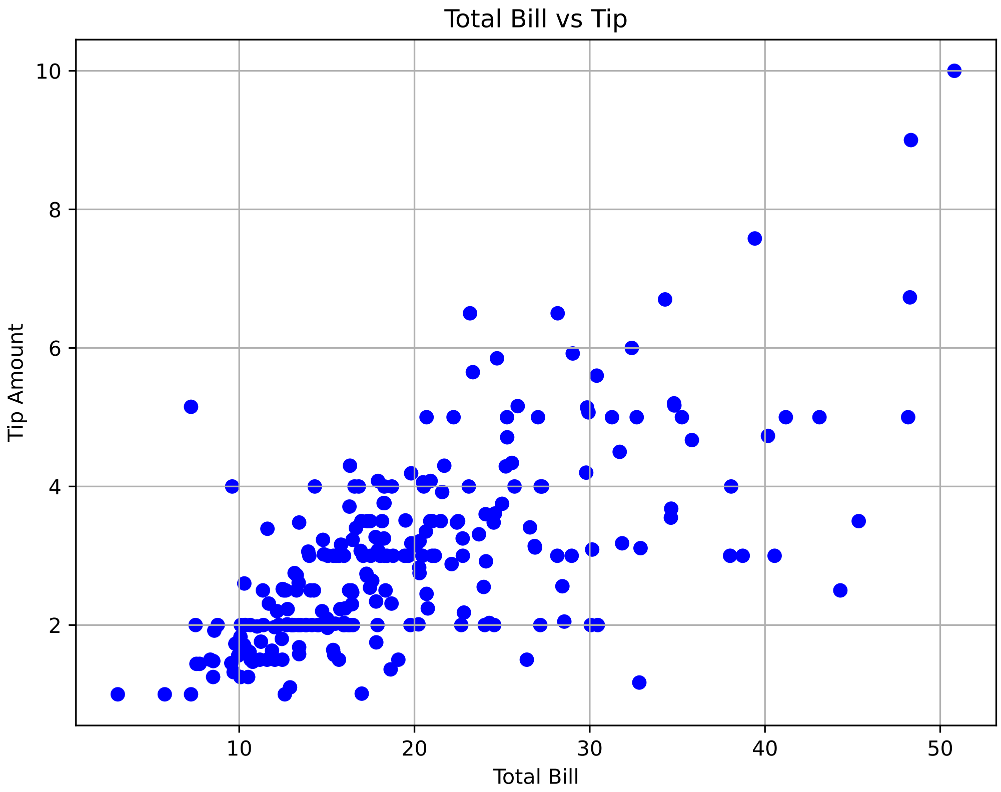
```
plt.xlabel('Total Bill')
```

```
Text(0.5, 0, 'Total Bill')
```

```
plt.ylabel('Tip Amount')
```

```
Text(0, 0.5, 'Tip Amount')
```

```
plt.grid(True)
plt.show()
```



To save a figure to a file, use `plt.savefig('filename.png')`. You can specify different formats (e.g., `.pdf`, `.svg`, `.jpg`) and adjust the resolution with the `dpi` parameter (e.g., `plt.savefig('figure.png', dpi=300)`). In Jupyter notebooks, call `savefig()` before `plt.show()`, as `show()` may clear the figure.

Suppose we want to create a scatter plot that distinguishes between smokers and non-smokers using different colors. We can do this by creating two separate scatter plots and adding them to the same axes

```
plt.figure(figsize=(8, 6))
```

```
<Figure size 800x600 with 0 Axes>
```

```
smokers = df[df['smoker'] == 'Yes']
non_smokers = df[df['smoker'] == 'No']

plt.scatter(smokers['total_bill'], smokers['tip'], color='red',
 ↪  label='Smokers')
```

```
<matplotlib.collections.PathCollection object at 0x16e3e0e10>
```

```
plt.scatter(non_smokers['total_bill'], non_smokers['tip'], color='blue',
 ↪  label='Non-Smokers')
```

```
<matplotlib.collections.PathCollection object at 0x16e36c190>
```

```
plt.title('Total Bill vs Tip by Smoking Status')
```

```
Text(0.5, 1.0, 'Total Bill vs Tip by Smoking Status')
```

```
plt.xlabel('Total Bill')
```

```
Text(0.5, 0, 'Total Bill')
```

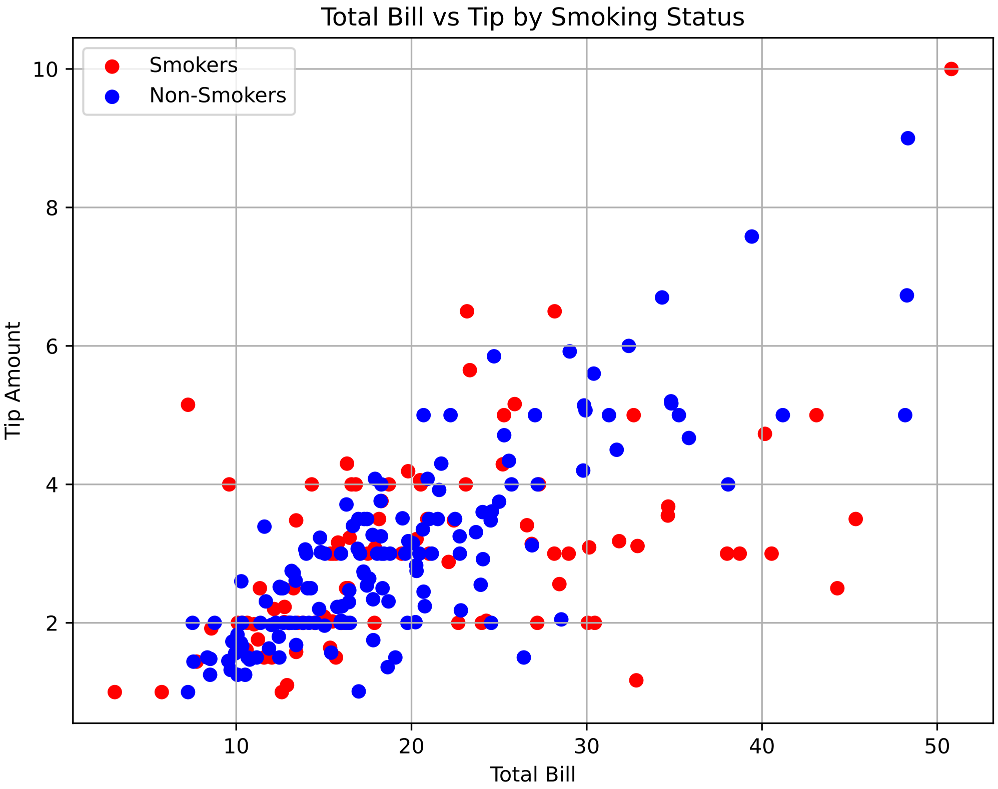
```
plt.ylabel('Tip Amount')
```

```
Text(0, 0.5, 'Tip Amount')
```

```
plt.legend()
```

```
<matplotlib.legend.Legend object at 0x16e36c7d0>
```

```
plt.grid(True)
plt.show()
```



We can also create multiple subplots within a single figure using Matplotlib's `subplots` function

```
fig, axes = plt.subplots(1, 2, figsize=(14, 6))
axes[0].scatter(smokers['total_bill'], smokers['tip'], color='red')
```

```
<matplotlib.collections.PathCollection object at 0x16e3282d0>
```

```
axes[0].set_title('Smokers')
```

```
Text(0.5, 1.0, 'Smokers')
```

```
axes[0].set_xlabel('Total Bill')
```

```
Text(0.5, 0, 'Total Bill')
```

```
axes[0].set_ylabel('Tip Amount')
```

```
Text(0, 0.5, 'Tip Amount')
```

```
axes[1].scatter(non_smokers['total_bill'], non_smokers['tip'], color='blue')
```

```
<matplotlib.collections.PathCollection object at 0x16e328410>
```

```
axes[1].set_title('Non-Smokers')
```

```
Text(0.5, 1.0, 'Non-Smokers')
```

```
axes[1].set_xlabel('Total Bill')
```

```
Text(0.5, 0, 'Total Bill')
```

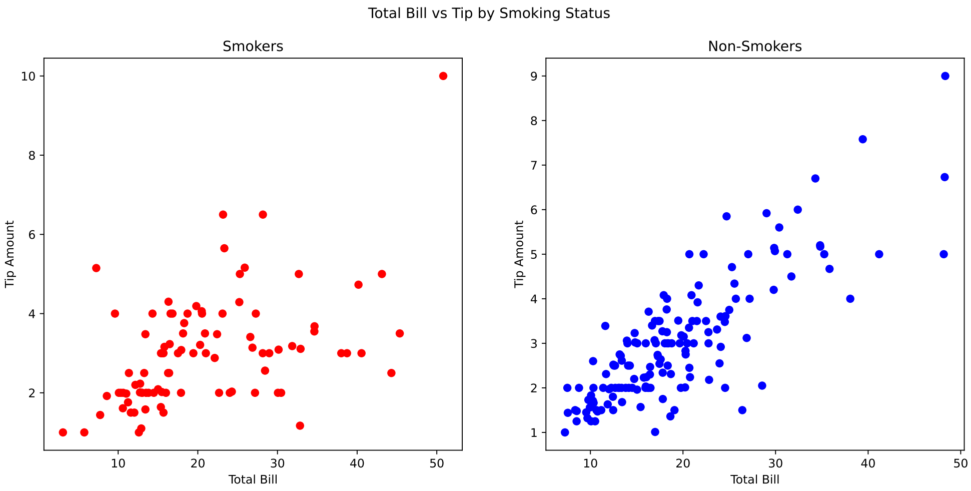
```
axes[1].set_ylabel('Tip Amount')
```

```
Text(0, 0.5, 'Tip Amount')
```

```
plt.suptitle('Total Bill vs Tip by Smoking Status')
```

```
Text(0.5, 0.98, 'Total Bill vs Tip by Smoking Status')
```

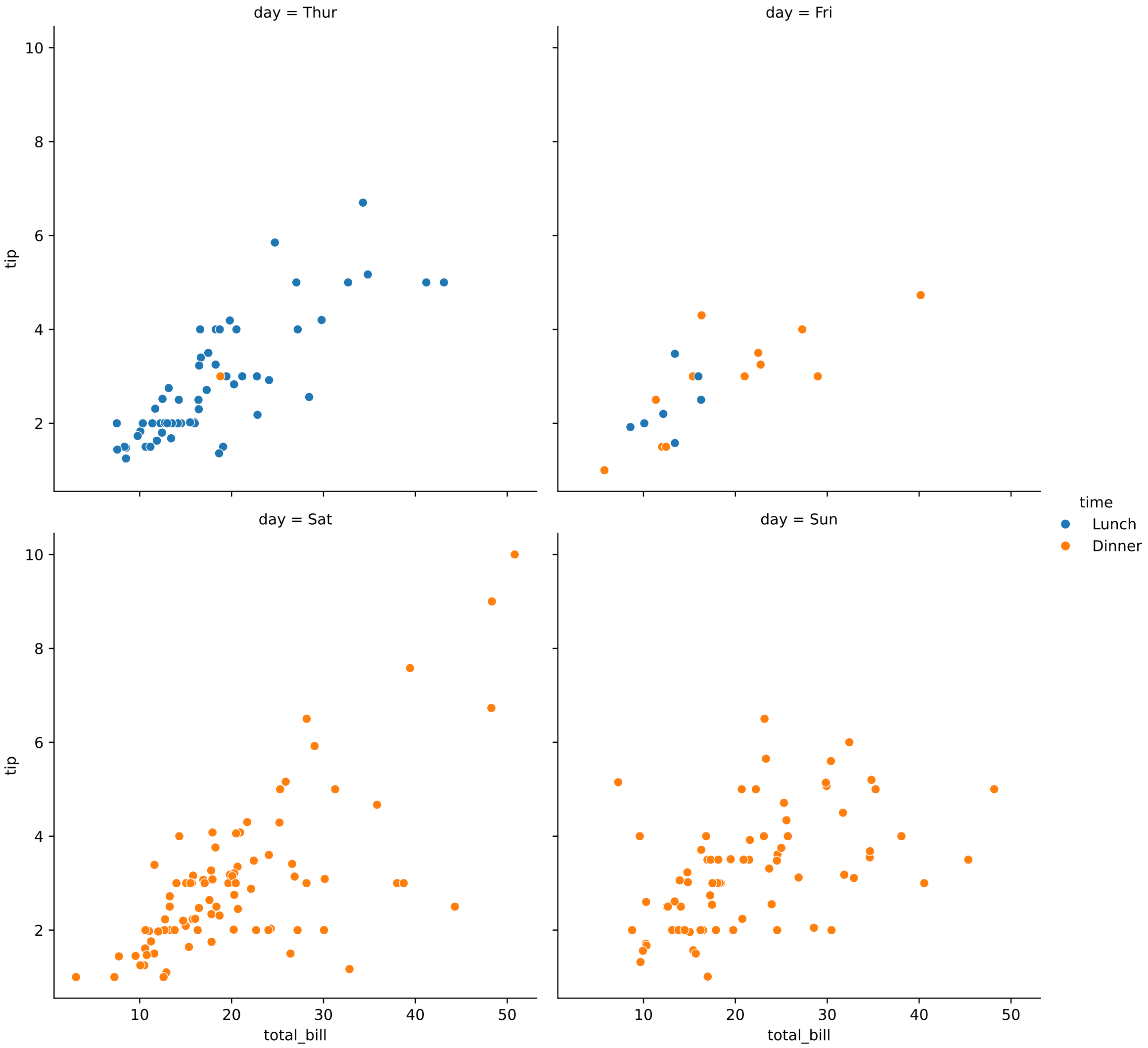
```
plt.show()
```



Seaborn provides a higher-level interface for creating attractive and informative statistical graphics. For example, we can create scatter plots distinguishing between different categories using the relplot function

```
sns.relplot(data=df, x="total_bill", y="tip", hue="time", col="day",
 ↪  col_wrap=2)
```

```
<seaborn.axisgrid.FacetGrid object at 0x16e25cec0>
```



where each subplot corresponds to a different day of the week, and points are colored based on whether the meal was lunch or dinner. We could have created the same plot using Matplotlib, but it would have required more code.

We can also create other types of plots using Seaborn, such as box plots to visualize the distribution of tips by day of the week

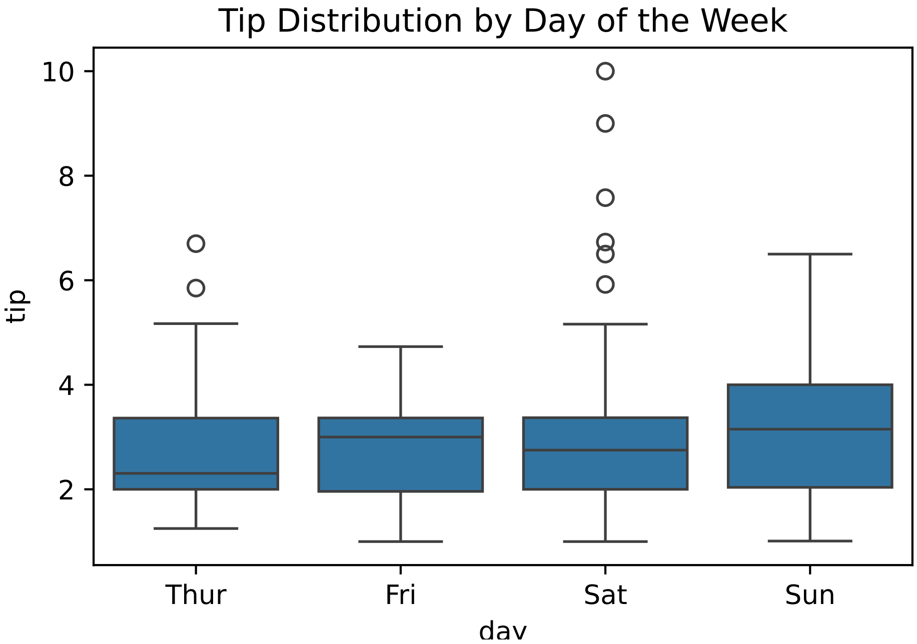
```
sns.boxplot(x='day', y='tip', data=df)
```

```
<Axes: xlabel='day', ylabel='tip'>
```

```
plt.title('Tip Distribution by Day of the Week')
```

```
Text(0.5, 1.0, 'Tip Distribution by Day of the Week')
```

```
plt.show()
```



As you can see, on Saturdays there are some very high tips compared to other days but the median tip on Fridays and Sundays still seems to be higher.

We can also create histograms to visualize the distribution of total bills

```
sns.histplot(df['total_bill'], bins=20, kde=True)
```

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

```
plt.title('Distribution of Total Bills')
```

```
Text(0.5, 1.0, 'Distribution of Total Bills')
```

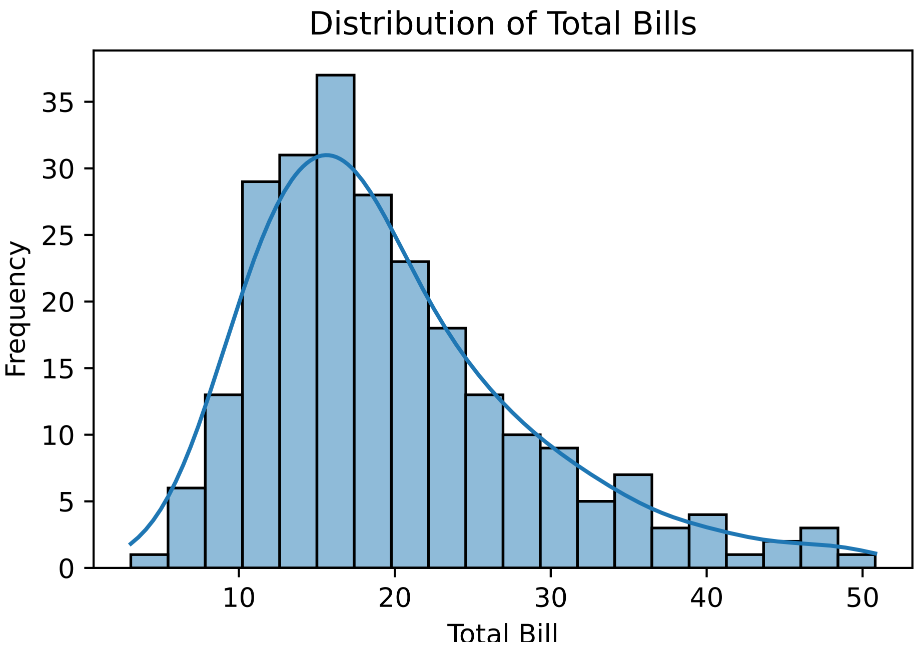
```
plt.xlabel('Total Bill')
```

```
Text(0.5, 0, 'Total Bill')
```

```
plt.ylabel('Frequency')
```

```
Text(0, 0.5, 'Frequency')
```

```
plt.show()
```



where the `kde=True` argument adds a kernel density estimate to the histogram, providing a smoothed curve that represents the distribution of total bills.

We can also create regression plots to visualize the relationship between total bill and tip amount

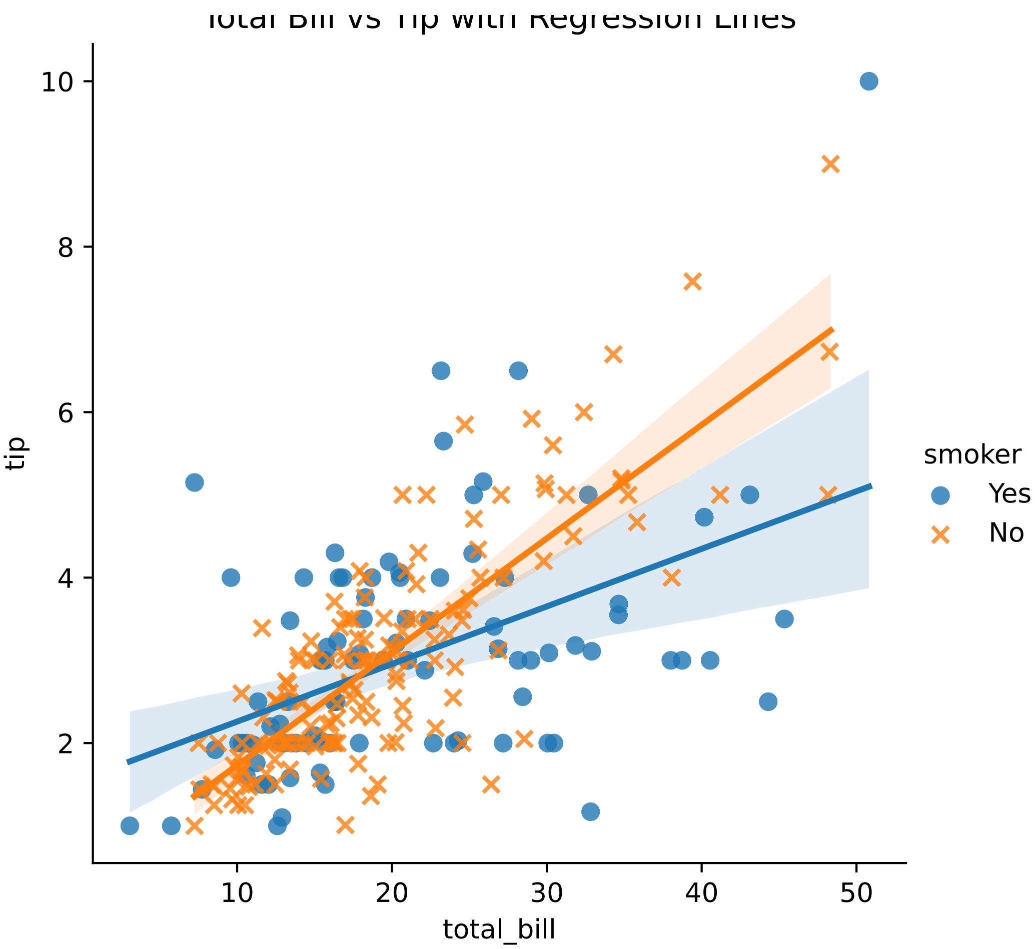
```
sns.lmplot(x='total_bill', y='tip', data=df, hue='smoker', markers=['o',
 ↪  'x'])
```

```
<seaborn.axisgrid.FacetGrid object at 0x16c8e1a90>
```

```
plt.title('Total Bill vs Tip with Regression Lines')
```

```
Text(0.5, 1.0, 'Total Bill vs Tip with Regression Lines')
```

```
plt.show()
```



which includes regression lines for smokers and non-smokers.

There are many more types of plots and customization options available in both Matplotlib and Seaborn. These libraries are powerful tools for data visualization in Python, and mastering them will greatly enhance your ability to communicate insights from data effectively. I recommend exploring their documentation and experimenting with different types of plots to become more familiar with their capabilities.

2.7 Working with Application Programming Interfaces (APIs)

Application Programming Interfaces (APIs) are a set of rules and protocols that allow different software applications to communicate with each other. They enable developers to access data and functionality from external services, libraries, or platforms without needing to understand the underlying code or infrastructure. Rather than downloading data files manually, APIs allow us to programmatically request and retrieve data directly from a web service.

In this section, we will have a brief look at how to use some common APIs for economic data retrieval using Python. We will cover the following:

- Banco de España's Statistics Web Service
- ECB Data Portal
- Fred API by the Federal Reserve Bank of St. Louis

These APIs provide access to a wide range of economic and financial data, including interest rates, exchange rates, inflation rates, GDP figures, and more. By using these APIs, we can automate the process of data retrieval, ensuring that we always have access to the most up-to-date information for our analyses. I highly recommend that you make use of APIs whenever possible to streamline your data collection process.

2.7.1 Banco de España's Statistics Web Service

Banco de España's Statistics Web Service provides a way to programmatically retrieve data from the Banco de España's databases including data from BIEST. Since Banco de España does not provide an official Python package to access their API, we can use the `requests` library to make HTTP requests and retrieve data in JSON (JavaScript Object Notation) format. We can then parse the JSON data and convert it into a Pandas DataFrame for further analysis.

To this end, we first import the necessary libraries

```python
import requests
import pandas as pd
```

Next, we define a class to interact with the Banco de España API[1]

[1] Note that creating the class is not strictly necessary, but it helps to organize the code.

```
class BancoDeEspanaAPI:
    def __init__(self, language='en'):
      self.language = language

    def request(self, url):
      response = requests.get(url)
      return response.json()

    def get_series(self, series, time_range='MAX'):

      # Prepare the series parameter
      if isinstance(series, list):
          series_list = ','.join(series)
      else:
          series_list = series

      # Download the data for the specified series
      url = f"https://app.bde.es/bierest/resources/srdatosapp/listaSeries?id⌋
 ↪  ioma={self.language}&series={series_list}&rango={time_range}"
      json_response = self.request(url)

      # Initialize an empty dataframe to store the results
      df = pd.DataFrame()

      # Go over each series in the response and extract the data
      for series_data in json_response:

        # Extract series name, dates, and values
        series_name = series_data['serie']
        dates = series_data['fechas']
        values = series_data['valores']

        # Add the data to the dataframe
        df[series_name] = pd.Series(data=values,
 ↪  index=pd.to_datetime(dates).date)

      # Sort the dataframe by index (date)
      df = df.sort_index()

      return df
```

We can then create an instance of the BancoDeEspanaAPI class and use its methods to retrieve data. For example, to get the latest data for a specific series, we can use the `get_series()` method

```
bde = BancoDeEspanaAPI()
df = bde.get_series(['DTNPDE2010_P0000P_PS_APU',
 ↪  'DTNSEC2010_S0000P_APU_SUMAMOVIL'])
```

Now, the requested series are in the DataFrame `df` and we can manipulate or analyze them as needed. For example, we can display the retrieved data

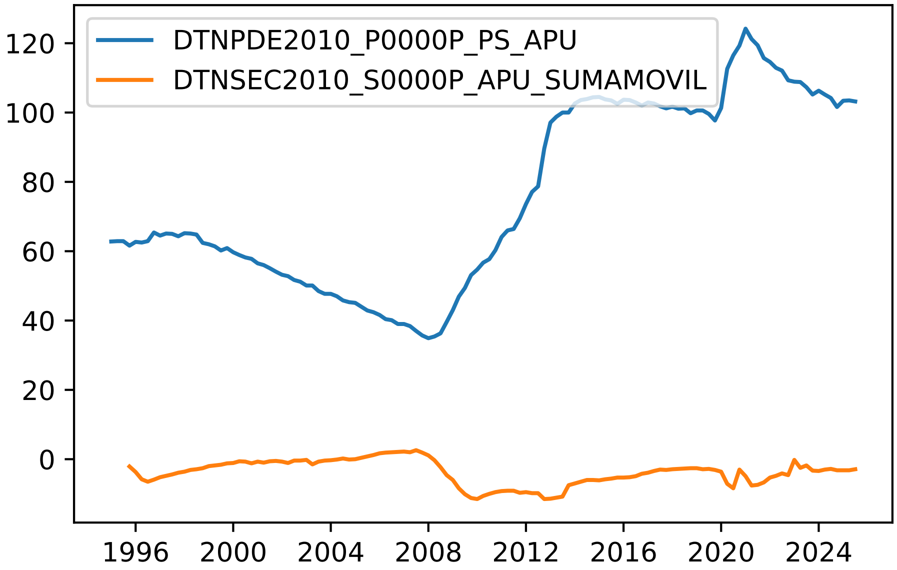
```
df.tail()
```

```
            DTNPDE2010_P0000P_PS_APU  DTNSEC2010_S0000P_APU_SUMAMOVIL
2024-07-01                     104.2                             -2.8
2024-10-01                     101.6                             -3.2
2025-01-01                     103.4                             -3.2
2025-04-01                     103.5                             -3.2
2025-07-01                     103.2                             -2.9
```

or plot it

```
df.plot()
```

```
<Axes: >
```



This is a very basic implementation of how to interact with the Banco de España API using Python. You can extend this class to include more functionality, such as handling different data formats, error handling, and more advanced data processing as needed. To get the series keys for the data you want to retrieve, you can use the BIEST tool provided by Banco de España.

2.7.2 ECB Data Portal & Other SDMX APIs

The ECB Data Portal provides access to a wide range of economic and financial data from the European Central Bank. Similar to Banco de España, the ECB does not provide an official Python package for their API. However, the ECB follows the SDMX standard for data exchange, which allows us to retrieve data in a structured format. We can use the `sdmx` library in Python to interact with the ECB API and retrieve data.

First, we import the necessary libraries

```
import sdmx
import pandas as pd
```

Then, we initialize a connection to the ECB API

```
ecb = sdmx.Client("ECB")
```

Suppose we want to retrieve the HICP inflation rate for Spain from January 2019 to June 2019. This series has the following key: `ICP.M.ES.N.000000.4.ANR`.

To download it we need to specify the appropriate parameters and make a request to the ECB API

```
key = 'M.ES.N.000000.4.ANR' # Need key without the 'ICP.' prefix
params = dict(startPeriod="2019-01", endPeriod="2019-06") # This is optional
data = ecb.data("ICP", key=key, params=params).data[0] # ICP prefix needs to
 ↪  be specified here
df = sdmx.to_pandas(data).to_frame()
```

Now, the requested data is in the DataFrame `df` and we can manipulate or analyze it as needed. For example, we can display the retrieved data

```
df.tail()
```

```
                                                                          value
FREQ REF_AREA ADJUSTMENT ICP_ITEM STS_INSTITUTION ICP_SUFFIX TIME_PERIOD
M    ES       N          000000   4               ANR        2019-02
1.1
                                                             2019-03
                                                             1.3
                                                             2019-04
                                                             1.6
                                                             2019-05
                                                             0.9
                                                             2019-06
                                                             0.6
```

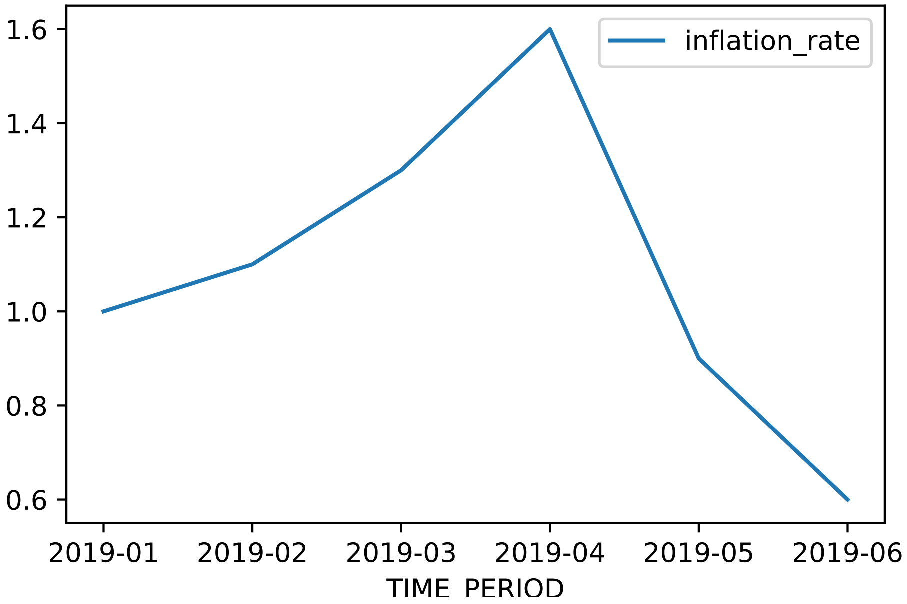
Note that this is a multi-index DataFrame. We can reset the index to make it easier to work with

```
df = df.reset_index()
df = df.set_index('TIME_PERIOD')
df = df.loc[:, ['value']]
df = df.rename(columns={'value': 'inflation_rate'})
```

We can plot the data as usual

```
df.plot()
```

```
<Axes: xlabel='TIME_PERIOD'>
```



These are just basic examples of how to interact with the ECB API using Python. The `sdmx` library supports many more features.

