FITPED | AI

Machine Learning

Jozef Kapusta
Ján Skalka
František Dařena
Kitti Szabó-Nagy
Małgorzata Przybyła-Kasperek
Vladimiras Dolgopolovas
Michal Munk
Lívia Kelebercová

www.fitped.eu

2024

Funded by the European Union

Erasmus+ FITPED-AI
Future IT Professionals Education in Artificial Intelligence
(Project 2021-1-SK01-KA220-HED-000032095)

Machine Learning

Published on

November 2024

Authors

Jozef Kapusta | Teacher.sk, Slovakia

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

František Dařena | Mendel University in Brno, Czech Republic

Kitti Szabó-Nagy | Constantine the Philosopher University in Nitra, Slovakia

Małgorzata Przybyła-Kasperek | University of Silesia in Katowice, Poland

Vladimiras Dolgopolovas | Vilnius University, Lithuania

Michal Munk | Constantine the Philosopher University in Nitra, Slovakia

Lívia Kelebercová | Constantine the Philosopher University in Nitra, Slovakia

Reviewers

Piet Kommers | Helix5, Netherland

Vaida Masiulionytė-Dagienė | Vilnius University, Lithuania

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Erasmus+ FITPED-AI

Future IT Professionals Education in Artificial Intelligence

Project 2021-1-SK01-KA220-HED-000032095

Funded by
the European Union

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or Slovak Academic Association for International Cooperation. Neither the European Union nor the granting authority can be held responsible for them.

ISBN 978-80-558-2228-0

TABLE OF CONTENTS

Introduction

Chapter 1

1.1 Introduction

1.1.1

Machine learning (ML) is a branch of artificial intelligence that allows computers to learn from data and make decisions or predictions without being explicitly programmed. It has become a crucial technology in many industries, from healthcare and finance to entertainment and self-driving cars. At its core, machine learning involves creating algorithms that can analyze and learn patterns from large datasets, enabling systems to improve their performance over time. The goal is for machines to recognize these patterns and use them to predict future outcomes or make decisions on new, unseen data.

Machine learning models can be applied to a wide variety of tasks, including classification, regression, clustering, and anomaly detection. For instance, in a classification task, a machine learning model might be used to classify emails as spam or not spam based on patterns it has learned from a training dataset. In a regression task, the model might predict continuous values, such as stock prices or the temperature for the next day, based on historical data.

To develop effective machine learning models, it's important to understand the underlying data and the methods used for training and evaluation. Data preprocessing is a crucial step to clean and prepare the data for modeling, and model evaluation metrics such as accuracy, precision, and recall are used to assess how well the model performs. Understanding these concepts and techniques will lay the foundation for exploring more advanced topics in machine learning.

1.1.2

Which of the following is true about supervised learning in machine learning?

- It involves training models on labeled data with known outputs.
- It involves training models on data with no labeled outputs.
- It requires the use of reinforcement signals like rewards and penalties.
- It is used exclusively for predicting continuous values.

1.1.3

Can a machine learn new knowledge?

One of the core debates about artificial intelligence (AI) revolves around whether machines can truly learn new knowledge and adapt to unfamiliar situations. Critics argue that machines lack genuine intelligence because they operate within the confines of predefined rules and instructions. If machines only perform tasks as prescribed, can they be considered intelligent? This skepticism highlights the importance of machine learning (ML), a subset of AI that equips systems with the ability to learn and improve from experience.

Machine learning can be broadly defined as a process by which a system enhances its performance over time based on experience. In 1983, Herbert A. Simon described learning as **"changes in a system that are adaptive in the sense that they allow the system to accomplish the same task or tasks from the same class of tasks a second time more efficiently and effectively."** This adaptability enables machines to evolve beyond their initial programming, becoming more competent at solving problems and handling tasks over time.

Machine learning involves two key components: **skill refinement** and **knowledge acquisition**. Skill refinement refers to a system's ability to improve its performance in solving tasks by simply performing them repeatedly. For instance, a robot might optimize its navigation through a maze after several attempts, gradually learning to minimize errors and maximize efficiency. This process mirrors how humans refine skills like playing a musical instrument or driving a car through practice.

Knowledge acquisition is the second vital element, wherein machines gather knowledge through experience. This means that systems can analyze patterns, recognize trends, and use the information they acquire to adapt their behavior or make predictions. For example, a machine learning model trained on historical weather data can "learn" to predict future weather conditions. This ability to adapt and learn enables machines to tackle a wide ran.

1.1.4

Which of the following statements are true about machine learning?

- It enables systems to improve their performance based on experience.
- Skill refinement involves improving task-solving by repetition.
- Machines acquire knowledge solely through explicit programming.
- Machine learning eliminates the need for data analysis.

1.1.5

Machine learning algorithms can solve the following groups of problems:

- A group of problems for which **there are no human experts**. For example, in modern manufacturing facilities, it is necessary to predict machine failures before they actually occur based on sensor analysis. Because the machines are new, there is no expert to give the programmer in question all the knowledge needed to create a computer system. A system built on machine learning can study the recorded data and infer prediction rules for subsequent machine failures.

- A group of problems **where experts exist, but are unable to explain their expertise**. This is the case for many recognition tasks, such as speech recognition, handwriting recognition, and natural language understanding. In fact, all humans demonstrate expert ability to solve these tasks, but none of them are able to describe in detail the steps they apply in solving them. Fortunately, humans can provide machines with examples of inputs and correct outputs for these tasks, so machine learning algorithms can learn to map inputs to correct outputs.
- A group of problems **where circumstances change rapidly**. In finance, for example, people would like to predict future stock market developments, consumer purchases or currency exchange rates. This data changes quite rapidly, so even if a programmer could create a good prediction program, it would have to be rewritten frequently. A learning program can relieve the programmer from constant modification and debugging by creating a set of prediction rules learned by learning.
- A group of applications **that must be configured for each user separately**. Consider, for example, a program for filtering unwanted e-mail. Each user will need different filters. It is unreasonable to expect each user to define their own rules, and it is also unfeasible to have a software engineer available to each user to update their rules. A system using machine learning is able to learn which emails a user rejects and thus maintain filtering rules automatically.

1.1.6

Which of the following is an example of a problem where machine learning is useful because experts cannot fully explain their expertise?

- Speech recognition
- Predicting future stock market developments
- Filtering unwanted emails for individual users
- Predicting machine failures in new manufacturing facilities

1.1.7

Which of the following are examples of problems where machine learning is particularly beneficial?

- Predicting future stock market developments
- Filtering spam emails for individual users
- Solving simple arithmetic problems

- Predicting machine failures in new manufacturing facilities

1.2 What is machine learning?

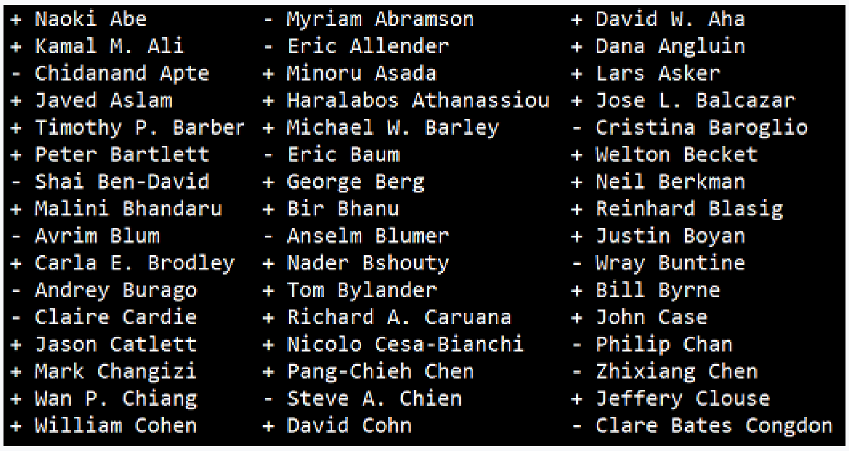
1.2.1

What is learning?

To illustrate where the main advantages, but also the issues of machine learning lie, we give an example, the so-called The Badges Game. The example was invented by Haym Hirsh, who at a machine learning conference in 1994 assigned a "+" or "-" to each registered participant. The label was assigned by some unknown function known only to the creator of the example. The designation depended only on the first and last name of the participant.

The task for the participants was to identify the unknown function used to generate the +/- sign.

The list looked something like this:



```
+ Naoki Abe          - Myriam Abramson       + David W. Aha
+ Kamal M. Ali       - Eric Allender         + Dana Angluin
- Chidanand Apte     + Minoru Asada          + Lars Asker
+ Javed Aslam        + Haralabos Athanassiou + Jose L. Balcazar
+ Timothy P. Barber  + Michael W. Barley     - Cristina Baroglio
+ Peter Bartlett     - Eric Baum             + Welton Becket
- Shai Ben-David     + George Berg           + Neil Berkman
+ Malini Bhandaru    + Bir Bhanu             + Reinhard Blasig
- Avrim Blum         - Anselm Blumer         + Justin Boyan
+ Carla E. Brodley   + Nader Bshouty         - Wray Buntine
- Andrey Burago      + Tom Bylander          + Bill Byrne
- Claire Cardie      + Richard A. Caruana    + John Case
+ Jason Catlett      + Nicolo Cesa-Bianchi   - Philip Chan
+ Mark Changizi      + Pang-Chieh Chen       - Zhixiang Chen
+ Wan P. Chiang      - Steve A. Chien        + Jeffery Clouse
+ William Cohen      + David Cohn            - Clare Bates Congdon
```

For a full list, you can see:
https://www.seas.upenn.edu/~cis5190/fall2018/assets/lectures/lecture-0/game.html

We do not need to build a machine learning model to solve this problem. However, it is important to think about how we would formalize this problem as a learning problem and what are the difficulties that arise in doing so.

When solving, it is important to remember that only the first and last names of the participants will not be enough even for a machine learning algorithm. New variables need to be generated from those names, e.g., the length of the full name, the length, i.e., the number of characters of the first name and the last name, the first character of the first name and its numbered code, the last character of the last name and its numbered code, the number of consonants, the number of vowels, and so on.

If we subsequently have the above mentioned variables - properties/attributes - calculated, we can deploy a machine learning algorithm that learns to add +/- tags for each name.

It is important to note that we do not need to know the exact function that Mr. Hirch created. We only try to estimate it, i.e. we try to copy the results of this feature as closely as possible. Mathematicians call this "good estimation" of the behaviour of a function as approximation. The algorithm of machine learning will therefore seek to approximate the function of Mr. Hirsh.

1.2.2

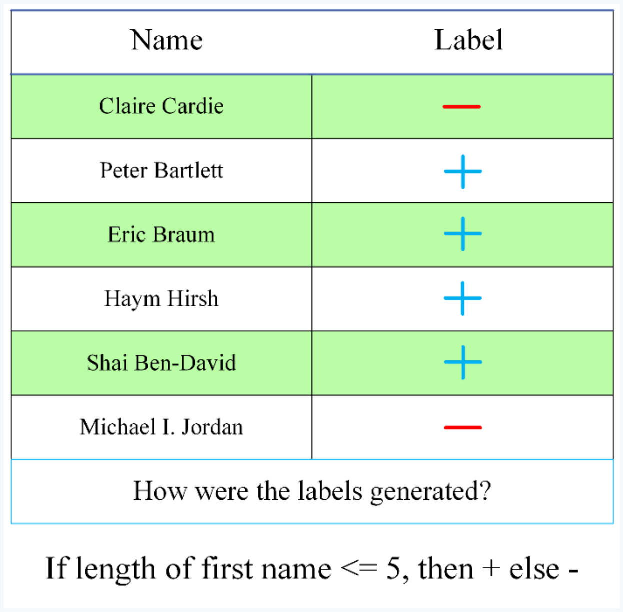
What is a function for approximating values?

- It is the replacement of given values with appropriate close numbers based on a function that is not entirely accurate, but it is still good to be usable.
- It is the replacement of given values with appropriate close numbers based on a function that exactly corresponds to the substituted values.

1.2.3

In the previous example, we stated that machine learning only tries to **approximate the real function**. We still don't know the original Hirsh function. For some names from an existing dataset, it is possible to approximate the +/- marks using rules such as:

- if the length (number of characters) of the name is less than or equal to 5 yes + otherwise -

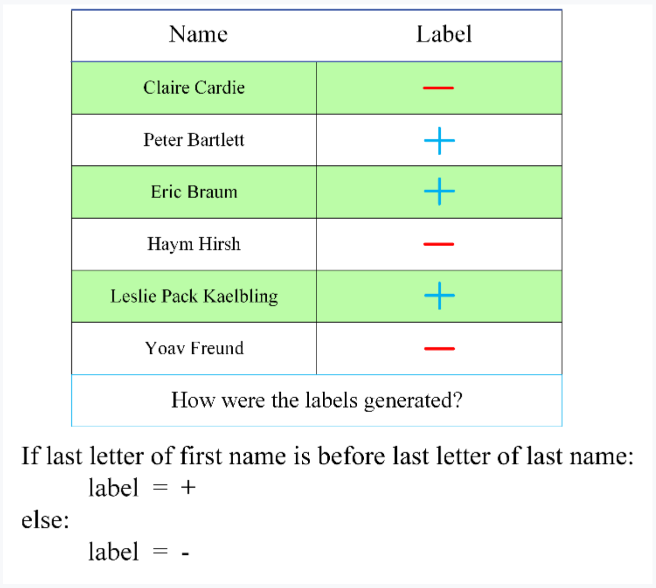


Name	Label
Claire Cardie	−
Peter Bartlett	+
Eric Braum	+
Haym Hirsh	+
Shai Ben-David	+
Michael I. Jordan	−
How were the labels generated?	

If length of first name <= 5, then + else -

or

- if the numerical code of the last letter of the name is smaller than the numerical code of the last letter of the surname yes + otherwise -



Name	Label
Claire Cardie	−
Peter Bartlett	+
Eric Braum	+
Haym Hirsh	−
Leslie Pack Kaelbling	+
Yoav Freund	−
How were the labels generated?	

```
If last letter of first name is before last letter of last name:
        label = +
else:
        label = -
```

1.2.4

What was the main objective of participants in "The Badges Game" introduced by Haym Hirsh?

- To identify the unknown function that assigned "+" or "-" to names
- To classify participants based on their field of research
- To guess the preferences of conference participants

- To evaluate the accuracy of machine learning models for sentiment analysis

1.2.5

The purpose of **The Badges Game** example was not to solve the problem with the unknown function for adding +/- badges to conference participants. With an example, we wanted to show that machine learning algorithms only try to approximate the real function. At the same time, we wanted to show that working to solve a problem using machine learning is not about "headlessly" deploying a randomly selected algorithm and expecting excellent results. Most of the work consists in preparing the dataset, adding new features, i.e. attributes, in carefully selecting a machine learning algorithm, evaluating the algorithm, and understanding the results.

For the application of machine learning, it is necessary to implement the following steps in most cases:

- Data preprocessing
- Extracting symptoms / features
- Creating a model
- Making a prediction
- Model testing and modification

Typical questions when applying machine learning are:

- How to represent input data?
- What deep background knowledge do we need?
- How does learning take place?

1.2.6

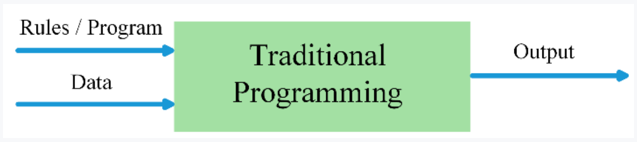
What does "The Badges Game" illustrate about machine learning?

- Machine learning can approximate unknown functions.
- The exact function used in a dataset may remain unknown.
- Machine learning guarantees discovering the exact underlying rules.
- Machine learning involves handling complex real-world data.

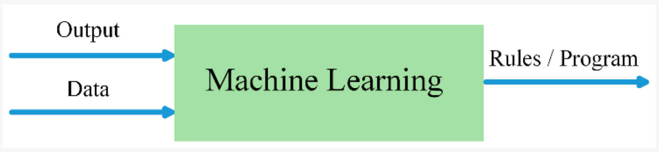
📖 1.2.7

So what is the difference between traditional programming and machine learning?

In traditional programming, we know the problem, we know the rules to solve it, and if we apply these rules to the input data, we get the result.



In machine learning, we know the input data and we also know what the result should be. We are looking for a model, i.e. for example rules, which can generally calculate the result from the input data.



In the following text, we present examples of how machine learning can improve a task based on experience (training data) with respect to a measure (metric) of performance.

- **Task:** Checkers game
- **Performance metric:** Percentage of games won against any opponent
- **Data:** Playing practice games against each other

-- -----------------------------

- **Task:** Recognizing handwritten words
- **Performance metric:** Percentage of correctly classified words
- **Data:** Database of annotated images of handwritten words

-- -----------------------------

- **Task:** Categorizing email messages as spam or ham.
- **Performance metric:** Percentage of email messages correctly classified.
- **Data:** A database of emails that have been manually annotated

-- -----------------------------

- **Task:** Driving on highways using sensors
- **Performance metric:** Average distance traveled before human-judged error
- **Data:** A sequence of images and steering commands recorded while observing a human driver.

1.2.8

Which statements correctly describe the differences between traditional programming and machine learning?

- In traditional programming, the rules to solve the problem are predefined.
- Machine learning uses input data and known results to find a model.
- Traditional programming relies on discovering rules from the data.
- Machine learning applies predefined rules to calculate results.

1.2.9

Complete the appropriate procedure and source data for each task.

Task: Predicting house prices

Performance metric: _____

Data: _____

Task: Classifying medical images to detect tumors

Performance metric: _____

Data: _____

Task: Recommending movies based on user preferences

Performance metric: _____

Data: _____

- A set of medical images annotated with labels indicating the presence or absence of tumors
- Percentage of recommended movies that the user watches
- A dataset of historical house prices, including features like location, size, and number of rooms
- A database of user ratings for movies and movie genres
- Mean absolute error between predicted and actual prices
- Accuracy of correctly identifying tumors

1.2.10

If we want to use machine learning to categorize which news belongs to fake news, the so-called fake news, what input (data) and output (output) data do we need to build such a classifier?

- A database of messages, along with information from manual annotators (people judging the messages) about which message is fake and which is genuine.
- Rules written by the state government by which institutions determine which message is fake.
- A list of people compiled by the state government who spread false information.
- Database of politicians who lie.

1.3 Basic principles

1.3.1

One of the most basic yet effective ways for a machine to acquire knowledge is by memorizing data on how to perform tasks. This method enables the system to apply this information to accomplish similar tasks more efficiently in the future. In machine learning and artificial intelligence, this is often referred to as "swotting." By retaining useful information, the system can perform tasks with improved accuracy and effectiveness over time, akin to humans remembering steps or strategies for solving problems.

A classic example of swotting can be found in Samuel's checkers-playing program developed in 1963. Samuel utilized a technique called the **mini-max search**, which explores possible moves in a game tree to select the optimal play. However, due to computational limitations, only a few levels of the game space could be explored in each turn. To make the search process more efficient, a **static evaluation function** was used to evaluate the quality of each move based on its position in the game. This evaluation was then remembered by the system for future reference.

The power of swotting in Samuel's checkers program came into play when the system encountered previously explored positions during future searches. When the search encountered a node that had been previously evaluated, the system could immediately use the memorized evaluation rather than recalculating the evaluation from scratch. This approach saved computational resources and increased the overall efficiency of the search process. The ability to recall past evaluations allowed the program to simulate a deeper search than it would have been able to achieve by recalculating everything from the beginning. In this way, the system "learned" to improve its decision-making process based on past experiences.

The swotting method exemplifies how memory and the ability to recall information can significantly enhance problem-solving in AI systems. By remembering prior evaluations or strategies, systems can avoid redundant calculations and focus on refining their performance.

1.3.2

Which of the following statements best describes the swotting method?

- It involves storing previous evaluations to enhance future decision-making.
- It relies solely on random moves to perform tasks efficiently.
- It uses a fixed set of rules without considering past experiences.
- It eliminates the need for static evaluation functions in machine learning.

1.3.3

In machine learning, the process of evaluation is a crucial component of decision-making algorithms, especially in games like checkers or in pattern recognition tasks. One effective way to evaluate different possible outcomes is through the **evaluation function**, which takes into account several factors relevant to the task.

In checkers, for example, the evaluation function is often constructed by combining information from multiple sources. The programs take into account several factors, e.g., the advantage in the number of stones, or the mobility of the stones when playing checkers.

Samuel, in his checkers-playing program, used an evaluation function in the form of a **polynomial** that considered different features of the game state. Each feature—such as the number of pieces or their mobility—was weighted and combined to produce a final score for the board's position. By adjusting these weights, the system could refine its ability to evaluate different positions more accurately. The strength of this approach is that it simplifies a complex task into measurable components, which can be adjusted based on experience.

in and +in$_{1122}$ +... and+in and in$_{nn}$

Similarly, in **pattern recognition** programs, which aim to classify input data into categories, the process of evaluating the importance of different features becomes essential. When building these types of systems, it is often challenging to know the exact weight to assign to each feature in advance. In such cases, an initial estimate of the weights can be used, and over time, the system can adjust these weights based on the feedback it receives. Features that are more strongly correlated with a successful outcome will increase in weight, while features that do not contribute as effectively will have their weights reduced or ignored altogether.

This process of adjusting weights and parameters over time is known as **learning by adapting parameters**. It's a fundamental concept in machine learning, as it allows the system to evolve and improve its decision-making ability based on past experiences. Whether evaluating board positions in a game or recognizing patterns in data, learning by adapting parameters enables the system to better handle complex tasks by refining its internal models based on new data.

1.3.4

Which of the following are examples of learning by adapting parameters?

- Adjusting weights based on experience to improve the accuracy of classification tasks.
- Combining multiple features into a single evaluation function for decision-making.
- Using a fixed set of parameters without any updates or adjustments.
- Allowing a machine to memorize tasks without altering internal parameters.

1.3.5

In machine learning, one important aspect of improving a system’s performance is adjusting the weights of different features or attributes. This process is known as **learning by adapting parameters**, and it is a key concept in many algorithms, including those used in game-playing programs and pattern recognition systems.

To design a method for adapting parameters, three things are crucial:

1. **Which weight to adjust** deciding which features or attributes should have their weights increased or decreased is the first step in improving the system.
2. **When to change the weight** - the system needs to determine the appropriate time to update the weights, which often depends on feedback from the results of predictions or decisions made.
3. **How much to change the weight** means amount by which the weight should be changed is often determined by how accurate or inaccurate the prediction was.

For example, when classifying patterns, the system will receive information about whether the classification was correct or incorrect. If the classification was correct, the system will increase the weights of the features that helped make that correct decision. On the other hand, if the classification was incorrect, the system will decrease the weights of the features that led to the wrong decision.

This is more complicated with gaming programs. At most, the program gets information at the end about who won. However, many moves contributed to the final result, of which several could have been erroneous. For example, Samuel took an approach where the evaluation function generates its own feedback. It was based on the consideration that the sequences of steps that lead to better positions can be considered good. The weights of the attributes that recommended them will increase.

Samuel's program was also taught by playing against itself, i.e. one copy of it played with unchanging weights and the other copy had the weights changed. At the end of the game, the attributes of the program that won were taken. The process of learning by adapting parameters is limited in nature, since it does not make any use of knowledge about the structure of the problem.

1.3.6

Which of the following is a key aspect of learning by adapting parameters?

- The system adjusts weights based on feedback from the final outcome of a game or task.
- The system changes its weights only after receiving detailed information about every individual action.
- The system uses fixed weights that are never updated.
- The system doesn't use any feedback to improve its performance.

1.3.7

Setting weights

In machine learning, when you're training a model to classify data, it is essential to assign **weights** to different features that will influence the final prediction. These weights help the model determine the importance of each feature in making accurate predictions. Let's consider a **classification problem** where we need to classify emails as either **spam** or **not spam**. We will use a set of features such as the frequency of certain words, the sender's email address, and the presence of links in the email.

For this example, let’s assume we have the following features:

1. Frequency of the word "free"
2. Number of links in the email
3. Sender is from a known domain (e.g., gmail.com)
4. Email length

We also have a training dataset where emails have already been labeled as **spam** or **not spam**.

1. Initializing weights

Initially, each feature will have an equal weight. This means that the model treats all features as equally important when making a prediction. In this case, each feature might have a weight of **0.25** (since there are 4 features). This is a starting point, and the weights will be updated as the model learns from the data.

2. Evaluating a sample email

Let’s now take a sample email and see how the system would apply the weights. For example, the email might contain:

- 3 occurrences of the word "free"
- 5 links
- The sender is from a known domain (gmail.com)
- The email length is 150 words

Let’s calculate the initial prediction score using the weights:

- Word "free" - frequency = 3, weight = 0.25 → score = 3 * 0.25 = 0.75
- Links - number of links = 5, weight = 0.25 → score = 5 * 0.25 = 1.25
- Sender from a known domain - yes = 1, weight = 0.25 → score = 1 * 0.25 = 0.25
- Email length - length = 150 words, weight = 0.25 → score = 150 * 0.25 = 37.5

3. Summing the scores

The total score for this email would be the sum of all individual scores:

- Total score = 0.75 + 1.25 + 0.25 + 37.5 = **39.75**

4. Adjusting the weights

After making an initial prediction based on the weights, the system compares the prediction to the actual label (whether the email was actually spam or not). If the model predicted the email to be **spam** correctly, it will increase the weights of the features that contributed to the correct prediction. For example:

- **Word "free"** - the word "free" is a strong indicator of spam, so we might increase its weight from **0.25 to 0.5**.
- **Links** - since emails with more links are likely to be spam, we increase the weight of the **links feature** from **0.25 to 0.4**.

- **Sender from a known domain** - if the sender is from a trusted domain, it might indicate that the email is not spam, so we reduce the weight of this feature from **0.25 to 0.2**.
- **Email length** - if longer emails are not strong indicators of spam, we leave the weight for **email length** unchanged at **0.25**.

After adjusting the weights based on the prediction and feedback, the system will be able to make more accurate predictions in the future.

5. Learning over time

As the model receives more feedback on its predictions (correct or incorrect), it will continue to adjust the weights of the features. This process, known as **learning by adapting parameters**, allows the model to gradually improve its accuracy over time. The weights become more reflective of the true importance of each feature in classifying emails as spam or not spam.

1.3.8

Which of the following is true when adapting weights in a machine learning model for classification?

- The model adjusts the weights based on the feedback it receives about its predictions.
- The model adjusts the weights based on the feedback it receives about its predictions.
- The model always keeps the weights the same for all features.
- Features that are more relevant to the prediction will have higher weights.

1.3.9

Let's consider an example from **medical diagnosis**, where we need to classify patients as either **high risk** or **low risk** for a certain disease based on multiple factors.

We will use a set of features such as:

1. Age of the patient
2. Blood pressure level
3. Cholesterol level
4. Exercise frequency

We also have a training dataset with labeled instances indicating whether a patient is **high risk** or **low risk**.

1. Initializing weights

Initially, we assign **equal weights** to all features. For simplicity, let's assume that we have 4 features, and each one has an initial weight of **0.25**. This means that the model considers all features as equally important when making its decision.

2. Evaluating a sample patient

Let's now take a sample patient with the following attributes:

- Age: 55 years
- Blood pressure: 150/90 (high)
- Cholesterol level: 230 (high)
- Exercise frequency: 1 time per week

We will calculate the prediction score using the weights for each feature:

- Age - 55 years, weight = 0.25 → score = 55 * 0.25 = 13.75
- Blood pressure - 150/90, weight = 0.25 → score = 1 * 0.25 = 0.25 (since high blood pressure is a risk factor)
- Cholesterol level - 230, weight = 0.25 → score = 1 * 0.25 = 0.25 (since high cholesterol is a risk factor)
- Exercise frequency - 1 time per week, weight = 0.25 → score = 1 * 0.25 = 0.25 (exercise is a protective factor)

3. Summing the scores

The total score for this patient would be:

- Total score = 13.75 + 0.25 + 0.25 + 0.25 = **14.5**

4. Adjusting the weights

After receiving the actual diagnosis (high risk), the system will adjust the weights of the features based on how well they predicted the result. Let's say the system receives feedback that the model made a correct prediction. The weights will be updated to reflect the importance of certain features:

- **Age** - as older patients tend to be at a higher risk, the weight of **age** might be increased from **0.25 to 0.4**.
- **Blood pressure** - since high blood pressure is a strong risk factor, the weight for **blood pressure** could be increased from **0.25 to 0.35**.

- **Cholesterol level** - high cholesterol is also a risk factor, so the weight for **cholesterol** might remain the same, or increase slightly from **0.25 to 0.3**.
- **Exercise frequency** - exercise frequency is a protective factor, so the weight for **exercise frequency** might be decreased from **0.25 to 0.2**.

5. Learning over time

As the model continues to process more patient data and receives feedback on its predictions, the weights will gradually become more reflective of which features are most important for predicting risk. For example, if the model continues to predict correctly that high cholesterol and high blood pressure are the most important indicators of risk, their weights will continue to increase, while less relevant features (such as exercise frequency) might have their weights reduced.

1.3.10

Which of the following steps occurs when the system adjusts the weights in a medical diagnosis model?

- The weights of the features that helped make correct predictions are increased.
- The weights of all features are decreased over time.
- The system ignores the features that led to incorrect predictions.
- The weights of features that were incorrect in making predictions are reduced.

Machine Learning Types

Chapter 2

2.1 Machine learning types

📖 2.1.1

Machine learning is a method of teaching machines to perform tasks by learning from data instead of being explicitly programmed. One of the fundamental aspects of machine learning is the **type of feedback provided during the learning process**. Feedback determines how the model learns, adapts, and improves.

There are three main types of machine learning: supervised learning, reinforcement learning, and unsupervised learning.

- In **supervised learning**, the model is trained on labeled data, where both the inputs and their corresponding outputs are known. This allows the model to learn by example. For instance, when teaching a system to recognize handwritten digits, images of numbers are paired with their correct labels (e.g., "5" or "9"). Supervised learning is particularly effective for tasks like classification (assigning categories) or regression (predicting values), where the correct output is always provided during training.
- In contrast, **reinforcement learning** operates by trial and error. Instead of receiving exact answers, the system learns through **rewards and penalties** based on its actions. A classic example is a robot navigating a maze. The robot might get rewarded for moving closer to the exit and penalized for hitting walls. Over time, it learns the best strategy to achieve its goal. This type of learning is commonly used in gaming, robotics, and dynamic environments where actions influence future outcomes.
- Finally, **unsupervised learning** (learning without a teacher) involves no labeled data or explicit guidance. Instead, the model explores the data and identifies hidden patterns or structures on its own. For example, in customer segmentation, an unsupervised model can group customers based on shared characteristics without knowing in advance what those groups represent. This approach is often used for clustering, anomaly detection, and exploratory data analysis.

📝 2.1.2

Assign the correct name of the learning type to the characteristics of learning.

the algorithm receives information about the evaluation of the action, but not about what the correct action should have been - _____

the algorithm has immediate availability of sensations about both inputs and outputs - _____

the algorithm does not receive any information about what the correct actions should be - _____

- Unsupervised learning
- Reinforcement learning
- Supervised learning

2.1.3

Supervised learning

Supervised learning is a fundamental approach to machine learning where models are trained on labeled datasets. Each input data point is paired with a corresponding output, allowing the system to learn by example. This process involves teaching the model to map inputs (e.g., an image) to known outputs (e.g., the digit represented in the image). For instance, a supervised learning system trained to recognize handwritten digits might be given thousands of labeled examples, such as an image of "5" paired with the label "5."

This type of learning is particularly effective for tasks like **classification**, where data is categorized into predefined groups (e.g., spam vs. non-spam emails), and **regression**, where the model predicts continuous values (e.g., predicting housing prices). By analyzing the labeled data, supervised learning algorithms learn to generalize, enabling them to make accurate predictions on new, unseen data.

Supervised learning is widely used in everyday applications, from speech recognition to fraud detection, and is known for its reliability when sufficient labeled data is available. However, the accuracy of these models heavily depends on the quality and size of the labeled dataset, as poor or insufficient data can lead to underperforming models.

2.1.4

What type of data does supervised learning use?

- Labeled data with inputs and outputs.
- Data without labels or guidance.
- Data with rewards and penalties.
- Randomly generated synthetic data.

2.1.5

Reinforcement learning

Reinforcement learning (RL) is a unique approach to machine learning that involves learning through rewards and penalties. Unlike supervised learning, where correct answers are provided for every input, RL systems learn by interacting with an environment and receiving feedback based on their actions. This feedback could be positive (reward) for desirable actions or negative (penalty) for undesirable ones.

For example, consider a robot navigating a maze. The robot might receive a reward when it moves closer to the exit and a penalty for hitting walls. Over time, the robot learns a strategy, or "policy," to maximize its rewards and efficiently navigate the maze. This trial-and-error approach makes reinforcement learning ideal for tasks where actions have a long-term impact on outcomes, such as gaming, robotics, and autonomous vehicles.

Reinforcement learning excels in dynamic and complex environments where clear rules for success are not predefined. However, designing effective reward systems and ensuring efficient exploration of the environment can be challenging. Despite these challenges, RL has led to remarkable achievements, such as AI systems defeating human champions in games like chess and Go.

2.1.6

Which of the following are characteristics of reinforcement learning?

- Rewards and penalties guide learning.
- Actions influence future outcomes.
- Models require labeled data.
- Explicit guidance is provided for every action.

2.1.7

Unsupervised learning

Unsupervised learning stands apart from supervised and reinforcement learning by operating on **unlabeled data**. Here, the model is not given explicit guidance or labeled examples. Instead, it explores the dataset to discover hidden patterns, structures, or relationships on its own. This makes unsupervised learning particularly valuable for exploratory data analysis and tasks where labeled data is unavailable or impractical to obtain.

A common application of unsupervised learning is **clustering**, where data points are grouped into clusters based on shared characteristics. For example, in customer segmentation, an unsupervised model might group customers based on purchasing habits, identifying potential target markets. Another example is **anomaly detection**, where the model flags data points that deviate significantly from the norm, such as fraudulent transactions.

While unsupervised learning is powerful for uncovering insights, interpreting its results often requires domain expertise, as the model does not assign labels or meanings to the patterns it identifies. Nevertheless, its ability to analyze vast amounts of unlabeled data makes it indispensable in fields like marketing, biology, and network security.

2.1.8

What is the primary characteristic of unsupervised learning?

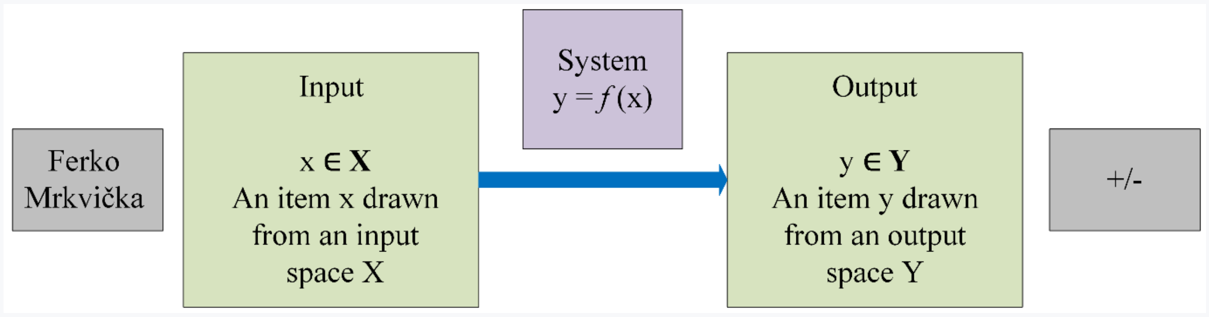
- It identifies patterns in unlabeled data.
- It uses labeled data to learn by example.
- It relies on rewards and penalties for guidance.
- It requires a predefined hypothesis space.

2.1.9

Learning with a teacher

Machine learning with a teacher, involves teaching a system to apply a function **f(x)** to an input **x**, resulting in an output **y=f(x)**. In this type of learning, the system is provided with examples of inputs and their corresponding outputs, allowing it to learn the relationship between them. The goal is for the system to generalize this knowledge and apply it to new, unseen data.

The necessity of supervised learning arises when we cannot explicitly define the function **f(x)** ourselves. For instance, we may not have the formula to solve a problem directly, but we can collect data containing inputs and the desired outputs. This data acts as a guide for the system to infer the underlying function. An example of such a scenario is Mr. Hirsh's problem: the input **x** is the name of a conference participant, and the output **f(x)** is a "+" or "-" sign, indicating some binary classification.


System
y = f(x)
Ferko
Mrkvička
Input
x ∈ X
An item x drawn from an input space X
Output
y ∈ Y
An item y drawn from an output space Y
+/-

When learning with a teacher, the algorithm doesn't guess randomly. Instead, it searches for the **best approximation of the true function** based on the given data. To do this, the algorithm explores a predefined range of possible functions, known as the **hypothesis space**. This space defines the set of potential functions the system evaluates to find the one that most accurately maps inputs to outputs.

This process is at the heart of many practical applications of machine learning. By training a model with supervised learning, we empower it to handle tasks such as predicting stock prices, identifying objects in images, or classifying email as spam or not. The quality of results depends heavily on the diversity of examples in the training data and the algorithm's ability to navigate the hypothesis space efficiently.

2.1.10

What is the hypothesis space in supervised learning?

- The set of potential functions considered by the algorithm.
- The space where training data is stored.
- The physical location of the machine learning system.
- The formula for the correct function f(x).

2.1.11

When learning with a teacher, the learning algorithm is provided with the correct output for a given input. Each ordered pair **(x,f(x))** represents an "example," where **x** is the input, and **f(x)** is the corresponding output of the function for that input. Given a collection of such examples of the function fff, the goal of the algorithm is to return a function hhh, called the **hypothesis**, which serves as an estimate or approximation of **f**.

The number of output classes (for classification problems) is predefined by the user. The systems do not rely on any additional domain-specific information beyond the provided training examples.

Many problems that at first glance do not look like classification problems can be transformed into classification problems. In the following examples, we show what constitutes the input **x** and the output **f(x)**, i.e., what constitutes an example for each classification task:

Diagnosis of the disease

- x: Patient characteristics (symptoms, laboratory tests)
- f(x): Disease (or maybe: recommended treatment)

Part-of-speech tagging

- x: English/Slovak sentence
- f(x): Parts of speech in a sentence

Face recognition

- x: Bitmap image of a person's face
- f(x): Name and surname of the person (or maybe: property)

Automatic control

- x: Bitmap image of the road surface in front of the car
- f(x): Degrees of steering wheel rotation

2.1.12

Choose which claims apply to learning with a teacher (supervised learning)

- For classification tasks, the number of classes is fixed and is determined by the user.
- The method does not use any other domain-specific information except for training examples.
- To evaluate the success of the method, the predicted values are compared with the actual values of the test set.
- Not suitable for regression types of tasks.

2.2 Model evaluation

📖 2.2.1

Machine learning is not just about creating algorithms - it is a structured process aimed at solving real-world problems through data. This process involves several key steps that ensure the model is well-suited to the domain, utilizes clean and relevant data, and produces meaningful results. Let’s break down these steps:

1. **Understanding the domain and objectives** - before diving into data, it is crucial to understand the problem domain and establish clear objectives. This involves defining what the system is supposed to achieve, identifying constraints, and leveraging prior knowledge about the field. For example, in developing a machine learning model for predicting customer churn, one must understand customer behaviors, business metrics, and why customers may leave.
2. **Data integration, selection, and pre-processing** - high-quality data is the foundation of effective ML models. Data from various sources is integrated, irrelevant or redundant data is removed, and missing or inconsistent data is addressed through cleansing. Pre-processing may also involve normalization, encoding categorical variables, and splitting the data into training and testing sets. These steps prepare the data to be useful and relevant for training the model.
3. **Creating models** involves selecting or designing the appropriate machine learning model to address the problem. Models are trained using the pre-processed data, allowing them to identify patterns or make predictions. For instance, regression models might predict continuous values like house prices, while classification models might identify spam emails. Experimentation with different models and tuning their hyperparameters are also part of this stage.
4. **Interpreting results** - a machine learning model is only as good as its outcomes. Interpreting results means analyzing the model's performance using metrics such as accuracy, precision, recall, or F1-score. It is also essential to evaluate whether the model generalizes well to unseen data and if its predictions make sense in the given domain.
5. **Deployment of models** is the final phase, successful models are deployed into real-world systems. This involves integrating the model with applications, monitoring its performance, and periodically retraining it with new data. For example, a deployed model predicting product recommendations on an e-commerce site continuously updates based on user behavior.

2.2.2

Which step in the machine learning process involves cleaning and preparing the data?

- Data integration and pre-processing
- Understanding the domain
- Creating models
- Deployment of models

2.2.3

List the individual steps/phases of machine learning in the correct order.

- <|span style="color: rgb(0, 0, 0);">Understanding the domain, taking into account prior knowledge and objectives<|/span>
- <|span style="color: rgb(0, 0, 0);">Data integration, selection, cleansing, pre-processing<|/span>
- <|span style="color: black;">Interpretation of results<|/span>
- <|span style="color: rgb(0, 0, 0);">Creating models<|/span>
- <|span style="color: black;">Deployment of discovered knowledge/models<|/span>

2.2.4

As a result of machine learning, a model, often represented as a formula or function, that approximates patterns in data. This model is created using historical data, often referred to as "examples," through a machine learning algorithm. Let's explore the process with a bank example and discuss how to evaluate the resulting model.

- **Model creation through historical data** - machine learning uses past data to predict future outcomes or classify new instances. Consider a application that determines whether a customer should be granted a loan based on their characteristics. Historical data, such as customer income, credit history, and whether they repaid past loans, serves as the input dataset. These examples are processed by a machine learning algorithm to generate a model that generalizes the patterns observed in the data.
- **Generalization** - the generated model is not a mere replication of the historical data. Instead, it acts as a generalization of the examples the system was trained on. For instance, the banking model might learn that customers with stable income and a good credit history are more likely to repay loans, applying this pattern to new, unseen customers. This generalization is what allows machine learning to make predictions or decisions in new scenarios.

- **Evaluating the model** - once the model is created, it’s essential to evaluate its effectiveness. Two critical questions arise:

1. Is the model good - this question examines whether the model's predictions align with real-world outcomes. It involves testing the model on data that was not part of its training to see how accurately it performs.
2. How good is it - to measure the model’s performance, metrics like accuracy, precision, recall, or mean squared error are used, depending on the task. For instance, in the banking application, accuracy might represent the percentage of correct loan predictions, while precision could focus on how many approved loans were correctly identified as repayable.

- **Importance of Continuous Evaluation** - models must be regularly assessed to ensure they remain effective as data patterns evolve. A model that performs well today may require updates or retraining in the future to maintain its utility.

2.2.5

Which statements about machine learning models are true?

- Models generalize patterns from historical data.
- Models can predict future outcomes using patterns from examples.
- Machine learning models replicate historical data exactly.
- Models do not need evaluation once created.

2.2.6

To assess the quality of a machine learning model, we simulate its performance on unseen data.

To answer questions about the quality of the model, it is necessary to simulate an estimate of our model as follows:

1. Remove some examples from a dataset
2. Create a model on remaining examples
3. Predict (estimate) deleted examples

This means that we need provide the machine learning algorithm with only a fraction of the examples we have and we use them to train and build the model. We will call these examples the training examples or the **training set**.

We use the remaining examples to test our model. These are examples that were not used when creating the model. We will call these examples test examples or a **test set**.

This is usually achieved by dividing our dataset into two parts at the begin of process. Whole process ise defined as:

1. **Dividing the dataset** - to begin, we set aside a portion of the dataset for testing purposes. This process involves: removing some examples from the dataset.
2. **Creating model** using the remaining examples (the **training set**) to create and train the model. The training set provides the algorithm with examples it can learn from, enabling it to build the model based on observed patterns.
3. After the model is created, its accuracy and effectiveness are **tested** using the **test set**, which contains examples not included in the training phase. These test examples have known outcomes, making it possible to evaluate the model's predictions against real-world results.

It should be noted that even in the remaining examples, we also have the corresponding outputs for individual inputs. For example, for a banking application, we also have information in the test set whether the client has repaid the loan or not. Therefore, if we bring examples of the test set to the input of our model, we can find out the predicted result by the model and compare it with the real historical result.

This approach is important because by separating training and test data, we ensure that the model’s evaluation is unbiased. If we used the same data for both training and testing, the model might perform well simply because it memorized the examples, not because it generalized well to new, unseen data.

2.2.7

Why is it important to evaluate a machine learning model using a test set?

- To test the model on examples it has not seen before.
- To allow the model to memorize the data.
- To increase the size of the training dataset.
- To improve the speed of model training.

📖 2.2.8

When creating models and interpreting the results, it is necessary to assess the suitability of the model, its correctness and accuracy. In the case of several models, it is necessary to choose a better model.

This brings us to the problem of How to measure accuracy? Which model is better? There is no clear answer to these questions.

For example, if we wanted to create a model for diagnosing a certain disease, and we know that 10 out of 10,000 samples are positive.

We would create multiple models. At first glance, the following statements about the models we have created seem correct:

- A: "the classification model has a success rate of 80%"
- B: "classification model is 400% better than random selection"
- C: "the classification model perfectly captures all positive cases"

However, if we look at these claims in more detail, we find the following potential issues.

- **A: "the classification model has a success rate of 80%".** This model looks very promising. However, if we create a model that labels all samples as negative, then with 10 positives out of 10,000 samples we will achieve a success rate of 99.9%. So if we do nothing and say that all samples are negative, we have a 99.9% success rate.
- **B: "classification model is 400% better than random selection".** Such a model would also look very promising at first glance. However, with 10 positives out of 10 000 samples, one positive out of 1000 samples will be randomly selected. If the algorithm is 400% better, then it can identify 4 positives out of 1000 samples. But is this estimate enough for us?
- **C: "the classification model perfectly captures all positive cases".** Of the above, perhaps the best-looking claim for a potential model, However, if we create a classifier with only one rule that each sample is positive, then we will also perfectly capture all positive cases. But would such a model be necessary?

It is clear from the above statements that we have to assess the suitability of a model from different perspectives and that numerical representations of suitability alone are not always sufficient.

2.2.9

Why is it insufficient to rely solely on the "success rate" when evaluating a classification model?

