Data Science

Diploma in Banking Supervision (CEMFI)

Joël Marbet

May 28, 2024

Table of contents

About this Course

This course serves as an **introduction to machine learning techniques used in data science**. While we will cover some of the underlying theory to get a better understanding of the methods we are going to use, the **emphasis will be on practical implementation**. Throughout the course, we will be using the programming language **Python**, which is the dominant programming language in this field.

The course is divided into two parts. In the first part, we will get a brief overview of the field, cover some basic concepts of machine learning and have a look at some of the most commonly used methods. In the second part, we will apply these methods to real-world problems, which hopefully will give you a starting point for your own projects. The course outline is as follows:

Part I: Overview and Methods

1. Introduction to Machine Learning
2. Basic Concepts
3. Decision Trees
4. Neural Networks
5. Additional Methods

Part II: Applications

6. Loan Default Prediction
7. House Price Prediction

The course is designed to be self-contained, meaning that you do not need any prior knowledge of machine learning to follow along.

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

We will be using Python for this course. For simplicity, we will be using the Anaconda distribution, which is a popular distribution of Python (and R) that aims to simplify the management of packages. We will also be using the Visual Studio Code (VS Code) as our code editor.

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 (M1) Graphical Installer. With an older Intel Mac, you can choose the 64-Bit 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.

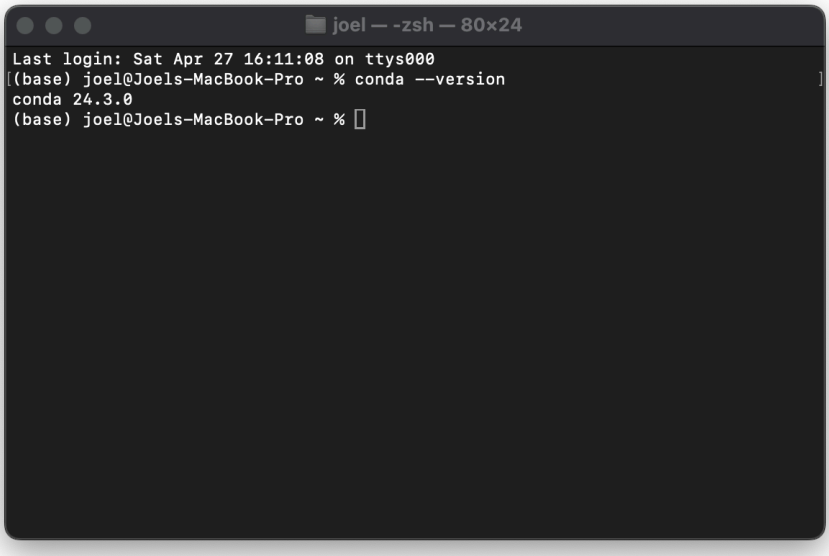


Figure 1: Terminal Output after Anaconda Installation

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://datascience.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 `datascience_course_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 datascience_course_cemfi
```

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

```
python --version
```

This should display Python version 3.8.8. If you see another Python version you might have forgotten to activate the environment or the environment was not created successfully.

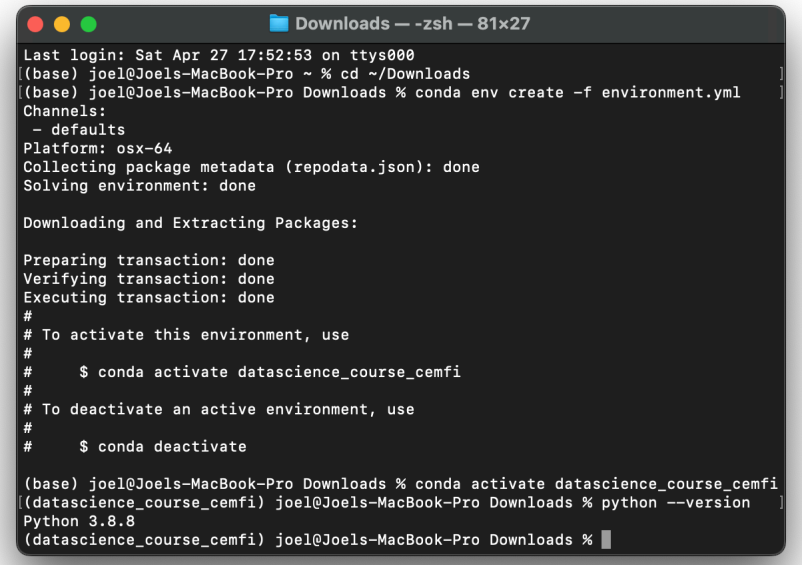


Figure 2: Terminal Output From Environment Creation

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://datascience.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).

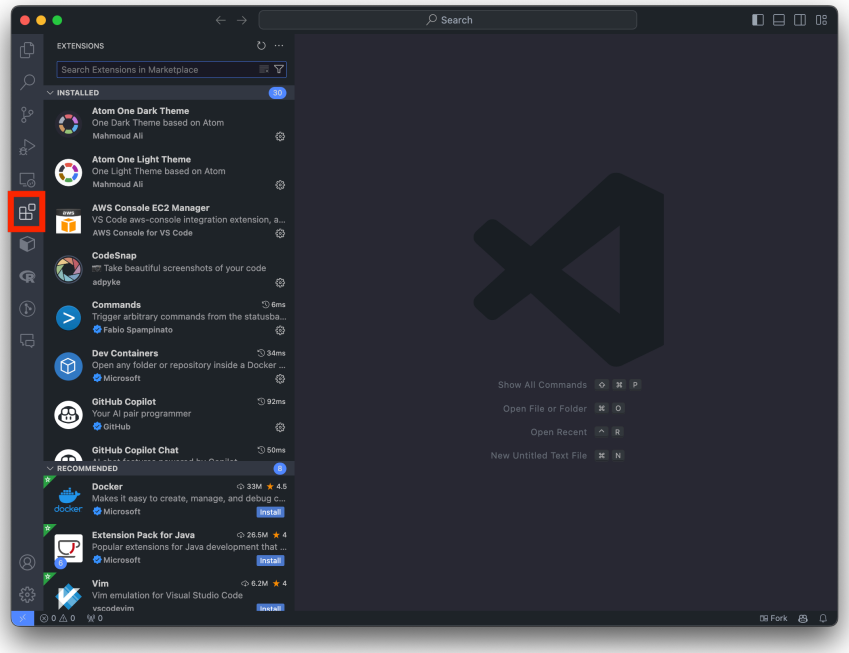


Figure 3: Installing Extensions in VSCode

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 `datascience_course_cemfi` kernel.

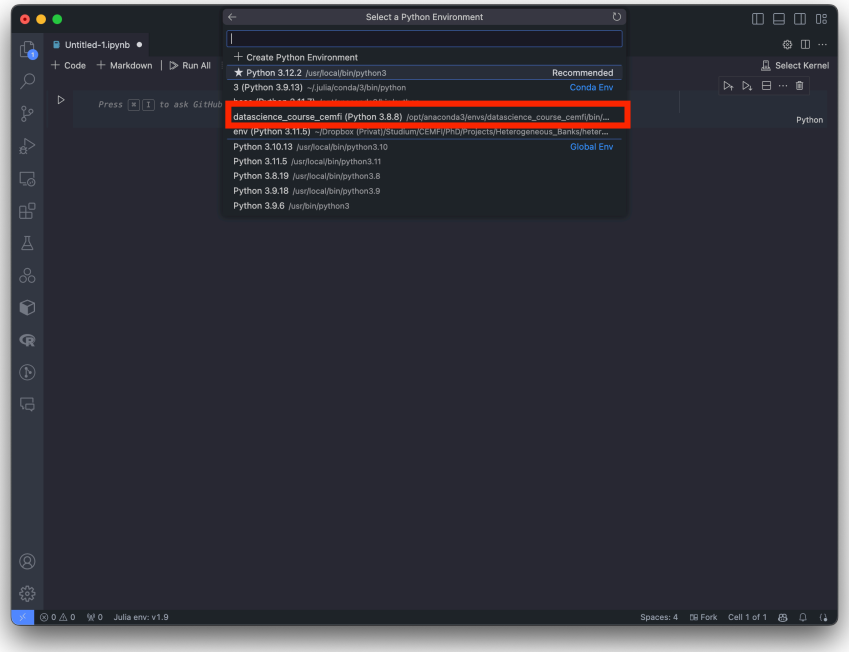


Figure 4: VSCode Jupyter Kernel Selection

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 down-

loaded the course materials and open the notebooks from there.

Part I

Overview and Methods

Chapter 1

Introduction

The hype around **artificial intelligence** (AI) reached new highs with the release of OpenAI's ChatGPT in late 2022. These drastic improvements in artificial intelligence have been fueled by **machine learning** (ML) methods that have become popular in recent years and have a **wide variety of applications** such as, for example,

- Computer vision,
- Speech recognition,
- Data mining,

and many more. These tools also have **many potential applications in economics and finance** and can be invaluable in extracting information from the evergrowing 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. Therefore, you have a unique **opportunity to apply these methods** to a wide range of unexplored problems.

The field can be very technical, but **barriers to entry are not as high as they may seem**. This course aims to provide you with the tools to apply machine learning methods to problems in economics and finance.

1.1 Taking Advantage of Machine Learning in Banking Supervision

You might have heard of some of the well-known **advances in the field of AI** from recent years such as

- DeepMind's AlphaGo can beat the best human Go players
- OpenAI's ChatGPT responds to complex text prompts
- Midjourney, DALL-E, and Stable Diffusion generate images from text

Figure 1.1: Go board (Source: Wikimedia)

- …

While these examples are impressive, you might wonder how these methods can help you in your work. There is a wide range of potential applications. Machine learning methods have been used **in practice** to

- *Predict* loan or firm defaults,
- *Detect* fraud (e.g., credit card fraud, or money laundering),
- *Interpret* large quantities of data, or
- *Forecast* economic variables

to just name a few examples. Bank for International Settlements (2021) provides an overview of how machine learning methods have been used at central banks.[1] The report also notes how machine learning methods can be used in the context of **financial supervision**

> These techniques can support supervisors' efficiency in: (i) covering traditional supervisory tasks (eg quality reporting, anomaly detection, sending of instructions); (ii) facilitating the assessment of micro-level fragilities; and (iii) identifying and tackling new emerging topics, such as climate-related financial risks, vulnerabilities from the Covid-19 pandemic, or the consequence of increased digitisation in finance (eg the development of fintechs).

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

[1] See https://www.bis.org/ifc/publ/ifcb57.htm for a more detailed overview

- 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).

In this course, we will only be able to scratch the surface of the field. However, I hope to provide you with the tools to get you started with machine learning and to apply these methods to novel problems.

1.2 What Is Machine Learning?

You might have already some idea of what machine learning is. In this section, we will provide a more formal definition, discuss the relation to statistics and econometrics, and distinguish between machine learning, artificial intelligence, and deep learning.

1.2.1 Definition

Let's start with the straightforward definition provided by Murphy (2012)

> [...] 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.

1.2.2 Relation 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.

1.2.3 Artificial Intelligence vs. Machine Learning vs. Deep Learning

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):

- **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.

We will have a look at neural networks and deep learning later on.

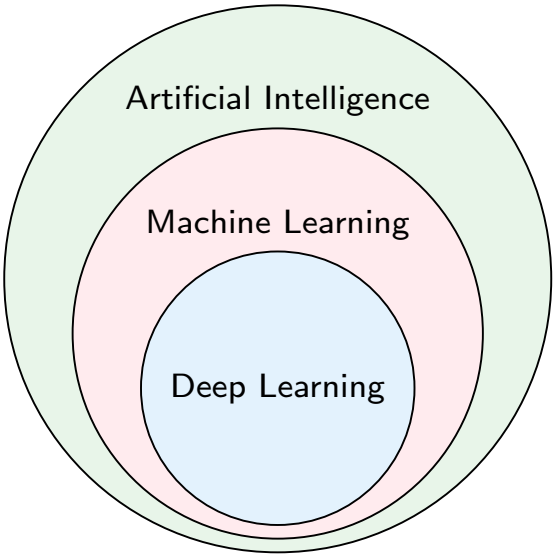


Figure 1.2: Artificial intelligence vs. Machine Learning vs. Deep Learning

1.3 Why Has Machine Learning Become Popular Only Recently?

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). 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**

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).

1.4 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. For **example**, our task might be to figure out whether a credit card transaction is fraudulent or not. Based on the data we have access to, two different types of learning might be appropriate:

- We *know* which transactions are fraudulent meaning that we need to learn a function that maps the transaction data (e.g., value of transaction, location, etc.) to the label "fraudulent" or "not fraudulent". This is an example of *supervised learning*.
- We *do not know* whether they are fraudulent or not meaning that we might want to find clusters in the data that group similar transactions. This is an example of *unsupervised learning*.

More generally, 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

The focus of this course will be on supervised learning, but we will also have a look at some unsupervised learning techniques if time allows. Let's have a closer look at the three types of learning.

> **i** Types of Learning in Practice
>
> Machine learning models might **combine different types of learning**. For example, ChatGPT is trained using a combination of supervised and reinforcement learning. Furthermore, some machine learning methods, such as neural networks, might be used as part of different types of learning.

1.4.1 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 $n = 1, \dots, N$. 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 ($\rightarrow$ Figure 1.3)
- From text input x to some coherent text response y ($\rightarrow$ ChatGPT)
- From text input x to a generated image y ($\rightarrow$ 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. $\mathcal{Y} = \{1, 2, 3, \ldots, C\}$
- **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

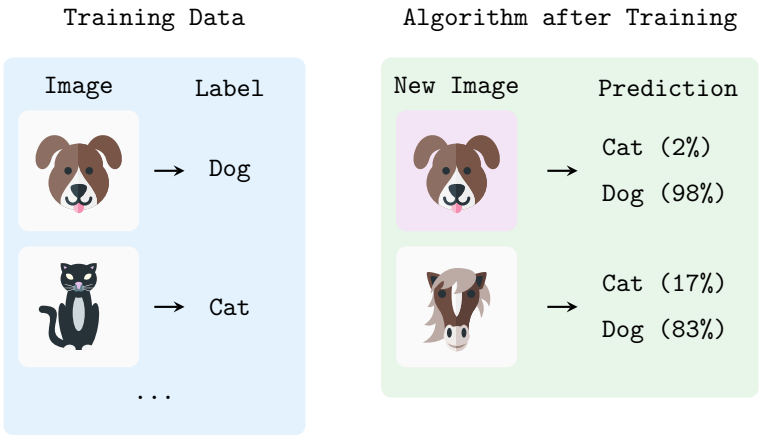


Figure 1.3: Training a machine learning algorithm to classify images of cats and dogs

Figure 1.3 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

In **regression tasks**, the variable that we want to predict is continuous. Linear and polynomial regression in Figure 1.4 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**,

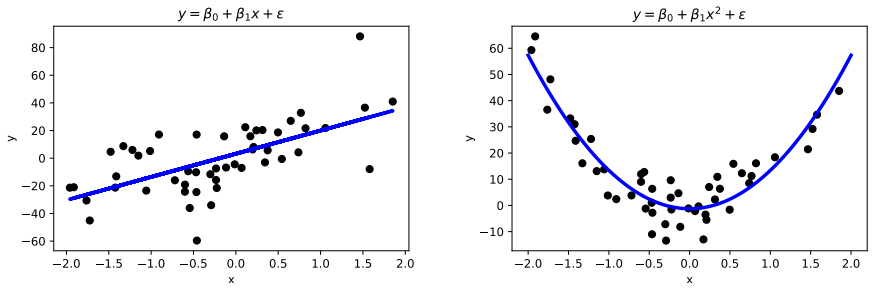


Figure 1.4: Linear and Polynomial Regression

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 1.4) of the relationship. This makes them very flexible and powerful tools. We will have a look at neural networks later on.

1.4.2 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" ($\rightarrow$ clustering)
- Finding **latent factors** that capture the "essence" of the data ($\rightarrow$ dimensionality reduction)

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

Clustering

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 1.5 shows an example of a **clustering task**. The dataset consists of

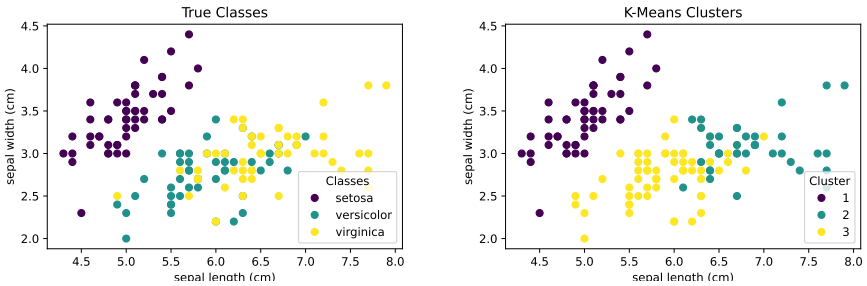


Figure 1.5: Clusters in data on iris flowers (left-hand side: true classes, right-hand side: k-means clusters)

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 1.5, 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.

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**.

1.4.3 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

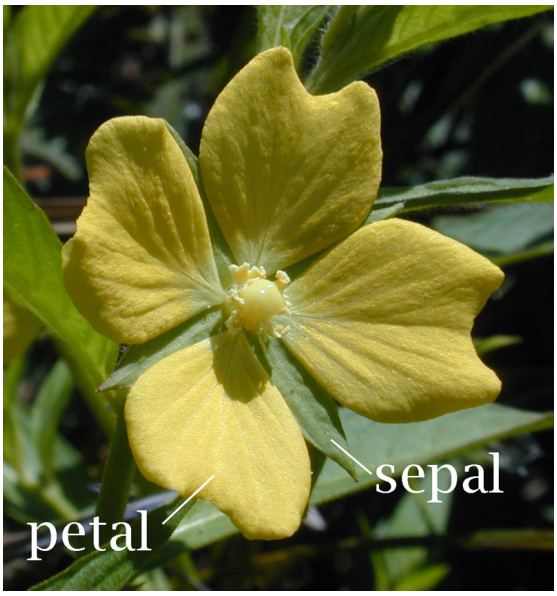


Figure 1.6: Petal vs Sepal (Source: Wikimedia)

- 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 **AlphaGo** that learned how to play Go.

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".

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.

i Mini-Exercise

Are the following tasks examples of supervised, or unsupervised 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

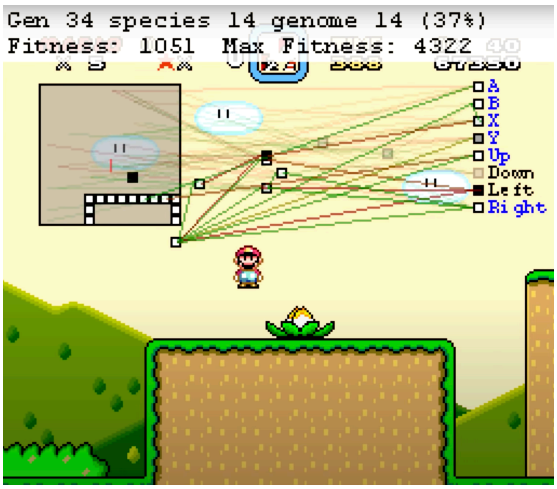


Figure 1.7: MarI/O playing Super Mario World (Source: YouTube)

labeled credit card transactions).

5. Recognizing handwritten digits in the MNIST dataset (see next section).

1.5 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 1.8) 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 1.8: MNIST (Source: Wikimedia)

Chapter 2

Basic Concepts

Now that we have a basic understanding of what machine learning is, let's dive into some concepts that are essential for understanding machine learning models. The **focus** of this section will be on **supervised learning** models. We will start with placing linear regression and logistic regression in a machine-learning context. We will then 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.

2.1 Linear Regression in a ML Context

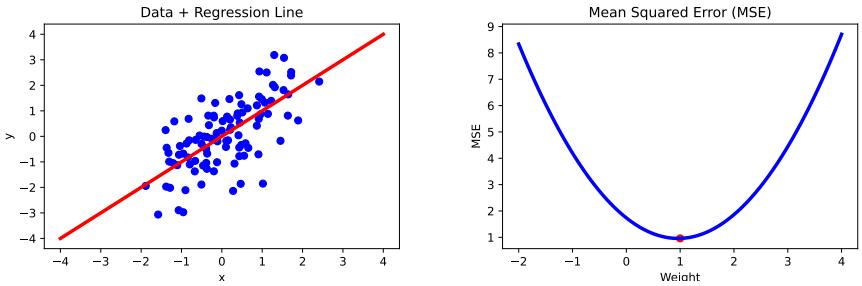


Figure 2.1: Linear Regression With a Single Feature x (i.e., $m = 1$) and Bias $b = 0$ (Note that this Plot is interactive in the HTML version)

You have already extensively studied linear regressions in the statistics and econometrics course, 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 2.1 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.

2.2 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 $w = [w_1, \dots, w_n]' \in \mathbb{R}^m$ 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 $p(y = 1|x; w, b) > t$.

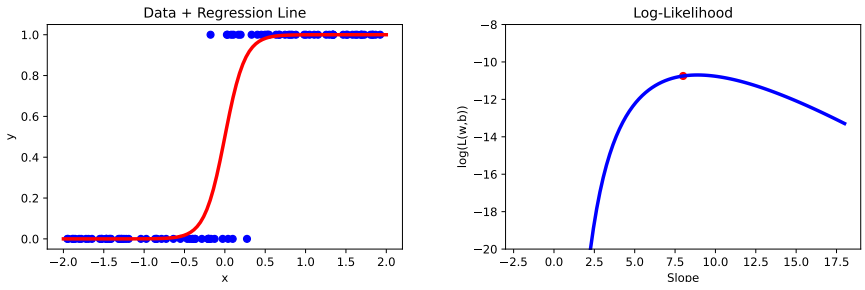


Figure 2.2: Logistic Regression With a Single Feature x (i.e., $m = 1$) and Bias $b = 0$ (Note that this Plot is interactive in the HTML version)

> ⚠ 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 2.2 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.

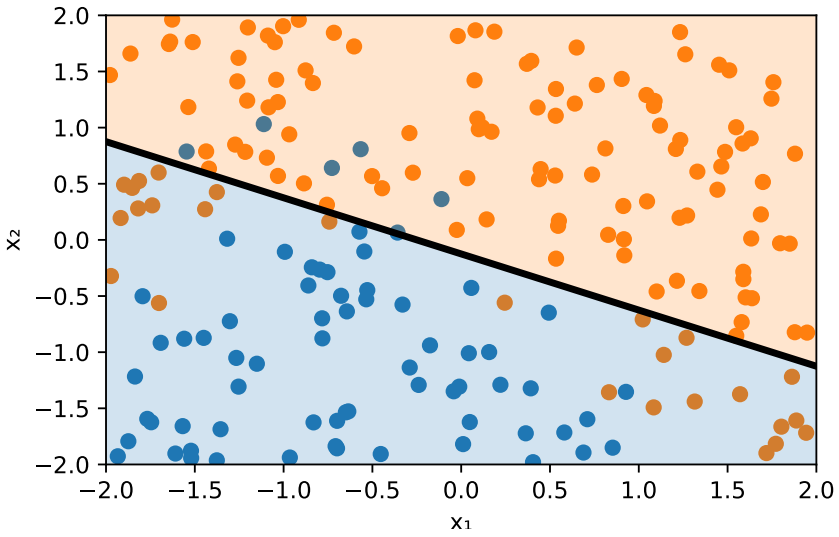


Figure 2.3: Decision Boundary - Logistic Regression with Features x_1 and x_2 (i.e., $m = 2$) (Note that this Plot is interactive in the HTML version)

This linearity also shows up in the **linear decision boundary** produced by a logistic regression in Figure 2.3. 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 2.3. We can remedy this issue by including higher order terms for x_1 and x_2 such as x_2^2 or x_1^3, 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.

2.3 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.

2.3.1 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.

2.3.2 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**

$$\text{Missclassification Rate} = \frac{\text{Number of incorrect predictions}}{\text{Total number of predictions}}.$$

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 $\hat{y} = 0$ 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 2.4. 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.

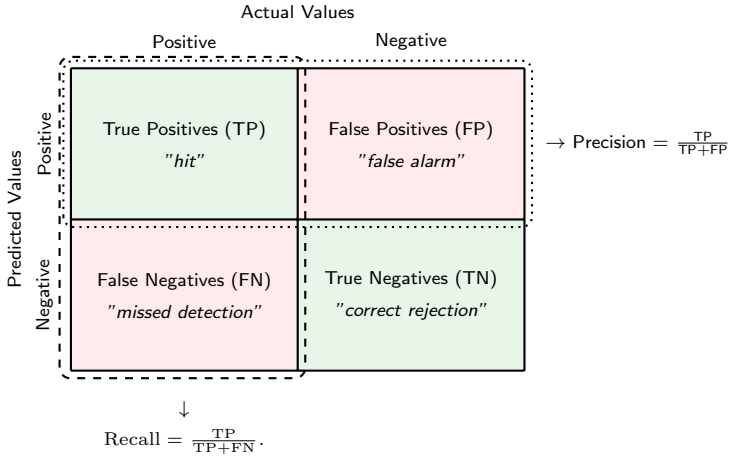


Figure 2.4: Confusion matrix

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.

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 **missclassification rate** is given by

$$\text{Missclassification Rate} = \frac{\text{FP} + \text{FN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}.$$

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.,

$$\text{FPR} = 1 - \text{TNR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

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 2.5 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 2.6 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.

2.4 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

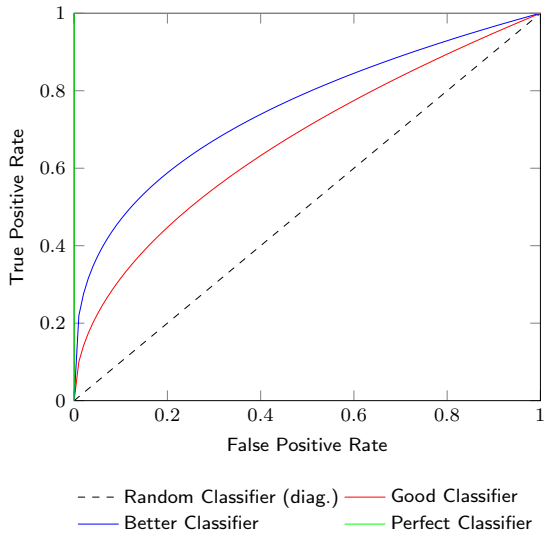


Figure 2.5: Receiver Operating Characteristic (ROC) Curve

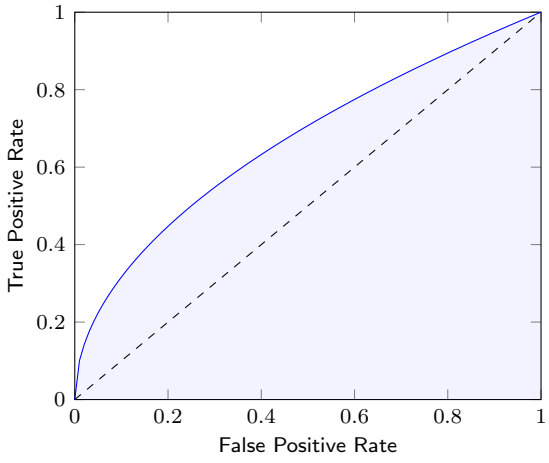


Figure 2.6: Area Under the Curve (AUC)

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.

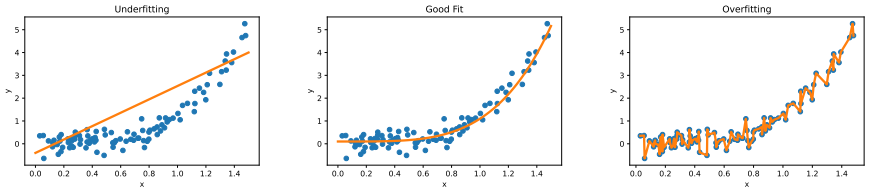


Figure 2.7: Examples of Overfitting and Underfitting

Figure 2.7 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.

2.4.1 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 $\text{Var}[\epsilon] = \sigma^2$. 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

$$\text{EPE}(x_0) = \mathbb{E}[(Y - \hat{f}(x_0))^2 | X = x_0] = \text{Bias}^2(\hat{f}(x_0)) + \text{Var}(\hat{f}(x_0)) + \sigma^2,$$

where $\text{Bias}(\hat{f}(x_0)) = \mathbb{E}[\hat{f}(x_0)] - f(x_0)$ is the bias at x_0, $\text{Var}(\hat{f}(x_0)) = \mathbb{E}[\hat{f}(x_0)^2] - \mathbb{E}[\hat{f}(x_0)]^2$ 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.

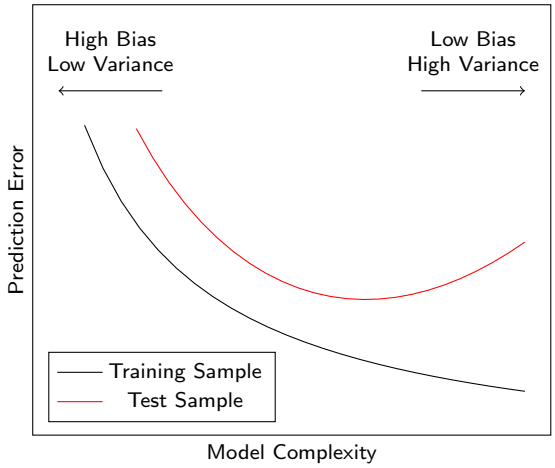


Figure 2.8: Model Complexity and Generalization Error (adapted from Hastie, Tibshirani, and Friedman 2009)

Figure 2.8 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.

2.4.2 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 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 depend-

ing 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 2.9 shows the LASSO regression loss for different levels of λ.

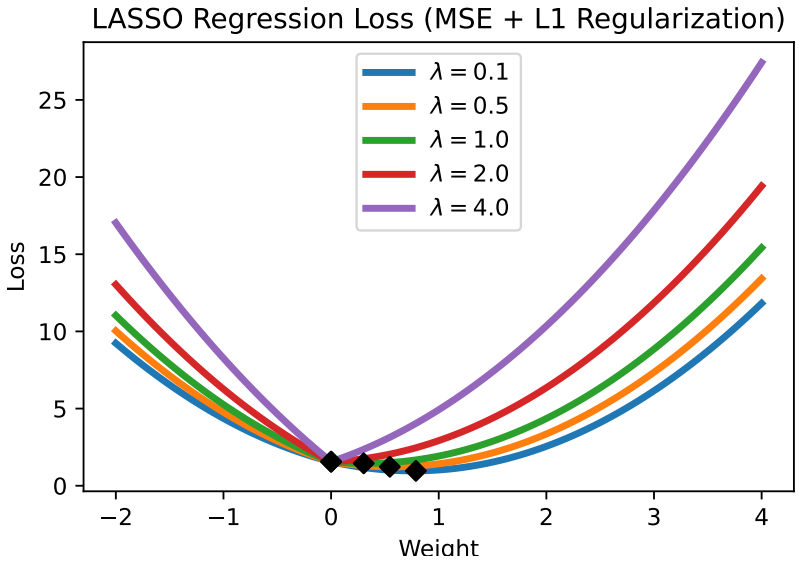


Figure 2.9: LASSO Regression Loss for Different Levels of λ

An L2 regularization in a linear regression context is called a **Ridge regression**. Its loss function is given by

$$\text{Loss} = \text{MSE}(y, x; w) + \lambda \sum_{i=1}^{m} w_i^2.$$

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 2.10 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.

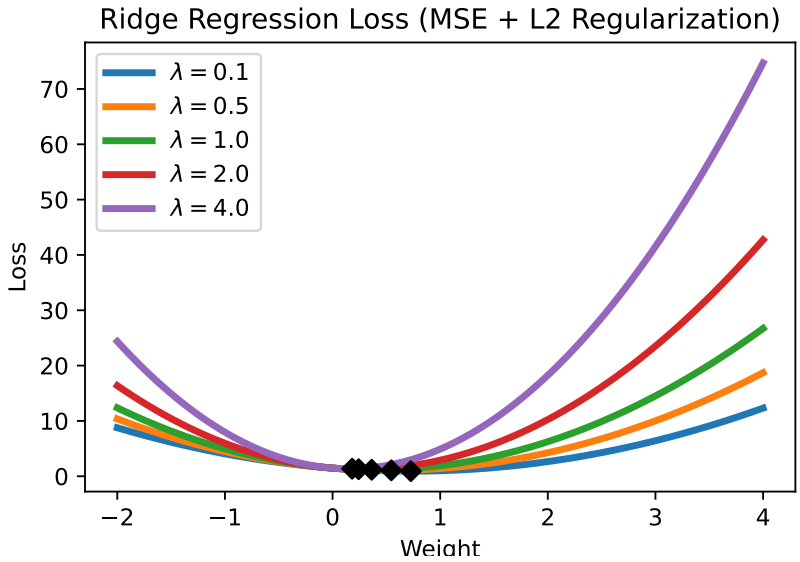


Figure 2.10: Ridge Regression Loss for Different Levels of λ

2.4.3 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 2.11 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.

> **i** 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 2.11: Option A - Splitting the Whole Dataset into Training, and Test Datasets

The second option shown in Figure 2.12 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.

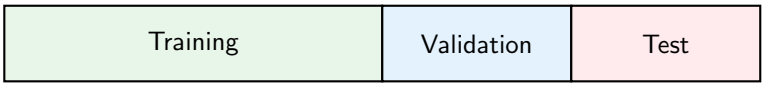


Figure 2.12: Option B - Splitting the Whole Dataset into Training, Test, and Validation Datasets

Common splits are 75% 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.

2.4.4 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 2.13, 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**.

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.

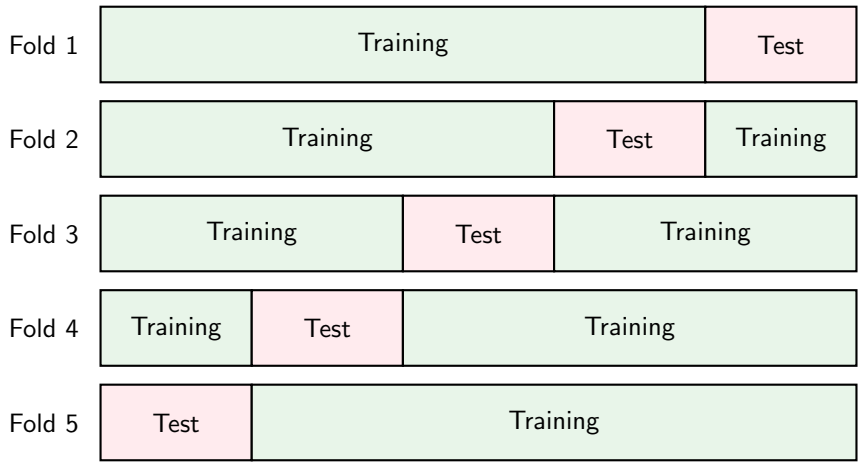


Figure 2.13: 5-Fold Cross-Validation

Mini-Exercise

Implement a 5-fold cross-validation for the logistic regression model in the Python example below. 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()
```

2.5 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

```
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.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
```

Then, we can load the data

```
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.

2.5.1 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
```

```
(1000000, 8)
```

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
```

	0	1	2	3	4
distance_from_home	57.877857	10.829943	5.091079	2.247564	44.190936
distance_from_last_transaction	0.311140	0.175592	0.805153	5.600044	0.566486
ratio_to_median_purchase_price	1.945940	1.294219	0.427715	0.362663	2.222767
repeat_retailer	1.000000	1.000000	1.000000	1.000000	1.000000
used_chip	1.000000	0.000000	0.000000	1.000000	1.000000
used_pin_number	0.000000	0.000000	0.000000	0.000000	0.000000
online_order	0.000000	0.000000	1.000000	1.000000	1.000000
fraud	0.000000	0.000000	0.000000	0.000000	0.000000

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)
```

	distance_from_home	distance_from_last_transaction	ratio_to_median_purchase_price	re
0	57.877857	0.311140	1.945940	1.
1	10.829943	0.175592	1.294219	1.
2	5.091079	0.805153	0.427715	1.
3	2.247564	5.600044	0.362663	1.
4	44.190936	0.566486	2.222767	1.
5	5.586408	13.261073	0.064768	1.
6	3.724019	0.956838	0.278465	1.
7	4.848247	0.320735	1.273050	1.
8	0.876632	2.503609	1.516999	0.
9	8.839047	2.970512	2.361683	1.
10	14.263530	0.158758	1.136102	1.
11	13.592368	0.240540	1.370330	1.
12	765.282559	0.371562	0.551245	1.
13	2.131956	56.372401	6.358667	1.
14	13.955972	0.271522	2.798901	1.
15	179.665148	0.120920	0.535640	1.
16	114.519789	0.707003	0.516990	1.
17	3.589649	6.247458	1.846451	1.
18	11.085152	34.661351	2.530758	1.
19	6.194671	1.142014	0.307217	1.