Other SDMX Data Providers

The SDMX standard is used by various international organizations for data exchange. Some other notable SDMX APIs include:

- Eurostat
- Bank for International Settlements (BIS)
- International Monetary Fund (IMF)
- OECD

You can find a list of SDMX data providers implemented in the `sdmx` package here. To use them in the code above you simply need to replace `'ECB'` with the appropriate provider name.

2.7.3 Fred API

The Fred API by the Federal Reserve Bank of St. Louis provides access to a vast amount of economic data, including interest rates, inflation rates, GDP figures, and more. To use the Fred API, we need to sign up for an API key on the Fred website. Once we have the API key, we can use the `pyfredapi` library in Python to interact with the Fred API and retrieve data.

The Fred API works a little differently from the previous two APIs we have seen since it requires an API key for authentication. You can sign up for a free API key on the Fred website. Note that these keys are personal and should not be shared publicly. For this reason, the key is not included directly in the code examples below. Instead, you should follow the instructions in the `pyfredapi` documentation to set up your API key securely.

Once we have set the API key, we import the necessary libraries

```
import pyfredapi as pf
```

Then, we can download the series for GDP (series ID: `GDP`) as follows

```
df = pf.get_series('GDP') # Note that you can provide the API key manually by
 ↪  adding the parameter api_key='YOUR_API_KEY' if you have not set it up as
 ↪  an environment variable
```

We can then display the retrieved data

```
df.tail()
```

```
    realtime_start realtime_end       date      value
314     2025-12-23   2025-12-23 2024-07-01  29511.664
315     2025-12-23   2025-12-23 2024-10-01  29825.182
316     2025-12-23   2025-12-23 2025-01-01  30042.113
317     2025-12-23   2025-12-23 2025-04-01  30485.729
318     2025-12-23   2025-12-23 2025-07-01  31095.089
```

Cleaning up the DataFrame a bit

```
df = df.rename(columns={'value': 'gdp'}) # Rename the 'value' column to 'gdp'
df['date'] = pd.to_datetime(df['date']) # Convert the 'date' column to
 ↪  datetime format
df = df.set_index('date') # Set the 'date' column as the index
df = df.loc[:, ['gdp']] # Keep only the 'gdp' column
```

Now it looks better

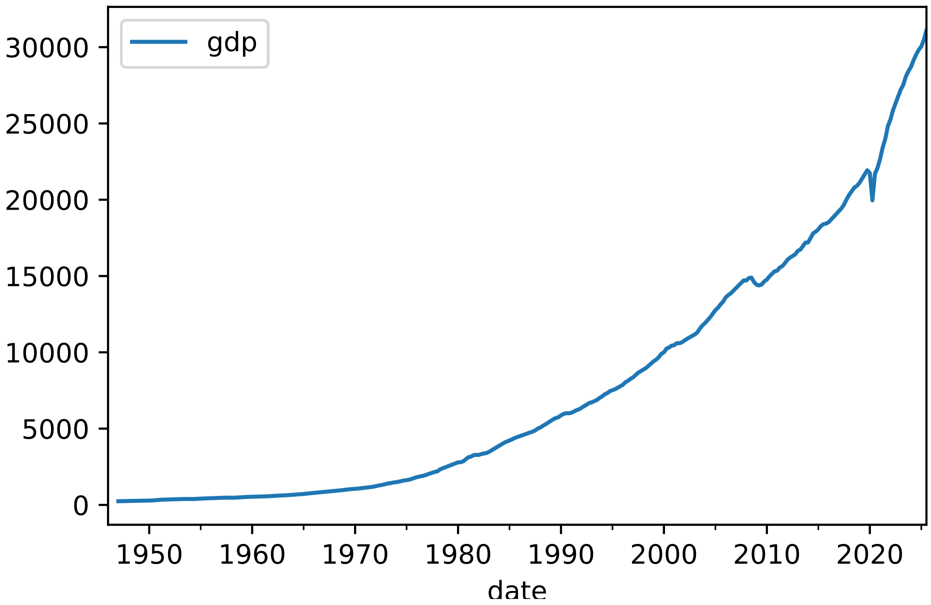
```
df.tail()
```

```
                    gdp
date
2024-07-01  29511.664
2024-10-01  29825.182
2025-01-01  30042.113
2025-04-01  30485.729
2025-07-01  31095.089
```

and to plot it, we can simply do

```
df.plot()
```

```
<Axes: xlabel='date'>
```



To see all the functionality provided by the `pyfredapi` library, please refer to the official documentation.

2.8 Good Practices

As you develop your Python programming skills, adopting good practices early will save you countless hours of frustration and make your code more maintainable, reproducible, and professional. This section covers essential practices that every Python programmer should follow, with particular emphasis on version control and virtual environments—two foundational tools that are often overlooked by beginners but are indispensable in professional settings. Due to their importance, we will briefly cover them here. However, we do not have the time to go into great detail in this course. Therefore, I encourage you to explore these topics further on your own.

2.8.1 Version Control with Git

Version control is perhaps the single most important practice for any programmer. It allows you to track changes to your code over time, collaborate with others, recover from mistakes, and maintain a complete history of your project's evolution. Git is the dominant version control system used in both academia and industry, and GitHub is the most popular platform for hosting Git repositories.

Think of Git as a sophisticated "undo" system for your entire project. Every time you make a commit, you create a snapshot of your project that you can return to at any time. This means you can experiment fearlessly—if your new approach doesn't work, you can simply revert to a previous state. Beyond this safety net, Git enables powerful collaboration workflows: multiple people can work on the same codebase simultaneously, with Git helping to merge their changes intelligently.

For academic research and data science projects, version control is equally crucial. It provides a complete audit trail of your analysis, which is essential for reproducibility. When someone asks about a result from six months ago, you can check out the exact code that produced it. When you discover an error, you can trace back to when it was introduced.

To get started with Git for your Python projects, you'll want to follow a basic workflow. First, initialize a Git repository in your project folder using `git init`. As you work, periodically stage your changes with `git add` and commit them with meaningful messages using `git commit -m "Description of changes"`. Push your commits to a remote repository on GitHub to back up your work and enable collaboration. A typical Git workflow looks like this:

```
# Initialize a new Git repository
git init

# Add files to staging area
git add script.py
```

```
# Commit changes with a descriptive message
git commit -m "Add data preprocessing function"

# Push to GitHub (after setting up remote)
git push origin main
```

These commands are meant to be run in your terminal or command prompt within your project directory. There are many graphical user interfaces (GUIs) and IDE integrations (like in VSCode) that can simplify these tasks if you prefer not to use the command line.

Some best practices for using Git include committing frequently with small, logical changes rather than massive commits that touch many files; writing clear commit messages that explain *why* you made the change, not just *what* changed; and using a `.gitignore` file to exclude data files, output files, and environment-specific files from version control. You should version control your code and configuration files, but avoid committing large datasets, model weights, or generated outputs—these should be stored separately or regenerated from your code.

Many cloud platforms like GitHub offer additional features beyond basic version control. Issues help track bugs and feature requests, pull requests facilitate code review before merging changes, and GitHub Actions can automate testing and deployment.

2.8.2 Virtual Environments and Package Management

Virtual environments are isolated Python installations that allow you to maintain different sets of packages for different projects. This solves a critical problem: different projects often require different versions of the same library. Without virtual environments, you'd be forced to use a single global installation of each package, which can lead to version conflicts and "it works on my machine" problems.

Consider a practical scenario: you're working on an older data analysis project that requires NumPy 1.20, but a new machine learning project needs NumPy 1.24 for compatibility with the latest PyTorch. Without virtual environments, you'd have to constantly uninstall and reinstall NumPy depending on which project you're working on. Virtual environments solve this elegantly by creating separate Python installations for each project, each with its own package versions.

Beyond avoiding conflicts, virtual environments make your projects reproducible. When you share your code with others or run it on a different machine, you need a way to specify exactly which package versions it requires. By creating an environment file (like `environment.yml` for conda or `requirements.txt` for pip), you provide a recipe that others can use to recreate your exact setup. This is essential for reproducible research and collaborative projects.

There are several different tools for managing virtual environments in Python. Two commonly used ones are conda and venv. Conda, which comes with Anaconda and Miniconda, is particularly popular in data science because it can manage both Python packages and system-level dependencies. It's especially useful when you need packages that require compiled code, like NumPy or PyTorch. The built-in venv module creates lighter-weight environments but only manages Python packages, requiring you to handle system dependencies separately.

In this course, we used conda to manage virtual environments. To create a virtual environment for a project, you would use:

```
# Create a new environment named 'myproject' with Python 3.11
conda create -n myproject python=3.11

# Activate the environment
conda activate myproject

# Install packages in the activated environment
conda install numpy pandas matplotlib

# Export environment to a file for reproducibility
conda env export > environment.yml

# Create environment from file on another machine
conda env create -f environment.yml
```

Once you've activated an environment, any packages you install or Python scripts you run will use that environment's isolated installation. When you're done working, you can deactivate it with `conda deactivate`. This workflow keeps each project's dependencies cleanly separated.

A good practice is to create a fresh virtual environment at the start of each new project and document its dependencies in an `environment.yml` file. Keep this file in your Git repository so others can recreate your setup. Update the file whenever you add new packages to your project. When sharing your code, include instructions for setting up the environment—this is often just a single command: `conda env create -f environment.yml`.

The combination of Git and virtual environments forms a foundation for reproducible computational work. Git tracks your code changes, while virtual environments ensure your code runs consistently across different machines and over time. Together, they transform ad-hoc scripts into professional, maintainable projects that you and others can build upon.

2.8.3 Code Organization and Documentation

Well-organized and documented code is easier to understand, maintain, and debug. As your projects grow beyond simple scripts, good organization becomes essential. Break your code into logical functions and modules rather than writing everything in a single long script. Each function should do one thing well and have a clear, descriptive name. Use docstrings to document what each function does, what parameters it expects, and what it returns.

Python docstrings are enclosed in triple quotes and appear immediately after a function definition. A good docstring explains the purpose of the function, describes parameters and return values, and may include usage examples. Here's a well-documented function:

```python
def calculate_portfolio_return(weights, returns):
    """
    Calculate the expected return of a portfolio.

    Parameters
    ----------
    weights : array-like
        Portfolio weights for each asset (should sum to 1)
    returns : array-like
        Expected returns for each asset

    Returns
    -------
    float
        Expected portfolio return

    Examples
    --------
    >>> weights = np.array([0.6, 0.4])
    >>> returns = np.array([0.10, 0.15])
    >>> calculate_portfolio_return(weights, returns)
    0.12
    """
    return np.dot(weights, returns)
```

For larger projects, organize your code into modules (separate `.py` files) grouped by functionality. Use meaningful file and variable names—`data_preprocessing.py` is much clearer than `utils.py`, and `interest_rate` is better than `x`. Follow the PEP 8 style guide for Python code, which covers naming conventions, indentation, and other formatting guidelines.

2.8.4 Error Handling and Debugging

Errors are an inevitable part of programming. Learning to handle them gracefully and debug effectively will make you a much more productive programmer. Python uses exceptions to signal errors. Rather than letting your program crash, you can catch exceptions and handle them appropriately using try-except blocks:

```
try:
    df = pd.read_csv('data.csv')
except FileNotFoundError:
    print("Error: data.csv not found. Please check the file path.")
    df = None
```

When debugging, you can use print statements strategically to understand what's happening in your code, use Python's built-in debugger (pdb) or VSCode's debugging features for more complex issues. The VSCode debugger lets you set breakpoints, step through code line by line, and inspect variable values—invaluable for tracking down subtle bugs.

Don't be discouraged when you encounter errors. Reading error messages carefully is a crucial skill—Python's error messages usually tell you exactly what went wrong and where. The traceback shows the sequence of function calls that led to the error, with the actual error at the bottom. Learning to parse these messages will help you fix issues quickly.

2.8.5 Using AI Tools for Coding

Modern AI tools like GitHub Copilot and Claude Code can significantly accelerate your coding, especially when you're learning. These tools can help you write boilerplate code, explain unfamiliar syntax, suggest solutions to common problems, and even debug errors. However, use them thoughtfully—treat them as helpful assistants, not replacements for understanding.

When using AI coding assistants, always read and understand the suggested code before using it. Don't blindly copy-paste without comprehension. These tools can make mistakes or suggest suboptimal solutions, so critical evaluation is essential. Use them to learn: if an AI suggests an unfamiliar approach, research why it works and when it's appropriate. Over time, you'll develop intuition for when AI suggestions are helpful versus when you need to think more carefully about the problem.

AI tools are particularly useful for learning new libraries or APIs, generating test cases, refactoring code, and getting past "blank page" syndrome when starting a new function. They're less reliable for complex algorithmic problems or domain-specific logic that requires deep understanding. Like any tool, they become more valuable as you learn to use them effectively.

Part II

Supervised Learning

Chapter 3

Basic Concepts in Supervised Learning

We have seen in the previous chapter that supervised learning is a type of machine learning where the model is trained on a **labeled dataset**, meaning that each input data point x has a corresponding output label y. The goal of supervised learning is to **learn a mapping from inputs to outputs** that can be used to **make predictions on new, unseen data**. We can distinguish between two main types of supervised learning tasks:

- **Regression** tasks, where the output variable y is continuous (e.g., predicting house prices, stock prices, etc.).
- **Classification** tasks, where the output variable y is categorical (e.g., spam detection, image recognition, etc.).

Depending on the type of task, different models and algorithms are used in supervised learning. This section provides an overview of some common supervised learning models and the key concepts involved in evaluating and improving their performance.

A key element of supervised learning is the **training process**, where the model learns from the labeled data by adjusting its parameters to **minimize a loss function**. The loss function quantifies the difference between the predicted outputs and the true labels, guiding the model to improve its predictions over time. This process is typically done using optimization algorithms such as gradient descent. In some special cases, closed-form solutions exist (e.g., linear regression), but **in most cases, numerical optimization methods are required**. Furthermore, note that the loss function used for regression tasks is typically different from the one used for classification tasks but the overall goal remains the same: to minimize the loss and improve the model's predictive accuracy.

We will start this section with placing two supervised learning models with which you are already familiar, **linear regression and logistic regression**, in a **machine-learning context**. We then have a brief look at another important supervised learning model, **k-nearest neighbors**. Later, we will discuss **how to evaluate regression and classification models** and introduce the concepts of **generalization and overfitting**. Finally, we will implement some of these concepts in Python.

3.1 Basic Supervised Learning Models

In this subsection, we will first quickly **review linear regression and logistic regression** in a machine-learning context. Since you are already familiar with these methods, we will use them to introduce some new concepts such as the **decision boundary** and **feature engineering**. Then, we will briefly discuss the **k-nearest neighbors** algorithm, which is a simple yet powerful supervised learning model that can be used for both regression and classification tasks.

3.1.1 Linear Regression in a ML Context

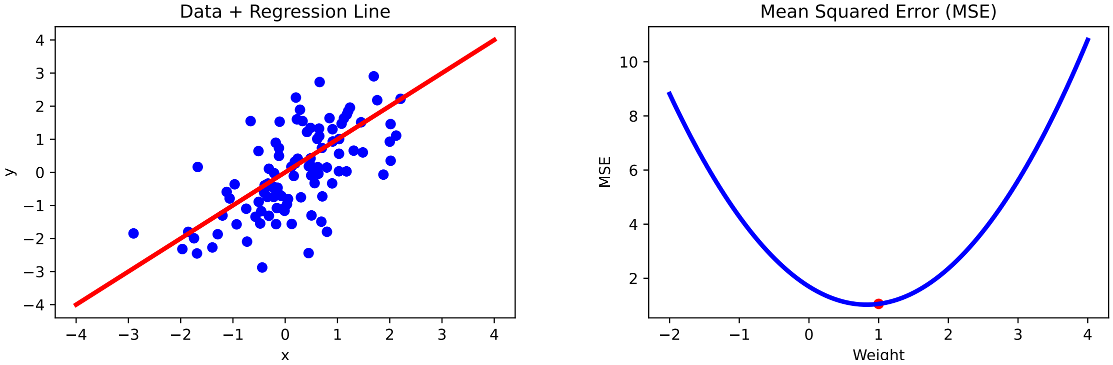


Figure 3.1: Linear Regression With a Single Feature x (i.e., $m = 1$) and Bias $b = 0$ (Note that this Plot is interactive in the HTML version)

You have already extensively studied linear regressions in courses of the program, so we will not discuss it in much detail. In machine learning, it is common to talk about **weights** w_i and **biases** b_i instead of coefficients β_i and intercept β_0, i.e., the linear regression model would be written as

$$y_n = b + \sum_{i=1}^{m} w_i x_{i,n} + \varepsilon_n \qquad n = 1, \dots, N$$

where w_i are the weights, b is the bias and N is the sample size. The weights and biases are found by **minimizing the empirical risk function or mean squared error (MSE) loss**, which is a measure of how well the model fits the data.

$$\text{MSE}(y, x; w, b) = \frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2$$

where y_n is the true value, $\hat{y}_n$ is the predicted (or fitted) value for observation n.

In the case of linear regression, there is a closed-form solution for the weights and biases that minimize the MSE. However, the **weights and biases have to be found numerically in many other machine learning models** since there is no closed-form solution. One can think of this, as the machine learning algorithm automatically moving a slider for the slope Figure 3.1 until the loss is minimized (i.e., the red dot is at the lowest possible point) and the model fits the data as well as possible.

3.1.2 Logistic Regression in a ML Context

Logistic regression is a widely used **classification model** $p(y|x; w, b)$ where $x \in \mathbb{R}^m$ is an input vector, and $y \in \{0, 1, \dots, C\}$ is a class label. We will focus on the binary case, meaning that $y \in \{0, 1\}$ but it is also possible to extend this to more than two classes. The probability that y_n is equal to 1 for observation n is given by

$$p(y_n = 1|x_n; w, b) = \frac{1}{1 + \exp(-b - \sum_{i=1}^{m} w_i x_{i,n})}$$

where $w = [w_1, \dots, w_m]' \in \mathbb{R}^m$ is a weight vector, and b is a bias term. Combining the probabilities for each observation n, we can write the **likelihood function** as

$$\mathcal{L}(w, b) = \prod_{n=1}^{N} p(y_n = 1|x_n; w, b)^{y_n} \left(1 - p(y_n = 1|x_n; w, b)\right)^{1-y_n}$$

or taking the natural logarithm of the likelihood function, we get the **log-likelihood function**

$$\log \mathcal{L}(w, b) = \sum_{n=1}^{N} y_n \log p(y_n = 1|x_n; w, b) + (1 - y_n) \log\left(1 - p(y_n = 1|x_n; w, b)\right).$$

To find the weights and biases, we need to numerically maximize the log-likelihood function (or minimize $-\log \mathcal{L}(w, b)$).

Adding a **classification threshold** t to a logistic regression yields a **decision rule** of the form

$$\hat{y} = 1 \Leftrightarrow p(y = 1|x; w, b) > t,$$

i.e., the model predicts that $y = 1$ if $p(y = 1|x; w, b) > t$.

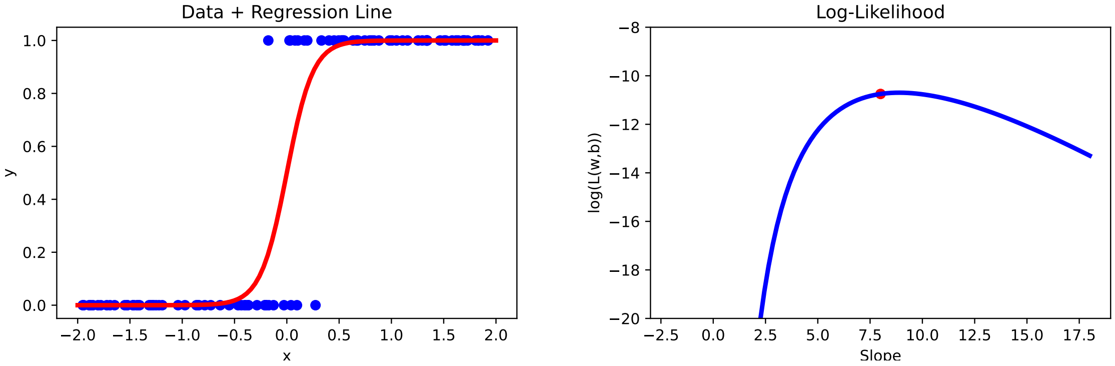


Figure 3.2: Logistic Regression With a Single Feature x (i.e., $m = 1$) and Bias $b = 0$ (Note that this Plot is interactive in the HTML version)

⚠ Terminology: Regression vs. Classification

Do not get confused about the fact that it is called logistic *regression* but is used for classification tasks. Logistic regression provides an estimate of the probability that $y = 1$ for given x, i.e., an estimate for $p(y = 1|x; w, b)$. To turn, this into a classification model, we also need a **classification threshold** value for $p(y = 1|x; w, b)$ above which we classify an observation as $y = 1$.

Figure 3.2 shows an interactive example of a logistic regression model. The left-hand side shows the data points and the regression line. The right-hand side shows the log-likelihood function with the red dot showing the value of the log-likelihood for the current value of w. The goal is to find the weight w in the regression line that maximizes the log-likelihood function (we assumed $b = 0$ for simplicity).

If you enable the classification threshold t, a data point is shown as dark blue if $p(y = 1|x; w, b) > t$, otherwise, it is shown in light blue. Note how the value of the threshold affects the classification of the data points for points in the middle. Essentially, for each classification threshold, we have a different classification model. But how do we choose the classification threshold? This is a topic that we will discuss in the next section.

Logistic regression belongs to the class of **generalized linear models** with logit as the link function. We could write

$$\log\left(\frac{p}{1-p}\right) = b + \sum_{i=1}^{m} w_i x_{i,n}$$

where $p = p(y_n = 1|x_n; w, b)$, which separates the linear part on the right-hand side from the logit on the left-hand side.

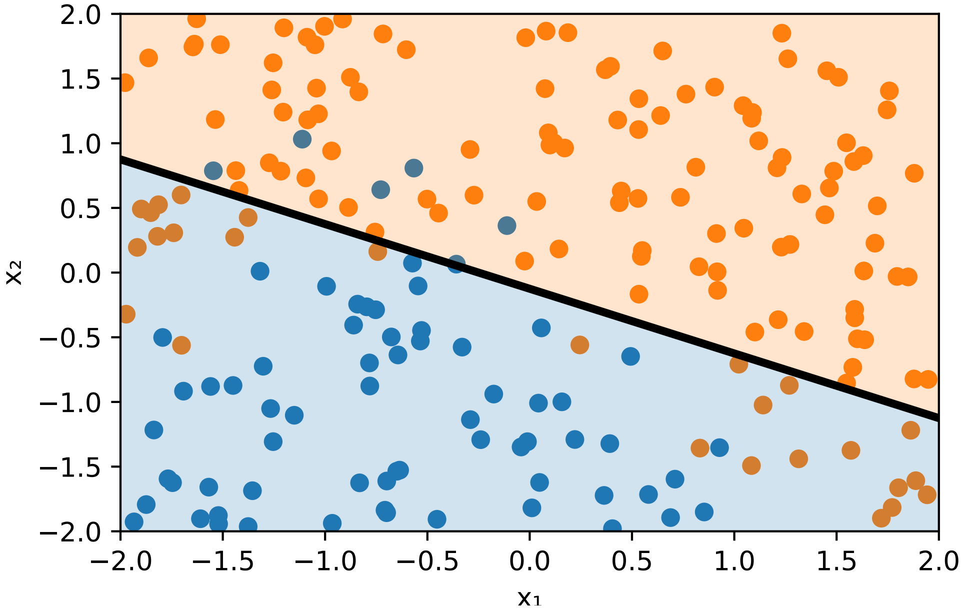


Figure 3.3: Decision Boundary - Logistic Regression with Features x_1 and x_2 (i.e., $m = 2$) (Note that this Plot is interactive in the HTML version)

This linearity also shows up in the **linear decision boundary** produced by a logistic regression in Figure 3.3. A decision boundary shows how a machine-learning model separates different classes in our data, i.e, how it would classify an arbitrary combination of (x_1, x_2). This linearity of the decision boundary can pose a problem if the two classes are not linearly separable as in Figure 3.3. We can remedy this issue by including higher order terms for x_1 and x_2 such as x_1^2 or x_2^3, which is a type of **feature engineering**. However, there are many forms of non-linearity that the decision boundary can have and we cannot try all of them. You might know the following phrase from a Tolstoy book

> "Happy families are all alike; every unhappy family is unhappy in its own way."

In the context of non-linear functions, people sometimes say

> "Linear functions are all alike; every non-linear function is non-linear in its own way."

During the course, we will learn more advanced machine-learning techniques that can produce non-linear decision boundaries without the need for feature engineering.

3.1.3 K-Nearest Neighbors

The K-Nearest Neighbors (KNN) algorithm is a simple and intuitive method for classification and regression. The KNN algorithm uses the K nearest neighbors of a data point to make a prediction. For example, in the case of a **regression** task, the prediction $\hat{y}$ for a new data point x is

$$\hat{y} = \frac{1}{K} \sum_{x_i \in N_k(x)} y_i$$

i.e., the average of the K nearest neighbors of x. In the case of a **classification** task, the prediction $\hat{y}$ is the majority class of the K nearest neighbors of x.

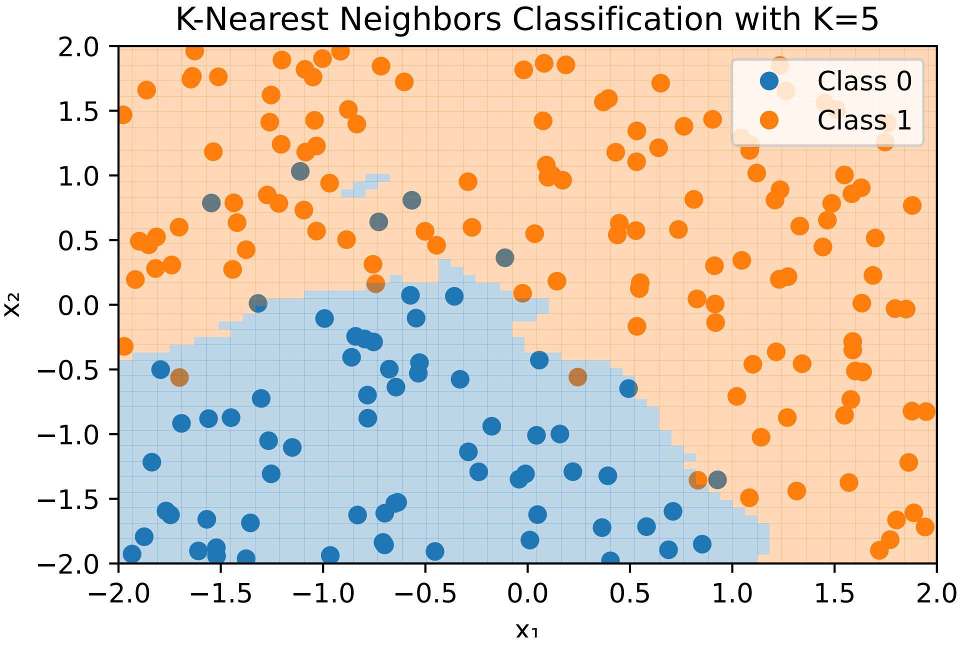


Figure 3.4: K-Nearest Neighbors Classification with $K = 5$ (Classification shown as Shaded Area)

Figure 3.4 shows an example of the K-Nearest Neighbors algorithm applied to the dataset that was used for the logistic regression example in Figure 3.2. For each point in the feature space, the KNN algorithm looks at the $K = 5$ nearest neighbors and classifies the point based on the majority class of these neighbors. Note that the color of the data point is the true value, while the shaded area shows the classification made by the KNN algorithm.

The **decision boundary is highly non-linear** and can adapt to the shape of the data. However, this flexibility comes at a cost. The KNN algorithm can be computationally expensive, especially for large datasets, since it needs to compute the distance between the new data point and all other data points in the training set. Additionally, the KNN algorithm is sensitive to the choice of K and the distance metric used to compute the nearest neighbors.

In the example, above we used the **Euclidean distance** to compute the nearest neighbors. The squared Euclidean distance between two points x and y in p-dimensional space is defined as

$$d(x_i, x_j) = \sum_{n=1}^{p}(x_{in} - x_{jn})^2 = \|x_i - x_j\|^2$$

where x_{in} and x_{jn} are the n-th feature of the i-th and j-th observation in our dataset, respectively.

Note that the Euclidean distance can be used in both regression and classification problems since it only applies to the features x and not the target variable y. However, if the features include categorical variables, we need to use a different distance metric that can accommodate such variables.

3.2 Model Evaluation

Suppose our machine learning model has learned the weights and biases that minimize the loss function. **How do we know if the model is any good?** In this section, we will discuss how to evaluate regression and classification models.