- It does not account for false positives and false negatives.
- Success rate measures only the speed of classification.
- It requires random sampling of data for validity.
- Success rate is the most reliable metric for classification models.

2.2.10

Which of the following statements highlight potential issues with evaluating classification models?

- A model with a high success rate might ignore positive cases entirely.
- Claims about improvements over random selection may still yield poor results.
- Models are always sufficient if they perfectly capture positive cases.
- High accuracy guarantees the model is suitable for all use cases.

2.3 Practical examples

2.3.1

Project: Introduction

In this object-lesson, we will learn the basics of Python, especially the so-called DataFrame, which precompiles tables of data. The latter is precisely the most commonly used data structure, which consists of labelled axes (rows and columns).

To load the data file into the DataFrame, we will use the pandas library, which we will import using the following command.

```
import pandas as pd
```

In this demonstration, we will work with Titanic passenger data stored at the URL https://priscilla.fitped.eu/data/pandas/titanic.csv.

Use the following command to read and write the data file.

```
data =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/titanic.c
sv')
print(data)
```

Program output:

```
     PassengerId  Survived  Pclass  \
0              1         0       3
1              2         1       1
2              3         1       3
3              4         1       1
4              5         0       3
..           ...       ...     ...
886          887         0       2
887          888         1       1
888          889         0       3
889          890         1       1
890          891         0       3

                                                  Name     Sex
Age  SibSp  \
0                              Braund, Mr. Owen Harris    male
22.0      1
1    Cumings, Mrs. John Bradley (Florence Briggs Th...  female
38.0      1
2                               Heikkinen, Miss. Laina  female
26.0      0
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)  female
35.0      1
4                             Allen, Mr. William Henry    male
35.0      0
..                                                 ...     ...
...    ...
886                              Montvila, Rev. Juozas    male
27.0      0
887                       Graham, Miss. Margaret Edith  female
19.0      0
888           Johnston, Miss. Catherine Helen "Carrie"  female
NaN      1
889                              Behr, Mr. Karl Howell    male
26.0      0
890                                Dooley, Mr. Patrick    male
32.0      0

     Parch            Ticket     Fare Cabin Embarked
```

```
0        0         A/5 21171    7.2500   NaN        S
1        0          PC 17599   71.2833   C85        C
2        0  STON/O2. 3101282    7.9250   NaN        S
3        0            113803   53.1000  C123        S
4        0            373450    8.0500   NaN        S
..     ...               ...       ...   ...      ...
886      0            211536   13.0000   NaN        S
887      0            112053   30.0000   B42        S
888      2        W./C. 6607   23.4500   NaN        S
889      0            111369   30.0000  C148        C
890      0            370376    7.7500   NaN        Q

[891 rows x 12 columns]
```

We will list the contents of only one specific column according to the following command:

```
dataAge = data['Age']
print(dataAge)
```

Program output:

```
0      22.0
1      38.0
2      26.0
3      35.0
4      35.0
       ...
886    27.0
887    19.0
888     NaN
889    26.0
890    32.0
Name: Age, Length: 891, dtype: float64
```

Sometimes it is necessary to find out the i-th record in the data file. This can be viewed using .iloc

```
rec = data.iloc[0]
print(rec)
```

Program output:

```
PassengerId                          1
```

```
Survived                               0
Pclass                                 3
Name             Braund, Mr. Owen Harris
Sex                                 male
Age                                 22.0
SibSp                                  1
Parch                                  0
Ticket                         A/5 21171
Fare                                7.25
Cabin                                NaN
Embarked                               S
Name: 0, dtype: object
```

To find out the data file types, use the following command:

```
typs = data.dtypes
print(typs)
```

Program output:

```
PassengerId      int64
Survived         int64
Pclass           int64
Name            object
Sex             object
Age            float64
SibSp            int64
Parch            int64
Ticket          object
Fare           float64
Cabin           object
Embarked        object
dtype: object
```

- and the length of the data file by using the len() function.

```
length = len(data)
print(length)
```

Program output:

```
891
```

When working in python in machine learning tasks, we often need to know the shape of our data (number of rows and columns).

- Using the following command, we find that our data file contains 891 columns and 12 rows.

```
shap = data.shape
print(shap)
```

Program output:

```
(891, 12)
```

The basic descriptive statistics of the dataset are returned by the describe() function.

```
print(data.describe())
```

Program output:

```
       PassengerId    Survived      Pclass         Age
SibSp  \
count   891.000000  891.000000  891.000000  714.000000
891.000000
mean    446.000000    0.383838    2.308642   29.699118
0.523008
std     257.353842    0.486592    0.836071   14.526497
1.102743
min       1.000000    0.000000    1.000000    0.420000
0.000000
25%     223.500000    0.000000    2.000000   20.125000
0.000000
50%     446.000000    0.000000    3.000000   28.000000
0.000000
75%     668.500000    1.000000    3.000000   38.000000
1.000000
max     891.000000    1.000000    3.000000   80.000000
8.000000

            Parch        Fare
count  891.000000  891.000000
mean     0.381594   32.204208
std      0.806057   49.693429
min      0.000000    0.000000
25%      0.000000    7.910400
```

```
50%       0.000000   14.454200
75%       0.000000   31.000000
max       6.000000  512.329200
```

2.3.2

Which solution correctly displays the Name column?

```
import pandas as pd
data =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/titanic.c
sv')
print(data['Name'])
data =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/titanic.c
sv')
print(data.'Name')
data = titanic
data['Name']()
```

2.3.3

Next, we will explore some of the features of the sklearn library, which is one of the most widely used libraries for machine learning

First, we need to determine our features (features, or x-data) that will be the input to the machine learning model and the end value (target, or y) that will be the output of the machine learning model.

The following code sample loads the Titanic passenger data into a DataFrame data structure and divides it into features and target, where features are all values except the last column Embarked and target is embarked.

```
import pandas as pd
data =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/titanic.c
sv')

X, y = data.iloc[:, :-1], data.iloc[:, [-1]]

print(X)
print(y)
```

Program output:

```
     PassengerId  Survived  Pclass  \
0              1         0       3
1              2         1       1
2              3         1       3
3              4         1       1
4              5         0       3
..           ...       ...     ...
886          887         0       2
887          888         1       1
888          889         0       3
889          890         1       1
890          891         0       3

                                                  Name     Sex
Age  SibSp  \
0                              Braund, Mr. Owen Harris    male
22.0      1
1    Cumings, Mrs. John Bradley (Florence Briggs Th...  female
38.0      1
2                               Heikkinen, Miss. Laina  female
26.0      0
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)  female
35.0      1
4                             Allen, Mr. William Henry    male
35.0      0
..                                                 ...     ...
...    ...
886                              Montvila, Rev. Juozas    male
27.0      0
887                       Graham, Miss. Margaret Edith  female
19.0      0
888           Johnston, Miss. Catherine Helen "Carrie"  female
NaN      1
889                              Behr, Mr. Karl Howell    male
26.0      0
890                                Dooley, Mr. Patrick    male
32.0      0

     Parch            Ticket     Fare Cabin
0        0         A/5 21171   7.2500   NaN
1        0          PC 17599  71.2833   C85
2        0  STON/O2. 3101282   7.9250   NaN
3        0            113803  53.1000  C123
```

```
4        0            373450   8.0500   NaN
..     ...               ...      ...   ...
886      0            211536  13.0000   NaN
887      0            112053  30.0000   B42
888      2        W./C. 6607  23.4500   NaN
889      0            111369  30.0000  C148
890      0            370376   7.7500   NaN

[891 rows x 11 columns]
    Embarked
0          S
1          C
2          S
3          S
4          S
..       ...
886        S
887        S
888        S
889        C
890        Q

[891 rows x 1 columns]
```

When solving machine learning tasks, we divide the data into training and testing data.

Using the training data, we train the machine learning model and then validate it on the test data.

The sklearn library provides a function train_test_split that splits the data into two variables, where the first variable (usually referred to as **X_train**) contains the data that will be used later for training, and the second variable (usually referred to as **X_test**) contains the data that will be used to validate the model.

We further divided our features and target data into training and test data in the ratio of 80:20 using the following commands.

```
from sklearn.model_selection import train_test_split

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

2.3.4

We have data with 4 features columns. Split the data into training and test data in a 70:30 ratio. Find the shape in the X_test variable. What is the correct shape of the X_test variable if the dataset from the given URL has 1000 rows?

- (300, 4)
- (1000, 3)
- (700, 4)
- (700, 4)
- (300, 3)

Classification

Chapter 3

3.1 Classification

📖 3.1.1

Classification is a fundamental concept in machine learning, used to predict categorical labels for data points. To classify, we need a classifier—an algorithm that can categorize the data into predefined classes. For example, in fraud management, we might have a dataset in a .csv file with two columns: one containing the message and the other containing a binary label (0 for non-hoax and 1 for hoax). The task of classification involves determining which messages are hoaxes and which are not based on patterns in the data. This process is often referred to as "training a model" and is one of the primary tasks in supervised learning.

The classification process typically involves two main stages: training and testing. In the training stage, a portion of the dataset (e.g., 75%) is used to teach the model. This is known as the training set. During training, the model examines the features of the messages (e.g., words or phrases) and learns patterns that indicate whether a message is a hoax or not. Once the model is trained, it proceeds to the testing phase, where it uses the remaining 25% of the dataset (the test set) to evaluate its performance by predicting whether the messages are hoaxes or not. The model then outputs a prediction, which is compared to the actual label (0 or 1).

In practice, the accuracy of the model is evaluated by comparing the predicted labels to the actual ones in the test set. A good classifier will be able to generalize well, meaning it can predict accurately on new, unseen data. By training and testing the model on different portions of the data, we ensure that the classifier is robust and not overfitting to a particular subset of the data.

📝 3.1.2

What is the main task of classification?

- Predict categorical labels
- Predict numerical values
- Summarize data
- Group similar data points

📝 3.1.3

Which stage of classification uses a portion of data to evaluate the model's performance?

- Training

- Testing
- Testing and training
- None of the above

📖 3.1.4

Binary classification

Binary classification is one of the simplest and most widely used types of classification tasks. It involves classifying data into one of two categories. A common example of binary classification is email spam detection, where the input data (features) might include properties such as the subject line, sender, or content of the email. The model’s task is to classify the email as either "spam" (represented by 1) or "not spam" (represented by 0).

To train a binary classifier, we use a labeled dataset where each message is already tagged as either spam or not. During the training phase, the classifier learns to distinguish between the characteristics of spam and non-spam emails by examining patterns in the data. For instance, it may learn that certain words like "free" or "offer" are commonly found in spam messages. After the training process, the model is tested on a separate portion of the data to evaluate its performance. The model then predicts whether new, unseen emails are spam or not based on the learned patterns.

The simplicity of binary classification makes it a popular choice for many real-world applications. It can be used for tasks like fraud detection, sentiment analysis, and medical diagnosis. However, the model's success depends on the quality and variety of the data it is trained on. If the data is not representative of the real-world scenarios the model will face, it may struggle to make accurate predictions.

📝 3.1.5

What does a binary classification model predict?

- Two classes
- More than two classes
- Continuous values
- Textual outputs

3.1.6

Which of the following is an example of binary classification?

- Spam detection in emails
- Email categorization by topic
- Image classification with multiple classes
- Predicting stock prices

3.1.7

Multiclass classification

Multiclass classification is used when the target variable has more than two classes. Unlike binary classification, which predicts between two categories, multiclass classification involves categorizing data into multiple possible classes. A common example of multiclass classification is categorizing types of flowers, where the output could be one of several flower species, such as "rose," "tulip," or "sunflower."

In multiclass classification, the model is trained on a dataset with more than two labels. For instance, if we wanted to classify flower types, the dataset would include images of different flowers with labels for each species. The model learns to identify patterns in the features (e.g., petal shape, color) that correspond to each class. After training, the model is tested on new data to check how well it can predict the correct flower species.

Multiclass classification algorithms typically use one-vs-all or one-vs-one strategies to handle multiple classes. The one-vs-all approach trains a classifier for each class, while the one-vs-one approach compares every pair of classes and predicts the most likely class. These strategies help the model make accurate predictions when there are more than two potential outputs.

3.1.8

Which of the following is a typical task for multiclass classification?

- Flower species categorization
- Spam detection
- Fraud detection
- Sentiment analysis

3.1.9

In multiclass classification, which approach compares each pair of classes?

- One-vs-one
- One-vs-all
- One-vs-many

3.1.10

Multi-label classification

Multi-label classification is a more complex type of classification in which each data point can be assigned multiple labels. For instance, in an image classification task, a single image might contain a dog, a person, and a table, all of which are objects present in the image. In such cases, the classifier does not predict a single class but rather multiple classes for each data point.

In multi-label classification, the model is trained to predict several labels for each input. Unlike binary and multiclass classification, where only one label is assigned per data point, multi-label classification requires the model to recognize multiple relevant labels. For example, an image classifier might output a vector where each element represents the presence or absence of a specific object in the image. After training, the model can predict the labels for new images, such as "dog," "person," and "table," based on the patterns it learned during training.

The challenge with multi-label classification lies in ensuring that the model does not assign irrelevant labels. For instance, if a model is trained to recognize animals, it should not label a table as an animal. To address this, techniques such as binary relevance or classifier chains are used. These methods ensure that the model considers the relationships between labels and makes more accurate predictions.

3.1.11

Which of the following is an example of multi-label classification?

- Classifying objects in an image (e.g., dog, person, table)
- Predicting a single disease diagnosis
- Predicting the genre of a movie
- Sentiment analysis

3.1.12

What is a common challenge in multi-label classification?

- Ensuring that irrelevant labels are not assigned
- Assigning a single label per data point
- Managing large datasets
- Handling missing values

3.2 Taxonomy of classification models

3.2.1

Classification models are vital tools in machine learning, used to sort data into categories or predict outcomes. They differ in how they process data, handle rules, and learn from patterns. This chapter divides classification models into three main categories: **symbolical**, **statistical**, and **subsymbolical**. These categories help us understand the underlying principles and capabilities of various models and determine when and why to use them.

The distinction is based on the type of logic or mathematics a model employs to make decisions:

- **Symbolical models** use explicit, human-readable rules and logic.
- **Statistical models** rely on probabilities and mathematical distributions.
- **Subsymbolical models** mimic biological processes or are inspired by complex systems, often working as black boxes.

Understanding this taxonomy is crucial for selecting the right model for specific tasks. For example, symbolical models excel in scenarios requiring transparency, while subsymbolical models are powerful for handling unstructured data like images. Statistical models strike a balance by leveraging mathematical rigor to generalize patterns effectively.

3.2.2

Choose correct category:

- _____ models mimic biological processes or are inspired by complex systems, often working as black boxes.

- _____ models use explicit, human-readable rules and logic.

- _____ models rely on probabilities and mathematical distributions.

- Ordinal
- Statistical
- Random
- Classification
- Neural
- Symbolical
- Subsymbolical

🕮 3.2.3

Symbolical models

Symbolical classification models are based on explicit rules or logic. These models provide a clear and interpretable decision-making process. They use **if-else rules** or similar logical structures to map input data to categories. This makes symbolical models ideal for applications where interpretability and transparency are paramount, such as fraud detection or medical diagnosis.

One of the most common symbolical models is the **decision tree**, which represents decisions as a tree-like structure. Each branch of the tree corresponds to a decision rule, and each leaf represents a predicted category. For example, in a decision tree predicting whether a person will buy a product, a rule might be:

```
If age is greater than 25 and income is high, predict "yes"
```

Another example is the **rule-based classifier**, which explicitly defines sets of rules for classification. For instance, a spam filter might use rules like

```
If the email contains the word 'free' and has a large number
of links, classify it as spam
```

The main advantage of symbolical models is their interpretability. However, they can struggle with complex data and may require manual effort to define rules or grow overly complex when handling large datasets. Despite this, their simplicity makes them a valuable tool for beginners and experts alike.

3.2.4

Which of the following is a characteristic of symbolical classification models?

- They use explicit rules and logic to make decisions.
- They rely on probabilities and mathematical distributions.
- They are designed to mimic biological processes.
- They operate entirely as black-box systems.

3.2.5

Statistical models

Statistical classification models are grounded in mathematics and probabilities. These models analyze the relationships between input features and output categories to make predictions. Unlike symbolical models, statistical models often require numerical data and work well with large datasets.

A well-known statistical model is **Logistic regression**, used for binary classification. For example, logistic regression might predict whether a student will pass or fail an exam based on study hours and attendance rates. The model calculates the probability of each outcome using a sigmoid function and assigns the category with the highest probability.

Another example is **Naive Bayes**, a probabilistic model based on Bayes' Theorem. It is widely used in text classification tasks like spam detection. For instance, Naive Bayes can classify an email as spam or not by calculating probabilities based on word occurrences.

Statistical models are valued for their mathematical rigor and ability to generalize well from data. However, they may require feature engineering, and their interpretability depends on the specific algorithm. Their balance between simplicity and performance makes them a popular choice across industries.

3.2.6

Which of the following are examples of statistical classification models?

- Logistic Regression
- Naive Bayes
- Decision Tree
- Neural Networks

3.2.7

Subsymbolical models

Subsymbolical models operate on principles inspired by biology or complex systems. They are often considered black-box models because their decision-making processes are not easily interpretable. These models excel in handling unstructured data like images, text, or sound, where patterns are not easily defined by rules or probabilities.

A classic example of a subsymbolical model is the **Artificial Neural Network (ANN)**. Inspired by the human brain, ANNs consist of layers of interconnected nodes (neurons) that process and transform data. For instance, a neural network can classify whether an image contains a cat or a dog by learning patterns in pixel data.

Another example is the **Support Vector Machine (SVM)**, which finds the optimal hyperplane to separate data points of different classes. While mathematical at its core, SVMs often operate on principles that are less interpretable, making them closer to subsymbolical models.

Subsymbolical models are powerful but computationally intensive. They require large datasets and advanced hardware for training. Additionally, their black-box nature makes them challenging to interpret, which can be a drawback in sensitive applications like healthcare.

3.2.8

Which of the following best describes subsymbolical classification models?

- They often function as black boxes and handle unstructured data well.
- They rely on explicit rules and decision trees.
- They are based entirely on probabilities and statistical methods.
- They are designed only for small datasets.

3.2.9

The three categories of classification models - symbolical, statistical, and subsymbolical - serve different purposes and excel in distinct scenarios. Symbolical models are favored for their transparency and simplicity, making them suitable for applications requiring interpretable decisions. Statistical models balance interpretability and performance, offering reliable results in many domains. Subsymbolical models, while complex, are indispensable for tasks involving unstructured data.

For example, consider a bank classifying customers for loan eligibility. A symbolical model like a decision tree might be used for its interpretability, allowing the bank to explain decisions to customers. In contrast, predicting customer churn might benefit from a statistical model like logistic regression, while detecting fraudulent transactions in real time could require the power of a neural network.

Choosing the right model depends on the nature of the data, the complexity of the task, and the need for interpretability.

3.2.10

Which factors influence the choice of a classification model category?

- The nature of the data (e.g., structured or unstructured)
- The need for interpretability
- The availability of feature rules
- The transparency of subsymbolical models

3.2.11

Application

The taxonomy of classification models guides their application across various industries. For example, in **healthcare**, symbolical models like rule-based classifiers are used for diagnosing diseases due to their interpretability. Statistical models such as logistic regression are employed for predicting patient outcomes, while subsymbolical models like neural networks power advanced image-based diagnostics.

In **finance**, statistical models like Naive Bayes are utilized for credit scoring, while subsymbolical models are used for detecting fraud in large volumes of transactional data. Symbolical models might assist in compliance checks where decisions need to be explainable.

The versatility of these categories also extends to fields like **education**, where statistical models predict student performance, and **technology**, where neural networks drive speech and image recognition. Understanding the strengths and limitations of each category ensures their optimal use in solving real-world problems.

3.2.12

Which classification model is most suitable for diagnosing diseases in a transparent manner?

- Rule-based classifier
- Logistic regression
- Neural network
- Support vector machine

Tree-based Learning

Chapter 4

4.1 Decision trees

🕮 4.1.1

Decision Trees (DTs) are a nonparametric supervised learning method used for both classification and regression tasks. They work by splitting the data into smaller subsets based on decision rules derived from the input features. This process is repeated recursively, resulting in a tree-like structure where each internal node represents a decision on an attribute, and each leaf node represents an outcome. Decision Trees are popular for their simplicity and intuitive structure, making them ideal for explaining predictions to non-technical audiences.

The advantages of decision trees include:

- They are simple to understand and interpret.
- Trees can be visualized.
- It does not require additional data preparation (e.g. data normalization, removal of blanks).
- It can handle both numeric and categorical data (however, the scikit-learn library does not yet support categorical variables).
- They handle the problem of classification into multiple classes.
- They belong to the so-called white box models. They are easy to explain and interpret.
- The model can be validated using statistical tests. This allows to be taken into account the reliability of the model.

The disadvantages of decision trees are as follows:

- Decision trees can produce overly complex trees that undergeneralize the data (overfitting).
- Decision trees can be unstable because small deviations in the data can lead to the generation of a completely different tree.
- Decision tree predictions are neither smooth nor continuous. They are piecewise constant approximations.
- If some classes dominate, the wrong trees may be generated.

Decision trees are versatile and widely used in areas such as customer segmentation, medical diagnosis, and credit scoring. For example, a bank might use a Decision Tree to determine whether a customer is eligible for a loan based on income, credit history, and employment status. However, in datasets where one class significantly dominates, Decision Trees might produce biased results. Techniques like balancing the dataset or using ensemble methods like Random Forest can help address this issue.

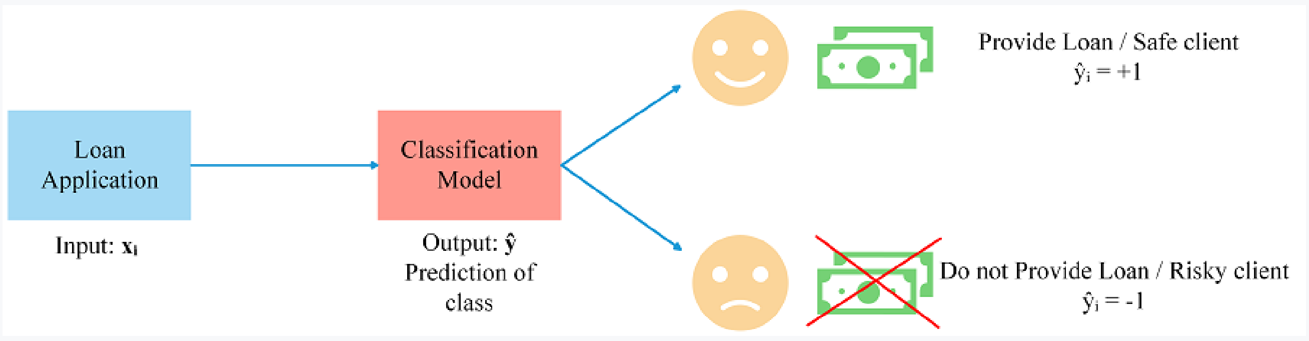
4.1.2

Which of the following are true about Decision trees?

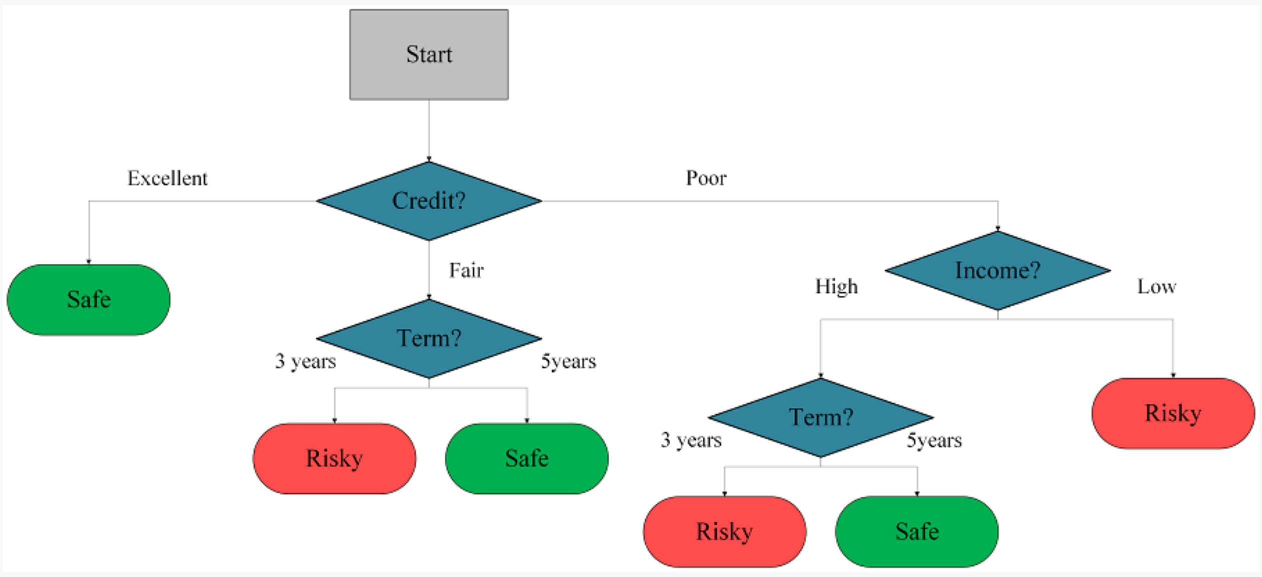
- They are simple to interpret and can be visualized.
- They do not require extensive data preparation.
- They produce smooth and continuous predictions.
- They are unaffected by small variations in the data.

4.1.3

We will demonstrate the creation of decision trees using a credit application as an example. The goal will be to create an application that, after inputting the monitored characteristics of a bank customer, decides whether or not the bank recommends granting credit to that customer. The main part of the application will be a classification model created by us, whose output will be a "Yes" or "No" recommendation.



For example, we can imagine a decision tree classification model as follows:

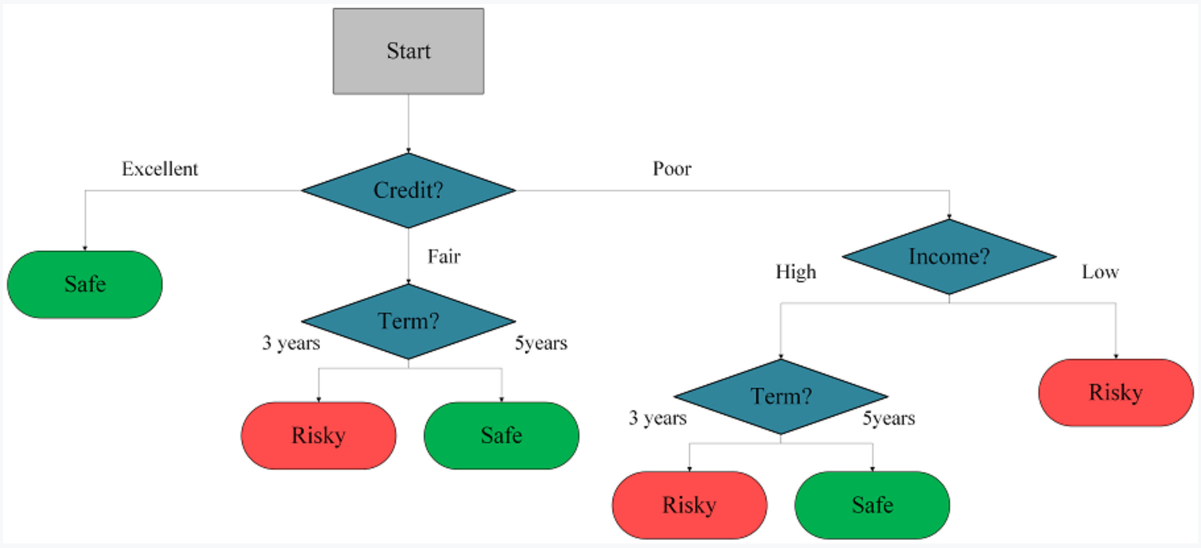


The decision tree represents a visualized set of rules for classification. In our example, a client who has a **fair** credit score and wants a **5-year loan** may apply for a loan at a bank. The created decision tree model finds that **if credit = fair**, then the length of the loan **Term** still needs to be checked. This is 5 years in the case of our client. The tree then shows that the client is marked as safe and the model recommends giving him credit.

4.1.4

According to the following decision tree model, determine the recommendation for the following client:

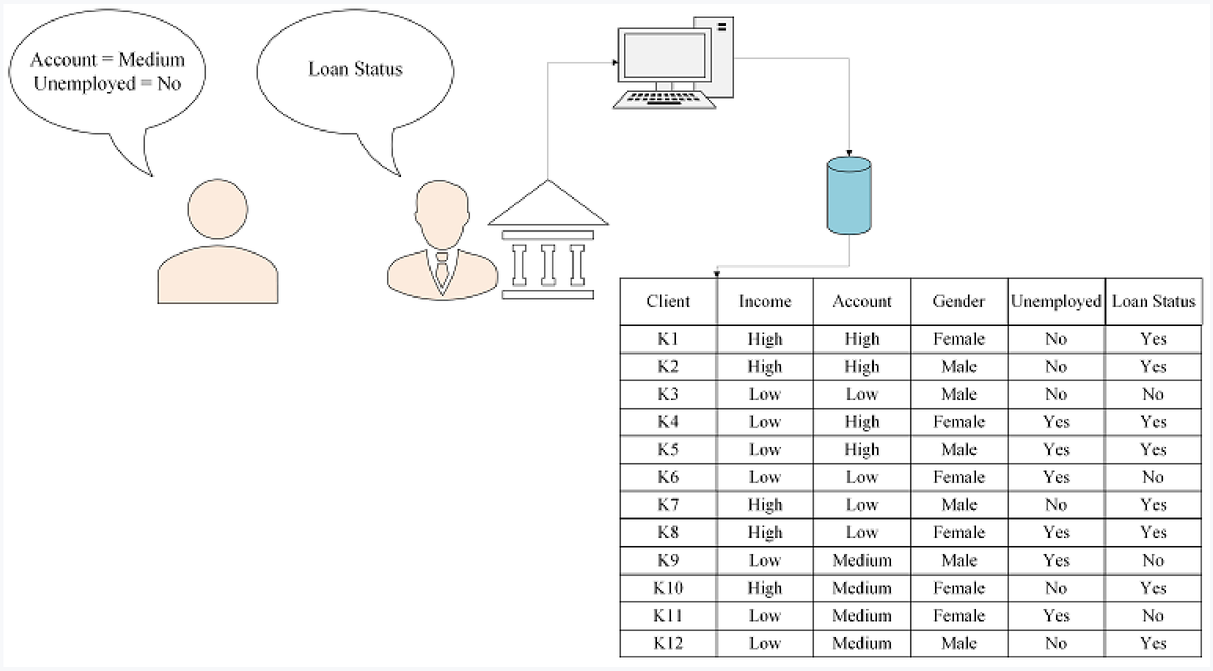
```
Client1 - (Term=5 years; Credit=poor; Income=high)
```



- Recommendation: Safe
- Recommendation: Risky

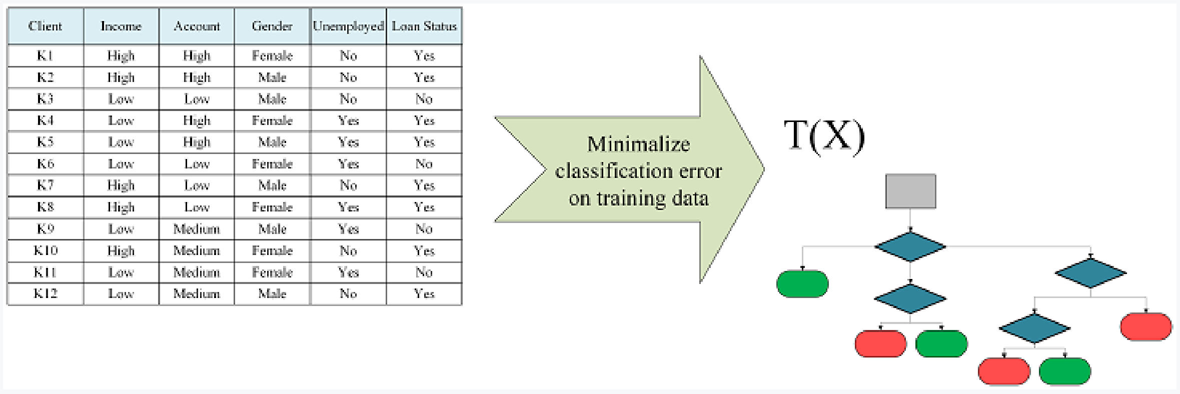
4.1.5

The question remains how to build such a decision tree model. Like all other machine learning models, decision trees will be created from historical data. Based on the historical data of previous clients, and their loans repayment or default, a decision tree will be created from this data.



Client	Income	Account	Gender	Unemployed	Loan Status
K1	High	High	Female	No	Yes
K2	High	High	Male	No	Yes
K3	Low	Low	Male	No	No
K4	Low	High	Female	Yes	Yes
K5	Low	High	Male	Yes	Yes
K6	Low	Low	Female	Yes	No
K7	High	Low	Male	No	Yes
K8	High	Low	Female	Yes	Yes
K9	Low	Medium	Male	Yes	No
K10	High	Medium	Female	No	Yes
K11	Low	Medium	Female	Yes	No
K12	Low	Medium	Male	No	Yes

Our goal will be not only to build a decision tree, but to find the best possible decision tree that will predict future credits with the smallest possible error.



Client	Income	Account	Gender	Unemployed	Loan Status
K1	High	High	Female	No	Yes
K2	High	High	Male	No	Yes
K3	Low	Low	Male	No	No
K4	Low	High	Female	Yes	Yes
K5	Low	High	Male	Yes	Yes
K6	Low	Low	Female	Yes	No
K7	High	Low	Male	No	Yes
K8	High	Low	Female	Yes	Yes
K9	Low	Medium	Male	Yes	No
K10	High	Medium	Female	No	Yes
K11	Low	Medium	Female	Yes	No
K12	Low	Medium	Male	No	Yes

4.1.6

What is the primary goal when creating a decision tree for credit applications?

- To predict future credits with the smallest possible error.
- To create a model that recommends credit with no human intervention.
- To ensure all clients are marked as safe.
- To manually check the loan terms for every client.

4.1.7

A large number of trees can be generated for the selected dataset. The exponentially large number of possible trees makes learning a decision tree difficult!

When generating a decision tree, it is important to evaluate each tree (the models created) and decide which one is better. Therefore, quantification of the quality of the tree is necessary. This mark can be determined in a number of ways, which we refer to as performance metrics.

Of these, the simplest metric appears to be: **classification error**. This is calculated as follows:

$$\text{Error} = \frac{\#\ \text{incorrect predictions}}{\#\ \text{total examples (samples)}}$$

According to the formula, it can be seen that the classification error reaches values from **0** to **1**. The closer to zero, the better classification. The best value is 0 of misclassified samples out of **n samples**. Therefore, the result will be **0**.

The worst value is **n errors** out of **n samples**, it means. **n / n = 1**

4.1.8

We have a decision tree. Out of 28 examples in the test set, 14 can classify correctly. What is his classification error?

- 14 / 28 = 0,5
- 1
- 0
- 28/14 = 2
- 14 / (28 + 14) = 0,33333

4.1.9

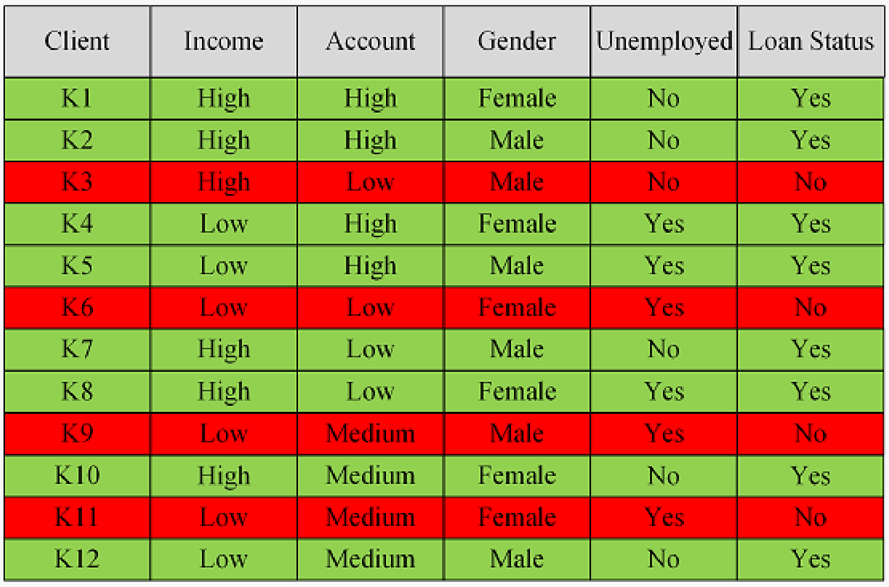
Which statements about classification error in decision trees are true?

- Classification error is a performance metric used to evaluate decision trees.
- The best classification error value is 1.
- A classification error of 0 indicates perfect classification with no errors.
- The worst classification error value is 0.

4.2 Greedy algorithm

4.2.1

The greedy algorithm is a step-by-step approach used to build decision trees. It focuses on making the locally optimal choice at each step with the hope that these choices will lead to a globally optimal solution. In the context of decision tree construction, the algorithm selects the attribute that best splits the dataset at each node, aiming to maximize the separation of classes.



Client	Income	Account	Gender	Unemployed	Loan Status
K1	High	High	Female	No	Yes
K2	High	High	Male	No	Yes
K3	High	Low	Male	No	No
K4	Low	High	Female	Yes	Yes
K5	Low	High	Male	Yes	Yes
K6	Low	Low	Female	Yes	No
K7	High	Low	Male	No	Yes
K8	High	Low	Female	Yes	Yes
K9	Low	Medium	Male	Yes	No
K10	High	Medium	Female	No	Yes
K11	Low	Medium	Female	Yes	No
K12	Low	Medium	Male	No	Yes

Imagine a training set with historical records of loan repayments. For example, this set may include 12 loans, where 8 loans were successfully repaid, and 4 loans were defaulted. The goal of the greedy algorithm is to split this data repeatedly based on the features available, creating a tree structure that can classify future loan applicants as likely to repay or default.

Steps of the Greedy algorithm:

1. **Choose the best attribute** for the root node, the algorithm evaluates all available attributes to determine which attribute provides the most effective split. Metrics such as information gain, Gini impurity, or entropy are often used for this evaluation.
2. **Split the dataset** into subsets based on the selected attribute. Each subset corresponds to one branch of the tree.
3. **Repeat the process** for each subset, the process is repeated recursively. The best attribute for splitting the subset is chosen, creating additional nodes and branches.
4. **Stop when criteria are met** - the algorithm stops when all data in a node belongs to the same class, or when further splits do not improve classification significantly.

For instance, if the training set includes attributes such as income, credit history, and loan purpose, the algorithm may first split the data based on credit history if it provides the most significant improvement in classification accuracy. Subsequent splits may focus on other attributes, forming a tree that mimics logical decision-making.

The greedy nature of the algorithm means it does not backtrack or reconsider earlier decisions. While efficient, this can lead to suboptimal solutions if the algorithm fails to consider the bigger picture.

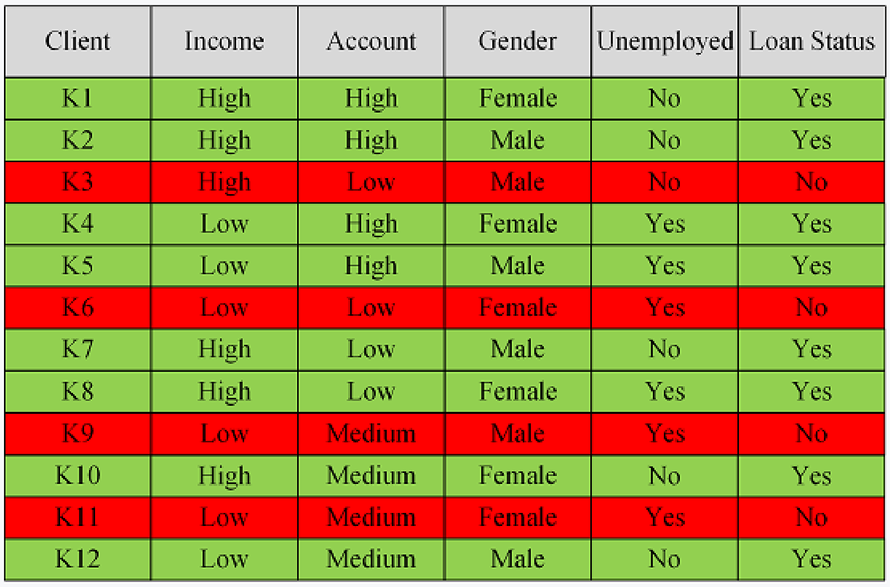
4.2.2

What is the primary goal of the greedy algorithm in decision tree construction?

- To make the best local decision at each step to maximize class separation
- To maximize the depth of the tree
- To backtrack and refine earlier splits
- To split the dataset randomly

4.2.3

From the table on the bank's past clients and their ability to repay the loan, find out how many high-income women (income = high, gender = female) have repaid their loan. Enter the number as the answer for this question.



Client	Income	Account	Gender	Unemployed	Loan Status
K1	High	High	Female	No	Yes
K2	High	High	Male	No	Yes
K3	High	Low	Male	No	No
K4	Low	High	Female	Yes	Yes
K5	Low	High	Male	Yes	Yes
K6	Low	Low	Female	Yes	No
K7	High	Low	Male	No	Yes
K8	High	Low	Female	Yes	Yes
K9	Low	Medium	Male	Yes	No
K10	High	Medium	Female	No	Yes
K11	Low	Medium	Female	Yes	No
K12	Low	Medium	Male	No	Yes

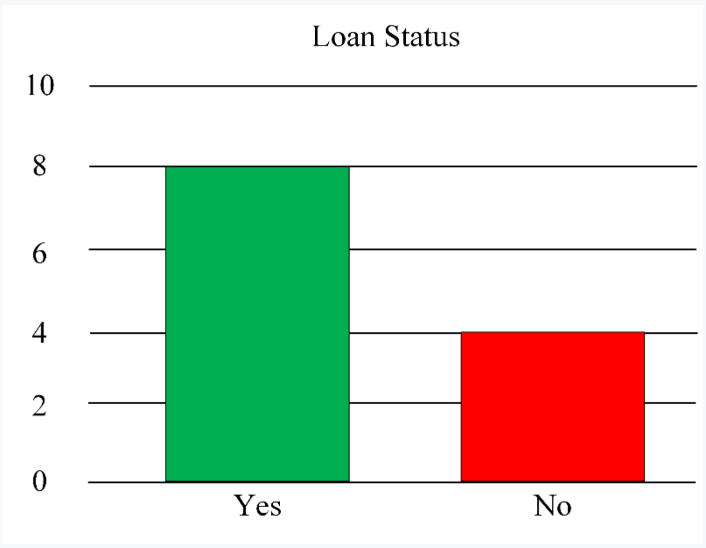
4.2.4

The first step of the greedy algorithm is:

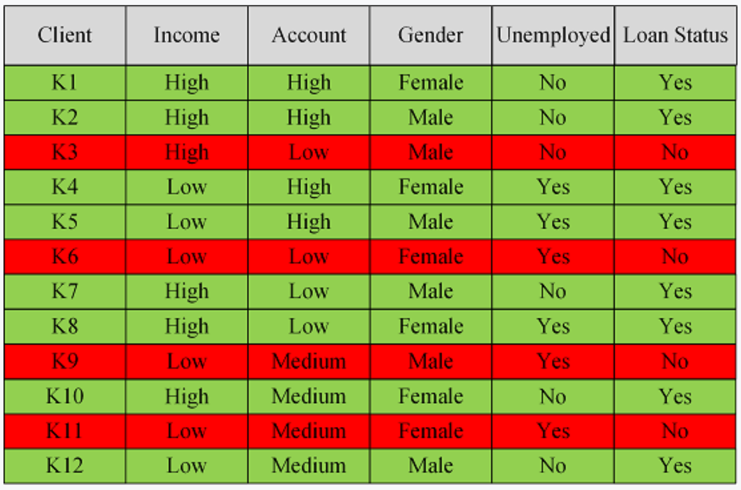
- **Start with an empty tree and calculate the classification error of the empty tree**

In the case of our dataset, where 8 clients have repaid the loan and 4 clients have not, the classification error will be 4/(4+8) = 0.3333

The frequency of distribution of clients to one of the two classes is visualized in the histogram.



The empty tree shows that if we do nothing further and say that all clients will be labelled "Yes", it means, they will be safe clients, then we make a classifier with a classification error of 0.3333.

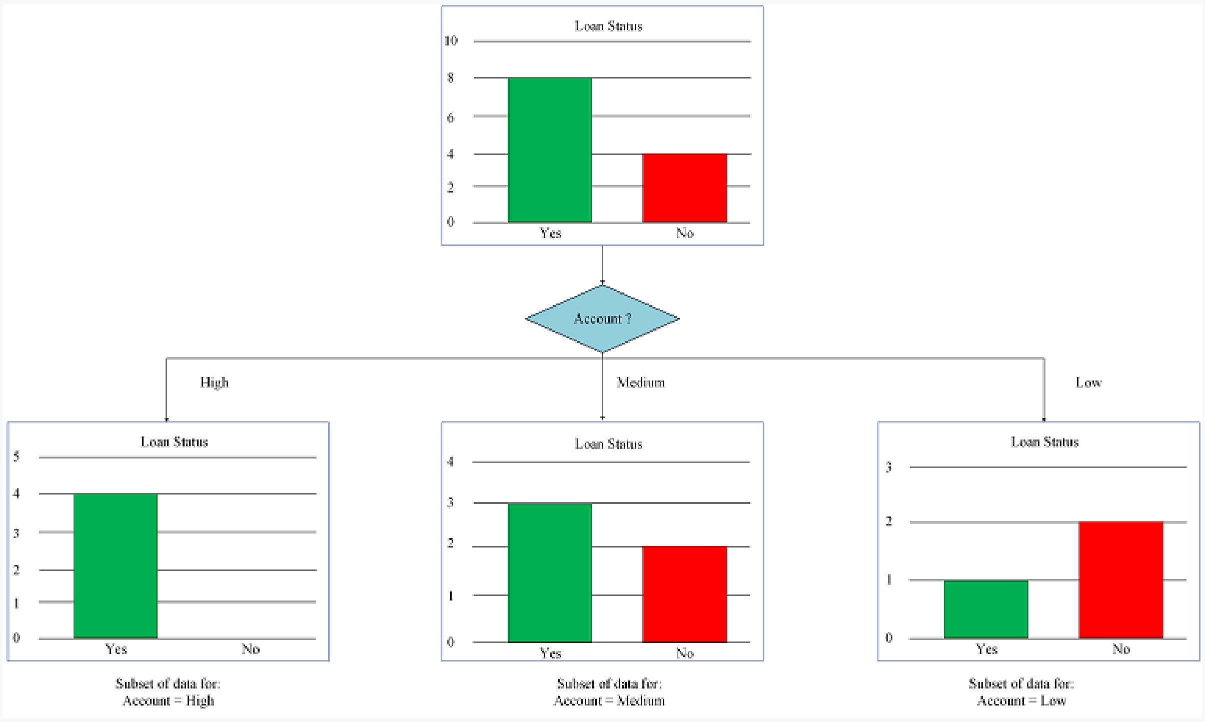


Client	Income	Account	Gender	Unemployed	Loan Status
K1	High	High	Female	No	Yes
K2	High	High	Male	No	Yes
K3	High	Low	Male	No	No
K4	Low	High	Female	Yes	Yes
K5	Low	High	Male	Yes	Yes
K6	Low	Low	Female	Yes	No
K7	High	Low	Male	No	Yes
K8	High	Low	Female	Yes	Yes
K9	Low	Medium	Male	Yes	No
K10	High	Medium	Female	No	Yes
K11	Low	Medium	Female	Yes	No
K12	Low	Medium	Male	No	Yes

The second step of the greedy algorithm will be:

- **Split the data by features/attributes.**

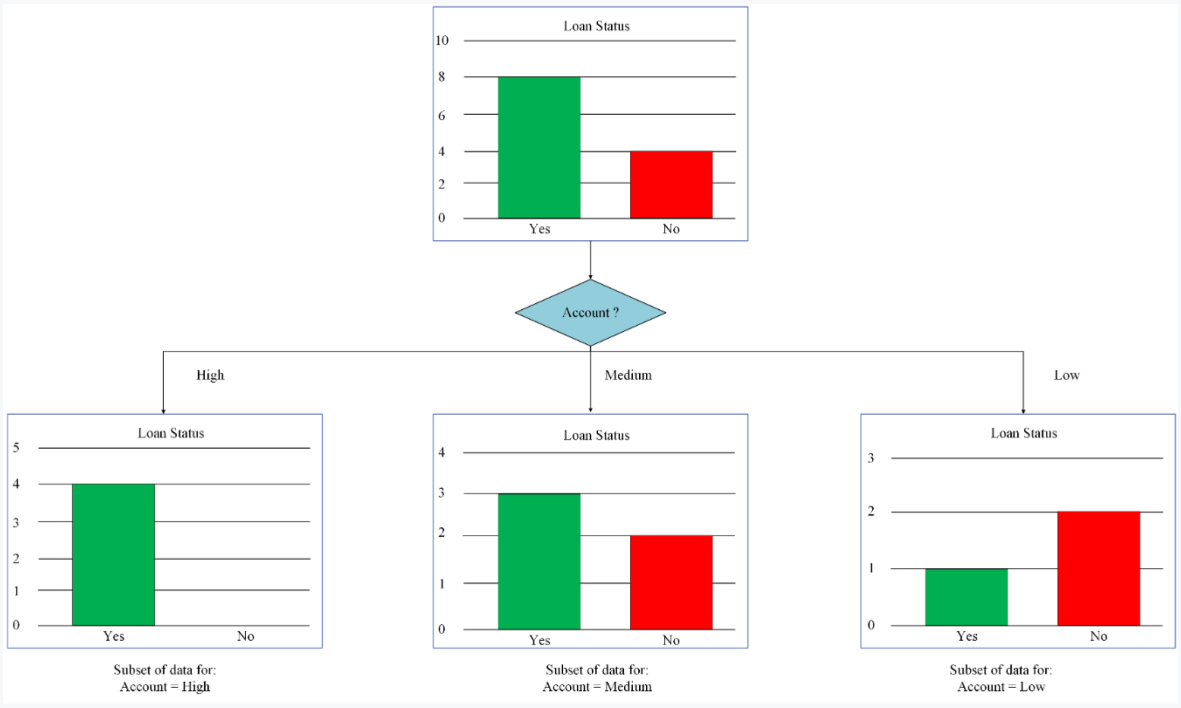
According to our training set we can split the data according to the **Account** property as follows:



For each subset, according to the splitting property of the account, we created histograms of the representation of the target variable in each subset.

4.2.5

According to the histograms, determine how many "Yes" and "No" loan repayment values were in the group with the medium account, it means for the property **Account = Medium**.



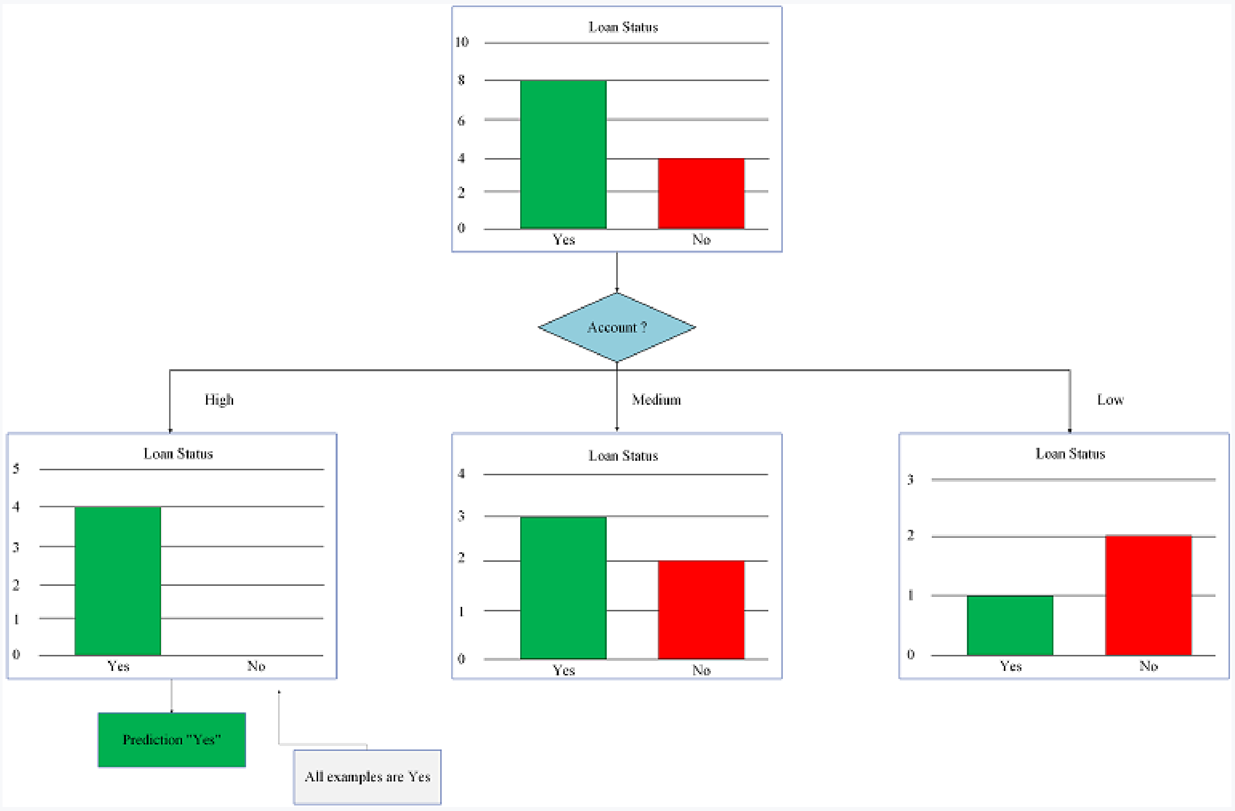
- Yes = 3; No = 2
- Yes = 1; No = 2
- Yes = 4; No = 0

📖 4.2.6

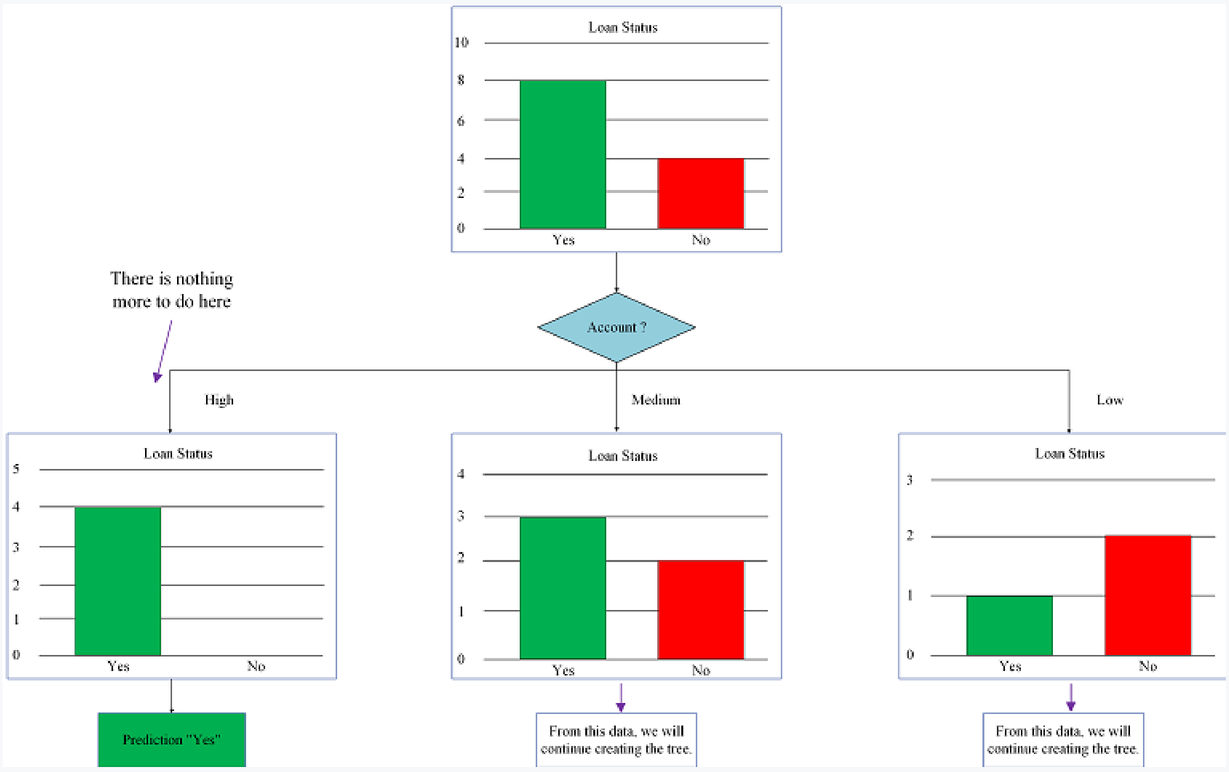
We just created a depth-1 tree with one splitting property - **Account**. The third step of the greedy algorithm is:

- **Make a prediction if possible**

Note that for the condition **Account = High**, all examples are in the class "**Yes**". This means that in the past, all clients who had a high account have repaid the loan. In this case, we can make a prediction because there is no data from any other class already in this branch of the tree.



In the case of medium and low accounts, there is no clear class. Nevertheless, we can also make a prediction here if necessary. We could make a prediction according to the majority of the class, i.e. for the medium account the prediction would be **Yes** and for the low account it would be **No**. However, this would only create a tree of depth 1. Therefore, a better option is to continue recursively creating a tree from each subset of the data.


Loan Status
Yes
No
Account ?
There is nothing more to do here
High
Medium
Low
Prediction "Yes"
From this data, we will continue creating the tree.
From this data, we will continue creating the tree.

4.2.7

We can now summarize the whole greedy algorithm. It looks like this:

```
Step 1: start with empty trees
Step 2: select a property for data splitting
Step 3: create a distribution according to the selected
property

For each distribution of the tree:
                   Step 4: If you cannot go any further, make
a prediction
                   Step 5: Otherwise go to Step 2 & continue
with recursion of this distribution
```

In the given algorithm we find 2 questionable parts. The first is **Step 2.** It is a problem of feature selection, i.e. Feature split selection.

The second questionable part is **Step 4,** which deals with the problem of stopping the tree creation, i.e. stopping condition.

We discuss both of these issues in the next chapter.

4.2.8

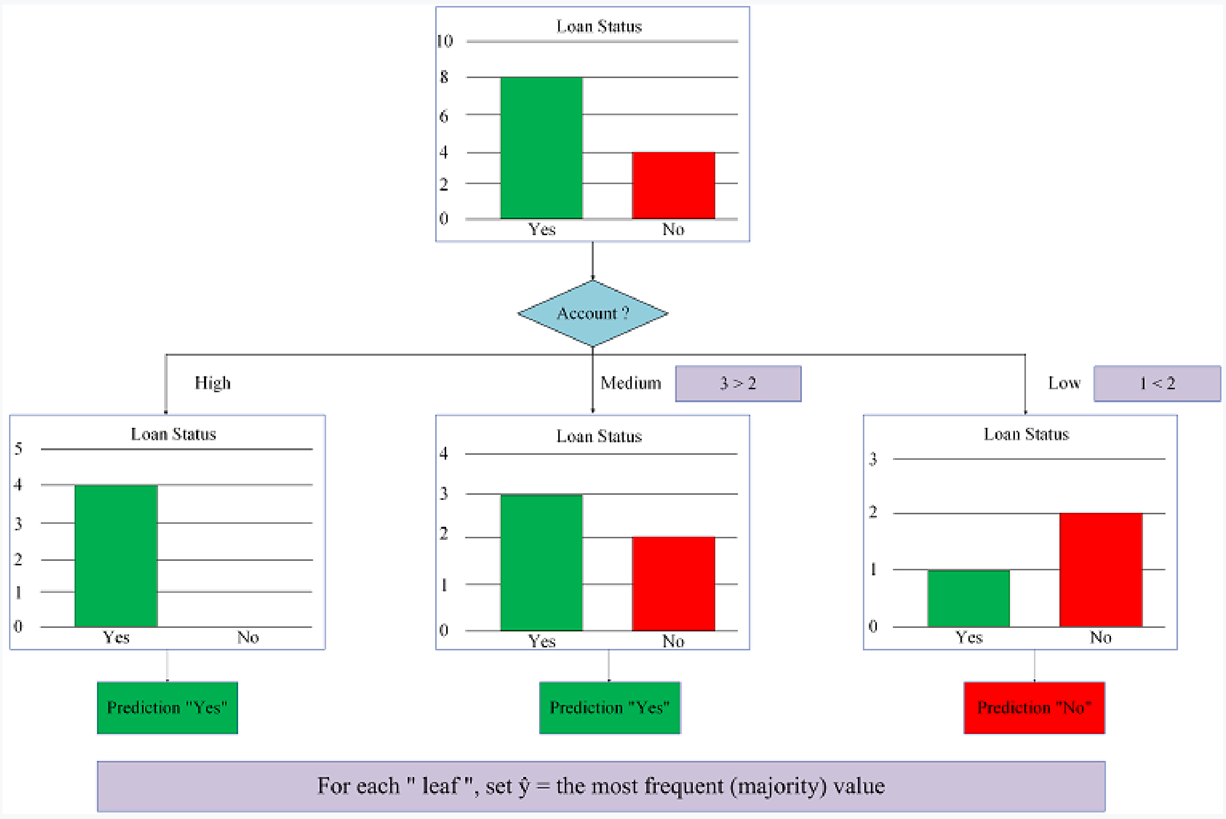
What are the questionable parts of the greedy algorithm when building decision trees?

- Feature split selection
- Stopping condition for tree creation
- Prediction based on random guessing
- Recursive creation of a tree
- Always creating a depth-1 tree

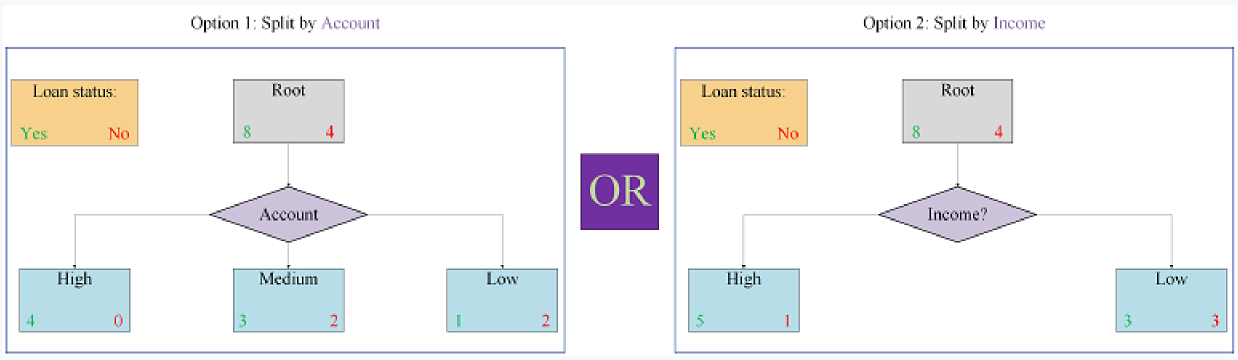
4.3 Choosing the best properties

4.3.1

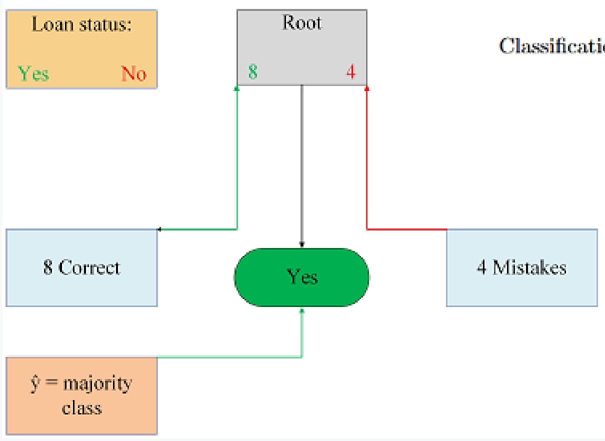
The first problem in the construction of decision trees is the selection of the best feature for distribution. This problem takes advantage of the "computational power of machine learning algorithms". In our example, the algorithm will proceed by creating a simple tree of depth 1 for each feature considered. In such a tree, it must always decide for prediction. Therefore, it will perform the prediction according to the most represented class in the data subset.



Let us now consider all the trees of depth 1 constructed in this way. To find the most appropriate feature, we use the classification error. For example, we can compare a tree for a distribution according to **Account** and a tree according to **Receipt**.

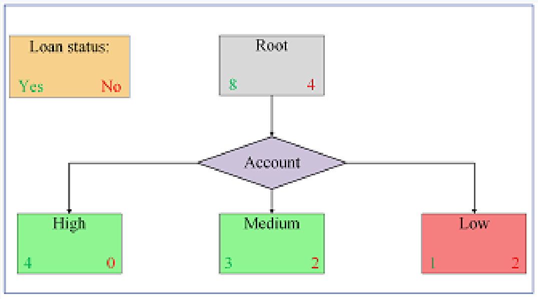


We perform the comparison according to the classification error. Thus, we select the feature whose tree has the smallest classification error. We also compare this with the empty tree.

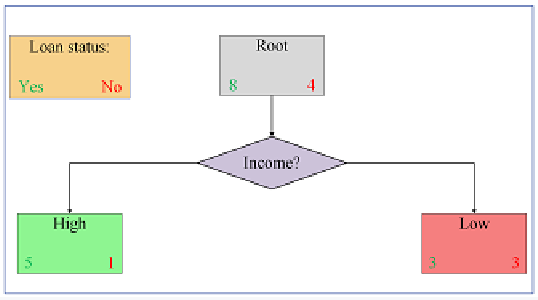


$$\text{Classification Error} = \frac{\text{Number of incorrect classifications}}{\text{Total number of examples}} = \frac{4}{8+4} = 0.33$$

Tree	Classification Error
(Root)	0.33

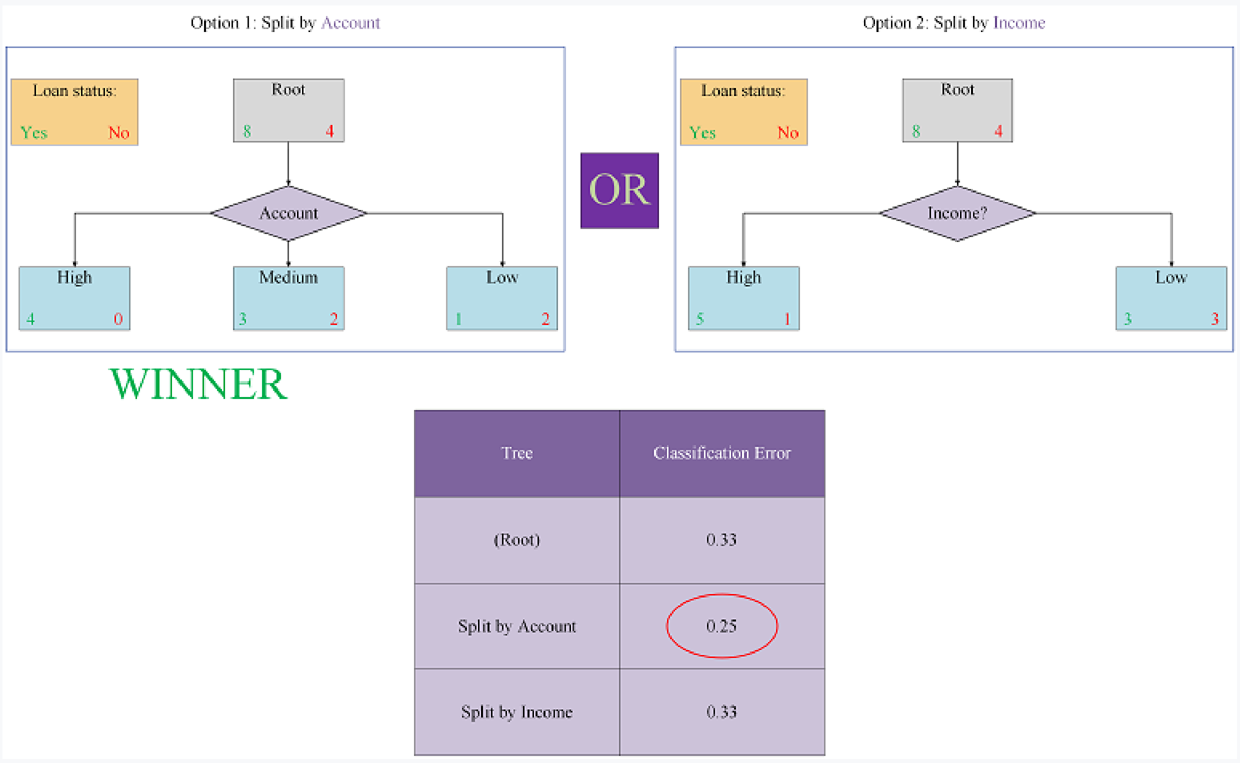


$$\text{Classification Error (Account)} = \frac{3}{8+4} = 0.25$$



$$\text{Classification Error (Income)} = \frac{4}{8+4} = 0.33$$

It remains for us to compare the results of the classification errors.



Tree	Classification Error
(Root)	0.33
Split by Account	0.25
Split by Income	0.33

The lowest classification error is obtained by dividing by **ACCOUNT**.

4.3.2

Correctly complete the formula for calculating the classification error and the best and worst possible value of the classification error

```
classification error =          _____
_____________________________________________
                                _____
```

The best possible value of the classification error is _____.

The worst possible value of the classification error is _____.

- number of incorrect predictions
- 1
- number of all examples
- 0

4.3.3

Feature split selection algorithm

The feature split selection is a crucial step in building decision trees. This process determines how the dataset at each node is divided, directly influencing the tree's accuracy and complexity. The aim is to identify the best feature to split the data in a way that minimizes misclassification. The algorithm for feature split selection follows a structured approach to ensure optimal results.

1. The algorithm starts with a given subset of data **M**, representing the current node in the decision tree. This subset contains historical information used to evaluate different features. The main idea is to systematically assess each feature to determine its ability to separate data into more homogeneous groups, thereby improving prediction accuracy.
2. For each feature **h_i(x)**, the algorithm partitions the subset **M** using a splitting rule associated with that feature. This creates a single-level decision tree. For instance, if the feature is "Account Balance," the data might be split into subsets like "High," "Medium," and "Low." Each subset is then evaluated for its classification accuracy.
3. The next step is to calculate the classification error for the resulting tree created by the feature **h_i(x)**. The classification error measures the proportion of misclassified samples, guiding the algorithm to identify splits that reduce errors.
4. After evaluating all features, the algorithm selects the function **h_*(x)** with the smallest classification error as the optimal feature for the split.

This approach ensures that the tree grows in a way that improves prediction accuracy with each level.

4.3.4

What is the primary goal of the feature split selection algorithm in decision trees?

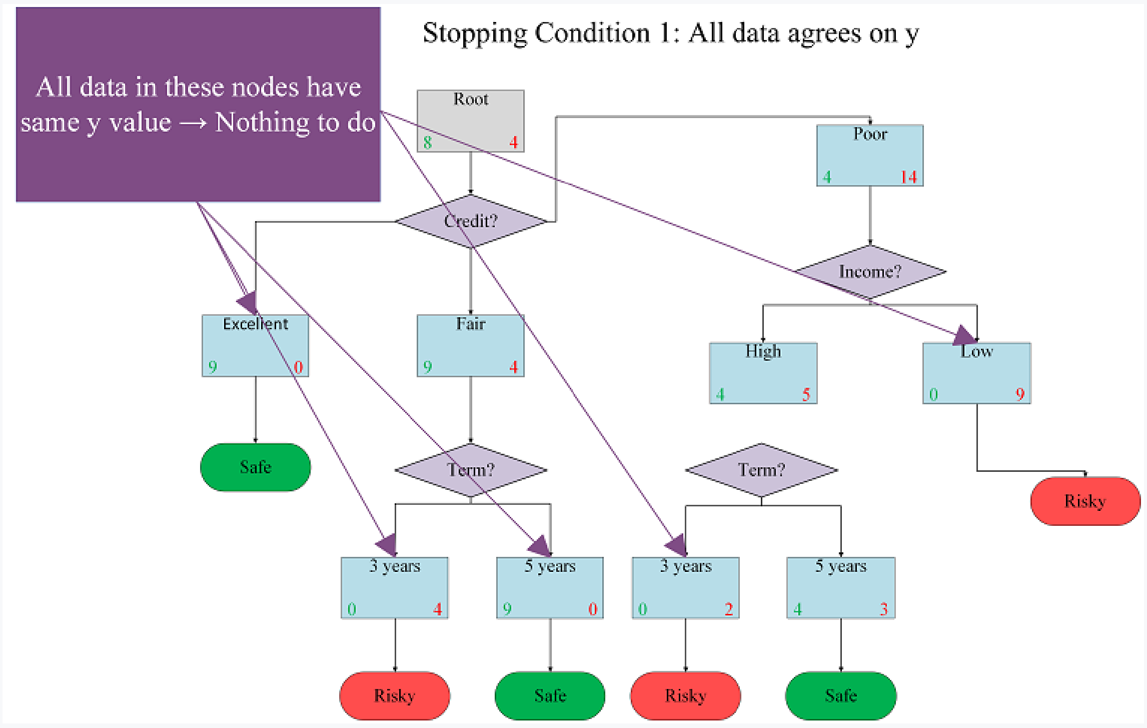
- To minimize classification error by selecting the best feature for splitting
- To maximize the depth of the decision tree
- To reduce the number of features in the dataset
- To eliminate outliers from the data

4.3.5

Stopping conditions

The stopping condition is a critical component of the greedy algorithm in decision tree construction. It determines when to halt the growth of the tree, ensuring that the model remains effective without becoming overly complex. Without a proper stopping condition, decision trees can grow excessively, leading to overfitting and poor generalization on new data.

1. One of the most straightforward stopping conditions occurs when **there is nothing left in the nodes to split**. This situation arises when all the samples in a node belong to the same class or when the data cannot be further divided based on the available features. For example, if all clients in a node have successfully repaid their loans, no further splitting is required, and the tree can terminate at that point.



2. Another stopping condition is when **all the features in the dataset have been used**. Since decision trees split data based on features, the algorithm must stop when no additional features are available for further partitioning. This ensures that the tree construction process remains finite and avoids unnecessary computations.

3. The third common stopping condition is when **further division does not lead to a lower classification error**. If the algorithm finds that splitting a node does not improve the tree's performance or reduces misclassification rates, it halts the process. This approach balances model complexity and prediction accuracy, preventing the creation of overly detailed trees that capture noise instead of meaningful patterns.

4.3.6

Which of the following are stopping conditions for the greedy algorithm in decision tree construction?

- There is nothing left in the nodes to split
- All features have been used
- The tree reaches a depth of 5
- Further division does not achieve a lower classification error

4.3.7

The previous examples aimed to simplify the process of decision tree creation to make it more understandable. While we used classification error as the criterion for feature selection, this metric is rarely applied in real-world tasks. Instead, more sophisticated measures, such as the **Gini Index** or **Information Gain**, are preferred. These metrics provide a more detailed evaluation of how well a feature splits the data into meaningful subsets.

Another simplification was the evaluation of the model's success using classification error alone. In practical scenarios, this approach is often insufficient. Instead, performance measures like **accuracy**, **precision**, **recall**, and **F1-score** are commonly employed to evaluate and compare models. Among these, accuracy is the quickest and simplest to calculate, making it a popular choice for initial evaluations.

Additionally, for ease of explanation, we focused only on categorical variables in tree formation. However, real-world datasets often contain both categorical and continuous variables. When dealing with continuous variables, decision tree algorithms introduce thresholds to divide the data. For example, instead of directly using "income" as a feature, a tree might use conditions like "income > $50,000" to split the data effectively.

Fortunately, modern programming languages provide libraries with ready-made methods for decision tree construction. These libraries handle the complex steps of tree creation, such as choosing the stopping condition or selecting the feature split metric. Users can customize these aspects by simply setting method parameters, making decision tree implementation both flexible and accessible for a wide range of tasks.

4.3.8

In a simple dataset, the features (rich, handsome) of the last four suitors of Gertrude B are recorded. The **relationship** feature tells whether Gertrude B. stayed with a suitor for more than 1 month, i.e. it records long-term relationships.

We want to create a decision tree model that will predict whether Gertrude will stay with her partner for more than a month, i.e. we will model the **relationship** feature. Which feature (**rich** or **handsome**) will be selected as the first feature to create such a decision tree? I.e. which of the two features will be selected as more appropriate for data- feature split selection?

- both features can be selected as appropriate, i.e. likely to be selected at random
- rich
- handsome

4.3.9

Which of the following metrics is commonly used for feature selection in real-world decision trees?

- Gini Index
- Classification error
- Accuracy
- Recall

4.4 Performance metrics

4.4.1

Now that we can create a decision tree using a greedy algorithm, the next step is to evaluate its performance. In machine learning, performance evaluation is crucial to understand how well a model generalizes to unseen data. For decision trees, two basic metrics are commonly used: **classification error** and **classification accuracy**.

Classification error measures the proportion of incorrectly classified instances in the dataset. It is calculated using the formula:

```
Classification Error = Number of incorrect classifications /
Total number of examples
```

This metric ranges from 0 to 1, where a value of 0 indicates perfect classification (no errors) and a value of 1 indicates that every prediction was incorrect.

Classification accuracy, on the other hand, evaluates the proportion of correctly classified instances. Its formula is:

```
Accuracy = 1 - Classification Error
```

Or equivalently:

```
Accuracy = Number of correct classifications / Total number of examples
```

These two metrics are inversely related. A low classification error implies high accuracy, and vice versa. For instance, if a decision tree has a classification error of 0.2 (20%), its accuracy would be 0.8 (80%).

While these metrics are intuitive and easy to calculate, they are not always sufficient to assess a model's true performance, especially in datasets with imbalanced classes. Advanced metrics like precision, recall, and F1-score are often used alongside these to provide a more comprehensive evaluation.

4.4.2

Which of the following statements about classification error and classification accuracy are true?

- Classification error measures the proportion of incorrectly classified instances.
- Classification accuracy is equal to 1−Classification Error.
- Classification accuracy can exceed 100%.
- Classification error is always higher than classification accuracy.

4.4.3

Metrics are essential in machine learning because they provide a **quantitative way to measure the performance** of models. When working with decision trees or any other model, metrics help to understand how well the model is performing and whether it meets the expectations. Without metrics, it would be difficult to assess whether a model is improving, how it compares to previous versions, or how it fares against other models.

Metrics are important for several reasons:

- **Quantifying model quality** - metrics transform the abstract concept of model performance into concrete numbers. For instance, classification accuracy tells us what percentage of our model's predictions were correct. Other metrics, such as precision, recall, and F1-score, allow us to further break down the performance based on specific classes, especially when dealing with imbalanced datasets.
- **Tracking improvements** - as you refine your machine learning models - whether by changing the model structure, tuning hyperparameters, or

improving the dataset - metrics help quantify improvements. They show how well the model adapts to changes over time, providing a clear picture of whether a particular change has had a positive impact.
- **Guiding model optimization** - metrics can also guide decisions on **pruning** and other model optimizations. In decision trees, for example, metrics can inform when to stop growing the tree or prune certain branches to prevent overfitting.
- **Benchmarking expectations** - metrics are also crucial for comparing **actual performance** to **expected or desired performance**. During model development, you may set performance goals based on the dataset, problem complexity, or domain. Metrics allow you to evaluate how close the model comes to meeting these goals.
- **Measuring progress** - over time, you want to track whether the model's performance is improving. By consistently using the same metrics, you can measure progress towards better accuracy, faster predictions, or other desired attributes.

4.4.4

Why are metrics important in machine learning?

- Metrics help to quantify the quality of the model and track improvements over time.
- Metrics can predict future data without needing a model.
- Metrics provide a solution to all types of machine learning problems.
- Metrics always guarantee that the model will be correct.

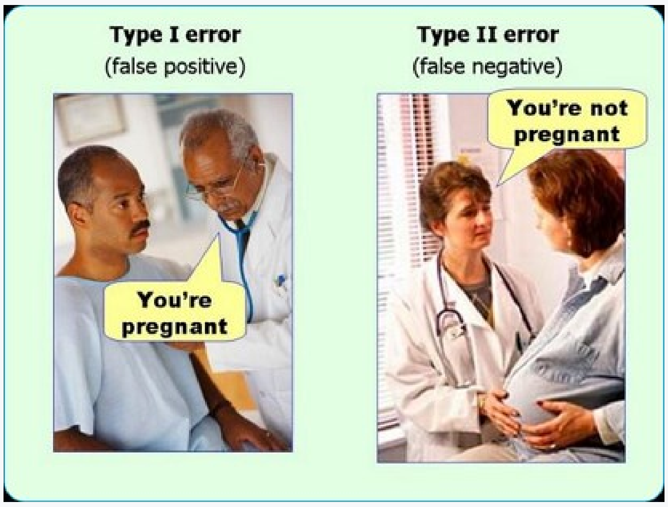
4.4.5

Classification errors

In classification tasks, evaluating the performance of a model requires an understanding of the types of errors it can make. Specifically, we focus on two main types of errors: **false positives** and **false negatives**. These errors have significant implications for how we measure the success of a model and how we can improve it.

- **False Positive** (Type I error) occurs when the model incorrectly classifies a negative example as positive. In other words, the model predicts the presence of a condition or event that does not exist. This type of error is also called a **false alarm**. For example, in a medical test designed to detect a disease, a false positive would mean that a healthy person is incorrectly diagnosed as sick. False positives are particularly problematic when the cost of a false alarm is high, such as in security systems or fraud detection.

- **False Negative** (Type II error) occurs when the model incorrectly classifies a positive example as negative. In this case, the model fails to identify the condition or event that is actually present. For example, in the same medical test scenario, a false negative would mean that a person who is actually sick is incorrectly classified as healthy. False negatives are often more dangerous because they can result in missed opportunities for intervention, such as failing to diagnose a patient with a treatable condition.



According to these two types of errors, we can calculate the performance metrics of the model-classifier.

Imagine a fraud detection system used by a bank to identify fraudulent transactions. If the system incorrectly labels legitimate transactions as fraud (false positives), customers might be inconvenienced with incorrect alerts. If it fails to detect actual fraudulent transactions (false negatives), the bank could suffer financial losses. In this case, the model needs to be optimized to minimize both types of errors, depending on which one has a higher cost.

4.4.6

Which of the following statements are correct regarding false positives and false negatives?

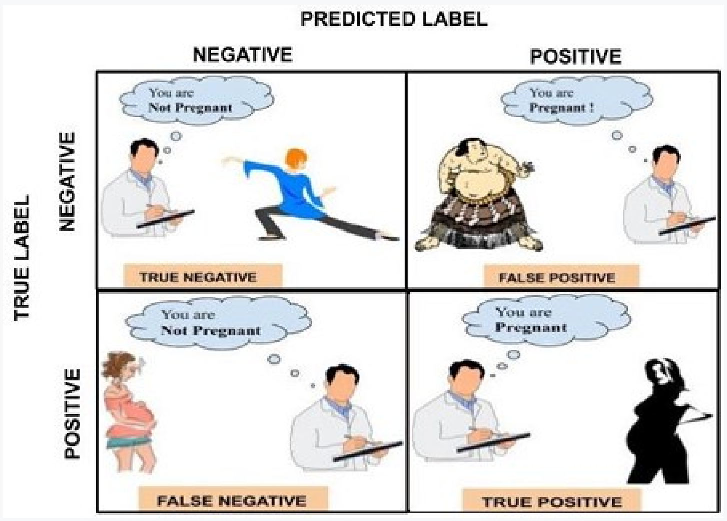
- False negatives occur when a positive instance is incorrectly classified as negative.
- False positives are always less costly than false negatives.
- False positives occur when a negative instance is incorrectly classified as positive.
- False negatives are always more costly than false positives.

4.4.7

In addition to understanding classification errors, it is also essential to recognize the types of classification **successes** that a model can achieve. These successes correspond to when the model makes correct predictions, either by correctly labeling a positive example as positive or a negative example as negative.

Types of classification success

1. **True Positive (TP)** occurs when a positive example is correctly classified as positive. In other words, the model correctly predicts the presence of the condition or event. For example, in a medical test, a true positive would mean that a patient who actually has a disease is correctly identified as having it.
2. **True Negative (TN)** happens when a negative example is correctly classified as negative. In this case, the model accurately predicts that the condition or event does not exist. For example, a true negative would mean that a healthy patient is correctly identified as not having the disease.



These two types of success contribute to the overall **accuracy** of the model, which measures how often the model makes the correct prediction, both for positive and negative instances.

4.4.8

Which of the following statements about classification success is true?

- A true positive occurs when a positive instance is correctly labeled as positive.
- A true negative occurs when a positive instance is incorrectly labeled as negative.
- A true negative occurs when a negative instance is incorrectly labeled as positive.

- A true positive occurs when a negative instance is incorrectly labeled as positive.

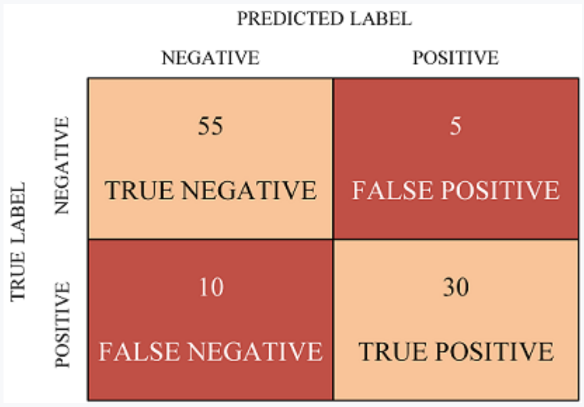
4.4.9

Confusion matrix

The **confusion matrix** is a tool used to summarize the performance of a classification model. It is a table that allows you to visualize how well the model is performing with respect to the actual class labels and predicted class labels. The confusion matrix includes:

- **True Positives (TP)** - correctly predicted positive cases.
- **True Negatives (TN)** - correctly predicted negative cases.
- **False Positives (FP)** - negative cases incorrectly classified as positive.
- **False Negatives (FN)** - positive cases incorrectly classified as negative.

The confusion matrix is a valuable tool for understanding how the model is performing in greater detail than just using a single metric like accuracy. It helps us see where the model is making mistakes, whether it is more prone to false positives or false negatives, and thus allows for more informed decision-making when tuning or evaluating the model.



then the **Accuracy** can be calculated as follows:

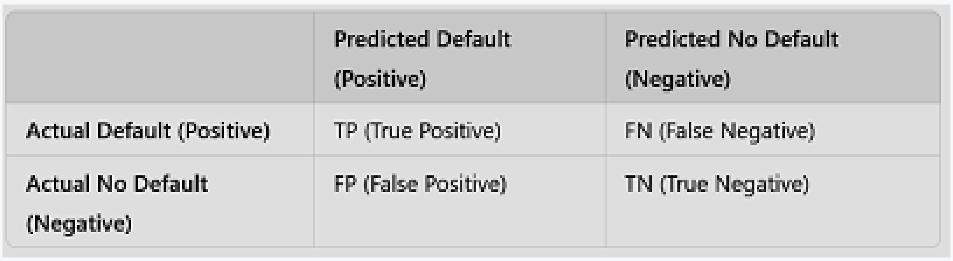
```
Accuracy = Number of correct classifications / Total number of
examples
         = (TrueNegative + TruePositive) / (TrueNegative +
FalsePositive + TruePositive + FalseNegative)

Accuracy = (TN + TP) / (TN + FP + TP + FN)
```

For the above mentioned confusion matrix, Accuracy is calculated by substituting into the formula:

```
Accuracy = (55 + 30) / (55 + 5 + 30 + 10)
```

Example: Credit risk classification

Let's imagine a credit risk classification model used by a bank to predict whether a customer will default on a loan. The model will classify customers into two categories: those who will default (positive class) and those who will not (negative class). The confusion matrix for this model might look like:



	Predicted Default (Positive)	Predicted No Default (Negative)
Actual Default (Positive)	TP (True Positive)	FN (False Negative)
Actual No Default (Negative)	FP (False Positive)	TN (True Negative)

This matrix shows how many instances were correctly classified as "default" (TP) and "no default" (TN), as well as how many defaults were missed (FN) and how many non-defaults were falsely flagged as defaults (FP).

4.4.10

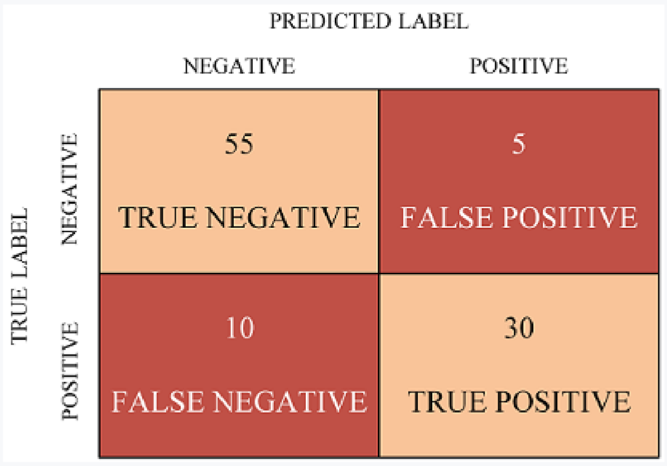
Which of the following statements about classification success is true?

- A true positive occurs when a positive instance is correctly labeled as positive.
- A true negative occurs when a positive instance is incorrectly labeled as negative.
- A true negative occurs when a negative instance is incorrectly labeled as positive.
- A true positive occurs when a negative instance is incorrectly labeled as positive.

4.5 Advanced metrics

4.5.1

Accuracy is a popular metric in machine learning for assessing a model's performance. However, is accuracy a good metric? In the previous example, we used the following confusion matrix



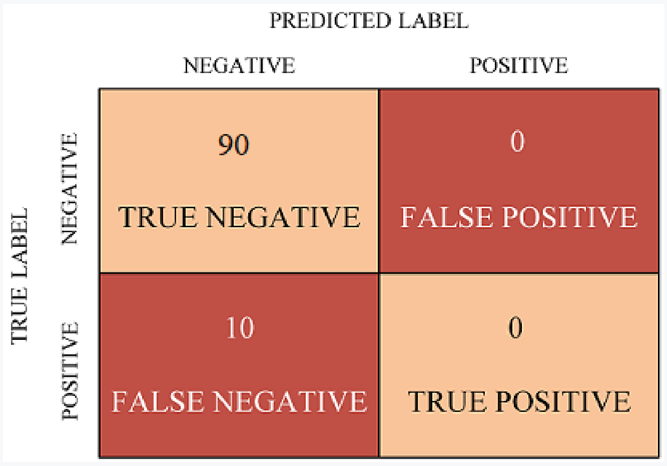
PREDICTED LABEL

TRUE LABEL	NEGATIVE	POSITIVE
NEGATIVE	55 TRUE NEGATIVE	5 FALSE POSITIVE
POSITIVE	10 FALSE NEGATIVE	30 TRUE POSITIVE

We calculated:

```
Accuracy = (TN + TP) / (TN + FP + TP + FN)
Accuracy = (55 + 30) / (55 + 5 + 30 + 10) = 0.85
```

However, its reliability decreases significantly when dealing with **imbalanced datasets**, where one class greatly outnumbers the other. Consider a heavily imbalanced dataset where 90% of the examples are negative. If we build a naive classifier that predicts all samples as **negative**, the confusion matrix looks like this:



PREDICTED LABEL

TRUE LABEL	NEGATIVE	POSITIVE
NEGATIVE	90 TRUE NEGATIVE	0 FALSE POSITIVE
POSITIVE	10 FALSE NEGATIVE	0 TRUE POSITIVE

Based on matrix, we can now calculate the accuracy:

```
Accuracy = (TN + TP) / (TN + FP + TP + FN)
Accuracy = (90 + 0) / (90 + 0 + 0 + 10) = 0.9
```

This classifier is clearly weak, as it completely fails to identify any positive cases, but the high accuracy score misleadingly suggests otherwise.

Using Accuracy in such scenarios, it can lead to misleading interpretation of the results. When datasets are imbalanced, accuracy is dominated by the majority class. It doesn't reflect how well the model identifies minority class instances, which are often the focus of interest (e.g., detecting fraud or disease).

Instead of accuracy, metrics like **precision**, **recall**, and the **F1-score** provide more balanced evaluations of model performance.

4.5.2

Why is accuracy not a good metric for imbalanced datasets?

- It does not account for class imbalances.
- It requires additional computations.
- It only works with balanced datasets.
- It depends entirely on precision and recall.

4.5.3

Prevalence

In addition to the other metrics, there is a dataset balance metric. Prevalence is a metric that reflects the balance of a dataset by quantifying the proportion of positive samples relative to the total number of samples. It is calculated as:

```
Prevalence = #positives / (#positives + #negatives)
```

The ideal prevalence value is around **0.5**, indicating an even balance between positive and negative samples. When prevalence deviates significantly from 0.5, it signals an imbalance that may affect the performance of machine learning models, especially those that rely on metrics like accuracy.

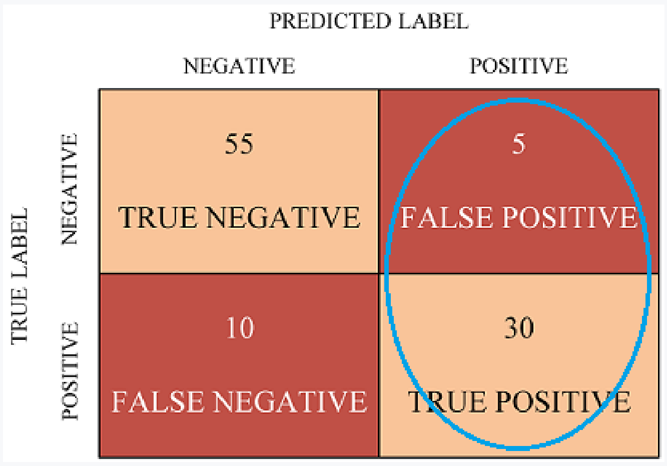
Importance of prevalence is based on:

1. **Dataset imbalance detection** - prevalence helps identify whether a dataset is skewed toward one class. For example, in medical datasets, where only 5% of samples may represent a rare disease, the prevalence is 0.05, indicating a significant imbalance.
2. **Metric selection** - when prevalence is far from 0.5, relying on accuracy as a performance metric can be misleading. Instead, other metrics like **F1-score**, **precision**, and **recall** become more reliable.

3. **Model choice and training** - understanding prevalence can guide decisions about balancing techniques, such as resampling or adjusting class weights, to ensure that minority classes are properly represented during training.

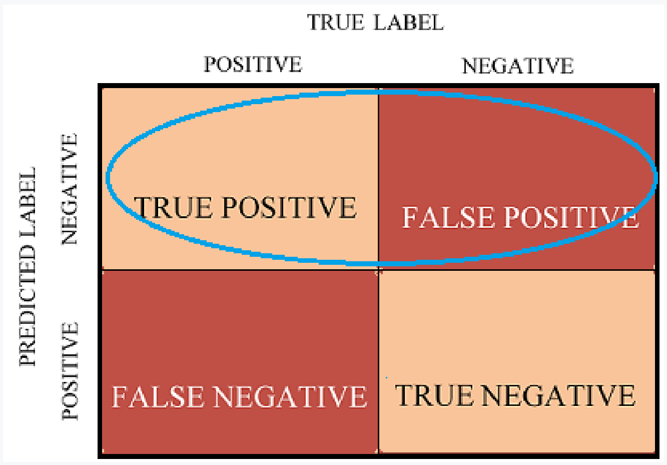
Confusion matrices can be presented in various formats, which is why checking row and column headers is crucial. Here are two examples:

Standard notation where rows represent the **actual** labels, and columns represent the **predicted** labels:



Inverted Notation

Some confusion matrices may flip the rows and columns:



It is always necessary to check the column and row headers in confusion matrix. Both notations contain the same information but interpret the axes differently, making careful observation essential.

4.5.4

Which of the following are true about prevalence?

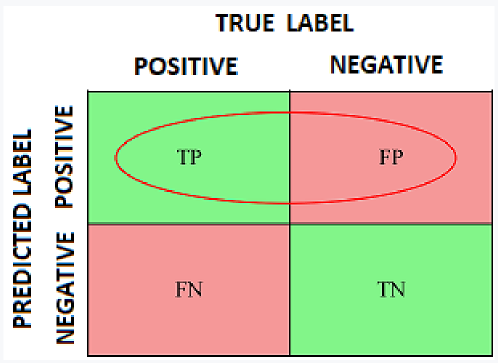
- It measures the balance of positive to total samples.
- An ideal prevalence value is around 0.5.
- It helps detect dataset imbalances.
- It is irrelevant to confusion matrices.

4.5.5

Precision

Precision is a critical performance metric for evaluating classifiers, especially in scenarios where the cost of false positives is high. It quantifies the proportion of correctly predicted positive examples out of all examples predicted as positive. Precision is calculated as:

```
Precision = True Positives (TP) / (True Positives (TP) +
False Positives (FP))
```



Accuracy should ideally achieve 1 (high) for a good classifier.

Precision indicates how reliable the classifier's positive predictions are. A high precision score means that when the model predicts a positive label, it is correct most of the time. This is especially important in applications such as:

1. Spam detection - a low precision in spam detection could mean that many legitimate emails are incorrectly flagged as spam.
2. Fraud detection - high precision ensures that flagged transactions are likely to be fraudulent, minimizing unnecessary interventions.
3. Medical diagnoses - for life-critical conditions, high precision reduces false alarms, ensuring that only true cases are flagged for further investigation.

Precision achieves the ideal value of **1** when the number of false positives (FP) is zero, meaning every positive prediction is correct. However, precision can be sensitive to class imbalances. For instance, in datasets where the positive class is rare, precision alone may not reflect the overall performance and should be considered alongside other metrics like recall and F1-score.

4.5.6

What does a high precision score indicate in a classification model?

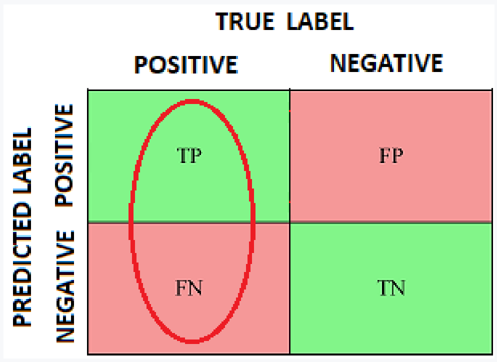
- Most positive predictions made by the model are correct.
- The model predicts both positive and negative classes equally well.
- The model minimizes false negatives effectively.
- The dataset is balanced.

4.5.7

Recall

Recall, also known as sensitivity or the true positive rate, is a critical metric in classification, especially in tasks where identifying all positive instances is essential. Recall quantifies the ability of the model to detect all actual positive cases and is calculated as:

```
Recall = TP / (TP + FN)
```



Recall emphasizes how many actual positive examples the classifier successfully identifies. A high recall is vital in scenarios where missing a positive case can have severe consequences. For example:

- Medical diagnoses - high recall ensures that patients with a condition are identified, minimizing false negatives that could delay treatment.
- Fraud detection - high recall ensures most fraudulent transactions are caught, reducing potential losses.

- Spam filtering - ensuring all spam emails are filtered (high recall) prevents unwanted messages from reaching the inbox.

For recall to achieve the ideal value of **1**, the number of false negatives (FN) must be zero. This means the model correctly identifies every actual positive case. However, focusing exclusively on recall can lead to increased false positives (FP), so recall is often balanced with precision using the F1-score.

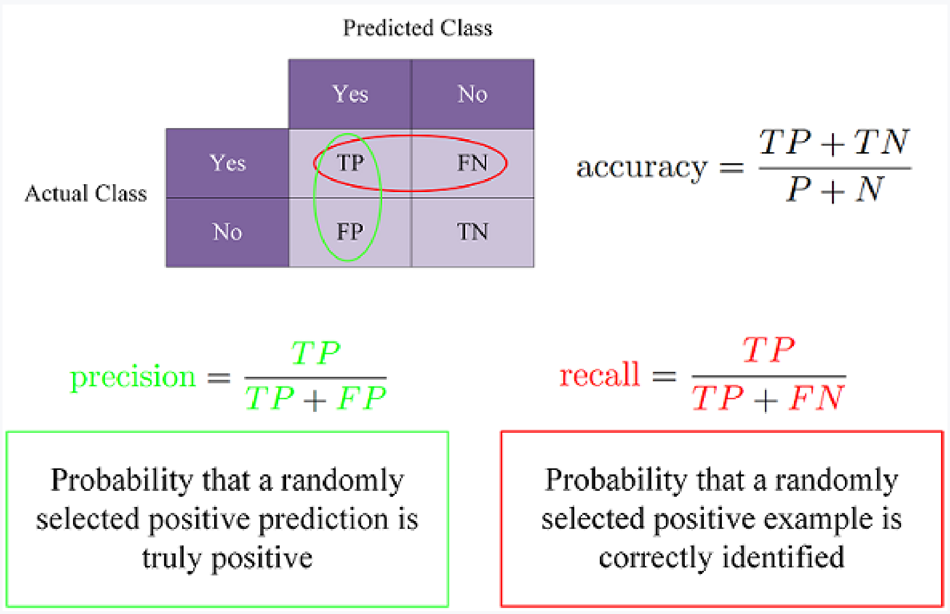
4.5.8

Which statements about recall are correct?

- Recall measures the ability of the model to detect all actual positive cases.
- Recall is calculated using true positives and false negatives.
- A high recall ensures that all predictions made by the model are correct.
- Recall is unrelated to false positives.

4.5.9

Review:


Predicted Class
Yes
No
Actual Class
Yes
No
TP
FN
FP
TN
$\text{accuracy} = \frac{TP+TN}{P+N}$
$\text{precision} = \frac{TP}{TP+FP}$
$\text{recall} = \frac{TP}{TP+FN}$
Probability that a randomly selected positive prediction is truly positive
Probability that a randomly selected positive example is correctly identified

4.5.10

Complete the formula

```
Precision = _____ / (_____)
```

- TP + TN
- TP + FP
- TN + FP
- TN
- FP
- TP

4.5.11

F1 score

The F1 score is a widely used performance metric in machine learning, especially in classification tasks where the balance between precision and recall is critical. It combines these two metrics into a single measure, providing a more comprehensive evaluation of the model.

The F1 score is calculated as the harmonic mean of precision and recall, given by:

```
F1_score = 2 * (Precission * Recall) / (Precission + Recall)
```

This metric ensures that both precision and recall are equally weighted. Unlike the arithmetic mean, the harmonic mean penalizes extreme values, making the F1 score sensitive to imbalances between precision and recall.

The F1 score is particularly useful in scenarios where:

- Imbalanced data - it balances the importance of false positives (FP) and false negatives (FN), which is essential when one class significantly outweighs the other.
- Evaluation trade-offs helps evaluate models where both precision and recall are important, such as medical diagnostics, spam detection, and fraud analysis.
- General model assessment - a high F1 score ensures the classifier is good at both detecting positive cases (high recall) and being precise in its predictions (high precision).

In an ideal classifier:

- **Precision = 1** - all positive predictions are correct (no false positives).
- **Recall = 1** - all actual positives are detected (no false negatives).

This results in an **F1 score = 1**, representing perfect balance and performance.

If precision or recall is significantly low, the F1 score will drop. For example:

- **High precision, low recall** indicates many positives are missed.
- **Low precision, high recall** indicates many negatives are wrongly classified as positives.

Thus, the F1 score is a better metric for overall performance than precision or recall alone.

4.5.12

What does the F1 score represent?

- The harmonic mean of precision and recall.
- The geometric mean of precision and recall.
- The arithmetic mean of precision and recall.
- The difference between precision and recall.

4.6 Practical examples

4.6.1

Project: Decision tree in action

We will demonstrate the creation of decision trees with a practical example in Python. In the practical example, we will try to create a decision tree model for loan prediction. We load the data for our model from the file uvery.csv using the Pandas library.

```
import pandas

loans =
pandas.read_csv('http://priscilla.fitped.eu/data/machine_learn
ing/loans.csv', sep=';')
```

With simple **head**() and **tail**() commands we can check our data. We can also use the **discribe**() method to display the basic statistics for our file. Since the file contains only categorical variables, the basic statistics will be very simple.

```
print("---------------------------")
print(loans.head())
print("---------------------------")
print(loans.tail())
print("---------------------------")
print(loans.describe())
```

Program output:

```
---------------------------
  Client Income Account  Gender Unemployed Credit
0     c1   high    high  female         no    yes
1     c2   high    high    male         no    yes
2     c3    low     low    male         no     no
3     c4    low    high  female        yes    yes
4     c5    low    high    male        yes    yes
---------------------------
   Client Income Account  Gender Unemployed Credit
7      c8   high     low  female        yes    yes
8      c9    low  medium    male        yes     no
9     c10   high  medium  female         no    yes
10    c11    low  medium  female        yes     no
11    c12    low  medium    male         no    yes
---------------------------
       Client Income Account  Gender Unemployed Credit
count      12     12      12      12         12     12
unique     12      2       3       2          2      2
top        c1    low    high  female         no    yes
freq        1      7       4       6          6      8
```

From the above results, it is easy to see that our set contains 12 examples. The target variable that our model will predict is the variable **Loan** with possible values **Yes** and **No.**

We will create the decision tree model using the **scikit-learn** library. It is one of the most widely used libraries for machine learning. However, in the case of decision tree models, this library cannot handle categorical variables. For this reason, we need to convert the categorical variables to numerical variables in our dataset.

By quick reasoning, a function can be created to convert categorical variables. In our example, we convert a categorical variable feature (i.e., a column in pandas) **Income**.

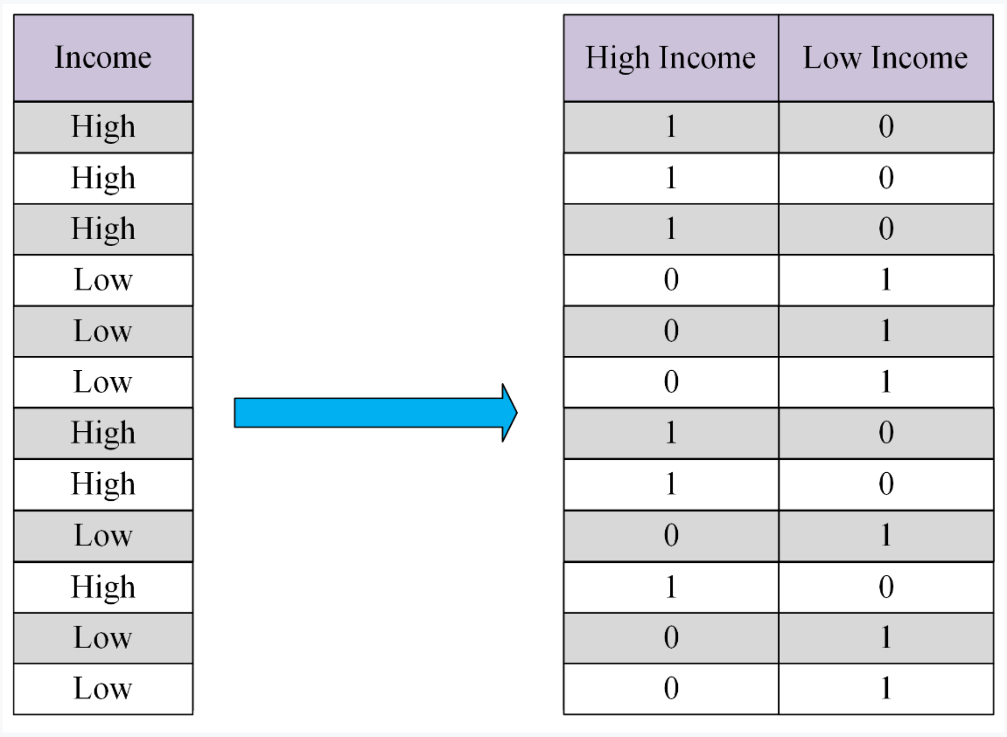
```
loans["Income_int"] = loans["Income"]
def cat2int(column):
    vals = list(set(column))
    for i, string in enumerate(column):
        column[i] = vals.index(string)
    return column

cat2int(loans['Income_int'])

print(loans.head())
```

Program output:

```
  Client Income Account  Gender Unemployed Credit Income_int
0     c1   high    high  female         no    yes          0
1     c2   high    high    male         no    yes          0
2     c3    low     low    male         no     no          1
3     c4    low    high  female        yes    yes          1
4     c5    low    high    male        yes    yes          1
```

Note the new feature **Income_int**. Its interpretation is easy as long as we also have the **Income** feature. However, we need to be aware of several shortcomings of this approach. The first shortcoming is that this conversion does not always set low income to 0, high income to 1. If we already consider multiple categorical values, e.g. slightly higher, medium, very low, etc. the clarity of the numerical values may be unclear. Especially, if we do not see the original **Income** column.

For these reasons, so-called dummies are used to convert categorical variables into numerical variables. Dummies create a new feature (column) for each value of a categorical variable. For the **Income** feature, the dummies will look as follows:



Income
High
High
High
Low
Low
Low
High
High
Low
High
Low
Low

High Income	Low Income
1	0
1	0
1	0
0	1
0	1
0	1
1	0
1	0
0	1
1	0
0	1
0	1

In Python, we can create dummies by simply calling the appropriate method.

```
loans =
pandas.get_dummies(loans,columns=["Income"],drop_first=False)

print(loans.head())
```

Program output:

```
  Client Account  Gender Unemployed Credit Income_int
Income_high  Income_low
0      c1     high  female         no     yes         0
True         False
1      c2     high    male         no     yes         0
True         False
2      c3      low    male         no      no         1
False         True
3      c4     high  female        yes     yes         1
False         True
4      c5     high    male        yes     yes         1
False         True
```

4.6.2

We can now use the previous information about categorical variables and dummies to create sample code where we load our dataset and transfer all the necessary features using dummies.

```
import pandas
loans =
pandas.read_csv('http://priscilla.fitped.eu/data/machine_learn
ing/loans.csv', sep=';')

print("---------------------------")
print(loans.head())

loans =
pandas.get_dummies(loans,columns=["Income"],drop_first=False)
loans=
pandas.get_dummies(loans,columns=["Account"],drop_first=False)
loans=
pandas.get_dummies(loans,columns=["Gender"],drop_first=False)
loans=
pandas.get_dummies(loans,columns=["Unemployed"],drop_first=Fal
se)
print("---------------------------")
print("Dataset after dummies:")
print("---------------------------")
print(loans.head())
```

Program output:

```
---------------------------
  Client Income Account  Gender Unemployed Credit
0     c1   high    high  female         no    yes
1     c2   high    high    male         no    yes
2     c3    low     low    male         no     no
3     c4    low    high  female        yes    yes
4     c5    low    high    male        yes    yes
---------------------------
Dataset after dummies:
---------------------------
  Client Credit  Income_high  Income_low  Account_high
Account_low  \
0     c1    yes         True       False          True
False
```

```
1      c2     yes          True         False          True
False
2      c3      no         False          True         False
True
3      c4     yes         False          True          True
False
4      c5     yes         False          True          True
False

   Account_medium  Gender_female  Gender_male  Unemployed_no
Unemployed_yes
0            False           True        False           True
False
1            False          False         True           True
False
2            False          False         True           True
False
3            False           True        False          False
True
4            False          False         True          False
True
```

Note, that after applying dummies, we not only created new features, but we also deleted the original features.

An interesting and often used method is the **Counters** method. This gives us a quick look at the distribution of values in each feature. For example, if we want to know the number of 0 and 1 values for the variable **Income_High**, we can use **Counters.**

```
from collections import Counter
print(Counter(loans.Income_high))
```

Program output:

```
Counter({False: 7, True: 5})
```

Use the **Counters** function to see how many **Yes** and **No** values are in the **Loan** feature.

```
from collections import Counter
## fill your code
print(Counter(___))
```

Program output:

```
Counter()
```

- 8 values **Yes** and 4 values **No**
- 4 values **Yes** and 8 values **No**
- 0 values **Yes** and 1 values **No**

4.6.3

Project: Go to play?

Analyze the given weather dataset and build a decision tree classification model to predict whether to "Play" or not, based on the weather conditions.

- Dataset: https://priscilla.fitped.eu/data/machine_learning/golf_nominal.csv

1. Load and pre-process the dataset

- The code snippet transforms **categorical columns** in the golf DataFrame into one-hot encoded columns using the pandas.get_dummies() function. E.g. Converts the categorical values in the **Outlook** column into separate binary (0/1) columns for each category (e.g., **Outlook_overcast**, **Outlook_rainy**, **Outlook_sunny**).

```
import pandas
from collections import Counter
from sklearn.tree import DecisionTreeClassifier # Import
Decision Tree Classifier
from sklearn.model_selection import train_test_split # Import
train_test_split function
from sklearn import metrics #Import scikit-learn metrics
module for accuracy calculation

golf =
pandas.read_csv('https://priscilla.fitped.eu/data/machine_lear
ning/golf_nominal.csv', sep=';')
# Create columns with categorical values
golf =
pandas.get_dummies(golf,columns=["Outlook"],drop_first=False)
```

```
golf =
pandas.get_dummies(golf,columns=["Temperature"],drop_first=Fal
se)
golf =
pandas.get_dummies(golf,columns=["Humidity"],drop_first=False)
golf =
pandas.get_dummies(golf,columns=["Windy"],drop_first=False)

print(golf.head())
```

Program output:

```
  Play  Outlook_overcast  Outlook_rainy  Outlook_sunny
Temperature_cool  \
0  yes              True          False          False
False
1  yes              True          False          False
True
2  yes              True          False          False
False
3  yes              True          False          False
False
4  yes             False           True          False
False

   Temperature_hot  Temperature_mild  Humidity_high
Humidity_normal  \
0             True             False           True
False
1            False             False          False
True
2            False              True           True
False
3             True             False          False
True
4            False              True           True
False

   Windy_False  Windy_True
0         True       False
1        False        True
2        False        True
3         True       False
4         True       False
```

- **golf.columns.difference(['Play'])** generates a list of column names excluding Play.
- **golf[...]** selects all columns from the golf DataFrame except the **Play** column.
- This creates the feature set **X**, which includes all the one-hot encoded columns for **Outlook, Temperature, Humidity, and Windy**.
- Creates **DecisionTreeClassifier()** and train model
- At the end print evaluation result - accuracy score

```
X = golf[golf.columns.difference(['Play'])]
y = golf.Play

X_train, X_test, y_train, y_test = train_test_split(X, y, 
test_size=0.5) # 50% training and 50% test

# Create Decision Tree classifer object
clf = DecisionTreeClassifier()

# Train Decision Tree Classifer
clf = clf.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
print(Counter(y_test))
```

Program output:

```
Accuracy: 0.5714285714285714
Counter({'yes': 4, 'no': 3})
```

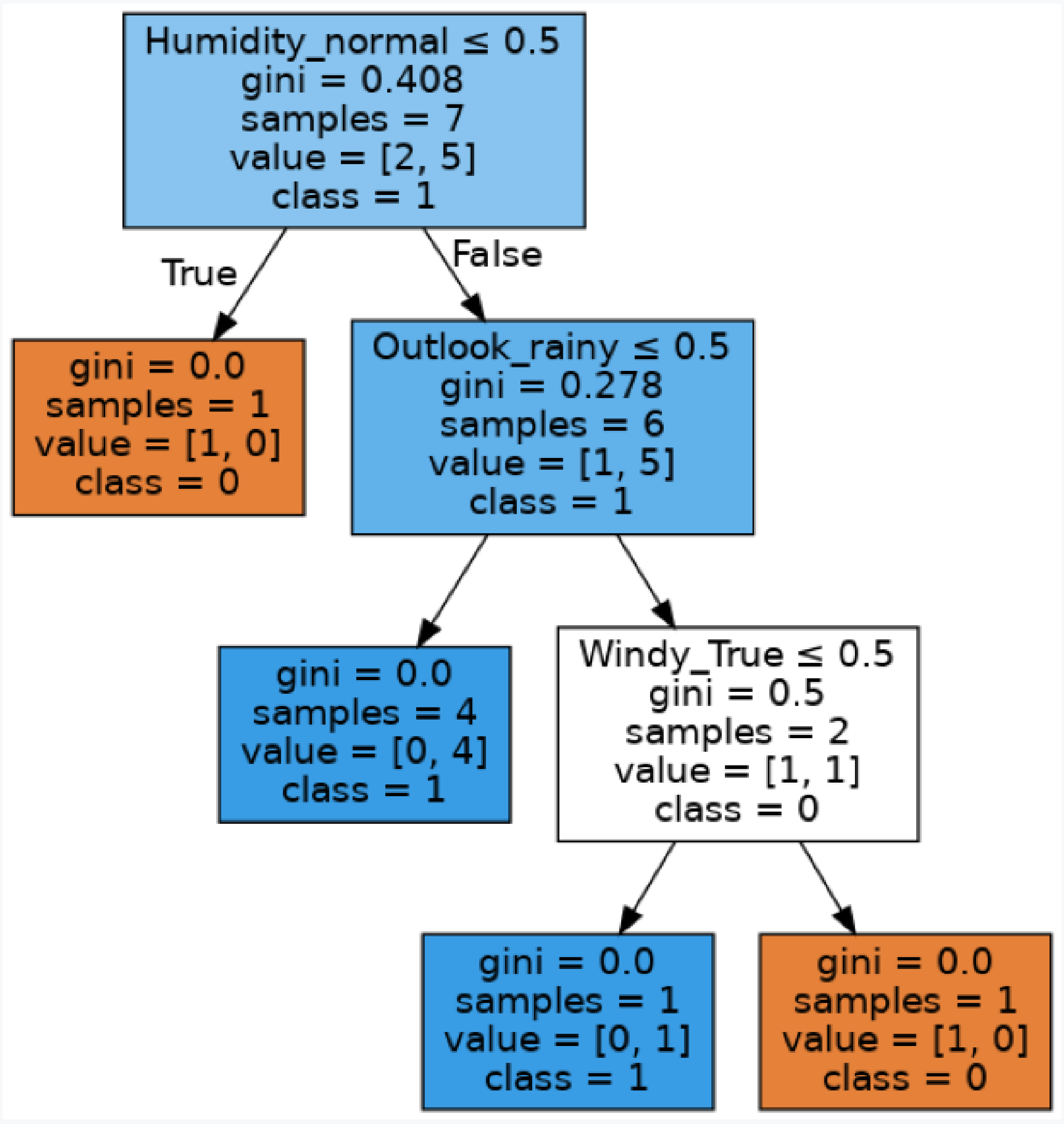
And shows decision tree structure to check the process

```
import matplotlib.pyplot as plt
from sklearn.tree import export_graphviz
from io import BytesIO
from IPython.display import display
import pydotplus
from collections import Counter
from sklearn import metrics
from PIL import Image as PILImage
import numpy as np
```

```
# Assuming clf, X_train, y_test, and y_pred are already
defined.
print("Accuracy:", metrics.accuracy_score(y_test, y_pred))
print(Counter(y_test))
cols = X_train.columns

# Generate DOT data
dot_data = StringIO()
export_graphviz(
    clf,
    out_file=dot_data,
    filled=True,
    rounded=False,
    special_characters=True,
    feature_names=cols,
    class_names=['0', '1']
)

# Create graph from DOT data
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())

# Render the PNG image to memory
png_image = graph.create_png()

# Load the PNG image into PIL for visualization
image = PILImage.open(BytesIO(png_image))

# Convert to a NumPy array and display with matplotlib
fig, ax = plt.subplots(figsize=(12, 12))
ax.imshow(np.array(image))
ax.axis('off')  # Turn off axes for cleaner output
plt.show()
```

Program output:

```
Accuracy: 0.5714285714285714
Counter({'yes': 4, 'no': 3})
```



4.6.4

Project: Titanic survival

In this project, we will guide you through the creation of a simple decision tree.

The example shows a decision on whether or not a person will survive on the Titanic, based on that person's features (characteristics).

- We will use the Titanic dataset. Local version: https://priscilla.fitped.eu/data/pandas/titanic.csv
- A description of the individual columns is available at https://data.world/nrippner/titanic-disaster-dataset

```
import pandas as pd

data =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/titanic.c
sv')
print(data)
```

Program output:

```
     PassengerId  Survived  Pclass  \
0              1         0       3
1              2         1       1
2              3         1       3
3              4         1       1
4              5         0       3
..           ...       ...     ...
886          887         0       2
887          888         1       1
888          889         0       3
889          890         1       1
890          891         0       3

                                                  Name     Sex
Age  SibSp  \
0                              Braund, Mr. Owen Harris    male
22.0      1
1    Cumings, Mrs. John Bradley (Florence Briggs Th...  female
38.0      1
2                               Heikkinen, Miss. Laina  female
26.0      0
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)  female
35.0      1
4                             Allen, Mr. William Henry    male
35.0      0
..                                                 ...     ...
...    ...
886                              Montvila, Rev. Juozas    male
27.0      0
887                       Graham, Miss. Margaret Edith  female
19.0      0
888           Johnston, Miss. Catherine Helen "Carrie"  female
NaN      1
889                              Behr, Mr. Karl Howell    male
26.0      0
```

```
890                                Dooley, Mr. Patrick    male
32.0      0

     Parch            Ticket     Fare Cabin Embarked
0        0         A/5 21171   7.2500   NaN        S
1        0          PC 17599  71.2833   C85        C
2        0  STON/O2. 3101282   7.9250   NaN        S
3        0            113803  53.1000  C123        S
4        0            373450   8.0500   NaN        S
..     ...               ...      ...   ...      ...
886      0            211536  13.0000   NaN        S
887      0            112053  30.0000   B42        S
888      2        W./C. 6607  23.4500   NaN        S
889      0            111369  30.0000  C148        C
890      0            370376   7.7500   NaN        Q

[891 rows x 12 columns]
```

To analyze whether a person would have survived the Titanic, we first decide which features from the dataset to use for classification. After reviewing the available data, we select the following features:

- **Pclass - p**assenger class (1st, 2nd, or 3rd class)
- **Sex**- gender of the passenger
- **Age** of the passenger
- **SibSp** - number of siblings or spouses traveling with the passenger
- **Parch** - number of parents or children traveling with the passenger
- **Embarked** - port of boarding (C = Cherbourg, Q = Queenstown, S = Southampton)

Our target variable for classification will be the **Survived** column, which indicates whether the passenger survived or not.

```
data = data[['Survived', 'Pclass', 'Sex', 'Age', 'SibSp',
'Parch', 'Embarked']]
print(data)
```

Program output:

```
     Survived  Pclass     Sex   Age  SibSp  Parch Embarked
0           0       3    male  22.0      1      0        S
1           1       1  female  38.0      1      0        C
2           1       3  female  26.0      0      0        S
```

```
3            1       1  female  35.0      1      0        S
4            0       3    male  35.0      0      0        S
..         ...     ...     ...   ...    ...    ...      ...
886          0       2    male  27.0      0      0        S
887          1       1  female  19.0      0      0        S
888          0       3  female   NaN      1      2        S
889          1       1    male  26.0      0      0        C
890          0       3    male  32.0      0      0        Q

[891 rows x 7 columns]
```

- This dataset has several null values (they are marked as NaN). We first delete them by removing those rows that contain such values.

```
data = data.dropna()
print(data)
```

Program output:

```
     Survived  Pclass     Sex   Age  SibSp  Parch Embarked
0           0       3    male  22.0      1      0        S
1           1       1  female  38.0      1      0        C
2           1       3  female  26.0      0      0        S
3           1       1  female  35.0      1      0        S
4           0       3    male  35.0      0      0        S
..        ...     ...     ...   ...    ...    ...      ...
885         0       3  female  39.0      0      5        Q
886         0       2    male  27.0      0      0        S
887         1       1  female  19.0      0      0        S
889         1       1    male  26.0      0      0        C
890         0       3    male  32.0      0      0        Q

[712 rows x 7 columns]
```

Note, that after removing the null values out of the original 891 records, only 712 records left.

If we want to preserve the number of records, we can apply other methods to deal with null values, e.g. replace them with (substitution).

- The **Embarked** column should be binarized. This is done using the **get_dummies** function.
- We also change the gender values male to 0 and female to 1.

```
data =
pd.get_dummies(data,columns=["Embarked"],drop_first=False)

data['Sex'] = data['Sex'].replace({'male': 0, 'female': 1})

print(data)
```

Program output:

```
     Survived  Pclass  Sex   Age  SibSp  Parch  Embarked_C
Embarked_Q  \
0           0       3    0  22.0      1      0       False
False
1           1       1    1  38.0      1      0        True
False
2           1       3    1  26.0      0      0       False
False
3           1       1    1  35.0      1      0       False
False
4           0       3    0  35.0      0      0       False
False
..        ...     ...  ...   ...    ...    ...         ...
...
885         0       3    1  39.0      0      5       False
True
886         0       2    0  27.0      0      0       False
False
887         1       1    1  19.0      0      0       False
False
889         1       1    0  26.0      0      0        True
False
890         0       3    0  32.0      0      0       False
True

     Embarked_S
0          True
1         False
2          True
3          True
4          True
..          ...
885       False
886        True
887        True
889       False
```

```
890       False

[712 rows x 9 columns]
```

Data are prepared, now we can proceed to split it into features and target, and also into training and testing in a ratio of 80:20.

We have added randomization to the **train_test_split** function, which will always guarantee a deterministic distribution.

```
X = data[data.columns.difference(['Survived'])]
y = data['Survived']

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

The data is prepared; we can create a decision tree model that will be trained.

We will also use randomization, to achieve always the same tree.

```
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier(random_state=42)
clf = clf.fit(X_train, y_train)
print(X_train)
```

Program output:

```
      Age  Embarked_C  Embarked_Q  Embarked_S  Parch  Pclass
Sex  SibSp
472  33.0       False       False        True      2       2
1      1
432  42.0       False       False        True      0       2
1      1
666  25.0       False       False        True      0       2
0      0
30   40.0        True       False       False      0       1
0      0
291  19.0        True       False       False      0       1
1      1
..    ...         ...         ...         ...    ...     ...
...    ...
```

```
93   26.0        False        False        True      2       3
0      1
135  23.0         True        False       False      0       2
0      0
338  45.0        False        False        True      0       3
0      0
549   8.0        False        False        True      1       2
0      1
131  20.0        False        False        True      0       3
0      0

[569 rows x 8 columns]
```

We have a decision tree model stored in the variable clf. Using the following line of code, we obtain a prediction for the test data, which we also display.

```
y_pred = clf.predict(X_test)
print(y_pred)
```

Program output:

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

It remains to find out the classification metrics. First, we find out the accuracy – i.e. we compare the predicted values and the actual survival values attributed to the test data.

```
from sklearn import metrics
acc = metrics.accuracy_score(y_test, y_pred)
print("Accuracy:", acc)
```

Program output:

```
Accuracy: 0.7342657342657343
```

Precision, recall a f1 score will be calculated similarly.

```
from sklearn import metrics
prec = metrics.precision_score(y_test, y_pred)
print("Precision:", prec)

from sklearn import metrics
rec = metrics.recall_score(y_test, y_pred)
print("Recall:", rec)

from sklearn import metrics
f1 = metrics.f1_score(y_test, y_pred)
print("F1 score:", f1)
```

Program output:

```
Precision: 0.7450980392156863
Recall: 0.6031746031746031
F1 score: 0.6666666666666666
```

Here's how the two examples would look as input for a **DecisionTreeClassifier** in Python:

```
import pandas as pd

# Assuming these are the feature names from the training data
feature_names = ['Age', 'Embarked_C', 'Embarked_Q',
'Embarked_S', 'Parch', 'Pclass','Sex', 'SibSp']

# Create a DataFrame for the test data
test_data = pd.DataFrame(
    [[29, 1, 0, 0, 0, 1, 0, 0],  # Example 1
     [45, 0, 1, 1, 1, 2, 1, 0]], # Example 2
    columns=feature_names
)

# Predict survival
predictions = clf.predict(test_data)
print(predictions)
```

Program output:

```
[0 1]
```

4.6.5

What is the correct code to create and train a decision tree?

```
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTree()
clf = clf.train(X_train, y_train)
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTree()
clf = clf.fit(X_train, y_train)
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier()
clf = clf.fit(X_train, y_train)
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier()
clf = clf.train(X_train, y_train)
```

4.6.6 Predicted values

Complete the code to find out what values the model returns for test data that represents 30% of the total available data.

file1.py

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

data =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/titanic.c
sv')

data = data[['Survived', 'Pclass', 'Sex', 'Age', 'SibSp',
'Parch', 'Embarked']]

data = data.dropna()

data =
pd.get_dummies(data,columns=["Embarked"],drop_first=False)

data['Sex'] = data['Sex'].replace({'male': 0, 'female': 1})
```

```
X = data[data.columns.difference(['Survived'])]
y = data['Survived']

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

clf = clf.fit(X_train, y_train)
```

4.6.7

How do we calculate accuracy?

```
acc = accuracy(y_true, y_pred)
acc = accuracy_score(y_true, y_pred)
acc = acc(y_true, y_pred)
acc = acc_score(y_true, y_pred)
```

4.6.8

Complete the calculation and listing of recall.

```
_____ sklearn import _____
rec = metrics._____(y_test, _____)
print("Recall:", _____ )
```

- recall_score
- metrics
- rec
- y_pred
- from

Tree Construction

Chapter 5

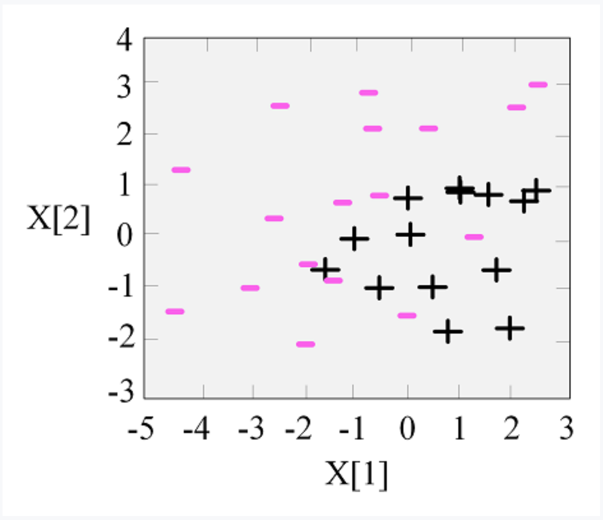
5.1 Overfitting

🕮 5.1.1

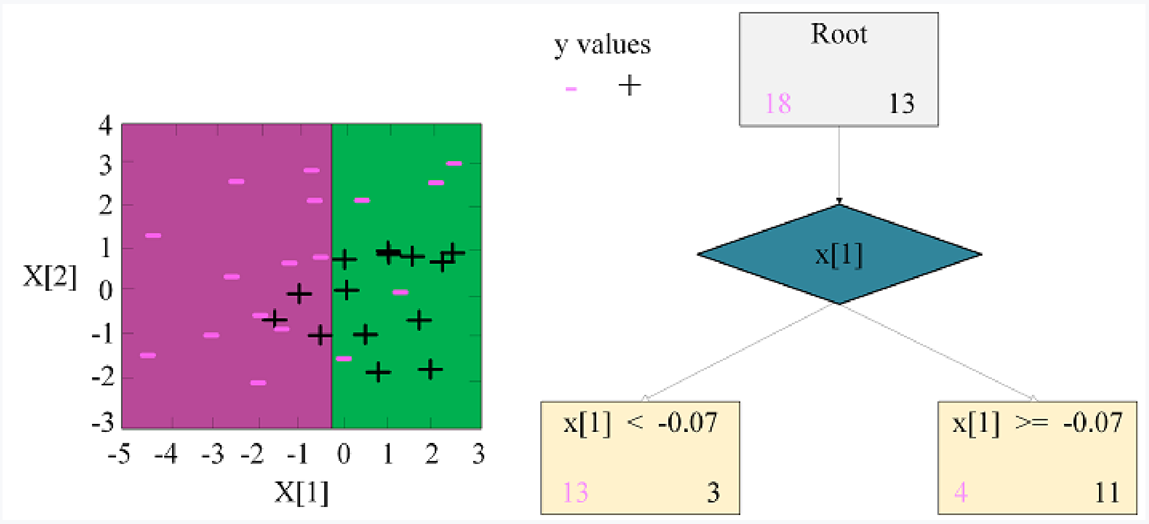
When creating the decision tree, we select individual features and sort the examples one by one according to the features. In this way, we reduce the so-called training error. Recall that the training error is computed as:

```
Classification Error = Number of incorrect classifications /
Total number of examples
```

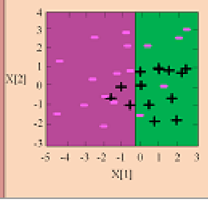
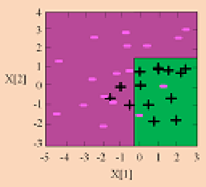
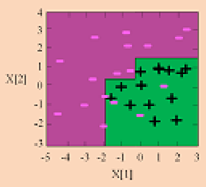
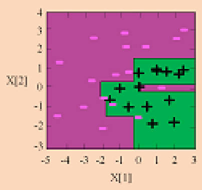
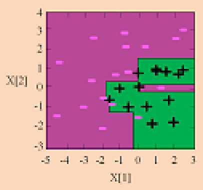
By selecting a feature for the distribution, we reduce this error. Consider only two possible features x[1] and x[2] , two classes + and - splitting as on the following picture.



By selecting the x[1] feature and choosing the threshold correctly, we can divide the examples as follows:

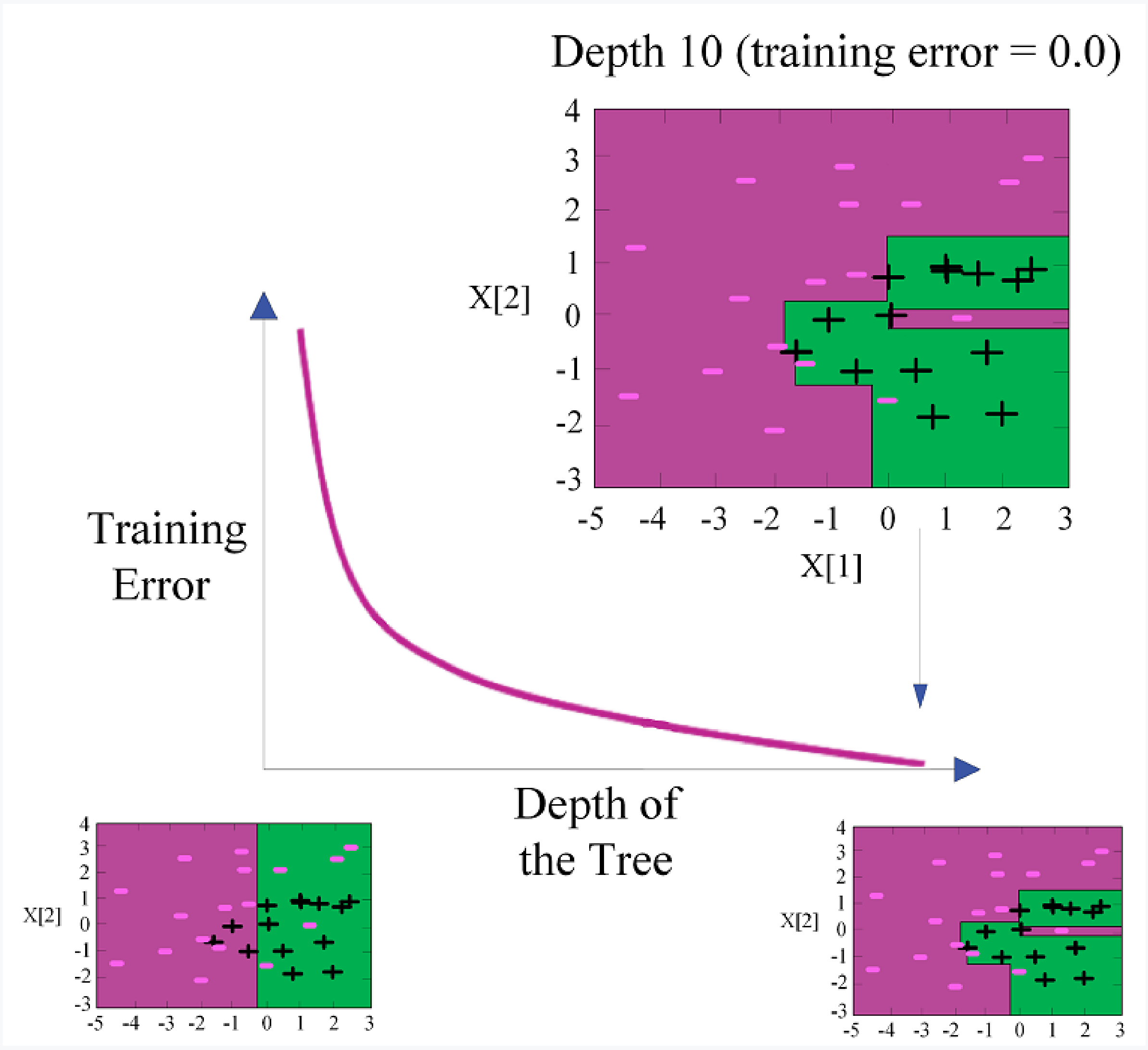


If we continued to build the tree further, we would actually increase its depth. For example, we would select x[2] feature for the next division. The question remains whether we could select x[1] and x[2] features again. In the case of numerical values, we can, but with a different threshold value. By adding more depths, we would be able to create a tree with an error equal to 0 in this example.

Tree depth	depth = 1	depth = 2	depth = 3	depth = 5	depth = 10
Trainig error	0.22	0.13	0.10	0.03	0.00
Decision boundary					

However, it is questionable whether such a deep tree, even with a classification error equal to 0, is really a good machine learning model.

From the above example, it is clear that the classification error, in our case the training error, decreases with the depth of the tree, i.e., deeper tree = lower training error.



5.1.2

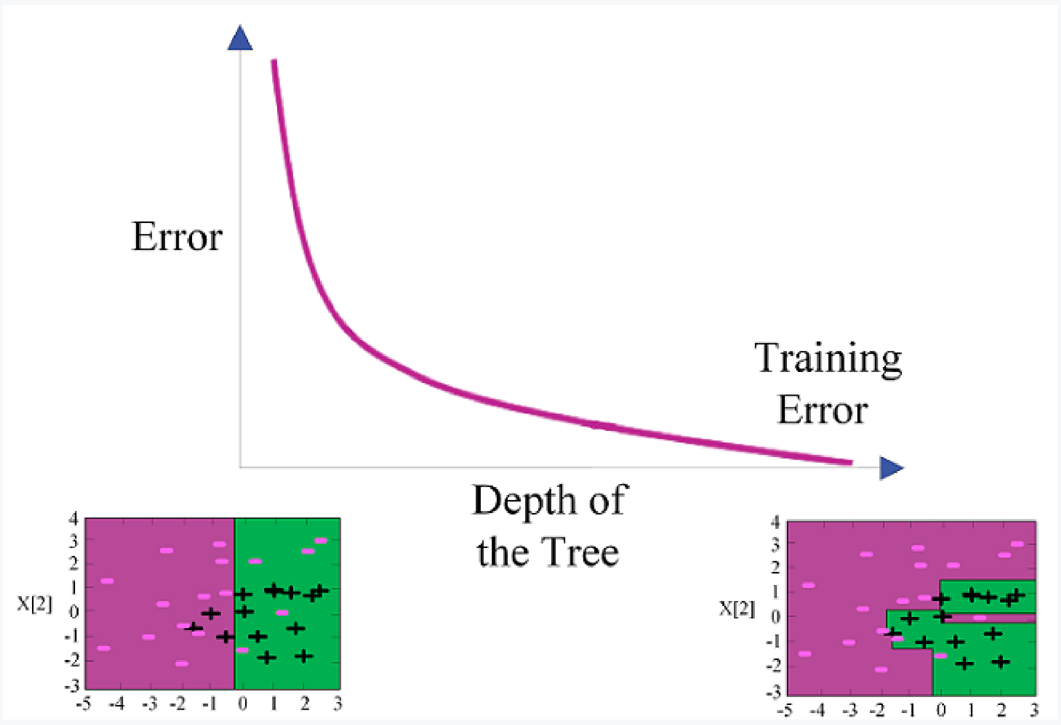
Which of the following statements about decision tree depth and training error is correct?

- Increasing the depth of the decision tree decreases the training error.
- A deeper decision tree always leads to better generalization.
- Features cannot be reused in a decision tree.
- A decision tree with zero training error is always the best model.

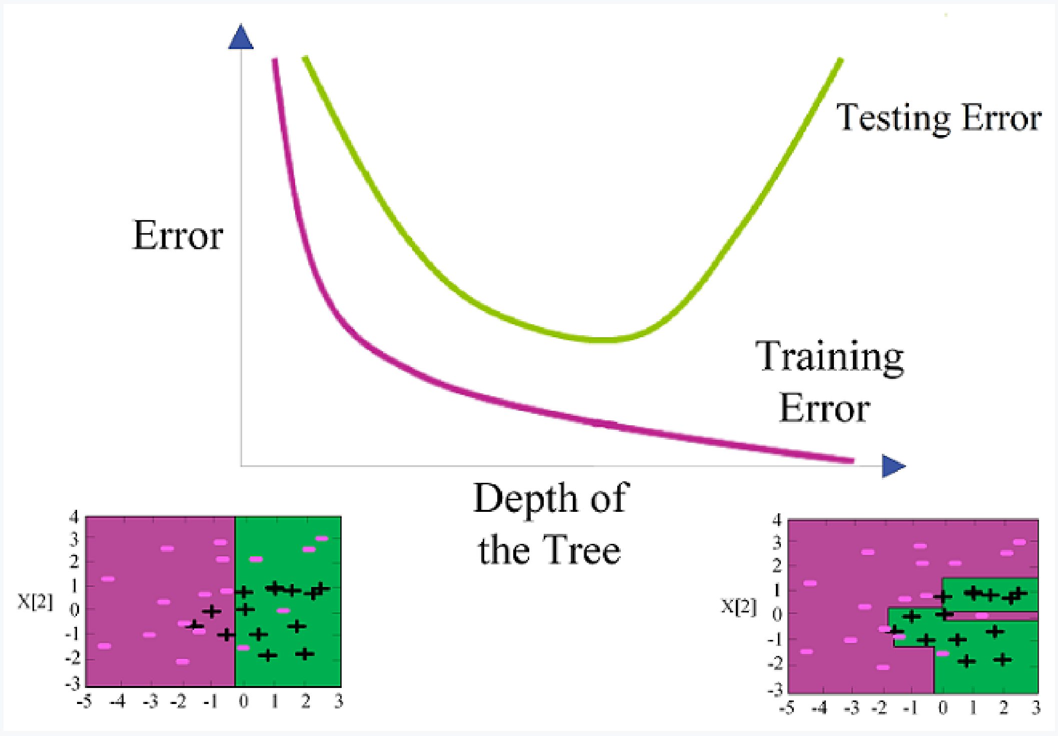
5.1.3

Overfitting

In the previous example, we found that the classification error decreases during training.



But we know that the model we have created always needs to be evaluated. This validation is done using examples that were not included in the creation of the model. In the previous chapters, we talked about the training set and the test set, whereby a model is built on the data from the training set and this model is validated using examples from the test set. If we validate the generated trees of available depths using test examples, we find that the error on the test examples will initially decrease, but at a certain model complexity this test error will increase.



We call this phenomenon overfitting. You will also find terms such as adaptation, exact copying or remodelling in the literature. Overfitting means that the model estimates the training examples perfectly, but it can no longer generalize and it estimates very poorly the examples other than the training ones.

5.1.4

What is the phenomenon called when a model fits the training data perfectly but performs poorly on unseen data?

- Overfitting
- Underfitting
- Generalization
- Regularization

5.1.5

Occam's razor

When building a machine learning model, one common issue is overfitting, where the model performs perfectly on the training data but struggles with new, unseen data. This occurs when the model becomes too complex, capturing not just the true patterns but also noise in the data. In such cases, even if the training error is zero, the model may not generalize well, leading to poor performance on the test set.

Therefore, the goal is not to minimize training error, but rather to minimize the **testing error**, which reflects the model's ability to generalize to new examples.

A key principle that guides the creation of simpler and more effective models is **Occam's razor**. This principle originates from William Occam, a philosopher and logician who lived in the 14th century. Occam's Razor suggests that when faced with multiple explanations for a phenomenon, the simplest one should be preferred. In the context of decision trees, this principle advises that among models with similar performance (i.e., similar classification errors), the simpler model should be chosen. A simpler tree is not only easier to interpret and understand, but it is also less likely to overfit the training data, leading to better generalization.

In practice, applying Occam's Razor means that when two decision trees have similar test errors, the one with fewer branches or depths should be selected. This approach balances model simplicity with performance, avoiding unnecessary complexity while still achieving good predictive accuracy. By following Occam's Razor, we create models that are easier to explain and less prone to overfitting, ultimately leading to better results on new data.

5.1.6

What is the main goal when creating a machine learning model?

- Minimize testing error
- Minimize training error
- Maximize model complexity
- Maximize training error

5.1.7

Which of the following is an interpretation of Occam's razor when applied to decision trees?

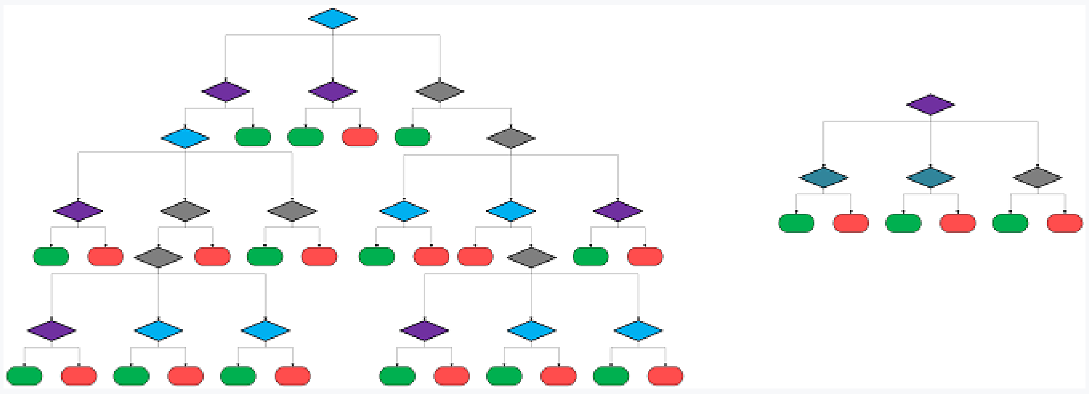
- The simpler decision tree should be chosen if it has similar performance to a more complex one
- If two trees have similar performance, the one with more branches should be chosen
- A complex tree is always better than a simple one
- Occam's Razor suggests choosing the more complex model

5.1.8

Simplest tree

When comparing decision trees, the question often arises: how can we determine which tree is the simplest? While visually, the simplest tree may appear to have fewer branches or nodes, it is important to note that for an algorithm, this visual evaluation is much more complex. In reality, when we compare decision trees, they often look very similar, making it difficult to judge which one is simpler based solely on appearance.

To make this comparison, we need a **numerical representation** of the complexity of the tree. The simplest tree is typically the one that minimizes complexity while still achieving a low testing error. Various metrics can be used to quantify the complexity of a decision tree. One common approach is to count the number of nodes and branches in the tree, but this is not always sufficient, especially when the trees are visually similar.



A more effective approach is to use **metrics like tree depth** and **node purity**. The depth of a tree refers to how many levels (or splits) it has, while node purity measures how homogenous the data is in each node. A tree with fewer levels and higher purity in each node can be considered simpler. Another approach is to use the **cost-complexity pruning** technique, which allows for the pruning of branches that do not significantly improve the tree's performance on the test set, effectively simplifying the tree without increasing the error rate.

Ultimately, numerical metrics give us a way to objectively compare decision trees and choose the one that strikes the best balance between simplicity and performance.

5.1.9

Which of the following metrics can be used to evaluate the complexity of a decision tree?

- Tree depth
- Number of nodes and branches
- Number of features in the dataset
- Accuracy on the training set

5.2 Tree building

5.2.1

When building decision trees, the goal is to create a model that strikes the right balance between complexity and accuracy. To achieve this, there are several methods for constructing an ideal decision tree, ensuring it has acceptable error while remaining simple enough to be interpretable and generalizable. These methods generally fall into two categories: **early stopping** and **pruning**.

Early stopping is a technique where the learning algorithm is stopped before the decision tree becomes too complex. The algorithm typically builds a tree by recursively splitting the data based on the features. However, if the tree is allowed to grow without any restrictions, it may become overly complex, leading to overfitting. Early stopping helps to prevent this by halting the tree-building process before the tree grows too deep or starts to capture too much noise in the data. This method relies on setting parameters such as maximum depth or minimum samples per leaf, limiting the tree's complexity from the start.

Pruning involves simplifying a tree after it has already been built. During the tree construction, the model may have created branches that do not contribute significantly to its accuracy. Pruning removes or "cuts back" these branches to reduce the tree's complexity while retaining as much accuracy as possible. This can be done using techniques like **cost-complexity pruning** or **post-pruning**, where the tree is evaluated on the test data, and unimportant branches are pruned to minimize the test error. Pruning helps to improve the generalization of the model, making it less likely to overfit the training data.

5.2.2

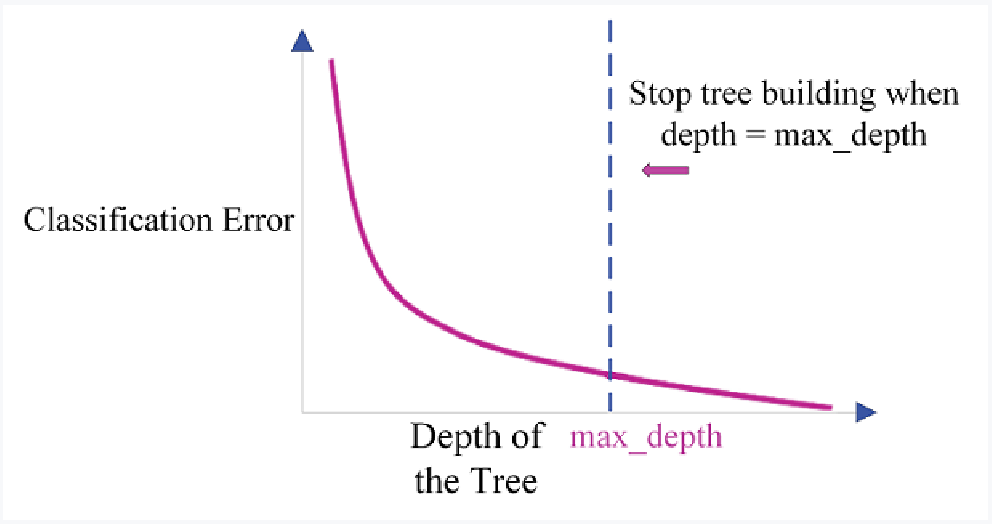
If we want to simplify the tree after its creation, this method calls:

- Pruning
- Early stopping
- Cross-validation
- Regularization

5.2.3

Maximum depth

When using **early stopping** to control the complexity of a decision tree, one of the simplest and most common conditions is to set a **maximum depth** for the tree. The maximum depth refers to the number of levels or splits the tree can have from the root node to the deepest leaf. By limiting the depth, we can prevent the tree from growing too large and complex, making it easier to interpret and reducing the risk of overfitting.

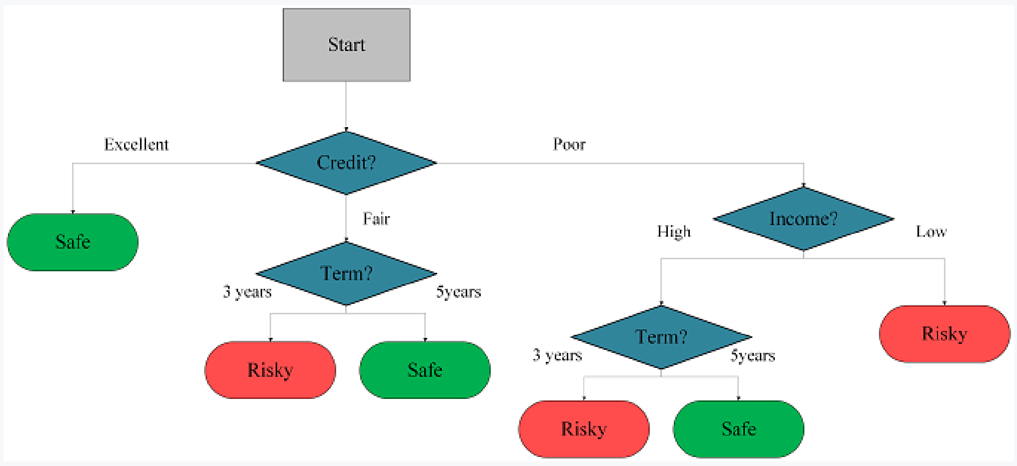


Setting the maximum depth is particularly useful because it ensures that the tree won't continue to split until it perfectly fits the training data. This helps to create a simpler model that is easier to visualize and understand. However, the challenge lies in determining the **appropriate depth**. If the depth is too shallow, the tree may underfit the data, meaning it won't capture enough complexity to make accurate predictions. On the other hand, if the depth is too deep, the tree may overfit the training data, capturing noise and making it less effective on unseen data.

To find the right balance, practitioners typically experiment with different depths and evaluate the model's performance on both the training and test sets. Cross-validation is often used to select the best depth by finding the value that minimizes the test error. Ultimately, the goal is to select a depth that maintains model simplicity without sacrificing too much accuracy.

5.2.4

What is the depth of the tree in the picture?



- tree depth = 3
- tree depth = 6
- tree depth = 1

5.2.5

What does setting the maximum depth of a decision tree do?

- It limits how many times the tree can split
- It increases the complexity of the tree
- It guarantees zero training error
- It prevents the tree from overfitting

5.2.6

Which of the following is a potential issue with setting the maximum depth of a decision tree?

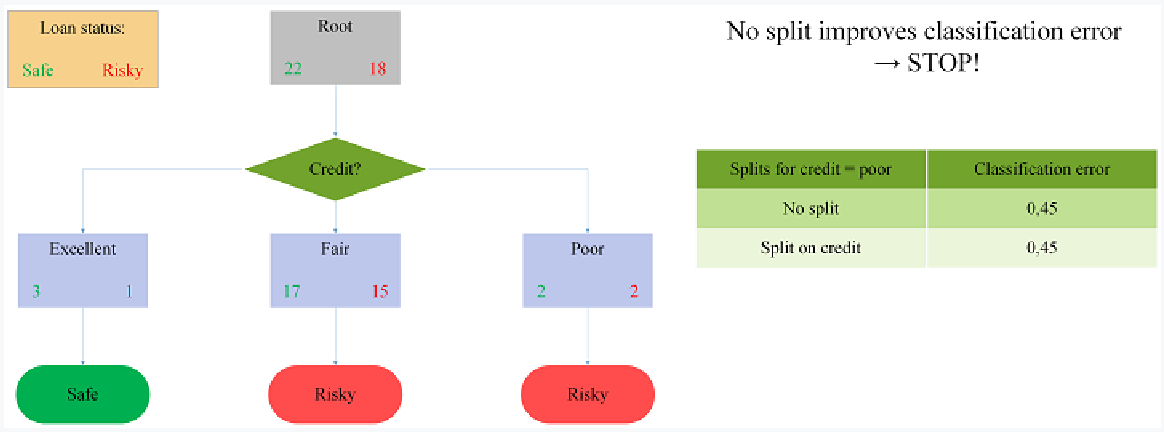
- If the depth is too shallow, the tree may underfit
- Setting the depth too high leads to simpler trees
- A deeper tree reduces interpretability
- A deeper tree guarantees better generalization

📖 5.2.7

Early stopping

When building decision trees, selecting the appropriate **maximum depth** is often challenging. Setting the depth too high or too low can impact the tree's ability to generalize well. One way to address this issue is to use a more dynamic approach for stopping tree growth: **monitoring the classification error** at each possible node expansion.

In this method, we check the classification error after every possible split in the tree. If the error does not decrease or decreases very little, it is an indication that further expansion may not improve the model's performance. This strategy helps avoid unnecessary complexity and prevents the model from overfitting the data. If the classification error stagnates or worsens after a split, it is wise to stop building the tree at that point, rather than continuing to add branches that will not contribute to improved performance.



Splits for credit = poor	Classification error
No split	0,45
Split on credit	0,45

In the example in the figure we have a tree with depth 1. Its classification error is (1 + 15 + 2) / 40 = 0.45

If we were to split the tree according to the **Credit** feature, the classification error would be 18 / (22 + 18) = 0.45

Under the early stopping condition, we would no longer continue generating the tree because the classification error would not decrease. This is an example of **early stopping** based on error monitoring, where we stop further expansion when additional splits do not contribute to improved classification accuracy.

5.2.8

What is the main idea behind early stopping based on classification error?

- Stopping tree expansion when the classification error stops decreasing
- Continuing to build the tree until maximum depth is reached
- Evaluating the tree on the test set before expanding
- Allowing the tree to grow as deep as possible

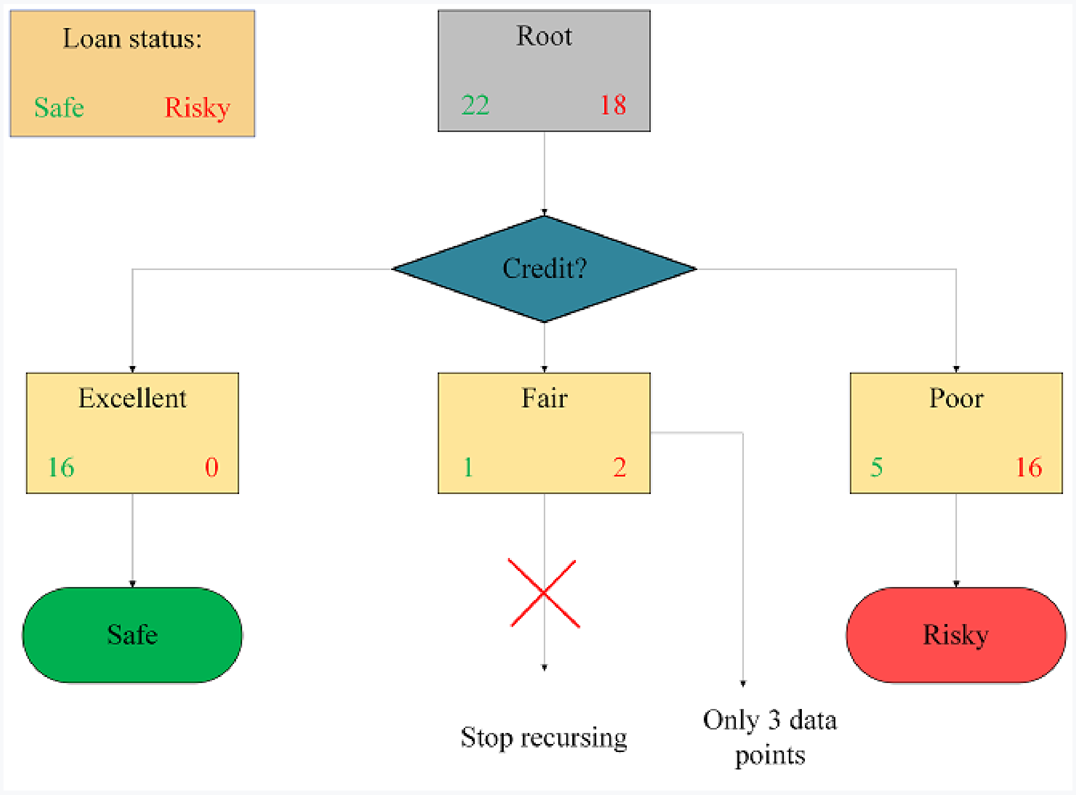
5.2.9

Number of instances

Another important condition for **early stopping** in decision tree generation is based on the **number of instances** (examples) in each node. As the tree grows, the number of examples in each node typically decreases. If a node contains very few examples, the predictions made by the tree may become unreliable. This is because small numbers of instances lead to greater variability in the classification, which reduces the confidence in the results.

The decision to stop tree generation when a node contains too few examples depends on both the **task** and the **dataset**. What qualifies as a "small" number of examples may vary: in some cases, a node with just 100 examples might be considered too small, while in other cases, this number could be acceptable. As a result, many tree generation algorithms allow you to set a **minimum number of instances** in a node. Once the number of examples in a node falls below this threshold, the algorithm will stop further splitting that node, regardless of whether additional splits could potentially improve the tree's accuracy.

This approach helps to avoid overfitting, particularly in situations where the tree might end up creating branches based on a very small subset of the data, which could lead to unreliable and less generalizable models. By setting a reasonable threshold for the minimum number of instances per node, you can ensure that the tree remains robust and that splits are made based on enough data to provide meaningful results.



5.2.10

Why is it important to stop tree generation when a node contains very few instances?

- Because the predictions for small nodes are often unreliable
- Because the tree would become too shallow
- Because the tree will become too deep
- Because splitting small nodes leads to overfitting

5.2.11

In decision tree generation, **early stopping** is a crucial strategy to prevent overfitting and ensure that the tree remains interpretable and generalizable. There are several approaches to achieving early termination of tree building, each with a specific purpose. Let's review these key approaches:

- **Setting the maximum tree depth -** is common approach focused on limit the depth of the tree. This is often done by setting a maximum depth, meaning the tree will not grow beyond a certain number of levels. This method is useful for ensuring the tree remains simple and easy to interpret. However, choosing the optimal depth can be challenging and might require experimentation.

- **Stopping when classification error is not reduced** monitor the **classification error** during the tree-building process. If the error stops decreasing, or decreases only marginally, further tree expansion may not improve the model's performance. This is an indicator that the tree has reached its optimal complexity and additional splits may lead to overfitting.
- **Stopping when nodes contain too few examples** - as the tree grows, the number of examples in each node typically decreases. If a node contains too few examples, its predictions become unreliable, and further splits may be based on insufficient data. To avoid this, tree generation can be stopped when a node contains fewer examples than a predefined threshold, ensuring that the tree does not become too specific to small subsets of the data.

5.2.12

What is overlearning or overfitting?

- A phenomenon where test error increases with model complexity.
- A phenomenon where test error decreases with model complexity.
- A phenomenon where the training error decreases with model complexity.
- A phenomenon where the training error increases with model complexity.

5.2.13

"Among competing hypotheses, the one with the fewest assumptions should be selected",

This statement is also called:

- The principle of Occam's Razor
- Simple existence principle
- Regression rule
- The principle of selection of assumptions

5.2.14

Select the rules that can be applied to stop the generation of the decision tree, i.e. the stopping condition.

- If all the features from the dataset have already been used
- If the classification error is not reduced by further development
- If there are only leaves in the tree that contain a number of examples less than the set minimum of leaf for unfolding - min_samples_leaf

- If the GINI index of the whole tree is less than -1

5.3 Tree pruning

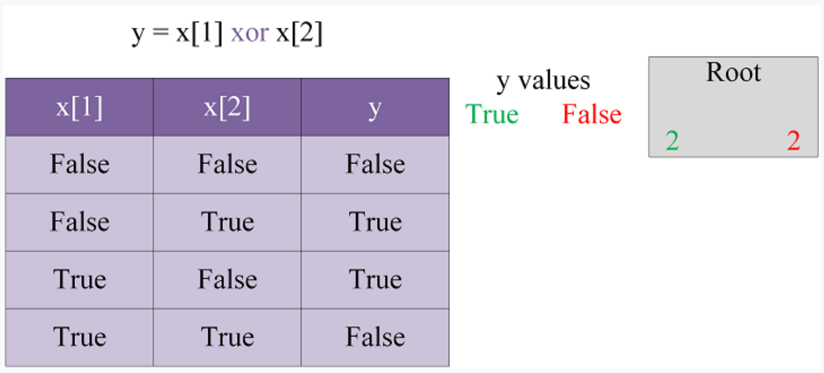
📖 5.3.1

XOR problem

The early stopping conditions mentioned in the previous section belong among the quick and easy approaches to prevent overlearning in decision trees.

For example, stopping tree generation, if the classification error does not decrease appears to be a successful approach. Of course, this approach also has its limits. We can demonstrate one concerning the XOR problem.

Consider four training examples that classify a target variable y for features/attributes **x[1]** and **x[2].**

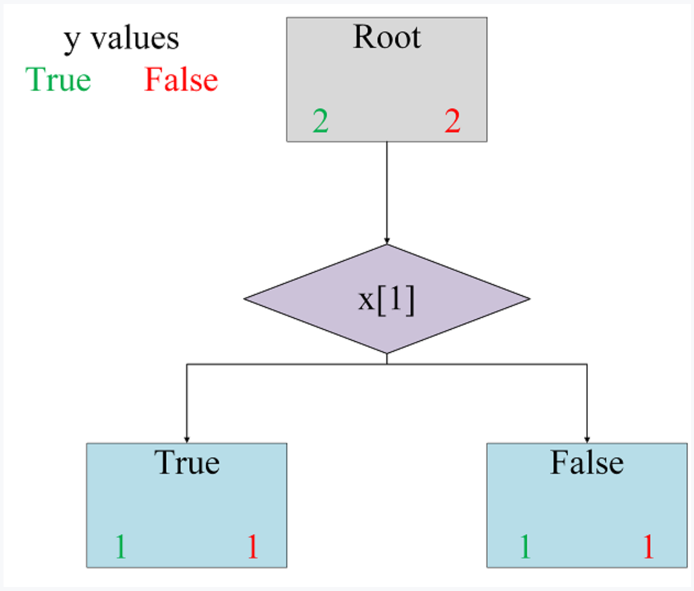

y = x[1] xor x[2]

x[1]	x[2]	y
False	False	False
False	True	True
True	False	True
True	True	False

y values
True False
Root
2
2

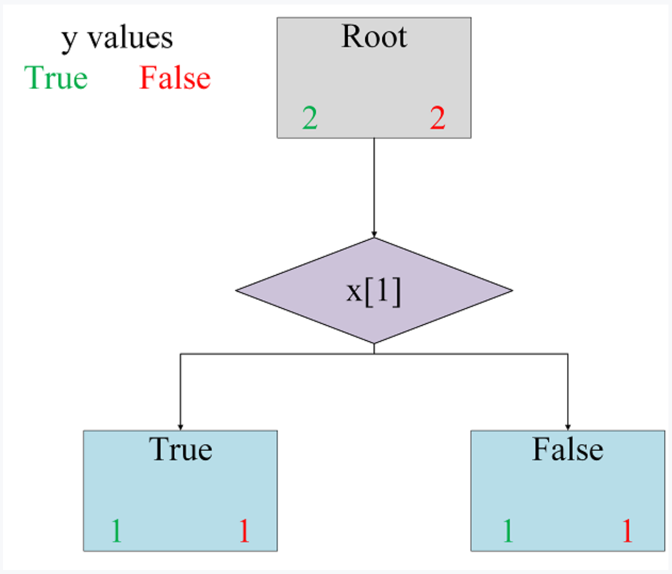
If we consider only these four training examples, the classification error of a classifier that classifies all examples as **True** (or even **False**) will be 2 / (2 +2) = 0.5

Generating the tree and dividing by the attribute **x[1]**, we get the following tree.



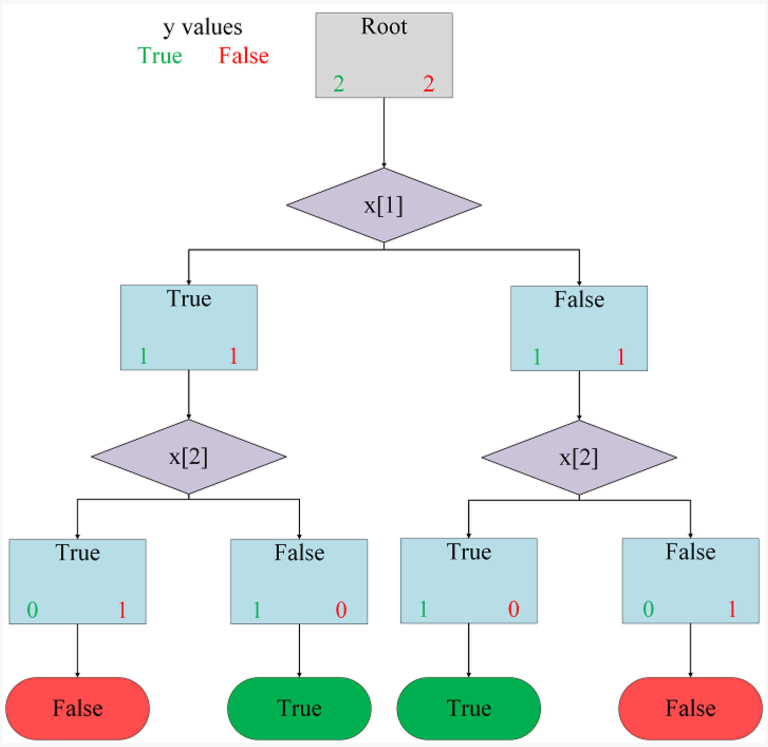
The classification error of this tree will be (1 + 1) / (2 +2) = 0.5

This means that the classification error has not decreased. If we stopped the tree generation early, we would not continue with the tree generation in this case. The final tree would look as follows:



Practically, we would not create any classifier.

However, if we further partition the tree according to the attribute **x[2]** we get the following tree with zero classification error.



The example above illustrates the problem of the early stopping approach.

To overcome this limitation, we can use **pruning**, which is the process of simplifying the tree after it has been fully grown. Pruning allows us to create a tree that balances complexity and performance by removing parts of the tree that do not significantly improve classification accuracy.

5.3.2

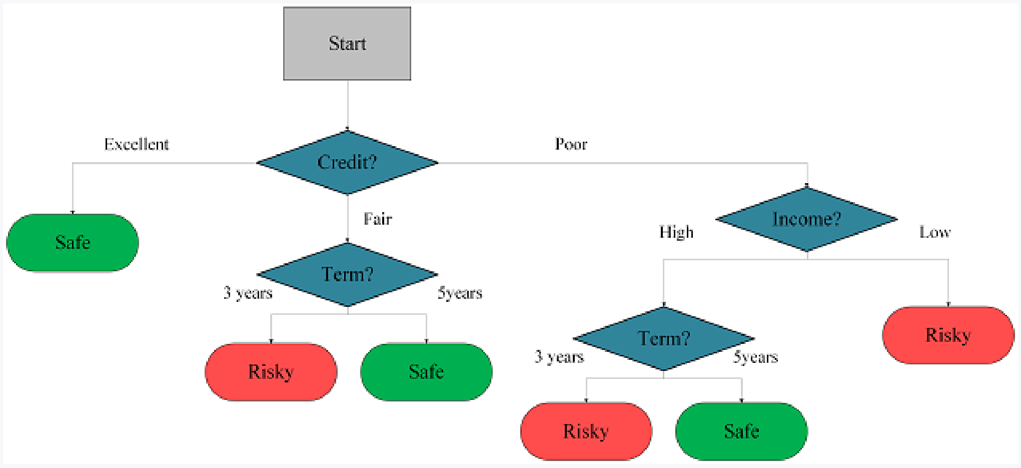
What is a potential issue with using early stopping based on classification error in decision tree generation?

- The tree might stop growing before it reaches the optimal depth
- The tree may become too simple and underfitting occurs
- The tree will always be able to perfectly classify all training examples
- Early stopping always leads to overfitting

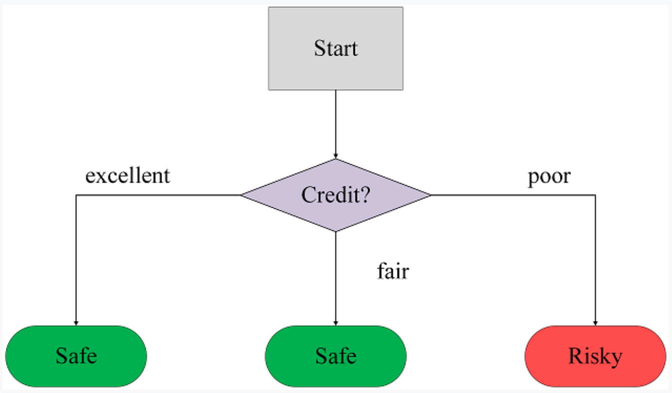
5.3.3

Consider two decision trees

Tree A:



Tree B:



Which one of the above trees is simpler?

- Tree B
- Tree A
- Equal

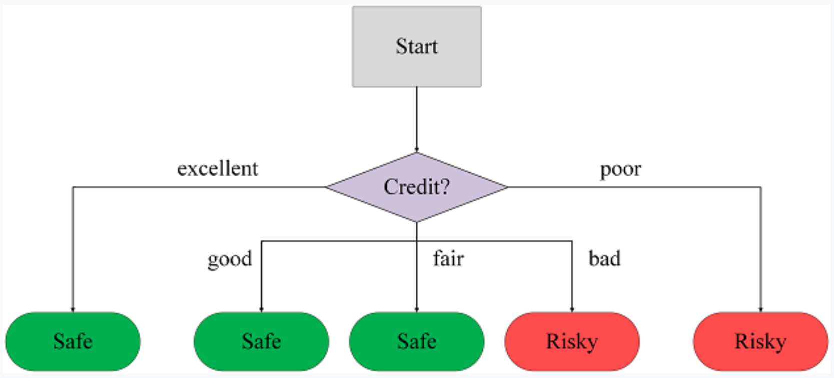
5.3.4

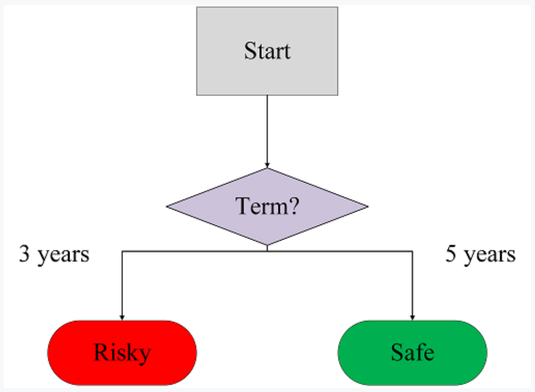
Pruning

Pruning is a technique used in decision tree generation where we initially train a complex tree and then simplify it later to avoid overfitting. The key idea behind pruning is that we want a model that generalizes well, which means finding the right balance between complexity and accuracy. In the case of different tree depths, it is not difficult to determine which of two trees is simpler. In the case of trees with the same depth, it is a more serious problem.

There are several metrics to determine the complexity of a tree. In our examples, we will consider **the number of leaves**.

For example, consider the following trees.





We express the number of nodes of the tree using the L variable.

- The number of nodes of the first tree S1 will be L(S1) = 5
- The number of nodes of the second tree S2 will be L(S2) = 2

By simple comparison, we find that the second tree is simpler, i.e., the first tree is more complex. Importantly, while it is obvious to a human observer which of the two trees is more complex, by expressing the node count metric, it is possible for the algorithm to detect the complexity of the tree as well.

5.3.5

What is the main goal of pruning in decision trees?

- To train a complex tree and simplify it later to avoid overfitting

- To create a tree with more leaves for better accuracy
- To train a shallow tree without any need for simplification
- To stop tree generation once the training error reaches zero

📖 5.3.6

Total cost function

In decision tree pruning, it's important to strike a balance between two key factors:

- how well the tree predicts data (its performance)
- how simple or complex the tree is {its complexity).

Achieving this balance ensures that we do not overfit the model or make it too simplistic, potentially underfitting the data.

To balance these two aspects, we use a **total cost function**. This function combines the **performance** of the tree (often measured by classification error) and its **complexity** (typically measured by the number of leaves). The performance metric, like classification error, quantifies how accurately the tree classifies examples, while the complexity is measured by the number of leaves or nodes.

The total cost function can be expressed by various ways:

We calculate it as

```
Total Cost = Measure of Fit + Measure of Complexity
```

where pas a **measure of fit,** we use a tree performance metric, e.g., classification error, and as a **measure of complexity,** the number of leaves in the tree.

Therefore, the formula can also be understood as follows:

```
Total Cost = Classification Error + Number of Leaf Nodes
```

or more appropriately, it can be expressed as follows:

```
Total Cost = Classification Error + λ × Number of Leaves
```

In this formula, **λ** (lambda) is a constant that adjusts the impact of the number of leaves on the total cost. This constant is crucial because the classification error is a value between 0 and 1, while the number of leaves is a positive integer. By using **λ**, we transform the number of leaves into a comparable range (0, 1), ensuring that both the error and complexity are on the same scale and can be properly balanced.

This allows us to choose a tree that doesn't just fit the training data well but also generalizes effectively to new, unseen data, avoiding both underfitting and overfitting.

5.3.7

How the so-called Total cost is calculated for the decision tree

- as the sum of the classification accuracy metric and the model complexity metric
- as the difference of the classification accuracy metric and the model complexity metric
- as a proportion of the classification accuracy metric and the model complexity metric
- as the sum of the classification accuracy metric and the dataset balance metric
- as the difference of the classification accuracy metric and the dataset balance metric
- as a proportion of the classification accuracy metric and the dataset balance metric

5.3.8

Why is the constant λ important in the total cost function for pruning decision trees?

- It ensures the number of leaves is on the same scale as the classification error
- It helps calculate the classification error
- It determines the maximum tree depth
- It adjusts the complexity of the training dataset

5.3.9

What are the two key factors that need to be balanced in decision tree pruning?

- Tree performance and complexity
- Training error and model depth
- Number of nodes and classification accuracy
- Model complexity and training time

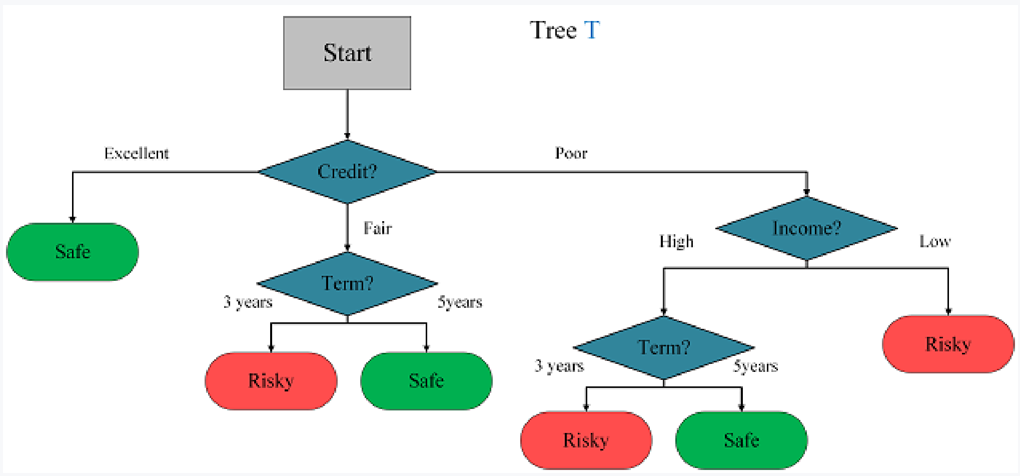
🕮 5.3.10

The tree pruning algorithm

The tree pruning algorithm is a key process in simplifying a decision tree while maintaining or improving its predictive accuracy. The goal of pruning is to remove unnecessary complexity by eliminating nodes that do not contribute significantly to the tree's performance. Below is the step-by-step approach to tree pruning:

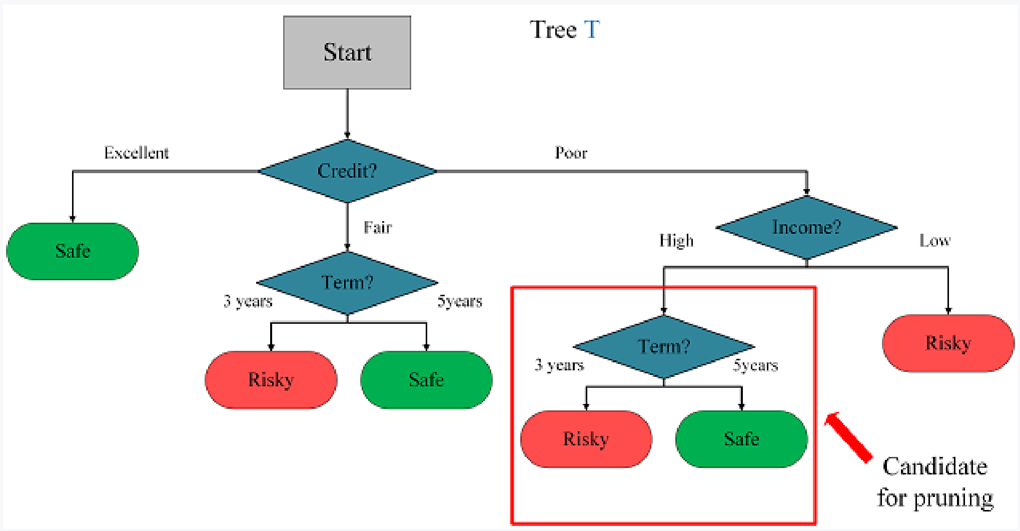
Step1 - Create a complete decision tree

- Begin by constructing a decision tree without any pruning. This tree will typically be complex and may overfit the training data, capturing noise and unnecessary patterns.



Step 2 - Identify a candidate node for removal

- Once the full tree is created, look for nodes that might not contribute significantly to the model's performance. These are potential candidates for removal.

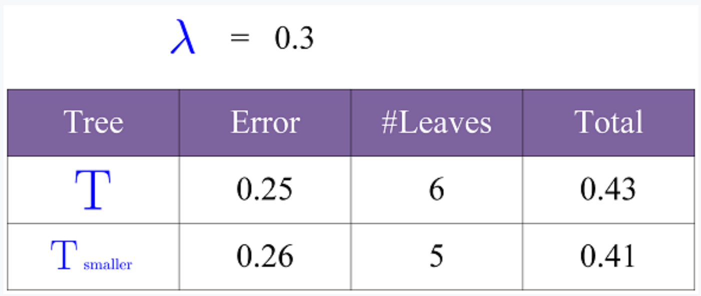


Step 3 - Calculate the total cost

- For each candidate node, calculate the total cost with the candidate for removal T and without candidate for removal $T_{smaller}$ according to the formula:

```
C(T) = Error(T) + λ × L(T)
```

The total costs in our case look as follows

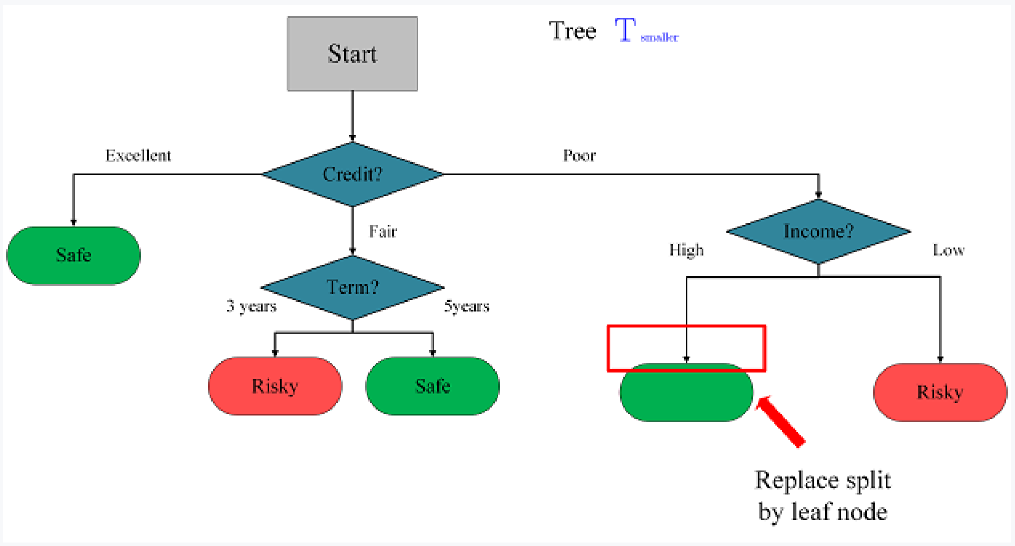


$\lambda = 0.3$

Tree	Error	#Leaves	Total
T	0.25	6	0.43
$T_{smaller}$	0.26	5	0.41

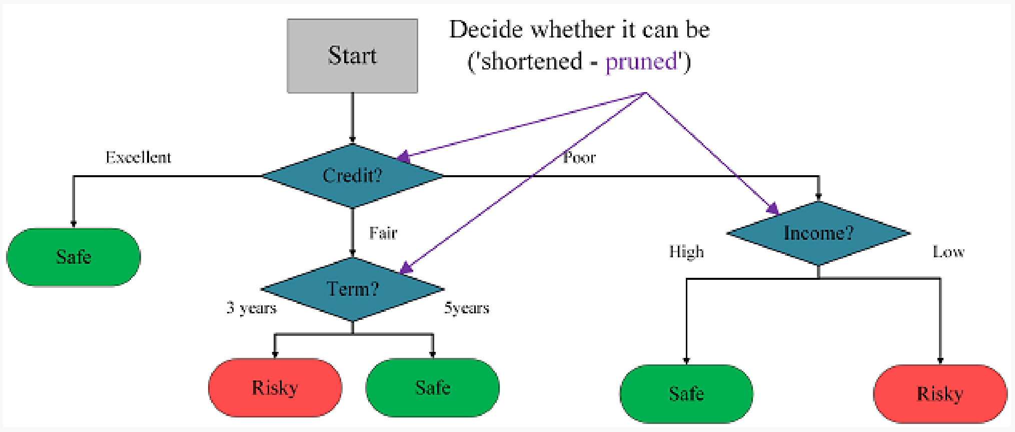
Step 4 - Compare the total cost

- If the total cost decreases after removing the candidate node, then proceed with the removal. This indicates that the tree has become simpler without a significant loss in predictive accuracy.



Step 5 - Repeat steps 2 - 4 for each node in the tree

- Continue the pruning process by repeating steps 2 through 4 for each node in the tree, until no further nodes can be removed without increasing the total cost.



5.3.11

What is the main goal of the tree pruning algorithm?

- To reduce the tree’s complexity without significantly increasing the error
- To make the tree deeper
- To calculate the training error
- To eliminate features from the dataset

5.3.12

At the end, we will discuss a more elaborate decision tree pruning algorithm, which helps in simplifying the tree while ensuring that it maintains its predictive accuracy. The goal is to reduce the tree's complexity and prevent overfitting without sacrificing performance.

The pruning algorithm works by starting at the bottom of the tree and moving upward. It applies the **prune_split** function to each decision node (denoted as M) in the tree. The steps of the pruning process are as follows:

- **Start at the bottom of the tree and move upwards** - the algorithm begins at the leaf nodes and progresses towards the root, checking each decision node for possible pruning. This process helps to reduce complexity by removing unnecessary branches.
- **Apply the prune_split function to each decision node M** - the **prune_split** function is applied to each node in the tree to evaluate whether pruning it would improve the overall tree performance.
- **Calculate the total cost of the tree T -** for the original tree, calculate the total cost using the formula: $C(T) = error(T) + \lambda \times L(T)$
- **Prune the subtree from node M** - after pruning the subtree at node M, the tree becomes smaller. We calculate the new total cost for the pruned tree (**$T_{smaller}$**) using the same formula: $C(T_{smaller}) = error(T_{smaller}) + \lambda \times L(T_{smaller})$

- **Compare the total costs** - if the total cost of the pruned tree (**C($T_{smaller}$)**) is less than the total cost of the original tree (**C(T)**), then the pruning is beneficial, and we prune the tree to create **$T_{smaller}$**.
- **Repeat for all decision nodes** - this process is repeated for each decision node, moving upwards through the tree, until no further beneficial pruning can be made.

5.3.13

Which of the following is used in the pruning algorithm to evaluate the performance of the tree before and after pruning?

- Classification error and the number of leaves
- Classification error and training set size
- Number of leaves and depth of the tree
- Classification error and the number of features

5.4 Missing (incomplete) data

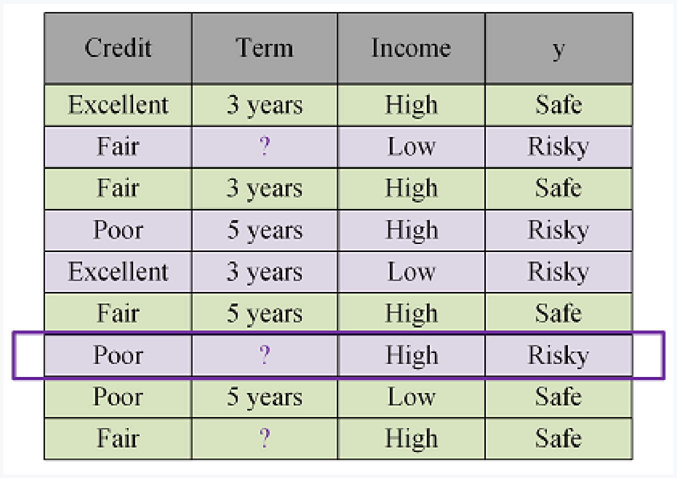
5.4.1

Incomplete data

In real-world machine learning tasks, handling incomplete data is a common and challenging problem. Datasets are rarely perfect and often contain missing values. These gaps in the data can arise due to various reasons, such as errors during data collection, privacy concerns, or the unavailability of certain features for specific instances.

An example of incomplete data can be seen in the case of a bank's customer dataset. Consider a situation where the bank has a set of customers who have been offered loans, while others only maintain accounts with the bank. One key column in this dataset might be the "maturity date" for the loans, which will obviously be missing for customers who did not take out loans. However, the bank still has valuable information about the customers who did not use credit services. This information could include details like account balance, transaction history, and customer demographics.

The challenge is how to effectively handle this incomplete data. For machine learning algorithms to make accurate predictions, it's crucial that missing data is either handled or appropriately imputed. Ignoring or discarding instances with missing data might lead to biased models or loss of valuable information.



Credit	Term	Income	y
Excellent	3 years	High	Safe
Fair	?	Low	Risky
Fair	3 years	High	Safe
Poor	5 years	High	Risky
Excellent	3 years	Low	Risky
Fair	5 years	High	Safe
Poor	?	High	Risky
Poor	5 years	Low	Safe
Fair	?	High	Safe

Various techniques can be employed to address this issue, such as:

1. Imputation fills in missing values with estimated or predicted values based on other features in the dataset.
2. Omitting missing data removes rows or columns with missing data, though this can lead to loss of information.
3. Using algorithms that handle missing data - some machine learning algorithms can directly handle missing data during model training, such as decision trees.

5.4.2

What is one of the main challenges in real-world machine learning tasks regarding datasets?

- Datasets are incomplete and contain missing values
- Datasets are always too small
- Datasets always have too many features
- Datasets always have balanced classes

5.4.3

Approach 1 (Remove missing data)

One common approach to handling missing data is to simply disregard or remove the examples (rows) or attributes (columns) with missing values. This method can be useful when only a small percentage of the dataset is incomplete. For instance, if only a few rows or attributes have missing data, removing these might not have a significant impact on the model’s performance or the dataset's overall integrity.

However, this approach can become problematic if a substantial portion of the dataset is missing information. For example, if 50% of the records in the dataset are missing an important attribute, simply removing those examples could significantly reduce the amount of usable data. This reduction can lead to biased models, as the model may not represent the diversity of the data accurately.

There are some advantages to removing examples or attributes with missing data:

1. Simplicity - this approach is straightforward to implement and easy to understand.
2. Universality - it can be applied to virtually any type of machine learning model, including decision trees, logistic regression, and linear regression.

However, the disadvantages should also be considered:

1. Loss of information - removing data points or attributes can remove valuable information, reducing the overall predictive power of the model.
2. When to remove - it is not always clear when it's better to remove rows (examples) or columns (features/attributes), especially when the missing data is distributed unevenly across the dataset.
3. Impact on model updates - if a model is updated with new data, missing input data could become a problem again, affecting the model's ability to make predictions effectively.

While removing incomplete data can be a quick fix, it should be applied cautiously, particularly when a large portion of the data is missing.

5.4.4

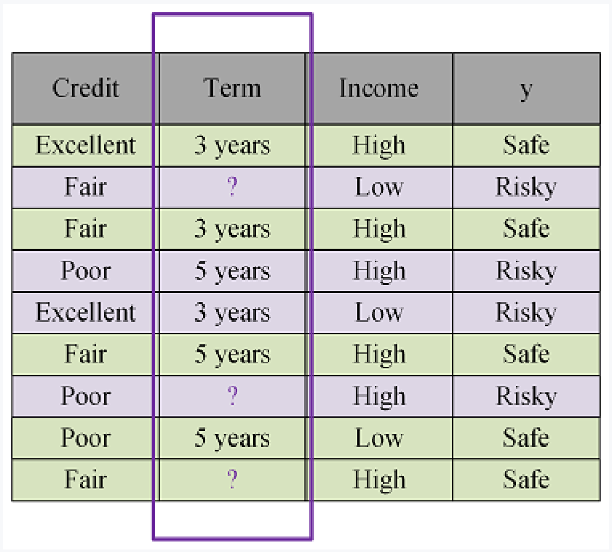
What is a potential disadvantage of removing examples or attributes with missing data?

- It may result in loss of important information
- It simplifies the model too much
- It always improves the model's performance
- It makes the dataset larger

5.4.5

Approach 2 (Data imputation)

Another common approach to handling missing data is **data imputation**, where the missing values are filled in with estimates. This is particularly useful when dealing with missing data in features or columns, as it helps preserve the dataset's size and ensures that the model can still use all available information.



Credit	Term	Income	y
Excellent	3 years	High	Safe
Fair	?	Low	Risky
Fair	3 years	High	Safe
Poor	5 years	High	Risky
Excellent	3 years	Low	Risky
Fair	5 years	High	Safe
Poor	?	High	Risky
Poor	5 years	Low	Safe
Fair	?	High	Safe

There are several methods for imputation, and the choice of method depends on the nature of the data and the distribution of the values. Some common imputation techniques include:

1. Filling with the most frequent value - for categorical features, one of the simplest methods is to replace the missing values with the most frequent value, or mode, in the column. This approach works well when the missing values are not randomly distributed, but it may not always capture the underlying patterns in the data.
2. Filling with the mean or median - for numerical data, filling missing values with the mean or median of the available data in the feature is a common strategy. The mean is typically used when the data is approximately normally distributed, while the median is preferred when the data has outliers or is skewed. The median is less sensitive to extreme values, making it more robust in many cases.
3. Interpolation - in some cases, especially for time-series data, interpolation can be used to estimate missing values based on the surrounding values. This can be linear interpolation or other forms, such as polynomial interpolation, depending on the data's structure.
4. Predictive modeling - for more complex scenarios, a model can be trained to predict the missing values based on the relationship between the missing attribute and other available attributes. This method can be more accurate but also more computationally expensive.

Each of these methods has its advantages and limitations, and the choice of imputation method should be based on the data type, distribution, and the extent of missingness. Imputation is generally preferred when retaining all the examples in the dataset is important, and it avoids the risk of losing too much data, as could happen when removing examples with missing values.

5.4.6

Which method is commonly used for imputing missing values in categorical features?

- Filling with the most frequent value
- Filling with the mean
- Interpolation
- Predictive modeling

5.4.7

Which of the following imputation methods is typically used for numerical data with outliers?

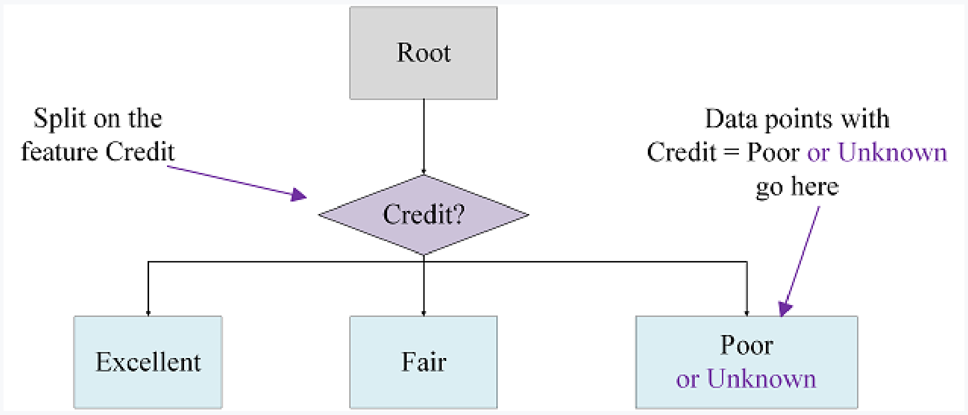
- Filling with the median
- Filling with the mean
- Interpolation
- Filling with the most frequent value

5.4.8

Approach 3 (Set unknown value)

A third approach for handling missing data is to **reason with the missing data**. This strategy can be particularly helpful when the missing data represents a specific situation, such as an unknown or unrecorded value, and can be treated as a valid category rather than attempting to estimate it based on other data points.

One simple way to apply this reasoning is by **assigning a special value** to missing data, such as "unknown" or "missing." This method is particularly useful when the absence of a value carries meaning or can be treated as a distinct category within the dataset. For example, in a customer dataset for a bank, if certain customers do not provide their income information, instead of discarding those records or filling in the missing values, we could assign the value "unknown" to represent missing income data.


Root
Split on the feature Credit
Credit?
Data points with Credit = Poor or Unknown go here
Excellent
Fair
Poor or Unknown

This approach is often used when:

- The missing values represent a specific category that has meaning, such as missing preferences, unrecorded transactions, or unknown responses.
- It is important for the model to recognize that certain data points are deliberately missing or unknown, and not necessarily a random omission.

Advantages of reasoning with missing data:

- It preserves all records without imputation, thus preventing loss of valuable data.
- It allows the model to treat the missing data as a separate class or category, which can sometimes reveal insights into the patterns of missingness.
- It avoids potential bias introduced by incorrect imputations or assumptions about the missing data.

Disadvantages:

- It may introduce additional complexity into the model by adding a new category (e.g., "unknown") which could affect classification or prediction tasks.
- It assumes that missing data is not random, but rather has some meaningful underlying reason for its absence. If this assumption is incorrect, the model could be misled.

5.4.9

What is the main advantage of adding a special value like "unknown" to missing data?

- It helps in maintaining the dataset's integrity by not discarding any records.
- It allows for a more accurate imputation of missing values.
- It prevents bias introduced by random missingness.
- It increases the computational cost of model training.

5.4.10

Which of the following would be an appropriate use of adding a special value like "unknown" to missing data?

- When missing data represents a meaningful category or class, such as unknown customer preferences.
- When missing data is truly random and has no specific pattern.
- When numerical values are missing and should be replaced by the mean or median.
- When missing data represents an error that should be ignored.

5.5 Practical tasks

5.5.1

Project: Decision tree optimization

Create a decision tree using Titanic data without setting the depth of the tree. Later create more suitable decision tree.

- Dataset: https://priscilla.fitped.eu/data/pandas/titanic.csv

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

# Load the Titanic dataset from a URL into a pandas DataFrame
data =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/titanic.c
sv')
```

```
# Select relevant columns from the dataset for the model
data = data[['Survived', 'Pclass', 'Sex', 'Age', 'SibSp',
'Parch', 'Embarked']]

# Drop rows with missing values (NaN) to ensure the model
doesn't encounter incomplete data
data = data.dropna()

# Convert the categorical 'Embarked' column into
dummy/indicator variables (one-hot encoding)
# drop_first=False ensures that the original column isn't
dropped, keeping all categories for analysis
data = pd.get_dummies(data, columns=["Embarked"],
drop_first=False)

# Replace 'male' and 'female' values in 'Sex' column with
numeric values: 'male' -> 0, 'female' -> 1
# Explicitly cast 'Sex' column to int after replacement to
avoid deprecation warning

# s = Series(['male', 'female'])
# replace_dict = {'male': '0', 'female': '1'} # replacements
maintain original types
# data['Sex'] = data['Sex'].replace(replace_dict)
pd.set_option('future.no_silent_downcasting', True)
data['Sex'] = data['Sex'].replace({'male': 0, 'female': 1})

# Split the data into features (X) and target (y)
X = data[data.columns.difference(['Survived'])]  # All columns
except 'Survived'
y = data['Survived']  # 'Survived' is the target variable

# Split the data into training and testing sets, with 20% of
the data for testing
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Initialize the Decision Tree Classifier with a fixed random
state for reproducibility
clf = DecisionTreeClassifier(random_state=42)

# Train the classifier on the training data
clf = clf.fit(X_train, y_train)
```

```
# Use the trained model to predict the target variable on the
testing data
y_pred = clf.predict(X_test)

# Import the metrics module to evaluate the model's
performance
from sklearn import metrics

# Calculate the accuracy of the model by comparing predicted
values with actual values
acc = metrics.accuracy_score(y_test, y_pred)

# Print the accuracy score
print("Accuracy:", acc)
```

Program output:

```
Accuracy: 0.7342657342657343
```

Find the depth of the generated tree.

```
print(clf.get_depth())
```

Program output:

```
17
```

The original tree has a depth of 17. Next, we find out how many leaves the tree has.

```
print(clf.get_n_leaves())
```

Program output:

```
151
```

The tree has up to 151 leaves. Based on the depth and number of leaves, we can conclude that this tree is complex given the number of data from which it is trained.

To confirm the reasoning, let us list all the accuracies of the tree using depths from 1 to 17.

```
# Loop through different tree depths (from 1 to 17)
for i in range(1, 18):
    # Initialize a DecisionTreeClassifier with the current
max_depth
    dtree = DecisionTreeClassifier(max_depth=i,
random_state=42)

    # Fit the decision tree model to the training data
(X_train and y_train)
    dtree.fit(X_train, y_train)

    # Predict the target variable (y_test) using the trained
model
    y_pred = dtree.predict(X_test)

    # Print the current tree depth and the accuracy of the
predictions on the test set
    print('Depth: ', i, ' accuracy:',
metrics.accuracy_score(y_test, y_pred))
```

Program output:

```
Depth:  1  accuracy: 0.7482517482517482
Depth:  2  accuracy: 0.7482517482517482
Depth:  3  accuracy: 0.7272727272727273
Depth:  4  accuracy: 0.7482517482517482
Depth:  5  accuracy: 0.7832167832167832
Depth:  6  accuracy: 0.7342657342657343
Depth:  7  accuracy: 0.7762237762237763
Depth:  8  accuracy: 0.7342657342657343
Depth:  9  accuracy: 0.7412587412587412
Depth:  10  accuracy: 0.7272727272727273
Depth:  11  accuracy: 0.7132867132867133
Depth:  12  accuracy: 0.7342657342657343
Depth:  13  accuracy: 0.7412587412587412
Depth:  14  accuracy: 0.7342657342657343
Depth:  15  accuracy: 0.7482517482517482
Depth:  16  accuracy: 0.7412587412587412
Depth:  17  accuracy: 0.7342657342657343
```

At depth 5, the tree is 4% more accurate on the test data than at the original depth of 17. We pruned the tree, it is less complex and yet more accurate. We have prevented the tree from overtraining.

Such a tree also has a smaller number of leaves, only 22.

```
# Initialize the DecisionTreeClassifier with max_depth set to
5 and random_state for reproducibility
dtree = DecisionTreeClassifier(max_depth=5, random_state=42)

# Fit the decision tree model to the training data
dtree.fit(X_train, y_train)

# Get and print the number of leaves in the decision tree
print(dtree.get_n_leaves())
```

Program output:

```
22
```

Another way to prevent overtraining is by using total impurity sheets and the so-called effective alpha tree (https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html).

So-called minimal cost complexity pruning recursively finds the weakest node. This is characterized by the effective alpha, and the nodes with the smallest effective alpha are pruned first.

The **sklearn** library provides a **cost_complexity_pruning_path** function whose return value is the effective alpha and the corresponding total leaf impurity.

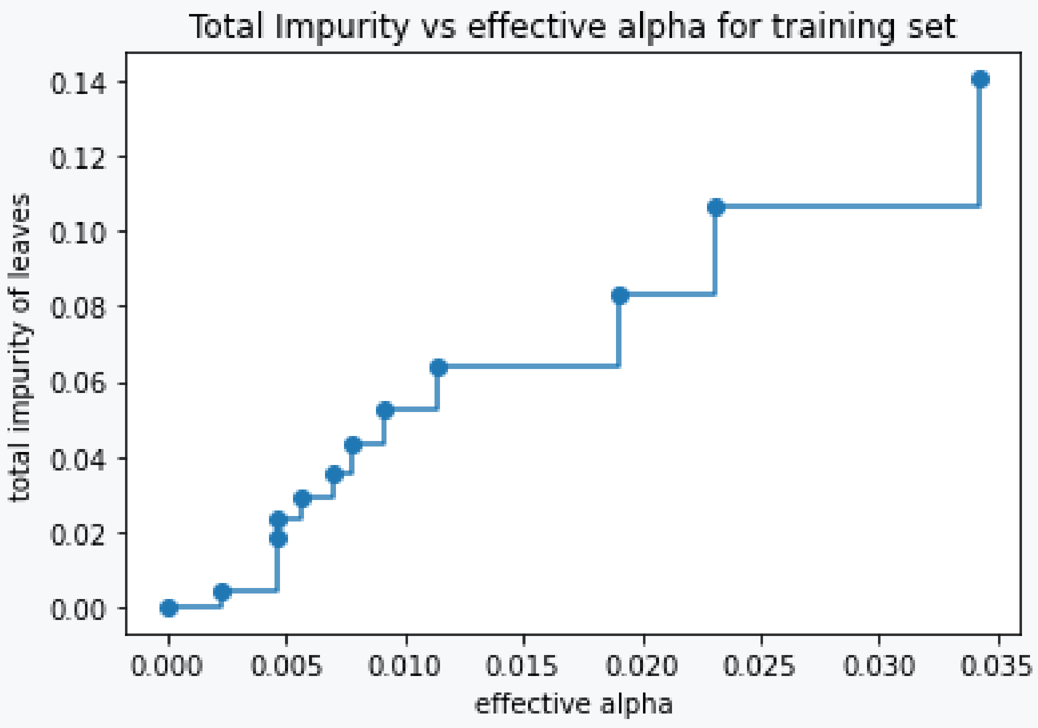
As the alpha value increases, more of the tree is pruned, increasing the total impurity of leaves.

```
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.tree import DecisionTreeClassifier

# Load the breast cancer dataset
X, y = load_breast_cancer(return_X_y=True)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=0)

# Create a decision tree classifier
clf = DecisionTreeClassifier(random_state=0)

# Compute the cost-complexity pruning path
path = clf.cost_complexity_pruning_path(X_train, y_train)
```

```
# Extract the alpha values and impurities from the pruning
path
ccp_alphas, impurities = path.ccp_alphas, path.impurities

# Create a plot to visualize the pruning path
fig, ax = plt.subplots()

# Plot the total impurity of leaves as a function of effective
alpha
ax.plot(ccp_alphas[:-1], impurities[:-1], marker="o",
drawstyle="steps-post")

# Label the axes and set the title
ax.set_xlabel("effective alpha")
ax.set_ylabel("total impurity of leaves")
ax.set_title("Total Impurity vs effective alpha for training
set")

# Display the plot
plt.show()
```

Program output:

```
/home/johny/.local/lib/python3.9/site-
packages/matplotlib/projections/__init__.py:63: UserWarning:
Unable to import Axes3D. This may be due to multiple versions
of Matplotlib being installed (e.g. as a system package and as
a pip package). As a result, the 3D projection is not
available.
  warnings.warn("Unable to import Axes3D. This may be due to
multiple versions of "
```



We train a decision tree using the effective alpha. The last value in ccp_alpha is the value that prunes the whole tree, clfs[-1] is the tree with one node.

```
# Initialize an empty list to store decision tree classifiers
clfs = []

# Loop through each value of ccp_alpha
for ccp_alpha in ccp_alphas:
    # Create a DecisionTreeClassifier with the current
ccp_alpha value
    clf = DecisionTreeClassifier(random_state=0,
ccp_alpha=ccp_alpha)

    # Train the classifier on the training data
    clf.fit(X_train, y_train)

    # Append the trained classifier to the list of classifiers
    clfs.append(clf)

# Print the number of nodes in the last tree and the
corresponding ccp_alpha value
print(
    "Number of nodes in the last tree is: {} with ccp_alpha:
{}".format(
        clfs[-1].tree_.node_count, ccp_alphas[-1]
    )
)
```

Program output:

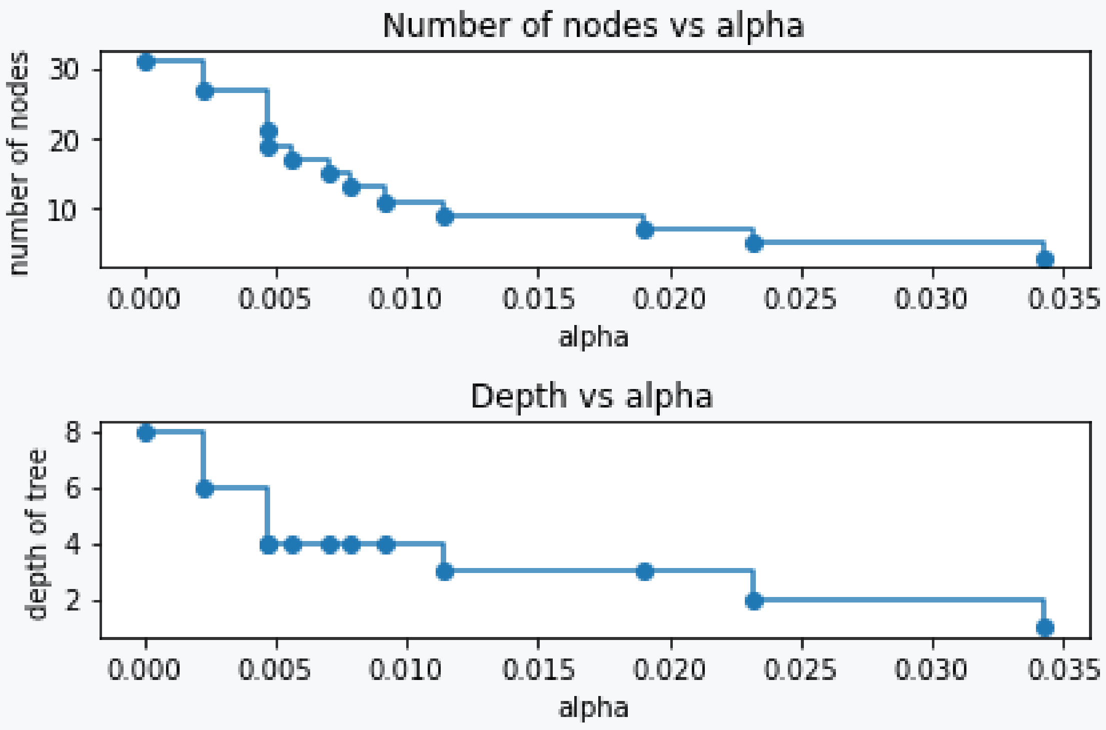
```
Number of nodes in the last tree is: 1 with ccp_alpha:
0.3272984419327777
```

Remove the last element clfs and ccp_alha.

The following graphs show how the number of nodes and the depth of the tree decreases with increasing alpha.

```
clfs = clfs[:-1]
ccp_alphas = ccp_alphas[:-1]

node_counts = [clf.tree_.node_count for clf in clfs]
depth = [clf.tree_.max_depth for clf in clfs]
fig, ax = plt.subplots(2, 1)
ax[0].plot(ccp_alphas, node_counts, marker="o",
drawstyle="steps-post")
ax[0].set_xlabel("alpha")
ax[0].set_ylabel("number of nodes")
ax[0].set_title("Number of nodes vs alpha")
ax[1].plot(ccp_alphas, depth, marker="o", drawstyle="steps-
post")
ax[1].set_xlabel("alpha")
ax[1].set_ylabel("depth of tree")
ax[1].set_title("Depth vs alpha")
fig.tight_layout()
```

Program output:

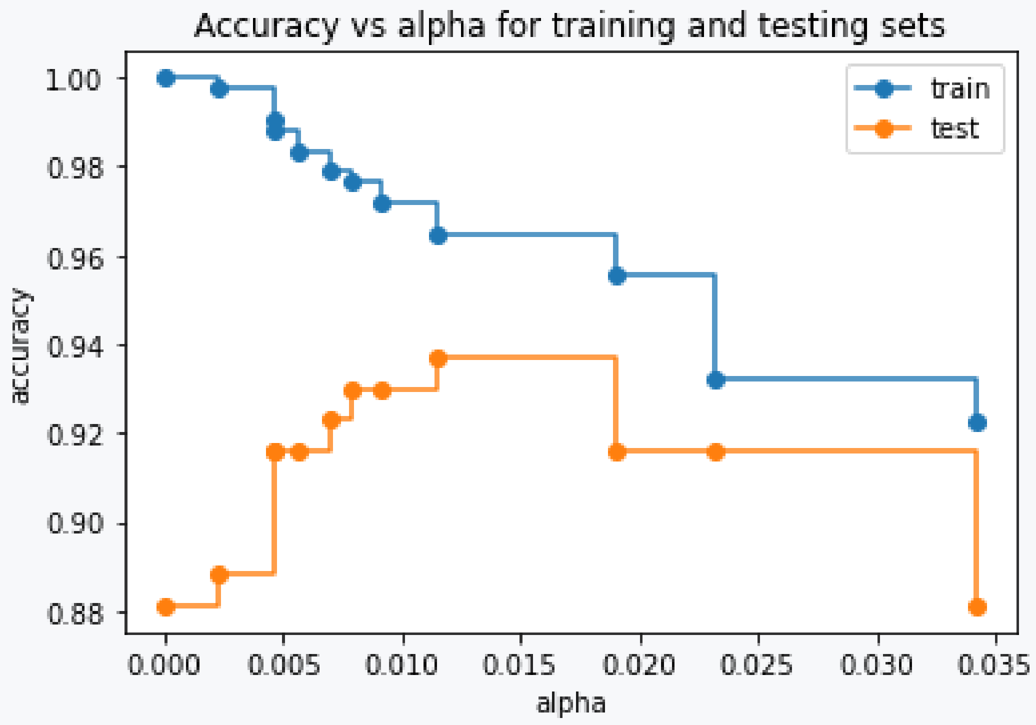


When ccp_alpha is set to zero and the other parameters are default, the tree is retrained, resulting in 100% training accuracy and 88% testing accuracy.

As alpha increases, more of the tree is pruned, leading to better generalization. In the following example, alpha is set to 0.015 to maximize testing accuracy.

```
train_scores = [clf.score(X_train, y_train) for clf in clfs]
test_scores = [clf.score(X_test, y_test) for clf in clfs]

fig, ax = plt.subplots()
ax.set_xlabel("alpha")
ax.set_ylabel("accuracy")
ax.set_title("Accuracy vs alpha for training and testing
sets")
ax.plot(ccp_alphas, train_scores, marker="o", label="train",
drawstyle="steps-post")
ax.plot(ccp_alphas, test_scores, marker="o", label="test",
drawstyle="steps-post")
ax.legend()
plt.show()
```

Program output:



5.5.2

Complete the code so that it correctly calculates tree complexity and alpha.

```
_____ = DecisionTreeClassifier(random_state=0)
_____ = clf. _____ (X_train, y_train)
ccp_alphas, _____ = path. _____ , _____ .impurities
```

5.5.3

Assign the correct functions.

- To obtain the depth of the decision tree, the following is used: _____

- To obtain the number of leaves in the decision tree, the following is used: _____

- To create a decision tree, the following is used: _____

- To train the decision tree, the following is used: _____

- .fit()
- .get_depth()
- .get_n_leaves()

- DecisionTreeClassifier()

⌨ 5.5.4 Maximum depth and number of leaves

Complete the code so that the tree has a maximum depth of 5 and list the number of leaves of the tree.

file1.py

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

data =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/titanic.c
sv')

data = data[['Survived', 'Pclass', 'Sex', 'Age', 'SibSp',
'Parch', 'Embarked']]

data = data.dropna()

data =
pd.get_dummies(data,columns=["Embarked"],drop_first=False)

data['Sex'] = data['Sex'].replace({'male': 0, 'female': 1})

X = data[data.columns.difference(['Survived'])]
y = data['Survived']

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

clf = DecisionTreeClassifier(random_state=42)
```

Metrics for Splitting Decisions

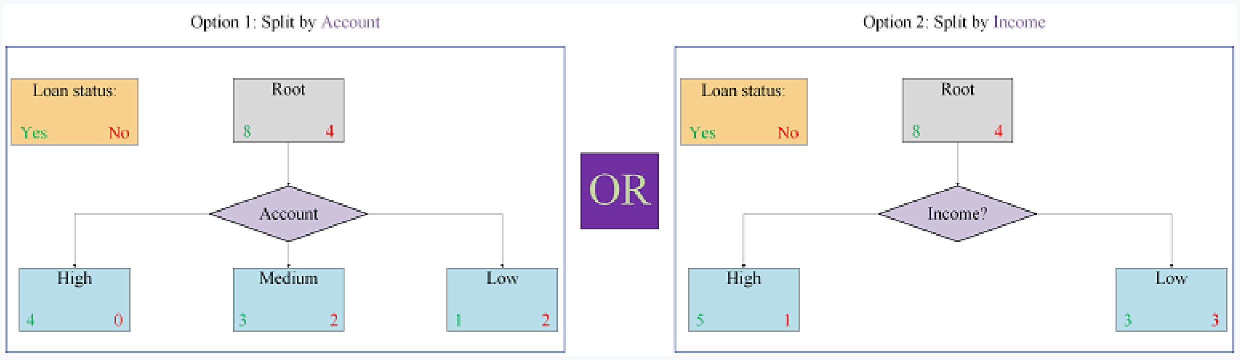
Chapter 6

6.1 GINI index

📖 6.1.1

Best features

One of the critical steps in constructing a decision tree is choosing the most suitable feature for splitting the data at each node. This selection impacts the tree's efficiency and accuracy in making predictions. The algorithm considers all potential trees of depth 1.

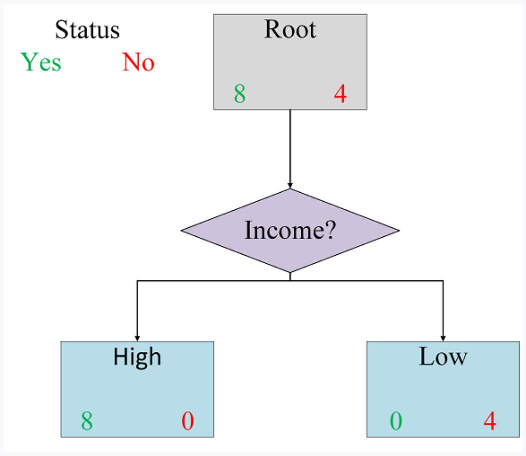


In the previous examples, we used classification error to find the most appropriate feature.

The algorithm evaluates all potential splits by considering the features that result in subsets with the least classification error. Classification error is calculated using the formula:

```
Classification Error = Number of incorrect classifications /
Total number of examples
```

In simple terms, the classification error measures how well a feature splits the dataset into groups that belong to a single class. A smaller classification error indicates that the split is more effective in creating homogeneous subsets. For example, a feature that perfectly divides the dataset into subsets containing only one class would have a classification error of zero.



This method involves the algorithm analyzing all possible trees of depth 1 (trees with only one split). By comparing the classification errors of these trees, the algorithm identifies the feature with the lowest error as the most appropriate for splitting. This process ensures that the tree grows by successively reducing disorder in the data.

The concept of disorder in this context refers to how mixed the subsets are after the split. Ideally, each subset should contain examples from only one class. The feature that achieves this outcome is considered optimal for creating a more accurate decision tree.

6.1.2

What does the classification error indicate in the context of decision trees?

- The disorder of the subsets after splitting
- The size of the dataset
- The time complexity of the algorithm
- The number of features in the dataset

6.1.3

Which statements are true about selecting the best feature in decision trees?

- A smaller classification error indicates a better feature for splitting.
- The classification error is calculated as incorrect classifications divided by the total examples.
- The algorithm evaluates trees of depth 2 to determine the best feature.
- A feature with zero classification error creates subsets of mixed classes.

6.1.4

GINI index

Decision trees require a way to measure the **impurity** or disorder in the data to make effective splits. While classification error can provide this measurement, it is often too simplistic because it assumes a linear relationship. Instead, most decision tree algorithms use more sophisticated metrics, such as **information gain** or the **GINI index**, to evaluate splits.

The **GINI index** quantifies the probability of misclassifying a randomly chosen element in a dataset based on its distribution across classes. In other words, it evaluates how "pure" a subset is after a split. If all elements in the subset belong to a single class, the subset is considered completely pure, and the GINI index is zero. Conversely, if the elements are evenly distributed across multiple classes, the GINI index approaches its maximum value of 0.5, reflecting maximal impurity.

The GINI index value ranges between 0 and 1:

- 0 - all elements belong to a single class (pure subset).
- 0.5 - elements are equally distributed among the classes.
- 1 - elements are completely randomly distributed across classes (complete impurity).

The GINI index is calculated using the formula:

--- ERROR ---

Where $\mathbf{p_i}$ represents the probability of an object being assigned to class **i**. By minimizing the GINI index at each split, decision trees create subsets that are more homogeneous, improving their predictive power.

6.1.5

What does a GINI index of 0 indicate?

- All elements belong to a single class.
- Elements are randomly distributed among classes.
- The data is evenly split between two classes.
- No splits have been made yet.

6.1.6

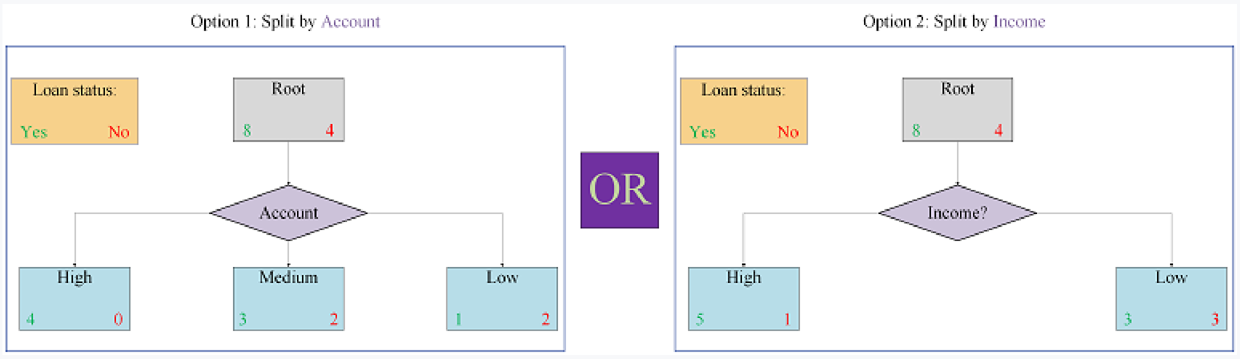
Which statements about the GINI index are true?

- It measures the probability of misclassification.
- It is used to evaluate splits in decision trees.
- It has a linear relationship with impurity.
- A GINI index of 0.5 indicates maximum purity.

6.1.7

Example

To understand the Gini index, let us walk through a practical example. Imagine a dataset with 12 examples and two features: **Account** and **Income**. Our goal is to determine which feature provides the best split for the dataset based on the target variable (e.g., "Yes" or "No"). We do this by calculating and comparing the Gini index for each feature.



We first compute the Gini index for the distribution under the **Account** features. The Gini index will be the weighted average of the Gini indexes of each subset.

Gini$_{account}$ = weighted average(Gini)

For each subset, we will calculate the Gini index according to the formula above

$$G = 1 - \sum_{i=1}^{n} p_i^2$$

- - where p_i is the probability of an object being assigned to a particular class.

Suppose the **Account** feature has three classes: **High**, **Medium**, and **Low**. We calculate the Gini index for each class:

- **Gini$_{\text{high}}$ = 1 - [(probability of "Yes")^2 + (probability of "No")^2]**
- **Gini$_{\text{medium}}$ = 1 - [(probability of "Yes")^2 + (probability of "No")^2]**
- **Gini$_{\text{low}}$ = 1 - [(probability of "Yes")^2 + (probability of "No")^2]**

In the case of the Gini for the class **High**, where the examples are divided into **4** from the value of the target variable **Yes** and **0** from the value of **No**, we calculate the Gini as follows:

Gini$_{\text{high}}$ = 1 - [(probability of "Yes")^2 + (probability of "No")^2] =

$$Gini_{high} = 1 - \left(probability_{\text{Yes}}^2 + probability_{\text{No}}^2\right) = \; = 1 - \left(\left(\frac{4}{4+0}\right)^2 + \left(\frac{0}{4+0}\right)^2\right) = 1 - \left(1^2 + 0^2\right) = 0$$

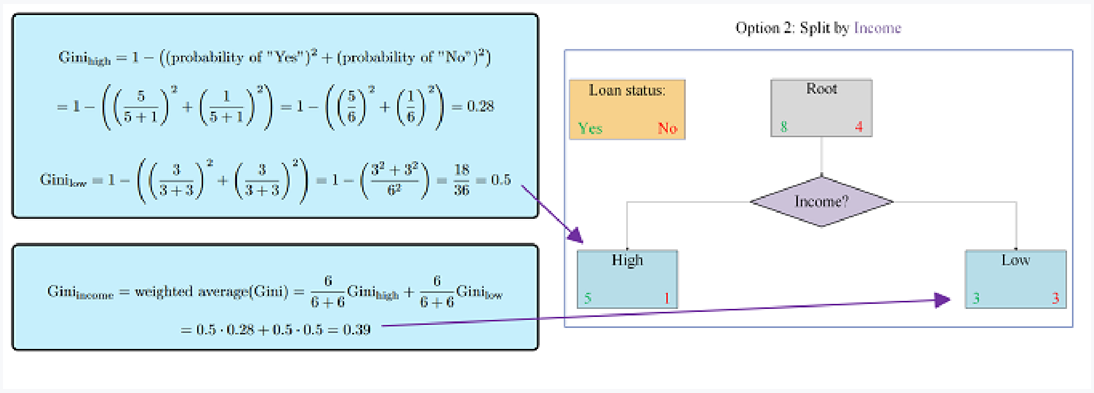
Similarly, we calculate the Gini for the **Medium** and **Low** classes

$$Gini_{medium} = 1 - \left(\left(\frac{3}{3+2}\right)^2 + \left(\frac{2}{3+2}\right)^2\right) = 1 - \left(\frac{3^2+2^2}{5^2}\right) = 1 - \frac{13}{25} = 0.48 \; Gini_{low} = 1 - \left(\left(\frac{1}{1+2}\right)^2 + \left(\frac{2}{1+2}\right)^2\right) = 1 - \left(\frac{1^2+2^2}{3^2}\right) = 1 - \frac{5}{9} = 0.45$$

The overall Gini index for the **Account** feature is the weighted average of the Gini indexes of its subsets, based on the number of examples in each subset.

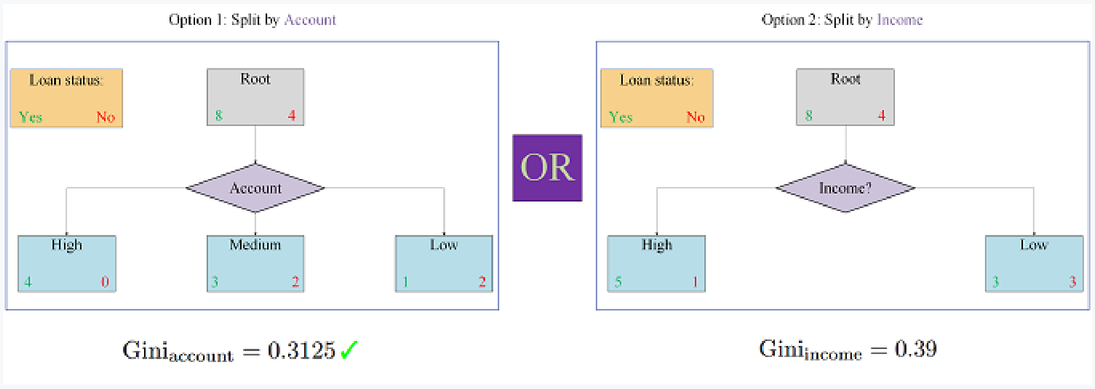
$$Gini_{aount} = weightedaverage(Gini) = \; = \frac{4}{4+5+3}Gini_{high} + \frac{5}{4+5+3}Gini_{medium} + \frac{3}{4+5+3}Gini_{low} = \; = \frac{4}{12}\cdot 0 + \frac{5}{12}\cdot 0.48 + \frac{3}{12}\cdot 0.45 = 0.3125$$

Similarly, calculate the Gini index for the **Income** feature using its subsets and probabilities.



$$\text{Gini}_{\text{high}} = 1 - ((\text{probability of "Yes"})^2 + (\text{probability of "No"})^2)$$
$$= 1 - \left(\left(\frac{5}{5+1}\right)^2 + \left(\frac{1}{5+1}\right)^2\right) = 1 - \left(\left(\frac{5}{6}\right)^2 + \left(\frac{1}{6}\right)^2\right) = 0.28$$
$$\text{Gini}_{\text{low}} = 1 - \left(\left(\frac{3}{3+3}\right)^2 + \left(\frac{3}{3+3}\right)^2\right) = 1 - \left(\frac{3^2+3^2}{6^2}\right) = \frac{18}{36} = 0.5$$
$$\text{Gini}_{\text{income}} = \text{weighted average(Gini)} = \frac{6}{6+6}\text{Gini}_{\text{high}} + \frac{6}{6+6}\text{Gini}_{\text{low}}$$
$$= 0.5 \cdot 0.28 + 0.5 \cdot 0.5 = 0.39$$

Finally, compare the Gini indexes for **Account** and **Income**. The feature with the lower Gini index is the better choice for splitting the data. In this example, the **Account** feature has a lower Gini index, making it the optimal choice for the split.


Option 1: Split by Account
Loan status: Yes No
Root 8 4
Account
High 4 0
Medium 3 2
Low 1 2
$\text{Gini}_{\text{account}} = 0.3125$ ✓
OR
Option 2: Split by Income
Loan status: Yes No
Root 8 4
Income?
High 5 1
Low 3 3
$\text{Gini}_{\text{income}} = 0.39$

6.1.8

What is the weighted average Gini index of a feature used for?

- Selecting the best feature for the split.
- Determining the number of classes in the dataset.
- Calculating the impurity of a feature's subsets.
- Measuring the entropy of the data.

6.1.9

Which statements about the Gini index are true?

- It is used to measure the impurity of data subsets.
- The Gini index is computed using probabilities of classes.
- A Gini index of 0 indicates maximum impurity.
- The feature with the higher Gini index is preferred for splitting.

6.2 Entropy

6.2.1

Entropy is a widely used measure to quantify impurity or disorder in a dataset. Borrowed from physics, it characterizes the degree of uncertainty or impurity in a system. In machine learning, entropy is a valuable metric to determine how well a dataset is split based on a specific feature during the formation of decision trees.

Given a dataset **S** with two classes (positive and negative examples), entropy is calculated as:

$\text{Entropy}(S) = - p_p \log_2 p_p - p_n \log_2 p_n$

Where:

- $\mathbf{p_p}$ - proportion of positive examples in **S**.
- $\mathbf{p_n}$ - proportion of negative examples in **S**.

Special rule: when $\mathbf{p_p}$ or $\mathbf{p_n}$ is zero, we define $\mathbf{0.\log_2(0) = 0}$, ensuring the calculation is valid.

Characteristics of Entropy:

- **Perfect purity** - if all examples in **S** belong to one class (e.g., all positive), entropy is 0. For instance if $\mathbf{p_p = 1}$ and $\mathbf{p_n = 0}$: $\mathbf{Entropy(S) = -1 \cdot \log_2(1) - 0 \cdot \log_2(0) = 0}$
- **Maximum uncertainty** - entropy reaches its maximum value of 1 when the dataset has an equal number of positive and negative examples ($p_p = p_n = 0.5$).
- **Intermediate values** when the dataset has uneven proportions of positive and negative examples, entropy lies between 0 and 1. For example, if 70% of examples are positive ($p_p = 0.7$) and 30% are negative ($p_n = 0.3$), entropy will reflect the degree of impurity.

Entropy helps in determining the best feature for splitting a dataset. Features that result in subsets with lower entropy (i.e., greater purity) are preferred, as they contribute to more effective classification.

By minimizing entropy at each decision node, we iteratively create a tree that splits the data into increasingly homogeneous subsets.

6.2.2

What does an entropy value of 0 indicate?

- All examples belong to a single class.
- Maximum disorder in the dataset.
- An equal number of positive and negative examples.
- Entropy cannot be zero.

6.2.3

Which of the following statements about entropy are true?

- Entropy measures the degree of impurity in a dataset.
- Entropy is used to identify the best feature for splitting datasets.
- Entropy is maximized when all examples are in one class.
- Entropy is 1 when the dataset has equal positive and negative examples.

6.2.4

Non-binary entropy

Entropy is a key metric in decision tree algorithms to evaluate the impurity of data splits. In the previous section, we introduced **binary entropy**, which is used for datasets with two classes. However, when datasets have more than two classes, we use a generalized form known as **non-binary entropy**.

Binary entropy is calculated for datasets with exactly two classes (e.g., positive and negative examples) using the formula:

Entropy(S) = $- p_p \log_2 p_p - p_n \log_2 p_n$

where p_p is the proportion of positive examples in S and p_n is the proportion of negative examples in S.

This entropy value quantifies the disorder or uncertainty in the dataset. It is specific to binary classification problems.

When a dataset has multiple classes (T>2), entropy is generalized to include all possible classes. The formula becomes:

--- ERROR ---

Where:

- p_t is a proportion of examples in class t relative to all classes T in the set H.
- T is total number of classes.

Non-binary entropy applies to datasets with multiple classes and accounts for the proportions of all classes. **Base of the Logarithm**:is always 2 because entropy measures the expected length of the encoding in bits. This aligns with the information-theoretic basis of entropy.

- An entropy of 0 means complete purity; all examples belong to one class.
- Higher entropy values indicate greater impurity or uncertainty due to a more balanced distribution across classes.

Both binary and non-binary entropy are used to evaluate potential splits in decision trees. The goal is to choose features that reduce entropy, creating subsets with higher purity and more homogeneous classes.

6.2.5

What does non-binary entropy measure?

- The impurity of data splits for multiple classes.
- The probability of positive examples in binary classification.
- The number of features in a dataset.
- The total number of examples in a dataset.

6.2.6

Which of the following are true about entropy in decision trees?

- Binary entropy applies to datasets with two classes.
- Entropy is used to measure impurity in data splits.
- Non-binary entropy uses the base-10 logarithm.
- Non-binary entropy is calculated using all class proportions.

6.2.7

Entropy quantifies the disorder or impurity in a dataset and is a fundamental concept in decision tree algorithms. Using a binary classification scenario, we can calculate entropy to measure the randomness in a given set of elements. Consider a binary entropy defined by the relation:

Entropy(S) = $- p_p \log_2 p_p - p_n \log_2 p_n$

If we have a set R1 containing 6 elements "**a**" and 2 elements "**b**".

R1 = {a, a, a, a, a, a, b, b}

We calculate the entropy for this set as follows:

Entropy(R1) = $-[6/8 * \log_2 6/8 + 2/8 * \log_2 2/8\,] = -(-0.3112 + (-0.5)) = 0.8112$

Next, consider the set R2, which has the same ratio of elements, but more elements of "**b**"

R2 = {a, a, b, b, b, b, b, b}

Entropy(R2) = -[2/8 * $\log_2$ 2/8 + 6/8 * $\log_2$ 6/8] = -(-0.5 + (-0.3112)) = 0.8112

Note that the entropy rate is the same. Thus, for entropy, it is not important which elements are more, what is important is the ratio of the number of elements in the set.

Next, we can consider the sets R3 and R4 and their entropies.

R3 = {a, b, b, b, b, b, b, b}

R4 = {a, a, a, b, b, b, b, b}

Entropy(R3) = -[1/8 * $\log_2$ 1/8 + 7/8 * $\log_2$ 7/8] = -(-0.375 + (-0.1686)) = 0.5436

Entropy(R4) = -[3/8 * $\log_2$ 3/8 + 5/8 * $\log_2$ 5/8] = -(-0.5306 + (-0.42374)) = 0.9543

Note that the set R3 is more ordered, i.e. it contains most of the elements from "**b**" and only one element from "**a**". Its entropy is therefore lower. Also note that the sets R1 and R2 have two elements different, therefore their entropy is greater than the entropy of R2 set, but less than the entropy of R4 set.

For completeness, we still present the calculation of the ideally ordered R5 set, i.e., the set with no impurity.

R5 = {a, a, a, a, a, a, a, a}

Entropy(R5) = -[8/8 * $\log_2$ 8/8 + 0/8 * $\log_2$ 0/8] = -(0 + 0) = 0

For comparison, we also present the most disordered set for 8 elements, the R6 set.

R5 = {a, a, a, a, b, b, b, b}

Entropy(R5) = -[4/8 * $\log_2$ 4/8 + 4/8 * $\log_2$ 4/8] = -(0.5*(-1) + 0.5*(-1)) = 1

Entropy increases with greater impurity in the dataset and decreases as the dataset becomes more ordered. It is a critical measure for identifying the best splits in decision tree algorithms.

6.2.8

What does entropy measure in a dataset?

- The degree of disorder or impurity.
- The sum of positive and negative examples.

- The total number of elements.
- The ratio of positive examples only.

6.2.9

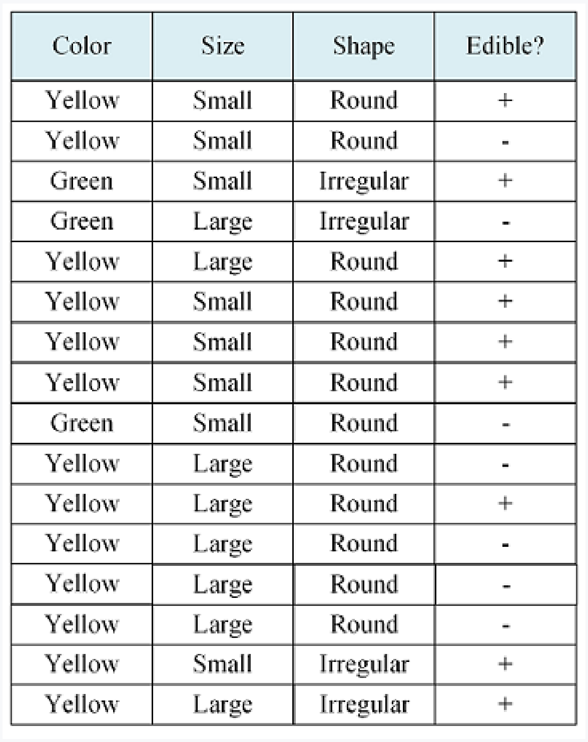
Which statements about entropy are correct?

- Entropy is 0 for an ideally ordered set.
- Entropy reaches its maximum when all classes are equally distributed.
- Entropy is independent of the proportions of classes in the dataset.
- Entropy values range from 0 to 1.

6.2.10

Practical example

Entropy measures the disorder or impurity in a dataset. Let's apply the entropy formula to a practical example involving a target variable **Edible** with 16 examples: 9 positive (+) and 7 negative (−).



Color	Size	Shape	Edible?
Yellow	Small	Round	+
Yellow	Small	Round	-
Green	Small	Irregular	+
Green	Large	Irregular	-
Yellow	Large	Round	+
Yellow	Small	Round	+
Yellow	Small	Round	+
Yellow	Small	Round	+
Green	Small	Round	-
Yellow	Large	Round	-
Yellow	Large	Round	+
Yellow	Large	Round	-
Yellow	Large	Round	-
Yellow	Large	Round	-
Yellow	Small	Irregular	+
Yellow	Large	Irregular	+

We calculate the entropy for the **Edible** variable as follows:

$\mathbf{Entropy_{Edible} = -(9/16 * \log_2 9/16 + 7/16 * \log_2 7/16) = 0.9836}$

The result of 0.9836 is a number very close to 1. Thus, it means that the set has a high degree of disorder.

6.2.11

Which are correct about the given entropy calculation?

- The target variable has more positive than negative examples.
- An entropy value of 0.98360.9836 suggests a balanced dataset.
- Entropy is calculated using base-10 logarithms.
- Entropy is directly proportional to the total number of examples.

6.3 Information Gain

6.3.1

In decision tree algorithms, **entropy** plays a vital role in determining the most appropriate attribute for splitting the dataset. However, entropy is not used directly; instead, it is applied as part of the **information gain** measure.

Information gain quantifies the expected reduction in entropy resulting from the distribution of examples based on a particular attribute. It helps identify the attribute that best separates the dataset into distinct classes.

The **Information Gain (S, A)** for an attribute AAA with respect to a set of examples SSS is defined as:

$$InformationGain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} \cdot Entropy(S_v)$$

Where:

- **Entropy(S)** is an entropy of the dataset **S**.
- **Values(A)** is set of all possible values of attribute **A**.
- **S_v** - subset of **S** where attribute **A** has value **v**.
- **$|S_v|/|S|$** is proportion of examples in **S_v**.

Steps:

1. Calculate the total entropy of the entire dataset **S**.

2. Split by attribute into subsets based on the values of attribute **A**.
3. Weighted entropy is calculated the entropy of each subset **S_v**, weighted by its proportion **$|S_v|/|S|$**.
4. Subtract weighted entropy of subsets from the total entropy to get the information gain.

The attribute with the highest information gain is selected for splitting because it provides the greatest reduction in uncertainty (disorder) in the dataset.

6.3.2

What does the information gain measure in decision trees represent?

- The expected reduction in entropy after splitting on an attribute.
- The amount of disorder in the dataset.
- The proportion of examples in each class.
- The total entropy of the dataset.

6.3.3

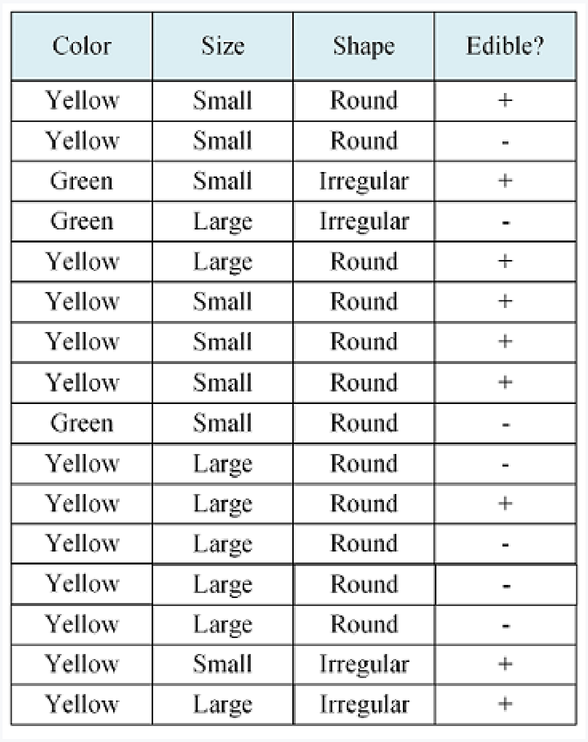
Which of the following are true about information gain?

- It requires entropy to be calculated.
- It helps select the best attribute for splitting.
- It is always a positive value.
- It is used to calculate the GINI index.

6.3.4

Example

In the previous section, we worked with the following dataset. We will show how to calculate the **information gain** for an attribute, using the **Size** attribute in a dataset. The target variable is **Edible**, with the 9 positive examples (+) and 7 negative examples (-), for a total of 16 examples.



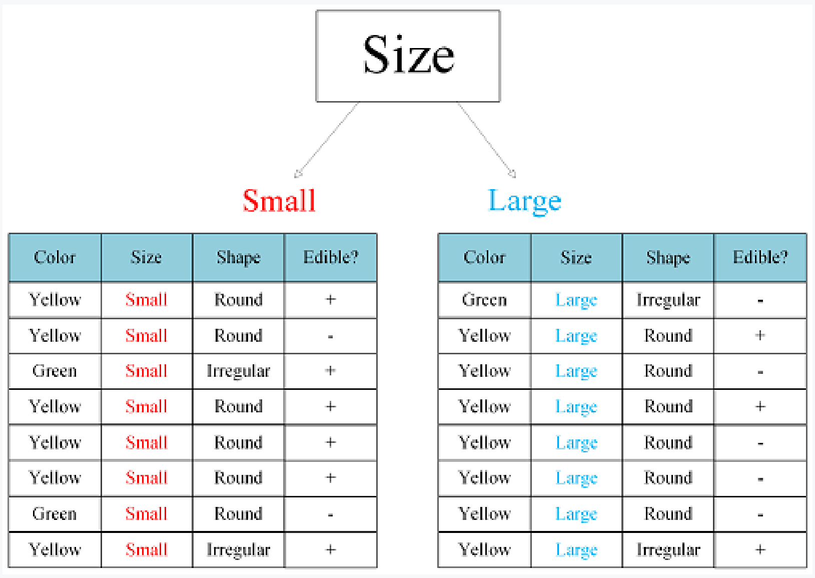
Color	Size	Shape	Edible?
Yellow	Small	Round	+
Yellow	Small	Round	-
Green	Small	Irregular	+
Green	Large	Irregular	-
Yellow	Large	Round	+
Yellow	Small	Round	+
Yellow	Small	Round	+
Yellow	Small	Round	+
Green	Small	Round	-
Yellow	Large	Round	-
Yellow	Large	Round	+
Yellow	Large	Round	-
Yellow	Large	Round	-
Yellow	Large	Round	-
Yellow	Small	Irregular	+
Yellow	Large	Irregular	+

In the previous section, we calculated the entropy for the target variable **Edible**:

Entropy$_{\text{edible}}$ = - (9/16 * log$_2$ 9/16 + 7/16 * log$_2$ 7/16) = 0.9836

When we split the dataset by the attribute **Size**, we obtain the following subsets:

- Subset for "small" contains 8 examples, with 6 positive (+) and 2 negative (-).
- Subset for "large" contains 8 examples, with 3 positive (+) and 5 negative (-).



Color	Size	Shape	Edible?
Yellow	Small	Round	+
Yellow	Small	Round	-
Green	Small	Irregular	+
Yellow	Small	Round	+
Yellow	Small	Round	+
Yellow	Small	Round	+
Green	Small	Round	-
Yellow	Small	Irregular	+

Color	Size	Shape	Edible?
Green	Large	Irregular	-
Yellow	Large	Round	+
Yellow	Large	Round	-
Yellow	Large	Round	+
Yellow	Large	Round	-
Yellow	Large	Round	-
Yellow	Large	Round	-
Yellow	Large	Irregular	+

We calculate the entropy for each subset:

- **$\text{Entropy}_{\text{size = small}}$ = - (6/8 * $\log_2$ 6/8 + 2/8 * $\log_2$ 2/8) = - (0.75 * $\log_2$ 0.75 + 0.25 * $\log_2$ 0.25) = 0.8113**
- **$\text{Entropy}_{\text{size = large}}$ = - (3/8 * $\log_2$ 3/8 + 5/8 * $\log_2$ 5/8) = 0.9544**

The total entropy for the **Size** attribute is the weighted average of the entropies of its subsets:

- **$\text{Entropy}_{\text{size}}$ = 8/16 * 0.8113 + 8/16 * 0.9544 = 0.8828**

Finally, we compute the information gain (or entropy reduction) of selecting the **Size** attribute, which we compute as the reduction of the entropy of the original dataset by the entropy of the **Size** attribute. The **information gain** is the reduction in entropy after the split:

- **InfGain(attrib) = Entropy(parent) - Entropy(attrib)**
- **InfGain(size) = Entropy(Edible) - Entropy(size) = 0.9836 - 0.8828 = 0.1008**

So, we obtained 0.1008 bits of information about the dataset by selecting "size" as the first branch of our decision tree.

6.3.5

What does the information gain for the attribute "Size" indicate?

- The reduction in entropy due to splitting by "Size."
- The entropy of the dataset after splitting by "Size."
- The amount of disorder in the "Size" attribute.
- The weighted average entropy of all subsets.

6.3.6

Which of the following are steps to calculate information gain?

- Compute the entropy of the entire dataset.
- Split the dataset by the attribute and compute subset entropies.
- Calculate the Gini index for the attribute.
- Use the ratio of classes to compute the weighted average entropy.

6.3.7

How is the probability of an object being assigned to a particular class calculated in a dataset?

- By dividing the number of objects in a specific class by the total number of objects in the dataset.
- By dividing the total number of objects in the dataset by the number of classes.
- By multiplying the total number of objects in the dataset by the number of classes.
- By using the logarithm of the total number of objects in a specific class.

6.4 How to use numeral values?

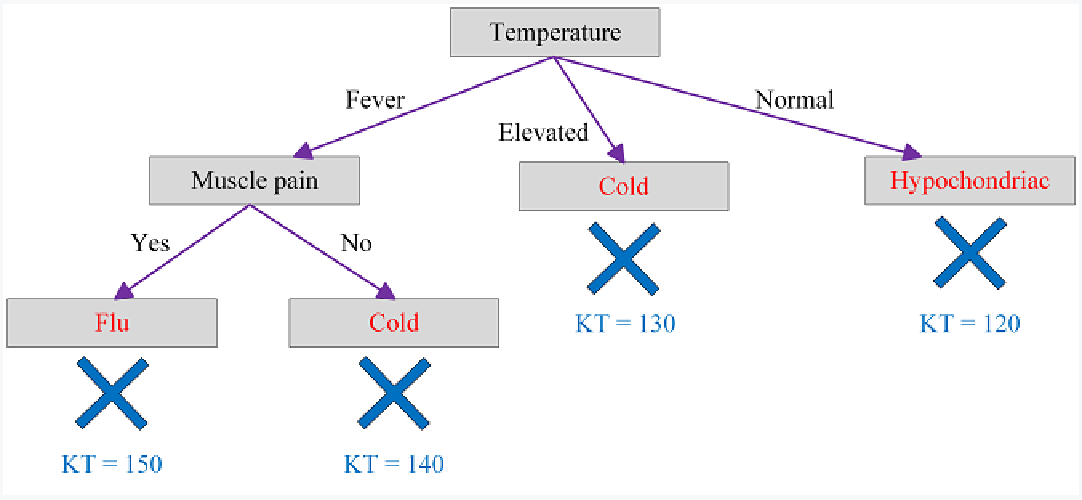
6.4.1

Decision trees are a versatile machine learning method capable of handling both categorical and numeric (continuous) data. While previous examples have focused on categorical data, it is essential to understand how decision trees process continuous variables, whether in the target variable or the individual attributes.

When the **target variable** is continuous, decision trees are referred to as **regression trees**. Unlike classification trees, which model nominal outcomes (e.g., "flu," "cold," "hypochondria"), regression trees predict a continuous outcome, such as blood pressure or temperature. In this case, the predicted value at each leaf node is usually the average of the target variable for the examples that fall into that node.

To determine the best split when working with continuous target variables, regression trees use metrics like the **standard deviation reduction (SDR)**. This metric helps identify splits that minimize the variability within subsets, ensuring that the resulting groups are as homogeneous as possible in terms of the target variable.

Continuous data can also appear in the **individual attributes**. The decision tree must decide how to split the data what is typically achieved by choosing a threshold value (e.g., "Age < 30") that maximizes information gain or minimizes the Gini index. The splitting process transforms continuous attributes into binary splits, enabling the tree to process them effectively.



6.4.2

What value do the regression trees model?

- continuous
- nominal
- categorical
- absent

6.4.3

What is typically used as the node selection metric in regression trees?

- Standard Deviation Reduction
- Gini Index
- Information Gain
- Entropy

6.4.4

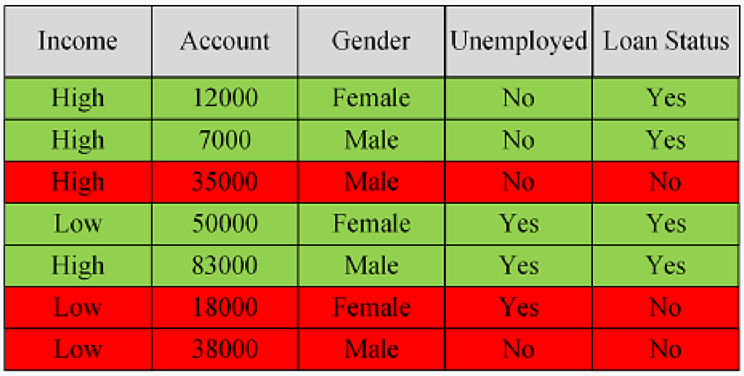
How are splits typically handled for continuous attributes in decision trees?

- By transforming them into binary splits using thresholds.
- By selecting a threshold value that maximizes information gain or minimizes impurity.
- By treating them as categorical data.
- By maximizing the standard deviation within subsets.

6.4.5

Use of continuous values

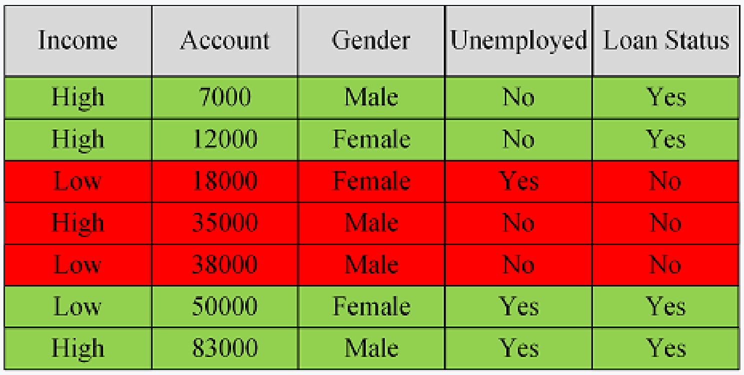
Let's explore how decision trees handle continuous values in attributes, specifically by calculating the **Gini index**. Consider the following dataset of loan applicants, where the attribute **Account** is continuous.



Income	Account	Gender	Unemployed	Loan Status
High	12000	Female	No	Yes
High	7000	Male	No	Yes
High	35000	Male	No	No
Low	50000	Female	Yes	Yes
High	83000	Male	Yes	Yes
Low	18000	Female	Yes	No
Low	38000	Male	No	No

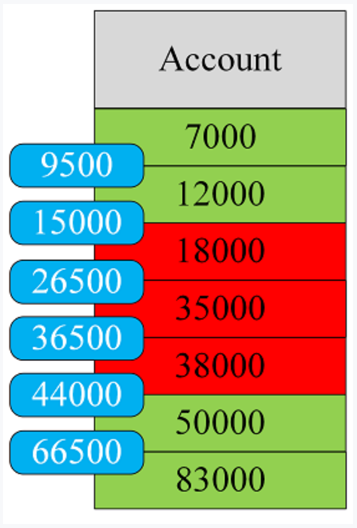
The **Account** variable is continuous. It is possible to calculate the degree of disorder for this attribute. In this example, we will use the Gini index as the measure of disorder. Therefore, we will calculate the Gini index of the **Account** attribute.

The first step will be to sort the dataset according to the values of the attribute **Account**.

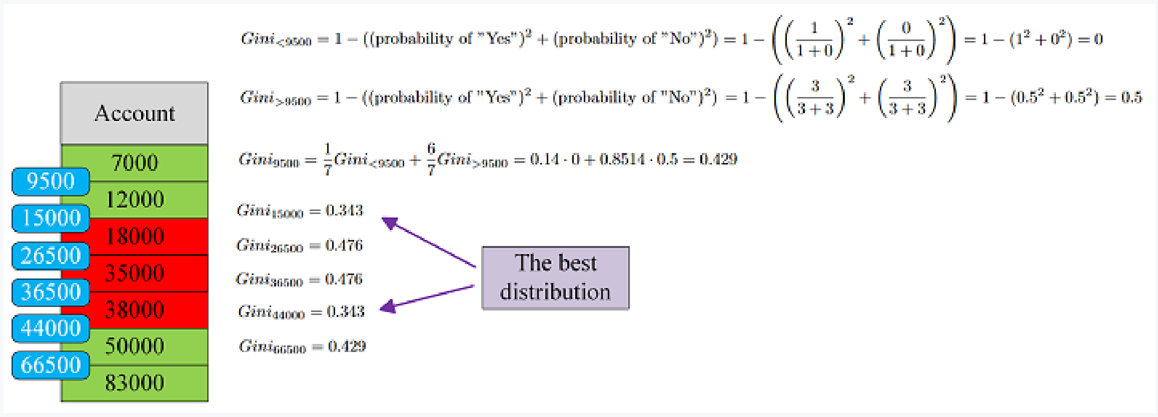


Income	Account	Gender	Unemployed	Loan Status
High	7000	Male	No	Yes
High	12000	Female	No	Yes
Low	18000	Female	Yes	No
High	35000	Male	No	No
Low	38000	Male	No	No
Low	50000	Female	Yes	Yes
High	83000	Male	Yes	Yes

In the second step, we calculate the average values of the individual data



Next, we will calculate GINI for each distribution



$$Gini_{<9500} = 1 - ((\text{probability of "Yes"})^2 + (\text{probability of "No"})^2) = 1 - \left(\left(\frac{1}{1+0}\right)^2 + \left(\frac{0}{1+0}\right)^2\right) = 1 - (1^2 + 0^2) = 0$$

$$Gini_{>9500} = 1 - ((\text{probability of "Yes"})^2 + (\text{probability of "No"})^2) = 1 - \left(\left(\frac{3}{3+3}\right)^2 + \left(\frac{3}{3+3}\right)^2\right) = 1 - (0.5^2 + 0.5^2) = 0.5$$

$$Gini_{9500} = \frac{1}{7}Gini_{<9500} + \frac{6}{7}Gini_{>9500} = 0.14 \cdot 0 + 0.8514 \cdot 0.5 = 0.429$$

$$Gini_{15000} = 0.343$$

$$Gini_{26500} = 0.476$$

$$Gini_{36500} = 0.476$$

$$Gini_{44000} = 0.343$$

$$Gini_{66500} = 0.429$$

Final Gini index for the column **Account** = 0.343. It is the lower Gini index of all distributions of the dataset according to average values.

6.4.6

What is the first step in calculating the Gini index for continuous attributes?

- Sort the attribute values in ascending order
- Calculate the Gini index for each subset
- Use average values as thresholds
- Split the dataset into two groups

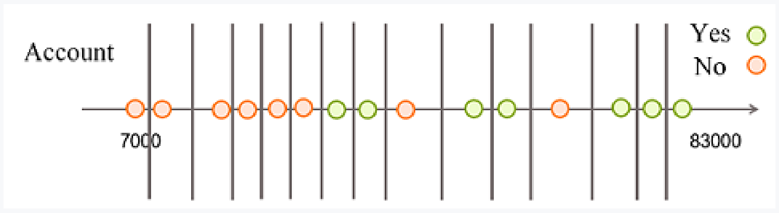
6.4.7

Review

When working with continuous attributes in decision trees, we need a method to split the data at meaningful points based on the attribute's values. The **Gini index** is used to measure the impurity or disorder in the dataset for each split. Below is a more detailed explanation of how to calculate the Gini index for continuous attributes.

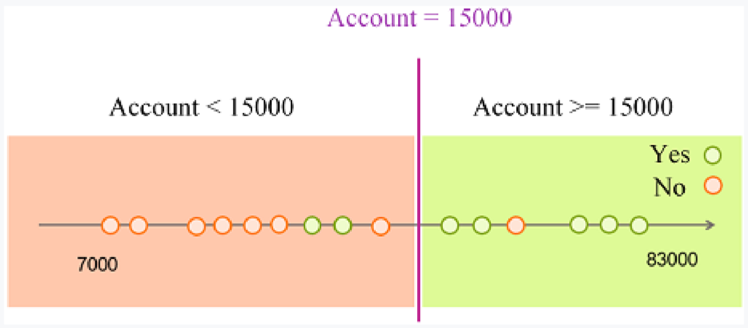
1. To begin, we **sort the dataset** by the continuous attribute in ascending order. Sorting is essential because it allows us to consider different threshold values between consecutive values for splitting the dataset.

2. Next, calculate the **average values** between consecutive attribute values. These averages will be used as possible split points



3. Next, only the calculated average values are considered

4. For each split, calculate the **Gini index** for the resulting subsets. Finally, select the **best split** by choosing the one with the **lowest Gini index**. The split that minimizes the Gini index will be the most informative, as it leads to the greatest reduction in impurity.



6.4.8

Why do we calculate the average values between consecutive pairs of sorted attribute values?

- To use them as thresholds for splitting the data
- To select the best subset

- To calculate the Gini index for the entire dataset
- To sort the dataset

6.4.9

What does the Gini index measure in decision trees?

- The impurity or disorder of the dataset
- The accuracy of the classification
- The number of positive examples
- The total size of the dataset

6.4.10

Which split is selected in decision trees when using the Gini index?

- The split with the lowest Gini index
- The split with the highest Gini index
- The split with the most examples
- The split with the most distinct attribute values

6.5 Decission trees strengths and weaknesses

6.5.1

Decision trees are a popular machine learning method due to their simplicity and interpretability. However, they also have certain limitations. Here, we summarize their strengths and weaknesses.

Strengths:

- Understandable rules - becision trees generate rules that are easy to interpret and explain.
- Low computational requirements for classification - once trained, decision trees classify data quickly and efficiently.
- Versatility - decision trees can handle both continuous and categorical variables.
- Feature importance - they clearly highlight the most important features used for prediction or classification.

Weaknesses:

- Decision trees may struggle with classification problems that have many classes but relatively few training examples.
- High training cost:

1. computationally expensive because each potential split must be evaluated to find the best one
2. in some cases, algorithms must search for the optimal combination of weights for fields
3. pruning, which simplifies the tree, can also be complex as it requires forming and comparing many candidate subtrees.

6.5.2

What is one major strength of decision trees?

- They are easy to interpret and explain.
- They always achieve perfect accuracy.
- They require high computational power for classification.
- They are only applicable to categorical variables.

6.5.3

Which of the following are the weaknesses of decision trees?

- They struggle with problems that have many classes and few training examples.
- They cannot handle continuous variables.
- They cannot indicate the most important features for prediction.
- They require complex models for classification.

6.5.4

Why can the training process for decision trees be computationally expensive?

- Each split candidate must be sorted and evaluated to find the best distribution.
- The classification phase requires high computational resources.
- Decision trees require manual tuning at each step.
- Decision trees cannot handle pruning or optimization.

6.5.5

What is a key advantage of decision trees when compared to some other algorithms?

- They can work seamlessly with both categorical and continuous variables.
- They require no feature selection process.
- They can handle missing data better than all other algorithms.
- They always outperform ensemble methods.

6.6 Practical tasks

6.6.1

Project: Titanic survival prediction

Use a decision tree algorithm to predict whether a person would survive the Titanic based on features **Pclass**, **Sex**, **Age**, **SibSp**, **Parch**.

- Train the model using the Titanic dataset and split the data into training and testing sets (70% train, 30% test).
- Evaluate the model's predictions on the test set and calculate its accuracy.

Dataset (copy): https://priscilla.fitped.eu/data/pandas/titanic.csv

We will create a decision tree that predicts whether or not a person would survive the Titanic just like in the previous sections. Let's calculate the accuracy of this model.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
import warnings
warnings.filterwarnings('ignore')

# Load Titanic dataset from the specified URL
data =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/titanic.c
sv')

# Select relevant columns for the model
data = data[['Survived', 'Pclass', 'Sex', 'Age', 'SibSp',
'Parch']]
```

```
# Remove rows with missing values to ensure clean data
data = data.dropna()

# Convert the 'Sex' column to numerical format: male -> 0,
female -> 1
pd.set_option('future.no_silent_downcasting', True)
data['Sex'] = data['Sex'].replace({'male': 0, 'female': 1})

# Define features (X) by excluding the target variable
'Survived'
X = data[data.columns.difference(['Survived'])]
# Define the target variable (y) as 'Survived'
y = data['Survived']

# Split the dataset into training and testing sets (70% train,
30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Initialize the DecisionTreeClassifier with a fixed random
state for reproducibility
clf = DecisionTreeClassifier(random_state=42)
# Train the classifier using the training data
clf = clf.fit(X_train, y_train)

# Predict the survival outcomes for the test dataset
y_pred = clf.predict(X_test)

from sklearn.metrics import accuracy_score
# Calculate and print the accuracy of the model
print(accuracy_score(y_test, y_pred))
```

Program output:

```
0.7767441860465116
```

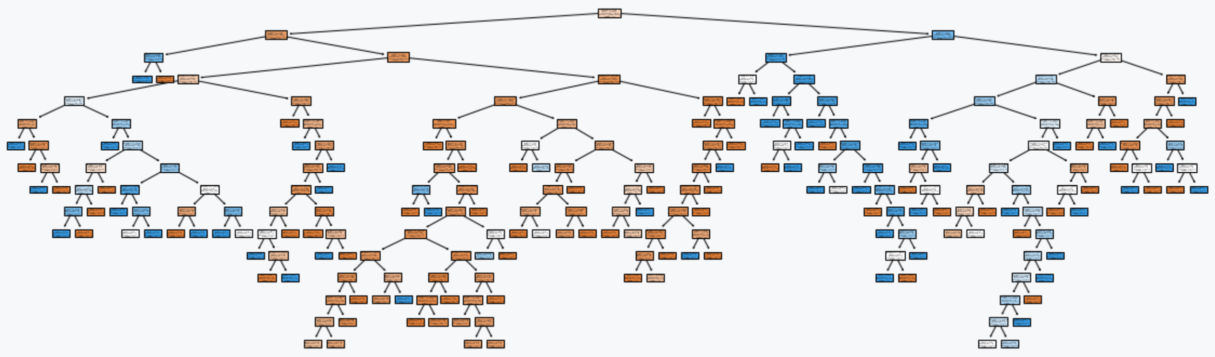
We found out that 77.67 % of cases are classified correctly.

With the following code, we can visualize the tree.

```
import matplotlib.pyplot as plt
from sklearn import tree
fig = plt.figure(figsize=(20,6))
```

```
_ = tree.plot_tree(clf,
                   feature_names = ['Pclass', 'Sex', 'Age',
'SibSp', 'Parch'],
                   class_names=['0','1'],
                   filled=True)
```

Program output:



We can see that the tree is complex. Let's change the default criterion for building the tree from *gini* to *entropy* and show the accuracy of the model.

```
# Initialize a DecisionTreeClassifier using the Gini criterion
dtree = DecisionTreeClassifier(criterion='gini',
random_state=42)
# Train the classifier on the training data
dtree.fit(X_train, y_train)
# Predict the survival outcomes for the test dataset using
Gini
pred = dtree.predict(X_test)
# Print the accuracy of the model using the Gini criterion
print('Criterion=gini', accuracy_score(y_test, pred))

# Initialize a DecisionTreeClassifier using the Entropy
criterion
dtree = DecisionTreeClassifier(criterion='entropy',
random_state=42)
# Train the classifier on the training data
dtree.fit(X_train, y_train)
# Predict the survival outcomes for the test dataset using
Entropy
pred = dtree.predict(X_test)
# Print the accuracy of the model using the Entropy criterion
print('Criterion=entropy', accuracy_score(y_test, pred))
```

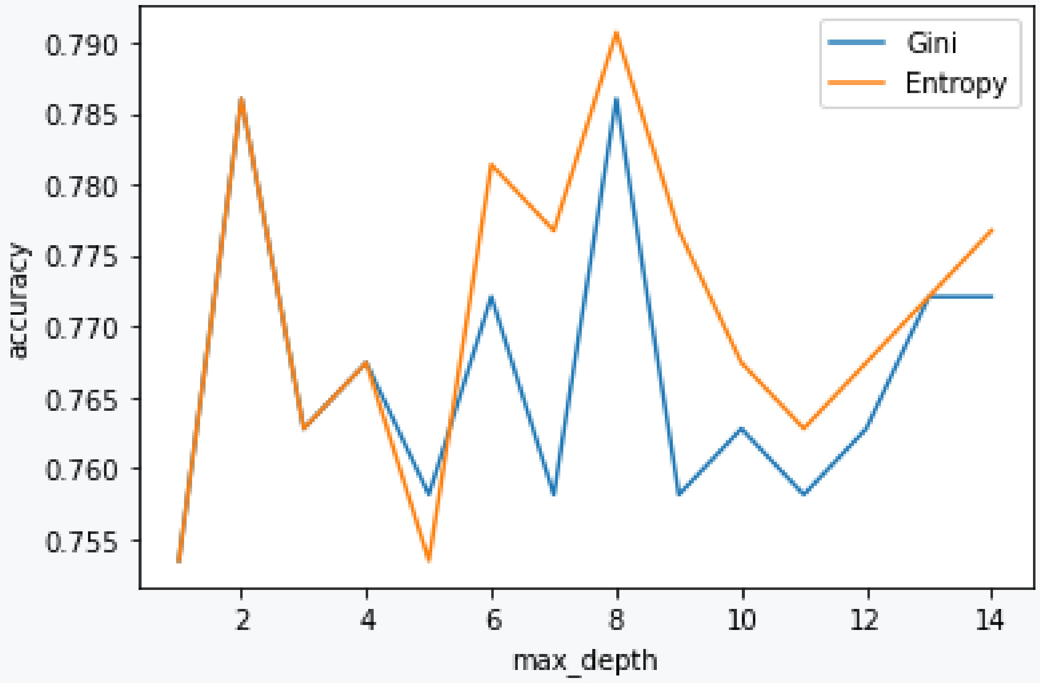
Program output:
```
Criterion=gini 0.7767441860465116
Criterion=entropy 0.7767441860465116
```

With gini (the default criterion for generating the tree) the accuracy is 77.67 %, and with entropy the accuracy is also 77.67 %.

- Let's see whether pruning the tree by changing the maximum depth of the tree gives us better results in any scenario.
- Let's create trees sequentially from depth 1 to 15 and visualize the results.

```
# Initialize lists to store the max_depth values and
corresponding accuracies
max_depth = []
acc_gini = []
acc_entropy = []

# Iterate over a range of max_depth values from 1 to 14
for i in range(1, 15):
    # Create and train a DecisionTreeClassifier using Gini
criterion with the current max_depth
    dtree = DecisionTreeClassifier(criterion='gini',
max_depth=i, random_state=42)
    dtree.fit(X_train, y_train)
    # Predict outcomes on the test set and append the accuracy
to acc_gini
    pred = dtree.predict(X_test)
    acc_gini.append(accuracy_score(y_test, pred))

    ####
    # Create and train a DecisionTreeClassifier using Entropy
criterion with the current max_depth
    dtree = DecisionTreeClassifier(criterion='entropy',
max_depth=i, random_state=42)
    dtree.fit(X_train, y_train)
    # Predict outcomes on the test set and append the accuracy
to acc_entropy
    pred = dtree.predict(X_test)
    acc_entropy.append(accuracy_score(y_test, pred))

    ####
    # Append the current max_depth value to the max_depth list
    max_depth.append(i)

# Combine the results into a DataFrame for visualization
d = pd.DataFrame({
    'acc_gini': pd.Series(acc_gini),
    'acc_entropy': pd.Series(acc_entropy),
    'max_depth': pd.Series(max_depth)
})

# Visualize the accuracy as a function of max_depth for both
criteria
plt.plot('max_depth', 'acc_gini', data=d, label='Gini')  #
Plot accuracy for Gini
plt.plot('max_depth', 'acc_entropy', data=d, label='Entropy')
# Plot accuracy for Entropy
plt.xlabel('max_depth')  # Label x-axis as max_depth
plt.ylabel('accuracy')  # Label y-axis as accuracy
plt.legend()  # Add a legend to distinguish Gini and Entropy
plots
plt.show()  # Display the plot
```

Program output:



On this graph we can see that by choosing entropy and tree depth of 7, we get the best accuracy of the model. Let's calculate the accuracy of the mentioned model.

```
# Initialize the DecisionTreeClassifier with 'entropy' as the
criterion and max_depth set to 8
clf = DecisionTreeClassifier(criterion='entropy', max_depth=8,
random_state=42)

# Fit the model to the training data
clf = clf.fit(X_train, y_train)

# Predict outcomes on the test set
y_pred = clf.predict(X_test)

# Calculate and print the accuracy of the predictions
from sklearn.metrics import accuracy_score
print(accuracy_score(y_test, y_pred))  # Display the accuracy
score
```

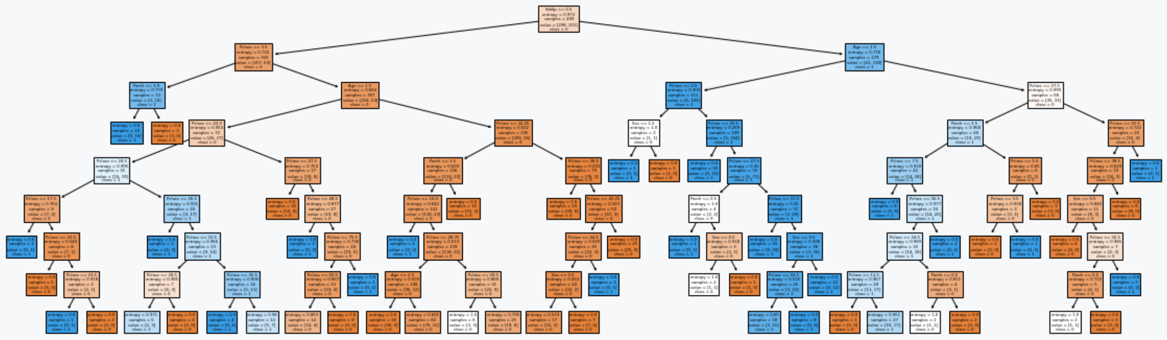
Program output:

```
0.7906976744186046
```

The accuracy of a decision tree with a depth of 8 and built by entropy has reached 79.07%. We increased the accuracy of the model by 1.4 % just by setting the function to create tree branches and limiting the depth of the tree. The built tree is less complex.

```
import matplotlib.pyplot as plt
from sklearn import tree
fig = plt.figure(figsize=(20,6))
_ = tree.plot_tree(clf,
                   feature_names = ['Pclass', 'Sex', 'Age',
'SibSp', 'Parch'],
                   class_names=['0','1'],
                   filled=True)
```

Program output:



6.6.2 Number of leaves after pruning

Find the number of leaves before and after pruning the tree and print it. Use the creation of a decision tree:

```
DecisionTreeClassifier(criterion = 'entropy', max_depth = 8,
random_state=42)
```

Print 2 values divided with a space

Create the trees with parameter of random_state=42

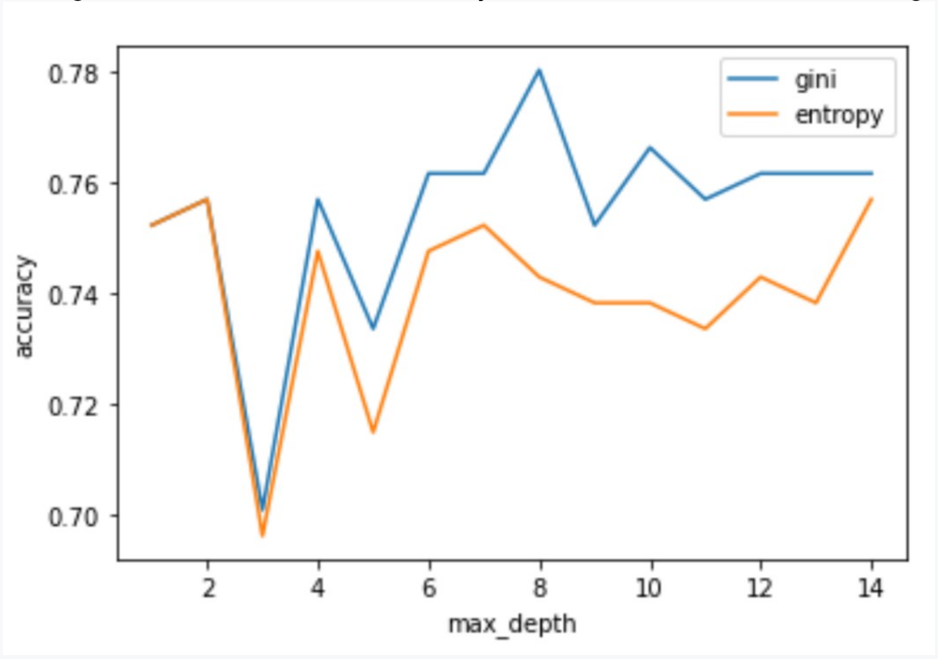
file1.py

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

data =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/titanic.c
sv')

data = data[['Survived', 'Pclass', 'Sex', 'Age', 'SibSp',
'Parch']]

data = data.dropna()

data['Sex'] = data['Sex'].replace({'male': 0, 'female': 1})

X = data[data.columns.difference(['Survived'])]
y = data['Survived']

```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

6.6.3

What setting of the decision tree would you use based on the following graph?