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
```

```
Index(['distance_from_home', 'distance_from_last_transaction',
       'ratio_to_median_purchase_price', 'repeat_retailer', 'used_chip',
       'used_pin_number', 'online_order', 'fraud'],
      dtype='object')
```

and the data types of the variables

```
df.dtypes
```

```
distance_from_home                float64
distance_from_last_transaction    float64
ratio_to_median_purchase_price    float64
```

```
repeat_retailer                 float64
used_chip                       float64
used_pin_number                 float64
online_order                    float64
fraud                           float64
dtype: object
```

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
```

	count	mean	std	min	25%	50%	75%
distance_from_home	1000000.0	26.628792	65.390784	0.004874	3.878008	9.967760	25.7439
distance_from_last_transaction	1000000.0	5.036519	25.843093	0.000118	0.296671	0.998650	3.35574
ratio_to_median_purchase_price	1000000.0	1.824182	2.799589	0.004399	0.475673	0.997717	2.09637
repeat_retailer	1000000.0	0.881536	0.323157	0.000000	1.000000	1.000000	1.00000
used_chip	1000000.0	0.350399	0.477095	0.000000	0.000000	0.000000	1.00000
used_pin_number	1000000.0	0.100608	0.300809	0.000000	0.000000	0.000000	0.00000
online_order	1000000.0	0.650552	0.476796	0.000000	0.000000	1.000000	1.00000
fraud	1000000.0	0.087403	0.282425	0.000000	0.000000	0.000000	0.00000

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()
```

	distance_from_home	distance_from_last_transaction	ratio_to_median_purchase_price
0	False	False	False
1	False	False	False
2	False	False	False
3	False	False	False
4	False	False	False
...	...	...	...
999995	False	False	False
999996	False	False	False
999997	False	False	False
999998	False	False	False
999999	False	False	False

or to make it easier to see, we can sum the number of missing values for each variable

```
df.isna().sum()
```

```
distance_from_home                0
distance_from_last_transaction    0
ratio_to_median_purchase_price    0
repeat_retailer                   0
used_chip                         0
used_pin_number                   0
online_order                      0
fraud                             0
dtype: int64
```

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()
```

```
<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
```

We can also check for duplicated rows with the `duplicated()` method.

```
df.loc[df.duplicated()]
```

distance_from_home	distance_from_last_transaction	ratio_to_median_purchase_price	repeat_retailer

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()
```

```
fraud
0.0    912597
1.0     87403
Name: count, dtype: int64
```

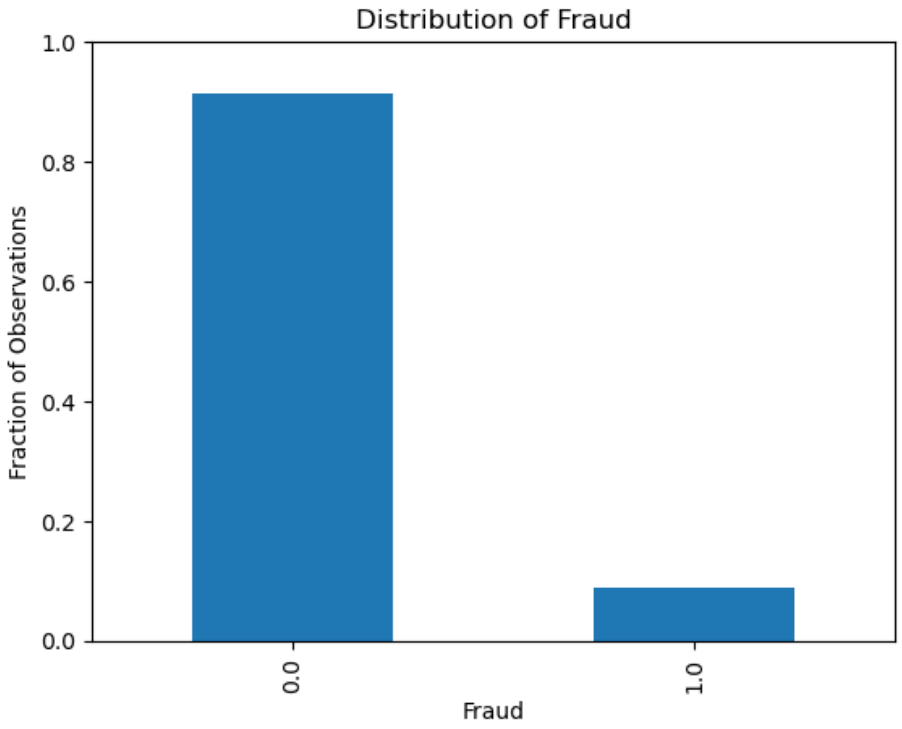
to get the count of each value. We can also use the `normalize=True` argument to get the fraction of observations instead of the count

```
df['fraud'].value_counts(normalize=True)
```

```
fraud
0.0    0.912597
1.0    0.087403
Name: proportion, dtype: float64
```

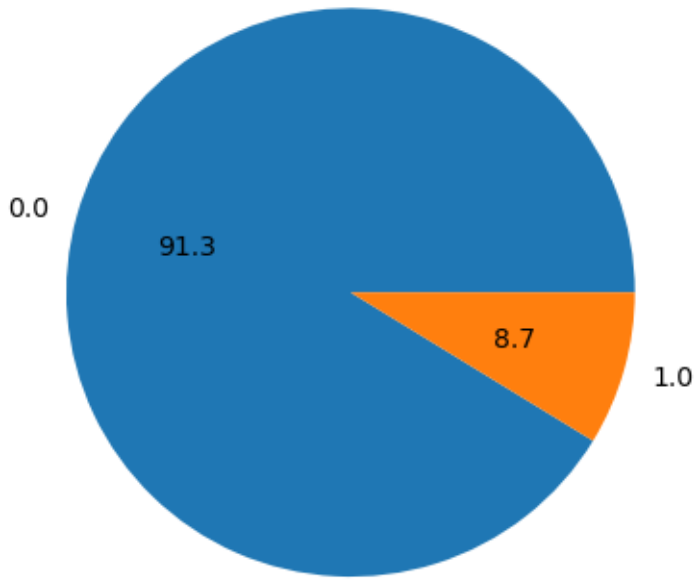
We can then plot it as follows

```
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

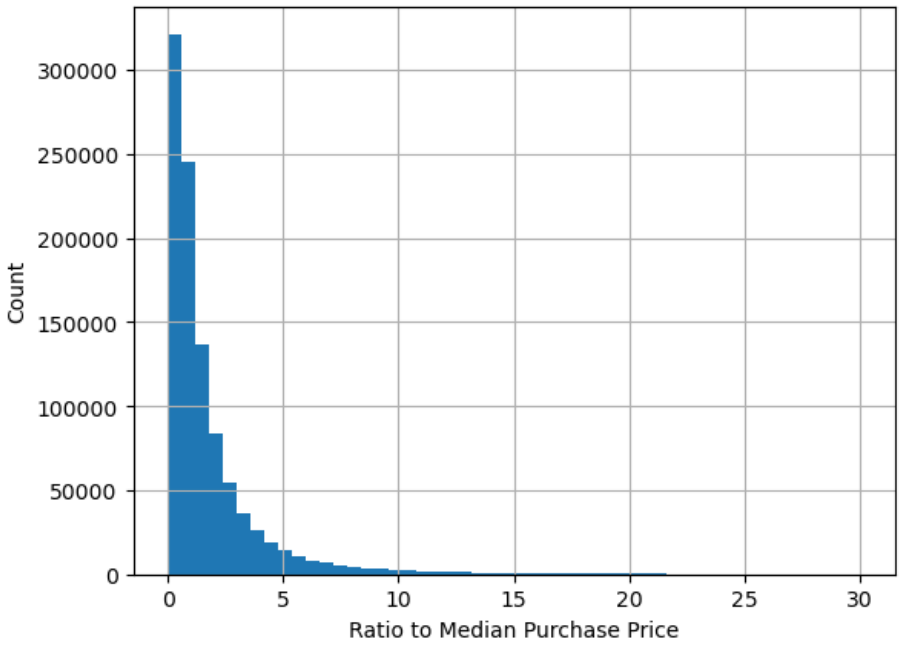
```
df.value_counts("fraud").plot.pie(autopct = "%.1f")
plt.ylabel('')
plt.show()
```



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%.

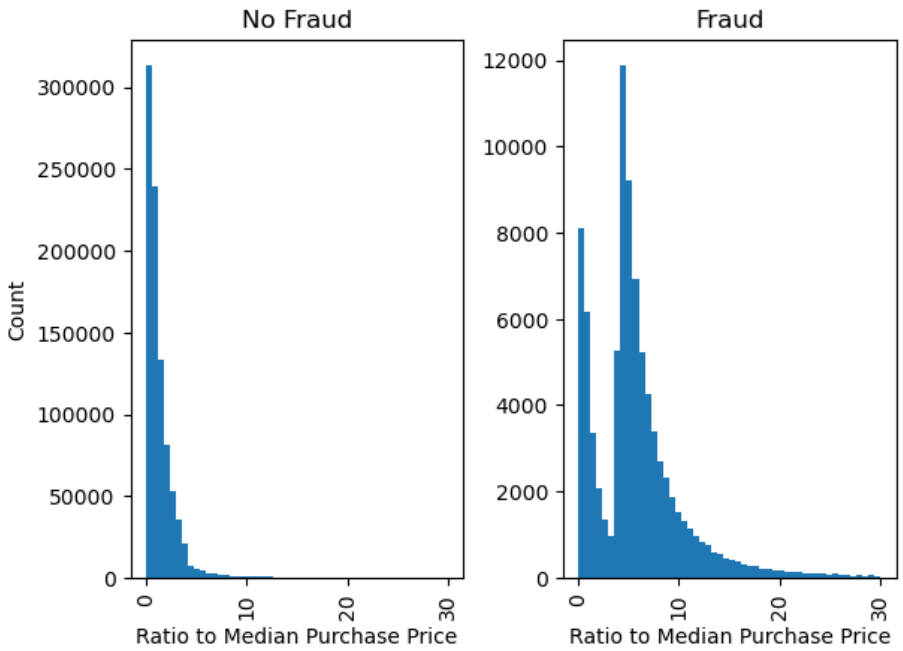
Let's look at some distributions. Most of the variables in the dataset are binary (0 or 1) variables. However, we also have some continuous variables. Let's plot the distribution of the variable `ratio_to_median_purchase_price`, which is a continuous variable.

```
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)
```

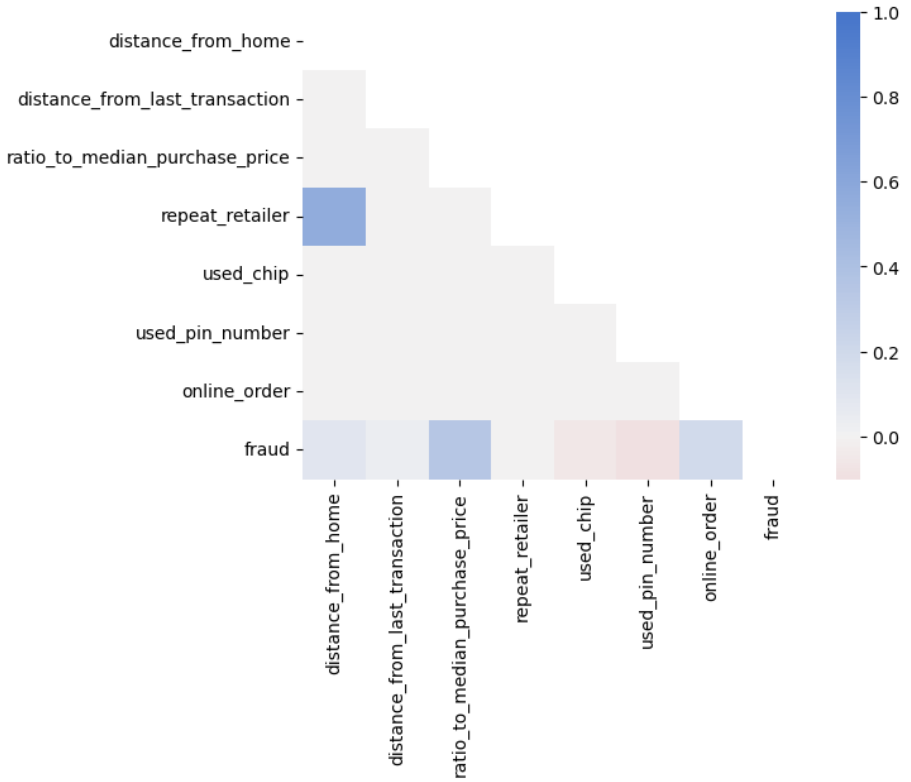
	distance_from_home	distance_from_last_transaction	ratio_to_median_
distance_from_home	1.000000	0.000193	-0.001374
distance_from_last_transaction	0.000193	1.000000	0.001013
ratio_to_median_purchase_price	-0.001374	0.001013	1.000000
repeat_retailer	0.143124	-0.000928	0.001374
used_chip	-0.000697	0.002055	0.000587
used_pin_number	-0.001622	-0.000899	0.000942
online_order	-0.001301	0.000141	-0.000330
fraud	0.187571	0.091917	0.462305

```
df.corr('spearman') # Spearman correlation (for monotonic
↪  relationships)
```

	distance_from_home	distance_from_last_transaction	ratio_t
distance_from_home	1.000000	-0.001068	-0.0001
distance_from_last_transaction	-0.001068	1.000000	-0.0001
ratio_to_median_purchase_price	-0.000152	-0.000111	1.0000
repeat_retailer	0.559724	-0.001352	0.0012
used_chip	-0.000118	-0.000165	-0.0000
used_pin_number	-0.000338	0.000555	0.0002
online_order	-0.001812	-0.001076	-0.0003
fraud	0.095032	0.034661	0.3428

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.

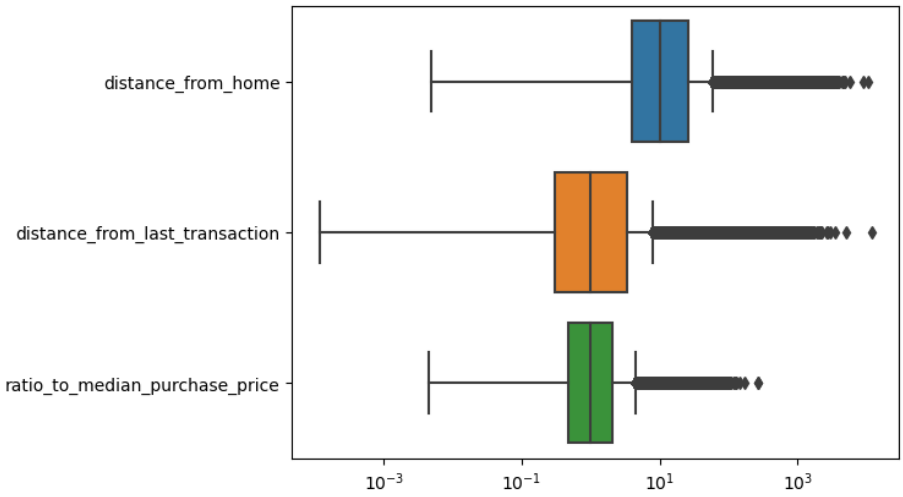
```
corr = df.corr('spearman')
cmap = sns.diverging_palette(10, 255, as_cmap=True) # Create a
 ↪  color map
mask = np.triu(np.ones_like(corr, dtype=bool)) # Create a mask to
 ↪  only show the lower triangle of the matrix
sns.heatmap(corr, cmap=cmap, vmax=1, center=0, mask=mask) #
 ↪  Create a heatmap 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 amoung our selected
 ↪  outliers
outliers['fraud'].value_counts()
```

```
fraud
1.0    53092
0.0    31294
Name: count, dtype: int64
```

```
df['fraud'].value_counts()
```

```
fraud
0.0    912597
1.0     87403
Name: count, dtype: int64
```

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.

> **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.

2.5.2 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))
```

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

The actual classes of the first five observations in the test dataset are

```
y_test.head(5)
```

```
217309    0.0
902387    0.0
175152    0.0
527113    0.0
973041    1.0
Name: fraud, dtype: float64
```

This seems to match quite well. Let's have a look at different performance metrics

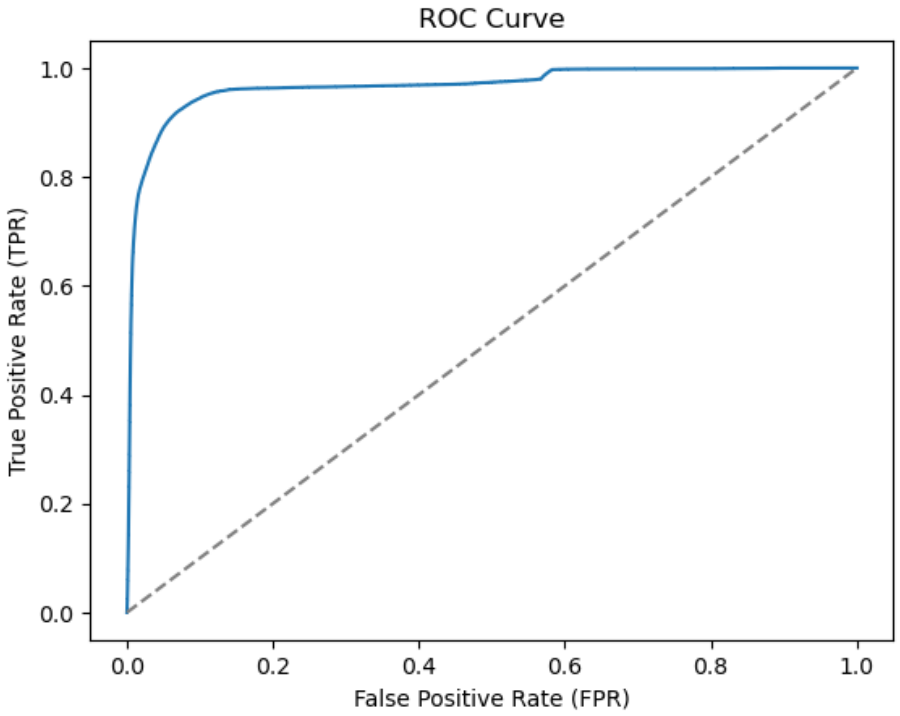
```
y_pred = clf.predict(X_test)
y_proba = clf.predict_proba(X_test)

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])}")
```

```
Accuracy: 0.95908
Precision: 0.8954682094038908
Recall: 0.6021128103428549
ROC AUC: 0.9671832218100465
```

As expected, the accuracy is quite high since we do not have many fraudulent transactions. Recall that the precision (Precision $= \frac{\text{TP}}{\text{TP+FP}}$) is the fraction of correctly predicted fraudulent transactions among all transactions transactions predicted to be fraudulent. The recall (Recall $= \frac{\text{TP}}{\text{TP+FN}}$) 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()
```



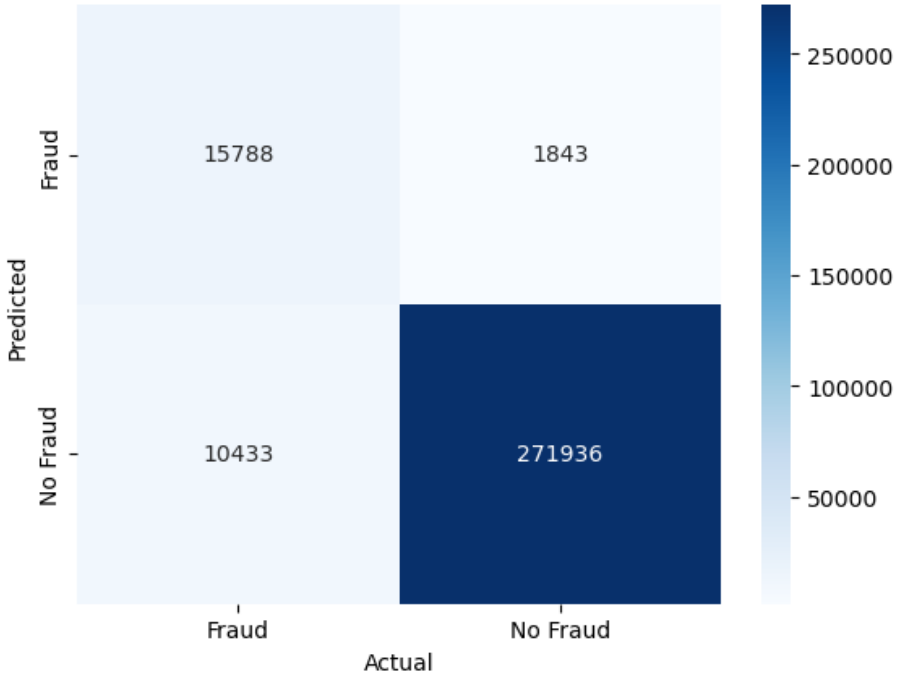
The confusion matrix for the test set can be computed as follows

```
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
```

```
array([[ 15788,   1843],
       [ 10433, 271936]])
```

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)}")
```

```
Accuracy: 0.9112033333333334
Precision: 0.49579121188932296
Recall: 0.9389420693337401
```

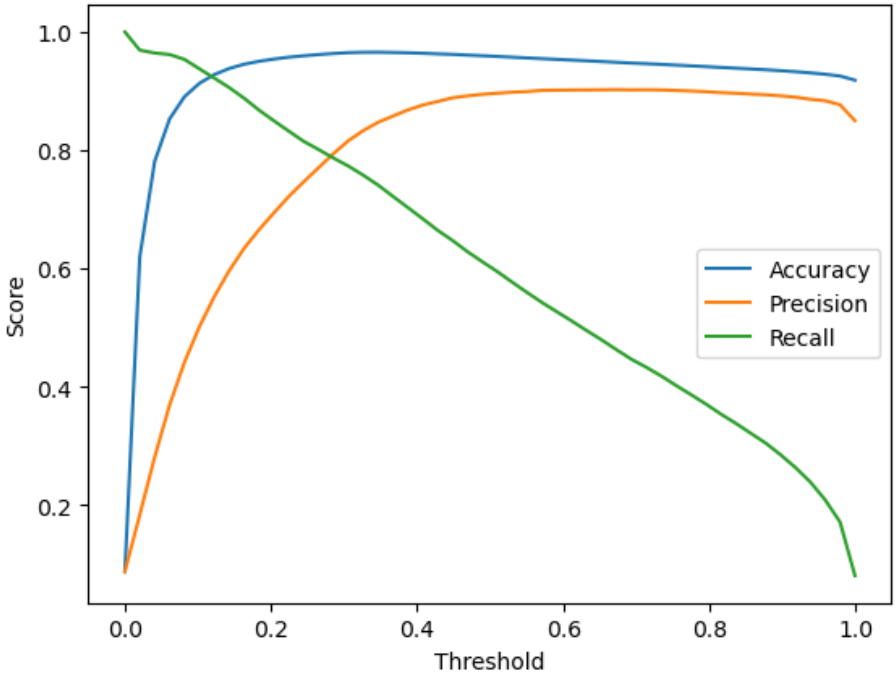
Setting a lower threshold increases the recall but decreases the precision. This is

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

```
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()
```



2.5.3 Conclusions

In this notebook, we have seen how to implement a logistic regression model in Python. We have loaded a dataset, explored and preprocessed it, and trained a logistic regression model 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 model. 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 notebook.

Chapter 3

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**.

3.1 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.

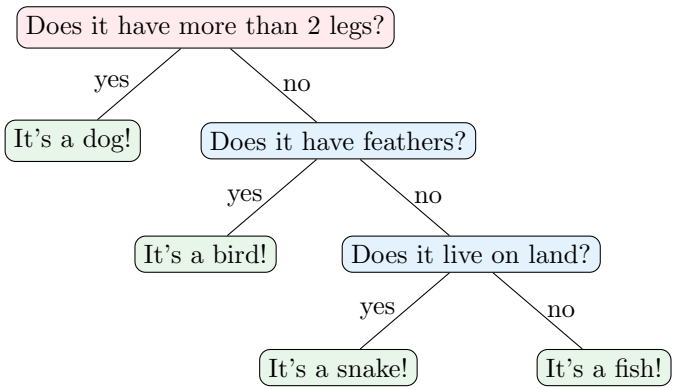


Figure 3.1: Classification Tree - Classification of Dogs, Snakes, Fish, and Birds based on their Features

Figure 3.1 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.

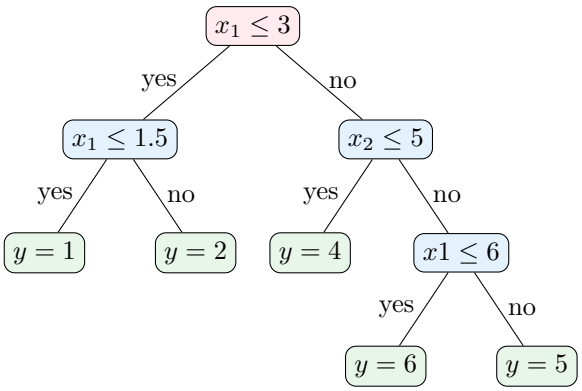


Figure 3.2: Regression Tree - Prediction of y based on x_1 and x_2

Figure 3.2 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 3.7 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

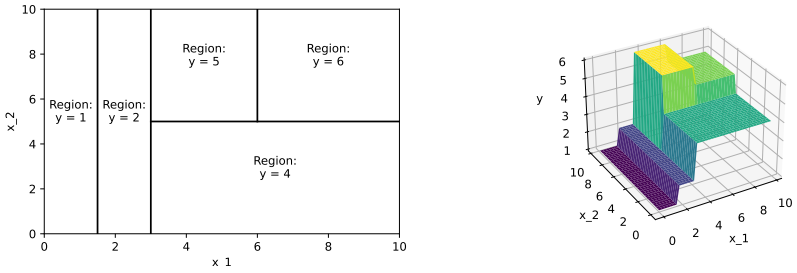
$$\hat{y} = \sum_{m=1}^{M} c_m \mathbb{1}(x \in R_m)$$

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.

i Mini-Exercise

Given the decision tree in Figure 3.2, 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. $(x_1, x_2) = (2, 8)$



(a) Regions (b) Predictions

Figure 3.3: Regression Tree - Regions and Predictions of Decision Tree in Figure 3.2

4. $(x_1, x_2) = (10, 4)$
5. $(x_1, x_2) = (7, 8)$

3.2 Terminology

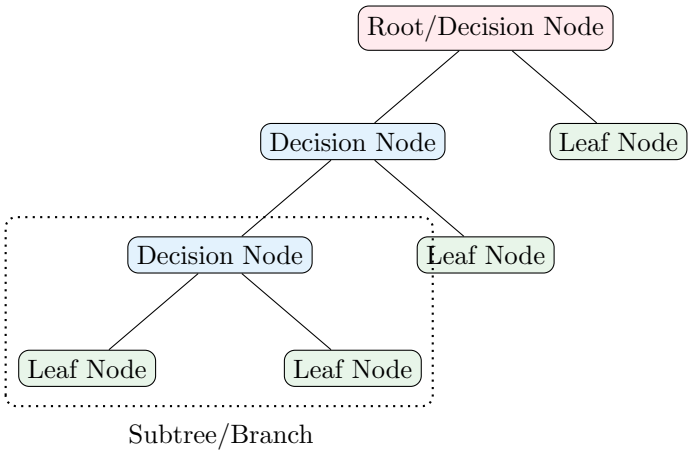


Figure 3.4: Decision Tree - Terminology

Figure 3.4 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 **decision nodes** (or leaf nodes) based on the values of the features. 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 3.4 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 3.4 is the parent node of the first decision node and leaf node).

3.3 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 3.5 shows an example of a candidate split.

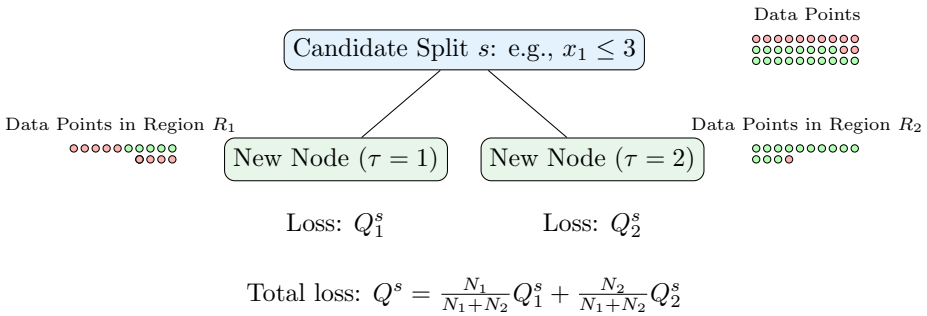


Figure 3.5: Example of Decision Tree Split

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.

3.3.1 Example: Classification Problem

Suppose you have the data in Table 3.1. 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 3.1: (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 3.2: Gini Impurities for Different CET1 Thresholds

CET1 Ratio Threshold	Q	Q	Q
8.8	0	0.44	0.38
9.8	0.5	0.2	0.29
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 3.2, the best split is at a CET1 ratio of 7.0%. 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 3.6. In particular, both, the split at a CET1 ratio of 9.8% 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.

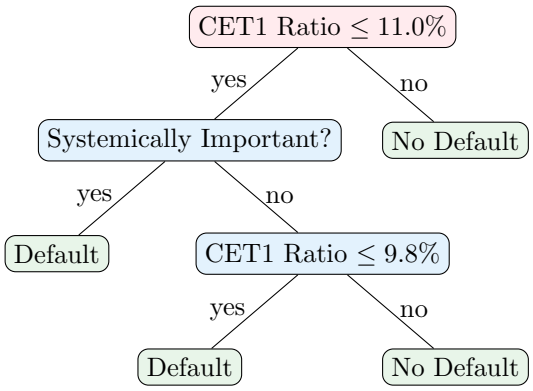


Figure 3.6: Classification Tree for Table 3.1

3.3.2 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 3.7 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.

Another way to prevent overfitting is to **prune** the tree, i.e., to remove nodes

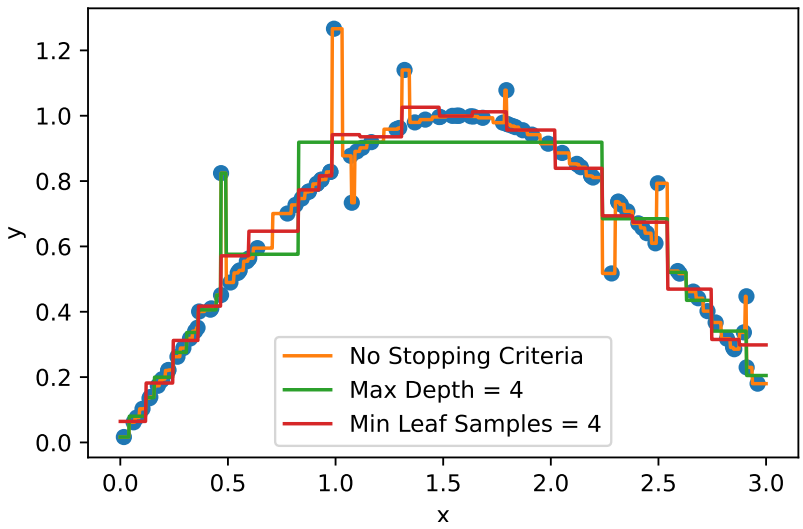


Figure 3.7: Regression Tree - Effect of Stopping Criteria

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 3.6 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?

3.4 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)

3.5 Random Forests

Decision trees can be unstable meaning that small changes in the training data can lead to large changes in the tree structure. One way to address this issue is to use **Random Forests**. Random Forests is an **ensemble method**: The idea 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 for each tree. This means that each tree is trained on a different subset of the data and a different subset of features.

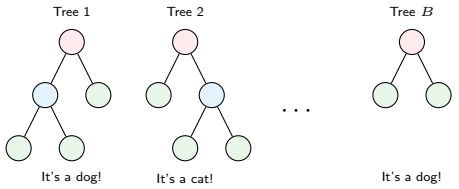


Figure 3.8: Random Forests - Ensemble of Decision Trees with Majority Decision "It's a dog!"

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.

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 N-fold cross-validation.

3.6 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 here 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.

3.7 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.

3.8 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