3.2.1 Regression Models

In the case of regression models, we can use the **mean squared error (MSE)** as a measure of how well the model fits the data. The MSE is defined as

$$\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2,$$

where y_n is the true value, $\hat{y}_n$ is the predicted value for observation n and N is the sample size. A low MSE indicates a good fit, while a high MSE indicates a poor fit. In the ideal case, the MSE is zero, meaning that the model perfectly fits the data. Related to the MSE is the **root mean squared error (RMSE)**, which is the square root of the MSE

$$\text{RMSE} = \sqrt{\text{MSE}}.$$

The RMSE is in the same unit as the target variable y and is easier to interpret than the MSE.

Regression models are sometimes also evaluated based on the **coefficient of determination** R^2. The R^2 is defined as

$$R^2 = 1 - \frac{\sum_{n=1}^{N}(y_n - \hat{y}_n)^2}{\sum_{n=1}^{N}(y_n - \bar{y})^2},$$

where $\bar{y}$ is the mean of the true values y_n. The R^2 is a measure of how well the model fits the data compared to a simple model that predicts the mean of the true values for all observations. The R^2 can take values between $-\infty$ and 1. A value of 1 indicates a perfect fit, while a value of 0 indicates that the model does not perform better than the simple model that predicts the mean of the true values for all observations. Note that the R^2 is a normalized version of the MSE

$$R^2 = 1 - \frac{N \times \text{MSE}}{\sum_{n=1}^{N}(y_n - \bar{y})^2}.$$

Thus, we would rank models based on the R^2 in the same way as we would rank them based on the MSE or the RMSE.

There are many more metrics but at this stage, we will only look at one more: the **mean-absolute-error (MAE)**. The MAE is defined as

$$\text{MAE} = \frac{1}{N}\sum_{n=1}^{N} |y_n - \hat{y}_n|.$$

The MAE is the average of the absolute differences between the true values and the predicted values. Note that the MAE does not penalize large errors as much as the MSE does.

3.2.2 Classification Models

In the case of classification models, we need different metrics to evaluate the performance of the model. We will discuss some of the most common metrics in the following subsections.

Basic Metrics

A key measure to evaluate a classification model, both binary and multiclass classification, is to look at how often it predicts the correct class. This is called the **accuracy** of a model

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}.$$

Related to this, one could also compute the **misclassification rate**

$$\text{Misclassification Rate} = \frac{\text{Number of incorrect predictions}}{\text{Total number of predictions}}.$$

While these measures are probably the most intuitive measures to assess the performance of a classification model, they **can be misleading** in some cases. For example, if we have a dataset with 95% of the observations in class 1 and 5% in class 0, a model that always predicts $y = 1$ (class 1) would have an accuracy of 95%. However, this model would not be very useful.

Confusion Matrices

In this and the following subsection, we focus on binary classification problems.

Let $\hat{y}$ denote the predicted class and y the true class. In a binary classification problem, we can make **two types of errors**. First, we can make an error because we predicted $\hat{y} = 1$ when $y = 0$, which is called a **false positive** (or a **"false alarm"**). Sometimes this is also called a **type I error**. Second, we can make an error because we predicted $\hat{y} = 0$ when $y = 1$, which is called a **false negative** (or a **"missed detection"**). Sometimes this is referred to as a **type II error**.

We can summarize the predictions of a classification model in a **confusion matrix** as seen in Figure 3.5. The confusion matrix is a 2×2 matrix that shows the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) of a binary classification model.

There is a **tradeoff** between the two types of errors. For example, you *could get fewer false negatives by predicting $\hat{y} = 1$ more often, but this would increase the number of false positives.* In the extreme case, if you only predict $\hat{y} = 1$ for all observations, you would have no false negatives at all. However, you would also have no true negatives making the model of questionable usefulness.

⚠ Confusion Matrix: Dependence on Classification Threshold t

The number of true positives, true negatives, false positives, and false negatives in the confusion matrix depends on the classification threshold t.

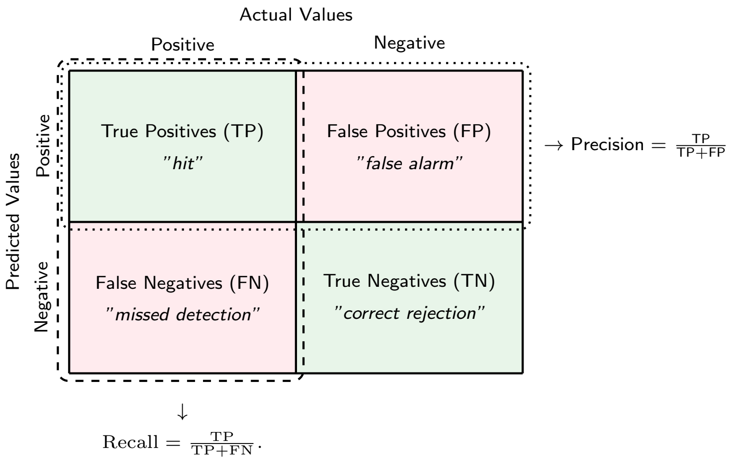


Figure 3.5: Confusion matrix

Note that we can compute the **accuracy** measure as a function of true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN)

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}},$$

while the **misclassification rate** is given by

$$\text{Misclassification Rate} = \frac{\text{FP} + \text{FN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}.$$

Another useful measure that can be derived from the confusion matrix is the **precision**. It measures the fraction of positive predictions that were actually correct, i.e.,

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

The **true positive rate (TPR)** or **recall** or **sensitivity** measures the fraction of actual positives that were correctly predicted, i.e.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

Analogously, **true negative rate (TNR)** or **specificity** measures the fraction of actual negatives that were correctly predicted, i.e.,

$$\text{TNR} = \frac{\text{TN}}{\text{FP} + \text{TN}}$$

Finally, the **false positive rate (FPR)** measures the fraction of actual negatives that were incorrectly predicted to be positive, i.e.,

$$\mathrm{FPR} = 1 - \mathrm{TNR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

Note that all of these measures can be computed for a given classification threshold t. They capture different aspects of the quality of the predictions of a classification model.

> ℹ Multiclass Classification
>
> In the case of multiclass classification, the confusion matrix is a $K \times K$ matrix, where K is the number of classes. The diagonal elements of the confusion matrix represent the number of correct predictions for each class, while the off-diagonal elements represent the number of incorrect predictions.
> Note that we can binarize multiclass classification problems, which allows us to use the same metrics as in binary classification. Two such binarization schemes are
>
> - **One-vs-Rest** (or **One-vs-All**): In this scheme, we train K binary classifiers, one for each class to distinguish it from all other classes. We can then use the class with the highest score as the predicted class for a new observation.
> - **One-vs-One**: In this scheme, we train $K(K-1)/2$ binary classifiers, one for each pair of classes. We can then use a majority vote to determine the class of a new observation.

Receiver Operating Characteristic (ROC) Curves and Area Under the Curve (AUC)

Figure 3.6 shows a **Receiver Operating Characteristic (ROC) curve** which is a graphical representation of the tradeoff between the true positive rate (TPR) and the false positive rate (FPR) for different classification thresholds. The ROC curve is a useful tool to visualize the performance of a classification model. The diagonal line in the ROC curve represents a random classifier. A classifier that is better than random will have a ROC curve above the diagonal line. The closer the ROC curve is to the top-left corner, the better the classifier.

The **Area Under the Curve (AUC)** of the ROC curve is a measure to compare different classification models. The AUC is a value between 0 and 1, where a value of 1 indicates a perfect classifier and a value of 0.5 indicates a random classifier. Figure 3.7 shows the AUC of a classifier as the shaded area under the ROC curve. Note that the AUC summarizes the ROC curve, which itself represents the quality of predictions of our classification model at different thresholds, in a single number.

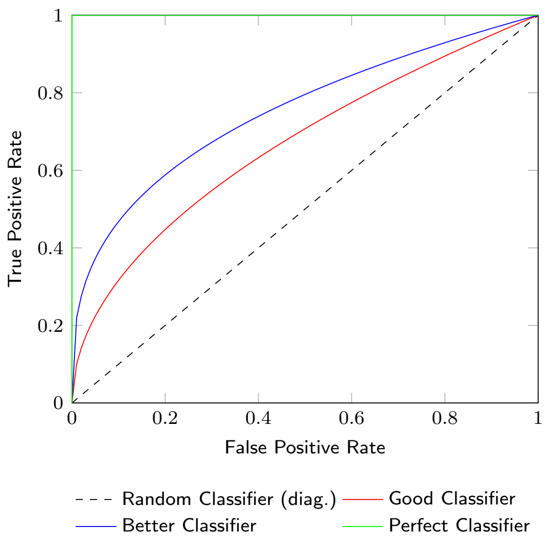


Figure 3.6: Receiver Operating Characteristic (ROC) Curve

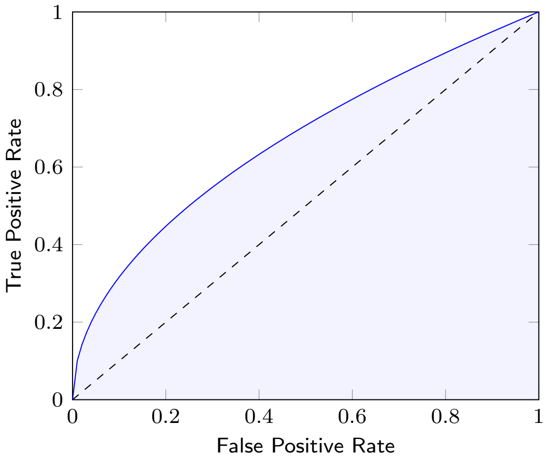


Figure 3.7: Area Under the Curve (AUC)

3.3 Generalization and Overfitting

Typically, we are not just interested in having a good fit for the dataset on which we are training a classification (or regression) model, after all, we already have the actual classes or realization of predicted variables in our dataset. What **we are really interested in is that a classification or regression model generalizes to new data**.

However, since the models that we are using are highly flexible, it can be the case that we have a very high accuracy during the training of our model but it does not provide good predictions when used on new data. This situation is called **overfitting**: we have a very good fit in our training dataset, but predictions for new data inputs are bad.

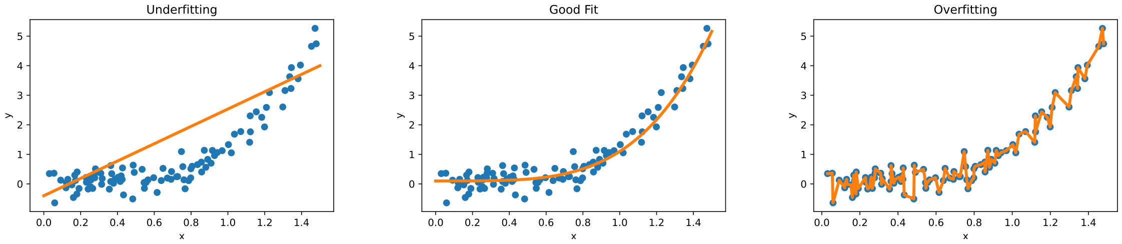


Figure 3.8: Examples of Overfitting and Underfitting

Figure 3.8 provides examples of overfitting and underfitting. The blue dots represent the training data x and y, the orange curve represents the fit of the model to the training data. The left plot shows an example of **underfitting**: the model is too simple to capture the underlying structure of the data. The middle plot shows a "good fit": the model captures the underlying structure of the data. The right plot shows an example of **overfitting**: the model is too complex and captures the noise in the data.

The interactive overfitting figure is not available in the PDF version of this book. Please view the HTML version to interact with the figure.

Figure 3.9

Figure 3.9 shows an example where the true data generating process (DGP) is a quadratic function but there is some added noise when we sample from the DGP. The blue dots represent the training data, the orange curve represents the fitted polynomial of order k. For example, if $k = 3$, we run a regression of y on a constant, x, x^2, and x^3. The left panel shows the training data and the fitted polynomial, the middle panel shows the training RMSE as a function of the polynomial order k, and the right panel shows the test RMSE as a function of the polynomial order k. As we increase the polynomial order (a measure of complexity in our model), the training RMSE decreases (we use more and more flexible models to fit the data), but the test RMSE eventually increases, indicating overfitting.

3.3.1 Bias-Variance Tradeoff

The concepts of **bias** and **variance** are useful to understand the tradeoff between underfitting and overfitting. Suppose that data is generated from the true model $Y = f(X) + \epsilon$, where ϵ is a random error term such that $\mathbb{E}[\epsilon] = 0$ and $\mathrm{Var}[\epsilon] = \sigma^2$. Let $\hat{f}(x)$ be the prediction of the model at x. One can show that the expected prediction error (or generalization error) of a model can be decomposed into three parts

$$\mathrm{EPE}(x_0) = \mathbb{E}[(Y - \hat{f}(x_0))^2 | X = x_0] = \mathrm{Bias}^2(\hat{f}(x_0)) + \mathrm{Var}(\hat{f}(x_0)) + \sigma^2,$$

where $\mathrm{Bias}(\hat{f}(x_0)) = \mathbb{E}[\hat{f}(x_0)] - f(x_0)$ is the bias at x_0, $\mathrm{Var}(\hat{f}(x_0)) = \mathbb{E}[\hat{f}(x_0)^2] - \mathbb{E}[\hat{f}(x_0)]^2$ is the variance at x_0, and σ^2 is the irreducible error, i.e., the error that cannot be reduced by any model. As model complexity increases, the bias tends to decrease, but the variance tends to increase. The following quote from Cornell lecture notes summarizes the bias-variance tradeoff well:

> Variance: Captures how much your classifier changes if you train on a different training set. How "over-specialized" is your classifier to a particular training set (overfitting)? If we have the best possible model for our training data, how far off are we from the average classifier?
>
> Bias: What is the inherent error that you obtain from your classifier even with infinite training data? This is due to your classifier being "biased" to a particular kind of solution (e.g. linear classifier). In other words, bias is inherent to your model.
>
> Noise: How big is the data-intrinsic noise? This error measures ambiguity due to your data distribution and feature representation. You can never beat this, it is an aspect of the data.

Figure 3.10 shows the relationship between the model complexity and the prediction error. A more complex model can reduce the prediction error only up to a certain point. After this point, the model starts to overfit the training data (it learns noise in the data), and the prediction error for the test data (i.e., data not used for model training) increases. Ideally, we would like to find the model complexity that minimizes the prediction error for the test data. We have seen this behavior in Figure 3.9 where the test RMSE first decreases and then increases as we increase the polynomial order k.

3.3.2 Regularization

One approach to avoid overfitting is to use **regularization**. Regularization adds a penalty term to the loss function that penalizes large weights. The most common regularization techniques are **L1 regularization** and **L2 regularization**. L1 regularization adds the sum of the absolute

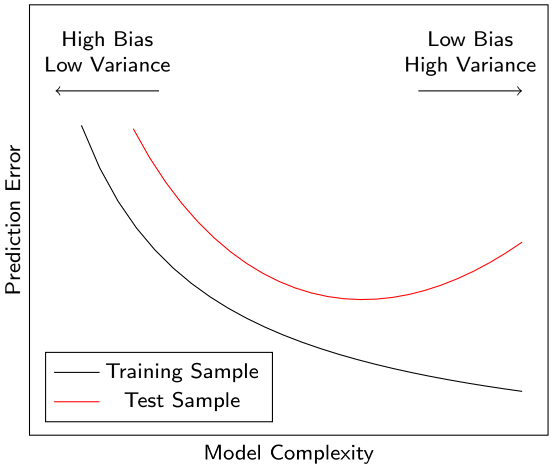


Figure 3.10: Model Complexity and Generalization Error (adapted from Hastie, Tibshirani, and Friedman 2009)

values of the weights to the loss function, while L2 regularization adds the sum of the squared weights to the loss function.

These techniques are applicable across a large range of ML models and depending on the type of model additional regularization techniques might be available. For example, in neural networks, **dropout regularization** is a common regularization technique that randomly removes a set of artificial neurons during training.

In the context of linear regressions, L1 regularization is also called **LASSO regression**. The loss function of LASSO regression is given by

$$\text{Loss} = \text{MSE}(y, x; w) + \lambda \sum_{i=1}^{m} |w_i|,$$

where $\text{MSE}(y, x; w)$ refers to the mean squared error (the standard loss function of a linear regression), λ is a hyperparameter that controls the strength of the regularization. Note that LASSO regression can also be used for **feature selection**, as it tends to set the weights of irrelevant features to zero. Figure 3.11 shows the LASSO regression loss for different levels of λ.

An L2 regularization in a linear regression context is called a **Ridge regression**. Its loss function is given by

$$\text{Loss} = \text{MSE}(y, x; w) + \lambda \sum_{i=1}^{m} w_i^2.$$

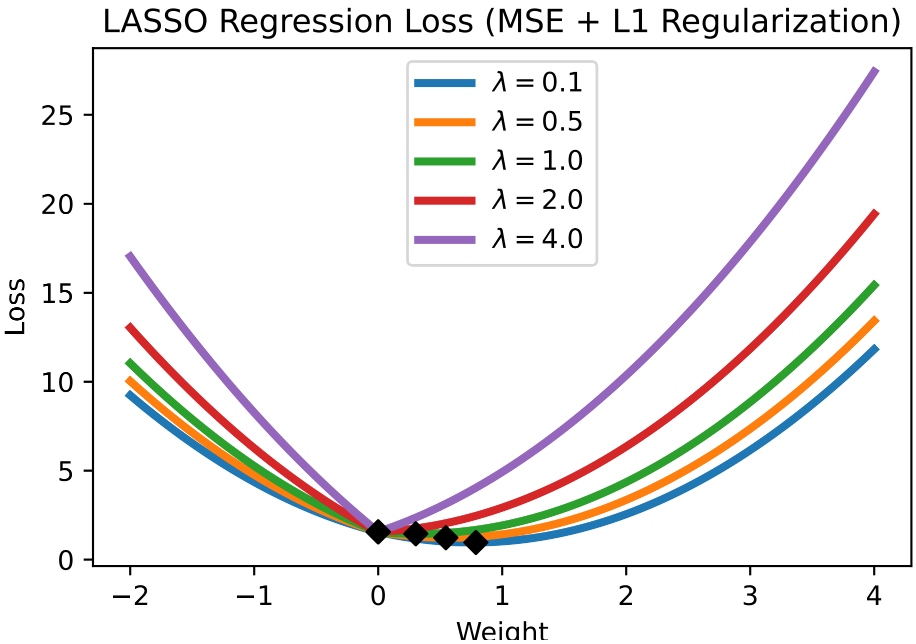


Figure 3.11: LASSO Regression Loss for Different Levels of λ

We will have a closer look at regularization in the application sections. For now, it is important to understand that regularization works by constraining the weights of the model (i.e., keeping the weights small), which can help to avoid overfitting (which might require some weights to be very large). Figure 3.12 shows the Ridge regression loss for different levels of λ. Note how the Ridge regression loss is smoother than the LASSO regression loss and that the weights are never set to exactly zero but just get closer and closer to zero.

3.3.3 Training, Validation, and Test Datasets

Regularization discussed in the previous section is a method to directly *prevent* overfitting. However, another approach to the issues is to **adjust our evaluation procedure** in a way that allows us *to detect overfitting*. To do this, we can split the dataset into several parts. The first option shown in Figure 3.13 is to split the dataset into a **training dataset** and a **test dataset**. The training dataset is used to train the model, while the test dataset is used to evaluate the model. Why does this help to detect overfitting? If the model performs well on the training dataset but poorly on the test dataset, this is a sign of overfitting. If the model performs well on the test dataset, this is a sign that the model generalizes well to new data.

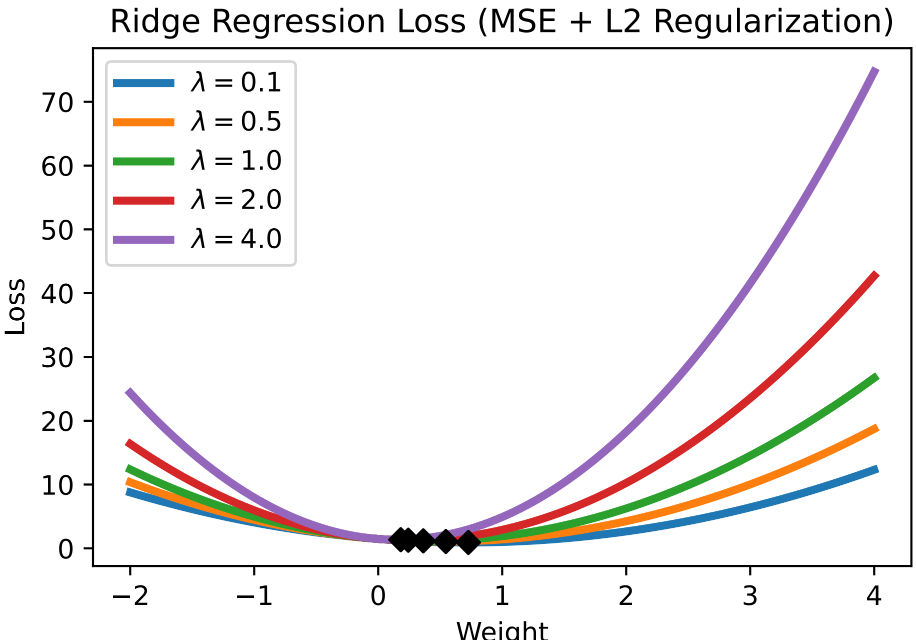


Figure 3.12: Ridge Regression Loss for Different Levels of λ

Difference with Terminology in Econometrics/Statistics

In econometrics/statistics, it is more common to talk about **in-sample** and **out-of-sample** performance. The idea is the same: the in-sample performance is the performance of the model on the training dataset, while the out-of-sample performance is the performance of the model on the test dataset.

Figure 3.13: Option A - Splitting the Whole Dataset into Training, and Test Datasets

The second option shown in Figure 3.14 is to split the dataset into a **training dataset**, a **validation dataset**, and a **test dataset**. The training dataset is used to train the model, the validation dataset is used to tune the hyperparameters of the model, and the test dataset is used to evaluate the model.

Common splits are 70% training and 30% test, or 80% training and 20% test in Option A. In Option B, a common split is 70% training, 15% validation, and 15% test.

Figure 3.14: Option B - Splitting the Whole Dataset into Training, Test, and Validation Datasets

3.3.4 Cross-Validation

Another approach to detect overfitting is to use **cross-validation**. There are different types of cross-validation but **k-fold cross-validation** is probably the most common. In k-fold cross-validation, shown in Figure 3.15, the dataset is split into k parts (called **folds**). The model is trained on $k-1$ folds and evaluated on the remaining fold. This process is repeated k times, each time using a different fold as the test fold. The performance of the model is then averaged over the k iterations. In practice, $k = 10$ is a common choice. If we set $k = N$, where N is the number of observations in the dataset, we call this **leave-one-out cross-validation** or **LOOCV**.

Figure 3.15: 5-Fold Cross-Validation

The advantage of cross-validation is that it allows us to use all the data for training and testing. The disadvantage is that it is computationally more expensive than a simple training-test split.

3.4 Python Implementation

Let's have a look at how to implement a logistic regression model in Python. First, we need to import the required packages

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, roc_auc_score, recall_score, precision_score, roc_curve
pd.set_option('display.max_columns', 50) # Display up to 50 columns
```

Let's download the dataset automatically, unzip it, and place it in a folder called `data` if you haven't done so already

```python
from io import BytesIO
from urllib.request import urlopen
from zipfile import ZipFile
import os.path

# Check if the file exists
if not os.path.isfile('data/card_transdata.csv'):

    print('Downloading dataset...')

    # Define the dataset to be downloaded
    zipurl = 'https://www.kaggle.com/api/v1/datasets/download/dhanushnarayananr/credit-card-fraud'

    # Download and unzip the dataset in the data folder
    with urlopen(zipurl) as zipresp:
        with ZipFile(BytesIO(zipresp.read())) as zfile:
            zfile.extractall('data')

    print('DONE!')

else:
    print('Dataset already downloaded!')
```

Then, we can load the data into a DataFrame using the `read_csv` function from the `pandas` library

```
df = pd.read_csv('data/card_transdata.csv')
```

Note that it is common to call this variable `df` which is short for DataFrame.

This is a **dataset of credit card transactions** from Kaggle.com. The target variable y is `fraud`, which indicates whether the transaction is fraudulent or not. The other variables are the features x of the transactions.

3.4.1 Data Exploration & Preprocessing

The first step whenever you load a new dataset is to familiarize yourself with it. You need to understand what the variables represent, what the target variable is, and what the data looks like. This is called **data exploration**. Depending on the dataset, you might need to preprocess it (e.g., check for missing values and duplicates, or create new variables) before you can use it to train a machine-learning model. This is called **data preprocessing**.

Basic Dataframe Operations

Let's see how many rows and columns the dataset has

```
df.shape
```

The dataset has 1 million rows (observations) and 8 columns (variables)! Now, let's have a look at the first few rows of the dataset with the `head()` method

```
df.head().T
```

If you would like to see more entries in the dataset, you can use the `head()` method with an argument corresponding to the number of rows, e.g.,

```
df.head(20)
```

Note that analogously you can also use the `tail()` method to see the last few rows of the dataset.

We can also check what the variables in our dataset are called

```
df.columns
```

and the data types of the variables

```
df.dtypes
```

In this case, all our variables are floating-point numbers (`float`). This means that they are numbers that have a fractional part such as 1.5, 3.14, etc. The number after `float`, `64` in this case refers to the number of bits that are used to represent this number in the computer's memory. With 64 bits you can store more decimals than you could with, for example, 32, meaning that the results of computations can be more precise. But for the topics discussed in this course, this is not very important. Other common data types that you might encounter are integers (`int`) such as 1, 3, 5, etc., or strings (`str`) such as `'hello'`, `'world'`, etc.

Let's dig deeper into the dataset and see some summary statistics

```
df.describe().T
```

With the `describe()` method we can see the count, mean, standard deviation, minimum, 25th percentile, median, 75th percentile, and maximum values of each variable in the dataset.

Checking for Missing Values and Duplicated Rows

It is also important to check for missing values and duplicated rows in the dataset. Missing values can be problematic for machine learning models, as they might not be able to handle them. Duplicated rows can also be problematic, as they might introduce bias in the model.

We can check for missing values (NA) that are encoded as None or `numpy.NaN` (Not a Number) with the `isna()` method. This method returns a boolean DataFrame (i.e., a DataFrame with `True` and `False` values) with the same shape as the original DataFrame, where `True` values indicate missing values.

```
df.isna()
```

or to make it easier to see, we can sum the number of missing values for each variable

```
df.isna().sum()
```

Luckily, there seem to be no missing values. However, you need to be careful! Sometimes missing values are encoded as empty strings `''` or `numpy.inf` (infinity), which are not considered missing values by the `isna()` method. If you suspect that this might be the case, you need to make additional checks.

As an alternative, we could also look at the `info()` method, which provides a summary of the DataFrame, including the number of non-null values in each column. If there are missing values, the number of non-null values will be less than the number of rows in the dataset.

```
df.info()
```

We can also check for duplicated rows with the `duplicated()` method.