```
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

# 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	ratio_to_median_purchase_price	re
0	57.877857	0.311140	1.945940	1.
1	10.829943	0.175592	1.294219	1.
2	5.091079	0.805153	0.427715	1.
3	2.247564	5.600044	0.362663	1.
4	44.190936	0.566486	2.222767	1.
5	5.586408	13.261073	0.064768	1.
6	3.724019	0.956838	0.278465	1.
7	4.848247	0.320735	1.273050	1.
8	0.876632	2.503609	1.516999	0.
9	8.839047	2.970512	2.361683	1.
10	14.263530	0.158758	1.136102	1.
11	13.592368	0.240540	1.370330	1.
12	765.282559	0.371562	0.551245	1.
13	2.131956	56.372401	6.358667	1.
14	13.955972	0.271522	2.798901	1.
15	179.665148	0.120920	0.535640	1.
16	114.519789	0.707003	0.516990	1.
17	3.589649	6.247458	1.846451	1.
18	11.085152	34.661351	2.530758	1.
19	6.194671	1.142014	0.307217	1.

```
df.describe()
```

	distance_from_home	distance_from_last_transaction	ratio_to_median_purchase_price	repeat_ret
count	1000000.000000	1000000.000000	1000000.000000	1000000.00
mean	26.628792	5.036519	1.824182	0.881536
std	65.390784	25.843093	2.799589	0.323157
min	0.004874	0.000118	0.004399	0.000000
25%	3.878008	0.296671	0.475673	1.000000
50%	9.967760	0.998650	0.997717	1.000000
75%	25.743985	3.355748	2.096370	1.000000
max	10632.723672	11851.104565	267.802942	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
```

3.8.1 Data Preprocessing

Since we have already explored the dataset in the previous notebook, 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)
```

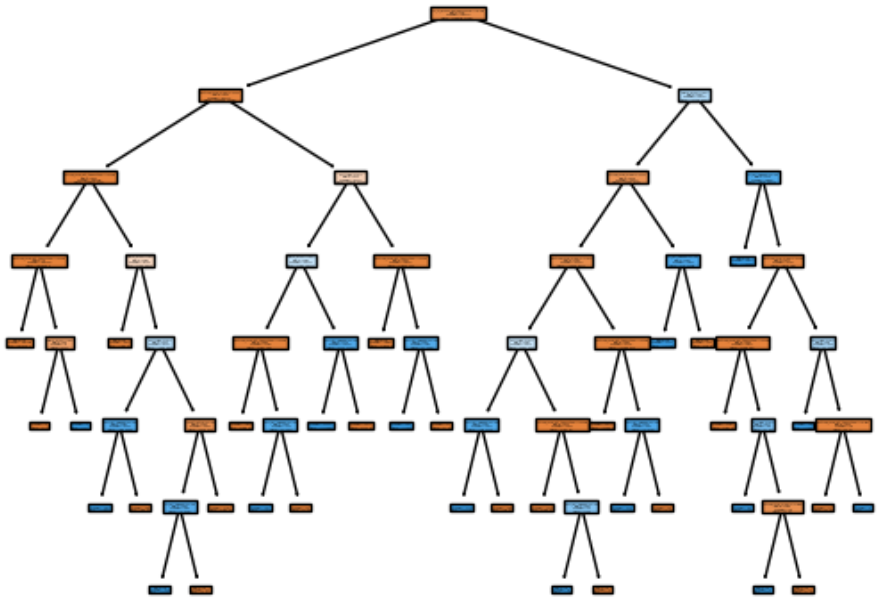
3.8.2 Implementing a Decision Tree Classifier

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)
```

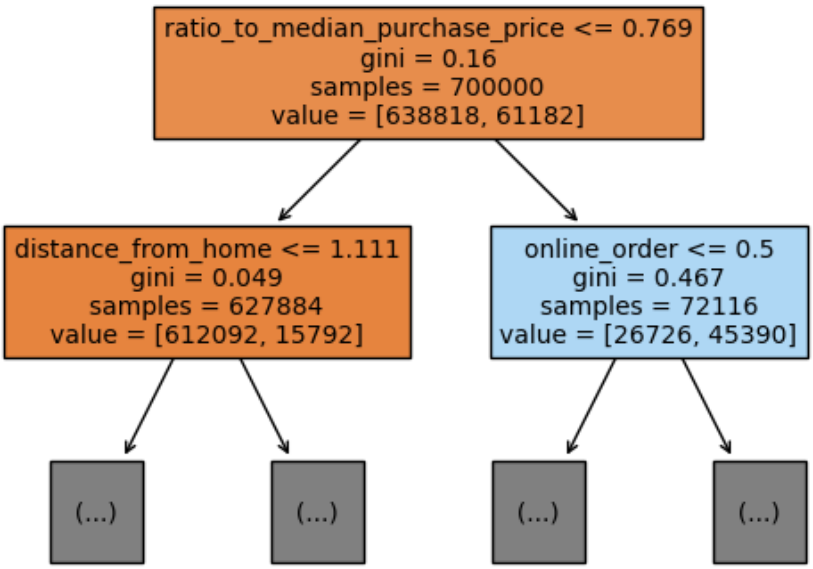
We can visualize the tree using the `plot_tree` function from the `sklearn.tree` module

```
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.

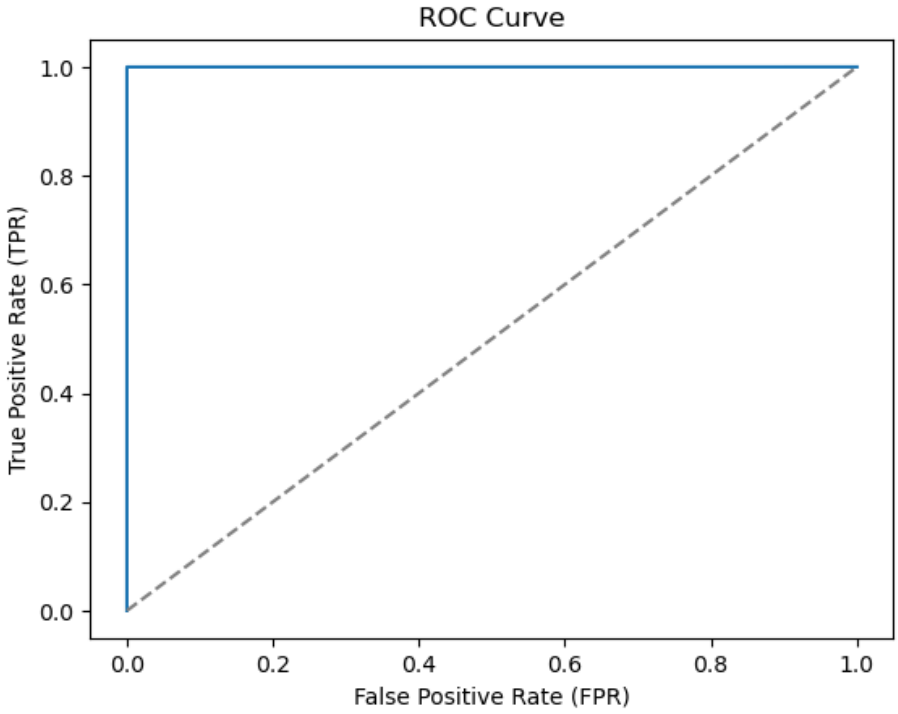
We can also make predictions using the model and evaluate its performance using the same functions as before

```
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)}")
print(f"Precision: {precision_score(y_test, y_pred_dt)}")
print(f"Recall: {recall_score(y_test, y_pred_dt)}")
print(f"ROC AUC: {roc_auc_score(y_test, y_proba_dt[:, 1])}")
```

```
Accuracy: 0.9999833333333333
Precision: 0.9999237223493517
Recall: 0.999885587887571
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.

3.8.3 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)}")
print(f"Precision: {precision_score(y_test, y_pred_rf)}")
print(f"Recall: {recall_score(y_test, y_pred_rf)}")
print(f"ROC AUC: {roc_auc_score(y_test, y_proba_rf[:, 1])}")
```

```
Accuracy: 0.9999833333333333
Precision: 1.0
Recall: 0.9998093131459517
ROC AUC: 0.9999999993035008
```

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.

3.8.4 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)}")
print(f"Precision: {precision_score(y_test, y_pred_xgb)}")
print(f"Recall: {recall_score(y_test, y_pred_xgb)}")
print(f"ROC AUC: {roc_auc_score(y_test, y_proba_xgb[:, 1])}")
```

```
Accuracy: 0.9983366666666667
Precision: 0.9893835616438356
```

```
Recall: 0.9916097784218756
ROC AUC: 0.999973496046352
```

Let's also check the confusion matrix to see where we still make mistakes

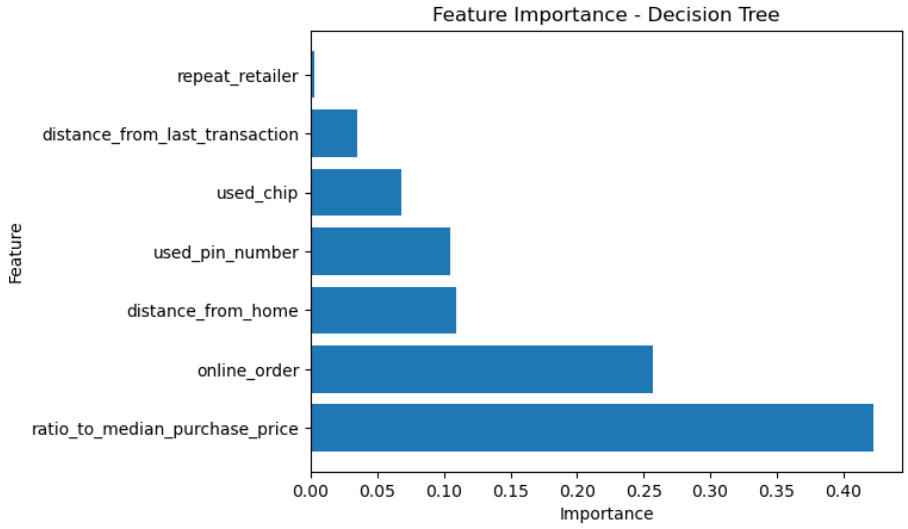
```
conf_mat = confusion_matrix(y_test, y_pred_xgb, labels=[1,
 ↪  0]).transpose() # Transpose the sklearn confusion matrix to
 ↪  match the convention in the lecture
sns.heatmap(conf_mat, annot=True, cmap='Blues', fmt='g',
 ↪  xticklabels=['Fraud', 'No Fraud'], yticklabels=['Fraud', 'No
 ↪  Fraud'])
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.

3.8.5 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()
```

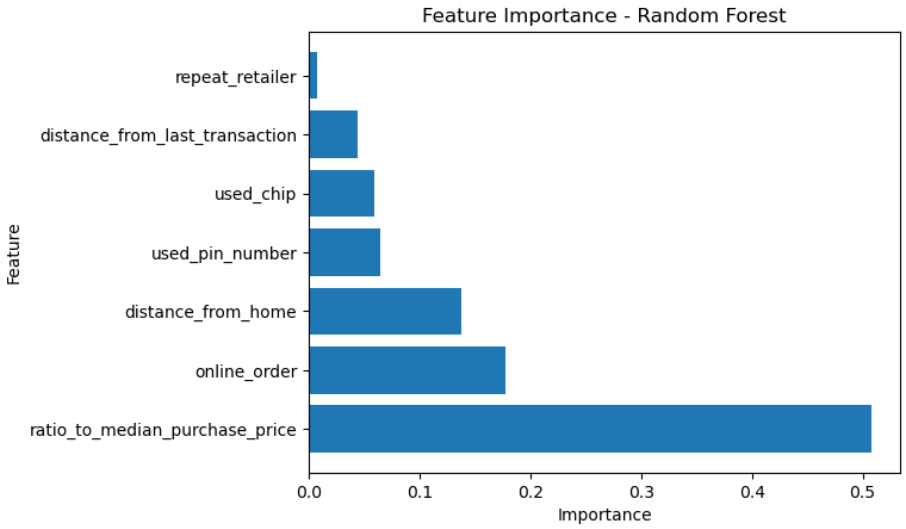


This shows that the `ratio_to_median_purchase_price` is the most important feature for determining whether a transaction is fraudulent or not. Whether a transaction is online, is important as well.

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_})
```

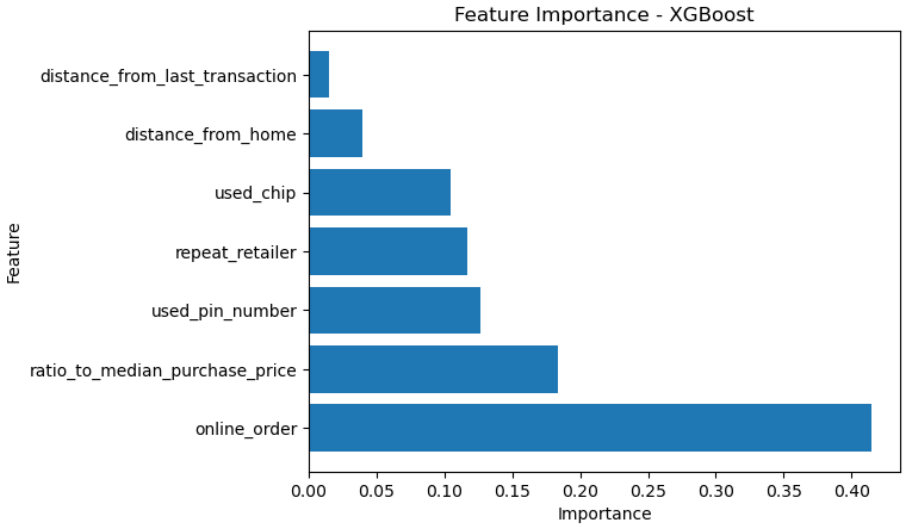
```
df_feature_importance_rf =
 ↪  df_feature_importance_rf.sort_values('Importance',
 ↪  ascending=False)

# 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()
```

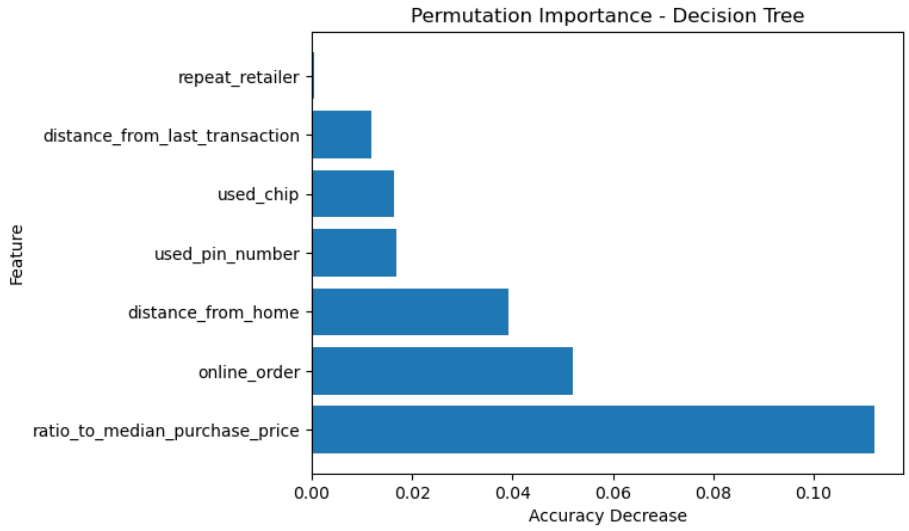


3.8.6 Permuation Importance

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

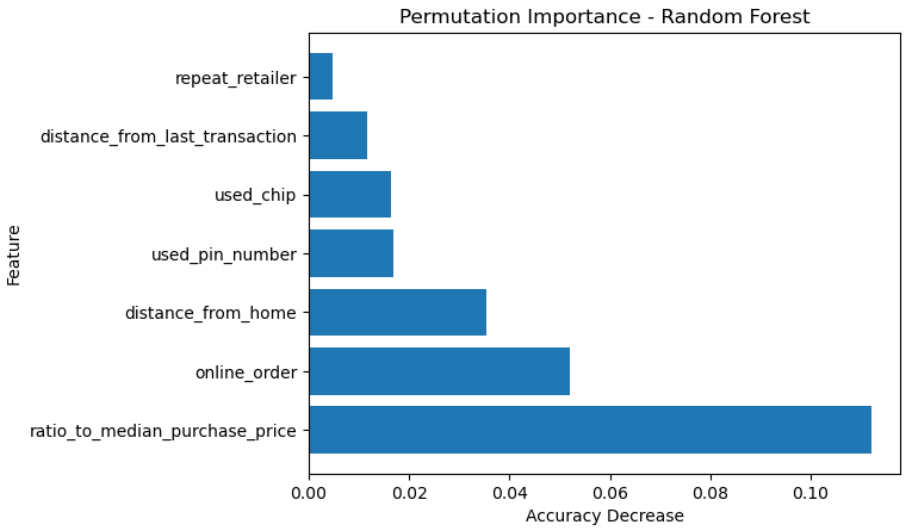
```
# Calculate the permutation importance
result_dt = permutation_importance(clf_dt, X_test, y_test,
 ↪  n_repeats=10, random_state=0)

# Create a DataFrame with the feature 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 feature 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()
```



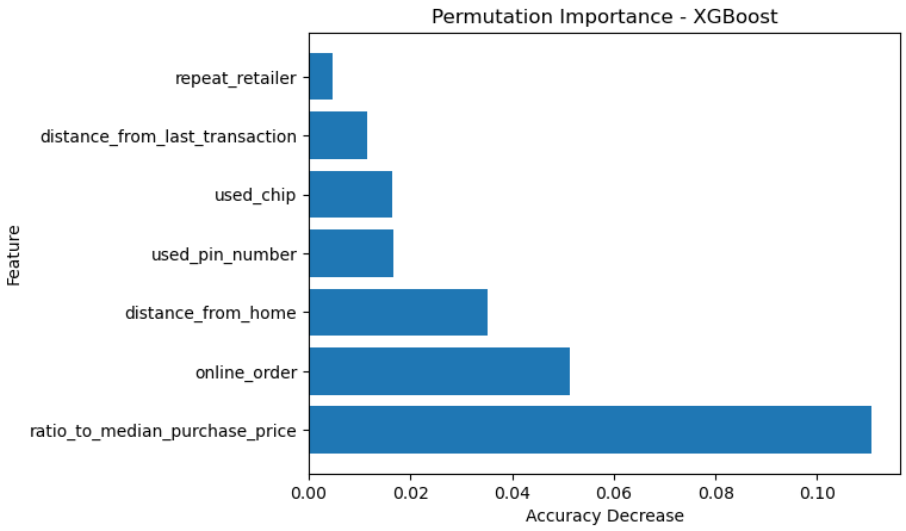
Let's also look at the permutation importance of the Random Forest model

```
# 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 feature 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 feature 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 feature 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 feature 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`.

3.8.7 Conclusions

In this notebook, 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 4

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.

4.1 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 4.1 shows a single-layer feedforward neural network with $N = 2$ 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 only consider 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

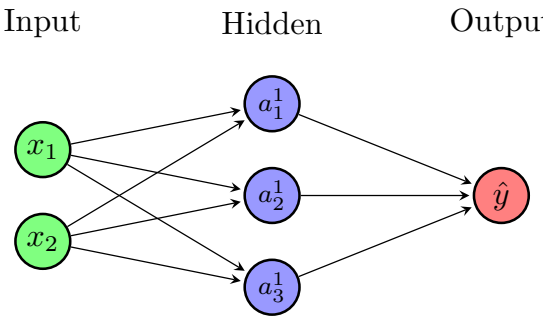


Figure 4.1: A Single-Layer Feedforward Neural Network

- Neural networks are **universal approximators** (can approximate any (Borel measurable) function)
- 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.

4.1.1 Origins of the Term "Neural Network"

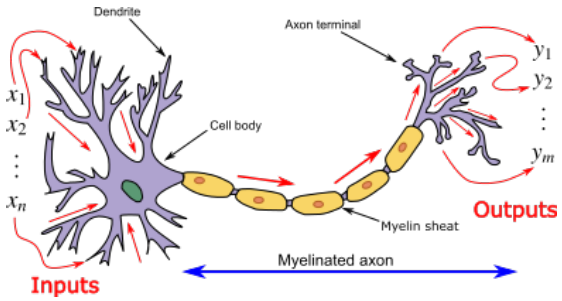


Figure 4.2: A biological neuron (Source: Wikipedia)

The term "neural network" originates in attempts to find mathematical representations of information processing in biological systems (Bishop 2006). The biological interpretation **not very important for research** anymore and one should not get too hung up on it. However, the interpretation can be useful when starting to learn about neural networks. Figure 4.2 shows a biological neuron.

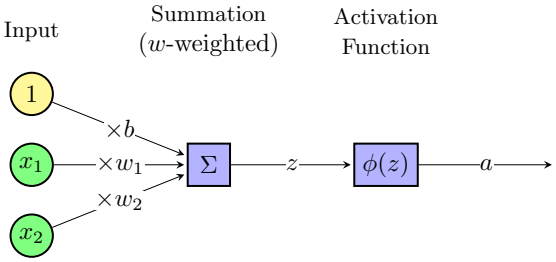


Figure 4.3: Artificial Neuron

4.2 An Artificial Neuron

Artificial neurons are the **basic building blocks** of neural networks. Figure 4.3 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.

4.2.1 Activation Functions

Common activation functions include

- Sigmoid: $\phi(z) = \frac{1}{1+e^{-z}}$
- Hyperbolic tangent: $\phi(z) = tanh(z)$
- Rectified linear unit (ReLU): $\phi(z) = \max(0, z)$
- Softplus: $\phi(z) = \log(1 + e^z)$

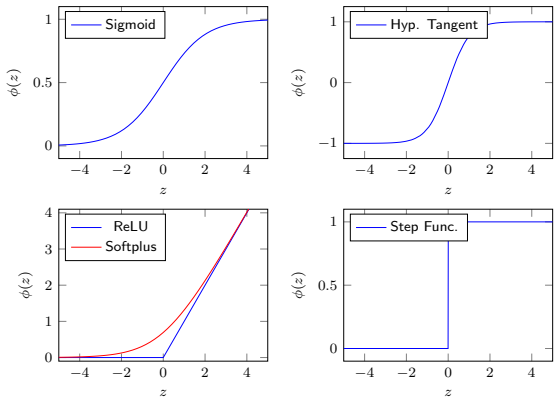


Figure 4.4: Activation Functions

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.

4.2.2 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 $b = -1$?

4.3 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 **single-layer neural network** 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 4.5 shows a single-layer feedforward neural network with $N = 2$ 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.

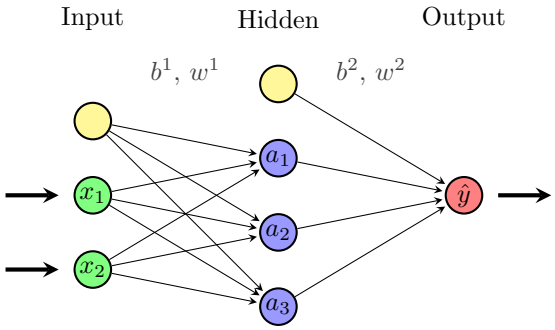


Figure 4.5: A Single-Layer Feedforward Neural Network with Biases shown explicitly

4.4 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 2.1 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.

4.5 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 4.6 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, the approximation is not very good. However, with more neurons, we could get a better approximation.

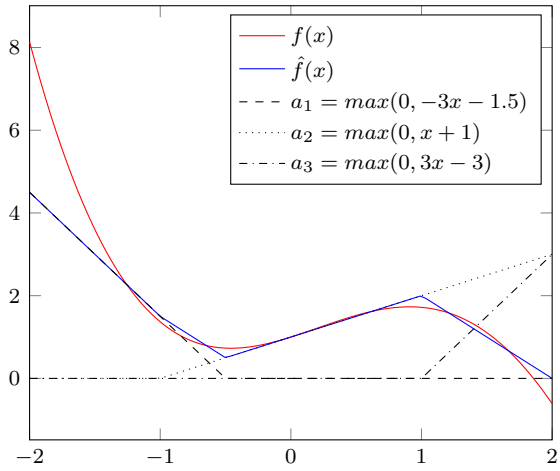


Figure 4.6: Approximation by a Neural Network

The HTML version of these notes shows an interactive version of Figure 4.6 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.

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 4.7 shows the interface of the TensorFlow Playground.

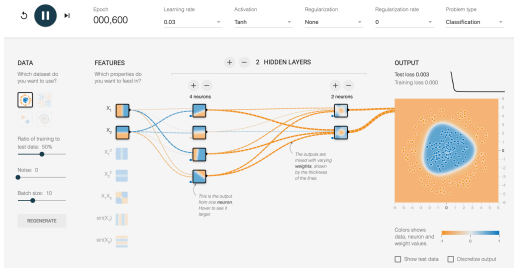


Figure 4.7: Tesorflow Playground

4.6 Deep Neural Networks

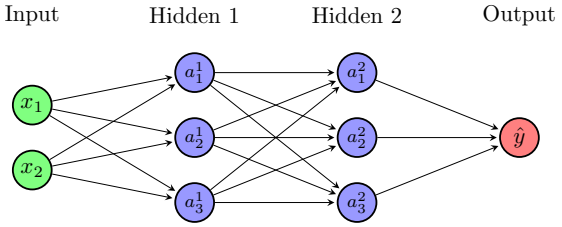


Figure 4.8: Deep Neural Network

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 learn more complicated things. For simple function approximation, a single hidden layer is sufficient. Figure 4.8 shows a deep neural network with two hidden layers.

The first hidden layer consists of M_1 artificial neurons with inputs $x_1, x_2, \ldots, x_N$

$$a_j^1 = \phi\left(b_j^1 + \sum_{i=1}^{N} w_{ji}^1 x_i\right)$$

The second hidden layer consists of M_2 artificial neurons with inputs $a_1^1, a_2^1, \ldots, a_{M_1}^1$

$$a_k^2 = \phi\left(b_k^2 + \sum_{j=1}^{M_1} w_{kj}^2 a_j^1\right)$$

After Q hidden layers, the output is defined as

$$y \cong g(x; w) = b^{Q+1} + \sum_{j=1}^{M_Q} w_j^{Q+1} a_j^Q$$

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.

4.7 Universal Approximation and the Curse of Dimensionality

Recall that we want to **approximate an unknown function** in supervised learning tasks

$$y = f(x)$$

where $y = (y_1, y_2, ..., y_K)'$ and $x = (x_1, x_2, ..., x_N)'$ 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.

4.8 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.

4.8.1 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.

4.8.2 Gradient Descent

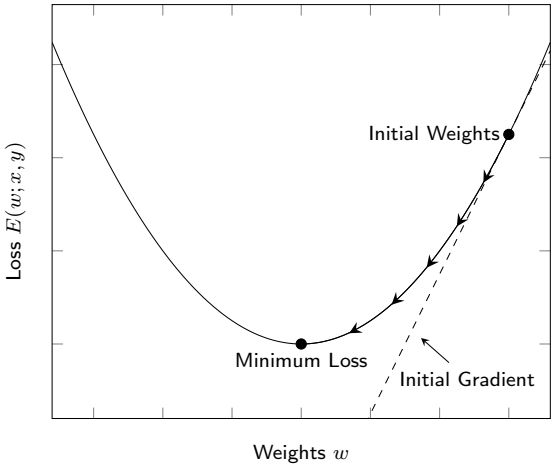


Figure 4.9: Gradient Descent

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 4.9 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 4.10 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.

4.8.3 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.,

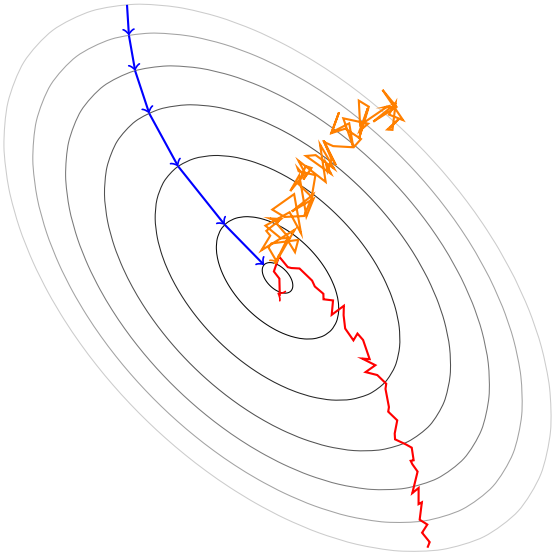


Figure 4.10: Comparison of Gradient Descent Types (blue: Full Batch, red: Minibatch, orange: Stochastic)

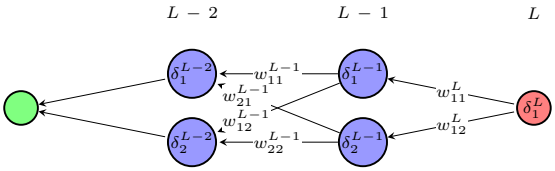


Figure 4.11: Backpropagation Algorithm

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 4.11. If you are interested in the details, I recommend reading the notes by Nielsen (2019).

4.9 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 lose 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.

4.10 Python Implementation

Let's have a look at how to implement a neural network in Python.

4.10.1 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

```
import numpy as np
```

Next, we define the activation function for which we use the sigmoid function

```
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

```
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} + {i=1}^N w{ji}^{1} x_i$

$ a_j = (z_j)$

and the output of the network is given by

$ g(x ; w) = b^{2}+ {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)
```

```
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.

4.10.2 Using Neural Networks in Sci-Kit Learn

Sci-kit 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.

```
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

# 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)
```

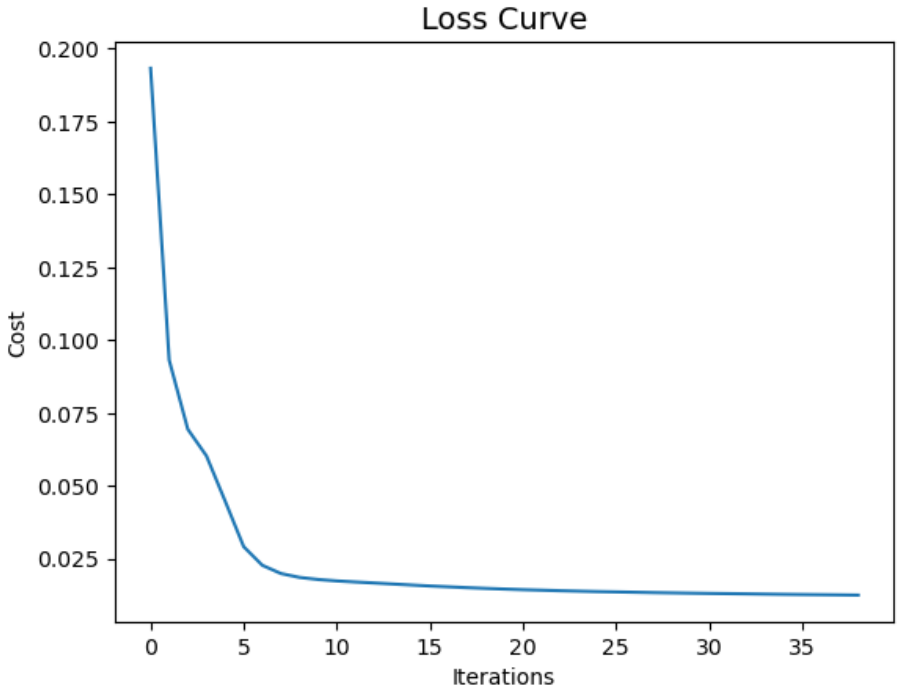
If you would like to use a neural network with multiple hidden layers, you can specify the number of nodes per hidden layer using the `hidden_layer_sizes` parameter. For example, the following code snippet shows how to use a neural network with two hidden layers, one with 5 nodes and the other with 4 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

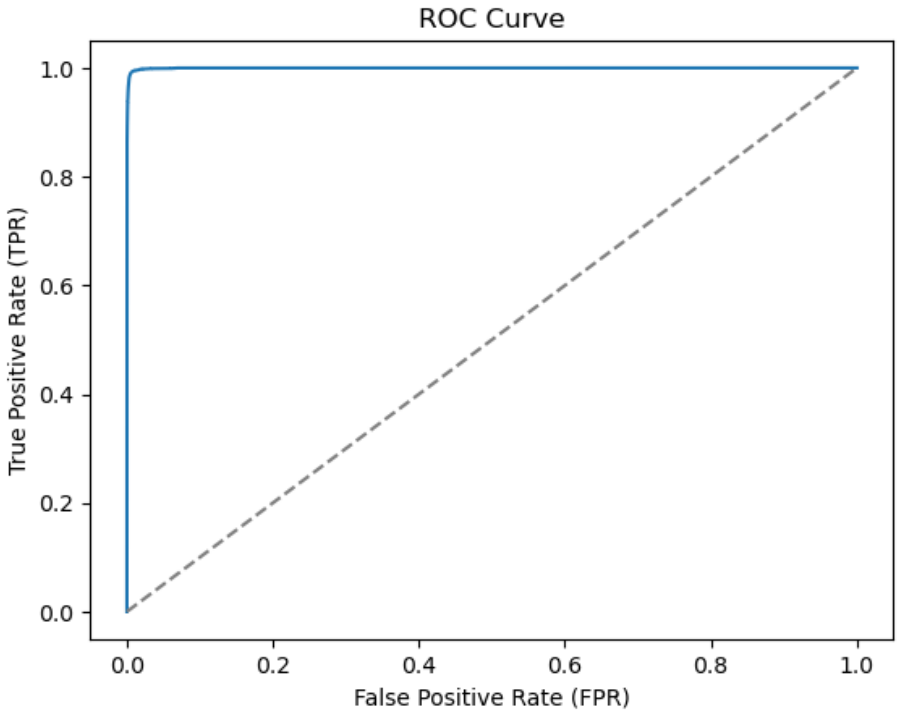
```
y_pred = clf.predict(X_test)
y_proba = clf.predict_proba(X_test)

print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
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])}")
```

```
Accuracy: 0.9955266666666667
Precision: 0.971747127308582
Recall: 0.9772319896266352
ROC AUC: 0.9996638991577014
```

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.

4.10.3 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: 0.002971
Epoch 19 loss: 0.010346
Epoch 29 loss: 0.010593
Epoch 39 loss: 0.004047
Epoch 49 loss: 0.014236
Epoch 59 loss: 0.006514
Epoch 69 loss: 0.004833
Epoch 79 loss: 0.011132
```

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 epoch 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}")
```

```
Epoch 99 loss: 0.006053
Epoch 199 loss: 0.005863
Epoch 299 loss: 0.005810
Epoch 399 loss: 0.005786
Epoch 499 loss: 0.005771
Epoch 599 loss: 0.005759
Epoch 699 loss: 0.005750
Epoch 799 loss: 0.005743
Epoch 899 loss: 0.005736
Epoch 999 loss: 0.005731
Epoch 1099 loss: 0.005726
Epoch 1199 loss: 0.005722
Epoch 1299 loss: 0.005719
Epoch 1399 loss: 0.005716
Epoch 1499 loss: 0.005714
Epoch 1599 loss: 0.005712
Epoch 1699 loss: 0.005710
Epoch 1799 loss: 0.005707
Epoch 1899 loss: 0.005705
Epoch 1999 loss: 0.005703
```

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)}")
print(f"Precision: {precision_score(y_test, y_pred)}")
print(f"Recall: {recall_score(y_test, y_pred)}")
```

```
Accuracy: 0.9987533333333334
```

```
Precision: 0.9946794258373206
Recall: 0.9910377178597307
```

Note that for simplicity we are reusing the sci-kit 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.

4.10.4 Conclusions

In this chapter, 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 5

Additional Methods

In this chapter, we will introduce some additional methods that are commonly used in machine learning. These methods include the K-Nearest Neighbors (KNN) algorithm and the K-means clustering algorithm.

5.1 K-Nearest Neighbors

The K-Nearest Neighbors (KNN) algorithm is a simple and intuitive method for classification and regression meaning that it belongs to the class of **supervised learning** methods. 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 5.1 shows an example of the K-Nearest Neighbors algorithm applied to a dataset with two classes. The decision boundary is shown as a shaded area.

5.2 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.

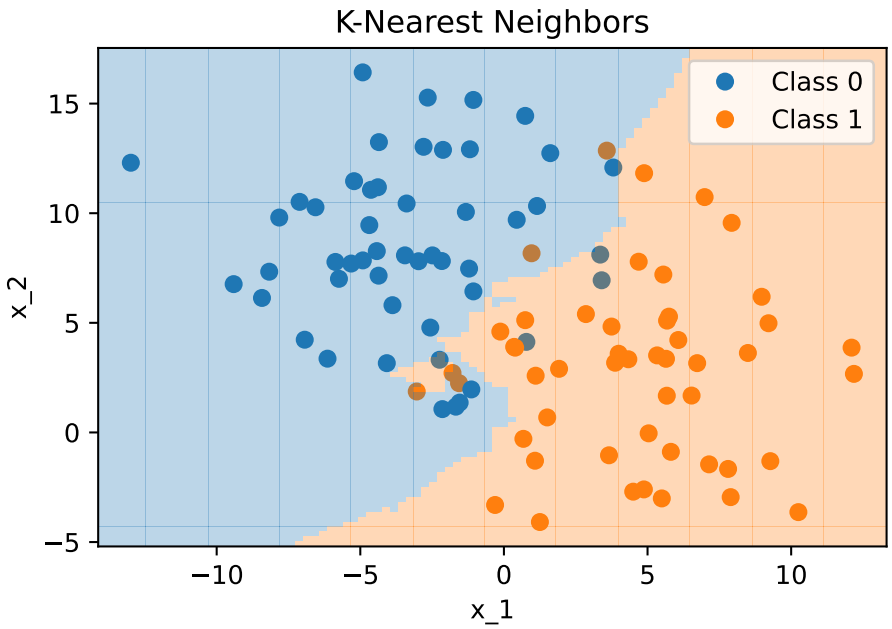


Figure 5.1: K-Nearest Neighbors Classification with $K = 5$ (Classification shown as Shaded Area)

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.

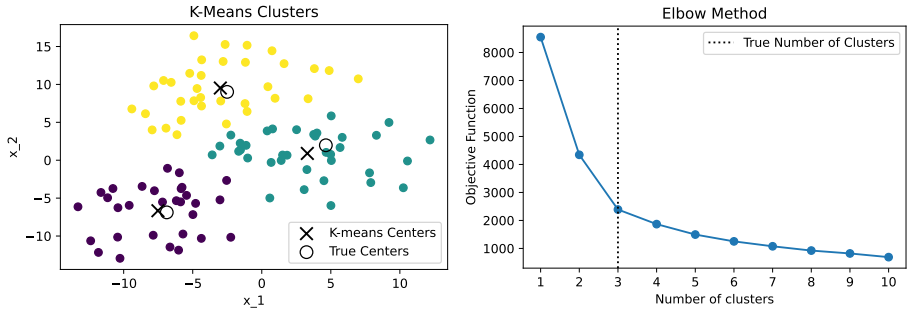


Figure 5.2: K-Means Clusters and Elbow Method

Figure 5.2 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.

5.3 Python Implementation

Let's have a look at how to implement KNN and K-means in Python. Again, we need to first import the required packages and load the data

```
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.neighbors import KNeighborsClassifier
from sklearn.cluster import KMeans
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

# 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	ratio_to_median_purchase_price	repeat_retaile
0	57.877857	0.311140	1.945940	1.0
1	10.829943	0.175592	1.294219	1.0
2	5.091079	0.805153	0.427715	1.0
3	2.247564	5.600044	0.362663	1.0
4	44.190936	0.566486	2.222767	1.0
5	5.586408	13.261073	0.064768	1.0
6	3.724019	0.956838	0.278465	1.0
7	4.848247	0.320735	1.273050	1.0
8	0.876632	2.503609	1.516999	0.0
9	8.839047	2.970512	2.361683	1.0
10	14.263530	0.158758	1.136102	1.0
11	13.592368	0.240540	1.370330	1.0
12	765.282559	0.371562	0.551245	1.0
13	2.131956	56.372401	6.358667	1.0
14	13.955972	0.271522	2.798901	1.0
15	179.665148	0.120920	0.535640	1.0
16	114.519789	0.707003	0.516990	1.0
17	3.589649	6.247458	1.846451	1.0
18	11.085152	34.661351	2.530758	1.0
19	6.194671	1.142014	0.307217	1.0

```
df.describe()
```

	distance_from_home	distance_from_last_transaction	ratio_to_median_purchase_price	repeat_ret
count	1000000.000000	1000000.000000	1000000.000000	1000000.00
mean	26.628792	5.036519	1.824182	0.881536
std	65.390784	25.843093	2.799589	0.323157
min	0.004874	0.000118	0.004399	0.000000
25%	3.878008	0.296671	0.475673	1.000000
50%	9.967760	0.998650	0.997717	1.000000
75%	25.743985	3.355748	2.096370	1.000000
max	10632.723672	11851.104565	267.802942	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
```

5.3.1 Data Preprocessing

Since we have already explored the dataset in the previous notebook, 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


# Define which features to scale with the StandardScaler and
 ↪  MinMaxScaler
for_standard_scaler = [
    'distance_from_home',
    'distance_from_last_transaction',
    'ratio_to_median_purchase_price',
]

# Apply the standard scaler (Note: we use the same mean and std
 ↪  for scaling the test set)
standard_scaler = StandardScaler()
scale_features(standard_scaler, X_train, for_standard_scaler)
scale_features(standard_scaler, X_test, for_standard_scaler,
 ↪  only_transform=True)
```

5.3.2 K-Nearest Neighbors (KNN)

We can now implement the KNN algorithm using the `KNeighborsClassifier` class from the `sklearn.neighbors` module. We will use the default value of $k = 5$ for the number of neighbors.

```
clf_knn = KNeighborsClassifier().fit(X_train, y_train)
```

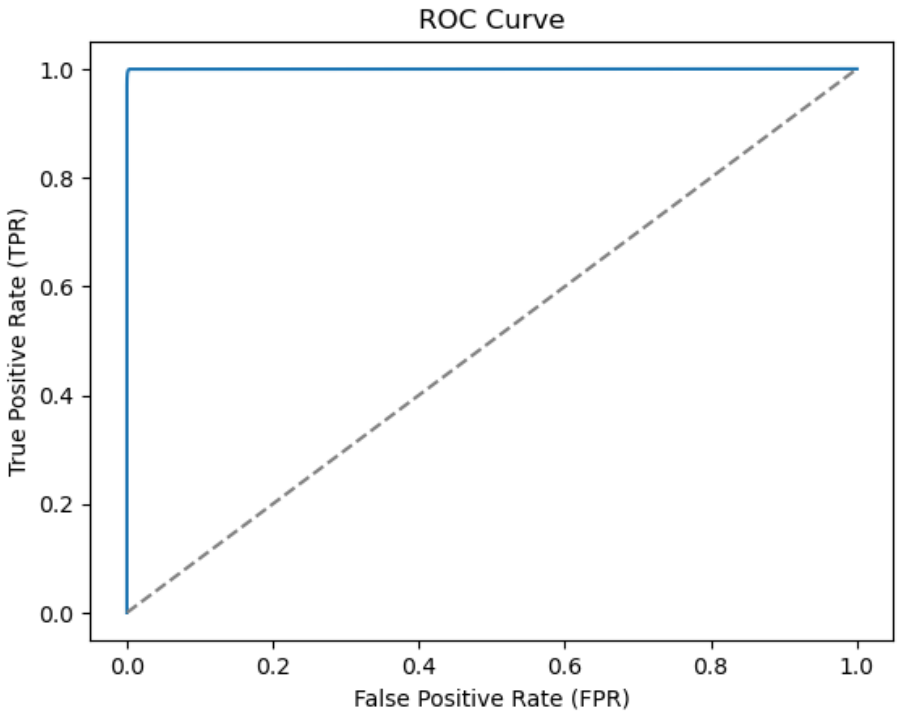
We can now use the trained model to make predictions on the test set and evaluate the model performance using the confusion matrix and accuracy score.

```
y_pred_knn = clf_knn.predict(X_test)
y_proba_knn = clf_knn.predict_proba(X_test)

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, y_proba_knn[:, 1])}")
```

```
Accuracy: 0.9987
Precision: 0.9935419771485345
Recall: 0.991571641051066
ROC AUC: 0.9997341251520317
```

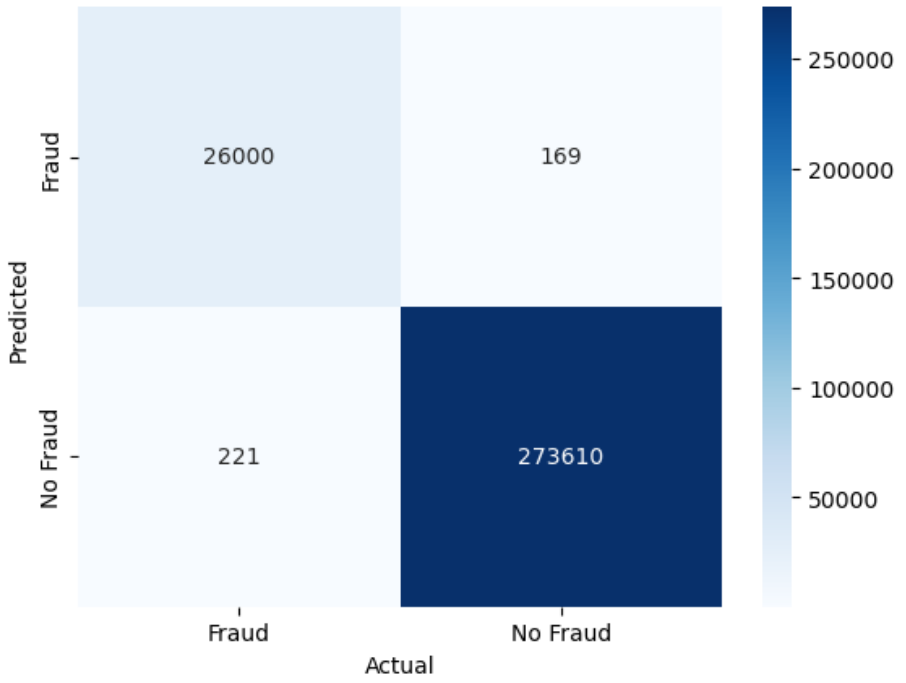
This seems to work quite well with a ROC AUC of 0.9997. We seem to have an almost perfect classifier. We can also plot the ROC curve to visualize the performance of the classifier

```
# Compute the ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_proba_knn[:, 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_knn, 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()
```



5.3.3 K-Means

This is the first example of an unsupervised learning algorithm meaning that we will ignore the labels in the training set. We will use the `KMeans` class from the `sklearn.cluster` module to implement the K-means algorithm. Note that we can not use categorical variables in the K-means algorithm, so we will only use the continuous variables in this example. Furthermore, to simplify interpretability we will only use two variables

```
continuous_variables = ['distance_from_home',
 ↪  'distance_from_last_transaction',
 ↪  'ratio_to_median_purchase_price']
n_clusters=2
kmeans = KMeans(n_clusters=n_clusters, random_state=42,
 ↪  n_init=10).fit(X_train[continuous_variables])
```

We can check the cluster centers using the `cluster_centers_` attribute of the trained model

```
kmeans.cluster_centers_
```

```
array([[-2.01860525e-05, -1.55050548e-03, -1.68843633e-01],
```

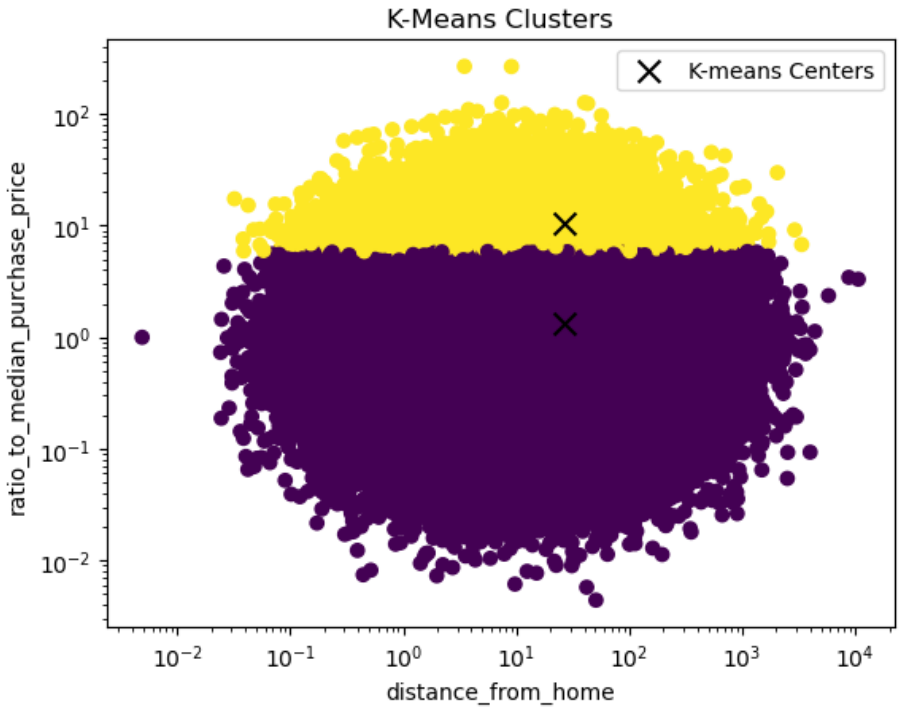
```
       [ 3.66287246e-04,  2.81347918e-02,  3.06376244e+00]])
```

Since we only have two variables we can easily visualize the clusters using a scatter plot. We first need to unscale the data to make the plot more interpretable

```
# Unscale the data
X_train_unscaled = X_train.copy()
X_train_unscaled[for_standard_scaler] =
 ↪  standard_scaler.inverse_transform(X_train[for_standard_scaler])
X_test_unscaled = X_test.copy()
X_test_unscaled[for_standard_scaler] =
 ↪  standard_scaler.inverse_transform(X_test[for_standard_scaler])
cluster_centers =
 ↪  standard_scaler.inverse_transform(kmeans.cluster_centers_)
```

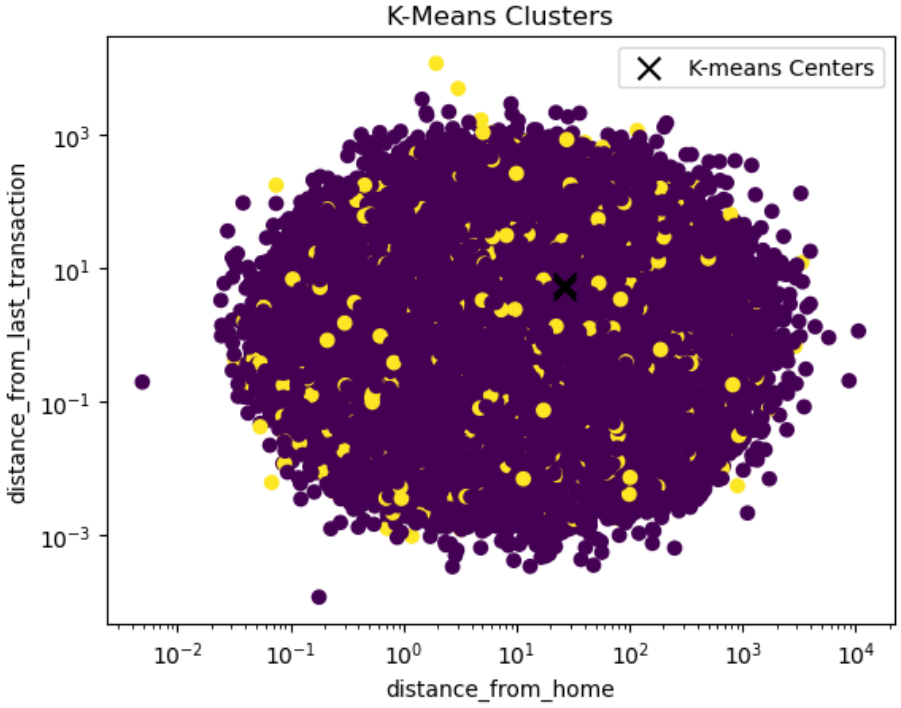
Then, we can create the scatter plot to see what the clusters look like

```
_, ax = plt.subplots()
scatter = ax.scatter(X_train_unscaled[continuous_variables[0]],
 ↪  X_train_unscaled[continuous_variables[2]], c=kmeans.labels_)
scatter = ax.scatter(cluster_centers[:, 0], cluster_centers[:,
 ↪  2], c='black', marker='x', s=100, label = 'K-means Centers')
ax.set(xlabel=continuous_variables[0],
 ↪  ylabel=continuous_variables[2])
ax.set_xscale('log')
ax.set_yscale('log')
ax.legend()
plt.title('K-Means Clusters')
plt.show()
```



Note that the centers might look a bit off because we are using log scales on the x and y-axis. In the other dimension, we don't have such a nice separation of the clusters

```
_, ax = plt.subplots()
scatter = ax.scatter(X_train_unscaled[continuous_variables[0]],
 ↪  X_train_unscaled[continuous_variables[1]], c=kmeans.labels_)
scatter = ax.scatter(cluster_centers[:, 0], cluster_centers[:,
 ↪  1], c='black', marker='x', s=100, label = 'K-means Centers')
ax.set(xlabel=continuous_variables[0],
 ↪  ylabel=continuous_variables[1])
ax.set_xscale('log')
ax.set_yscale('log')
ax.legend()
plt.title('K-Means Clusters')
plt.show()
```



But what do these two clusters represent? We can check the mean of the target variable `fraud` for each cluster to get an idea of what the clusters represent

```
X_train_unscaled['cluster'] = kmeans.labels_
X_train_unscaled.query('cluster == 1').describe().T
```

	count	mean	std	min	25%	50%
distance_from_home	36679.0	26.727628	63.910540	0.032026	3.895800	10.09870
distance_from_last_transaction	36679.0	5.780037	71.723799	0.000966	0.296198	1.000376
ratio_to_median_purchase_price	36679.0	10.470287	6.811775	2.209891	6.871869	8.384989
repeat_retailer	36679.0	0.879522	0.325524	0.000000	1.000000	1.000000
used_chip	36679.0	0.351754	0.477524	0.000000	0.000000	0.000000
used_pin_number	36679.0	0.102756	0.303645	0.000000	0.000000	0.000000
online_order	36679.0	0.649063	0.477270	0.000000	0.000000	1.000000
cluster	36679.0	1.000000	0.000000	1.000000	1.000000	1.000000

```
X_train_unscaled.query('cluster == 0').describe().T
```

	count	mean	std	min	25%	50%	75%
distance_from_home	663321.0	26.694233	66.097113	0.004874	3.880252	9.969293	25.80790
distance_from_last_transaction	663321.0	4.988030	22.054240	0.000118	0.296681	0.998050	3.351187
ratio_to_median_purchase_price	663321.0	1.347721	1.226094	0.004399	0.455014	0.929070	1.838841
repeat_retailer	663321.0	0.881468	0.323238	0.000000	1.000000	1.000000	1.000000
used_chip	663321.0	0.350518	0.477133	0.000000	0.000000	0.000000	1.000000
used_pin_number	663321.0	0.100512	0.300682	0.000000	0.000000	0.000000	0.000000
online_order	663321.0	0.650643	0.476767	0.000000	0.000000	1.000000	1.000000
cluster	663321.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

```
y_train[X_train_unscaled['cluster'] == 0].mean()
```

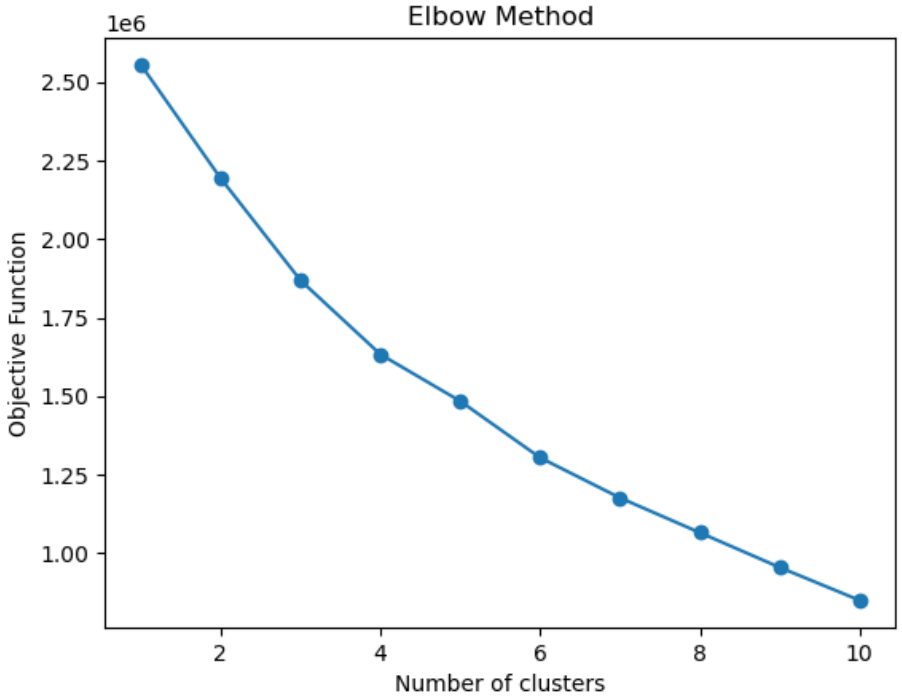
```
0.057474435454327545
```

```
y_train[X_train_unscaled['cluster'] == 1].mean()
```

```
0.6286430927778838
```

There does not seem to be a clear difference between the two clusters except for the difference in the mean of the ratio_to_median_purchase_price variable. This is not necessarily very surprising since we only used three variables in the clustering algorithm. However, due to the correlation of `ratio_to_median_purchase_price` we have more fraudulent transactions in one cluster than the other. To be able to carry out a more meaningful clustering analysis using K-means we would need a different dataset with more quantitative variables. Nevertheless, let's also check the elbow method to how many clusters it would suggest

```
interias = [KMeans(n_clusters=n, n_init=10).fit(X_train).inertia_
 ↪  for n in range(1, 11)]
_, ax = plt.subplots()
ax.plot(range(1, 11), interias, marker='o')
ax.set(xlabel='Number of clusters', ylabel='Objective Function')
plt.title('Elbow Method')
plt.show()
```



There does not seem to be a clear elbow in the plot. Finally, we can also make predictions on the test set using the trained K-means model

```
kmeans.predict(X_test[continuous_variables])
```

```
array([0, 0, 0, ..., 0, 0, 1], dtype=int32)
```

This assigns each observation in the test set to one of the two clusters.

5.3.4 Conclusions

We have seen how to implement a KNN algorithm for classification and a K-means algorithm for clustering in Python using the `sklearn` package. We have also seen how to evaluate the performance of the KNN algorithm using the confusion matrix, accuracy score, precision, recall, and ROC AUC. We have also seen how to visualize the clusters created by the K-means algorithm and tried to apply the ellow method.

Part II

Applications

Chapter 6

Loan Default Prediction

The following application 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 first 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()
```

Tab

Variable Name	Description
SeriousDlqin2yrs	Person experienced 90 days past due delinquency or wor
RevolvingUtilizationOfUnsecuredLines	Total balance on credit cards and personal lines of credit
age	Age of borrower in years
NumberOfTime30-59DaysPastDueNotWorse	Number of times borrower has been 30-59 days past due
DebtRatio	Monthly debt payments, alimony,living costs divided by
MonthlyIncome	Monthly income
NumberOfOpenCreditLinesAndLoans	Number of Open loans (installment like car loan or mort
NumberOfTimes90DaysLate	Number of times borrower has been 90 days or more pas
NumberRealEstateLoansOrLines	Number of mortgage and real estate loans including hom
NumberOfTime60-89DaysPastDueNotWorse	Number of times borrower has been 60-89 days past due
NumberOfDependents	Number of dependents in family excluding themselves (s

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 occured) and 0 if the loan does not default (i.e., no serious delinquency occured) . 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 Data Preprocessing

The dataset is now loaded into a pandas DataFrame. Let's have a look at the first few rows of the dataset to get an idea of what the data looks like.

```
df.head()
```

	Unnamed: 0	SeriousDlqin2yrs	RevolvingUtilizationOfUnsecuredLines	age	NumberOfTime30-59DaysPast
0	1	1	0.766127	45	2
1	2	0	0.957151	40	0
2	3	0	0.658180	38	1
3	4	0	0.233810	30	0
4	5	0	0.907239	49	1

The column `Unnamed: 0` seems to be a superfluous index column that we could drop. Let's do that

```
df = df.drop('Unnamed: 0', axis=1)
```

Furthermore, the order of the column names in the dataset is not very intuitive. Let's reorder the columns in the dataset

```
orderedList = [
    'SeriousDlqin2yrs',
    'age',
    'NumberOfDependents',
    'MonthlyIncome',
```

```
    'DebtRatio',
    'RevolvingUtilizationOfUnsecuredLines',
    'NumberOfOpenCreditLinesAndLoans',
    'NumberRealEstateLoansOrLines',
    'NumberOfTime30-59DaysPastDueNotWorse',
    'NumberOfTime60-89DaysPastDueNotWorse',
    'NumberOfTimes90DaysLate'
]

df = df.loc[:, orderedList]
```

Let's also have a look at the data types of the columns in the dataset and whether there are any missing values

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150000 entries, 0 to 149999
Data columns (total 11 columns):
 #   Column                                Non-Null Count   Dtype
---  ------                                --------------   -----
 0   SeriousDlqin2yrs                      150000 non-null  int64
 1   age                                   150000 non-null  int64
 2   NumberOfDependents                    146076 non-null  float64
 3   MonthlyIncome                         120269 non-null  float64
 4   DebtRatio                             150000 non-null  float64
 5   RevolvingUtilizationOfUnsecuredLines  150000 non-null  float64
 6   NumberOfOpenCreditLinesAndLoans       150000 non-null  int64
 7   NumberRealEstateLoansOrLines          150000 non-null  int64
 8   NumberOfTime30-59DaysPastDueNotWorse  150000 non-null  int64
 9   NumberOfTime60-89DaysPastDueNotWorse  150000 non-null  int64
 10  NumberOfTimes90DaysLate               150000 non-null  int64
dtypes: float64(4), int64(7)
memory usage: 12.6 MB
```

Note that the column `MonthlyIncome` and `NumberOfDependents` seem to have missing values. Before we drop these missing values or impute them, let's have a look at the distribution of our target variable `SeriousDlqin2yrs`

```
df['SeriousDlqin2yrs'].value_counts(normalize=True)
```

```
SeriousDlqin2yrs
0    0.93316
1    0.06684
Name: proportion, dtype: float64
```

As with the example that we have seen during one of our previous lectures,

the dataset seems to be quite imbalanced. Only about 6.7% of the loans have defaulted. This is something that we need to keep in mind when treating the missing values and when building our models.

Let's see what happens to the distribution of the target variable if we drop the missing values

```
df.dropna().value_counts("SeriousDlqin2yrs", normalize=True)
```

```
SeriousDlqin2yrs
0    0.930514
1    0.069486
Name: proportion, dtype: float64
```

It seems to have almost no impact on the distribution of the target variable. This is good news. Let's compare some other statistics of the dataset before and after dropping the missing values

```
df.describe().T
```

	count	mean	std	min	25%	50%
SeriousDlqin2yrs	150000.0	0.066840	0.249746	0.0	0.000000	0.000
age	150000.0	52.295207	14.771866	0.0	41.000000	52.00
NumberOfDependents	146076.0	0.757222	1.115086	0.0	0.000000	0.000
MonthlyIncome	120269.0	6670.221237	14384.674215	0.0	3400.000000	5400.
DebtRatio	150000.0	353.005076	2037.818523	0.0	0.175074	0.366
RevolvingUtilizationOfUnsecuredLines	150000.0	6.048438	249.755371	0.0	0.029867	0.154
NumberOfOpenCreditLinesAndLoans	150000.0	8.452760	5.145951	0.0	5.000000	8.000
NumberRealEstateLoansOrLines	150000.0	1.018240	1.129771	0.0	0.000000	1.000
NumberOfTime30-59DaysPastDueNotWorse	150000.0	0.421033	4.192781	0.0	0.000000	0.000
NumberOfTime60-89DaysPastDueNotWorse	150000.0	0.240387	4.155179	0.0	0.000000	0.000
NumberOfTimes90DaysLate	150000.0	0.265973	4.169304	0.0	0.000000	0.000

```
df.dropna().describe().T
```

	count	mean	std	min	25%	50%
SeriousDlqin2yrs	120269.0	0.069486	0.254280	0.0	0.000000	0.000
age	120269.0	51.289792	14.426684	0.0	40.000000	51.00
NumberOfDependents	120269.0	0.851832	1.148391	0.0	0.000000	0.000
MonthlyIncome	120269.0	6670.221237	14384.674215	0.0	3400.000000	5400.
DebtRatio	120269.0	26.598777	424.446457	0.0	0.143388	0.296
RevolvingUtilizationOfUnsecuredLines	120269.0	5.899873	257.040685	0.0	0.035084	0.177
NumberOfOpenCreditLinesAndLoans	120269.0	8.758475	5.172835	0.0	5.000000	8.000
NumberRealEstateLoansOrLines	120269.0	1.054519	1.149273	0.0	0.000000	1.000
NumberOfTime30-59DaysPastDueNotWorse	120269.0	0.381769	3.499234	0.0	0.000000	0.000

	count	mean	std	min	25%
NumberOfTime60-89DaysPastDueNotWorse	120269.0	0.187829	3.447901	0.0	0.000000
NumberOfTimes90DaysLate	120269.0	0.211925	3.465276	0.0	0.000000

It looks like the statistics before and after dropping the missing values are quite similar, except for the variable `DebtRatio`, where we have substantially lower means and standard deviation. Let's also have a look at the distribution of the variables for the rows that we have dropped

```
df.loc[df.isna().any(axis=1)].describe().T
```

	count	mean	std	min	25%
SeriousDlqin2yrs	29731.0	0.056137	0.230189	0.0	0.000000
age	29731.0	56.362349	15.438786	21.0	46.000000
NumberOfDependents	25807.0	0.316310	0.809944	0.0	0.000000
MonthlyIncome	0.0	NaN	NaN	NaN	NaN
DebtRatio	29731.0	1673.396556	4248.372895	0.0	123.00000
RevolvingUtilizationOfUnsecuredLines	29731.0	6.649421	217.814854	0.0	0.016027
NumberOfOpenCreditLinesAndLoans	29731.0	7.216071	4.842720	0.0	4.000000
NumberRealEstateLoansOrLines	29731.0	0.871481	1.034291	0.0	0.000000
NumberOfTime30-59DaysPastDueNotWorse	29731.0	0.579866	6.255361	0.0	0.000000
NumberOfTime60-89DaysPastDueNotWorse	29731.0	0.452995	6.242076	0.0	0.000000
NumberOfTimes90DaysLate	29731.0	0.484612	6.250408	0.0	0.000000

Again, the mean of the dropped rows seems to be substantially higher for the variable `DebtRatio` suggesting that the missing values are not missing entirely at random. Note, however, that the standard deviation is lower meaning that the dropped observations are more similar to each other in the `DebtRatio` dimension. From our data dictionary, we know that the `DebtRatio` is defined as