```
df.loc[df.duplicated()]
```

Luckily, there are also no duplicated rows.

Data Visualization

Let's continue with some data visualization. We can use the `matplotlib` library to create plots. We have already imported the library at the beginning of the notebook.

Let's start by plotting the distribution of the target variable `fraud` which can only take values zero and one. We can type

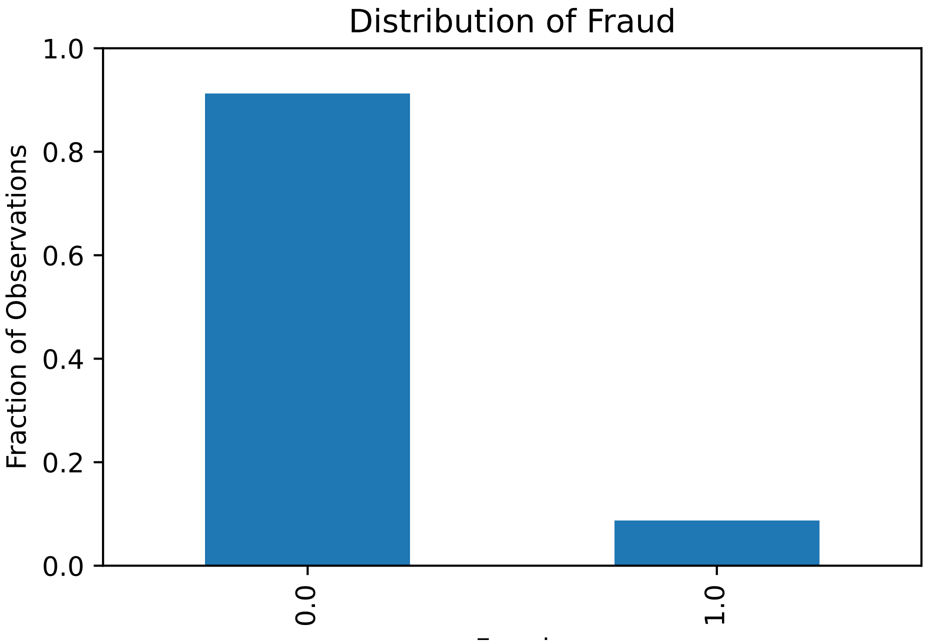
```
df['fraud'].value_counts()
```

to get the count of each value. We can also use the `normalize=True` argument to get the fraction of observations instead of the count

```
df['fraud'].value_counts(normalize=True)
```

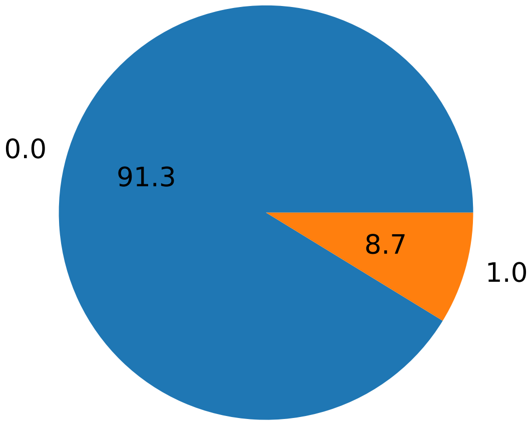
We can then plot it as follows

```
df['fraud'].value_counts(normalize=True).plot(kind='bar')
plt.xlabel('Fraud')
plt.ylabel('Fraction of Observations')
plt.title('Distribution of Fraud')
ax = plt.gca()
ax.set_ylim([0.0, 1.0])
plt.show()
```



Alternatively, we can plot it as a pie chart

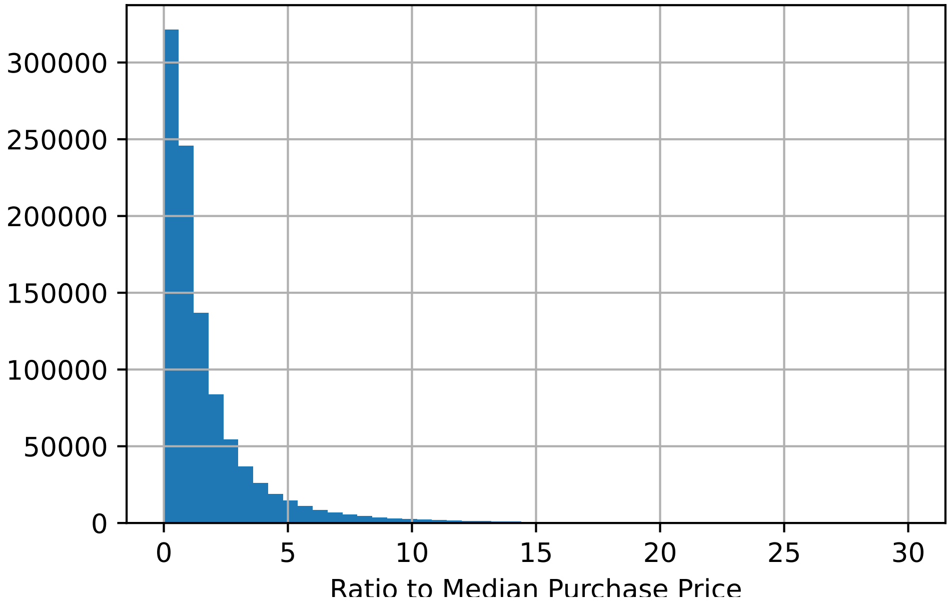
```
df.value_counts("fraud").plot.pie(autopct = "%.1f")
plt.ylabel('')
plt.show()
```



Our dataset seems to be quite imbalanced, as only 8.7% of the transactions are fraudulent. This is a common problem in fraud detection datasets, as fraudulent transactions are usually very rare. We will need to **keep this in mind** when evaluating our machine learning model: the accuracy measure will be very high even for bad models, as the model can just predict that all transactions are not fraudulent and still get an accuracy of 91.3%.

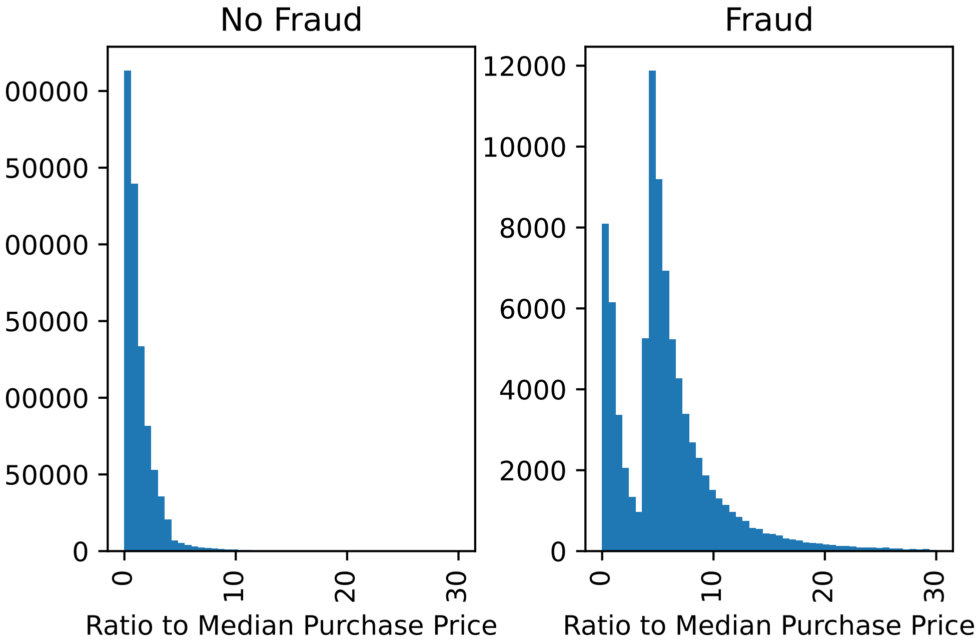
Let's look at some distributions. Most of the variables in the dataset are binary (0 or 1) variables. However, we also have some continuous variables. Let's plot the distribution of the variable `ratio_to_median_purchase_price`, which is a continuous variable.

```
df['ratio_to_median_purchase_price'].hist(bins = 50, range=[0, 30])
plt.xlabel('Ratio to Median Purchase Price')
plt.ylabel('Count')
plt.show()
```



We can also plot the distribution of the variable `ratio_to_median_purchase_price` by the target variable `fraud` to see if there are any differences between fraudulent and non-fraudulent transactions

```
fig, ax = plt.subplots(1,2)
df['ratio_to_median_purchase_price'].hist(bins = 50, range=[0, 30],
 ↪  by=df['fraud'], ax = ax)
ax[0].set_xlabel('Ratio to Median Purchase Price')
ax[1].set_xlabel('Ratio to Median Purchase Price')
ax[0].set_ylabel('Count')
ax[0].set_title('No Fraud')
ax[1].set_title('Fraud')
plt.show()
```



There are indeed some differences between fraudulent and non-fraudulent transactions. For example, fraudulent transactions seem to have a higher ratio to the median purchase price, which is expected as fraudsters might try to make large transactions to maximize their profit.

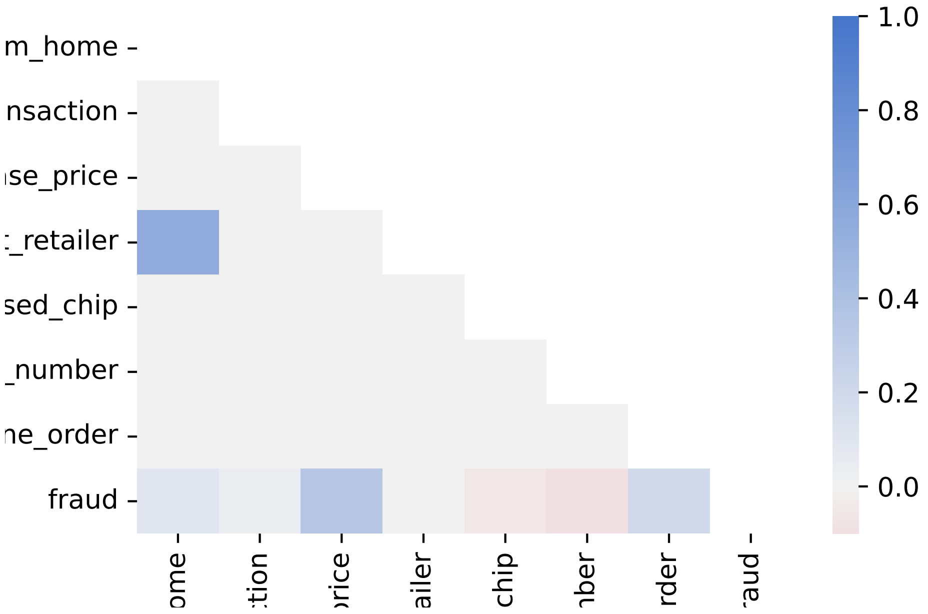
We can also look at the correlation between the variables in the dataset. The correlation is a measure of how two variables move together

```
df.corr() # Pearson correlation (for linear relationships)
```

```
df.corr('spearman') # Spearman correlation (for monotonic relationships)
```

This is still a bit hard to read. We can visualize the correlation matrix with a heatmap using the Seaborn library, which we have already imported at the beginning of the notebook.

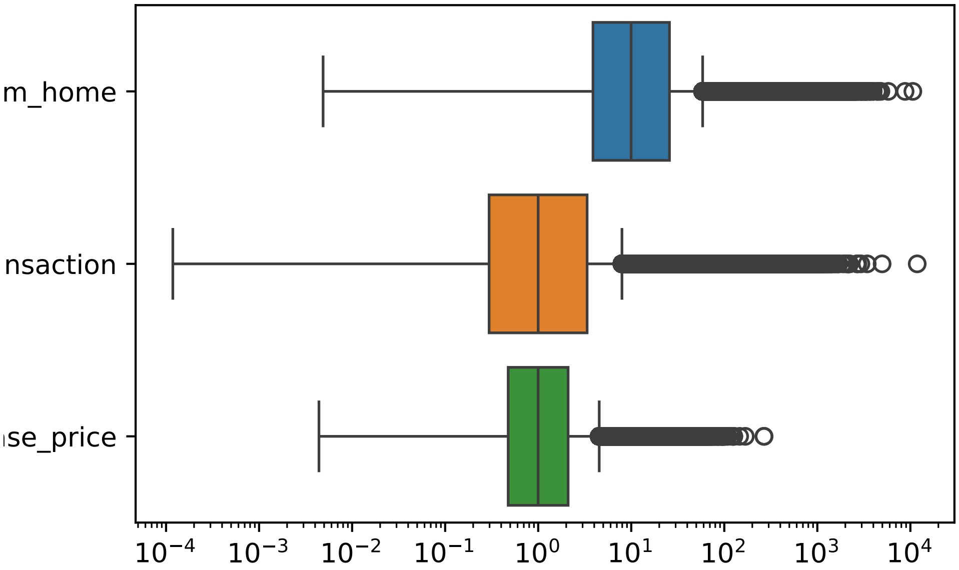
```
corr = df.corr('spearman')
cmap = sns.diverging_palette(10, 255, as_cmap=True) # Create a color map
mask = np.triu(np.ones_like(corr, dtype=bool)) # Create a mask to only show
 ↪  the lower triangle of the matrix
sns.heatmap(corr, cmap=cmap, vmax=1, center=0, mask=mask) # Create a heatmap
 ↪  of the correlation matrix (Note: vmax=1 makes sure that the color map
 ↪  goes up to 1 and center=0 are used to center the color map at 0)
plt.show()
```



Note how `ratio_to_median_purchase_price` is positively correlated with `fraud`, which is expected as we saw in the previous plot that fraudulent transactions have a higher ratio to the median purchase price. Furthermore, `used_chip` and `used_pin_number` are negatively correlated with `fraud`, which makes sense as transactions, where the chip or the pin is used, are supposed to be more secure.

We can also plot boxplots to visualize the distribution of the variables

```
selector = ['distance_from_home', 'distance_from_last_transaction',
 ↪  'ratio_to_median_purchase_price'] # Select the variables we want to plot
plt.figure()
ax = sns.boxplot(data = df[selector], orient = 'h')
ax.set(xscale = "log") # Set the x-axis to a logarithmic scale to better
 ↪  visualize the data
plt.show()
```



Boxplots are a good way to visualize the distribution of a variable, as they show the median, the interquartile range, and the outliers. Each of the distributions shown in the boxplots above has a long right tail, which explains the large number of outliers. However, you have to be careful: you cannot just remove these outliers since these are likely to be fraudulent transactions.

Let's see how many fraudulent transactions we would remove if we blindly remove the outliers according to the interquartile range

```
# Compute the interquartile range
Q1 = df['ratio_to_median_purchase_price'].quantile(0.25)
Q3 = df['ratio_to_median_purchase_price'].quantile(0.75)
IQR = Q3 - Q1

# Identify outliers based on the interquartile range
threshold = 1.5
outliers = df[(df['ratio_to_median_purchase_price'] < Q1 - threshold * IQR)
 ↪  | (df['ratio_to_median_purchase_price'] > Q3 + threshold * IQR)]

# Count the number of fraudulent transactions among our selected outliers
outliers['fraud'].value_counts()
```

```
df['fraud'].value_counts()
```

53092 of 87403 (more than half!) of our fraudulent transactions would be removed if we would have blindly removed the outliers according to the interquartile range. This is a significant number of observations, which would likely hurt the performance of our machine-learning model. Therefore, we should not remove these outliers. It would make the imbalance of our dataset even worse.

Splitting the Data into Training and Test Sets

Before we can train a machine learning model, we need to split our dataset into a training set and a test set.

```
X = df.drop('fraud', axis=1) # All variables except `fraud`
y = df['fraud'] # Only our fraud variables
```

The training set is used to train the model, while the test set is used to evaluate the model. We will use the `train_test_split` function from the `sklearn.model_selection` module to split our dataset. We will use 70% of the data for training and 30% for testing. We will also set the `stratify` argument to `y` to make sure that the distribution of the target variable is the same in the training and test sets. Otherwise, we might randomly not have any fraudulent transactions in the test set, which would make it impossible to correctly evaluate our model.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
 ↪  test_size = 0.3, random_state = 42)
```

Scaling Features

To improve the performance of our machine learning model, we should scale the features. This is especially important for models that are sensitive to the scale of the features, such as logistic regression. We will use the `StandardScaler` class from the `sklearn.preprocessing` module to scale the features. The `StandardScaler` class scales the features so that they have a mean of 0 and a standard deviation of 1. Since we don't want to scale features that are binary (0 or 1), we will define a small function that scales only the features that we want

```
def scale_features(scaler, df, col_names, only_transform=False):

    # Extract the features we want to scale
    features = df[col_names]

    # Fit the scaler to the features and transform them
    if only_transform:
        features = scaler.transform(features.values)
    else:
        features = scaler.fit_transform(features.values)

    # Replace the original features with the scaled features
    df[col_names] = features
```

Then, we need to run the function

```
col_names = ['distance_from_home', 'distance_from_last_transaction',
 ↪  'ratio_to_median_purchase_price']
scaler = StandardScaler()
scale_features(scaler, X_train, col_names)
scale_features(scaler, X_test, col_names, only_transform=True)
```

Note that we only fit the scaler to the training set and then transform both the training and test set. This ensures that the same values for the features produce the same output in the training and test set. Otherwise, if we fit the scaler to the test data as well, the meaning of certain values in the test set might change, which would make it impossible to evaluate the model correctly.

> ℹ Mini-Exercise
>
> Try switching to `MinMaxScaler` instead of `StandardScaler` and see how it affects the performance of the model. `MinMaxScaler` scales the features so that they are between 0 and 1.

3.4.2 Implementing Logistic Regression

Now that we have explored and preprocessed our dataset, we can move on to the next step: training a machine learning model. We will use a logistic regression model to predict whether a transaction is fraudulent or not.

Using the `LogisticRegression` class from the `sklearn.linear_model` module, fitting the model to the data is straightforward using the `fit` method

```
clf = LogisticRegression().fit(X_train, y_train)
```

We can then use the `predict` method to predict the class of the test set

```
clf.predict(X_test.head(5))
```

The actual classes of the first five observations in the test dataset are

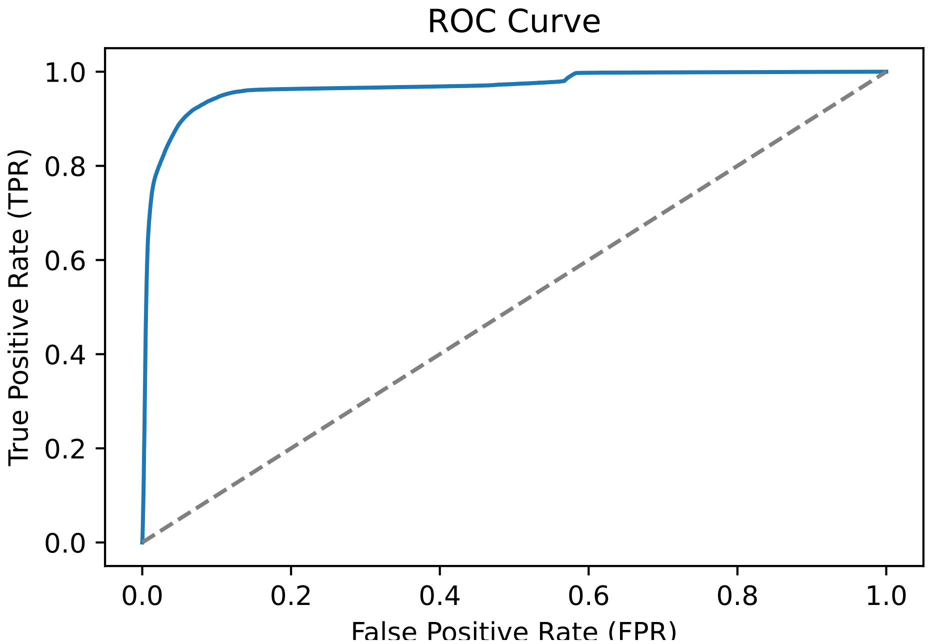
```
y_test.head(5)
```

This seems to match quite well. Let's have a look at different performance metrics

```
y_pred = clf.predict(X_test)
y_proba = clf.predict_proba(X_test)

print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
print(f"Precision: {precision_score(y_test, y_pred)}")
print(f"Recall: {recall_score(y_test, y_pred)}")
print(f"ROC AUC: {roc_auc_score(y_test, y_proba[:, 1])}")
```

As expected, the accuracy is quite high since we do not have many fraudulent transactions. Recall that the precision ($\text{Precision} = \frac{\text{TP}}{\text{TP}+\text{FP}}$) is the fraction of correctly predicted fraudulent transactions among all transactions predicted to be fraudulent. The recall ($\text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}}$) is the fraction of correctly predicted fraudulent transactions among the actual fraudulent transactions. The ROC AUC is the area under the curve for the receiver operating characteristic (ROC) curve.

```
# Compute the ROC curve
y_proba = clf.predict_proba(X_test)
fpr, tpr, thresholds = roc_curve(y_test, y_proba[:,1])

# Plot the ROC curve
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1], linestyle='--', color='grey')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve')
plt.show()
```

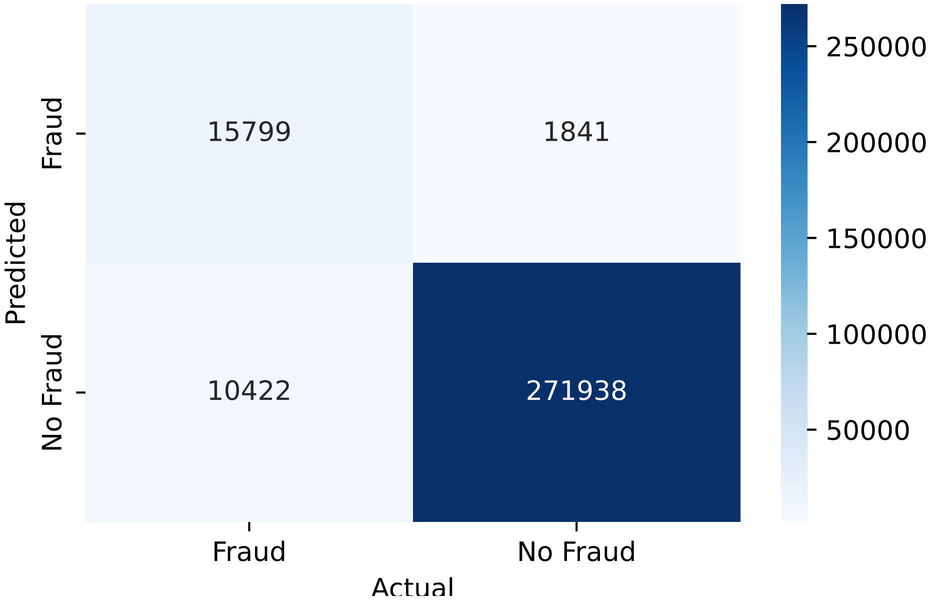


The confusion matrix for the test set can be computed as follows

```
conf_mat = confusion_matrix(y_test, y_pred, labels=[1, 0]).transpose() #
 ↪  Transpose the sklearn confusion matrix to match the convention in the
 ↪  lecture
conf_mat
```

We can also plot the confusion matrix as a heatmap

```
sns.heatmap(conf_mat, annot=True, cmap='Blues', fmt='g',
 ↪  xticklabels=['Fraud', 'No Fraud'], yticklabels=['Fraud', 'No Fraud'])
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.show()
```



As you can see, we have mostly true negatives and true positives. However, there is still a significant number of false negatives, which means that we are missing fraudulent transactions, and a significant number of false positives, which means that we are predicting transactions as fraudulent that are not fraudulent.

If we would like to use a threshold other than 0.5 to predict the class of the test set, we can do so as follows

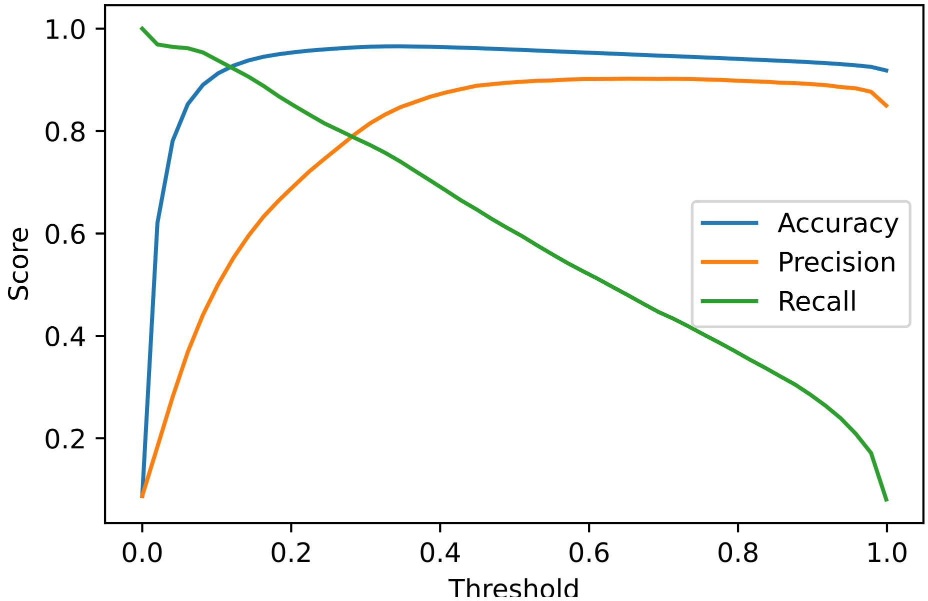
```
# Alternative threshold
threshold = 0.1

# Predict the class of the test set
y_pred_alt = (y_proba[:, 1] >= threshold).astype(int)

# Show the performance metrics
print(f"Accuracy: {accuracy_score(y_test, y_pred_alt)}")
print(f"Precision: {precision_score(y_test, y_pred_alt)}")
print(f"Recall: {recall_score(y_test, y_pred_alt)}")
```

Setting a lower threshold increases the recall but decreases the precision. This is because we are more likely to predict a transaction as fraudulent, which increases the number of true positives but also the number of false positives.

What the correct threshold is depends on the problem at hand. For example, if the cost of missing a fraudulent transaction is very high, you might want to set a lower threshold to increase the recall. If the cost of falsely predicting a transaction as fraudulent is very high, you might want to set a higher threshold to increase the precision.

We can also plot the performance metrics for different thresholds

```python
N = 50
thresholds_array = np.linspace(0.0, 0.999, N)
accuracy_array = np.zeros(N)
precision_array = np.zeros(N)
recall_array = np.zeros(N)

# Compute the performance metrics for different thresholds
for ii, thresh in enumerate(thresholds_array):
    y_pred_alt_tmp = (y_proba[:, 1] > thresh).astype(int)
    accuracy_array[ii] = accuracy_score(y_test, y_pred_alt_tmp)
    precision_array[ii] = precision_score(y_test, y_pred_alt_tmp)
    recall_array[ii] = recall_score(y_test, y_pred_alt_tmp)

# Plot the performance metrics
plt.plot(thresholds_array, accuracy_array, label='Accuracy')
plt.plot(thresholds_array, precision_array, label='Precision')
plt.plot(thresholds_array, recall_array, label='Recall')
plt.xlabel('Threshold')
plt.ylabel('Score')
plt.legend()
plt.show()
```



3.4.3 Implementing K-Nearest Neighbors

As an alternative to logistic regression, we can also use a K-Nearest Neighbors (KNN) classifier. The KNN classifier is a simple and intuitive machine learning model that classifies a new observation based on the classes of its k-nearest neighbors in the training set.

Let's restrict ourselves to variables that are continuous for the KNN classifier

```
continuous_vars = ['distance_from_home', 'distance_from_last_transaction',
 ↪  'ratio_to_median_purchase_price']
X_train_knn = X_train[continuous_vars]
X_test_knn = X_test[continuous_vars]
```

We can create a KNN classifier with k=5 and fit it to the training data

```
knn = KNeighborsClassifier(n_neighbors=5).fit(X_train_knn, y_train)
y_pred_knn = knn.predict(X_test_knn)
```

We can then evaluate the performance of the KNN classifier using the same performance metrics as before

```
print(f"Accuracy: {accuracy_score(y_test, y_pred_knn)}")
print(f"Precision: {precision_score(y_test, y_pred_knn)}")
print(f"Recall: {recall_score(y_test, y_pred_knn)}")
print(f"ROC AUC: {roc_auc_score(y_test, knn.predict_proba(X_test_knn)[:,
 ↪  1])}")
```

The KNN classifier seems to perform slightly worse than the logistic regression model. Part of the reason for this is that we only used three features for the KNN classifier, while we used all features for the logistic regression model.

ℹ Mini-Exercise

Try different values of `n_neighbors` (k) and plot how it affects the performance of the KNN classifier. *Hint:* You can use a loop to train the KNN classifier for different values of k and store the performance metrics in arrays. Then, you can plot the performance metrics as a function of k.

ℹ Mini-Exercise

Implement a 5-fold cross-validation for the logistic regression and K-Nearest Neighbors classifiers. Use the `cross_val_score` function from the `sklearn.model_selection` module.

```
# Import the cross_val_score function
from sklearn.model_selection import cross_val_score

# Apply 5-fold cross-validation to the classifier clf
cv_scores = cross_val_score(clf, X, y, cv=5, scoring='roc_auc')

# Mean of the cross-validation scores
cv_scores.mean()
```

3.4.4 Conclusions

In this section, we have seen how to implement a logistic regression model and a K-Nearest Neighbors classifier in Python. We have loaded a dataset, explored and preprocessed it, and trained a logistic regression model and a K-Nearest Neighbors classifier to predict whether a transaction is fraudulent or not. We have evaluated the model using different performance metrics and have seen how the choice of threshold affects the performance of the model.

There are many ways to improve the performance of the models. For example, we could try different machine learning models, or engineer new features. We could also try to deal with the imbalanced dataset by using techniques such as oversampling or undersampling. However, this is beyond the scope of this section.

Chapter 4

Decision Trees

Now that we have covered some of the basics of machine learning, we can start looking at some of the most popular machine learning algorithms. In this chapter, we will focus on **Decision Trees** and **tree-based ensemble methods** such as **Random Forests** and **(Gradient) Boosted Trees**.

4.1 What is a Decision Tree?

Decision trees, also called **Classification and Regression Trees (CART)** are a popular **supervised learning method**. As the name CART suggests, they are **used for both classification and regression** problems. They are simple to understand and interpret, and the process of building a decision tree is intuitive. Decision trees are also the **foundation of more advanced ensemble methods** like Random Forests and Boosting.

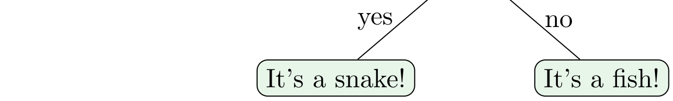


Figure 4.1: Classification Tree - Classification of Dogs, Snakes, Fish, and Birds based on their Features

Figure 4.1 shows an example of a decision tree for a classification problem, i.e., a **classification tree**. In this case, the decision tree is used to classify animals into four categories: dogs, snakes, fish, and birds. The tree asks a series of questions about the features of the animal (e.g., number of legs, feathers, and habitat) and uses the answers to classify the animal. This means that the tree partitions the feature space into different regions that are associated with a particular class label.

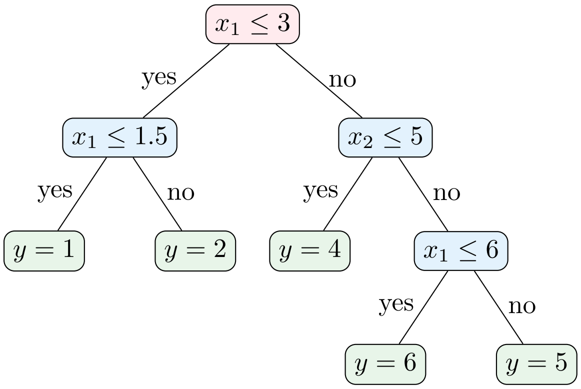


Figure 4.2: Regression Tree - Prediction of y based on x_1 and x_2

Figure 4.2 shows an example of a decision tree for a regression problem, i.e., a **regression tree**. In this case, the decision tree is used to predict some continuous variable y (e.g., a house price) based on features x_1 and x_2 (e.g., number of rooms and size of the property). As Figure 4.3 shows, the regression tree partitions the (x_1, x_2)-space into different regions that are associated with a predicted value y. Mathematically, the prediction of a regression tree can be expressed as

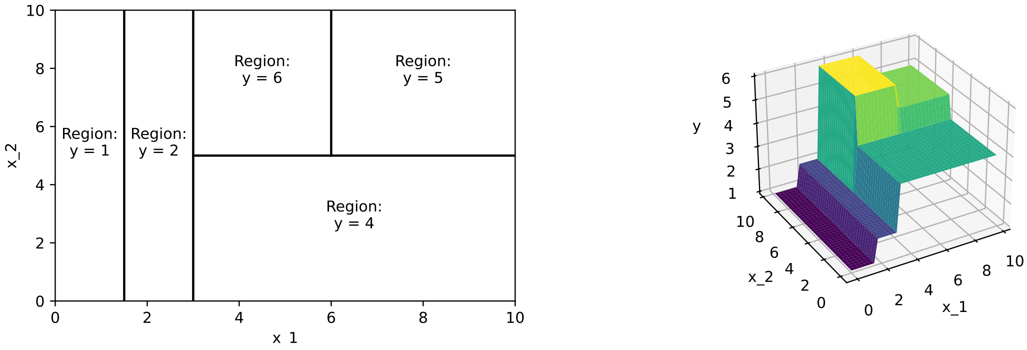
$$\hat{y} = \sum_{m=1}^{M} c_m \mathbb{1}(x \in R_m)$$

where R_m are the regions of the feature space, c_m are the predicted (i.e., average) values in the regions, $\mathbb{1}(x \in R_m)$ is an indicator function that is 1 if x is in region R_m and 0 otherwise, and M is the number of regions.

Mini-Exercise

Given the decision tree in Figure 4.2, what would be the predicted value of y for the following data points?