```
data_dict.loc[data_dict['Variable Name'] ==
 ↪  'DebtRatio'].style.hide()
```

Table 6.6

Variable Name	Description	Type
DebtRatio	Monthly debt payments, alimony,living costs divided by monthy gross income	percer

So, it seems that the `DebtRatio` is the ratio of the monthly debt payments to the monthly gross income. We actually have `MonthlyIncome` in our dataset!

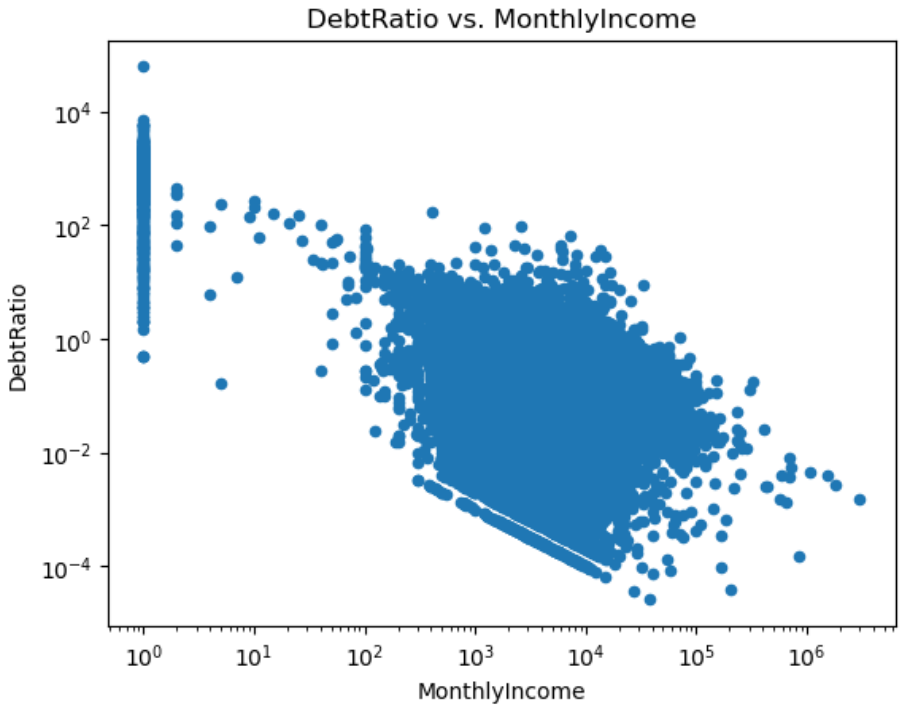
```
data_dict.loc[data_dict['Variable Name'] ==
 ↪  'MonthlyIncome'].style.hide()
```

Table 6.7

Variable Name	Description	Type
MonthlyIncome	Monthly income	real

It does not say whether this is in gross or net terms though. Nevertheless, let's have a look at the relationship between the `DebtRatio` and the `MonthlyIncome`

```
ax = df.plot.scatter(x='MonthlyIncome', y='DebtRatio')
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel('MonthlyIncome')
ax.set_ylabel('DebtRatio')
ax.set_title('DebtRatio vs. MonthlyIncome')
plt.show()
```



This looks rather odd. Note how there are a lot of monthly incomes that are close to zero. Furthermore, there is a weird gap going through the scatter points.

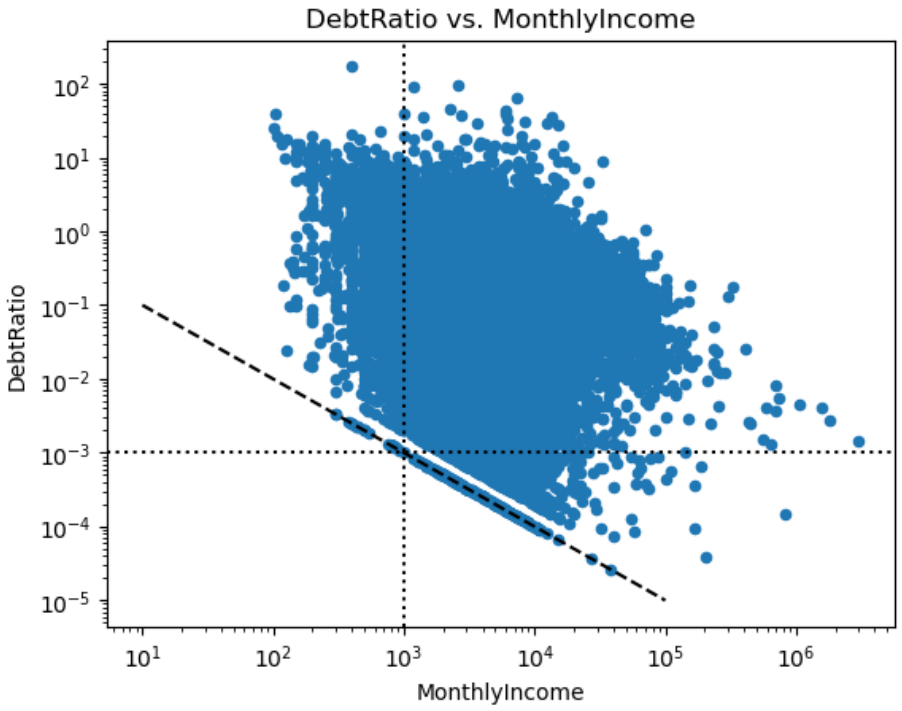
We can look at the descriptive statistics of the rows with `MonthlyIncome` less than 100

```
df.query('MonthlyIncome <= 100').describe().T
```

	count	mean	std	min	25%
SeriousDlqin2yrs	2301.0	0.036506	0.187586	0.0	0.000000
age	2301.0	47.740113	16.199176	21.0	35.000000
NumberOfDependents	2301.0	0.778792	1.192441	0.0	0.000000
MonthlyIncome	2301.0	1.837027	11.408271	0.0	0.000000
DebtRatio	2301.0	1370.529300	2752.843610	0.0	79.000000
RevolvingUtilizationOfUnsecuredLines	2301.0	3.604101	125.553453	0.0	0.022243
NumberOfOpenCreditLinesAndLoans	2301.0	7.171230	4.869628	0.0	4.000000
NumberRealEstateLoansOrLines	2301.0	0.742721	0.904984	0.0	0.000000
NumberOfTime30-59DaysPastDueNotWorse	2301.0	0.549326	6.132226	0.0	0.000000
NumberOfTime60-89DaysPastDueNotWorse	2301.0	0.428509	6.124274	0.0	0.000000
NumberOfTimes90DaysLate	2301.0	0.438505	6.128357	0.0	0.000000

These observations seem to have a higher `debtRatio` than the rest of the dataset but are less likely to default on their loans (the mean of `SeriousDlqin2yrs` is equal to the fraction of defaulting loans). Given that they have no income (or essentially no income), this seems rather odd and is likely due to an error during data entry/collection. Since there are only a small number of observations with `MonthlyIncome` less than 100, we can probably drop them. Let's look at the same figure for `MonthlyIncome` greater than 100

```
ax = df.query('MonthlyIncome >
 ↪  100').plot.scatter(x='MonthlyIncome', y='DebtRatio')
ax.set_xscale('log')
ax.set_yscale('log')
ax.axvline(10**3, color='black', linestyle=':')
ax.axhline(10**(-3), color='black', linestyle=':')
ax.plot([10, 10**5], [10**(-1), 10**(-5)], color='black',
 ↪  linestyle='--')
ax.set_xlabel('MonthlyIncome')
ax.set_ylabel('DebtRatio')
ax.set_title('DebtRatio vs. MonthlyIncome')
plt.show()
```

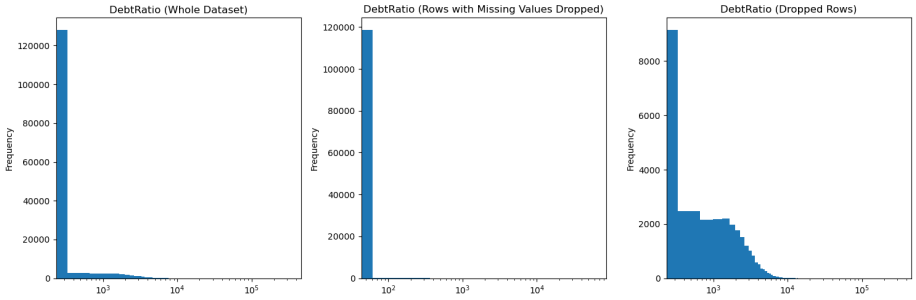


This looks better but note how the scatter points below the gap seem to line up with the line $\frac{1}{\text{MonthlyIncome}}$. Thus, there seems to be another potential data entry/collection error since the debt in the raw data has likely been just set to 1 for these observations. If this was a real dataset, we would need to investigate this further and maybe talk to the people who have sent us the data. However, given that this is just an example, we leave it as is.

Let's also have a look at the distribution of `DebtRatio` variable

```
fig, ax = plt.subplots(1, 3, figsize=(15, 5))

df['DebtRatio'].plot.hist(bins=1000, ax=ax[0])
df.dropna()['DebtRatio'].plot.hist(bins=1000, ax=ax[1])
df.loc[df.isna().any(axis=1), 'DebtRatio'].plot.hist(bins=1000,
 ↪  ax=ax[2])

ax[0].set_xscale('log')
ax[1].set_xscale('log')
ax[2].set_xscale('log')

ax[0].set_title('DebtRatio (Whole Dataset)')
ax[1].set_title('DebtRatio (Rows with Missing Values Dropped)')
ax[2].set_title('DebtRatio (Dropped Rows)')
```

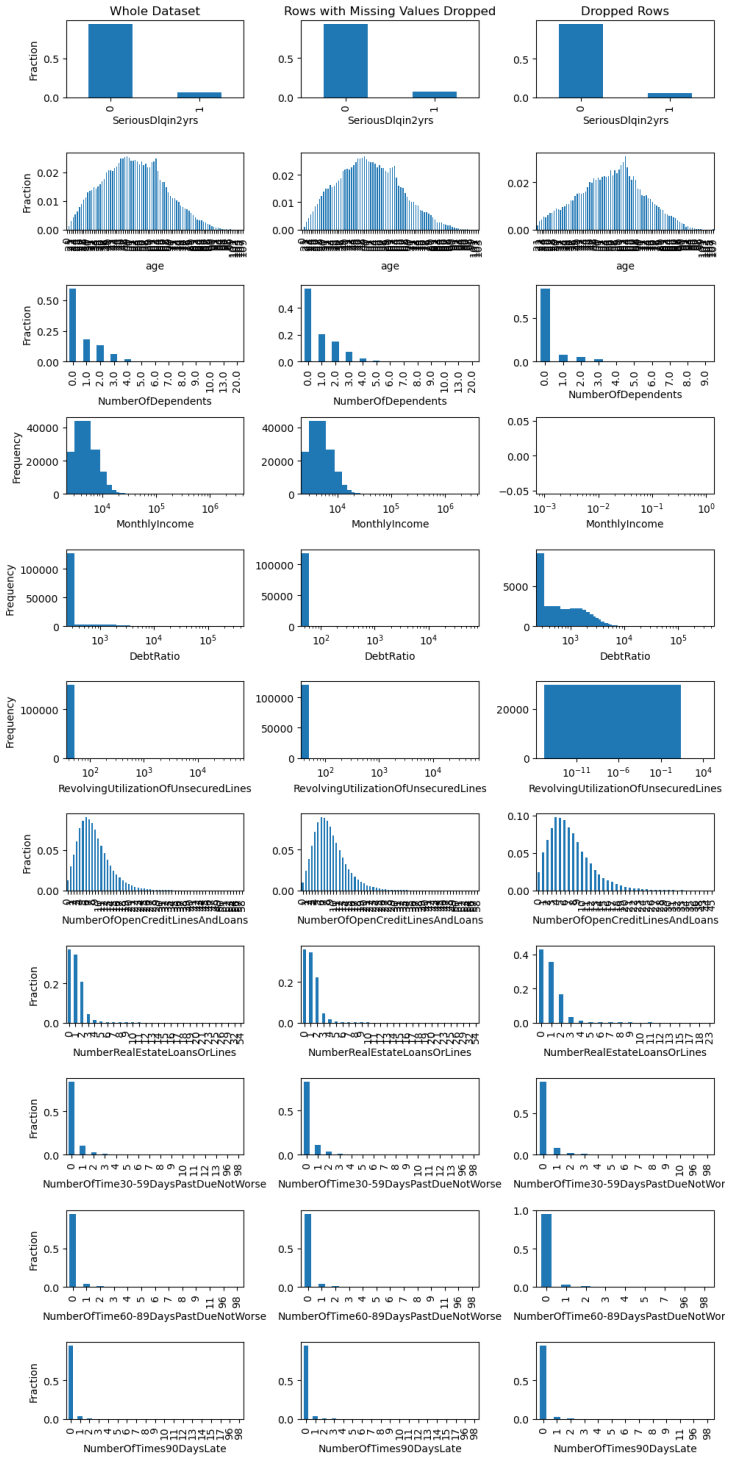
```
fig.tight_layout()
plt.show()
```



Where we can again see the high `DebtRatio` values for the rows with missing values. We can also have a look at the distribution of all the variables in the dataset

```
fig, ax = plt.subplots(df.shape[1], 3, figsize=(10, 20))

for ii, col in enumerate(df.columns):

    # Plot the distribution of the variable for the whole
     ↪  dataset, the dataset with missing values dropped, and the
     ↪  dropped rows
    if col in ('SeriousDlqin2yrs', 'age',
     ↪  'NumberOfTime30-59DaysPastDueNotWorse',
     ↪  'NumberOfOpenCreditLinesAndLoans',
     ↪  'NumberOfTimes90DaysLate',
     ↪  'NumberRealEstateLoansOrLines',
     ↪  'NumberOfTime60-89DaysPastDueNotWorse',
     ↪  'NumberOfDependents'):

        # Use a bar plot for discrete variables

↪   df[col].value_counts(normalize=True).sort_index().plot.bar(ax=ax[ii,0])

↪   df.dropna()[col].value_counts(normalize=True).sort_index().plot.bar(ax=ax[ii,1])
        df.loc[df.isna().any(axis=1),
↪   col].value_counts(normalize=True).sort_index().plot.bar(ax=ax[ii,2])

        # Set the y-axis label
        ax[ii,0].set_ylabel('Fraction')
        ax[ii,1].set_ylabel('')
        ax[ii,2].set_ylabel('')
```

```
    else:

        # Use a histogram for continuous variables
        df[col].plot.hist(bins=1000, ax=ax[ii,0])
        df.dropna()[col].plot.hist(bins=1000, ax=ax[ii,1])
        df.loc[df.isna().any(axis=1), col].plot.hist(bins=1000,
↪   ax=ax[ii,2])

        # Set the x-axis to a logarithmic scale for the
         ↪  continuous variables
        ax[ii,0].set_xscale('log')
        ax[ii,1].set_xscale('log')
        ax[ii,2].set_xscale('log')

        # Set the x-axis label
        ax[ii,0].set_xlabel(col)
        ax[ii,1].set_xlabel(col)
        ax[ii,2].set_xlabel(col)

        # Set the y-axis label
        ax[ii,0].set_ylabel('Frequency')
        ax[ii,1].set_ylabel('')
        ax[ii,2].set_ylabel('')

ax[0,0].set_title('Whole Dataset')
ax[0,1].set_title('Rows with Missing Values Dropped')
ax[0,2].set_title('Dropped Rows')

fig.tight_layout()
plt.show()
```


Whole Dataset
Rows with Missing Values Dropped
Dropped Rows
SeriousDlqin2yrs
age
NumberOfDependents
MonthlyIncome
DebtRatio
RevolvingUtilizationOfUnsecuredLines
NumberOfOpenCreditLinesAndLoans
NumberRealEstateLoansOrLines
NumberOfTime30-59DaysPastDueNotWorse
NumberOfTime60-89DaysPastDueNotWorse
NumberOfTimes90DaysLate
Fraction
Frequency

This shows another potential issue with our dataset. Checkout the variable `NumberOfTime30-59DaysPastDueNotWorse`. It seems that there are a some observations with values greater than 90. This seems rather odd. Let's have a look at the data dictionary

```
data_dict.loc[data_dict['Variable
 ↪  Name'].isin(['NumberOfTime30-59DaysPastDueNotWorse',
 ↪  'NumberOfTime60-89DaysPastDueNotWorse',
 ↪  'NumberOfTimes90DaysLate'])].style.hide()
```

Table 6.9

Variable Name	Description
NumberOfTime30-59DaysPastDueNotWorse	Number of times borrower has been 30-59 days past due but no wors
NumberOfTimes90DaysLate	Number of times borrower has been 90 days or more past due.
NumberOfTime60-89DaysPastDueNotWorse	Number of times borrower has been 60-89 days past due but no wors

The data dictionary does not mention anything about values above 90. These values may have a special meaning such as being a flag for missing values. Let's have a look at the distribution of the target variable for the rows with values greater than 90

```
df.loc[df['NumberOfTime30-59DaysPastDueNotWorse'] > 90,
 ↪  'SeriousDlqin2yrs'].value_counts()
```

```
SeriousDlqin2yrs
1    147
0    122
Name: count, dtype: int64
```

```
df.loc[df['NumberOfTime60-89DaysPastDueNotWorse'] > 90,
 ↪  'SeriousDlqin2yrs'].value_counts()
```

```
SeriousDlqin2yrs
1    147
0    122
Name: count, dtype: int64
```

```
df.loc[df['NumberOfTimes90DaysLate'] > 90,
 ↪  'SeriousDlqin2yrs'].value_counts()
```

```
SeriousDlqin2yrs
1    147
0    122
Name: count, dtype: int64
```

```
df.loc[(df['NumberOfTimes90DaysLate'] > 90) &
 ↪  (df['NumberOfTime60-89DaysPastDueNotWorse'] > 90) &
 ↪  (df['NumberOfTime30-59DaysPastDueNotWorse'] > 90),
 ↪  'SeriousDlqin2yrs'].value_counts()
```

```
SeriousDlqin2yrs
1    147
0    122
Name: count, dtype: int64
```

There seems to be a very high number of defaults for these observations (more than half), which makes sense given the meaning of these variables. Furthermore, the observations with above 90 in one category have it above 90 in the other categories as well. Thus, this might not be a data entry/collection error and these are just borrowers who commonly fail to make loan payments.

Given that Alonso Robisco and Carbó Martínez (2022) seem to be dropping the missing values, let's do the same for our dataset

```
df = df.dropna()
```

and let's also drop the rows with `MonthlyIncome` less than (or equal) 100

```
df = df.query('MonthlyIncome > 100')
```

to eliminate some of the potential data entry/collection errors.

Then double-check that we have no missing values left

```
df.isna().sum()
```

```
SeriousDlqin2yrs                        0
age                                     0
NumberOfDependents                      0
MonthlyIncome                           0
DebtRatio                               0
RevolvingUtilizationOfUnsecuredLines    0
NumberOfOpenCreditLinesAndLoans         0
NumberRealEstateLoansOrLines            0
NumberOfTime30-59DaysPastDueNotWorse    0
NumberOfTime60-89DaysPastDueNotWorse    0
NumberOfTimes90DaysLate                 0
dtype: int64
```

or, alternatively,

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
Index: 117968 entries, 0 to 149999
Data columns (total 11 columns):
 #   Column                                Non-Null Count   Dtype
---  ------                                --------------   -----
 0   SeriousDlqin2yrs                      117968 non-null  int64
 1   age                                   117968 non-null  int64
 2   NumberOfDependents                    117968 non-null  float64
 3   MonthlyIncome                         117968 non-null  float64
 4   DebtRatio                             117968 non-null  float64
 5   RevolvingUtilizationOfUnsecuredLines  117968 non-null  float64
 6   NumberOfOpenCreditLinesAndLoans       117968 non-null  int64
 7   NumberRealEstateLoansOrLines          117968 non-null  int64
 8   NumberOfTime30-59DaysPastDueNotWorse  117968 non-null  int64
 9   NumberOfTime60-89DaysPastDueNotWorse  117968 non-null  int64
 10  NumberOfTimes90DaysLate               117968 non-null  int64
dtypes: float64(4), int64(7)
memory usage: 10.8 MB
```

All good! We should also check for duplicated rows with the `duplicated()` method

```
df.loc[df.duplicated()]
```

	SeriousDlqin2yrs	age	NumberOfDependents	MonthlyIncome	DebtRatio	RevolvingUtilizationOfU
7920	0	22	0.0	820.0	0.0	1.0
8840	0	23	0.0	820.0	0.0	1.0
15546	0	22	0.0	929.0	0.0	0.0
17265	0	22	0.0	820.0	0.0	1.0
21190	0	22	0.0	820.0	0.0	1.0
...	...	...	...	...	...	...
143750	0	23	0.0	820.0	0.0	1.0
144153	0	28	0.0	2200.0	0.0	1.0
144922	0	40	0.0	3500.0	0.0	0.0
148419	0	22	0.0	1500.0	0.0	0.0
149993	0	22	0.0	820.0	0.0	1.0

and look at the statistics of the duplicated rows

```
df.loc[df.duplicated()].describe().T
```

	count	mean	std	min	25%	50%	75%
SeriousDlqin2yrs	72.0	0.013889	0.117851	0.0	0.0	0.0	0.0
age	72.0	24.902778	8.868618	21.0	22.0	22.5	24.0
NumberOfDependents	72.0	0.000000	0.000000	0.0	0.0	0.0	0.0

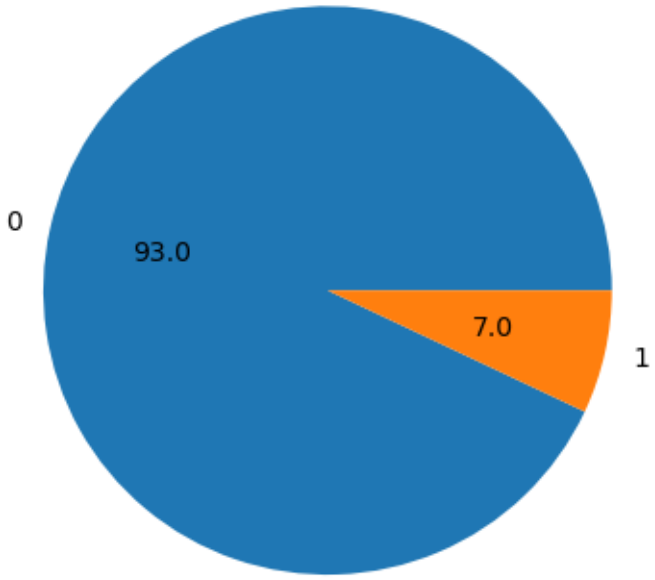
	count	mean	std	min	25%	50%
MonthlyIncome	72.0	1031.527778	542.007873	764.0	820.0	820.
DebtRatio	72.0	0.017594	0.104816	0.0	0.0	0.0
RevolvingUtilizationOfUnsecuredLines	72.0	0.500000	0.503509	0.0	0.0	0.5
NumberOfOpenCreditLinesAndLoans	72.0	1.458333	0.749413	0.0	1.0	1.0
NumberRealEstateLoansOrLines	72.0	0.000000	0.000000	0.0	0.0	0.0
NumberOfTime30-59DaysPastDueNotWorse	72.0	0.000000	0.000000	0.0	0.0	0.0
NumberOfTime60-89DaysPastDueNotWorse	72.0	0.000000	0.000000	0.0	0.0	0.0
NumberOfTimes90DaysLate	72.0	0.013889	0.117851	0.0	0.0	0.0

There are indeed 72 duplicated rows in the dataset. However, given the variables in our dataset, which are mostly discrete, the fact that monthly income seems to be generally rounded, it does not seem implausible that some rows might appear multiple times in the dataset, simply because some observations have the same values for all variables. Thus, we will keep the duplicated rows in the dataset.

6.6 Data Exploration

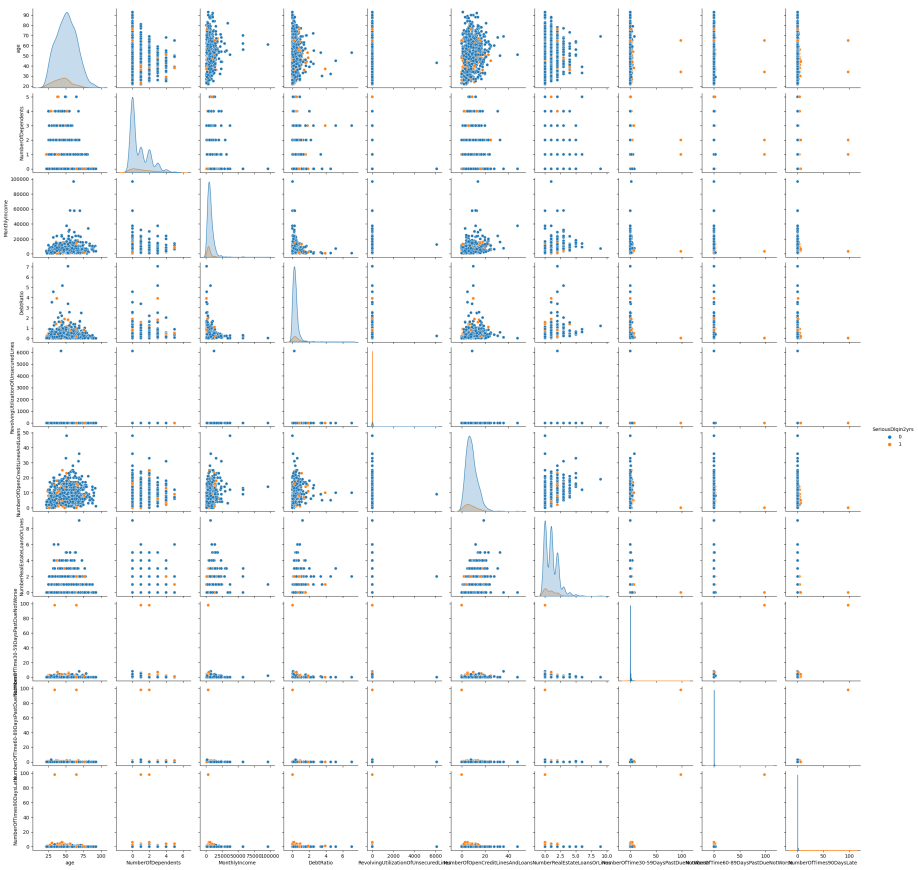
Let's start by looking at the distribution of the target variable `SeriousDlqin2yrs` in our preprocessed dataset

```
df.value_counts("SeriousDlqin2yrs").plot.pie(autopct = "%.1f")
plt.ylabel('')
plt.show()
```



We have already looked at some variables selectively. To do it more broadly, we can look at the pair plot of the dataset. A pair plot shows the pairwise relationships between the variables in our dataset. On the diagonal, we are plotting the kernel density estimate

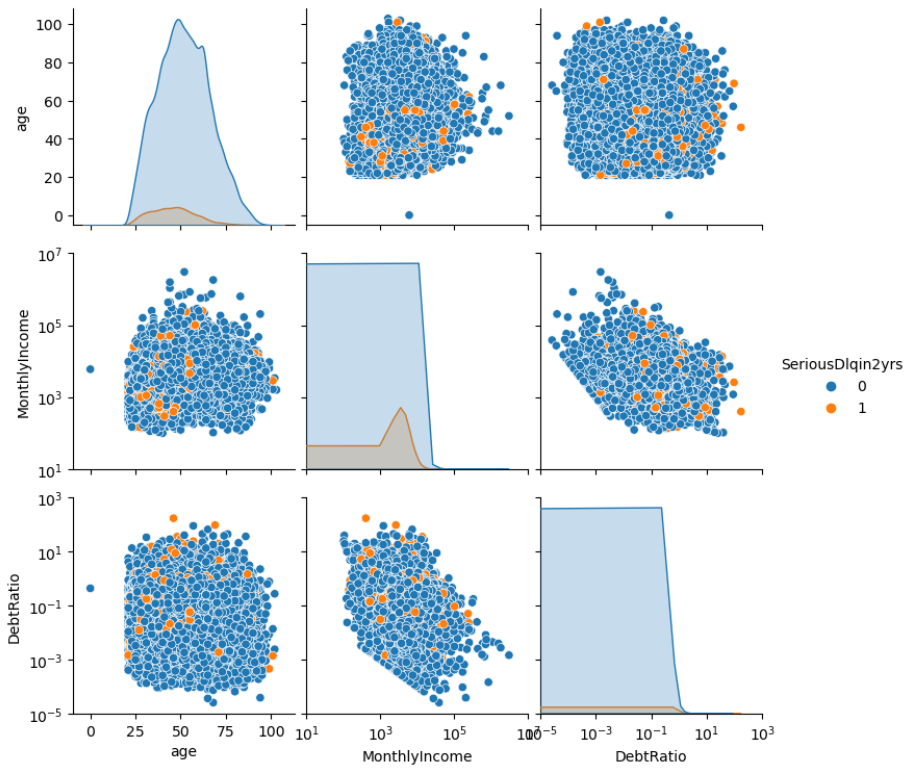
```
sns.pairplot(df.sample(1000), hue='SeriousDlqin2yrs')
```



Note that we are plotting all variables in different colors based on whether our target variable `SeriousDlqin2yrs` is 0 or 1. Furthermore, since it is computationally quite demanding to create this plot, we have sampled only 1000 rows from the dataset. Since we have many variables, some of them with very skewed distributions, and also several discrete variables, it might make sense to look only at a subset

```
pp = sns.pairplot(df[['age', 'MonthlyIncome', 'DebtRatio',
 ↪  'SeriousDlqin2yrs']], hue='SeriousDlqin2yrs')

# Fix the x-axis and y-axis scales
for ax in pp.axes.flat:
    if ax.get_xlabel() == 'MonthlyIncome':
        ax.set(xscale="log")
        ax.set_xlim(10**1, 10**7)
    if ax.get_ylabel() == 'MonthlyIncome':
        ax.set(yscale="log")
        ax.set_ylim(10**1, 10**7)
```

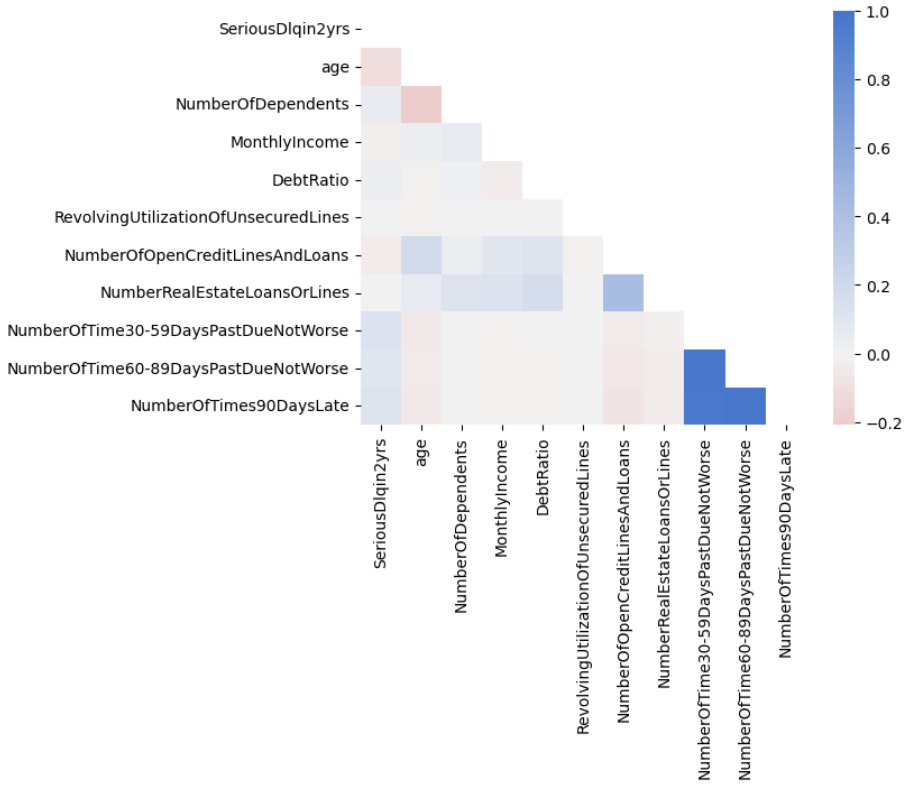
```
    if ax.get_xlabel() == 'DebtRatio':
        ax.set(xscale="log")
        ax.set_xlim(10**(-5), 10**3)
    if ax.get_ylabel() == 'DebtRatio':
        ax.set(yscale="log")
        ax.set_ylim(10**(-5), 10**3)
```



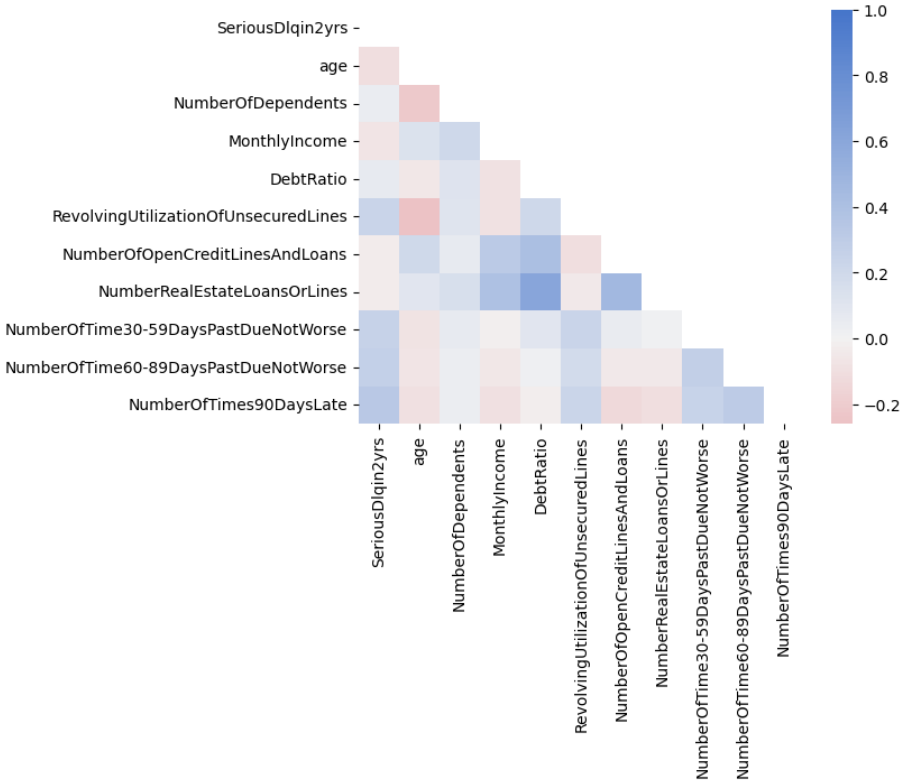
We can continue with the analysis of our dataset by looking at the correlation matrix of the variables in the dataset. We will calculate both the Pearson correlation (linear relationship) and the Spearman correlation (monotonic relationship) and create a heatmap of both correlation matrices

```
corr = df.corr() # Calculate the Pearson correlation (linear
 ↪  relationship)
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()
```

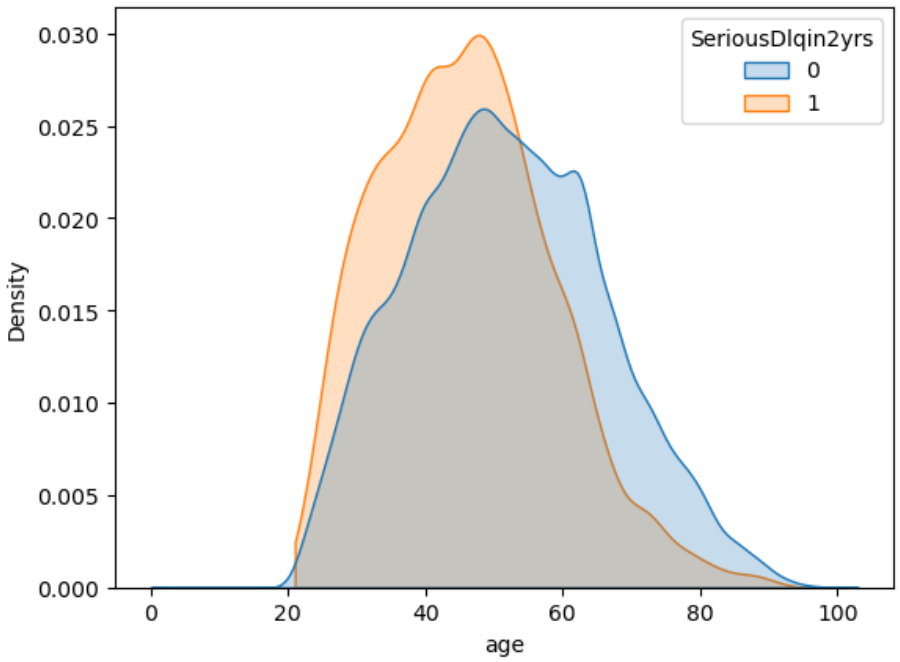



```
corr = df.corr('spearman') # Calculate the Spearman correlation
 ↪  (monotonic relationship)
cmap = sns.diverging_palette(10, 255, as_cmap=True) # Create a
 ↪  color map
mask = np.triu(np.ones_like(corr, dtype=bool)) # Create a mask to
 ↪  only show the lower triangle of the matrix
sns.heatmap(corr, 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()
```



It seems that `age` is negatively correlated with default (`SeriousDlqin2yrs`) which we can also see in the kernel density estimate of the `age` variable

```
sns.kdeplot(data=df, x='age', hue='SeriousDlqin2yrs', cut=0,
 ↪  fill=True, common_norm=False)
plt.show()
```



but then `MonthlyIncome` is also negatively correlated with default and with `age`. Thus, likely the relationship between `age` and default is driven by `MonthlyIncome`.

Furthermore, the variables `NumberOfTime30-59DaysPastDueNotWorse`, `NumberOfTime60-89DaysPastDueNotWorse`, and `NumberOfTimes90DaysLate` are highly correlated with each other and with the target variable `SeriousDlqin2yrs`. This is not surprising given that these variables are all related to the number of times a borrower has been past due on a loan payment. `RevolvingUtilizationOfUnsecuredLines` is also highly correlated with the target variable and with the number of times a borrower has been past due on a loan payment. This is also not surprising given that the `RevolvingUtilizationOfUnsecuredLines` is the ratio of the amount of money owed to the amount of credit available.

6.7 Implementation of Loan Default Prediction Models

We have explored our dataset and are now ready to implement machine learning algorithms for loan default prediction. Let's start by importing the required libraries

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import confusion_matrix, accuracy_score,
 ↪  roc_auc_score, recall_score, precision_score, roc_curve
from joblib import dump, load
```

6.7.1 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('SeriousDlqin2yrs', axis=1) # All variables except
 ↪  `SeriousDlqin2yrs`
y = df['SeriousDlqin2yrs'] # Only SeriousDlqin2yrs
```

We follow Alonso Robisco and Carbó Martínez (2022) and use 80% of the data for training and 20% 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 defaulted loans 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 = 0.2, random_state = 42)
```

6.7.2 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. We will use the `MinMaxScaler` class from the `sklearn.preprocessing` module to scale the features. The `MinMaxScaler` class scales the features so that they have a minimum of 0 and a maximum of 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

scaler = MinMaxScaler()
scale_features(scaler, X_train, X_train.columns)
scale_features(scaler, X_test, X_test.columns,
 ↪  only_transform=True)
```

Note that we have very skewed distributions for some variables in our dataset. This might make the `MinMaxScaler` less effective and there might be gains from more carefully scaling different variables. However, for the sake of simplicity, we will use the `MinMaxScaler` for all variables.

We have fully preprocessed and explored our dataset. The next step will be our main task: the implementation of machine learning algorithms for loan default prediction.

6.7.3 Evaluation Criertia

We will evaluate the performance of our machine-learning models using the following metrics:

- **Accuracy**: The proportion of correctly classified instances
- **Precision**: The proportion of true positive predictions among all positive predictions
- **Recall**: The proportion of true positive predictions among all actual positive instances
- **ROC AUC**: The area under the receiver operating characteristic curve

Furthermore, we will plot the ROC curve for each model to visualize the trade-off between the true positive rate and the false positive rate. To make the evaluation of our models more convenient, we will define a function that computes these metrics and plots the ROC curve for a given model

```
def evaluate_model(clf, X_train, y_train, X_test, y_test,
 ↪  label=''):

    # Compute predictions and probabilities for tha training and
     ↪  test set
    y_pred_train = clf.predict(X_train)
    y_proba_train = clf.predict_proba(X_train)
    y_pred_test = clf.predict(X_test)
    y_proba_test = clf.predict_proba(X_test)
```

```
    # Print accuracy measures

    print(f"------------------------------------------------------------------")
    print(f"Metrics: {label}")

    print(f"------------------------------------------------------------------")
    print(f"Accuracy (Train): {accuracy_score(y_train,
    y_pred_train)}")
    print(f"Precision (Train): {precision_score(y_train,
    y_pred_train)}")
    print(f"Recall (Train): {recall_score(y_train,
    y_pred_train)}")
    print(f"ROC AUC (Train): {roc_auc_score(y_train,
    y_proba_train[:, 1])}")

    print(f"------------------------------------------------------------------")
    print(f"Accuracy (Test): {accuracy_score(y_test,
    y_pred_test)}")
    print(f"Precision (Test): {precision_score(y_test,
    y_pred_test)}")
    print(f"Recall (Test): {recall_score(y_test, y_pred_test)}")
    print(f"ROC AUC (Test): {roc_auc_score(y_test,
    y_proba_test[:, 1])}")

    print(f"------------------------------------------------------------------")

    # Compute the ROC curve
    fpr_train, tpr_train, thresholds_train = roc_curve(y_train,
    y_proba_train[:, 1])
    fpr_test, tpr_test, thresholds_test = roc_curve(y_test,
    y_proba_test[:, 1])

    # Plot the ROC curve
    plt.plot(fpr_train, tpr_train, label = "Train")
    plt.plot(fpr_test, tpr_test, label = "Test")
    plt.plot([0, 1], [0, 1], linestyle='--', color='grey')
    plt.xlabel('False Positive Rate (FPR)')
    plt.ylabel('True Positive Rate (TPR)')
    plt.title(f'ROC Curve: {label}')
    plt.legend()
    plt.show()
```

While we compute all of these metrics, we will focus on the ROC AUC score as our main evaluation metric.

6.7.4 Logistic Regression

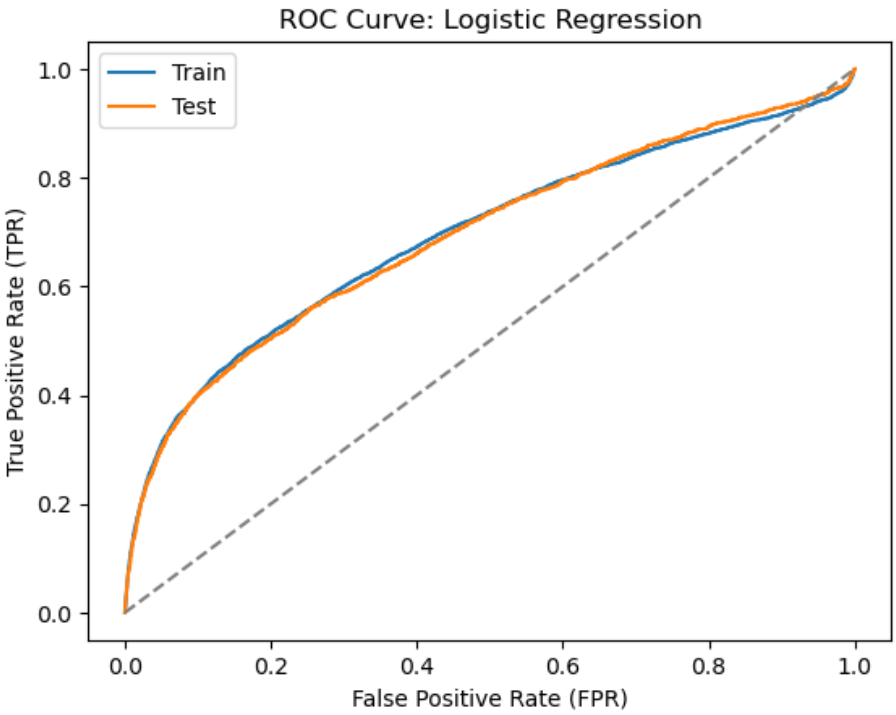
Let's start with a simple logistic regression model. We will use the `LogisticRegression` class from the `sklearn.linear_model` module to train a logistic regression model. We will use the `lbfgs` solver and set the `max_iter` parameter to 5000 to make sure that the optimization algorithm converges. We will also set the `penalty` parameter to `None` to avoid regularization.

```
clf_logistic = LogisticRegression(penalty = None, solver =
↪  'lbfgs', max_iter = 5000).fit(X_train, y_train)
```

Let's evaluate the performance of the logistic regression model

```
evaluate_model(clf_logistic, X_train, y_train, X_test, y_test,
↪  label = 'Logistic Regression')
```

```
------------------------------------------------------------
Metrics: Logistic Regression
------------------------------------------------------------
Accuracy (Train): 0.9306588679085341
Precision (Train): 0.5787234042553191
Recall (Train): 0.041100030220610456
ROC AUC (Train): 0.6936171984517471
------------------------------------------------------------
Accuracy (Test): 0.9302788844621513
Precision (Test): 0.5490196078431373
Recall (Test): 0.033836858006042296
ROC AUC (Test): 0.6926238627317243
------------------------------------------------------------
```



The model does not perform as well as what we have seen in previous lectures. The ROC AUC score is only around 0.7. Note again that the accuracy score is quite high but this is due to the imbalanced nature of the dataset.

6.7.5 Decision Tree

Let's now train a decision tree classifier. We will use the `DecisionTreeClassifier` class from the `sklearn.tree` module to train a decision tree classifier. We will set the `max_depth` parameter to 7 as in Alonso Robisco and Carbó Martínez (2022) to avoid overfitting.

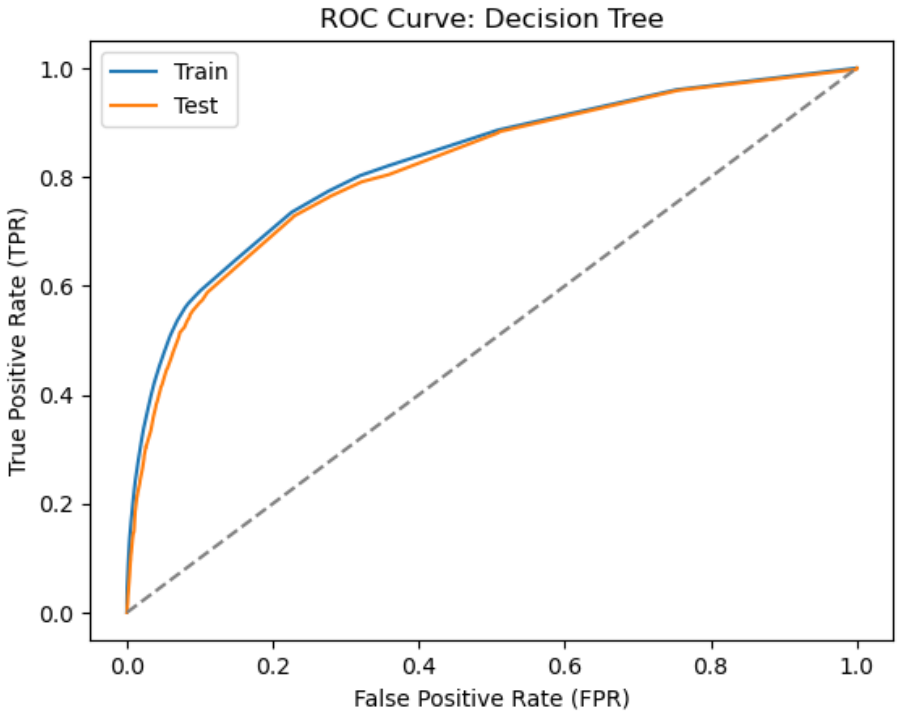
```
clf_tree = DecisionTreeClassifier(max_depth=7).fit(X_train,
↪  y_train)
```

Then, let's evaluate the performance of the decision tree classifier

```
evaluate_model(clf_tree, X_train, y_train, X_test, y_test, label
↪  = 'Decision Tree')
```

```
------------------------------------------------------------
Metrics: Decision Tree
------------------------------------------------------------
Accuracy (Train): 0.936348994426431
```

```
Precision (Train): 0.6407185628742516
Recall (Train): 0.2101843457237836
ROC AUC (Train): 0.8265896261153018
----------------------------------------------------------------
Accuracy (Test): 0.9323556836483852
Precision (Test): 0.5529622980251346
Recall (Test): 0.18610271903323264
ROC AUC (Test): 0.8159823399376106
----------------------------------------------------------------
```



The decision tree classifier performs better than the logistic regression model with a ROC AUC score of around 0.77. This is not surprising given that decision trees are more flexible models that can capture non-linear relationships in the data.

6.7.6 Random Forest

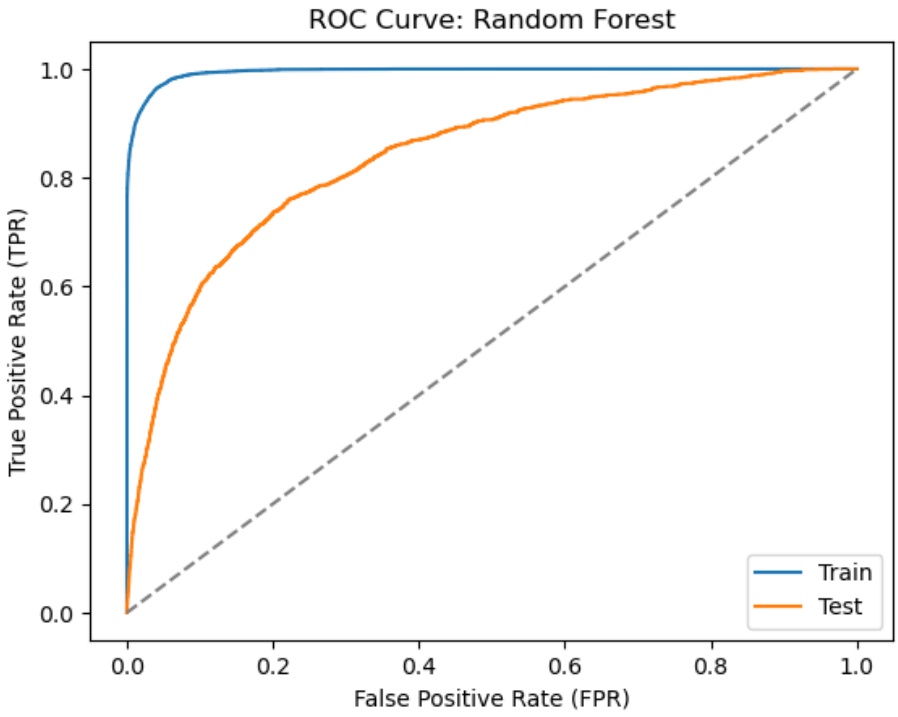
Let's now train a random forest classifier. We will use the `RandomForestClassifier` class from the `sklearn.ensemble` module to train a random forest classifier. We will set the `max_depth` parameter to 20 and the `n_estimators` parameter to 100 as in Alonso Robisco and Carbó Martínez (2022).

```
clf_forest = RandomForestClassifier(max_depth=20, n_estimators =
↪  100).fit(X_train, y_train)
```

Then, let's evaluate the performance of the random forest classifier

```
evaluate_model(clf_forest, X_train, y_train, X_test, y_test,
↪  label = 'Random Forest')
```

```
------------------------------------------------------------
Metrics: Random Forest
------------------------------------------------------------
Accuracy (Train): 0.9755441117256871
Precision (Train): 1.0
Recall (Train): 0.6512541553339377
ROC AUC (Train): 0.9948736572824639
------------------------------------------------------------
Accuracy (Test): 0.9314232431974231
Precision (Test): 0.5333333333333333
Recall (Test): 0.17885196374622356
ROC AUC (Test): 0.8402779803214324
------------------------------------------------------------
```



This is a good example of the dangers of not using a test set for the evaluation of a model. The random forest classifier performs very well on the training set with a ROC AUC score of close to 1.0. However, it performs much worse on the test set with a ROC AUC score of around 0.83. Nevertheless, the random forest classifier still outperforms the logistic regression and decision tree classifiers.

6.7.7 XGBoost

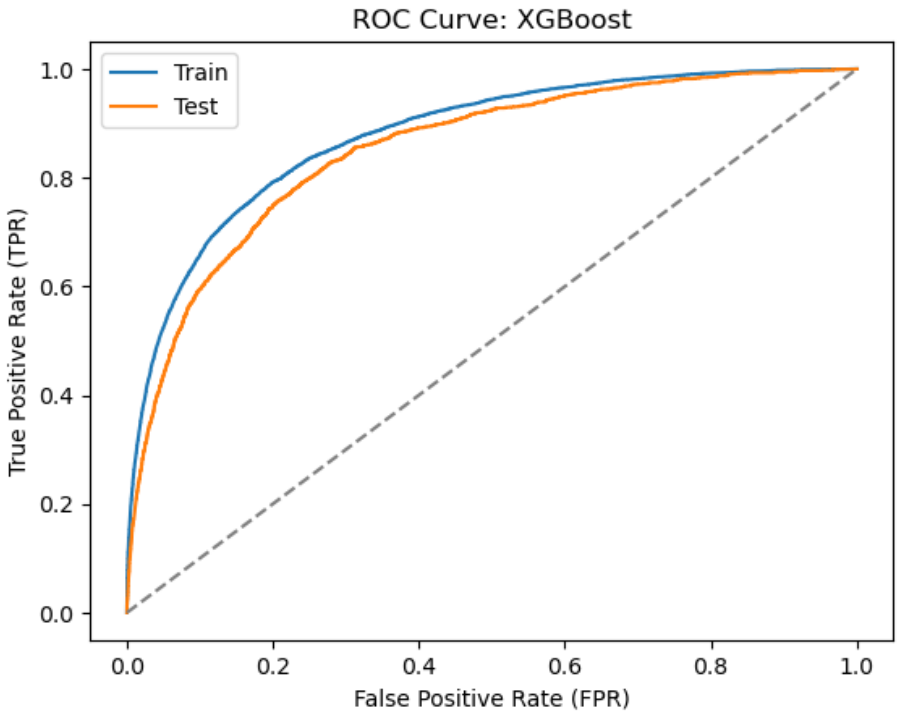
Let's now train an XGBoost classifier. We will use the `XGBClassifier` class from the `xgboost` module to train an XGBoost classifier. We will set the `max_depth` parameter to 5 and the `n_estimators` parameter to 40 as in Alonso Robisco and Carbó Martínez (2022).

```
clf_xgb = XGBClassifier(max_depth = 5, n_estimators = 40,
↪  random_state = 0).fit(X_train, y_train)
```

Then, let's evaluate the performance of the XGBoost classifier

```
evaluate_model(clf_xgb, X_train, y_train, X_test, y_test, label =
↪  'XGBoost')
```

```
------------------------------------------------------------
Metrics: XGBoost
------------------------------------------------------------
Accuracy (Train): 0.9389238561468201
Precision (Train): 0.6947992700729927
Recall (Train): 0.23012994862496222
ROC AUC (Train): 0.8778826700467908
------------------------------------------------------------
Accuracy (Test): 0.9330762058150377
Precision (Test): 0.5701107011070111
Recall (Test): 0.18670694864048337
ROC AUC (Test): 0.8514290998289821
------------------------------------------------------------
```



The XGBoost classifier performs quite well with an ROC AUC score of around 0.83. This is the best performance we have seen so far.

6.7.8 Neural Network

Finally, let's train a neural network classifier. We will use the `MLPClassifier` class from the `sklearn.neural_network` module to train a neural network classifier. We will set the `activation` parameter to `relu`, the `solver` parameter to `adam`, and the `hidden_layer_sizes` parameter to `(300,200,100)` as in Alonso Robisco and Carbó Martínez (2022). We will also set the `random_state` parameter to 42 to make the results reproducible.

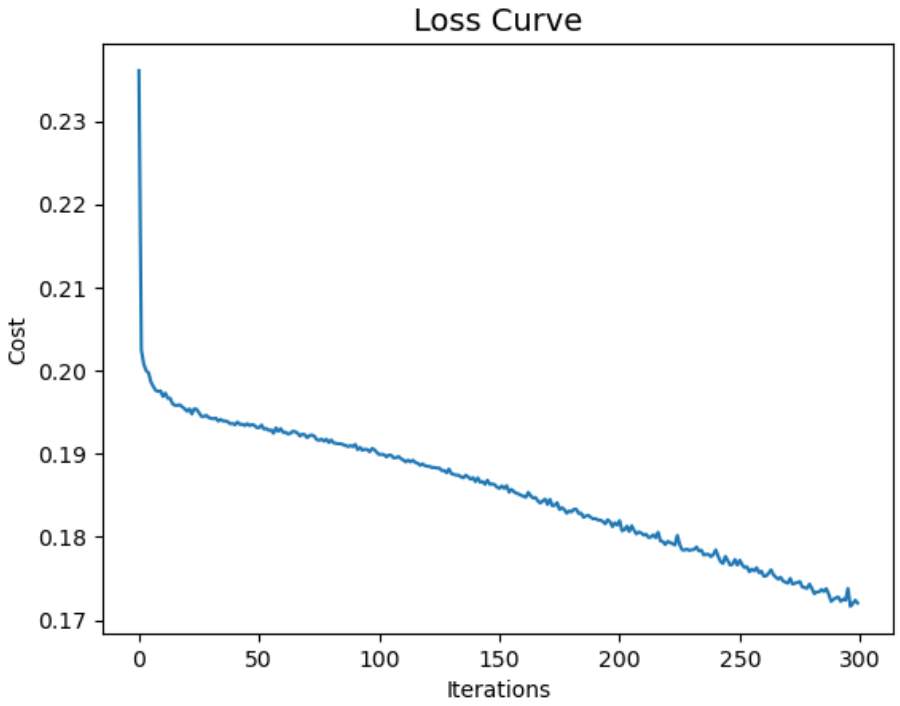
```
#clf_nn = MLPClassifier(activation='relu', solver='adam',
 ↪  hidden_layer_sizes=(300,200,100), random_state=42, max_iter =
 ↪  300, verbose=True).fit(X_train, y_train)
#dump(clf_nn, 'clf_nn.joblib')
```

Since training the neural network classifier can take a long time, we have saved the trained model to a file called `clf_nn.joblib`. We can load the model from the file using the `load` function from the `joblib` module

```
clf_nn = load('clf_nn.joblib')
```

Let's check the loss curve of the neural network classifier

```
plt.plot(clf_nn.loss_curve_)
plt.title("Loss Curve", fontsize=14)
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.show()
```

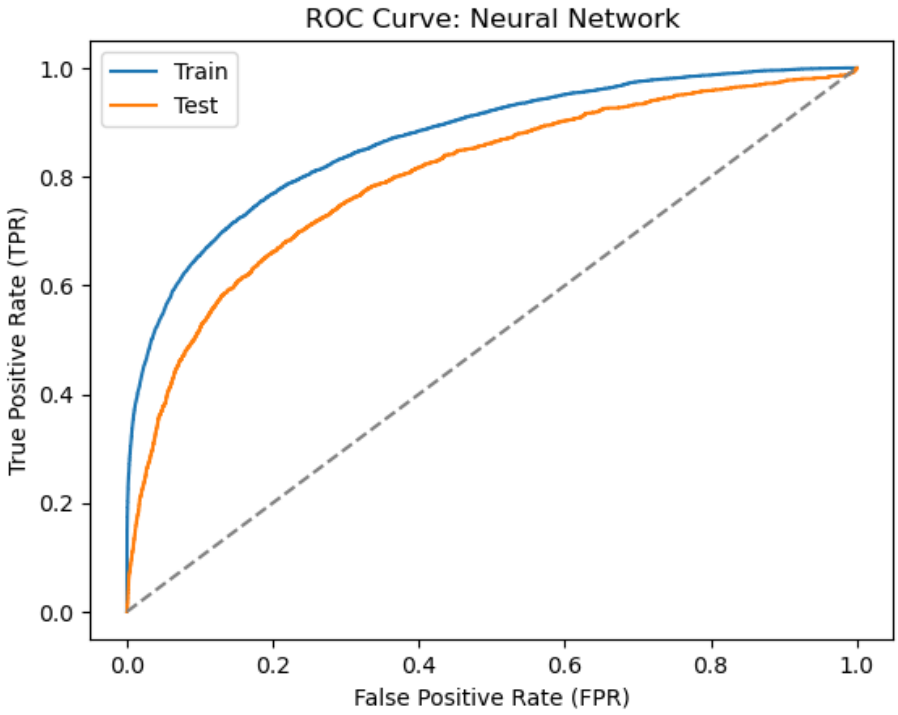


Then, let's evaluate the performance of the neural network classifier

```
evaluate_model(clf_nn, X_train, y_train, X_test, y_test, label =
↪  'Neural Network')
```

```
------------------------------------------------------------
Metrics: Neural Network
------------------------------------------------------------
Accuracy (Train): 0.9465742683366182
Precision (Train): 0.8080531665363565
Recall (Train): 0.31233000906618313
ROC AUC (Train): 0.8682599095370773
------------------------------------------------------------
Accuracy (Test): 0.9282020852759176
Precision (Test): 0.4694835680751174
```

```
Recall (Test): 0.18126888217522658
ROC AUC (Test): 0.7983637135044449
----------------------------------------------------------------
```



6.8 Overview of the Results

Looking at all the models side by side, we can see that

```
results = pd.DataFrame({
    'Model': ['Logistic Regression', 'Decision Tree', 'Random
    ↪ Forest', 'XGBoost', 'Neural Network'],
    'ROC AUC (Train)': [roc_auc_score(y_train,
    ↪ clf_logistic.predict_proba(X_train)[:, 1]),
                        roc_auc_score(y_train,
↪ clf_tree.predict_proba(X_train)[:, 1]),
                        roc_auc_score(y_train,
↪ clf_forest.predict_proba(X_train)[:, 1]),
                        roc_auc_score(y_train,
↪ clf_xgb.predict_proba(X_train)[:, 1]),
                        roc_auc_score(y_train,
↪ clf_nn.predict_proba(X_train)[:, 1])],
    'ROC AUC (Test)': [roc_auc_score(y_test,
    ↪ clf_logistic.predict_proba(X_test)[:, 1]),
```

```
                        roc_auc_score(y_test,
 ↪  clf_tree.predict_proba(X_test)[:, 1]),
                        roc_auc_score(y_test,
 ↪  clf_forest.predict_proba(X_test)[:, 1]),
                        roc_auc_score(y_test,
 ↪  clf_xgb.predict_proba(X_test)[:, 1]),
                        roc_auc_score(y_test,
 ↪  clf_nn.predict_proba(X_test)[:, 1])]
})
results
```

	Model	ROC AUC (Train)	ROC AUC (Test)
0	Logistic Regression	0.693617	0.692624
1	Decision Tree	0.826590	0.815982
2	Random Forest	0.994874	0.840278
3	XGBoost	0.877883	0.851429
4	Neural Network	0.868260	0.798364

But can we do better? Alonso Robisco and Carbó Martínez (2022) have also applied feature engineering to the dataset. Let's see if we can improve the performance of our models by adding some additional features.

6.9 Feature Engineering and Model Improvement

We will add the square of each feature to the dataset to create additional features as in Alonso Robisco and Carbó Martínez (2022). We will use the `assign` method of the `pandas` DataFrame to add the squared features to the dataset

```
X2 = df.drop('SeriousDlqin2yrs', axis=1) # All variables except
 ↪  `SeriousDlqin2yrs`
y2 = df['SeriousDlqin2yrs'] # Only SeriousDlqin2yrs
X2 = X2.assign(**X2.pow(2).add_suffix('_sq')) # Add the squared
 ↪  features to the dataset
```

Then, we will split the dataset into a training set and a test set and scale the features

```
X2_train, X2_test, y2_train, y2_test = train_test_split(X2, y2,
 ↪  stratify=y2, test_size = 0.2, random_state = 42)
scaler = MinMaxScaler()
scale_features(scaler, X2_train, X2_train.columns)
scale_features(scaler, X2_test, X2_test.columns,
 ↪  only_transform=True)
```

Let's train the models again with the new dataset and evaluate their performance

```
clf_logistic2 = LogisticRegression(penalty = None, solver =
 ↪  'lbfgs', max_iter = 5000).fit(X2_train, y2_train)
clf_tree2 = DecisionTreeClassifier(max_depth=7).fit(X2_train,
 ↪  y2_train)
clf_forest2 = RandomForestClassifier(max_depth=20, n_estimators =
 ↪  100).fit(X2_train, y2_train)
clf_xgb2 = XGBClassifier(max_depth = 5, n_estimators = 40,
 ↪  random_state = 0).fit(X2_train, y2_train)
#clf_nn2 = MLPClassifier(activation='relu', solver='adam',
 ↪  hidden_layer_sizes=(300,200,100), random_state=42, max_iter =
 ↪  300, verbose=True).fit(X2_train, y2_train)
#dump(clf_nn2, 'clf_nn2.joblib')
```

The neural network classifier takes a long time to train, so we will load the model from the file `clf_nn2.joblib` that we saved earlier

```
clf_nn2 = load('clf_nn2.joblib')
```

Furthermore, we will also add a LASSO penalty to the logistic regression model to see if we can improve its performance

```
clf_logistic_lasso2 = LogisticRegression(penalty = 'l1', solver =
 ↪  'liblinear').fit(X2_train, y2_train)
```

Let's evaluate the performance of the models with the new features in the dataset

```
results2 = pd.DataFrame({
    'Model': ['Logistic Regression', 'Decision Tree', 'Random
     ↪  Forest', 'XGBoost', 'Neural Network', 'Logistic LASSO'],
    'ROC AUC (Train)': [roc_auc_score(y2_train,
     ↪  clf_logistic2.predict_proba(X2_train)[:, 1]),
                        roc_auc_score(y2_train,
 ↪  clf_tree2.predict_proba(X2_train)[:, 1]),
                        roc_auc_score(y2_train,
 ↪  clf_forest2.predict_proba(X2_train)[:, 1]),
                        roc_auc_score(y2_train,
 ↪  clf_xgb2.predict_proba(X2_train)[:, 1]),
                        roc_auc_score(y2_train,
 ↪  clf_nn2.predict_proba(X2_train)[:, 1]),
                        roc_auc_score(y2_train,
 ↪  clf_logistic_lasso2.predict_proba(X2_train)[:, 1])
    ],
    'ROC AUC (Test)': [roc_auc_score(y2_test,
     ↪  clf_logistic2.predict_proba(X2_test)[:, 1]),
```

```
                        roc_auc_score(y2_test,
 ↪  clf_tree2.predict_proba(X2_test)[:, 1]),
                        roc_auc_score(y2_test,
 ↪  clf_forest2.predict_proba(X2_test)[:, 1]),
                        roc_auc_score(y2_test,
 ↪  clf_xgb2.predict_proba(X2_test)[:, 1]),
                        roc_auc_score(y2_test,
 ↪  clf_nn2.predict_proba(X2_test)[:, 1]),
                        roc_auc_score(y2_test,
 ↪  clf_logistic_lasso2.predict_proba(X2_test)[:, 1])
    ]
})
results2
```

	Model	ROC AUC (Train)	ROC AUC (Test)
0	Logistic Regression	0.810256	0.813330
1	Decision Tree	0.826549	0.816039
2	Random Forest	0.993775	0.838637
3	XGBoost	0.877883	0.851429
4	Neural Network	0.871913	0.794420
5	Logistic LASSO	0.804400	0.810694

The models with the new features in the dataset perform better than the models without the new features. The random forest classifier and the XGBoost classifier have the best performance with ROC AUC scores of around 0.84 and 0.85, respectively.