1. $(x_1, x_2) = (1, 1)$
2. $(x_1, x_2) = (2, 2)$
3. $(x_1, x_2) = (2, 8)$
4. $(x_1, x_2) = (10, 4)$



(a) Regions

(b) Predictions

Figure 4.3: Regression Tree - Regions and Predictions of Decision Tree in Figure 4.2

> 5. $(x_1, x_2) = (7, 8)$

4.2 Terminology

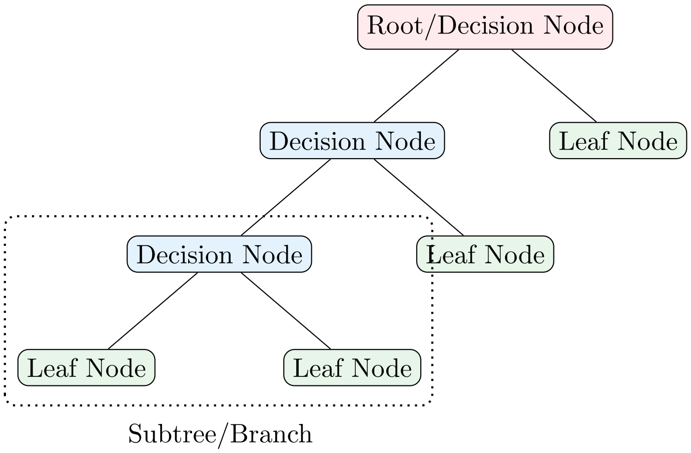


Figure 4.4: Decision Tree - Terminology

Figure 4.4 shows some of the terminology that you might encounter in decision trees. The **root node** is the first node in the tree. The root node is split into child nodes, which can be either **decision nodes** or **leaf nodes**. The decision nodes are further split into decision nodes or leaf nodes. The leaf nodes represent the final prediction of the model. A **subtree** or **branch** is a part of the tree that starts at a decision node and ends at a leaf node. The **depth** of a tree is the length of the longest path from the root node to a leaf node.

Furthermore, one can also differentiate between **child** and **parent nodes**. A child node is a node that results from a split (e.g., the first (reading from the top) decision node and leaf node in Figure 4.4 are child nodes of the root node). The parent node is the node that is split to create the child nodes (e.g., the root node in Figure 4.4 is the parent node of the first decision node and leaf node).

4.3 How To Grow a Tree

A key question is how to determine the order of variables and thresholds that are used in all the splits of a decision tree. There are different algorithms to grow a decision tree, but the most common one is the **CART algorithm**. The CART algorithm is a **greedy algorithm** that grows the tree in a **top-down** manner. The reason for this algorithm choice is that it is computationally infeasible to consider all possible (fully grown) trees to find the best-performing one. So, the CART algorithm grows the tree in a step-by-step manner choosing the splits in a greedy manner (i.e., choosing the one that performs best at that step). This means that the algorithm does not consider the future consequences of the current split and may not find the optimal tree.

The basic idea is to find a split that minimizes some loss function Q^s and to repeat this recursively for all resulting child nodes. Suppose we start from zero, meaning that we first need to determine the root node. We compute the loss function Q^s for all possible splits s that we can make. This means we need to consider all variables in our dataset (and all split thresholds) and choose the one that minimizes the loss Q^s. We then repeat this process for each of the child nodes, and so on, until we reach a stopping criterion. Figure 4.5 shows an example of a candidate split.

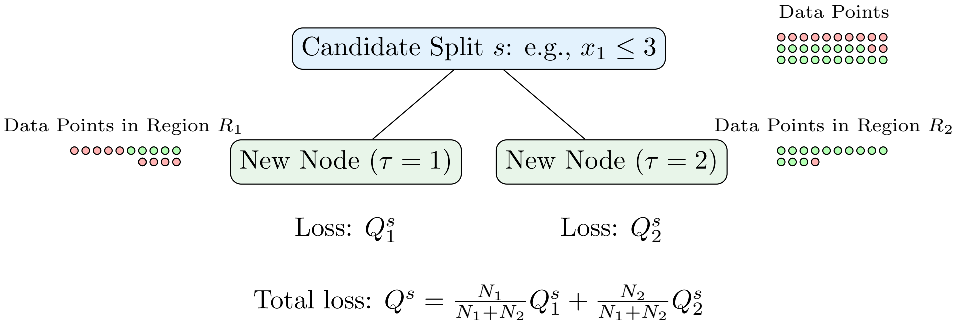

Data Points
Candidate Split s: e.g., $x_1 \leq 3$
Data Points in Region R_1
New Node ($\tau = 1$)
New Node ($\tau = 2$)
Data Points in Region R_2
Loss: Q_1^s
Loss: Q_2^s
Total loss: $Q^s = \frac{N_1}{N_1+N_2}Q_1^s + \frac{N_2}{N_1+N_2}Q_2^s$

Figure 4.5: Example of Decision Tree Split

Let τ denote the index of a leaf node with each leaf node τ corresponding to a region R_τ with N_τ data points. In the case of a classification problem, the loss function is typically either the

Gini impurity

$$Q_\tau^s = \sum_{k=1}^{K} p_{\tau k}(1 - p_{\tau k}) = 1 - \sum_{k=1}^{K} p_{\tau k}^2$$

or the **cross-entropy**

$$Q_\tau^s = -\sum_{k=1}^{K} p_{\tau k} \log(p_{\tau k})$$

where $p_{\tau k}$ is the proportion of observations in region R_τ that belong to class k and K is the number of classes. Note that both measures become zero when all observations in the region belong to the same class (i.e., $p_{\tau k} = 1$ or $p_{\tau k} = 0$). This is the ideal case for a classification problem: we say that the node is **pure**.

In the case of a regression problem, the loss function is typically the **mean squared error (MSE)**

$$Q_\tau^s = \frac{1}{N_\tau} \sum_{i \in R_\tau} (y_i - \hat{y}_\tau)^2$$

where $\hat{y}_\tau$ is the predicted value of the target variable y in region R_τ

$$\hat{y}_\tau = \frac{1}{N_\tau} \sum_{i \in R_\tau} y_i,$$

i.e., the average of the target variable in region R_τ.

The **total loss of a split** Q^s is then the weighted sum of the loss functions of the child nodes

$$Q^s = \frac{N_1}{N_1 + N_2} Q_1^s + \frac{N_2}{N_1 + N_2} Q_2^s$$

where N_1 and N_2 are the number of data points in the child nodes.

Once we have done this for the root node, we repeat the process for each child node. Then, we repeat it for the child nodes of the child nodes, and so on, until we reach a stopping criterion. The stopping criterion can be, for example, a maximum depth of the tree, a minimum number of data points in a leaf node, or a minimum reduction in the loss function.

4.3.1 Example: Classification Problem

Suppose you have the data in Table 4.1. The goal is to predict whether a bank will default based on two features: whether the bank is systemically important and its Common Equity Tier 1 (CET1) ratio (i.e., the ratio of CET1 capital to risk-weighted assets). The CET1 ratio is a measure of a bank's financial strength.

Table 4.1: (Made-up) Data for Classification Problem (Bank Default Prediction)

Default	Systemically Important Bank	CET1 Ratio (in %)
Yes	No	8.6
No	No	9
Yes	Yes	10.6
Yes	Yes	10.8
No	No	11.2
No	No	11.5
No	Yes	12.4

Given that you only have two features, CET1 Ratio and whether it is a systemically important bank, you only have two possible variables for the root node. However, since CET1 is a continuous variable, there are potentially many thresholds that you could use to split the data. To find this threshold, we need to calculate the **Gini impurity** of each possible split and choose the one that minimizes the impurity.

Table 4.2: Gini Impurities for Different CET1 Thresholds

CET1 Ratio Threshold	Q	Q	Q
8.8	0	0.44	0.38
9.8	0.5	0.48	0.49
10.7	0.44	0.38	0.4
11	0.38	0	0.21
11.35	0.48	0	0.34
11.95	0.5	0	0.43

According to Table 4.2, the best split is at a CET1 ratio of 11.0%. The Gini impurity for CET1 $\leq$ 11% is 0.38, the Gini impurity of CET1 $>$ 11% is 0, and the total impurity is 0.21. However, we could also split based on whether a bank would be systemically important. In this case, the Gini impurity of the split is 0.40. This means that the best split is based on the CET1 ratio. We split the data into two regions: one with a CET1 ratio of 11.0% or less and one with a CET1 ratio of more than 11.0%.

Note that the child node for a CET1 ratio of more than 11.0% is already pure, i.e., all banks in this region are not defaulting. However, the child node for a CET1 ratio of 11.0% or less is not pure meaning that we can do additional splits as shown in Figure 4.6. In particular, both, the split at a CET1 ratio of 8.8% and the split based on whether a bank is systemically important yield a Gini impurity of 0.25. We choose the split based on whether a bank is systemically important as the next split, which means we can do the final split based on the CET1 ratio.

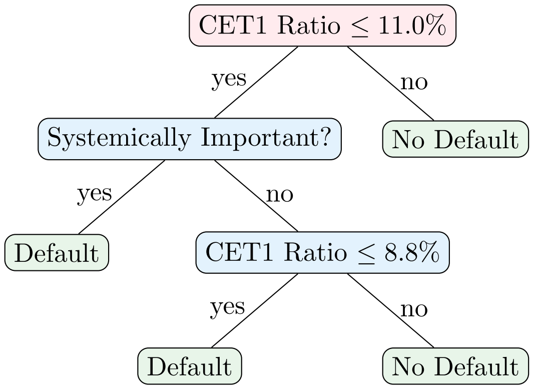


Figure 4.6: Classification Tree for Table 4.1

4.3.2 Stopping Criteria and Pruning a Tree

A potential problem with decision trees is that they can **overfit** the training data. In principle, we can get the error down to zero if we just make enough splits. This means that the tree can become too complex and capture noise in the data rather than the underlying relationship. To prevent this, we usually set some early stopping criteria like

- A maximum depth of the tree,
- A minimum number of data points in a leaf node,
- A minimum number of data points required in a decision node for a split,
- A minimum reduction in the loss function, or
- A maximum number of leaf nodes,

which will prevent the tree from growing too large and all the nodes from becoming pure. We can also use a combination of these criteria. In the Python applications, we will see how to set some of these stopping criteria.

Figure 4.7 shows an example of how stopping criteria affect the fit of a decision tree. Note that without any stopping criteria, the tree fits the data perfectly but is likely to overfit. By setting a maximum depth or a minimum number of data points in a leaf node, we can prevent the tree from overfitting the data.

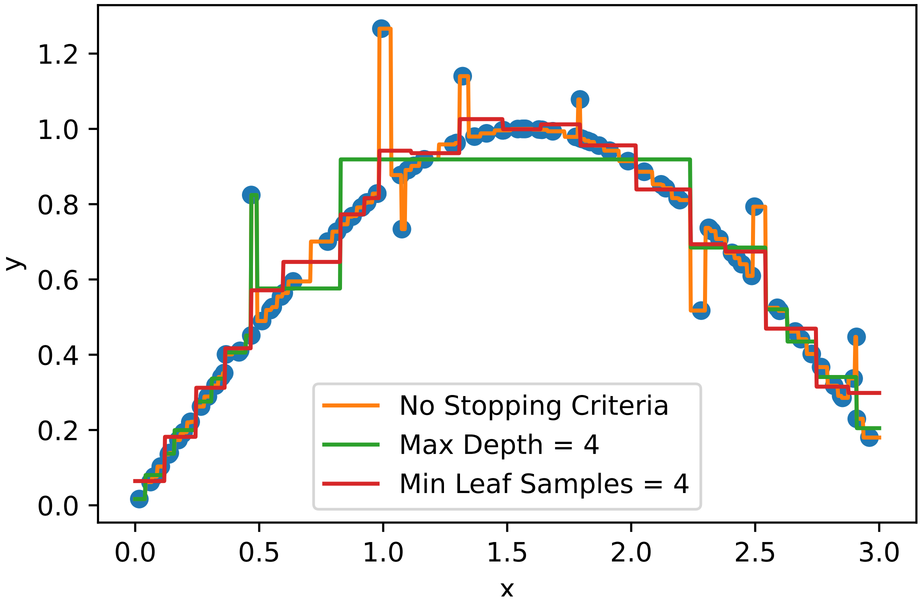


Figure 4.7: Regression Tree - Effect of Stopping Criteria

Another way to prevent overfitting is to **prune** the tree, i.e., to remove nodes from the tree according to certain rules. This is done *after* (not during) growing the tree. One common approach is to use **cost-complexity pruning**. The idea is related to regularization that we have seen before, i.e., we add a term to the loss functions above that penalizes tree complexity. The pruning process is controlled by a hyperparameter λ that determines the trade-off between the complexity of the tree and its fit to the training data.

ℹ Mini-Exercise

How would the decision tree in Figure 4.6 look like if

1. we required a minimum of 2 data points in a leaf node?
2. we required a maximum depth of 2?
3. we required a maximum depth of 2 and a minimum of 3 data points in a leaf node?
4. we required a minimum of 3 data points for a split?
5. we required a minimum of 5 data points for a split?

4.4 Advantages and Disadvantages

As noted by Murphy (2022), decision trees are popular because of some of the **advantages** they offer

- Easy to interpret
- Can handle mixed discrete and continuous inputs
- Insensitive to monotone transformations of the inputs
- Automatic variable selection
- Relatively robust to outliers
- Fast to fit and scale well to large data sets
- Can handle missing input features[1]

Their **disadvantages** include

- Not very accurate at prediction compared to other kinds of models (note, for example, the piece-wise constant nature of the predictions in regression problems)
- They are **unstable**: small changes to the input data can have large effects on the structure of the tree (small changes at the top can affect the rest of the tree)

4.5 Ensemble Methods

Decision trees are powerful models, but they can be unstable. To address these issues, we can use **ensemble methods** that combine multiple decision trees to improve the performance of the model. The two most popular ensemble methods are **Random Forests** and **Boosting**.

4.5.1 Random Forests

The idea of **Random Forests** is to build a **large number of trees** (also called weak learners in this context), each of which is **trained on a random subset of the data**. The predictions of the trees are then averaged in regression tasks or determined through majority voting in the case of classification tasks to make the final prediction. Training multiple trees on random subsets of the data is also called **bagging** (short for **bootstrap aggregating**). Random Forests adds an additional layer of randomness by selecting a random subset of features at each split. This means that each tree is trained on a different subset of the data and considers different features at each decision node.

The basic steps of the **Random Forest algorithm** are as follows:

[1] Note to handle missing input data one can use "backup" variables that are correlated with the variable of interest and can be used to make a split whenever the data is missing. Such splits are called **surrogate splits**. In the case of categorical variables, one can also use a separate category for missing values.

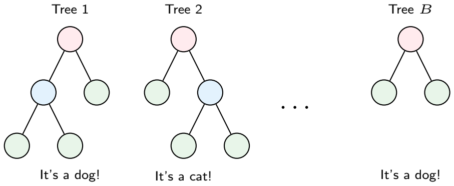


Figure 4.8: Random Forests - Ensemble of Decision Trees with Majority Decision “It’s a dog!”

1. **Bootstrapping**: Randomly draw N samples with replacement from the training data.
2. **Grow a tree**: For each node of the tree, randomly select m features from the p features in the bootstrap dataset and find the best split based on these m features.
3. **Repeat**: Repeat steps 1 and 2 B times to grow B trees.
4. **Prediction**: To get the prediction for a new data point, average the predictions of all trees in the case of regression or use a majority vote in the case of classification.

Note that because we draw samples with replacement, some samples will not be included in the bootstrap sample. These samples are called **out-of-bag (OOB) samples**. The OOB samples can be used to estimate the performance of the model without the need for cross-validation since it is “performed along the way” (Hastie, Tibshirani, and Friedman (2009)). The OOB error is almost identical to the error obtained through k-fold cross-validation.

4.5.2 Boosting

Another popular ensemble method is **Boosting**. The idea behind boosting is to train a sequence of weak learners (e.g., decision trees), each of which tries to correct the mistakes of the previous one. The predictions of the weak learners are then combined to make the final prediction. Note how this differs from Random Forests where the trees are trained independently of each other in parallel, while in boosting we sequentially train the trees to fix the mistakes of the previous ones. The basic steps can be roughly summarized as follows:

1. **Initialize the model**: Construct a base tree with just a root node. In the case of a regression problem, the prediction could be the mean of the target variable. In the case of a classification problem, the prediction could be the log odds of the target variable.
2. **Train a weak learner**: Train a weak learner on the data. The weak learner tries to correct the mistakes of the previous model.
3. **Update the model**: Update the model by adding the weak learner to the model. The added weak learner is weighted by a learning rate η.
4. **Repeat**: Repeat steps 2 and 3 until we have grown B trees.

XGBoost (eXtreme Gradient Boosting) is a popular implementation of the (gradient) boosting algorithm. It is known for its performance and is widely used in machine learning competitions. The algorithm is based on the idea of gradient boosting, which is a generalization of boosting. We will see how to implement XGBoost in Python but will not go into the details of the algorithm here. Other popular implementations of the boosting algorithm are AdaBoost and LightGBM.

4.5.3 Interpreting Ensemble Methods

A downside of using ensemble methods is that you lose the interpretability of a single decision tree. However, there are ways to interpret ensemble methods. One way is to look at the **feature importance**. Feature importance tells you how much each feature contributes to the reduction in the loss function. The idea is that features that are used in splits that lead to a large reduction in the loss function are more important. Murphy (2022) shows that the feature importance of feature k is

$$R_k(b) = \sum_{j=1}^{J-1} G_j \mathbb{I}(v_j = k)$$

where the sum is over all non-leaf (internal) nodes, G_j is the loss reduction (gain) at node j, and $v_j = k$ if node j uses feature k. Simply put, we sum up all gains of the splits that use feature k. Then, we average over all trees in our ensemble to get the feature importance of feature k

$$R_k = \frac{1}{B} \sum_{b=1}^{B} R_k(b).$$

Note that the resulting R_k are sometimes normalized such that the maximum value is 100. This means that the most important feature has a feature importance of 100 and all other features are scaled accordingly. Note that feature importance can in principle also be computed for a single decision tree.

⚠ Warning

Note that feature importance tends to favor continuous variables and variables with many categories (for an example see here). As an alternative, one can use **permutation importance** which is a model-agnostic way to compute the importance of different features. The idea is to shuffle the values of a feature in the test data set and see how much the model performance decreases. The more the performance decreases, the more important the feature is.

4.6 Python Implementation

Let's have a look at how to implement a decision tree in Python. Again, we need to first import the required packages and load the data

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, roc_auc_score, recall_score, precision_score, roc_curve
from sklearn.inspection import permutation_importance
pd.set_option('display.max_columns', 50) # Display up to 50 columns
from io import BytesIO
from urllib.request import urlopen
from zipfile import ZipFile
import os.path

# Check if the file exists
if not os.path.isfile('data/card_transdata.csv'):

    print('Downloading dataset...')

    # Define the dataset to be downloaded
    zipurl = 'https://www.kaggle.com/api/v1/datasets/download/dhanushnarayananr/credit-card-fraud'

    # Download and unzip the dataset in the data folder
    with urlopen(zipurl) as zipresp:
        with ZipFile(BytesIO(zipresp.read())) as zfile:
            zfile.extractall('data')

    print('DONE!')

else:

    print('Dataset already downloaded!')
```

```
Dataset already downloaded!
```

```
# Load the data
df = pd.read_csv('data/card_transdata.csv')
```

This is the **dataset of credit card transactions** from Kaggle.com which we have used before. Recall that the target variable y is `fraud`, which indicates whether the transaction is fraudulent or not. The other variables are the features x of the transactions.

```
df.head(20)
```

```
    distance_from_home  distance_from_last_transaction  \
0            57.877857                        0.311140
1            10.829943                        0.175592
2             5.091079                        0.805153
3             2.247564                        5.600044
4            44.190936                        0.566486
5             5.586408                       13.261073
6             3.724019                        0.956838
7             4.848247                        0.320735
8             0.876632                        2.503609
9             8.839047                        2.970512
10           14.263530                        0.158758
11           13.592368                        0.240540
12          765.282559                        0.371562
13            2.131956                       56.372401
14           13.955972                        0.271522
15          179.665148                        0.120920
16          114.519789                        0.707003
17            3.589649                        6.247458
18           11.085152                       34.661351
19            6.194671                        1.142014

    ratio_to_median_purchase_price  repeat_retailer  used_chip  \
0                         1.945940              1.0        1.0
1                         1.294219              1.0        0.0
2                         0.427715              1.0        0.0
3                         0.362663              1.0        1.0
4                         2.222767              1.0        1.0
5                         0.064768              1.0        0.0
6                         0.278465              1.0        0.0
```

```
7                         1.273050              1.0        0.0
8                         1.516999              0.0        0.0
9                         2.361683              1.0        0.0
10                        1.136102              1.0        1.0
11                        1.370330              1.0        1.0
12                        0.551245              1.0        1.0
13                        6.358667              1.0        0.0
14                        2.798901              1.0        0.0
15                        0.535640              1.0        1.0
16                        0.516990              1.0        0.0
17                        1.846451              1.0        0.0
18                        2.530758              1.0        0.0
19                        0.307217              1.0        0.0

    used_pin_number  online_order  fraud
0               0.0           0.0    0.0
1               0.0           0.0    0.0
2               0.0           1.0    0.0
3               0.0           1.0    0.0
4               0.0           1.0    0.0
5               0.0           0.0    0.0
6               0.0           1.0    0.0
7               1.0           0.0    0.0
8               0.0           0.0    0.0
9               0.0           1.0    0.0
10              0.0           1.0    0.0
11              0.0           1.0    0.0
12              0.0           0.0    0.0
13              0.0           1.0    1.0
14              0.0           1.0    0.0
15              1.0           1.0    0.0
16              0.0           0.0    0.0
17              0.0           0.0    0.0
18              0.0           1.0    0.0
19              0.0           0.0    0.0
```

```
df.describe()
```

```
       distance_from_home  distance_from_last_transaction  \
count      1000000.000000                  1000000.000000   
mean            26.628792                        5.036519   
std             65.390784                       25.843093   
min              0.004874                        0.000118   
```

```
25%              3.878008                        0.296671
50%              9.967760                        0.998650
75%             25.743985                        3.355748
max          10632.723672                    11851.104565

       ratio_to_median_purchase_price  repeat_retailer       used_chip  \
count                  1000000.000000   1000000.000000  1000000.000000
mean                         1.824182         0.881536        0.350399
std                          2.799589         0.323157        0.477095
min                          0.004399         0.000000        0.000000
25%                          0.475673         1.000000        0.000000
50%                          0.997717         1.000000        0.000000
75%                          2.096370         1.000000        1.000000
max                        267.802942         1.000000        1.000000

       used_pin_number    online_order           fraud
count   1000000.000000  1000000.000000  1000000.000000
mean          0.100608        0.650552        0.087403
std           0.300809        0.476796        0.282425
min           0.000000        0.000000        0.000000
25%           0.000000        0.000000        0.000000
50%           0.000000        1.000000        0.000000
75%           0.000000        1.000000        0.000000
max           1.000000        1.000000        1.000000
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 8 columns):
 #   Column                          Non-Null Count    Dtype  
---  ------                          --------------    -----  
 0   distance_from_home              1000000 non-null  float64
 1   distance_from_last_transaction  1000000 non-null  float64
 2   ratio_to_median_purchase_price  1000000 non-null  float64
 3   repeat_retailer                 1000000 non-null  float64
 4   used_chip                       1000000 non-null  float64
 5   used_pin_number                 1000000 non-null  float64
 6   online_order                    1000000 non-null  float64
 7   fraud                           1000000 non-null  float64
dtypes: float64(8)
memory usage: 61.0 MB
```

4.6.1 Data Preprocessing

Since we have already explored the dataset in the previous section, we can skip that part and move directly to the data preprocessing.

We will again split the data into training and test sets using the `train_test_split` function

```
X = df.drop('fraud', axis=1) # All variables except `fraud`
y = df['fraud'] # Only our fraud variables
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
 ↪  test_size = 0.3, random_state = 42)
```

Then we can do the feature scaling to ensure our non-binary variables have mean zero and variance 1

```
def scale_features(scaler, df, col_names, only_transform=False):

    # Extract the features we want to scale
    features = df[col_names]

    # Fit the scaler to the features and transform them
    if only_transform:
        features = scaler.transform(features.values)
    else:
        features = scaler.fit_transform(features.values)

    # Replace the original features with the scaled features
    df[col_names] = features

col_names = ['distance_from_home', 'distance_from_last_transaction',
 ↪  'ratio_to_median_purchase_price']
scaler = StandardScaler()
scale_features(scaler, X_train, col_names)
scale_features(scaler, X_test, col_names, only_transform=True)
```

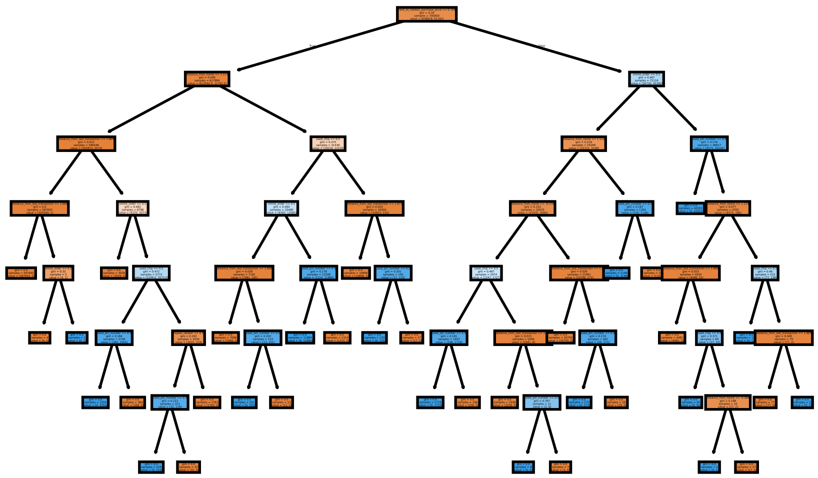
4.6.2 Implementing a Decision Tree Classifier

We can now implement a decision tree model using the `DecisionTreeClassifier` class from the `sklearn.tree` module. Fitting the model to the data is almost the same as when we used logistic regression

```
clf_dt = DecisionTreeClassifier(random_state=0).fit(X_train, y_train)
```

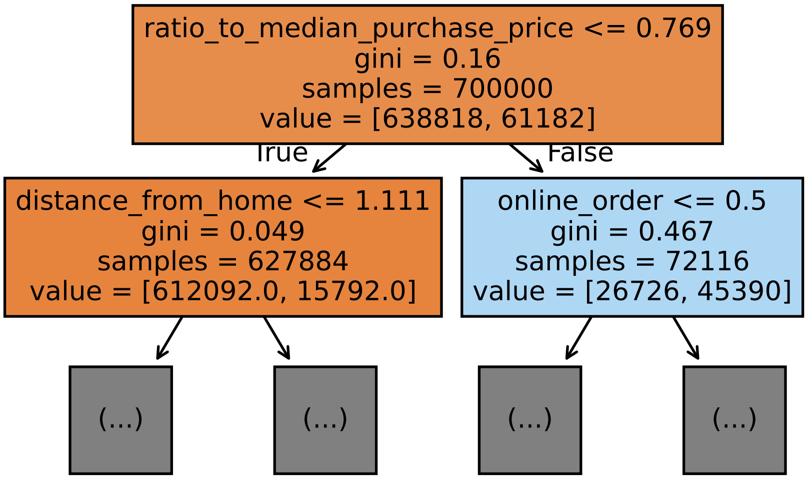
We can visualize the tree using the `plot_tree` function from the `sklearn.tree` module

```
plot_tree(clf_dt, filled=True, feature_names = X_train.columns.to_list())
plt.show()
```



The tree is quite large and it's difficult to see details. Let's only look at the first level of the tree

```
plot_tree(clf_dt, max_depth=1, filled=True, feature_names =
 ↪  X_train.columns.to_list(), fontsize=10)
plt.show()
```



Recall from the data exploration that `ratio_to_median_purchase_price` was highly correlated with fraud. The decision tree model seems to have picked up on this as well since the first split is based on this variable. Also, note that the order in which the variables are split can differ between different branches of the tree.

We can also make predictions using the model and evaluate its performance using the same functions as before

```
y_pred_dt = clf_dt.predict(X_test)
y_proba_dt = clf_dt.predict_proba(X_test)

print(f"Accuracy: {accuracy_score(y_test, y_pred_dt)}")
```

```
Accuracy: 0.9999833333333333
```

```
print(f"Precision: {precision_score(y_test, y_pred_dt)}")
```

```
Precision: 0.9999237223493517
```

```
print(f"Recall: {recall_score(y_test, y_pred_dt)}")
```

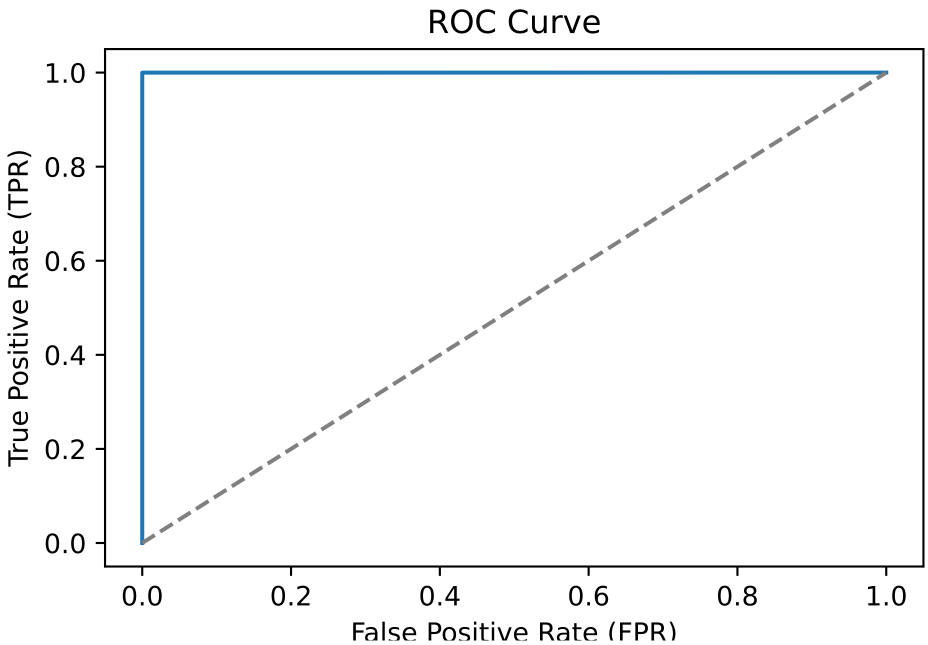
```
Recall: 0.999885587887571
```

```
print(f"ROC AUC: {roc_auc_score(y_test, y_proba_dt[:, 1])}")
```

```
ROC AUC: 0.999939141362689
```

The decision tree performs substantially better than the logistic regression. The ROC AUC score is much closer to the maximum value of 1 and we have an almost perfect classifier

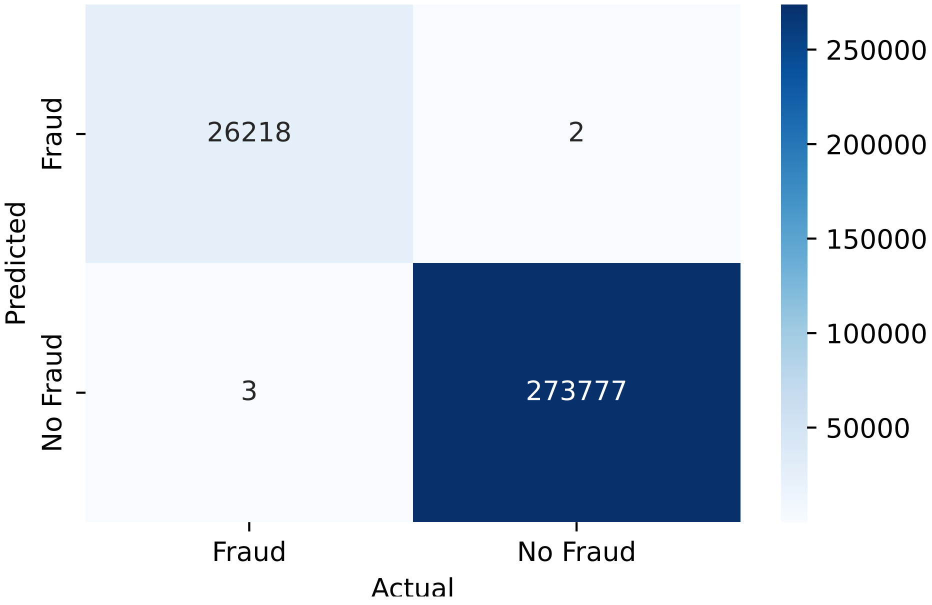
```
# Compute the ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_proba_dt[:, 1])

# Plot the ROC curve
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1], linestyle='--', color='grey')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve')
plt.show()
```



Let's also check the confusion matrix to see where we still make mistakes

```
conf_mat = confusion_matrix(y_test, y_pred_dt, labels=[1, 0]).transpose() #
↪  Transpose the sklearn confusion matrix to match the convention in the
↪  lecture
```

```
sns.heatmap(conf_mat, annot=True, cmap='Blues', fmt='g',
 ↪  xticklabels=['Fraud', 'No Fraud'], yticklabels=['Fraud', 'No Fraud'])
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.show()
```



There are only 3 false negatives, i.e., fraudulent transactions that we did not detect. There are also 2 false positives, i.e., “false alarms”, where non-fraudulent transactions were classified as fraudulent. The decision tree classifier is almost perfect which is a bit suspicious. We might have been lucky in the sense that the training and test sets were split in a way that the model performs very well. We should not expect this to be the case in general. It might be better to use cross-validation to get a more reliable estimate of the model’s performance.

4.6.3 Implementing a Random Forest Classifier

We can also implement a random forest model using the `RandomForestClassifier` class from the `sklearn.ensemble` module. Fitting the model to the data is almost the same as when we used logistic regression and decision trees

```
clf_rf = RandomForestClassifier(random_state = 0).fit(X_train, y_train)
```

Note that it takes a bit longer to train the Random Forest since we have to train many trees (the default setting is 100). We can also make predictions using the model and evaluate its performance using the same functions as before

```
y_pred_rf = clf_rf.predict(X_test)
y_proba_rf = clf_rf.predict_proba(X_test)

print(f"Accuracy: {accuracy_score(y_test, y_pred_rf)}")
```

```
Accuracy: 0.9999833333333333
```

```
print(f"Precision: {precision_score(y_test, y_pred_rf)}")
```

```
Precision: 1.0
```

```
print(f"Recall: {recall_score(y_test, y_pred_rf)}")
```

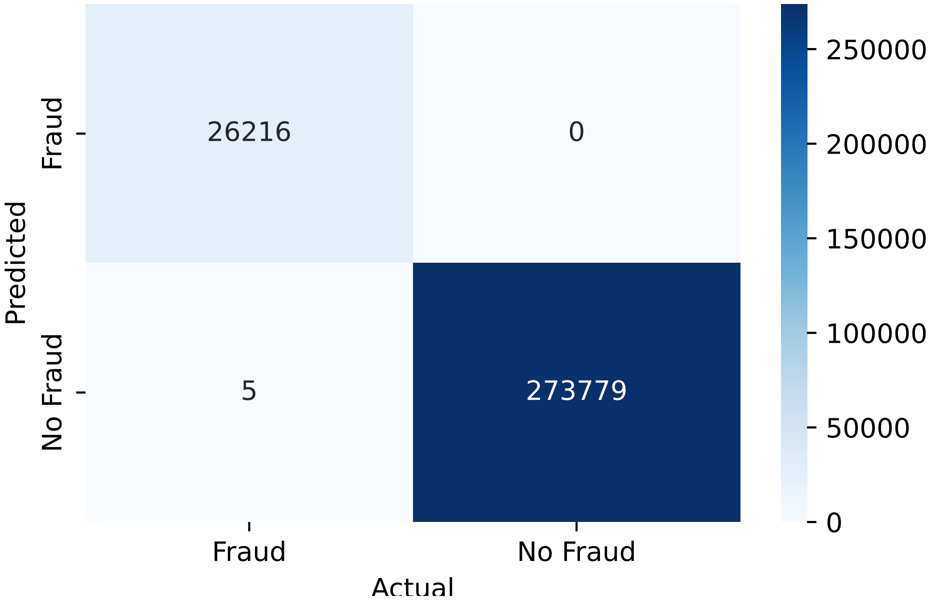
```
Recall: 0.9998093131459517
```

```
print(f"ROC AUC: {roc_auc_score(y_test, y_proba_rf[:, 1])}")
```

```
ROC AUC: 0.999999999164201
```

As expected, the Random Forest performs better than the Decision Tree in the metrics we have used. Now, let's also check the confusion matrix to see where we still make mistakes

```
conf_mat = confusion_matrix(y_test, y_pred_rf, labels=[1, 0]).transpose() #
 ↪  Transpose the sklearn confusion matrix to match the convention in the
 ↪  lecture
sns.heatmap(conf_mat, annot=True, cmap='Blues', fmt='g',
 ↪  xticklabels=['Fraud', 'No Fraud'], yticklabels=['Fraud', 'No Fraud'])
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.show()
```



There are still some false negatives, but the number of false positives has decreased compared to the Decision Tree model.

4.6.4 Implementing a XGBoost Classifier

Let's also have a look at the XGBoost classifier. We can implement the model using the `XGBClassifier` class from the `xgboost` package. Fitting the model to the data is almost the same as when we used logistic regression, decision trees, and random forests, even though it is not part of the `sklearn` package. This is because the `xgboost` package is designed to work well with the `sklearn` package. Let's fit the model to the data

```
clf_xgb = XGBClassifier(random_state = 0).fit(X_train, y_train)
```

```
y_pred_xgb = clf_xgb.predict(X_test)
y_proba_xgb = clf_xgb.predict_proba(X_test)

print(f"Accuracy: {accuracy_score(y_test, y_pred_xgb)}")
```

```
Accuracy: 0.9983333333333333
```

```
print(f"Precision: {precision_score(y_test, y_pred_xgb)}")
```

```
Precision: 0.9896440129449838
```

```
print(f"Recall: {recall_score(y_test, y_pred_xgb)}")
```

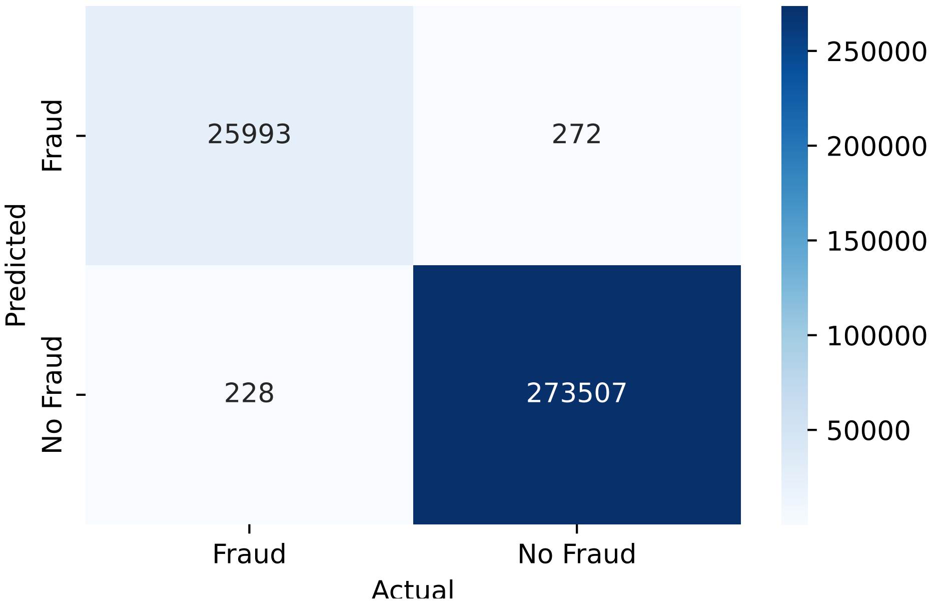
```
Recall: 0.9913046794553984
```

```
print(f"ROC AUC: {roc_auc_score(y_test, y_proba_xgb[:, 1])}")
```

```
ROC AUC: 0.9999735930686904
```

Let's also check the confusion matrix to see where we still make mistakes

```
conf_mat = confusion_matrix(y_test, y_pred_xgb, labels=[1, 0]).transpose() #
↪  Transpose the sklearn confusion matrix to match the convention in the
↪  lecture
sns.heatmap(conf_mat, annot=True, cmap='Blues', fmt='g',
↪  xticklabels=['Fraud', 'No Fraud'], yticklabels=['Fraud', 'No Fraud'])
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.show()
```

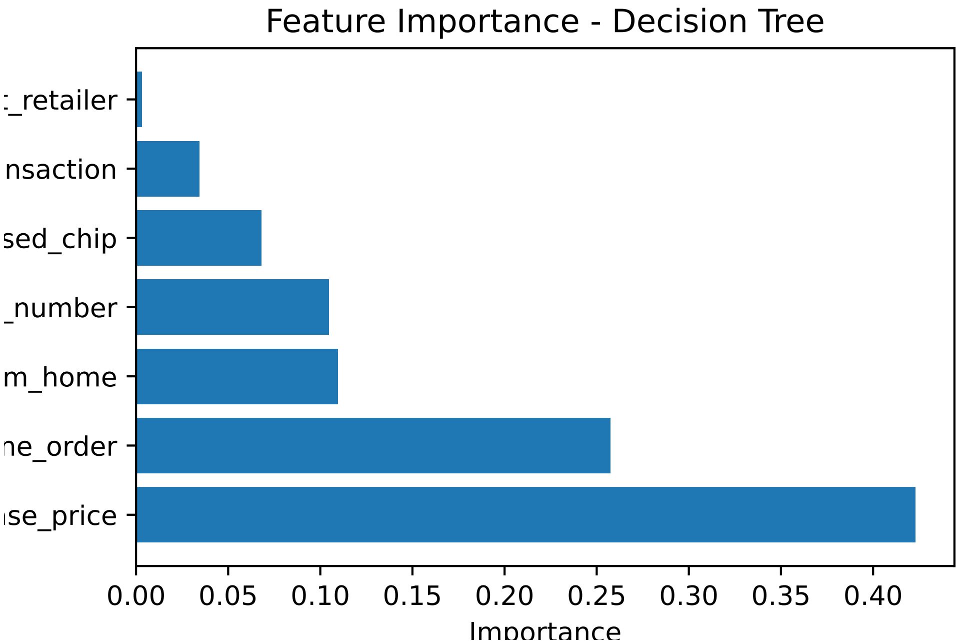


The XGBoost model seems to perform a bit worse than the Random Forest model. There are more false negatives and false positives. However, the model is still very good at detecting fraudulent transactions and has a high ROC AUC score. Adjusting the hyperparameters of the model might improve its performance.

4.6.5 Feature Importance

We can also look at the feature importance of each model. The feature importance is a measure of how much each feature contributes to the model's predictions. Let's start with the Decision Tree model

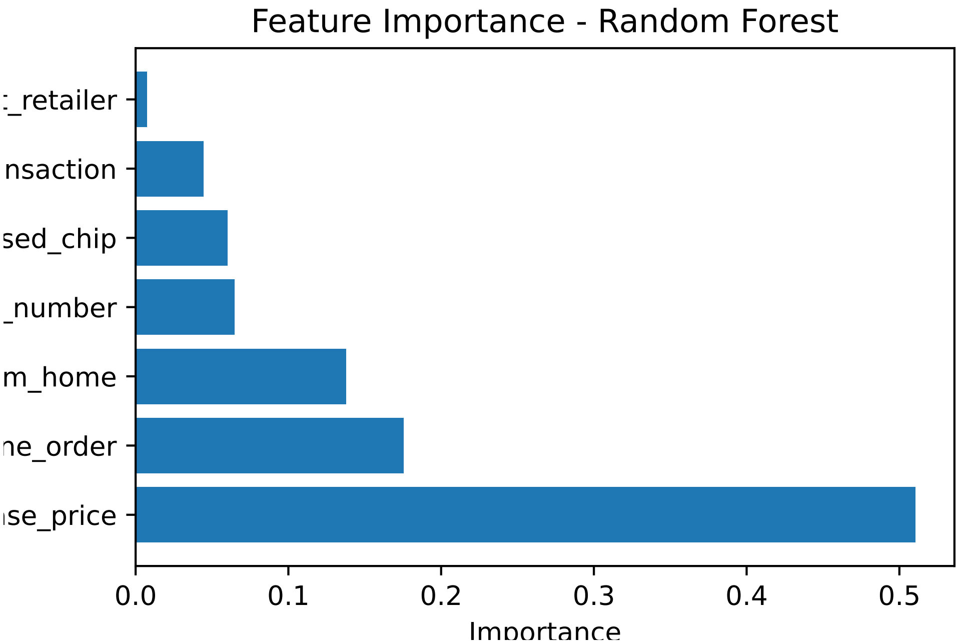
```
# Create a DataFrame with the feature importance
df_feature_importance_dt = pd.DataFrame({'Feature': X_train.columns,
 ↪  'Importance': clf_dt.feature_importances_})
df_feature_importance_dt = df_feature_importance_dt.sort_values('Importance',
 ↪  ascending=False)

# Plot the feature importance
plt.barh(df_feature_importance_dt['Feature'],
 ↪  df_feature_importance_dt['Importance'])
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importance - Decision Tree')
plt.show()
```



This shows that the `ratio_to_median_purchase_price` is the most important feature for determining whether a transaction is fraudulent or not. Whether a transaction is online, is important as well.

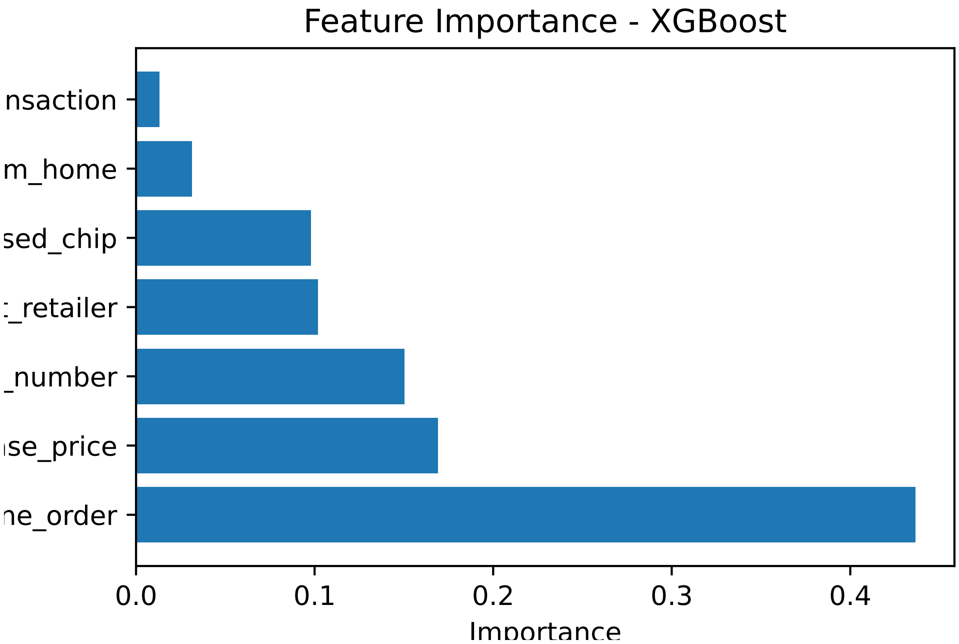
Let's also look at the feature importance of the Random Forest model

```
# Create a DataFrame with the feature importance
df_feature_importance_rf = pd.DataFrame({'Feature': X_train.columns,
 ↪  'Importance': clf_rf.feature_importances_})
df_feature_importance_rf = df_feature_importance_rf.sort_values('Importance',
 ↪  ascending=False)

# Plot the feature importance
plt.barh(df_feature_importance_rf['Feature'],
 ↪  df_feature_importance_rf['Importance'])
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importance - Random Forest')
plt.show()
```



Somewhat surprisingly, XGBoost seems to have picked up on different features than the Decision Tree and Random Forest models. The most important feature is `online_order`, followed by `ratio_to_median_purchase_price` as you can see from the plot below

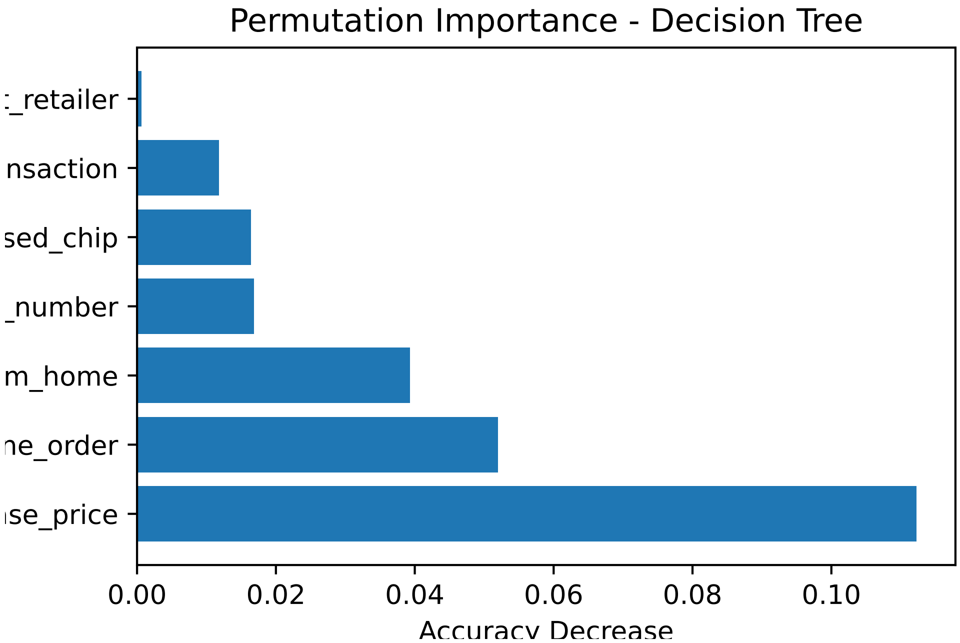
```
# Create a DataFrame with the feature importance
df_feature_importance_xgb = pd.DataFrame({'Feature': X_train.columns,
 ↪  'Importance': clf_xgb.feature_importances_})
df_feature_importance_xgb =
 ↪  df_feature_importance_xgb.sort_values('Importance', ascending=False)

# Plot the feature importance
plt.barh(df_feature_importance_xgb['Feature'],
 ↪  df_feature_importance_xgb['Importance'])
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importance - XGBoost')
plt.show()
```



4.6.6 Permutation Importance

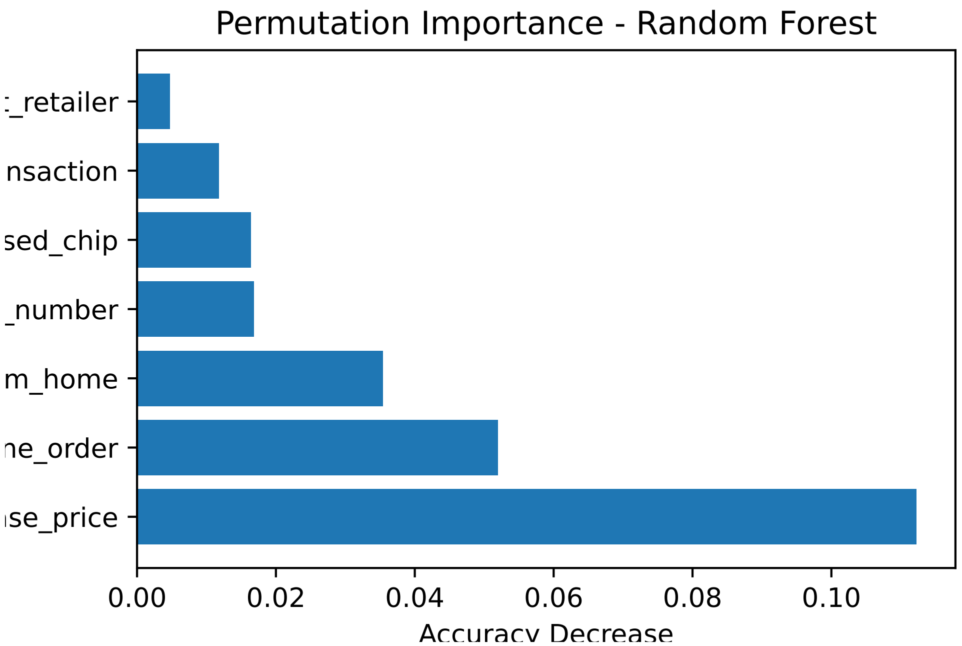
We can also look at the permutation importance of each model. The permutation importance is a measure of how much each feature contributes to the model's predictions. The permutation importance is calculated by permuting the values of each feature and measuring how much the model's performance decreases. Let's start with the Decision Tree model

```
# Calculate the permutation importance
result_dt = permutation_importance(clf_dt, X_test, y_test, n_repeats=10,
 ↪  random_state=0)

# Create a DataFrame with the permutation importance
df_permutation_importance_dt = pd.DataFrame({'Feature': X_train.columns,
 ↪  'Importance': result_dt.importances_mean})
df_permutation_importance_dt =
 ↪  df_permutation_importance_dt.sort_values('Importance', ascending=False)

# Plot the permutation importance
plt.barh(df_permutation_importance_dt['Feature'],
 ↪  df_permutation_importance_dt['Importance'])
plt.xlabel('Accuracy Decrease')
plt.ylabel('Feature')
plt.title('Permutation Importance - Decision Tree')
plt.show()
```



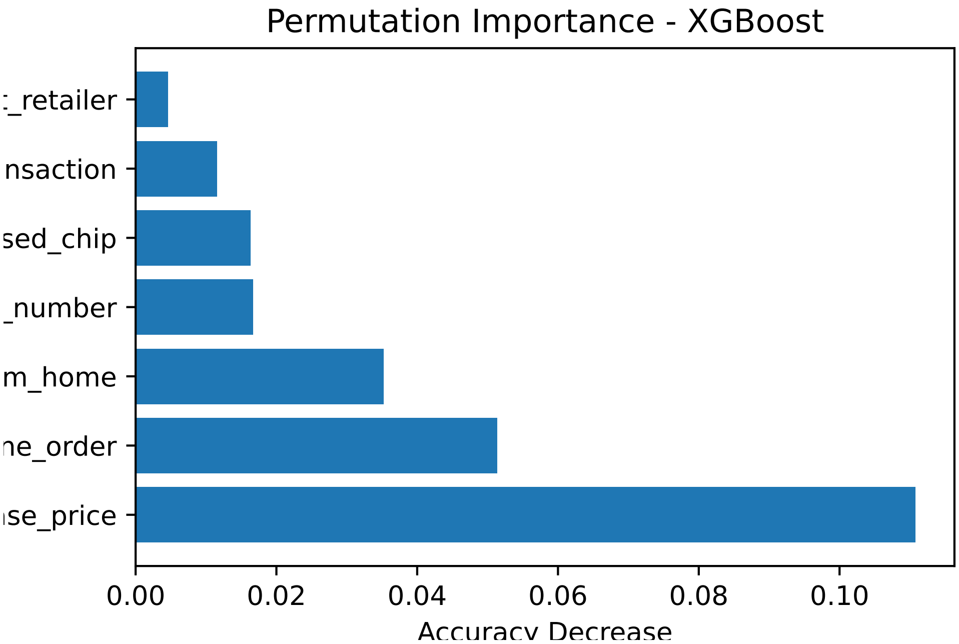
Let's also look at the permutation importance of the Random Forest model

```
# Calculate the permutation importance
result_rf = permutation_importance(clf_rf, X_test, y_test, n_repeats=10,
 ↪  random_state=0)

# Create a DataFrame with the permutation importance
df_permutation_importance_rf = pd.DataFrame({'Feature': X_train.columns,
 ↪  'Importance': result_rf.importances_mean})
df_permutation_importance_rf =
 ↪  df_permutation_importance_rf.sort_values('Importance', ascending=False)

# Plot the permutation importance
plt.barh(df_permutation_importance_rf['Feature'],
 ↪  df_permutation_importance_rf['Importance'])
plt.xlabel('Accuracy Decrease')
plt.ylabel('Feature')
plt.title('Permutation Importance - Random Forest')
plt.show()
```



Let's also look at the permutation importance of the XGBoost model

```
# Calculate the permutation importance
result_xgb = permutation_importance(clf_xgb, X_test, y_test, n_repeats=10,
 ↪  random_state=0)

# Create a DataFrame with the permutation importance
df_permutation_importance_xgb = pd.DataFrame({'Feature': X_train.columns,
 ↪  'Importance': result_xgb.importances_mean})
df_permutation_importance_xgb =
 ↪  df_permutation_importance_xgb.sort_values('Importance', ascending=False)