6.10 Feature Importance

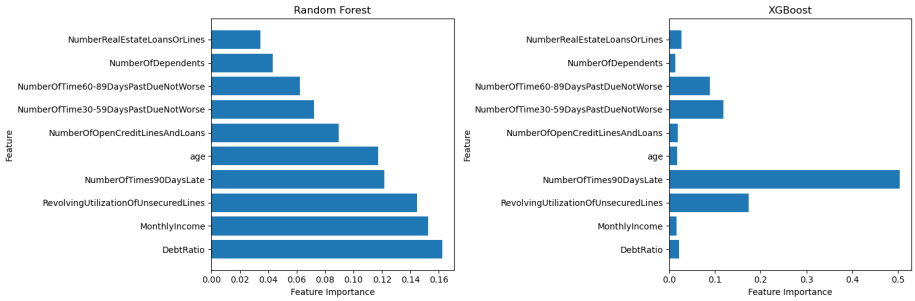
We can also look at the feature importance of the random forest classifier and the XGBoost classifier to see which features are most important for predicting loan defaults. We will use the `feature_importances_` attribute of the random forest classifier and the XGBoost classifier to get the feature importances

```
feature_importances_forest = clf_forest2.feature_importances_
feature_importances_xgb = clf_xgb2.feature_importances_
```

Then, we will create a bar plot of the feature importances for the random forest classifier and the XGBoost classifier

```
fig, ax = plt.subplots(1, 2, figsize=(15, 5))

df_feature_importance_forest = pd.DataFrame({'Feature':
 ↪  X_train.columns, 'Importance':
 ↪  clf_forest.feature_importances_})
```

```
df_feature_importance_forest =
 ↪  df_feature_importance_forest.sort_values('Importance',
 ↪  ascending=False)

df_feature_importance_xgb = pd.DataFrame({'Feature':
 ↪  X_train.columns, 'Importance': clf_xgb.feature_importances_})
df_feature_importance_xgb =
 ↪  df_feature_importance_xgb.set_index('Feature')
df_feature_importance_xgb =
 ↪  df_feature_importance_xgb.loc[df_feature_importance_forest['Feature'],
 ↪  ]

# Random Forest
ax[0].barh(df_feature_importance_forest['Feature'],
 ↪  df_feature_importance_forest['Importance'])
ax[0].set_title('Random Forest')
ax[0].set_xlabel('Feature Importance')
ax[0].set_ylabel('Feature')

# XGBoost
ax[1].barh(df_feature_importance_forest['Feature'],
 ↪  df_feature_importance_xgb['Importance'])
ax[1].set_title('XGBoost')
ax[1].set_xlabel('Feature Importance')
ax[1].set_ylabel('Feature')

fig.tight_layout()
plt.show()
```



6.11 Conclusions

We have successfully implemented machine learning algorithms for loan default prediction. We have explored the dataset, preprocessed the data, trained several machine learning models, and evaluated their performance. We have also

applied feature engineering to the dataset and improved the performance of the models. The random forest classifier and the XGBoost classifier have the best performance with ROC AUC scores of around 0.84 and 0.85, respectively. We have also looked at the feature importance of the random forest classifier and the XGBoost classifier to see which features are most important for predicting loan defaults.

Chapter 7

House Price Prediction

The focus of the previous examples in this course was on classification problems. However, regression problems are also quite common in practice and it will be what we will try to explore in this application.

7.1 Problem Setup

The dataset that we will be using is the Kaggle dataset called "House Sales in King County, USA". As far as I know, this was not used in a Kaggle competition. However, it is a quite popular dataset on Kaggle. The description reads:

> This dataset contains house sale prices for King County, which includes Seattle. It includes homes sold between May 2014 and May 2015.
>
> It's a great dataset for evaluating simple regression models.

This means that the dataset is a snapshot of house prices in King County, USA, between May 2014 and May 2015. The task, then, is quite straightforward: given a set of features, we want to predict the price of a house.

7.2 Dataset

Unfortunately, the dataset does not have a detailed description of the variables. However, in the comment section, some users found references with variable descriptions. The variables in the dataset should be as follows:

Variable	Description
id	Unique ID for each home sold
date	Date of the home sale

Variable	Description
price	Price of each home sold
bedrooms	Number of bedrooms
bathrooms	Number of bathrooms, where .5 accounts for a room with a toilet but no shower
sqft_living	Square footage of the apartments' interior living space
sqft_lot	Square footage of the land space
floors	Number of floors
waterfront	A dummy variable for whether the apartment was overlooking the waterfront or not
view	An index from 0 to 4 of how good the view of the property was
condition	An index from 1 to 5 on the condition of the apartment
grade	An index from 1 to 13, where 1-3 falls short of building construction and design, 7 has an average level of construction and design, and 11-13 have a high quality level of construction and design.
sqft_above	The square footage of the interior housing space that is above ground level
sqft_basement	The square footage of the interior housing space that is below ground level
yr_built	The year the house was initially built
yr_renovated	The year of the house's last renovation
zipcode	What zipcode area the house is in
lat	Lattitude
long	Longitude
sqft_living15	The square footage of interior housing living space for the nearest 15 neighbors
sqft_lot15	The square footage of the land lots of the nearest 15 neighbors

7.3 Putting the Problem into the Context of the Course

The problem of predicting house prices is a **regression problem** which belongs to the type of **supervised learning** problems. We will use the same tools that we have used in the previous examples to solve this problem. The main difference is that we will be using regression models instead of classification models.

7.4 Setting up the Environment

We will start by setting up the environment by importing the necessary libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

We can then load the dataset into a DataFrame

```
df = pd.read_csv('data/kc_house_data.csv')
```

7.5 Data Exploration

As with any new dataset, we first need to familiarize ourselves with the data. We will start by looking at the first few rows of the dataset.

```
df.head(4).T # Transpose the dataframe for readability
```

	0	1	2	3
id	7129300520	6414100192	5631500400	2487200875
date	20141013T000000	20141209T000000	20150225T000000	20141209T000000
price	221900.0	538000.0	180000.0	604000.0
bedrooms	3	3	2	4
bathrooms	1.0	2.25	1.0	3.0
sqft_living	1180	2570	770	1960
sqft_lot	5650	7242	10000	5000
floors	1.0	2.0	1.0	1.0
waterfront	0	0	0	0
view	0	0	0	0
condition	3	3	3	5
grade	7	7	6	7
sqft_above	1180	2170	770	1050
sqft_basement	0	400	0	910
yr_built	1955	1951	1933	1965
yr_renovated	0	1991	0	0
zipcode	98178	98125	98028	98136
lat	47.5112	47.721	47.7379	47.5208
long	-122.257	-122.319	-122.233	-122.393
sqft_living15	1340	1690	2720	1360
sqft_lot15	5650	7639	8062	5000

and for reference, we can also run `df.info()` again to see the data types of the variables

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             21613 non-null  int64
 1   date           21613 non-null  object
 2   price          21613 non-null  float64
 3   bedrooms       21613 non-null  int64
 4   bathrooms      21613 non-null  float64
 5   sqft_living    21613 non-null  int64
 6   sqft_lot       21613 non-null  int64
 7   floors         21613 non-null  float64
 8   waterfront     21613 non-null  int64
 9   view           21613 non-null  int64
 10  condition      21613 non-null  int64
 11  grade          21613 non-null  int64
 12  sqft_above     21613 non-null  int64
 13  sqft_basement  21613 non-null  int64
 14  yr_built       21613 non-null  int64
 15  yr_renovated   21613 non-null  int64
 16  zipcode        21613 non-null  int64
 17  lat            21613 non-null  float64
 18  long           21613 non-null  float64
 19  sqft_living15  21613 non-null  int64
 20  sqft_lot15     21613 non-null  int64
dtypes: float64(5), int64(15), object(1)
memory usage: 3.5+ MB
```

What immediately stands out is that the `date` column does not seem to be a proper datetime object. So, let's fix that

```
df['date'] = pd.to_datetime(df['date'])
```

```
df.head().T
```

	0	1	2	3
id	7129300520	6414100192	5631500400	2487200875
date	2014-10-13 00:00:00	2014-12-09 00:00:00	2015-02-25 00:00:00	2014-12-09 00:00:00
price	221900.0	538000.0	180000.0	604000.0
bedrooms	3	3	2	4
bathrooms	1.0	2.25	1.0	3.0
sqft_living	1180	2570	770	1960

	0	1	2	3	4
sqft_lot	5650	7242	10000	5000	8080
floors	1.0	2.0	1.0	1.0	1.0
waterfront	0	0	0	0	0
view	0	0	0	0	0
condition	3	3	3	5	3
grade	7	7	6	7	8
sqft_above	1180	2170	770	1050	1680
sqft_basement	0	400	0	910	0
yr_built	1955	1951	1933	1965	1987
yr_renovated	0	1991	0	0	0
zipcode	98178	98125	98028	98136	98074
lat	47.5112	47.721	47.7379	47.5208	47.6168
long	-122.257	-122.319	-122.233	-122.393	-122.045
sqft_living15	1340	1690	2720	1360	1800
sqft_lot15	5650	7639	8062	5000	7503

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             21613 non-null  int64
 1   date           21613 non-null  datetime64[ns]
 2   price          21613 non-null  float64
 3   bedrooms       21613 non-null  int64
 4   bathrooms      21613 non-null  float64
 5   sqft_living    21613 non-null  int64
 6   sqft_lot       21613 non-null  int64
 7   floors         21613 non-null  float64
 8   waterfront     21613 non-null  int64
 9   view           21613 non-null  int64
 10  condition      21613 non-null  int64
 11  grade          21613 non-null  int64
 12  sqft_above     21613 non-null  int64
 13  sqft_basement  21613 non-null  int64
 14  yr_built       21613 non-null  int64
 15  yr_renovated   21613 non-null  int64
 16  zipcode        21613 non-null  int64
 17  lat            21613 non-null  float64
 18  long           21613 non-null  float64
 19  sqft_living15  21613 non-null  int64
```

```
 20  sqft_lot15     21613 non-null  int64
dtypes: datetime64[ns](1), float64(5), int64(15)
memory usage: 3.5 MB
```

Much better! Note how the variable type changed for `date`. On the topic of variable types, it seems surprising that `bathrooms` and `floors` are of type `float64`. Let's check if there is anything unusual about these variables

```
df['bathrooms'].value_counts()
```

```
bathrooms
2.50    5380
1.00    3852
1.75    3048
2.25    2047
2.00    1930
1.50    1446
2.75    1185
3.00     753
3.50     731
3.25     589
3.75     155
4.00     136
4.50     100
4.25      79
0.75      72
4.75      23
5.00      21
5.25      13
0.00      10
5.50      10
1.25       9
6.00       6
0.50       4
5.75       4
6.75       2
8.00       2
6.25       2
6.50       2
7.50       1
7.75       1
Name: count, dtype: int64
```

```
df['floors'].value_counts()
```

```
floors
1.0    10680
```

```
2.0     8241
1.5     1910
3.0      613
2.5      161
3.5        8
Name: count, dtype: int64
```

It seems that the number of bathrooms and floors is not always an integer. This is a bit surprising, but a possible interpretation is that in the case of bathrooms, smaller bathrooms with e.g., only a toilet and a sink are counted as 0.5 bathrooms, while a full bathroom would also need a shower or a bathtub. The same logic could apply to floors, where a split-level house could have, e.g., 1.5 floors. This is just a guess, but it seems plausible.

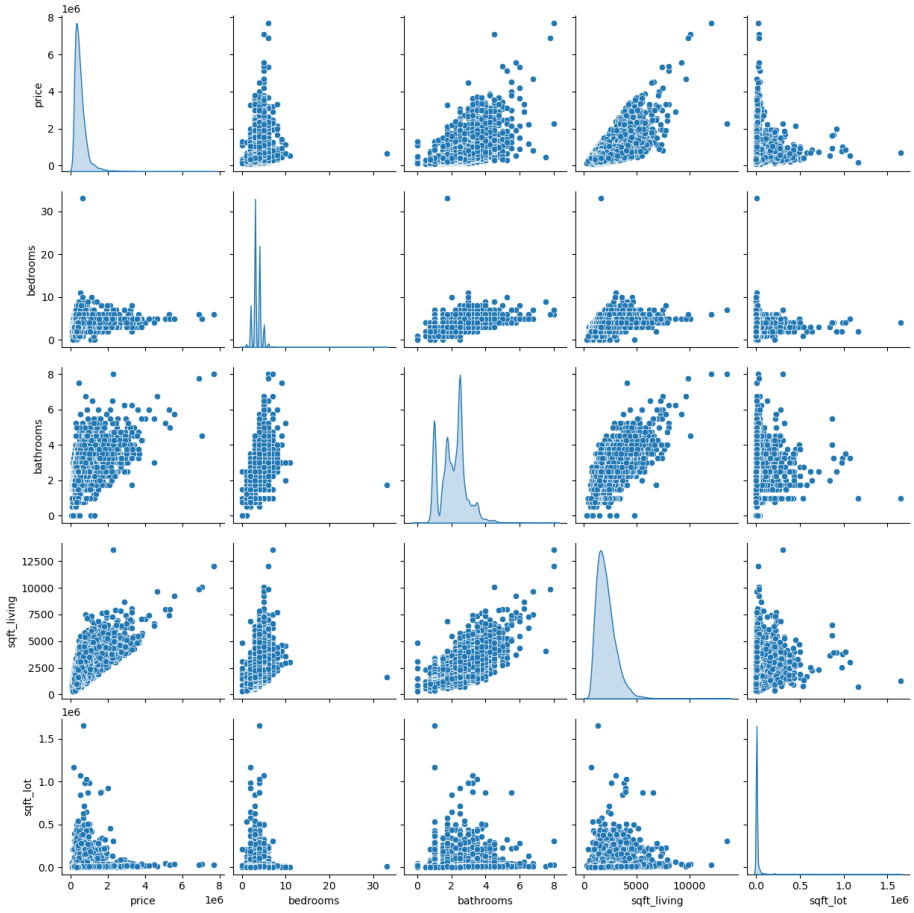
Note also that there do not seem to be any missing values, at least none were encoded as such. Now, let's look at the summary statistics of the dataset

```
df.describe().T
```

	count	mean	min	25%	50%
id	21613.0	4580301520.864988	1000102.0	2123049194.0	390493041
date	21613	2014-10-29 04:38:01.959931648	2014-05-02 00:00:00	2014-07-22 00:00:00	2014-10-16
price	21613.0	540088.141767	75000.0	321950.0	450000.0
bedrooms	21613.0	3.370842	0.0	3.0	3.0
bathrooms	21613.0	2.114757	0.0	1.75	2.25
sqft_living	21613.0	2079.899736	290.0	1427.0	1910.0
sqft_lot	21613.0	15106.967566	520.0	5040.0	7618.0
floors	21613.0	1.494309	1.0	1.0	1.5
waterfront	21613.0	0.007542	0.0	0.0	0.0
view	21613.0	0.234303	0.0	0.0	0.0
condition	21613.0	3.40943	1.0	3.0	3.0
grade	21613.0	7.656873	1.0	7.0	7.0
sqft_above	21613.0	1788.390691	290.0	1190.0	1560.0
sqft_basement	21613.0	291.509045	0.0	0.0	0.0
yr_built	21613.0	1971.005136	1900.0	1951.0	1975.0
yr_renovated	21613.0	84.402258	0.0	0.0	0.0
zipcode	21613.0	98077.939805	98001.0	98033.0	98065.0
lat	21613.0	47.560053	47.1559	47.471	47.5718
long	21613.0	-122.213896	-122.519	-122.328	-122.23
sqft_living15	21613.0	1986.552492	399.0	1490.0	1840.0
sqft_lot15	21613.0	12768.455652	651.0	5100.0	7620.0

Let's have a look at the pair plot of some of the quantitative variables

```
sns.pairplot(df[['price', 'bedrooms', 'bathrooms', 'sqft_living',
↪  'sqft_lot']], diag_kind='kde')
```



Unsurprisingly, there seems to be a positive correlation between the square footage of the living area (or number of bedrooms, or number of bathrooms) and the price of a house. However, there does not seem to be such a relationship between the square footage of the lot and the price. This is more surprising given that land prices can be very high in some areas. However, if these "houses" include many apartments (that do not include the land they are built on), this could explain the lack of a relationship. There also seems to be one house with more than 30 bedrooms. This seems a bit unusual, so let's have a closer look

```
df.query('bedrooms > 30').T
```

	15870
id	2402100895
date	2014-06-25 00:00:00
price	640000.0
bedrooms	33

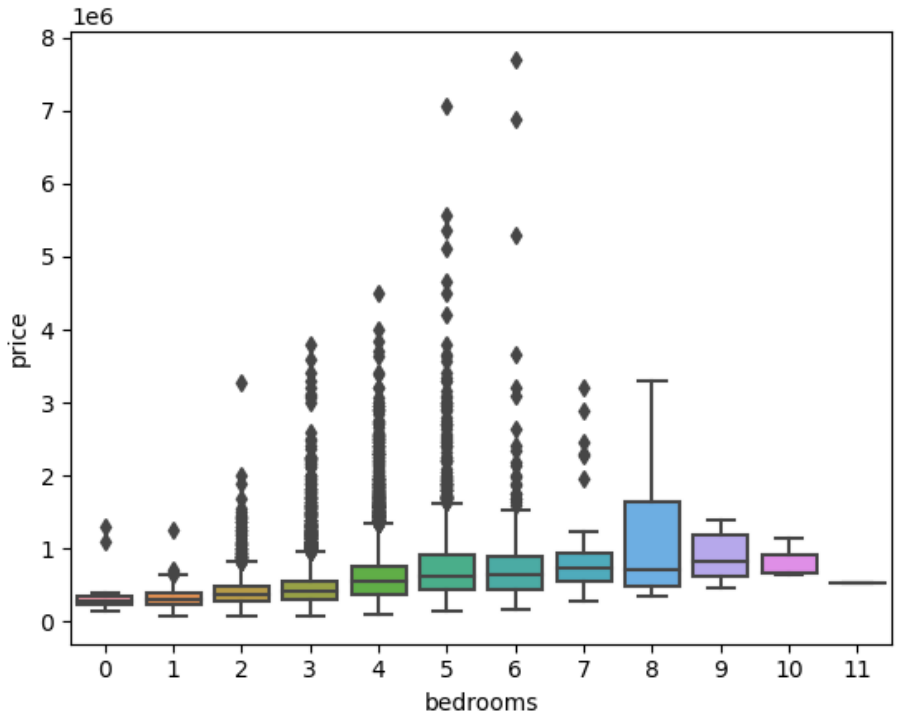
	15870
bathrooms	1.75
sqft_living	1620
sqft_lot	6000
floors	1.0
waterfront	0
view	0
condition	5
grade	7
sqft_above	1040
sqft_basement	580
yr_built	1947
yr_renovated	0
zipcode	98103
lat	47.6878
long	-122.331
sqft_living15	1330
sqft_lot15	4700

What a bargain! A house with 33 bedrooms for only $640000! However, it just has 1.75 bathrooms. It's maybe not that good of a deal after all. Considering that 1040 square feet corresponds to around 96 m^2. This seems like an error in the data. We will remove this observation from the dataset

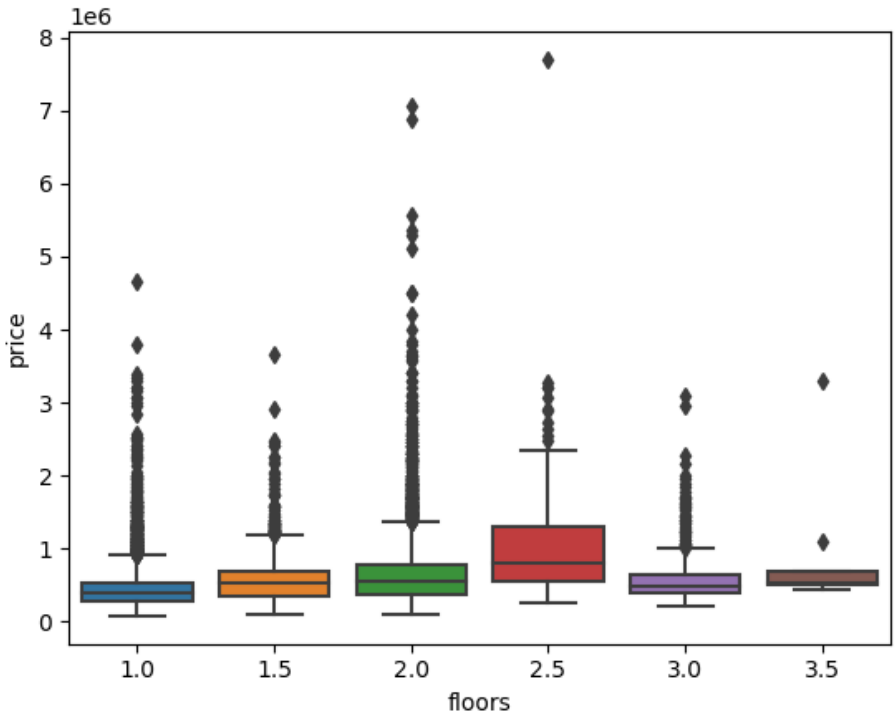
```
df = df.query('bedrooms < 30')
```

We could also look at the distribution of the number of bedrooms and floors and how if affects prices

```
sns.boxplot(x=df['bedrooms'],y=df['price'])
```




```
sns.boxplot(x=df['floors'],y=df['price'])
```



There seems to be great variability in the prices for a given number of bedrooms or floors.

Interestingly, we also have latitudinal and longitudinal information. We can use this to plot the houses on a map. Let's do that

```
import folium
from folium.plugins import HeatMap

# Initalize the map
m = folium.Map(location=[47.5112, -122.257])

# Create Layers and add them to the map
layer_heat_map = folium.FeatureGroup(name='Heat Map').add_to(m)
layer_most_expensive = folium.FeatureGroup(name='10 Most
 ↪  Expensive Houses').add_to(m)
folium.LayerControl().add_to(m)

# Add a heatmap to a layer
data = df[['lat', 'long',
 ↪  'price']].groupby(['lat','long']).mean().reset_index().values.tolist()
 ↪  # Note for latitudes and longitudes that show up multiple
 ↪  times, we take the mean()
```

```
HeatMap(data, radius=8).add_to(layer_heat_map)

# Add the 10 most expensive houses to a layer
df_most_expensive_houses = df.sort_values(by=['price'],
 ↪  ascending=False).head(10)
for indice, row in df_most_expensive_houses.iterrows():
    folium.Marker(
        location=[row["lat"], row["long"]],
        popup=f"Price: {row['price']}",
        icon=folium.map.Icon(color='red')
    ).add_to(layer_most_expensive)

m
```

```
<folium.folium.Map at 0x7fca18232a00>
```

The 10 most expensive houses seem to be close to the waterfront and looking at the actual data, we can see that about half of them are indeed overlooking the waterfront

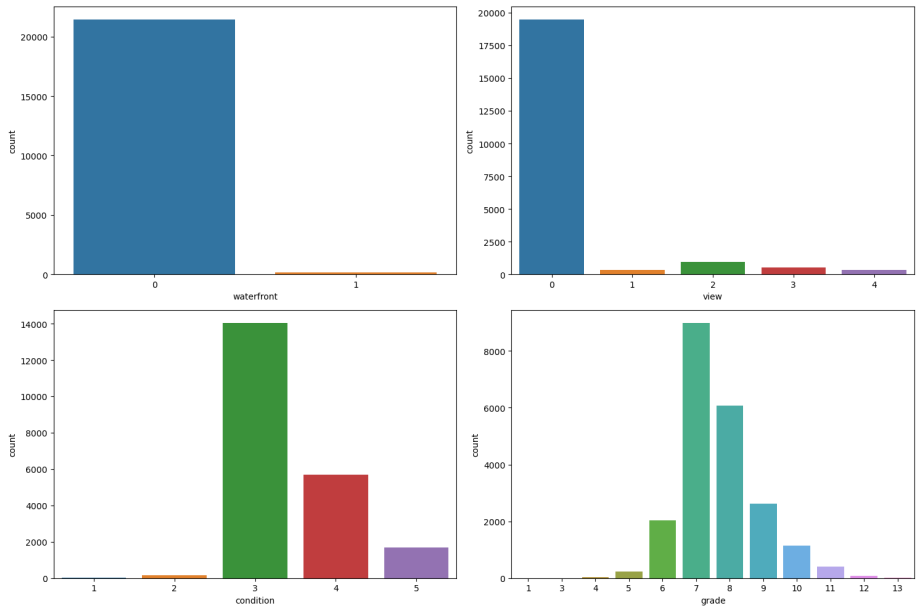
```
df_most_expensive_houses['waterfront']
```

```
7252    0
3914    1
9254    0
4411    0
1448    0
1315    1
1164    1
8092    1
2626    1
8638    0
Name: waterfront, dtype: int64
```

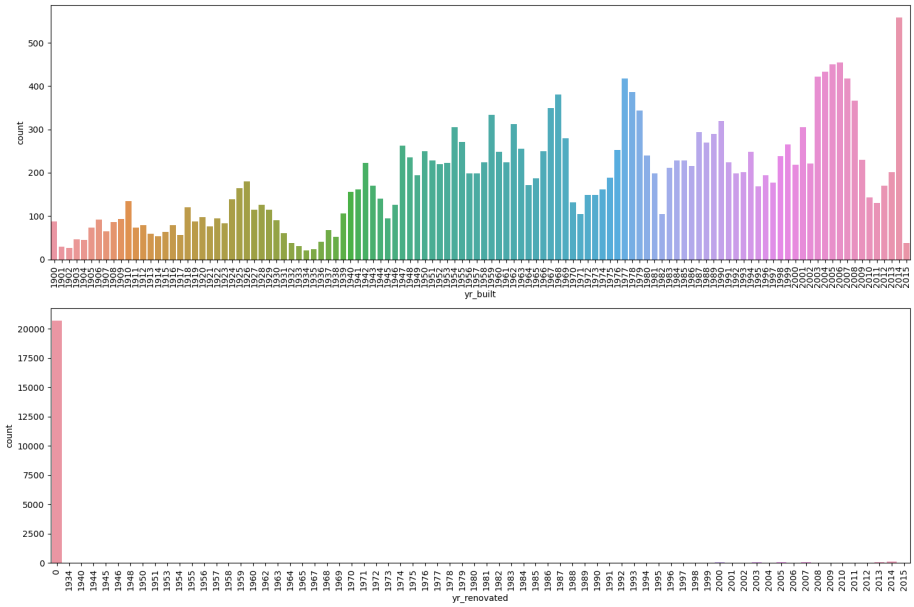
The heatmap also shows that the most expensive houses are located in the north-western part of the county, in or near Seattle.

Finally, let's look at the distribution of some of the discrete variables in the dataset

```
fig, axes = plt.subplots(2, 2, figsize=(15, 10))
variables = ['waterfront', 'view', 'condition', 'grade']

for var, ax in zip(variables, axes.flatten()):
    sns.countplot(x=var, data=df, ax=ax)

plt.tight_layout()
```




```
fig, axes = plt.subplots(2, 1, figsize=(15, 10))
variables = ['yr_built', 'yr_renovated']

for var, ax in zip(variables, axes.flatten()):
    sns.countplot(x=var, data=df, ax=ax)

for ax in axes.flatten():
    if ax.get_xlabel() in ('yr_built', 'yr_renovated'):
        ax.tick_params(axis='x', labelrotation=90)

plt.tight_layout()
```



There seems to be some cyclicality in `yr_built`. We could probably infer housing booms and busts if we analyze it carefully. `yr_renovated` seems to have a lot of zeros, which could mean that many houses have never been renovated. Let's check what's going on here

```
df['yr_renovated'].value_counts()
```

```
yr_renovated
0       20698
2014       91
2013       37
2003       36
2005       35
          ...
1951        1
1959        1
1948        1
1954        1
1944        1
Name: count, Length: 70, dtype: int64
```

Indeed, almost all of the houses seem to have a zero. However, some houses have values different from zero, so it might indeed be the case that houses with a value of zero have never been renovated. We could also check if the year of renovation is after the year the house was built

```
df.query('yr_renovated != 0 and yr_renovated < yr_built')
```

id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sq

With this command, we selected all observations where `yr_renovated` is different from zero and `yr_renovated < yr_built`. Since there were no rows selected, there do not seem to be any errors in the dataset in this respect.

Another thing we can check is whether there are errors in the square footage variables. For example, we could check if the sum of `sqft_above` and `sqft_basement` is equal to `sqft_living`

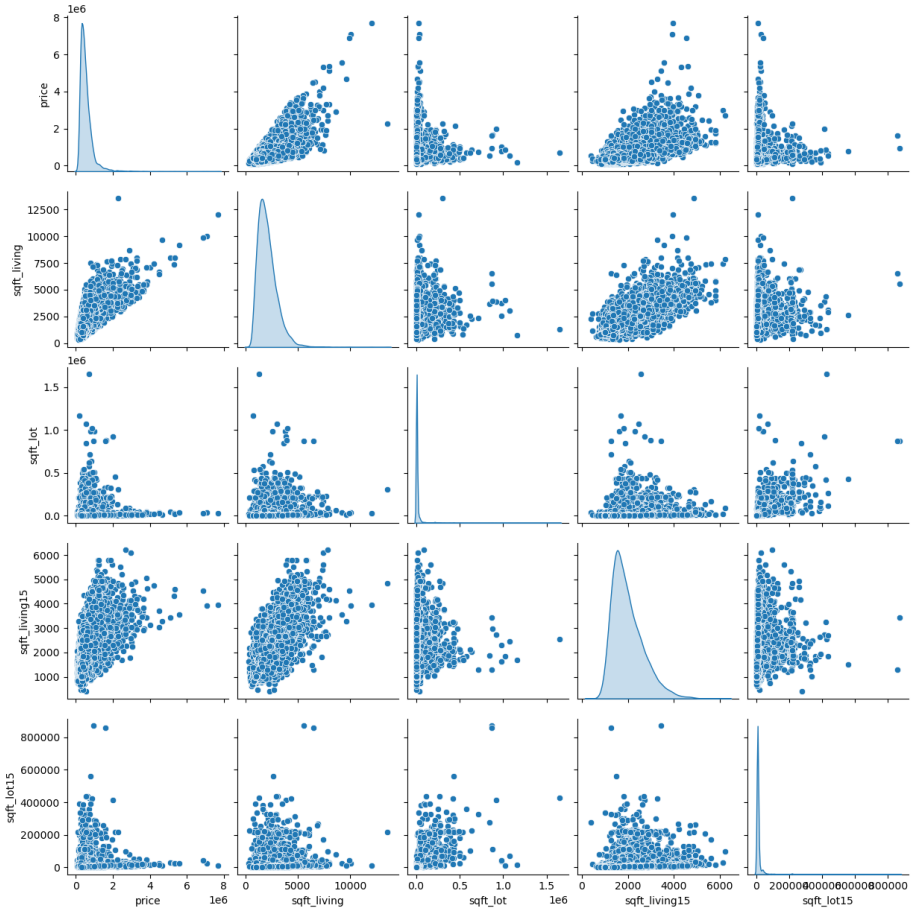
```
df.query('sqft_above + sqft_basement != sqft_living')
```

id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sq

This indeed seems to be correct for all observations.

We haven't looked at the square footage of the 15 nearest neighbors yet. Let's check how it relates to price and the square footage of the house itself

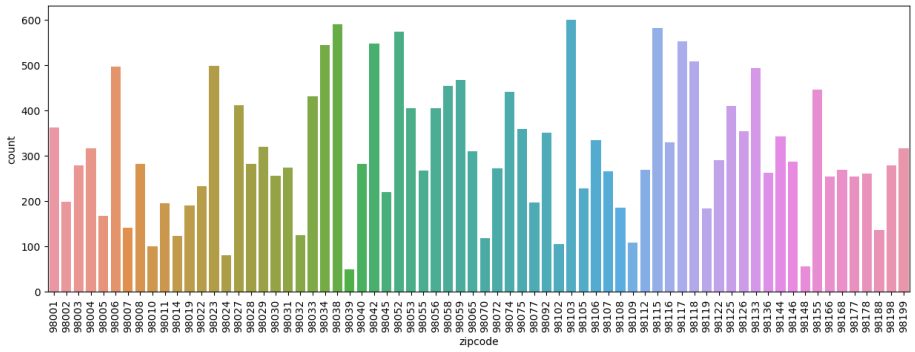
```
sns.pairplot(df[['price', 'sqft_living', 'sqft_lot',
 ↪  'sqft_living15', 'sqft_lot15']], diag_kind='kde')
```



There seems to be a positive relationship between the square footage of the living area of the house and the square footage of the living area of the 15 nearest neighbors. There also seems to be a positive relationship with price. This likely just reflects the fact that neighborhoods tend to have houses of similar sizes and prices.