# Plot the permutation importance
plt.barh(df_permutation_importance_xgb['Feature'],
 ↪  df_permutation_importance_xgb['Importance'])
plt.xlabel('Accuracy Decrease')
plt.ylabel('Feature')
plt.title('Permutation Importance - XGBoost')
plt.show()
```



Here the results for the three models are quite similar. The most important feature is `ratio_to_median_purchase_price`, followed by `online_order`.

4.6.7 Conclusions

In this section, we have seen how to implement decision trees, random forests, and XGBoost classifiers in Python. We have also seen how to evaluate the performance of these models using metrics such as accuracy, precision, recall, and ROC AUC. We have seen that the Random Forest and XGBoost models perform better than the Decision Tree model. Furthermore, we looked at the feature and permutation importance of each model to see which features are most important for determining whether a transaction is fraudulent or not.

Chapter 5

Neural Networks

In this chapter, we have a look at neural networks which are a popular machine learning method. We will cover the basics of neural networks and how they can be trained.

5.1 What is a Neural Network?

Neural networks are at the core of many cutting-edge machine learning models. They can be used as both a **supervised and unsupervised learning method**. In this course, we will focus on their application in supervised learning where they are used for both **regression and classification** tasks. While they are conceptually not much more difficult to understand than decision trees, a neural network is **not as easy to interpret as a decision tree**. For this reason, they are often called black boxes, meaning that it is not so clear what is happening inside. Furthermore, neural networks tend to be more difficult to train and for tabular data, which is the type of structured data that you will typically encounter, gradient-boosted decision trees tend to perform better. Nevertheless, since neural networks are what enabled many of the recent advances in AI, they are an important topic to cover, even if it is only to better understand what has been driving recent innovations.

It is common to represent neural networks as directed graphs. Figure 5.1 shows a feedforward neural network with one hidden layer, $N = 2$ inputs, $M = 3$ neurons in the hidden layer, and a single output. The input layer is connected to the hidden layer, which is connected to the output layer. For simplicity, we will mainly work with neural networks that are feedforward (i.e. their graphs are acyclical), with dense layers (i.e. each layer is fully connected to the previous), and without connections that skip layers.

As we will see later on, under certain (relatively weak) conditions

- Neural networks are **universal approximators** (can approximate any (Borel measurable) function)

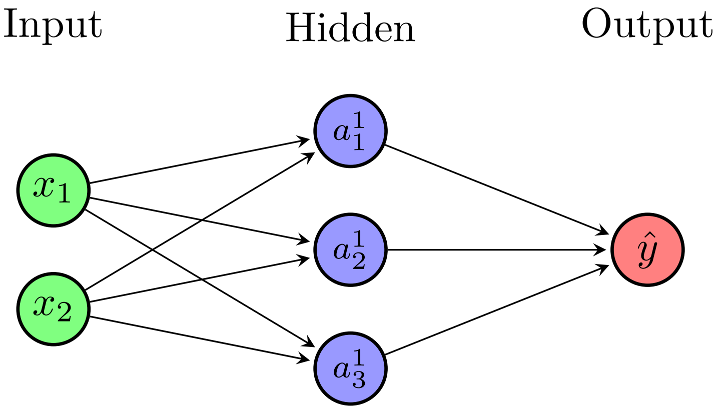


Figure 5.1: A Feedforward Neural Network with One Hidden Layer

- Neural networks **break the curse of dimensionality** (can handle very high dimensional functions)

This makes them interesting for a wide range of fields in economics, e.g., quantitative macroeconomics or econometrics. However, neural networks are not a magic bullet, and there are some downsides in terms of the large data requirements, interpretability and training difficulty.

5.1.1 Origins of the Term "Neural Network"

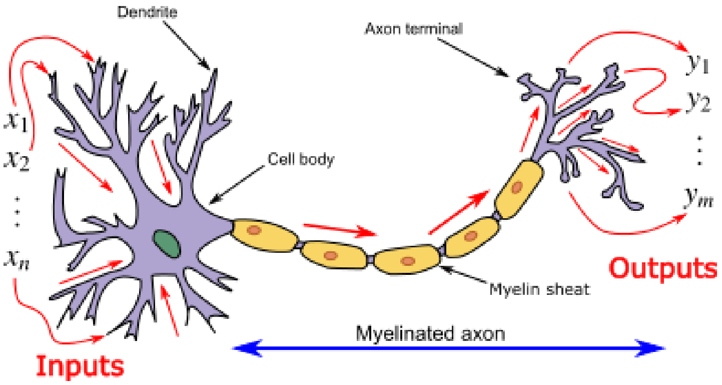


Figure 5.2: A biological neuron (Source: Wikipedia)

The term "neural network" originates in attempts to find mathematical representations of information processing in biological systems (Bishop 2006). Although biological neurons provided the initial inspiration, the artificial neurons used in modern machine learning bear only a superficial resemblance to their biological counterparts. The biological analogy can be helpful when first learning about neural networks, but it should not be taken too literally. Figure 5.2 shows a biological neuron for reference.

5.2 An Artificial Neuron

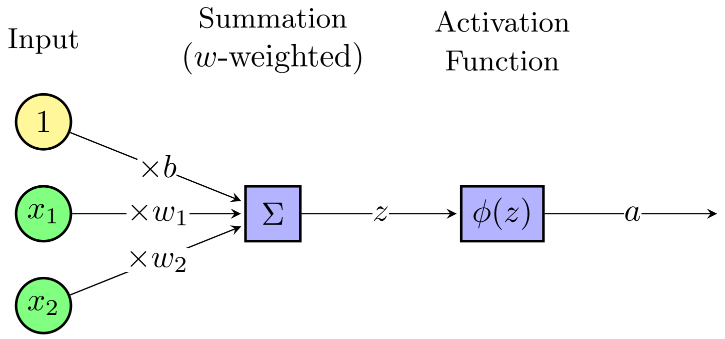


Figure 5.3: Artificial Neuron

Artificial neurons are the **basic building blocks** of neural networks. Figure 5.3 shows a single artificial neuron. The N inputs denoted $x = (x_1, x_2, \dots, x_N)'$ are linearly combined into z using weights w and bias b

$$z = b + \sum_{i=1}^{N} w_i x_i = \sum_{i=0}^{N} w_i x_i$$

where we defined an additional input $x_0 = 1$ and $w_0 = b$.

The linear combination z is transformed using an **activation function** $\phi(z)$.

$$a = \phi(z) = \phi\left(\sum_{i=0}^{N} w_i x_i\right)$$

The activation function **introduces non-linearity** into the neural network and allows it to learn highly non-linear functions. The particular choice of activation function depends on the application.

This should look familiar to you already. If we set $\phi(z) = z$, we get a *linear regression* model and if we set $\phi(z) = \frac{1}{1+e^{-z}}$, we get a *logistic regression* model. This is because the basic building block, the artificial neuron, is a generalized linear model.

5.2.1 Activation Functions

Common activation functions include

- Sigmoid: $\phi(z) = \frac{1}{1+e^{-z}}$
- Hyperbolic tangent: $\phi(z) = \tanh(z)$

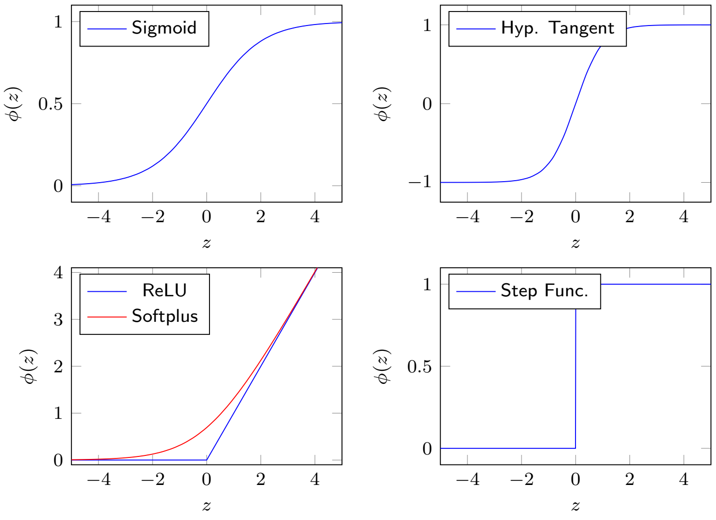


Figure 5.4: Activation Functions

- Rectified linear unit (ReLU): $\phi(z) = \max(0, z)$
- Softplus: $\phi(z) = \log(1 + e^z)$

ReLU has become popular in deep neural networks in recent years because of its good performance in these applications. Since economic problems usually involve smooth functions, softplus can be a good alternative.

5.2.2 A Special Case: Perceptron

Perceptrons were developed in the 1950s and have only one artificial neuron. Perceptrons use a **step function** as an activation function

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise}, \end{cases}$$

Perceptrons can be used for basic classification. However, the step function is usually not used in neural networks because it is not differentiable at $z = 0$ and zero everywhere else. This makes it unsuitable for the back-propagation algorithm, which is used for determining the network weights.

> ℹ Mini-Exercise
>
> What would the decision boundary of a perceptron look like if we have two inputs x_1 and x_2 and the weights $w_1 = 1$, $w_2 = 1$, and $b = -1$?

5.3 Building a Neural Network from Artificial Neurons

We can build a neural network by stacking multiple artificial neurons. For this reason, it is sometimes also called a **multilayer perceptron** (MLP). A neural network with a **single hidden layer** is a linear combination of M artificial neurons a_j

$$a_j = \phi(z_j) = \phi\left(b_j^1 + \sum_{i=1}^{N} w_{ji}^1 x_i\right)$$

with the output defined as

$$g(x; w) = b^2 + \sum_{j=1}^{M} w_j^2 a_j$$

where N is the number of inputs, M is the number of neurons in the hidden layer, and w are the weights and biases of the network. The width of the neural network is M.

Figure 5.5 shows a feedforward neural network with a single hidden layer, $N = 2$ inputs, $M = 3$ neurons in the hidden layer, and a single output. Note that the biases can be thought of as additional weights that are multiplied by a constant input of 1.

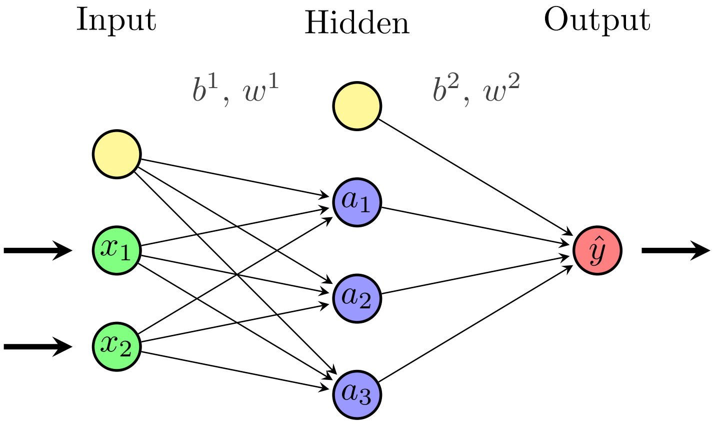


Figure 5.5: A Feedforward Neural Network with One Hidden Layer (Biases shown explicitly)

5.4 Relation to Linear Regression

Note that if we use a **linear activation function**, e.g. $\phi(x) = x$, the neural network **collapses to a linear regression**

$$y \cong g(x;w) = \tilde{w}_0 + \sum_{i=1}^{N} \tilde{w}_i x_i$$

with appropriately defined regression coefficients $\tilde{w}$.

Recall that in our description of Figure 3.1 we argued that a machine learning algorithm would automatically turn the slider to find the best fit. This is exactly what the training algorithm has to do to train a neural network.

5.5 A Simple Example

Suppose we want to approximate $f(x) = \exp(x) - x^3$ with 3 neurons. The approximation might be

$$\hat{f}(x) = a_1 + a_2 - a_3$$

where

$$a_1 = max(0, -3x - 1.5)$$

$$a_2 = max(0, x + 1)$$

$$a_3 = max(0, 3x - 3)$$

Our neural network in this case uses ReLU activation functions and has all weights equal to one in the output layer. Figure 5.6 shows the admittedly poor approximation of $f(x)$ by $\hat{f}(x)$ using this neural network. Given the piecewise linear nature of the ReLU activation function and the low number of neurons, the approximation is not very good. However, with more neurons, we could get a better approximation.

The HTML version of these notes shows an interactive version of Figure 5.6 where you can adjust the weights of the neural network to approximate a simple dataset. As you can see there, it is quite tricky to find parameters that approximate the function well. This is where the training algorithm comes in. It will automatically adjust the weights to minimize a loss function.

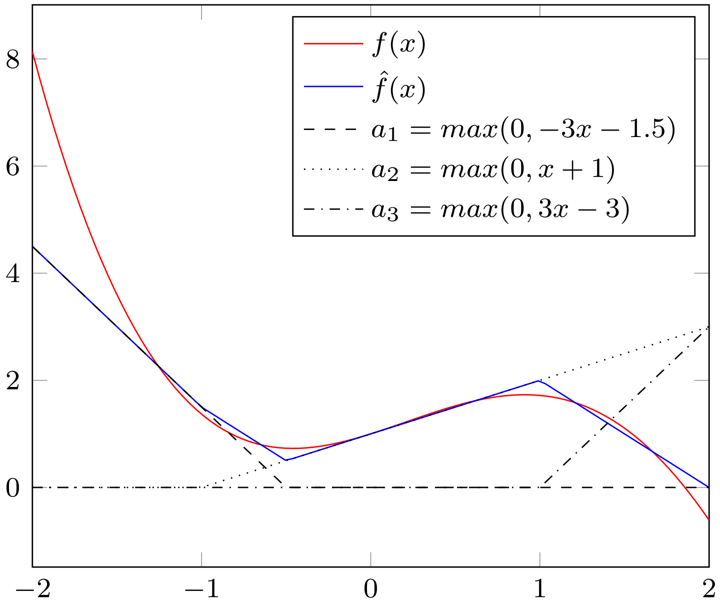
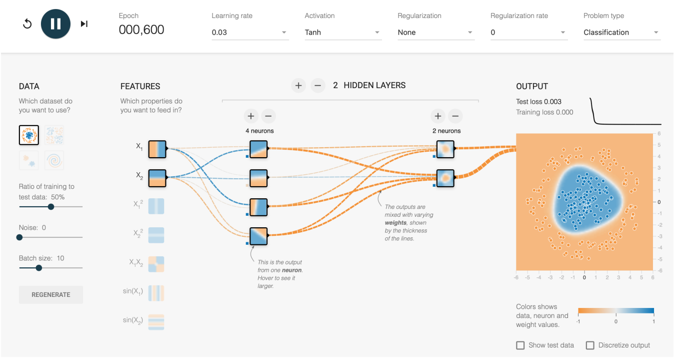


Figure 5.6: Approximation by a Neural Network

TensorFlow Playground

If you want to play around with neural networks, you can use the TensorFlow Playground: https://playground.tensorflow.org. It is a web-based tool that allows you to experiment with neural networks and see how they learn. Figure 5.7 shows the interface of the TensorFlow Playground.


Epoch 000,600

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

Which properties do you want to feed in?

Test loss 0.003

Training loss 0.000

This is the output from one **neuron**. Hover to see it larger.

The outputs are mixed with varying **weights**, shown by the thickness of the lines.

Colors shows data, neuron and weight values.

□ Show test data □ Discretize output

Figure 5.7: TensorFlow Playground

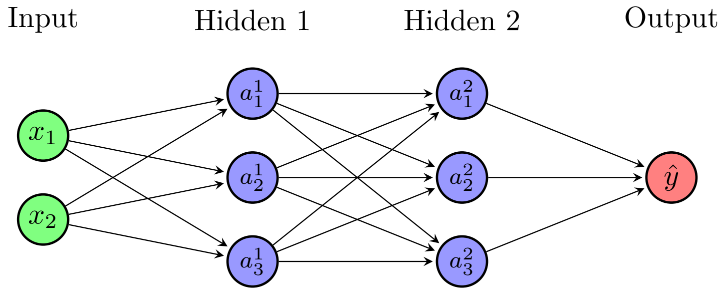


Figure 5.8: Deep Neural Network

5.6 Deep Neural Networks

Deep neural networks have more than one hidden layer. The number of hidden layers is also called the **depth** of the neural network. Deep neural networks can model more complex relationships. For simple function approximation, a single hidden layer is sufficient. Figure 5.8 shows a deep neural network with two hidden layers.

The first hidden layer consists of M_1 artificial neurons with inputs $x_1, x_2, \dots, x_N$

$$a^1_j = \phi\left(b^1_j + \sum_{i=1}^{N} w^1_{ji} x_i\right)$$

The second hidden layer consists of M_2 artificial neurons with inputs $a^1_1, a^1_2, \dots, a^1_{M_1}$

$$a^2_k = \phi\left(b^2_k + \sum_{j=1}^{M_1} w^2_{kj} a^1_j\right)$$

After Q hidden layers, the output is defined as

$$y \cong g(x; w) = b^{Q+1} + \sum_{j=1}^{M_Q} w^{Q+1}_j a^Q_j$$

Note that the activation functions do not need to be the same everywhere. In principle, we could vary the activation functions even within a layer.

5.7 Universal Approximation and the Curse of Dimensionality

Recall that we want to **approximate an unknown function** in supervised learning tasks

$$y = f(x)$$

where $y = (y_1, y_2, \dots, y_K)'$ and $x = (x_1, x_2, \dots, x_N)'$ are vectors. The function $f(x)$ could stand for many different functions in economics (e.g. a value function, a policy function, a conditional expectation, a classifier, ...).

It turns out that neural networks are **universal approximators** and **break the curse of dimensionality**. The universal approximation theorem by Hornik, Stinchcombe, and White (1989) states:

> A neural network with at least one hidden layer can approximate any Borel measurable function mapping finite-dimensional spaces to any desired degree of accuracy.

Breaking the curse of dimensionality (Barron, 1993)

> A one-layer NN achieves integrated square errors of order $O(1/M)$, where M is the number of nodes. In comparison, for series approximations, the integrated square error is of order $O(1/(M^{2/N}))$ where N is the dimensions of the function to be approximated.

5.8 Training a Neural Network: Determining Weights and Biases

We have not yet discussed how to determine the weights and biases. The weights and biases w are selected to **minimize a loss function**

$$E(w; X, Y) = \frac{1}{N} \sum_{n=1}^{N} E_n(w; x_n, y_n)$$

where N refers to the number of input-output pairs that we use for training and $E_n(w; x_n, y_n)$ refers to the loss of an individual pair n.

For notational simplicity, I will write $E(w)$ and $E_n(w)$ in the following or in some cases even omit argument w.

5.8.1 Choice of Loss Function

The choice of loss function depends on the problem at hand. In regressions, one often uses a **mean squared error (MSE) loss**

$$E_n(w; x_n, y_n) = \frac{1}{2} \left\| g\left(x_n; w\right) - y_n \right\|^2$$

In classification problems, one often uses a **cross-entropy loss**

$$E_n(w; x_n, y_n) = \sum_{k=1}^{K} y_{nk} \log(g_k(x_n; w))$$

where k refers to kth class (or kth element) in the output vector.

5.8.2 Gradient Descent

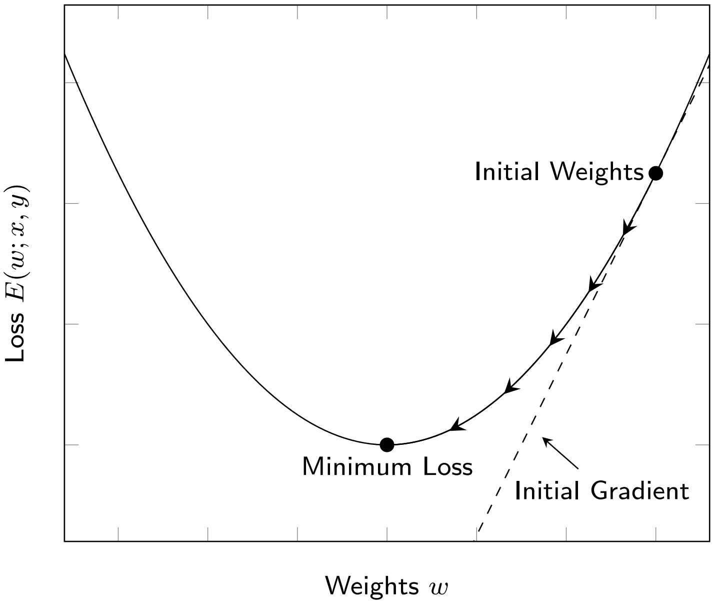


Figure 5.9: Gradient Descent

The weights and biases are determined by minimizing the loss function using a **gradient descent algorithm**. The basic idea is to compute how the loss changes with the weights w and step into the direction that reduces the loss. Figure 5.9 shows a simple example of a loss function and the gradient descent algorithm. The basic steps of the algorithm are

1. Initialize weights (e.g. draw from Gaussian distribution)

$$w^{(0)} \sim N(0, I)$$

2. Compute the gradient of the loss function with respect to weights

$$\nabla E(w^{(i)}) = \frac{1}{N} \sum_{n=1}^{N} \nabla E_n \left(w^{(i)}\right)$$

3. Update weights (make a small step in the direction of the negative gradient)

$$w^{(i+1)} = w^{(i)} - \eta \nabla E \left(w^{(i)}\right)$$

where $\eta > 0$ is the learning rate.

4. Repeat Steps 2 and 3 until a terminal condition (e.g. fixed number of iterations) is reached.

If we use the batch gradient descent algorithm described above, we might get stuck in a local minimum. To avoid this, we can use

- **Stochastic gradient descent:** Use only a single observation to compute the gradient and update the weights for each observation

$$w^{(i+1)} = w^{(i)} - \eta \nabla E_n \left(w^{(i)}\right)$$

- **Minibatch gradient descent:** Use a small batch of observations (e.g. 32) to compute the gradient and update the weights for each minibatch

These algorithms are less likely to get stuck in a shallow local minimum of the loss function because they are "noisier". Figure 5.10 shows a comparison of the different gradient descent algorithms. Minibatch gradient descent is probably the most commonly used and is also what we will be using in our implementation in Python.

5.8.3 Backpropagation Algorithm

Computing the gradient seems to be a daunting task since a weight in the first layer in a deep neural network affects the loss function potentially through thousands of "paths". The **backpropagation algorithm** (Rumelhart et al., 1986) provides an efficient way to evaluate the gradient. The basic idea is to go backward through the network to evaluate the gradient as shown in Figure 5.11. If you are interested in the details, I recommend reading the notes by Nielsen (2019).

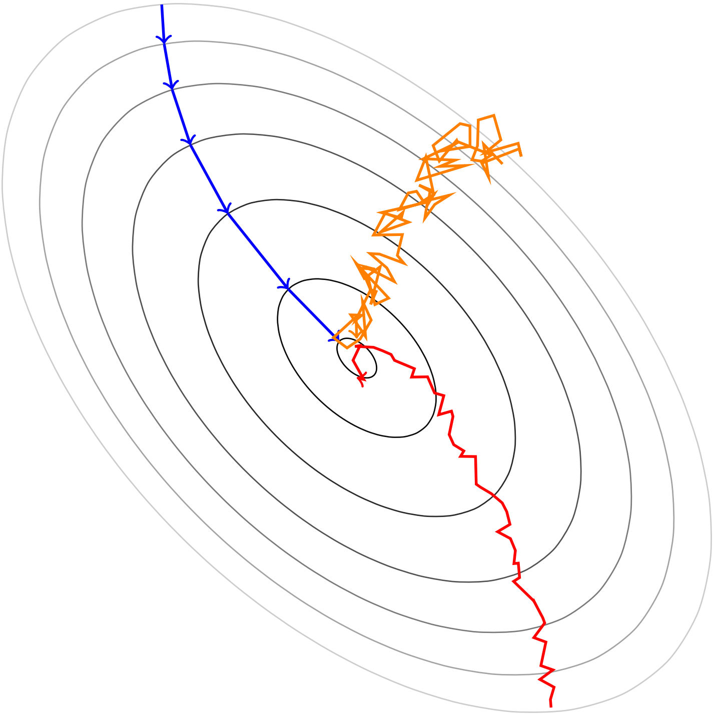


Figure 5.10: Comparison of Gradient Descent Types (blue: Full Batch, red: Minibatch, orange: Stochastic)

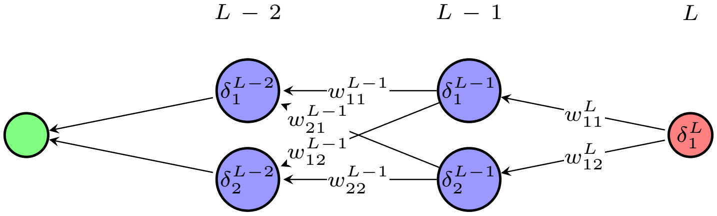


Figure 5.11: Backpropagation Algorithm

5.9 Practical Considerations

From a practical perspective, there are many more things to consider. Often times it's beneficial to do some (or all) of the following

- *Input/output normalization:* (e.g. to have unit variance and mean zero) can improve the performance of the NN
- *Check for overfitting:* by splitting the dataset into a *training dataset* and a *test dataset*
- *Regularization:* to avoid overfitting (e.g., add a term to loss function that penalizes large weights)
- *Adjust the learning rate:* η during training

We have already discussed some of these topics in the context of other machine learning algorithms.

5.10 More Advanced Neural Architectures

Neural networks come in many different architectures. So far, we have only discussed **feedforward neural networks**. Other popular architectures include

- **Convolutional neural networks (CNNs):** Designed for grid-like data such as images. CNNs use convolutional layers that apply small filters sliding across the input to detect local patterns like edges and textures. Early layers detect simple features, while deeper layers combine these into complex patterns (e.g., faces, objects). Applications include image classification, medical imaging, facial recognition, and self-driving cars.
- **Recurrent neural networks (RNNs):** Designed for sequential data where order matters. Unlike feedforward networks, RNNs have connections that loop back, maintaining a "memory" of previous inputs through a hidden state. Variants like Long Short-Term Memory (LSTM) networks address the "vanishing gradient" problem for long sequences. Applications include time series forecasting, speech recognition, and sentiment analysis.
- **Transformer networks:** Use an attention mechanism that allows the model to weigh the importance of different parts of the input when producing each output. Unlike RNNs, Transformers process all positions simultaneously, making them faster to train. Applications include large language models (GPT, Claude), machine translation, and even protein structure prediction (AlphaFold).

We will have a closer look at transformer networks in the chapter on generative AI since transformers were one of the key innovations that made the recent advances in this field possible.

5.11 Python Implementation

Let's have a look at how to implement a neural network in Python.

5.11.1 Implementing the Feedforward Part of a Neural Network

As a small programming exercise and to improve our understanding of neural networks, let's implement the feedforward part of a neural network from scratch. We will have to calculate the output of the network for some given weights and biases, as well as some inputs. Let's start by importing the necessary libraries

```python
import numpy as np
```

Next, we define the activation function for which we use the sigmoid function

```python
def activation_function(x):
    return 1/(1+np.exp(-x)) # sigmoid function
```

Now, we define the feedforward function which calculates the output of the neural network given some inputs, weights, and biases. The function takes the inputs, weights, and biases as arguments and returns the output of the network

```python
def feedforward(inputs, w1, w2, b1, b2):

    # Compute the pre-activation values for the first layer
    z = b1 + np.matmul(w1, inputs)

    # Compute the post-activation values for the first layer
    a = activation_function(z)

    # Combine the post-activation values of the first layer to an output
    g = b2 + np.matmul(w2, a)

    return g
```

Mathematically, the function computes the following

$$z = b^1 + w^1 x$$

$$a = \phi(z)$$

$$g = b^2 + w^2 a$$

and returns g at the end. We have written this using matrix notation to make it more compact. Remember that node j in the hidden layer is given by

$$z_j = b_j^1 + \sum_{i=1}^{N} w_{ji}^1 x_i$$

$$a_j = \phi(z_j)$$

and the output of the network is given by

$$g(x; w) = b^2 + \sum_{j=1}^{M} w_j^2 a_j.$$

Let's test the function with some example inputs, weights and biases

```
# Define the weights and biases
w1 = np.array([[0.1, 0.2], [0.3, 0.4]]) # 2x2 matrix
w2 = np.array([0.5, 0.6]) # 1-d vector
b1 = np.array([0.1, 0.2]) # 1-d vector
b2 = 0.3

# Define the inputs
inputs = np.array([1, 2]) # 1-d vector

# Compute the output of the network
feedforward(inputs, w1, w2, b1, b2)
```

```
np.float64(1.0943291429384328)
```

To operationalize this, we would also need to define a loss function and an optimization algorithm to update the weights and biases. However, this is beyond the scope of this course.

5.11.2 Using Neural Networks in scikit-learn

Scikit-learn provides a simple interface to use neural networks. However, it is not as flexible as the more commonly used PyTorch or TensorFlow. We can reuse the **dataset of credit card transactions** from Kaggle.com to demonstrate how to use neural networks in scikit-learn.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, roc_auc_score, recall_score, precision_score, roc_curve
pd.set_option('display.max_columns', 50) # Display up to 50 columns
from io import BytesIO
from urllib.request import urlopen
from zipfile import ZipFile
import os.path

# Check if the file exists
if not os.path.isfile('data/card_transdata.csv'):

    print('Downloading dataset...')

    # Define the dataset to be downloaded
    zipurl = 'https://www.kaggle.com/api/v1/datasets/download/dhanushnarayananr/credit-card-fraud'

    # Download and unzip the dataset in the data folder
    with urlopen(zipurl) as zipresp:
        with ZipFile(BytesIO(zipresp.read())) as zfile:
            zfile.extractall('data')

    print('DONE!')

else:

    print('Dataset already downloaded!')
```

```
Dataset already downloaded!
```

```
# Load the data
df = pd.read_csv('data/card_transdata.csv')

# Split the data into training and test sets
X = df.drop('fraud', axis=1) # All variables except `fraud`
y = df['fraud'] # Only our fraud variables
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
↪   test_size = 0.3, random_state = 42)

# Scale the features
def scale_features(scaler, df, col_names, only_transform=False):

    # Extract the features we want to scale
    features = df[col_names]

    # Fit the scaler to the features and transform them
    if only_transform:
        features = scaler.transform(features.values)
    else:
        features = scaler.fit_transform(features.values)

    # Replace the original features with the scaled features
    df[col_names] = features

col_names = ['distance_from_home', 'distance_from_last_transaction',
↪   'ratio_to_median_purchase_price']
scaler = StandardScaler()
scale_features(scaler, X_train, col_names)
scale_features(scaler, X_test, col_names, only_transform=True)
```

Recall that the target variable y is `fraud`, which indicates whether the transaction is fraudulent or not. The other variables are the features x of the transactions.

To use a neural network for a classification task, we can use the `MLPClassifier` class from scikit-learn. The following code snippet shows how to use a neural network with one hidden layer with 16 nodes

```
clf = MLPClassifier(hidden_layer_sizes=(16,), random_state=42,
↪   verbose=False).fit(X_train, y_train)
```

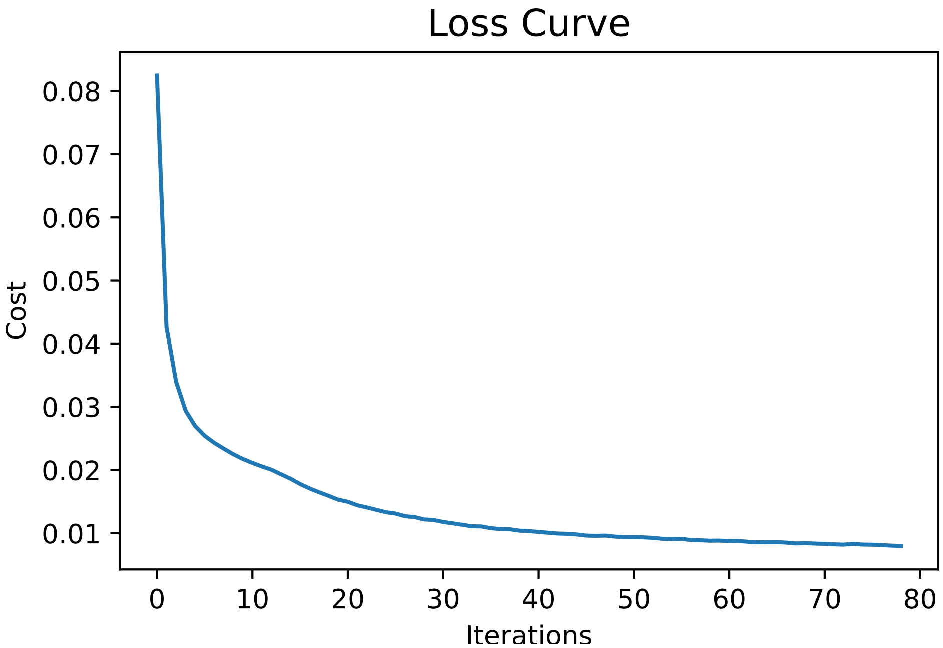
If you would like to use a neural network with multiple hidden layers, you can specify the number of nodes per hidden layer using the `hidden_layer_sizes` parameter. For example, the following code snippet shows how to use a neural network with two hidden layers, one with 5 nodes and the other with 4 nodes

```
clf = MLPClassifier(alpha=1e-5, hidden_layer_sizes=(5,4),
 ↪  activation='logistic', random_state=42).fit(X_train, y_train)
```

Note that the `alpha` parameter specifies the regularization strength, the `activation` parameter specifies the activation function (by default it uses `relu`) and the `random_state` parameter specifies the seed for the random number generator (useful for reproducible results).