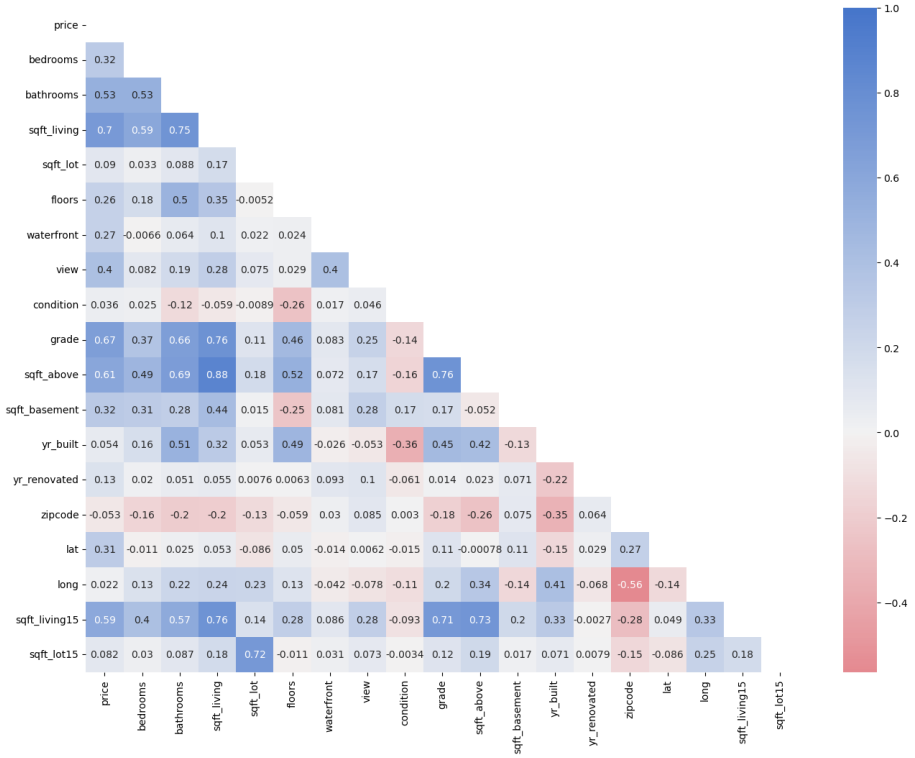
Finally, let's look at the distribution of the zip codes in the dataset

```
plt.figure(figsize=(15, 5))
sns.countplot(x='zipcode', data=df)
plt.xticks(rotation=90)
plt.show()
```



This likely doesn't tell us much, but it's interesting to see that some zip codes are much more common than others. Finally, we can again look at the correlation between variables in our dataset

```
f, ax = plt.subplots(figsize=(16, 12))
corr = df.drop(['id', 'date'], axis=1).corr()
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, annot=True, 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()
```



7.6 Implementation of House Price Prediction Models

We have explored our dataset and are now ready to implement machine learning algorithms for house price prediction. Let's start by importing the required libraries

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler,
 ↪  OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Lasso, LassoCV
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from xgboost import XGBRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error,
 ↪  mean_absolute_error, r2_score
from joblib import dump, load
```

7.6.1 Data Preprocessing

The dataset seems to be pretty clean already. Let's check again the number of missing values

```
df.isnull().sum()
```

```
id               0
date             0
price            0
bedrooms         0
bathrooms        0
sqft_living      0
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
grade            0
sqft_above       0
sqft_basement    0
yr_built         0
yr_renovated     0
zipcode          0
lat              0
long             0
sqft_living15    0
sqft_lot15       0
dtype: int64
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 21612 entries, 0 to 21612
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             21612 non-null  int64
 1   date           21612 non-null  datetime64[ns]
 2   price          21612 non-null  float64
 3   bedrooms       21612 non-null  int64
 4   bathrooms      21612 non-null  float64
 5   sqft_living    21612 non-null  int64
 6   sqft_lot       21612 non-null  int64
 7   floors         21612 non-null  float64
 8   waterfront     21612 non-null  int64
 9   view           21612 non-null  int64
```

```
 10  condition      21612 non-null  int64
 11  grade          21612 non-null  int64
 12  sqft_above     21612 non-null  int64
 13  sqft_basement  21612 non-null  int64
 14  yr_built       21612 non-null  int64
 15  yr_renovated   21612 non-null  int64
 16  zipcode        21612 non-null  int64
 17  lat            21612 non-null  float64
 18  long           21612 non-null  float64
 19  sqft_living15  21612 non-null  int64
 20  sqft_lot15     21612 non-null  int64
dtypes: datetime64[ns](1), float64(5), int64(15)
memory usage: 3.6 MB
```

There don't seem to be any missing values. However, we could still check for duplicates

```
df.duplicated().sum()
```

```
0
```

There also don't seem to be any duplicates.

There are some variables such as `id`, `zipcode`, `lat` and `long` which likely don't provide very useful information given the other variables in the dataset. We will drop these variables

```
df = df.drop(['id', 'zipcode', 'lat', 'long'], axis=1)
df.head().T
```

	0	1	2	3
date	2014-10-13 00:00:00	2014-12-09 00:00:00	2015-02-25 00:00:00	2014-12-09 00:00:00
price	221900.0	538000.0	180000.0	604000.0
bedrooms	3	3	2	4
bathrooms	1.0	2.25	1.0	3.0
sqft_living	1180	2570	770	1960
sqft_lot	5650	7242	10000	5000
floors	1.0	2.0	1.0	1.0
waterfront	0	0	0	0
view	0	0	0	0
condition	3	3	3	5
grade	7	7	6	7
sqft_above	1180	2170	770	1050
sqft_basement	0	400	0	910
yr_built	1955	1951	1933	1965
yr_renovated	0	1991	0	0
sqft_living15	1340	1690	2720	1360

	0	1	2	3	4
sqft_lot15	5650	7639	8062	5000	7503

Binning & Encoding

Furthermore, we will need to convert the `date` variable into something that can be used in a machine-learning model. We will extract the year and month from the date and drop the original `date` variable

```
df['year_sale'] = pd.DatetimeIndex(df['date']).year
df['month_sale'] = pd.DatetimeIndex(df['date']).month
```

Furthermore, we can convert `yr_built` and `yr_renovated` into the age of the house and the number of years since the last renovation

```
df['age'] = df['year_sale'] - df['yr_built']
df['years_since_renovation'] = df['year_sale'] -
 ↪  np.maximum(df['yr_built'], df['yr_renovated'])
```

If the house has never been renovated, `years_since_renovation` will be equal to the age of the house. We can drop the original `yr_built`, `yr_renovated`, and `date` variables

```
df = df.drop(['yr_built', 'yr_renovated', 'date'], axis=1)
```

Let's check the summary statistics of the dataset again

```
df.describe().T
```

	count	mean	std	min	25%	50%	75%
price	21612.0	540083.518786	367135.061269	75000.0	321837.50	450000.00	645000.00
bedrooms	21612.0	3.369471	0.907982	0.0	3.00	3.00	4.00
bathrooms	21612.0	2.114774	0.770177	0.0	1.75	2.25	2.50
sqft_living	21612.0	2079.921016	918.456818	290.0	1426.50	1910.00	2550.00
sqft_lot	21612.0	15107.388951	41421.423497	520.0	5040.00	7619.00	10688.25
floors	21612.0	1.494332	0.539991	1.0	1.00	1.50	2.00
waterfront	21612.0	0.007542	0.086519	0.0	0.00	0.00	0.00
view	21612.0	0.234314	0.766334	0.0	0.00	0.00	0.00
condition	21612.0	3.409356	0.650668	1.0	3.00	3.00	4.00
grade	21612.0	7.656904	1.175477	1.0	7.00	7.00	8.00
sqft_above	21612.0	1788.425319	828.094487	290.0	1190.00	1560.00	2210.00
sqft_basement	21612.0	291.495697	442.580931	0.0	0.00	0.00	560.00
sqft_living15	21612.0	1986.582871	685.392610	399.0	1490.00	1840.00	2360.00
sqft_lot15	21612.0	12768.828984	27304.756179	651.0	5100.00	7620.00	10083.25
year_sale	21612.0	2014.322969	0.467622	2014.0	2014.00	2014.00	2015.00

	count	mean	std	min	25%	50%
month_sale	21612.0	6.574449	3.115377	1.0	4.00	6.00
age	21612.0	43.316722	29.375731	-1.0	18.00	40.00
years_since_renovation	21612.0	40.935730	28.813764	-1.0	15.00	37.00

Finally, we need to take care of the categorical variables in the dataset. We will use one-hot (aka 'one-of-K' or 'dummy') encoding for this purpose

```
# Define for which variables to do the one-hot encoding
categorical_variables = ['view', 'condition', 'grade']

# Initialize the encoder
encoder = OneHotEncoder(sparse_output=False)

# Apply the one-hot encoding to the desired columns
one_hot_encoded =
 ↪  encoder.fit_transform(df[categorical_variables])

# Convert the results to a DataFrame
df_one_hot_encoded = pd.DataFrame(one_hot_encoded,
 ↪  columns=encoder.get_feature_names_out(['view', 'condition',
 ↪  'grade']), index=df.index)

# Concatenate the one-hot encoded columns with the original
 ↪  DataFrame
df_encoded = pd.concat([df, df_one_hot_encoded], axis=1)

# Drop the old, unencoded columns from the old Dataframe
df_encoded = df_encoded.drop(categorical_variables, axis=1)
```

You can see that now we have many more dummy variables taking values zero or one in our dataset

```
df_encoded.describe().T
```

	count	mean	std	min	25%	50%
price	21612.0	540083.518786	367135.061269	75000.0	321837.50	450000.00
bedrooms	21612.0	3.369471	0.907982	0.0	3.00	3.00
bathrooms	21612.0	2.114774	0.770177	0.0	1.75	2.25
sqft_living	21612.0	2079.921016	918.456818	290.0	1426.50	1910.00
sqft_lot	21612.0	15107.388951	41421.423497	520.0	5040.00	7619.00
floors	21612.0	1.494332	0.539991	1.0	1.00	1.50
waterfront	21612.0	0.007542	0.086519	0.0	0.00	0.00
sqft_above	21612.0	1788.425319	828.094487	290.0	1190.00	1560.00

	count	mean	std	min	25%	50%	75%
sqft_basement	21612.0	291.495697	442.580931	0.0	0.00	0.00	560.00
sqft_living15	21612.0	1986.582871	685.392610	399.0	1490.00	1840.00	2360.00
sqft_lot15	21612.0	12768.828984	27304.756179	651.0	5100.00	7620.00	10083.25
year_sale	21612.0	2014.322969	0.467622	2014.0	2014.00	2014.00	2015.00
month_sale	21612.0	6.574449	3.115377	1.0	4.00	6.00	9.00
age	21612.0	43.316722	29.375731	-1.0	18.00	40.00	63.00
years_since_renovation	21612.0	40.935730	28.813764	-1.0	15.00	37.00	60.00
view_0	21612.0	0.901721	0.297698	0.0	1.00	1.00	1.00
view_1	21612.0	0.015362	0.122990	0.0	0.00	0.00	0.00
view_2	21612.0	0.044559	0.206337	0.0	0.00	0.00	0.00
view_3	21612.0	0.023598	0.151797	0.0	0.00	0.00	0.00
view_4	21612.0	0.014760	0.120595	0.0	0.00	0.00	0.00
condition_1	21612.0	0.001388	0.037232	0.0	0.00	0.00	0.00
condition_2	21612.0	0.007959	0.088857	0.0	0.00	0.00	0.00
condition_3	21612.0	0.649223	0.477224	0.0	0.00	1.00	1.00
condition_4	21612.0	0.262771	0.440149	0.0	0.00	0.00	1.00
condition_5	21612.0	0.078660	0.269214	0.0	0.00	0.00	0.00
grade_1	21612.0	0.000046	0.006802	0.0	0.00	0.00	0.00
grade_3	21612.0	0.000139	0.011781	0.0	0.00	0.00	0.00
grade_4	21612.0	0.001342	0.036607	0.0	0.00	0.00	0.00
grade_5	21612.0	0.011197	0.105226	0.0	0.00	0.00	0.00
grade_6	21612.0	0.094299	0.292252	0.0	0.00	0.00	0.00
grade_7	21612.0	0.415510	0.492821	0.0	0.00	0.00	1.00
grade_8	21612.0	0.280770	0.449386	0.0	0.00	0.00	1.00
grade_9	21612.0	0.120998	0.326132	0.0	0.00	0.00	0.00
grade_10	21612.0	0.052471	0.222980	0.0	0.00	0.00	0.00
grade_11	21612.0	0.018462	0.134618	0.0	0.00	0.00	0.00
grade_12	21612.0	0.004164	0.064399	0.0	0.00	0.00	0.00
grade_13	21612.0	0.000602	0.024519	0.0	0.00	0.00	0.00

Given that these categorical variables are ordinal, this might have not been strictly necessary. However, is required if you have data that is not ordinal.

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_encoded.drop('price', axis=1) # All variables except
 ↪  `SeriousDlqin2yrs`
y = df_encoded[['price']] # Only SeriousDlqin2yrs
```

We will use 80% of the data for training and 20% for testing. Note that since our target variable is continuous, we don't need to stratify the split

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
 ↪  test_size = 0.2, random_state = 42)
```

Scaling Features

To improve the performance of our machine learning model, we should scale the features. We will use the `StandardScaler` and `MinMaxScaler`class from the `sklearn.preprocessing` module to scale the features. The `StandardScaler` scales each feature to have a mean of 0 and a standard deviation of 1. The `MinMaxScaler` scales each feature to a given range, usually 0 to 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


# Define which features to scale with the StandardScaler and
 ↪  MinMaxScaler
for_standard_scaler = [
    'bedrooms',
    'bathrooms',
    'sqft_living',
    'sqft_lot',
    'floors',
    'sqft_above',
    'sqft_basement',
    'sqft_living15',
    'sqft_lot15',
    'age',
    'years_since_renovation'
]

for_min_max_scaler = [
    'year_sale',
    'month_sale'
```

```
]

# Apply the standard scaler (Note: we use the same mean and std
 ↪  for scaling the test set)
standard_scaler = StandardScaler()
scale_features(standard_scaler, X_train, for_standard_scaler)
scale_features(standard_scaler, X_test, for_standard_scaler,
 ↪  only_transform=True)

# Apply the minmax scaler (Note: we use the same min and max for
 ↪  scaling the test set)
minmax_scaler = MinMaxScaler()
scale_features(minmax_scaler, X_train, for_min_max_scaler)
scale_features(minmax_scaler, X_test, for_min_max_scaler,
 ↪  only_transform=True)

# Apply standard scaler to the target variable
target_scaler = StandardScaler()
y_train = pd.DataFrame(target_scaler.fit_transform(y_train),
 ↪  columns=['price'])
y_test = pd.DataFrame(target_scaler.transform(y_test),
 ↪  columns=['price'])
```

7.6.2 Evaluation Criertia

We will evaluate our models based on the following criteria

- **Root Mean Squared Error (MSE)**: Square root of the mean of the squared differences between the predicted and the actual values
- **Mean Absolute Error (MAE)**: Mean of the absolute differences between the predicted and the actual values
- **R-squared (R2)**: Proportion of the variance in the dependent variable that is predictable from the independent variables

We define a function that a function that will calculate these metrics for us

```
def evaluate_model(model, X_train, y_train, X_test, y_test,
 ↪  label='', print_results=True):

    # Predict the target variable
    y_pred_train = model.predict(X_train)
    y_pred_test = model.predict(X_test)

    # Transform the target variable back to the original scale
    # (This makes it easier to interpret the RMSE and MAE)
    y_train_inv = target_scaler.inverse_transform(y_train)
```

```
    y_test_inv = target_scaler.inverse_transform(y_test)
    y_pred_train_inv =
↪   target_scaler.inverse_transform(y_pred_train.reshape(-1, 1))
    y_pred_test_inv =
↪   target_scaler.inverse_transform(y_pred_test.reshape(-1, 1))

    # Calculate the evaluation metrics
    rmse_train = mean_squared_error(y_train_inv,
↪   y_pred_train_inv, squared=False)
    rmse_test = mean_squared_error(y_test_inv, y_pred_test_inv,
↪   squared=False)
    mae_train = mean_absolute_error(y_train_inv,
↪   y_pred_train_inv)
    mae_test = mean_absolute_error(y_test_inv, y_pred_test_inv)
    r2_train = r2_score(y_train_inv, y_pred_train_inv)
    r2_test = r2_score(y_test_inv, y_pred_test_inv)

    # Print the evaluation metrics
    if print_results:

        ↪  print(f"-----------------------------------------------------------------")
        print(f"Metrics: {label}")

        ↪  print(f"-----------------------------------------------------------------")
        print(f"RMSE (Train): {rmse_train}")
        print(f"MAE (Train): {mae_train}")
        print(f"R2 (Train): {r2_train}")

        ↪  print(f"-----------------------------------------------------------------")
        print(f"RMSE (Test): {rmse_test}")
        print(f"MAE (Test): {mae_test}")
        print(f"R2 (Test): {r2_test}")

        ↪  print(f"-----------------------------------------------------------------")

    return rmse_train, rmse_test, mae_train, mae_test, r2_train,
     ↪  r2_test
```

7.6.3 Linear Regression

We will start by training a simple linear regression model using only a few basic features

```
basic_features = ['bedrooms', 'bathrooms', 'sqft_living']
reg_lin_basic = LinearRegression().fit(X_train[basic_features],
 ↪  y_train)
```

We can evaluate the model using the function we defined earlier

```
evaluate_model(reg_lin_basic, X_train[basic_features], y_train,
 ↪  X_test[basic_features], y_test, label = 'Linear Regression
 ↪  (Basic Features)');
```

```
------------------------------------------------------------
Metrics: Linear Regression (Basic Features)
------------------------------------------------------------
RMSE (Train): 253683.12629128053
MAE (Train): 168994.9071419931
R2 (Train): 0.5085183567620137
------------------------------------------------------------
RMSE (Test): 271925.6970016718
MAE (Test): 174452.3090166579
R2 (Test): 0.5073277848405491
------------------------------------------------------------
```

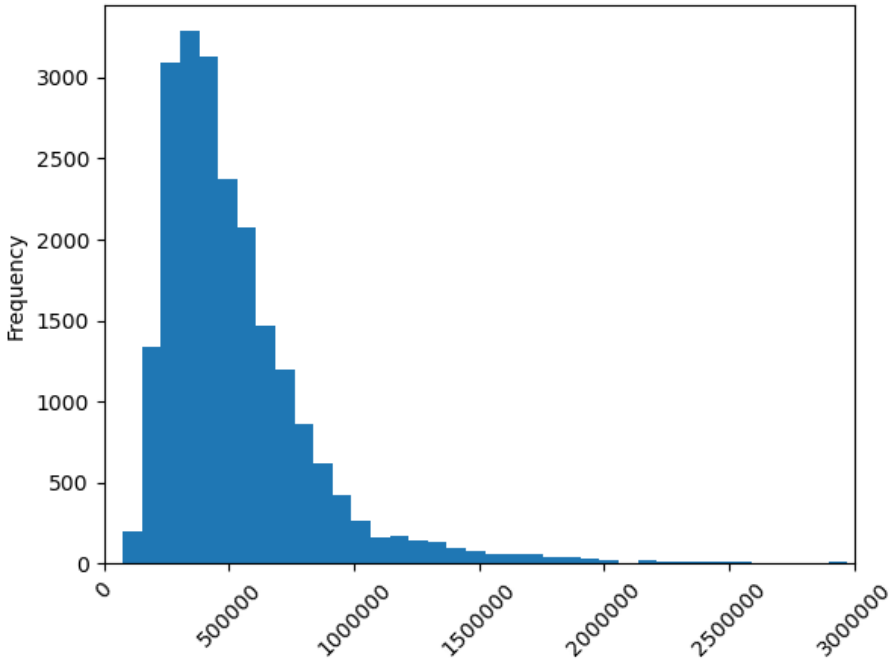
We are not doing that badly with a RMSE of around $250000 if we take into account the minimum and maximum prices in the dataset

```
print(f'Min Price: {df["price"].min()}, Max Price:
 ↪  {df["price"].max()}')
```

```
Min Price: 75000.0, Max Price: 7700000.0
```

and the distribution of prices

```
ax = df['price'].plot.hist(bins=100)
ax.ticklabel_format(useOffset=False,style='plain')
ax.tick_params(axis='x', labelrotation=45)
ax.set_xlim(0,3000000)
```



Let's now try a linear regression but with all the features

```
reg_lin = LinearRegression().fit(X_train, y_train)
```

We can evaluate the model using the function we defined earlier

```
evaluate_model(reg_lin, X_train, y_train, X_test, y_test, label =
↪  'Linear Regression (All Features)');
```

```
------------------------------------------------------------
Metrics: Linear Regression (All Features)
------------------------------------------------------------
RMSE (Train): 203680.13271422562
MAE (Train): 133387.29115116654
R2 (Train): 0.6831735158523966
------------------------------------------------------------
RMSE (Test): 215930.28643225055
MAE (Test): 137547.23127374452
R2 (Test): 0.6893404625153258
------------------------------------------------------------
```

The performance of the model has improved. Since we have a large sample size but relatively few regressors it is unlikely to overfit. Note, however, that if we add more regressors, e.g., squared and cubed features, etc. we might run into trouble at a certain point. That's why it's important to use the train-test split

to check that our model generalizes.

7.6.4 LASSO Regression

One way to deal with overfitting in a linear regression is to use LASSO regression. LASSO regression is a type of linear regression that uses a penalty (or regularization) term to shrink the coefficients of the regressors towards zero. Essentially, LASSO selects a subset of features, which can help to prevent overfitting. We will use the `Lasso` class from the `sklearn.linear_model` module to train a LASSO regression model

```
reg_lasso = Lasso(alpha=0.1).fit(X_train, y_train)
```

We can evaluate the model using the function we defined earlier

```
evaluate_model(reg_lasso, X_train, y_train, X_test, y_test, label
 ↪  = 'LASSO Regression');
```

```
----------------------------------------------------------------
Metrics: LASSO Regression
----------------------------------------------------------------
RMSE (Train): 254661.31873140397
MAE (Train): 163764.59579244166
R2 (Train): 0.5047207803287292
----------------------------------------------------------------
RMSE (Test): 273446.0239203981
MAE (Test): 168730.33198849438
R2 (Test): 0.5018033587261654
----------------------------------------------------------------
```

This model is doing a bit worse than a standard linear regression. However, we just chose the value of the penalty term α arbitrarily. We can use cross-validation to find the best value of α

```
reg_lasso_cv = LassoCV(cv=5, random_state=42).fit(X_train,
 ↪  y_train.values.ravel())
```

This command repeatedly runs 5-fold cross-validation for a LASSO regression using different values of α. The α that minimizes the mean squared error is then stored in the `alpha_` attribute of the model

```
reg_lasso_cv.alpha_
```

```
0.0007018253833076978
```

This α is much smaller than our initial value. Let's see how well it does in terms of the RMSE

```
evaluate_model(reg_lasso_cv, X_train, y_train, X_test, y_test,
 ↪  label = 'LASSO Regression (CV)');
```

```
------------------------------------------------------------
Metrics: LASSO Regression (CV)
------------------------------------------------------------
RMSE (Train): 204360.76389300276
MAE (Train): 134141.73731503487
R2 (Train): 0.6810525207180653
------------------------------------------------------------
RMSE (Test): 217146.28161042708
MAE (Test): 138085.31092628682
R2 (Test): 0.6858316989155063
------------------------------------------------------------
```

It's always a good idea to use cross-validation to find the best hyperparameters for your model. For more complicated models with several hyperparameter choices, one can use `GridSearchCV` or `RandomizedSearchCV` from `sklearn` to find the hyperparameters.

We can check which coefficients the LASSO regression has shrunk to zero because of the regularization term

```
X_train.columns[np.abs(reg_lasso_cv.coef_) < 1e-12]
```

```
Index(['sqft_above', 'view_1', 'view_3', 'condition_1', 'condition_3',
       'grade_1', 'grade_3'],
      dtype='object')
```

Compare this to the linear regression where none of the coefficients were zero

```
X_train.columns[(np.abs(reg_lin.coef_) < 1e-12).reshape(-1)]
```

```
Index([], dtype='object')
```

7.6.5 Decision Tree

We will now train a decision tree regressor on the data

```
reg_tree = DecisionTreeRegressor(random_state=42).fit(X_train,
 ↪  y_train)
```

We can evaluate the model using the function we defined earlier

```
evaluate_model(reg_tree, X_train, y_train, X_test, y_test, label
 ↪  = 'Decision Tree');
```

```
------------------------------------------------------------
Metrics: Decision Tree
```

```
------------------------------------------------------------
RMSE (Train): 0.0
MAE (Train): 0.0
R2 (Train): 1.0
------------------------------------------------------------
RMSE (Test): 276927.1448808092
MAE (Test): 163650.69442516772
R2 (Test): 0.48903797314415076
------------------------------------------------------------
```

The decision tree perfectly fits the training data but does not generalize well to the test data. Why did this happen? We did not change any of the default hyperparameters of the decision tree which resulted in the decision tree overfitting, i.e., it learned the noise in the training data. We can try to reduce the depth of the tree to prevent overfitting

```
reg_tree = DecisionTreeRegressor(max_depth=10,
 ↪  random_state=42).fit(X_train, y_train)
evaluate_model(reg_tree, X_train, y_train, X_test, y_test, label
 ↪  = 'Decision Tree');
```

```
------------------------------------------------------------
Metrics: Decision Tree
------------------------------------------------------------
RMSE (Train): 151229.68262754745
MAE (Train): 104245.46532488744
R2 (Train): 0.8253380836313531
------------------------------------------------------------
RMSE (Test): 242624.93510350658
MAE (Test): 139455.8139319333
R2 (Test): 0.6077811751936795
------------------------------------------------------------
```

This seems to have improved the performance of the model. However, we need a more rigorous way to find the best hyperparameters. One such way is to use grid search, which tries many different hyperparameter values. We, then, combine this with cross-validation to find the best hyperparameters for the decision tree. `GridSearchCV` from the `sklearn` package does exactly that

```
param_grid = {
    'max_depth': [5, 10, 15, 20],
    'min_samples_split': [2, 5, 10, 15],
    'min_samples_leaf': [1, 2, 5, 10]
}

reg_tree_cv =
 ↪  GridSearchCV(DecisionTreeRegressor(random_state=42),
 ↪  param_grid, cv=5).fit(X_train, y_train)
```

Note that `param_grid` is a dictionary where the keys are the hyperparameters of the decision tree and the values are lists of the values we want to try. The best hyperparameters are stored in the `best_params_` attribute of the model

```
reg_tree_cv.best_params_
```

```
{'max_depth': 10, 'min_samples_leaf': 5, 'min_samples_split': 15}
```

We can then evaluate the model using the best hyperparameters

```
evaluate_model(reg_tree_cv, X_train, y_train, X_test, y_test,
 ↪  label = 'Decision Tree (CV)');
```

```
------------------------------------------------------------
Metrics: Decision Tree (CV)
------------------------------------------------------------
RMSE (Train): 165756.35013566245
MAE (Train): 111454.50933401058
R2 (Train): 0.7901714932986897
------------------------------------------------------------
RMSE (Test): 233527.64612876996
MAE (Test): 137387.4802938534
R2 (Test): 0.6366424628160059
------------------------------------------------------------
```

Note that using `reg_tree_cv` as the model to be evaluated uses automatically the best estimator. Alternatively, we could also use `best_estimator_` attribute in `evaluate_model`

```
reg_tree_cv.best_estimator_
```

```
DecisionTreeRegressor(max_depth=10, min_samples_leaf=5, min_samples_split=15,
                      random_state=42)
```

7.6.6 Random Forest

We will now train a random forest regressor on the data

```
reg_rf = RandomForestRegressor(random_state=42).fit(X_train,
 ↪  y_train)
```

We can evaluate the model using the function we defined earlier

```
evaluate_model(reg_rf, X_train, y_train, X_test, y_test, label =
 ↪  'Random Forest');
```

```
------------------------------------------------------------
Metrics: Random Forest
```

```
------------------------------------------------------------
RMSE (Train): 66638.2969151954
MAE (Train): 42418.54030134768
R2 (Train): 0.9660865542789411
------------------------------------------------------------
RMSE (Test): 207350.8910584885
MAE (Test): 120473.06515845477
R2 (Test): 0.713536442057418
------------------------------------------------------------
```

Let's use grid search with cross-validation to find the best hyperparameters for the random forest

```
param_grid = {
    'max_depth': [5, 10, 15, 20],
    'n_estimators': [50, 100, 150, 200, 300],
}

#reg_rf_cv = GridSearchCV(RandomForestRegressor(random_state=42),
 ↪  param_grid, cv=5).fit(X_train, y_train)
#dump(reg_rf_cv, 'reg_rf_cv.joblib')
```

```
reg_rf_cv = load('reg_rf_cv.joblib')
```

We are trying 20 different hyperparameter combinations and for each parameter combination we will have to estimate the model 5 times (5-fold cross-validation). This might take a while. The best hyperparameters are stored in the `best_params_` attribute of the model

```
reg_rf_cv.best_params_
```

```
{'max_depth': 20, 'n_estimators': 300}
```

We can then evaluate the model using the best hyperparameters

```
evaluate_model(reg_rf_cv, X_train, y_train, X_test, y_test, label
 ↪  = 'Random Forest (CV)');
```

```
------------------------------------------------------------
Metrics: Random Forest (CV)
------------------------------------------------------------
RMSE (Train): 72654.48965997726
MAE (Train): 49716.92916875867
R2 (Train): 0.959686634834322
------------------------------------------------------------
RMSE (Test): 206073.35722748775
MAE (Test): 120254.77646893183
R2 (Test): 0.7170554960056299
```

```
----------------------------------------------------------------
```

The tuned random forest model performs a bit better than the one with the default values. However, the improvement is not that big. This is likely because the default values of the random forest are already quite good. We could try to test more hyperparameters in the grid search. Note that we chose the highest value for both parameters. Thus, we could try even higher values. However, this would increase the computational time.

7.6.7 XGBoost

We will now train an XGBoost regressor on the data

```
reg_xgb = XGBRegressor(random_state=42).fit(X_train, y_train)
```

We can evaluate the model using the function we defined earlier

```
evaluate_model(reg_xgb, X_train, y_train, X_test, y_test, label =
 ↪  'XGBoost');
```

```
----------------------------------------------------------------
Metrics: XGBoost
----------------------------------------------------------------
RMSE (Train): 101571.54001161143
MAE (Train): 76626.15270638122
R2 (Train): 0.9212105237017153
----------------------------------------------------------------
RMSE (Test): 204360.86407090884
MAE (Test): 120757.43200613001
R2 (Test): 0.7217385587721041
----------------------------------------------------------------
```

Let's use grid search with cross-validation to find the best hyperparameters for the XGBoost

```
param_grid = {
    'max_depth': [5, 10, 15, 20],
    'n_estimators': [50, 100, 150, 200, 300],
}

#reg_xgb_cv = GridSearchCV(XGBRegressor(random_state=42),
 ↪  param_grid, cv=5).fit(X_train, y_train)
#dump(reg_xgb_cv, 'reg_xgb_cv.joblib')
```

```
reg_xgb_cv = load('reg_xgb_cv.joblib')
```

The best hyperparameters are stored in the `best_params_` attribute of the model

```
reg_xgb_cv.best_params_
```

```
{'max_depth': 5, 'n_estimators': 50}
```

We can then evaluate the model using the best hyperparameters

```
evaluate_model(reg_xgb_cv, X_train, y_train, X_test, y_test,
 ↪  label = 'XGBoost (CV)');
```

```
------------------------------------------------------------
Metrics: XGBoost (CV)
------------------------------------------------------------
RMSE (Train): 134323.31888964228
MAE (Train): 99419.23493894239
R2 (Train): 0.862207059779255
------------------------------------------------------------
RMSE (Test): 201839.3762899356
MAE (Test): 123656.21579704488
R2 (Test): 0.7285628038252234
------------------------------------------------------------
```

Again, the tuned XGBoost model performs a bit better than the one with the default values. However, the improvement is not that big.

7.6.8 Neural Network

Finally, let's try to train a neural network on the data

```
#reg_nn = MLPRegressor(random_state=42,
 ↪  verbose=True).fit(X_train, y_train)
#dump(reg_nn, 'reg_nn.joblib')
```

```
reg_nn = load('reg_nn.joblib')
```

We can evaluate the model using the function we defined earlier

```
evaluate_model(reg_nn, X_train, y_train, X_test, y_test, label =
 ↪  'Neural Network');
```

```
------------------------------------------------------------
Metrics: Neural Network
------------------------------------------------------------
RMSE (Train): 142063.16391660276
MAE (Train): 101370.39765456515
R2 (Train): 0.8458700261274621
------------------------------------------------------------
RMSE (Test): 201217.79803902542
MAE (Test): 125534.39997216879
```

```
R2 (Test): 0.7302320486389542
----------------------------------------------------------------
```

We can try to improve the performance of the neural network by tuning the hyperparameters. We will use grid search with cross-validation to find the best hyperparameters for the neural network

```
param_grid = {
    'hidden_layer_sizes': [(100,), (100, 100), (200,), (200,
     ↪  100)],
    'alpha': [0.0001, 0.001, 0.01, 0.1],
}

#reg_nn_cv = GridSearchCV(MLPRegressor(random_state=42,
 ↪  verbose=True), param_grid, cv=5).fit(X_train, y_train)
#dump(reg_nn_cv, 'reg_nn_cv.joblib')
```

```
reg_nn_cv = load('reg_nn_cv.joblib')
```

The best hyperparameters are stored in the `best_params_` attribute of the model

```
reg_nn_cv.best_params_
```

```
{'alpha': 0.1, 'hidden_layer_sizes': (100,)}
```

We can then evaluate the model using the best hyperparameters

```
evaluate_model(reg_nn_cv, X_train, y_train, X_test, y_test, label
 ↪  = 'Neural Network');
```

```
----------------------------------------------------------------
Metrics: Neural Network
----------------------------------------------------------------
RMSE (Train): 155891.59268622578
MAE (Train): 108538.19421278372
R2 (Train): 0.814403608721182
----------------------------------------------------------------
RMSE (Test): 192879.83144507415
MAE (Test): 122558.79964553488
R2 (Test): 0.7521258686276469
----------------------------------------------------------------
```

The tuned neural network model performs a bit better than the one with the default values.

7.7 Model Evaluation

Let's summarize the results of our models

```
models = {
    "Linear Regression" : reg_lin,
    "LASSO Regression" : reg_lasso_cv,
    "Decision Tree" : reg_tree_cv,
    "Random Forest" : reg_rf_cv,
    "XGBoost" : reg_xgb_cv,
    "Neural Network" : reg_nn_cv
}

results = pd.DataFrame(columns=['Model', 'RMSE Train', 'RMSE
 ↪  Test', 'MAE Train', 'MAE Test', 'R2 Train', 'R2 Test'])

for modelName in models:

    # Evaluate the current model
    rmse_train, rmse_test, mae_train, mae_test, r2_train, r2_test
 ↪  = evaluate_model(models[modelName], X_train, y_train, X_test,
 ↪  y_test, print_results=False)

    # Store the results
    res = {
        'Model': modelName,
        'RMSE Train': rmse_train,
        'RMSE Test': rmse_test,
        'MAE Train': mae_train,
        'MAE Test': mae_test,
        'R2 Train': r2_train,
        'R2 Test': r2_test
    }

    df_tmp = pd.DataFrame(res, index=[0])

    results = pd.concat([results, df_tmp], axis=0,
 ↪  ignore_index=True)

# Sort the results by the RMSE of the test set
results = results.sort_values(by='RMSE
 ↪  Test').reset_index(drop=True)

results
```

	Model	RMSE Train	RMSE Test	MAE Train	MAE Test	R2 Train
0	Neural Network	155891.592686	192879.831445	108538.194213	122558.799646	0.814404
1	XGBoost	134323.318890	201839.376290	99419.234939	123656.215797	0.862207
2	Random Forest	72654.489660	206073.357227	49716.929169	120254.776469	0.959687
3	Linear Regression	203680.132714	215930.286432	133387.291151	137547.231274	0.683174
4	LASSO Regression	204360.763893	217146.281610	134141.737315	138085.310926	0.681053
5	Decision Tree	165756.350136	233527.646129	111454.509334	137387.480294	0.790171

7.8 Conclusion

In this application, we have seen how to implement machine learning models for regression problems. We have used a dataset of house prices in King County, USA, to predict the price of a house based on a set of features. We have trained several models, including linear regression, LASSO regression, decision trees, random forests, XGBoost, and neural networks. We have used grid search with cross-validation to find the best hyperparameters for the models. We have evaluated the models based on the root mean squared error, mean absolute error, and R-squared. With a bit more careful hyperparameter tuning, we could likely improve the performance of the models even further and the ranking of the models might change.

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.

Bank for International Settlements. 2021. "Machine learning applications in central banking." IFC Bulletin 57. https://www.bis.org/ifc/publ/ifcb57.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.

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.

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.

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.