We can check the loss curve to see how the neural network loss declined during training

```
plt.plot(clf.loss_curve_)
plt.title("Loss Curve", fontsize=14)
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.show()
```



We can then use the same way to evaluate the neural network performance as we did for the other ML models

```
y_pred = clf.predict(X_test)
y_proba = clf.predict_proba(X_test)

print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
```

```
Accuracy: 0.99748
```

```
print(f"Precision: {precision_score(y_test, y_pred)}")
```

```
Precision: 0.9916592655519945
```

```
print(f"Recall: {recall_score(y_test, y_pred)}")
```

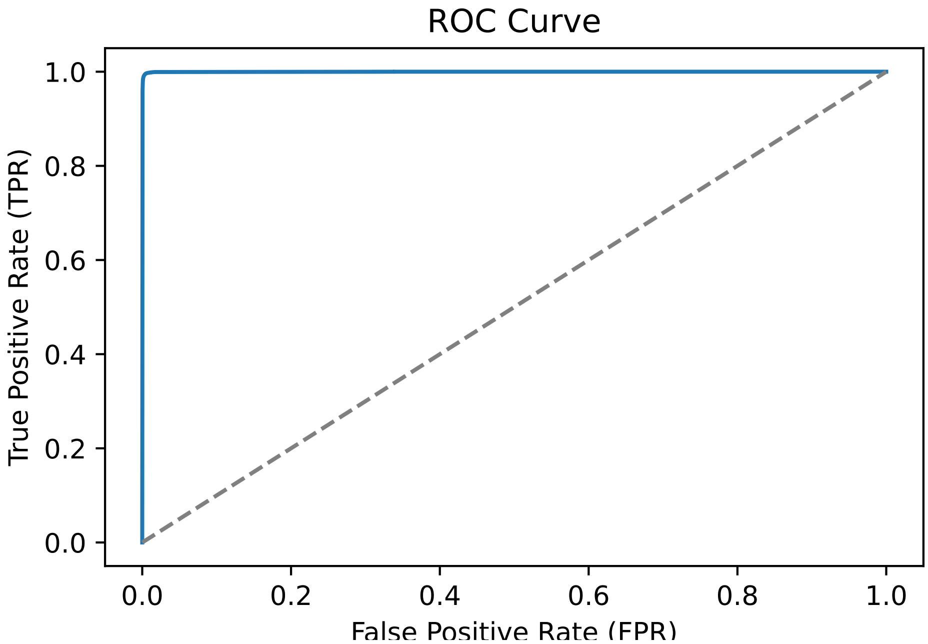
```
Recall: 0.9794058197627855
```

```
print(f"ROC AUC: {roc_auc_score(y_test, y_proba[:, 1])}")
```

```
ROC AUC: 0.9998789639015943
```

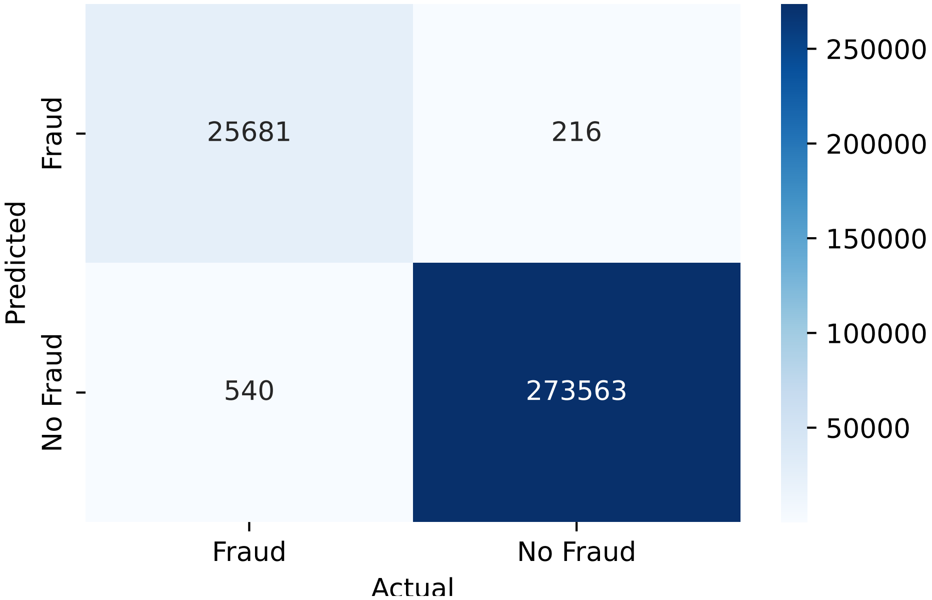
The neural network performs substantially better than the logistic regression. As in the case of the tree-based methods, the ROC AUC score is much closer to the maximum value of 1 and we have an almost perfect classifier

```
# Compute the ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_proba[:, 1])

# Plot the ROC curve
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1], linestyle='--', color='grey')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve')
plt.show()
```



Let's also check the confusion matrix to see where we still make mistakes

```
conf_mat = confusion_matrix(y_test, y_pred, labels=[1, 0]).transpose() #
 ↪  Transpose the sklearn confusion matrix to match the convention in the
 ↪  lecture
sns.heatmap(conf_mat, annot=True, cmap='Blues', fmt='g',
 ↪  xticklabels=['Fraud', 'No Fraud'], yticklabels=['Fraud', 'No Fraud'])
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.show()
```



There are around 270 false negatives, i.e., a fraudulent transaction that we did not detect. There are also around 980 false positives, i.e., "false alarms", where non-fraudulent transactions were classified as fraudulent.

5.11.3 Using Neural Networks in PyTorch

While it is possible to use neural networks in scikit-learn, it is more common to use PyTorch or TensorFlow for neural networks. PyTorch is a popular deep-learning library that is widely used in academia and industry. In this section, we will show how to use PyTorch to build a simple neural network for the same credit card fraud detection task.

> ⚠ Feel Free to Skip This Section
>
> This section might be a bit more challenging than what we have looked at previously. If you think that you are not ready for this, feel free to skip this section. This is mainly meant to be a starting point for those who are interested in learning more about neural networks.
> For a more in-depth introduction to PyTorch, I recommend that you check out the official PyTorch tutorials. This section, in particular, builds on the Learning PyTorch with Examples tutorial.

Let's start by importing the necessary libraries

```
import torch
from torch.utils.data import DataLoader, TensorDataset
```

Then, let's prepare the data for PyTorch. We need to convert the data in our DataFrame to PyTorch tensors

```
X_train_tensor = torch.tensor(X_train.values, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
```

Note that we also converted the input values to `float32` for improved training speed and the target values to `long` which is a type of integer (remember our target `y` can only take values zero or one). Next, we need to create a `DataLoader` object to load the data in mini-batches during the training process

```
dataset = TensorDataset(X_train_tensor, y_train_tensor)
dataloader = DataLoader(dataset, batch_size=200, shuffle=True)
dataset_size = len(dataloader.dataset)
```

Next, we define the neural network model using the `nn` module from PyTorch

```
model = torch.nn.Sequential(
    torch.nn.Linear(7, 16), # 7 input features, 16 nodes in the hidden layer
    torch.nn.ReLU(),        # ReLU activation function
    torch.nn.Linear(16, 2) # 16 nodes in the hidden layer, 2 output nodes
 ↪  (fraud or no fraud)
)
```

We also need to define the loss function and the optimizer. We will use the cross-entropy loss function and the Adam optimizer

```
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-5)
 ↪  # Adam optimizer with learning rate of 0.001 and L2 regularization
 ↪  (analogous to alpha in scikit-learn)
```

We can now train the neural network using the following code snippet

```
for epoch in range(80):

    # Loop over batches in an epoch using DataLoader
    for id_batch, (X_batch, y_batch) in enumerate(dataloader):

        # Compute the predicted y using the neural network model with the
         ↪  current weights
        y_batch_pred = model(X_batch)

        # Compute the loss
        loss = loss_fn(y_batch_pred, y_batch)

        # Reset the gradients of the loss function to zero
        optimizer.zero_grad()

        # Compute the gradient of the loss with respect to model parameters
        loss.backward()

        # Update the weights by taking a "step" in the direction that reduces
         ↪  the loss
        optimizer.step()

    if epoch % 10 == 9:
        print(f"Epoch {epoch} loss: {loss.item():>7f}")
```

```
Epoch 9 loss: 0.013850
Epoch 19 loss: 0.013232
Epoch 29 loss: 0.003188
Epoch 39 loss: 0.001493
Epoch 49 loss: 0.008496
Epoch 59 loss: 0.012090
Epoch 69 loss: 0.008941
Epoch 79 loss: 0.006905
```

Note that here we are updating the model weights for each mini-batch in the dataset and go over the whole dataset 80 times (epochs). We print the loss every 10 epochs to see how the loss decreases over time.

The following snippet shows how to use full-batch gradient descent instead of mini-batch gradient descent

```
for epoch in range(2000):
    # Compute the predicted y using the neural network model with the current
     ↪  weights
    y_epoch_pred = model(X_train_tensor)

    # Compute the loss
    loss = loss_fn(y_epoch_pred, y_train_tensor)

    # Reset the gradients of the loss function to zero
    optimizer.zero_grad()

    # Compute the gradient of the loss with respect to model parameters
    loss.backward()

    # Update the weights by taking a "step" in the direction that reduces the
     ↪  loss
    optimizer.step()

    # Print the loss every 100 epochs
    if epoch % 100 == 99:
        print(f"Epoch {epoch} loss: {loss.item():>7f}")
```

Note that in this version we are updating the model weights 2000 times (epochs) and printing the loss every 100 epochs. We can now evaluate the model on the test set

```
X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32)
y_pred = torch.argmax(model(X_test_tensor), dim=1).numpy()

print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
```

```
Accuracy: 0.99572
```

```
print(f"Precision: {precision_score(y_test, y_pred)}")
```

```
Precision: 0.9862719862719863
```

```
print(f"Recall: {recall_score(y_test, y_pred)}")
```

```
Recall: 0.9644559704054002
```

Note that for simplicity we are reusing the scikit-learn metrics to evaluate the model.

However, our neural network trained in PyTorch does not perform exactly the same as the neural network trained in scikit-learn. This is likely because of different hyperparameters or different initializations of the weights. In practice, it is common to experiment with different hyperparameters to find the best model or to use grid search and cross-validation to try many values and find the best-performing ones.

5.11.4 Conclusions

In this section, we have learned about neural networks, which are the foundation of deep learning. We have seen how to implement parts of a simple neural network from scratch and how to use neural networks in scikit-learn and PyTorch.

Chapter 6

Practice Session I

The application in this practice session is inspired by the empirical example in "Measuring the model risk-adjusted performance of machine learning algorithms in credit default prediction" by Alonso Robisco and Carbó Martínez (2022). However, since we are not interested in model risk-adjusted performance, the application will purely focus on the implementation of machine learning algorithms for loan default prediction.

6.1 Problem Setup

The dataset that we will be using was used in the Kaggle competition "Give Me Some Credit". The description of the competition reads as follows:

> Banks play a crucial role in market economies. They decide who can get finance and on what terms and can make or break investment decisions. For markets and society to function, individuals and companies need access to credit.
>
> Credit scoring algorithms, which make a guess at the probability of default, are the method banks use to determine whether or not a loan should be granted. This competition requires participants to improve on the state of the art in credit scoring, by predicting the probability that somebody will experience financial distress in the next two years.
>
> The goal of this competition is to build a model that borrowers can use to help make the best financial decisions.
>
> Historical data are provided on 250,000 borrowers and the prize pool is $5,000 ($3,000 for first, $1,500 for second and $500 for third).

Unfortunately, there won't be any prize money today. However, the experience that you can gain from working through an application like this can be invaluable. So, in a way, you are still winning!

6.2 Dataset

Let's download the dataset automatically, unzip it, and place it in a folder called `data` if you haven't done so already

```
from io import BytesIO
from urllib.request import urlopen
from zipfile import ZipFile
import os.path

# Check if the file exists
if not os.path.isfile('data/Data Dictionary.xls') or not
 ↪  os.path.isfile('data/cs-training.csv'):

    print('Downloading dataset...')

    # Define the dataset to be downloaded
    zipurl = 'https://www.kaggle.com/api/v1/datasets/download/brycecf/give-m⌋
 ↪  e-some-credit-dataset'

    # Download and unzip the dataset in the data folder
    with urlopen(zipurl) as zipresp:
        with ZipFile(BytesIO(zipresp.read())) as zfile:
            zfile.extractall('data')

    print('DONE!')

else:

    print('Dataset already downloaded!')
```

```
Dataset already downloaded!
```

Then, we can have a look at the data dictionary that is provided with the dataset. This will give us an idea of the variables that are available in the dataset and what they represent

```
import pandas as pd
data_dict = pd.read_excel('data/Data Dictionary.xls', header=1)
data_dict.style.hide()
```

Table 6.1

Variable Name	Description
SeriousDlqin2yrs	Person experienced 90 days past due delinquency or worse
RevolvingUtilizationOfUnsecuredLines	Total balance on credit cards and personal lines of credit except
age	Age of borrower in years
NumberOfTime30-59DaysPastDueNotWorse	Number of times borrower has been 30-59 days past due but no
DebtRatio	Monthly debt payments, alimony,living costs divided by monthy
MonthlyIncome	Monthly income
NumberOfOpenCreditLinesAndLoans	Number of Open loans (installment like car loan or mortgage) a
NumberOfTimes90DaysLate	Number of times borrower has been 90 days or more past due.
NumberRealEstateLoansOrLines	Number of mortgage and real estate loans including home equit
NumberOfTime60-89DaysPastDueNotWorse	Number of times borrower has been 60-89 days past due but no
NumberOfDependents	Number of dependents in family excluding themselves (spouse,

The variable y that we want to predict is `SeriousDlqin2yrs` which indicates whether a person has been 90 days past due on a loan payment (serious delinquency) in the past two years. This target variable is 1 if the loan defaults (i.e., serious delinquency occurred) and 0 if the loan does not default (i.e., no serious delinquency occurred). The other variables are features that we can use to predict this target variable such as the age of the borrower and the monthly income of the borrower.

6.3 Putting the Problem into the Context of the Course

Given the description of the competition and the dataset, we can see that this is a **supervised learning problem**. We have a target variable that we want to predict, and we have features that we can use to predict this target variable. The target variable is binary, i.e., it can take two values: 0 or 1. The value 0 indicates that the loan will not default, while the value 1 indicates that the loan will default. Thus, this is a **binary classification problem**.

6.4 Setting up the Environment

We will start by setting up the environment by importing the necessary libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

and loading the dataset

```
df = pd.read_csv('data/cs-training.csv')
```

6.5 Exercises

Note that the exercises build on each other. You can sometimes skip exercises but the results for later exercises will depend on the previous ones. If you get stuck, you can skip to the next exercise and try to come back to the previous one later.

6.5.1 Exercise 1: Familiarization with the Dataset

Tasks:

1. Display the first 5 rows of the dataset. What do you notice about the column names?
2. There appears to be an unnecessary index column. Identify it and remove it from the DataFrame
3. Use `.info()` to check the data types and identify which columns have missing values

Hints:

- The `.head()` method shows the first rows
- Look for columns that seem to duplicate the index
- The `axis` parameter in `.drop()` specifies whether you're dropping rows or columns

```
# Your code here. Add additional code cells as needed.
```

6.5.2 Exercise 2: Understanding the Target Variable

Tasks:

1. What is the proportion of defaulted vs non-defaulted loans in the dataset? Use `value_counts(normalize=True)`
2. Based on this distribution, would you say the dataset is balanced or imbalanced?
3. Why might class imbalance be problematic for machine learning? What evaluation metrics should we be careful about?

Hints:

- The target variable is `SeriousDlqin2yrs`

- Think about what accuracy would be if a model just predicted the majority class

```
# Your code here. Add additional code cells as needed.
```

6.5.3 Exercise 3: Handling Missing Values and Data Quality Issues

Tasks:

1. Use `.dropna()` combined with `value_counts()` to check if dropping missing values significantly changes the target variable distribution
2. Drop the rows with missing values for the rest of the exercises.
3. How many rows were dropped due to missing values?
4. Verify that there are no missing values remaining in the dataset.
5. Check for duplicate rows. How many are there? Should you remove them?

Hints:

- Use `df.loc[df.isna().any(axis=1)]` to select rows with any missing values
- Pay attention to the mean and standard deviation differences

> ℹ Note
>
> Note that in a real application, you would want to carefully consider how to handle missing data rather than just dropping rows. Imputation methods or models that can handle missing data directly might be more appropriate depending on the context. Furthermore, in this specific dataset, dropping some of the missing values also removes some of the other data quality issues by chance. In practice, you would want to investigate and address these issues separately.

```
# Your code here. Add additional code cells as needed.
```

6.5.4 Exercise 4: Exploratory Data Analysis

Tasks:

1. Create a pie chart (or histogram) showing the distribution of the target variable in your cleaned dataset
2. Generate a pair plot for `age`, `MonthlyIncome`, `DebtRatio`, and `SeriousDlqin2yrs` using seaborn's `pairplot()` with `hue='SeriousDlqin2yrs'`
3. Calculate and visualize correlation matrices using a heatmap
4. Which features appear most correlated with loan default?

5. Are there any features that are highly correlated with each other? What issues could this cause?

Hints:

- Use `sns.pairplot()` with the `hue` parameter for coloring by class
- Use `df.corr()` for Pearson correlation
- `sns.heatmap()` can visualize correlation matrices
- Use `np.triu()` to create a mask for the upper triangle

```
# Your code here. Add additional code cells as needed.
```

6.5.5 Exercise 5: Preparing Data for Machine Learning Algorithms

Tasks:

1. Separate features (`X`) from the target variable (`y`)
2. Split the data into training (80%) and test (20%) sets using `train_test_split`. Use `stratify=y` to maintain class proportions and `random_state=42` for reproducibility
3. Apply `MinMaxScaler` to normalize the features. **Important:** Fit the scaler only on training data, then transform both training and test data

Hints:

- Use `df.drop('column_name', axis=1)` for features
- The `stratify` parameter ensures balanced splits
- Fitting on test data causes "data leakage" - avoid this!
- Create a helper function for scaling if you want cleaner code

```
# Your code here. Add additional code cells as needed.
```

6.5.6 Exercise 6: Defining Evaluation Metrics

Tasks:

1. Write a function `evaluate_model(clf, X_train, y_train, X_test, y_test, label='')` that:

    - Computes predictions and predicted probabilities
    - Prints Accuracy, Precision, Recall, and ROC AUC for both training and test sets
    - Plots the ROC curve for both training and test sets

2. Why is it important to evaluate on both training and test data?

3. Given our imbalanced dataset, which metric(s) should we focus on and why?

Hints:

- Use `clf.predict()` for class predictions and `clf.predict_proba()` for probabilities
- Import metrics from `sklearn.metrics`: `accuracy_score`, `precision_score`, `recall_score`, `roc_auc_score`, `roc_curve`
- Plot both curves on the same figure for comparison
- Add a diagonal reference line for the ROC plot
- Use `label` parameter to differentiate models in outputs, e.g., `label='Logistic Regression'`

```
# Your code here. Add additional code cells as needed.
```

6.5.7 Exercise 7: Training Classification Models

Tasks:

Train the following models and evaluate each using your evaluation function:

1. **Logistic Regression**: Use `penalty=None`, `solver='lbfgs'`, `max_iter=5000`
2. **Decision Tree**: Use `max_depth=7`
3. **Random Forest**: Use `max_depth=20`, `n_estimators=100`
4. **XGBoost**: Use `max_depth=5`, `n_estimators=40`, `random_state=0`
5. **Neural Network**: Use `MLPClassifier` with `activation='relu'`, `solver='adam'`, `hidden_layer_sizes=(300, 200, 100)`, `max_iter=300`, `random_state=42`

For each model:

- Fit on training data
- Evaluate using your evaluation function
- Note the training vs test performance

Hints:

- Import from: `sklearn.linear_model`, `sklearn.tree`, `sklearn.ensemble`, `xgboost`
- Use `.fit(X_train, y_train)` to train each model
- Watch for signs of overfitting (training » test performance)
- Training the neural network may take several minutes

```
# Your code here. Add additional code cells as needed.
```

6.5.8 Exercise 8: Results Comparison

Tasks:

1. Create a DataFrame comparing all models with columns: Model, ROC AUC (Train), ROC AUC (Test)
2. Which model performed best on the test set?
3. Which model showed the largest gap between training and test performance? What does this suggest?

Hints:

- Use `pd.DataFrame()` with a dictionary
- The gap between train/test performance indicates overfitting

```
# Your code here. Add additional code cells as needed.
```

6.5.9 Exercise 9: Feature Engineering

Tasks:

1. Create squared versions of all features and add them to the dataset (use `.pow(2)` and `.add_suffix('_sq')`)
2. Re-split and re-scale the data with the new features
3. Retrain all models with the expanded feature set
4. Compare the new results with the original. Did feature engineering help?

Optional: Add a Logistic Regression with L1 (LASSO) penalty using `penalty='l1'` and `solver='liblinear'`. How does it perform?

Hints:

- Use `X.assign(**X.pow(2).add_suffix('_sq'))` for compact feature creation
- Remember to fit a new scaler on the new training data
- LASSO can help with feature selection when you have many features

```
# Your code here. Add additional code cells as needed.
```

6.5.10 Exercise 10: Reflection and Discussion

Tasks:

1. What additional steps could improve model performance (e.g., hyperparameter tuning, handling class imbalance, more feature engineering)?
2. In a real banking context, would you prefer a model with higher precision or higher recall? Why?
3. What are the ethical considerations when deploying such a model for loan decisions?

Hints:

- No code required; reflect on practical and ethical aspects

Part III

Natural Language Processing

Chapter 7

Overview of Natural Language Processing (NLP)

Chapter 8

Classical NLP Approaches

Chapter 9

Practice Session II

Part IV

Generative AI

Chapter 10

Overview of Generative AI

Chapter 11

Large Language Models

Chapter 12

Practice Session III

References

Alonso Robisco, Andrés, and José Manuel Carbó Martínez. 2022. "Measuring the model risk-adjusted performance of machine learning algorithms in credit default prediction." *Financial Innovation* 8 (1). https://doi.org/10.1186/s40854-022-00366-1.

Aruoba, S. Boragan, and Thomas Drechsel. 2022. "Identifying Monetary Policy Shocks: A Natural Language Approach." CEPR Discussion Paper DP17133. CEPR.

Athey, Susan, and Guido W. Imbens. 2019. "Machine Learning Methods That Economists Should Know About." *Annual Review of Economics* 11: 685–725. https://doi.org/10.1146/annurev-economics-080217-053433.

Bank for International Settlements. 2021. "Machine learning applications in central banking." IFC Bulletin 57. https://www.bis.org/ifc/publ/ifcb57.pdf.

———. 2025. "Governance and implementation of artificial intelligence in central banks." IFC Report 18. https://www.bis.org/ifc/publ/ifc_report_18.pdf.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning.* Edited by Michael Jordan, Jon Kleinberg, and Bernhard Schölkopf. Information Science and Statistics. Springer Science+Business Media, LLC. https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. 2020. "Language Models are Few-Shot Learners." Technical Report arXiv:2005.14165. https://doi.org/10.48550/arXiv.2005.14165.

Dell, Melissa. 2025. "Deep Learning for Economists." *Journal of Economic Literature* 63 (1): 5–58. https://doi.org/10.1257/jel.20241733.

Fernández-Villaverde, Jesús, Samuel Hurtado, and Galo Nuño. 2023. "Financial Frictions and the Wealth Distribution." *Econometrica* 91 (3): 869–901. https://doi.org/10.3982/ecta18180.

Fernández-Villaverde, Jesús, Joël Marbet, Galo Nuño, and Omar Rachedi. 2024. "Inequality and the Zero Lower Bound." Working Paper 2407. Banco de España.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press. http://www.deeplearningbook.org.

Gorodnichenko, Yuriy, Tho Pham, and Oleksandr Talavera. 2023. "The Voice of Monetary Policy." *American Economic Review* 113 (2): 548–84. https://doi.org/10.1257/aer.20220129.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning - Data Mining, Inference, and Prediction.* Second Edition. Springer.

Kaji, Tetsuya, Elena Manresa, and Guillaume Pouliot. 2023. "An Adversarial Approach to

Structural Estimation." *Econometrica* 91 (6): 2041–63. https://doi.org/10.3982/ecta18707.

Kase, Hanno, Leonardo Melosi, and Matthias Rottner. 2022. "Estimating Nonlinear Heterogeneous Agents Models with Neural Networks." Federal Reserve Bank of Chicago. https://doi.org/10.21033/wp-2022-26.

Korinek, Anton. 2023. "Generative AI for Economic Research: Use Cases and Implications for Economists." *Journal of Economic Literature* 61 (4): 1281–1317. https://doi.org/10.1257/jel.20231736.

Laney, Doug. 2001. "3D Data Management: Controlling Data Volume, Velocity, and Variety." Research Note G00158589. Meta Group.

Maliar, Lilia, Serguei Maliar, and Pablo Winant. 2021. "Deep learning for solving dynamic economic models." *Journal of Monetary Economics* 122 (September): 76–101. https://doi.org/10.1016/j.jmoneco.2021.07.004.

McCulloch, Warren S., and Walter Pitts. 1943. "A logical calculus of the ideas immanent in nervous activity." *The Bulletin of Mathematical Biophysics* 5 (4): 115–33. https://doi.org/10.1007/bf02478259.

McKinney, Wes. 2022. *Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter.* Third Edition. O'Reilly Media. https://wesmckinney.com/book/.

Microsoft. 2024. "Deep learning vs. machine learning in Azure Machine Learning." Website. https://learn.microsoft.com/en-us/azure/machine-learning/concept-deep-learning-vs-machine-learning?view=azureml-api-2.

Mitchell, Tom. 1997. *Machine Learning.* McGraw Hill. https://www.cs.cmu.edu/~tom/mlbook.html.

Murphy, Kevin P. 2012. *Machine Learning: A Probabilistic Perspective.* Cambridge: MIT Press. https://probml.github.io/pml-book/book0.html.

———. 2022. *Probabilistic Machine Learning: An Introduction.* MIT Press. https://probml.github.io/pml-book/book1.html.

———. 2023. *Probabilistic Machine Learning: Advanced Topics.* MIT Press. https://probml.github.io/pml-book/book2.html.

Nielsen, Michael. 2019. *Neural Networks and Deep Learning.* http://neuralnetworksanddeeplearning.com.

Rosenblatt, F. 1958. "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological Review* 65 (6): 386–408. https://doi.org/10.1037/h0042519.

Schmidhuber, Jürgen. 2022. "Annotated History of Modern AI and Deep Learning." Technical Report IDSIA-22-22. https://doi.org/10.48550/arXiv.2212.11279.

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction.* Second Edition. MIT Press. http://incompleteideas.net/book/the-book-2nd.html.

Varian, Hal R. 2014. "Big Data: New Tricks for Econometrics." *Journal of Economic Perspectives* 28 (2): 3–28. https://doi.org/10.1257/jep.28.2.3.

Appendix A

Unsupervised Learning

In this chapter, we will introduce some unsupervised learning methods that are commonly used in machine learning.

A.1 K-means Clustering

K-means is a method that is used for finding clusters in a set of unlabeled data meaning that it is an **unsupervised learning** method. For the algorithm to work, one has to choose a fixed number of clusters K for which the algorithm will then try to find the cluster centers (i.e., the means) using an iterative procedure. The **basic algorithm** proceeds as follows given a set of initial guesses for the K cluster centers:

1. Assign each data point to the nearest cluster center
2. Recompute the cluster centers as the mean of the data points assigned to each cluster

The algorithm iterates over these two steps until the cluster centers do not change or the change is below a certain threshold. As an **initial guess**, one can use, for example, K randomly chosen observations as cluster centers.

We need some **measure of disimilarity** (or distance) to assign data points to the nearest cluster center. The most common choice is the Euclidean distance. The squared Euclidean distance between two points x and y in p-dimensional space is defined as

$$d(x_i, x_j) = \sum_{n=1}^{p} (x_{in} - x_{jn})^2 = \|x_i - x_j\|^2$$

where x_{in} and x_{jn} are the n-th feature of the i-th and j-th observation in our dataset, respectively.

The objective function of the K-means algorithm is to minimize the sum of squared distances between the data points and their respective cluster centers

$$\min_{C,\{m_k\}_{k=1}^{K}} \sum_{k=1}^{K} \sum_{C(i)=k} \|x_i - m_k\|^2$$

where second sum sums up over all elements i in cluster k and μ_k is the cluster center of cluster k.

The K-means algorithm is **sensitive to the initial choice of cluster centers**. To mitigate this, one can run the algorithm multiple times with different initial guesses and choose the solution with the smallest objective function value.

The scale of the data can also have an impact on the clustering results. Therefore, it is often recommended to **standardize the data** before applying the K-means algorithm. Furthermore, the Euclidean distance is **not well suited for binary or categorical data**. Therefore, one should only use the K-means algorithm for continuous data.

How to choose the number of clusters K? One can use the so-called **elbow method** to find a suitable number of clusters. The elbow method plots the sum of squared distances (i.e., the objective function of K-means) for different K. The idea is to choose the number of clusters at the “elbow” of the curve, i.e., the point where the curve starts to flatten out. Note that the curve starts to flatten out when adding more clusters does not significantly reduce the sum of squared distances anymore. This usually happens to be the case when the number of clusters exceeds the “true” number of clusters in the data. However, this is just a heuristic and it might not always be easy to identify the “elbow” in the curve.

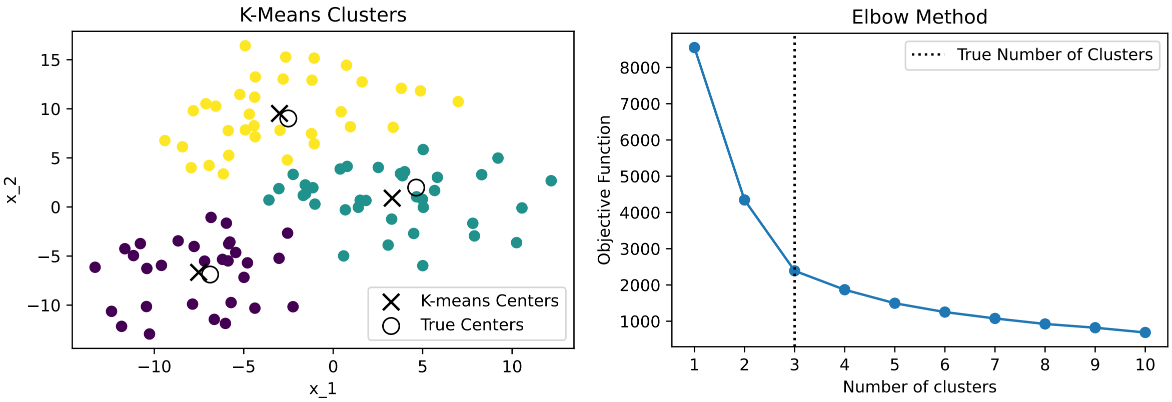


Figure A.1: K-Means Clusters and Elbow Method

Figure A.1 shows an example of the K-means clustering algorithm applied to a dataset with 3 clusters. The left-hand side shows the clusters found by the K-means algorithm, while the right-hand side shows the elbow method to find the optimal number of clusters. The elbow method suggests that the optimal number of clusters is 3, which is the true number of clusters in the dataset.