```
clf = DecisionTreeClassifier(criterion='gini', max_depth=10,
random_state = 42)
clf = DecisionTreeClassifier(criterion='gini', max_depth=8,
random_state = 42)
clf = DecisionTreeClassifier(criterion='entropy',
max_depth=10, random_state = 42)
clf = DecisionTreeClassifier(criterion='entropy', max_depth=8,
random_state = 42)
```

6.6.4

Project: Regression decision tree

Create a regression decision tree to predict the age of an opossum based on its features, such as **hdlngth**, **skullw**, **totlngth**, **taill**, **footlgth**, **earconch**, **eye**, **chest**, **belly**. Use a dataset with these characteristics, split it into training and testing sets, and train the model.

Dataset:

- original: Lindenmayer, D. B., Viggers, K. L., Cunningham, R. B., and Donnelly, C. F. 1995. Morphological variation among columns of the mountain brushtail possum, Trichosurus caninus Ogilby (Phalangeridae: Marsupialia). Australian Journal of Zoology 43: 449-458.
- local: https://priscilla.fitped.eu/data/machine_learning/possum.csv

```
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

# Load the dataset from the given URL into a pandas DataFrame
df =
pd.read_csv('https://priscilla.fitped.eu/data/machine_learning
/possum.csv')

# Display the first few rows of the dataset to inspect its
structure and contents
print(df)

# Print the information about the dataset, including column
names, non-null values, and data types
print(df.info())
```

Program output:

```
     case  site    Pop sex  age  hdlngth  skullw  totlngth
taill  footlgth  \
0       1     1    Vic   m  8.0     94.1    60.4      89.0
36.0      74.5
1       2     1    Vic   f  6.0     92.5    57.6      91.5
36.5      72.5
2       3     1    Vic   f  6.0     94.0    60.0      95.5
39.0      75.4
3       4     1    Vic   f  6.0     93.2    57.1      92.0
38.0      76.1
4       5     1    Vic   f  2.0     91.5    56.3      85.5
36.0      71.0
..    ...   ...    ...  ..  ...      ...     ...       ...
...       ...
99    100     7  other   m  1.0     89.5    56.0      81.5
36.5      66.0
100   101     7  other   m  1.0     88.6    54.7      82.5
39.0      64.4
```

```
101   102      7  other   f  6.0      92.4    55.0       89.0
38.0      63.5
102   103      7  other   m  4.0      91.5    55.2       82.5
36.5      62.9
103   104      7  other   f  3.0      93.6    59.9       89.0
40.0      67.6

     earconch   eye  chest  belly
0        54.5  15.2   28.0   36.0
1        51.2  16.0   28.5   33.0
2        51.9  15.5   30.0   34.0
3        52.2  15.2   28.0   34.0
4        53.2  15.1   28.5   33.0
..        ...   ...    ...    ...
99       46.8  14.8   23.0   27.0
100      48.0  14.0   25.0   33.0
101      45.4  13.0   25.0   30.0
102      45.9  15.4   25.0   29.0
103      46.0  14.8   28.5   33.5

[104 rows x 14 columns]

RangeIndex: 104 entries, 0 to 103
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   case      104 non-null    int64
 1   site      104 non-null    int64
 2   Pop       104 non-null    object
 3   sex       104 non-null    object
 4   age       102 non-null    float64
 5   hdlngth   104 non-null    float64
 6   skullw    104 non-null    float64
 7   totlngth  104 non-null    float64
 8   taill     104 non-null    float64
 9   footlgth  103 non-null    float64
 10  earconch  104 non-null    float64
 11  eye       104 non-null    float64
 12  chest     104 non-null    float64
 13  belly     104 non-null    float64
dtypes: float64(10), int64(2), object(2)
memory usage: 11.5+ KB
None
```

The data file contains records about 104 opossums. The records contain the age of the opossum, sex, length of the head, legs, etc.

For age feature, 2 data are missing and for footlgth feature one data is missing, we will remove these records.

```
# Remove rows with any missing values from the dataset
df = df.dropna()

# Print the updated dataset information to confirm that no
missing values remain
print(df.info())
```

Program output:

```

Index: 101 entries, 0 to 103
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   case      101 non-null    int64
 1   site      101 non-null    int64
 2   Pop       101 non-null    object
 3   sex       101 non-null    object
 4   age       101 non-null    float64
 5   hdlngth   101 non-null    float64
 6   skullw    101 non-null    float64
 7   totlngth  101 non-null    float64
 8   taill     101 non-null    float64
 9   footlgth  101 non-null    float64
 10  earconch  101 non-null    float64
 11  eye       101 non-null    float64
 12  chest     101 non-null    float64
 13  belly     101 non-null    float64
dtypes: float64(10), int64(2), object(2)
memory usage: 11.8+ KB
None
```

We have 101 records left to work with. We will prepare our features and target value.

Features are all numeral characteristics of the animal; the target value is the age of the opossum.

```
# Select features (excluding the target variable and
irrelevant columns) for prediction
X = df.drop(["case", "site", "Pop", "sex", "age"], axis=1)

# Set the target variable (age) to be predicted
y = df["age"]

# Print the features and the target variable to confirm the
selection
print(X)
print(y)
```

Program output:

```
     hdlngth  skullw  totlngth  taill  footlgth  earconch
eye  chest  belly
0       94.1    60.4      89.0   36.0      74.5      54.5
15.2   28.0   36.0
1       92.5    57.6      91.5   36.5      72.5      51.2
16.0   28.5   33.0
2       94.0    60.0      95.5   39.0      75.4      51.9
15.5   30.0   34.0
3       93.2    57.1      92.0   38.0      76.1      52.2
15.2   28.0   34.0
4       91.5    56.3      85.5   36.0      71.0      53.2
15.1   28.5   33.0
..       ...     ...       ...    ...       ...       ...
...    ...    ...
99      89.5    56.0      81.5   36.5      66.0      46.8
14.8   23.0   27.0
100     88.6    54.7      82.5   39.0      64.4      48.0
14.0   25.0   33.0
101     92.4    55.0      89.0   38.0      63.5      45.4
13.0   25.0   30.0
102     91.5    55.2      82.5   36.5      62.9      45.9
15.4   25.0   29.0
103     93.6    59.9      89.0   40.0      67.6      46.0
14.8   28.5   33.5

[101 rows x 9 columns]
0      8.0
1      6.0
2      6.0
3      6.0
4      2.0
      ...
```

```
99     1.0
100    1.0
101    6.0
102    4.0
103    3.0
Name: age, Length: 101, dtype: float64
```

We will divide the data into training and testing, similar to the standard decision tree.

We will use an 80:20 distribution

```
from sklearn.model_selection import train_test_split
# Split the dataset into training and testing sets, with 80%
for training and 20% for testing
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

All that remains is to build a regression decision tree model

```
from sklearn.tree import DecisionTreeRegressor

# Initialize the DecisionTreeRegressor model with a fixed
random state for reproducibility
model = DecisionTreeRegressor(random_state=42)

# Train the model on the training data
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)

# Print the actual ages from the test set
print('real age')
print(y_test.to_numpy())

# Separator for readability
print('-----')

# Print the predicted ages
print('predicted age')
print(y_pred)
```

Program output:

```
real age
[2. 2. 7. 6. 4. 3. 4. 5. 9. 8. 5. 3. 1. 2. 3. 2. 3. 4. 5. 4.
1.]
-----
predicted age
[3. 3. 6. 6. 4. 2. 7. 2. 6. 6. 2. 3. 2. 3. 3. 4. 3. 1. 3. 4.
1.]
```

With the regression tree, we calculate the prediction accuracy using *Root mean square error (RMSE).*

The value represents the standard deviation of the residuals (prediction error).

```
# Import mean_squared_error from sklearn.metrics to calculate
the root mean squared error (RMSE)
from sklearn.metrics import mean_squared_error

# Calculate the RMSE (Root Mean Squared Error) between the
real and predicted values
rmse = mean_squared_error(y_test, y_pred, squared=False)

# Print the RMSE value to evaluate the model's performance
print(rmse)
```

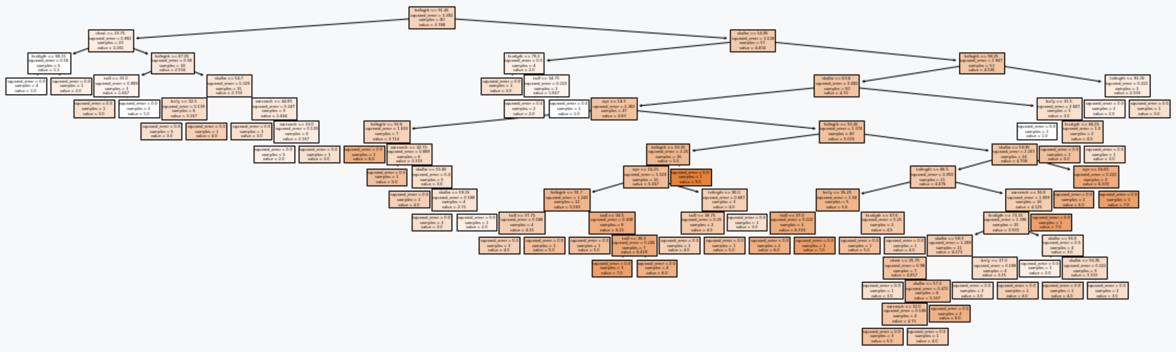
Program output:

```
1.7320508075688772
```

Similar to standard decision trees, we can plot the generated tree.

```
import matplotlib.pyplot as plt
from sklearn import tree
fig = plt.figure(figsize=(20,6))
_ = tree.plot_tree(model,
                   feature_names = ['hdlngth', 'skullw',
'totlngth', 'taill', 'footlgth', 'earconch', 'eye', 'chest',
'belly' ],
                   filled=True)
```

Program output:



6.6.5

Choose the correct statement.

- A regression decision tree predicts a numeral value.
- A regression decision tree predicts a categorical value.

6.6.6

Complete the correct code to create a regression decision tree model with a generation depth of maximum 4.

```
from sklearn.tree import _____

model = _____(random_state=42, _____=4)
_____._____(X_train, y_train)
```

Random Forest

Chapter 7

7.1 Ensemble learning

📖 7.1.1

Decision trees are easy to build, easy to use and easy to interpret. Despite their many advantages, they are not very successful in practice.

In practice, a combination of a large number of decision trees or other classification methods is quite successful. These methods are referred to as ensemble machine learning methods.

Why the combination of several methods is successful can be easily illustrated by the following example.

Assume a game where we roll a die:

- If number 1 or 2 is rolled, our opponent wins,
- if 3, 4, 5, or 6 is rolled, we win.

It is obvious that the chance of our winning is 4:2 or 2:1

Consider the following options:

- Game 1 - let's play 100 times, with a bet per game of 1 EURO
- Game 2 - let's play 10 times with a bet of 10 EUR
- Game 3 - let's play once, the bet is EUR 100.

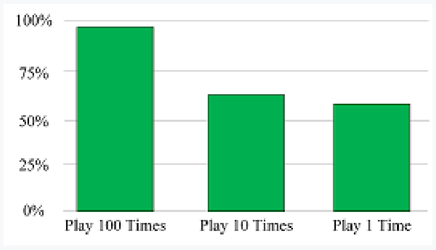
The expected value of the win is the same for all three games

- **Game1** = (P(4/6) * 1)*100 = 0.6666 * 1 * 100 = EUR 66.66
- **Game2** = (P(4/6) * 10)*10 = 0.6666 * 10 * 10 = EUR 66.66
- **Game3** = (P(4/6) * 100)*1 = 0.6666 * 100 * 1 = EUR 66.66

The difference lies in the independent measurements. Although the expected values are the same, the distributions of the results are significantly different!

When are we the most sure that we will definitely earn money?

These are still random rolls of the dice. In the graph we show the observed results (% of our winnings) when simulating the game.



From this simple example, it is clear that a single tree, with a large weight of its classification, can be successful, but also very unsuccessful in classification. We minimize the risk (i.e. we increase the success), if we use multiple trees with small classification weight.

7.1.2

Why is it advantageous to combine multiple decision trees in ensemble learning methods instead of relying on a single tree?

- It increases the accuracy and reduces the risk of making errors in classification.
- It reduces the need for large datasets.
- It makes the model easier to interpret.
- It simplifies the model-building process.

7.1.3

In the dice game example, how does the number of games played affect the certainty of the outcome?

- The more games played, the more consistent the results become.
- The more games played, the less certain we are about the outcome.
- The number of games does not affect the outcome.
- The more games played, the higher the expected value of the win.

7.1.4

Ensemble learning

In machine learning, there is an approach where multiple models are created and combined to improve the accuracy and reliability of predictions. This is known as **ensemble learning**. The idea behind ensemble learning is simple: instead of relying on one model to make predictions, we use a group (or ensemble) of models. When combined, these models can lead to better performance than any single model in isolation. This improvement in accuracy comes from the collective wisdom of the models, especially when they are trained on diverse data or employ different algorithms. By aggregating the predictions of multiple models, ensemble methods aim to enhance the generalization ability of the system.

One of the primary benefits of ensemble learning is that it can reduce the variance of the model's predictions. Variance refers to the extent to which a model's predictions change based on different training data. A high variance typically leads to overfitting, where a model performs well on the training data but poorly on new, unseen data. When predictions from multiple models are averaged, the variance can be significantly reduced, especially if the individual models' predictions are mutually independent. This averaging process helps to smooth out inconsistencies and fluctuations in the predictions of individual models.

Additionally, individual models in an ensemble can be re-learned or adjusted to improve their performance. However, the ensemble itself may be resistant to the need for relearning. This means that even if one or more of the models in the ensemble need adjustment, the ensemble as a whole may still produce reliable predictions. The robustness of ensemble methods makes them highly effective in real-world applications, where the data can be noisy or incomplete.

Ensemble methods are classified into three main approaches: **bagging**, **boosting**, and **stacking**. Each of these methods combines the predictions of multiple models in different ways.

- **Bagging** (Bootstrap Aggregating) involves training multiple models independently on different subsets of the data and then averaging their predictions.
- **Boosting** involves training models sequentially, with each new model focusing on the mistakes made by previous ones.
- **Stacking** combines predictions from different models and then trains a final model to aggregate these predictions into a final output. These approaches vary in how they handle the models and how they combine their predictions, but they all leverage the power of multiple models to enhance performance.

7.1.5

What is one of the primary benefits of using ensemble learning methods?

- They increase the accuracy of the model.
- They make the model faster to train.
- They reduce the complexity of the model.

7.1.6

Which of the following are the basic approaches of ensemble learning?

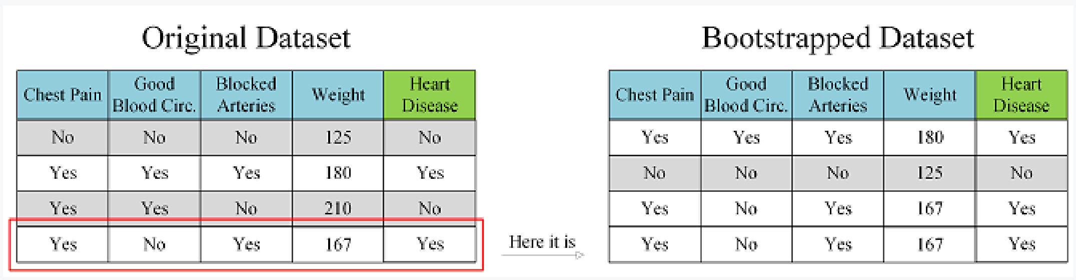
- Bagging
- Boosting
- Clustering

7.1.7

Bagging

In machine learning, **bagging** (short for Bootstrap Aggregating) is a technique used to improve the performance and stability of algorithms, particularly decision trees. Bagging works by creating multiple subsets of the original training set through a random sampling process. Each subset is used to train a separate model, and the final prediction is based on the **majority voting** or **averaging** of the outputs from all individual models. The goal of bagging is to reduce the variance of the model, which helps prevent overfitting and improves the model's generalization to new data.

One of the key methods that utilizes bagging is the **Random Forest** algorithm. Random Forest builds an ensemble of decision trees, where each tree is trained on a different subset of the data created by bagging. In this context, bagging helps to reduce the variance of decision trees, which are often prone to overfitting when trained on small datasets or noisy data. Each decision tree is trained independently on a random sample, and the final output is obtained by averaging the predictions (for regression tasks) or by majority voting (for classification tasks) from all the trees in the forest.



Original Dataset

Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
No	No	No	125	No
Yes	Yes	Yes	180	Yes
Yes	Yes	No	210	No
Yes	No	Yes	167	Yes

Bootstrapped Dataset

Chest Pain	Good Blood Circ.	Blocked Arteries	Weight	Heart Disease
Yes	Yes	Yes	180	Yes
No	No	No	125	No
Yes	No	Yes	167	Yes
Yes	No	Yes	167	Yes

The process of creating random subsets of the original data involves two key techniques: **random sampling of rows** and **random selection of features**. In bagging, some rows in the dataset may be duplicated, and others may be omitted to create different training sets for each model. Additionally, when building each decision tree, not all features (columns) of the dataset are used. Instead, a random subset of features is chosen, making each tree unique. This randomization reduces the influence of individual data points or features, which helps to minimize overfitting and leads to better model performance when generalizing to new data.

In summary, bagging and Random Forest are powerful techniques for improving the accuracy and robustness of machine learning models. By training multiple models on different subsets of the data and combining their outputs, these methods significantly reduce the variance of the predictions, making them highly effective for a variety of tasks, particularly classification problems.

7.1.8

What is the primary purpose of bagging in machine learning?

- To reduce the variance of the model
- To increase the model complexity
- To improve the accuracy of a single model

7.1.9

Which of the following are characteristics of Random Forest?

- It uses random subsets of data and features to train multiple decision trees
- It applies bagging to reduce variance
- It trains a single decision tree using the entire dataset
- It only uses the most important features for training

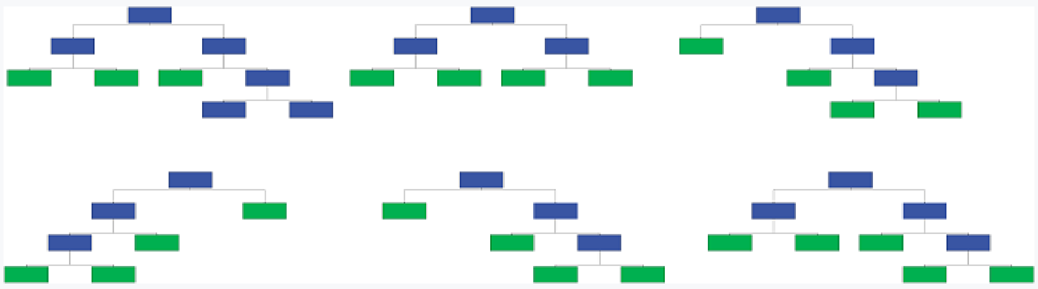
7.2 Random forest

7.2.1

Aggregation

In the Random Forest method, bagging is employed to create multiple decision trees using random subsets of the training data. This technique reduces overfitting and enhances the model's ability to generalize on unseen data. Bagging helps by training each decision tree on a different sample of data, and each tree is built independently. Once all the decision trees are trained, their outputs are combined in a process called aggregation.

Aggregation in Random Forest is the final step where the results from all individual decision trees are combined to make a single prediction. For classification tasks, the output of the Random Forest model is determined by majority voting. This means that each tree "votes" on the class label, and the class with the most votes is selected as the final prediction. For example, if a Random Forest model consists of 100 trees, and 60 trees predict class A and 40 predict class B, then class A will be chosen as the final classification.



This majority voting approach helps reduce the influence of individual trees that may make errors, leading to a more robust and accurate classification. The aggregation process is particularly valuable in ensemble learning because it leverages the strength of many models, each offering a slightly different perspective on the data. By aggregating their predictions, the model becomes more resilient to noise or outliers in the data, ultimately improving its overall accuracy and performance.

In summary, the aggregation step in Random Forest uses majority voting to combine the predictions of multiple decision trees, making the final output more reliable. This process is essential for reducing bias and variance, which are key factors in improving the performance of machine learning models.

7.2.2

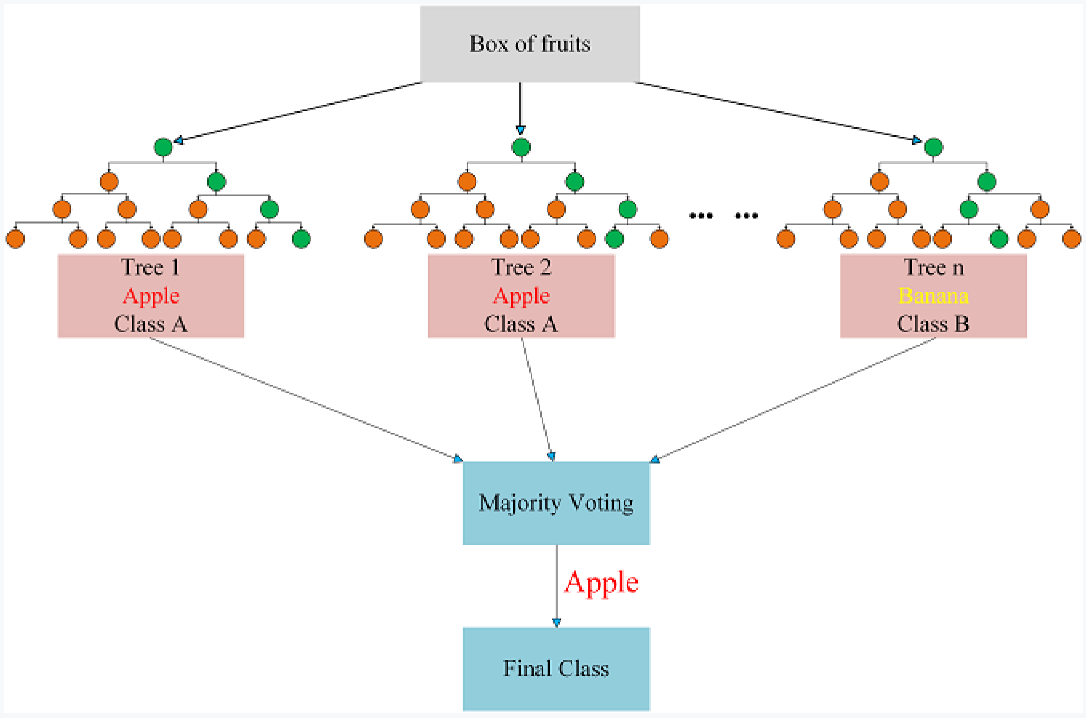
What is the main purpose of the aggregation step in Random Forest?

- To combine the results of all trees using majority voting
- To train individual decision trees
- To create random subsets of the data

7.2.3

The bootstrap sample is taken from the real training dataset data. There is a high probability that each sample will not contain a unique data.

Each model is obtained from a different bootstrap sample and trained independently. Each model generates results. At the end, a majority voting takes place.



7.2.4

Which of the following statements about aggregation in Random Forest is correct?

- The output is based on the class with the highest number of votes
- Majority voting is used to aggregate the outputs of individual trees
- Only the most accurate decision tree is used for the final prediction
- Aggregation helps reduce variance by considering predictions from multiple models

7.2.5

Random Forest algorithm

The Random Forest algorithm is a popular ensemble method used for both classification and regression tasks. It operates by creating a set of decision trees, each built using a random subset of the training data. This diversity in the data used to create each tree is key to the power of Random Forest. The algorithm follows a systematic procedure to build these decision trees and then aggregates their predictions to deliver a final result.

1. In Random Forest, a number of random records, typically referred to as **n samples**, are selected from a dataset that contains **k records**. This process

(bagging) is make the dataset randomly sampled, and the records are chosen independently with replacement.
2. For each of these random samples, individual decision trees are constructed. Each tree is trained using the selected subset of data, which ensures that the models are diverse and capture different aspects of the data.
3. Once the trees are trained, each tree generates an output. In a classification problem, the output will be a predicted class label, while in regression, the output is a predicted numeric value.
4. After all the decision trees have made their predictions, the final output of the Random Forest model is produced by either majority voting (for classification) or averaging (for regression). Majority voting ensures that the class predicted by the most trees is selected as the final result, whereas averaging takes the average of all tree outputs for regression tasks.

7.2.6

Which of the following accurately describes the steps of the Random Forest algorithm?

- Random records are selected from the dataset for each individual decision tree
- The final output is determined by the majority voting or averaging of all decision tree predict
- All decision trees are trained on the entire dataset to ensure consistency.
- Each decision tree is built using the same subset of features and records from the dataset.

7.2.7

Important features of Random Forest

- Diversity - not all features (attributes or variables) are considered when building each individual decision tree. This randomness leads to trees that are different from one another, which increases the diversity of the model.
- Immune to multidimensionality - since each decision tree is built using a subset of features, the algorithm can handle high-dimensional data better than some other models. It reduces the risk of overfitting by ensuring that trees don’t rely on all available features.
- Parallelization - Random Forest allows for parallelization because each tree is created independently. This means that the training process can be distributed across multiple processors or cores, making it faster and more efficient when dealing with large datasets.
- Split train-test - unlike traditional methods where the dataset must be split into training and testing sets, Random Forest inherently handles this

problem. Since each decision tree is trained on a random subset of the data, there will always be data points that any given tree has not seen, which helps assess model performance without needing to manually separate the data.
- Stability - because they rely on the results of multiple trees. By aggregating the predictions of many trees, the algorithm minimizes the risk of overfitting and provides more reliable predictions, especially in the presence of noisy data.

7.2.8

Which of the following are important features of Random Forest?

- Parallelization allows for faster tree creation
- It is immune to multidimensionality due to feature randomness
- Each tree uses all available features to make predictions
- It always requires a separate training and testing set

7.2.9

Random Forest vs. Decision Trees

When comparing Random Forest to Decision Trees, it's important to understand their characteristics, advantages, and drawbacks.

Decision Trees are a fundamental machine learning method. They are simple to understand and interpret, which is one of their biggest advantages. However, decision trees have a tendency to **overfit**, especially when the model is allowed to grow deep without constraints. Overfitting occurs when a model captures too much detail from the training data, including noise, which results in poor performance on unseen data. Despite this, decision trees are **fast to build**, making them useful when computational efficiency is crucial. In addition, decision trees take the entire dataset as input to generate their model, which means that **all examples in the dataset** influence the tree's structure.

Random Forest **addresses the overfitting problem** that decision trees often face. In a Random Forest, multiple decision trees are built from **different subsets** of the data. Each tree is trained on a random sample of the data, and the final output is determined by averaging (for regression) or majority voting (for classification) from all the trees. This technique reduces the risk of overfitting since the trees are less likely to all make the same errors. However, Random Forest models are **slower to build** because of the large number of trees that need to be trained and aggregated. The random selection of observations and features during the training process makes Random Forest a **more robust method**, improving generalization and accuracy.

While **Random Forest is generally more accurate and stable** due to its ensemble nature, it comes at the cost of increased computational time. In contrast, **Decision Trees are faster to train but are more prone to overfitting**. Both techniques have their place depending on the problem at hand, with Random Forest being a more suitable choice when model performance is prioritized over speed.

7.2.10

Which of the following are true about Decision Trees and Random Forests?

- Decision Trees are prone to overfitting, while Random Forests reduce overfitting by averaging or voting.
- Decision Trees take the entire dataset as input, while Random Forests use only subsets of the data to build trees.
- Random Forests are faster to build than Decision Trees.
- Random Forests generate a single decision tree based on the entire dataset.

7.2.11

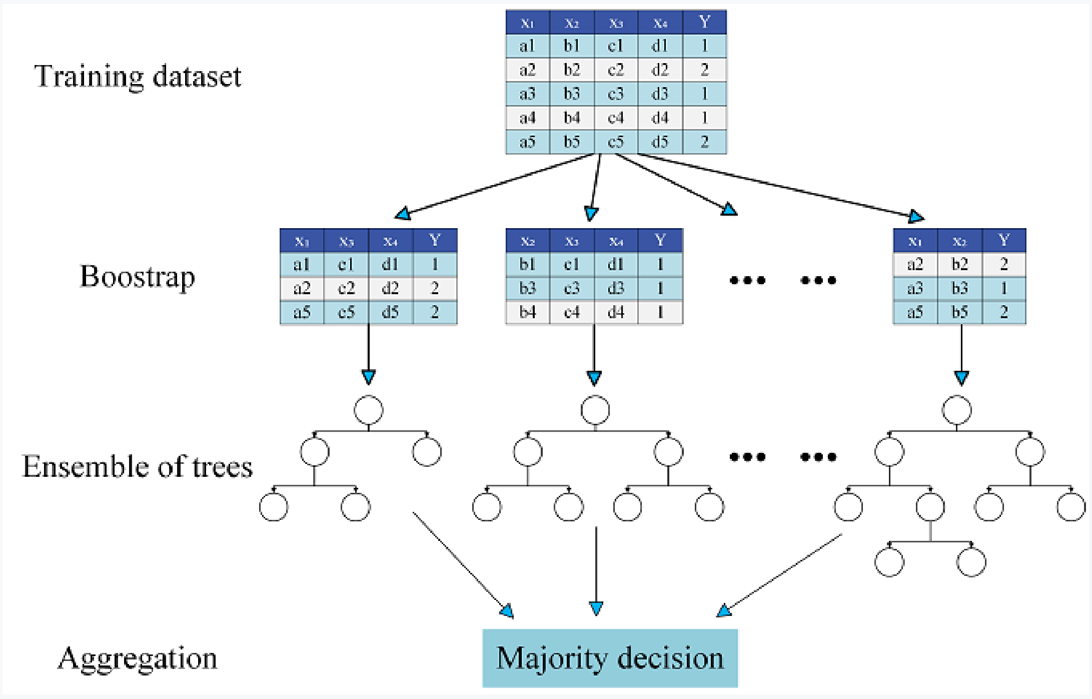
Advantages and disadvantages

Random Forest is a powerful ensemble learning method, widely used for both classification and regression problems. This method leverages the strength of multiple decision trees, which contributes to its numerous advantages. One of the primary benefits of Random Forest is its ability to **solve the issue of overfitting**, which is a common challenge in decision trees. By training multiple trees on random subsets of the data and using majority voting or averaging to make predictions, Random Forest reduces the impact of overfitting and generalizes better to unseen data.

Random Forest can **handle datasets with missing values**. Unlike some models that require complete data, it is still effective when there are null or missing values in the dataset. Its **ability to parallelize** is another advantage, as it allows the trees to be trained simultaneously, making efficient use of computational resources. Furthermore, the use of multiple trees ensures **stability in the model's predictions**, as the final output is based on the average results from many decision trees, thus reducing variance. Random Forest also **maintains diversity** because it does not consider all attributes when building each tree, which enhances the model's robustness and prevents it from becoming too specialized on specific features.

Random Forest is not without its disadvantages. One of the major drawbacks is its **complexity**. Unlike decision trees, where one can follow a clear path to make predictions, the workings of a Random Forest are **less transparent**. The sheer number of trees and the randomization process make it challenging to interpret the model. Additionally, training Random Forest models can be **computationally expensive** and time-consuming, especially as the number of trees grows. The need for each individual tree to generate an output whenever a prediction is made can slow down the process, especially in real-time applications.

While Random Forest is a highly effective tool, it lacks the interpretability that some other models provide, which is often referred to as "not being able to see into the forest."



X_1	X_2	X_3	X_4	Y
a1	b1	c1	d1	1
a2	b2	c2	d2	2
a3	b3	c3	d3	1
a4	b4	c4	d4	1
a5	b5	c5	d5	2

X_1	X_3	X_4	Y
a1	c1	d1	1
a2	c2	d2	2
a5	c5	d5	2

X_2	X_3	X_4	Y
b1	c1	d1	1
b3	c3	d3	1
b4	c4	d4	1

X_1	X_2	Y
a2	b2	2
a3	b3	1
a5	b5	2

7.2.12

What are the benefits of Random forest?

- It solves the problem of overfitting
- It works well even if the data contains null/missing values
- It exhibits the features of parallelization
- It combines GINI index and Entropy within a single tree
- It also works at higher degrees of polynomial

7.2.13

Which of the following are advantages of Random Forest?

- Random Forest works well with missing data in the dataset.
- Random Forest can be used for both classification and regression.

- Random Forest is computationally inexpensive to train.
- Random Forest is transparent and easy to interpret.

7.3 Other ensemble learning methods

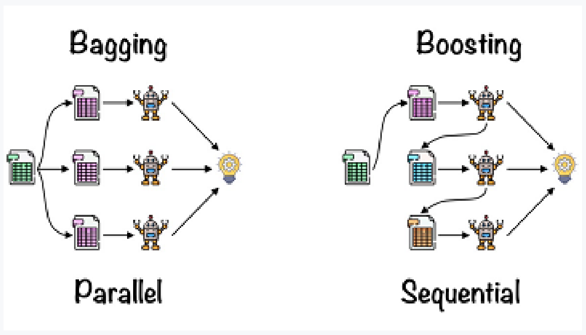
📖 7.3.1

Boosting

There were three basic techniques in ensemble machine learning methods: bagging, boosting, stacking.

Boosting is another popular ensemble learning technique, which is different from bagging in several key ways. While bagging involves training multiple models in parallel, boosting operates sequentially. This means that in boosting, each model is built one after another, with each subsequent model trying to correct the errors made by the previous ones. Boosting is particularly powerful because it transforms "weak learners" - models that perform slightly better than random guessing - into a strong predictive model by focusing on correcting previous mistakes, which improves accuracy.

A key feature of boosting is the way it assigns weights to the input patterns. Initially, each pattern has the same weight, but when a pattern is misclassified, its weight is increased. This ensures that the subsequent models pay more attention to the misclassified patterns, trying to correct the mistakes made earlier in the process. The final prediction in boosting is made through weighted voting, where each model's contribution to the overall decision depends on its performance.



Unlike bagging, where multiple models are created independently, boosting builds models sequentially, making it more sensitive to the data and its distribution. Boosting is known for achieving higher accuracy than bagging in many cases, especially when the models are weak learners that have room for improvement. However, the sequential nature of boosting means that it can be computationally more expensive and slower to train compared to bagging.

Boosting also offers an advantage in hyperparameter configuration. Since boosting focuses on correcting errors from previous iterations, it can be more fine-tuned and better configured, making it highly adaptable to different datasets. This ability to adjust and improve continuously makes boosting a powerful technique for improving model performance.

7.3.2

What is bagging?

- a technique that creates different subsets of the training set
- a technique that combines "weak learners" into a strong sequential model for the highest possible accuracy
- a technique that compares all the metrics of success of a single decision tree
- a metric of the equilibrium representation of classes in the dataset

7.3.3

Which of the following statements about boosting are true?

- Boosting gives more weight to misclassified input patterns.
- Boosting uses simple models (weak learners) and builds a strong sequential model.
- Boosting runs in parallel, like bagging.
- In boosting, each model is trained independently from the others.

7.3.4

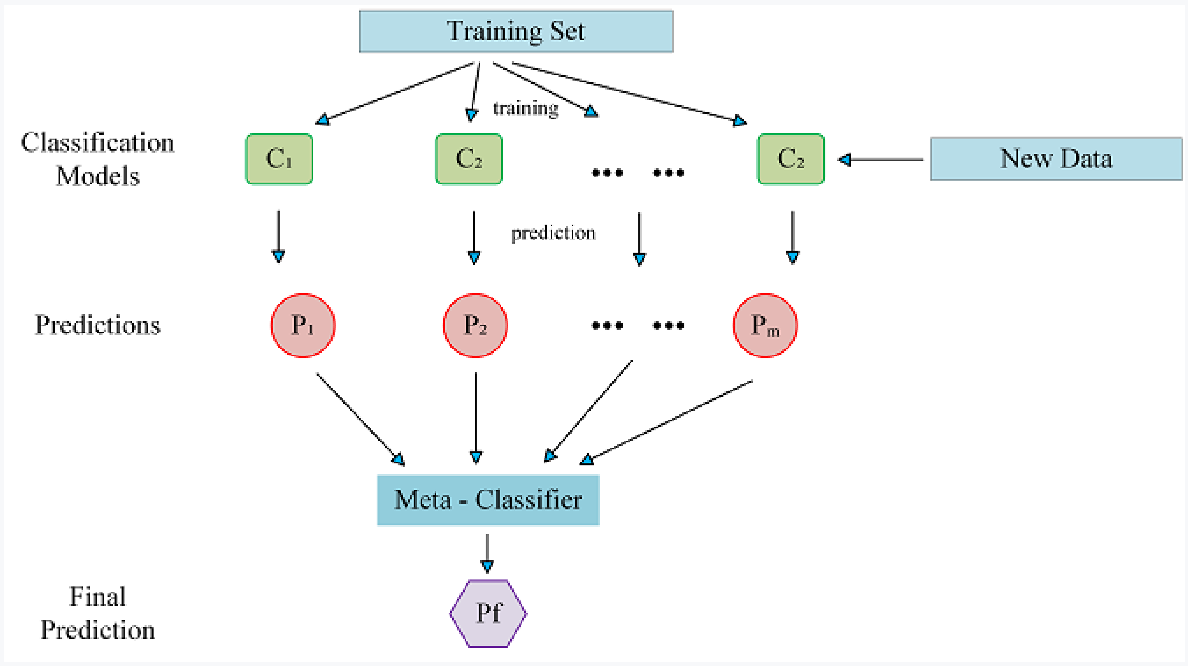
Voting

The Voting technique is a powerful ensemble method where different classifiers, or models, are combined to make predictions. Unlike bagging and boosting, which rely on either majority voting or weighted voting based on individual model outputs, voting involves a different approach: it aggregates the predictions from several base classifiers to make the final prediction. These base classifiers can be any type of machine learning model, and they contribute to the final decision in a way that goes beyond just simple averaging or voting.

In Voting, the individual classifiers cast votes for the predicted class, and the final prediction is made based on how many votes a class receives. However, the key feature of this method is that the reliability of each classifier's vote is considered at the next level of the meta-classifier. This means that some classifiers, which are more reliable, might have a stronger influence on the final decision compared to less reliable classifiers. The meta-classifier essentially learns from the results of these base classifiers, refining the final prediction based on the reliability of each classifier.

This technique is particularly useful when we want to combine the strengths of multiple learning algorithms. For example, if we have a decision tree, a support vector machine (SVM), and a logistic regression model, each of these models might perform differently on different aspects of the data. By combining them using the Voting method, we can leverage their complementary strengths, leading to a more robust and accurate final prediction. Voting can be seen as a simple yet effective way to increase the performance of a model by bringing together different approaches to make the final decision.

Voting can be applied in both classification and regression tasks. While in classification, we use the majority vote (for classification tasks) or average (for regression tasks), in more complex setups, we can use a weighted voting system where different classifiers contribute differently based on their accuracy. This makes the Voting technique flexible and adaptable to different types of problems.



7.3.5

Which of the following is a characteristic of the Voting technique?

- It combines different models to make a final prediction.
- It only works for classification tasks.
- It always applies majority voting, regardless of classifier reliability.
- It uses only one model to make predictions.

7.3.6

Which of the following statements about the Voting technique is true?

- Voting considers the reliability of each classifier in determining the final output.
- In Voting, base classifiers are combined to improve the robustness of the prediction.
- In Voting, the final prediction is based on the average output of all classifiers.
- The Voting technique only uses decision trees to make predictions.

7.4 Practical tasks

7.4.1

If we use the **scikit-learn** library to create a Random Forest, we can set the following parameters:

- **n_estimators** - the number of trees that the algorithm creates before averaging the predictions.
- **max_features**- the maximum number of elements that the Random Forest considers to be a node distribution.
- **mini_sample_leaf**- specifies the minimum number of leaves needed to distribute the inner node.
- **n_jobs** - information about the number of processors that can be used. If the value is 1, only one processor can be used, but if the value is -1, there is no limit.
- **random_state** - checks the randomness of the sample. The model will always produce the same results if it has a certain value of random state
- **oob_score** - OOB stand for Out Of the Bag. This is a random forest cross-validation method. A portion of the sample is not used to train the data, but it is used to evaluate its performance

7.4.2

Project: Random Forest for Titanic dataset

Building a random forest is a very similar process to building a decision tree. A random forest is actually a model that consists of several individual decision trees.

The final predicted value of the random forest is most often the most frequent resulting value from the individual decision trees.

The following example shows a solution to the Titanic survival prediction using a random forest.

- Dataset: https://priscilla.fitped.eu/data/pandas/titanic.csv

We load the dataset and prepare a suitable training and test set as in the previous sections.

```
# Importing libraries
import pandas as pd
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')

# Reading the Titanic dataset from a URL
data =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/titanic.c
sv')

# Selecting relevant columns for analysis
data = data[['Survived', 'Pclass', 'Sex', 'Age', 'SibSp',
'Parch']]

# Removing any rows with missing values
data = data.dropna()

# Convert the 'Sex' column to numerical format: male -> 0,
female -> 1
pd.set_option('future.no_silent_downcasting', True)
data['Sex'] = data['Sex'].replace({'male': 0, 'female': 1})

# Creating feature matrix X by excluding 'Survived' column
X = data[data.columns.difference(['Survived'])]

# Defining target variable y as 'Survived' column
y = data['Survived']

# Splitting data into training (70%) and test (30%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

We will build a Random Forest model using the sklearn library:

- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

```
# Importing the RandomForestClassifier from sklearn.ensemble
for building a Random Forest model
from sklearn.ensemble import RandomForestClassifier

# Initializing the Random Forest model with a fixed random
seed for reproducibility
rf_model = RandomForestClassifier(random_state=42)
```

We will train the created model and make predictions using the test set.

```
# Fitting the Random Forest model to the training data
rf_model.fit(X_train, y_train)

# Predicting the target variable (Survived) using the trained
Random Forest model on the test data
y_pred = rf_model.predict(X_test)

# Printing the predicted values for the test data
print(y_pred)
```

Program output:

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

We compare the values with the real y_test values.

```
print(y_test.to_numpy())
```

Program output:

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

We compute the prediction accuracy in the same way as for decision trees.

```
from sklearn import metrics
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

Program output:

```
Accuracy: 0.786046511627907
```

We compare the results with the default decision tree model.

```
# Importing the DecisionTreeClassifier from sklearn.tree
from sklearn.tree import DecisionTreeClassifier

# Creating an instance of the DecisionTreeClassifier with a
fixed random state for reproducibility
clf = DecisionTreeClassifier(random_state=42)

# Training the DecisionTreeClassifier on the training data
(X_train, y_train)
clf.fit(X_train, y_train)

# Predicting the target variable (Survived) on the test data
(X_test)
y_pred_DT = clf.predict(X_test)

# Printing the accuracy of the DecisionTreeClassifier model on
the test data
print("Accuracy:", metrics.accuracy_score(y_test, y_pred_DT))
```

Program output:

```
Accuracy: 0.7767441860465116
```

The accuracy of the random forest model compared to the decision tree is 0.93% higher.

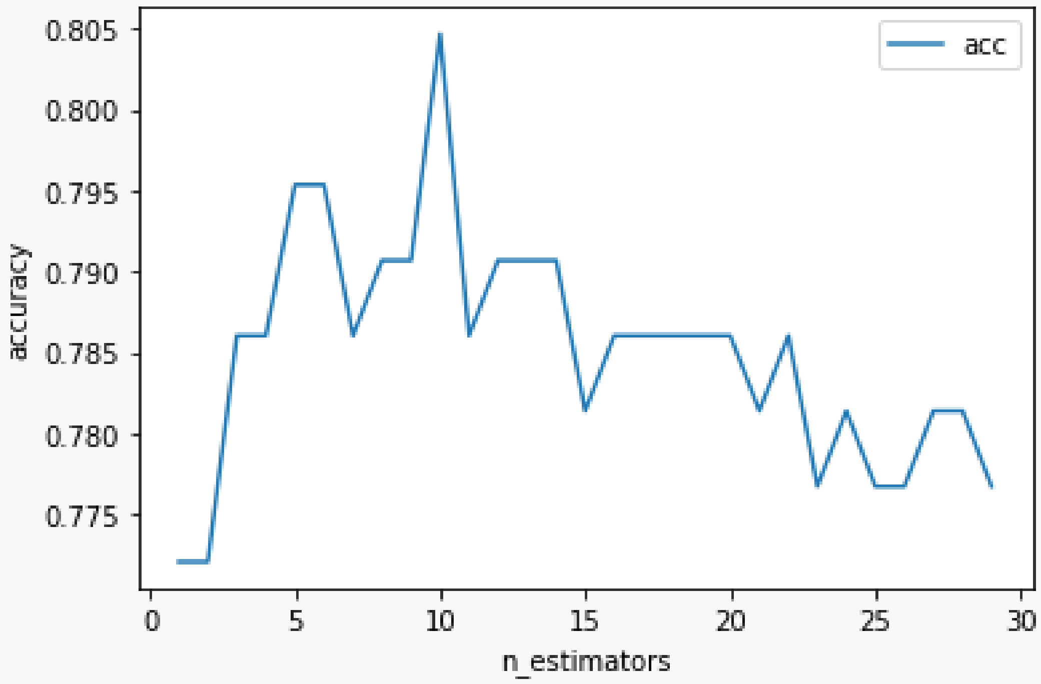
- The random forest, like the decision tree, can be fitted with parameters.

The following example shows how the prediction accuracy of a random forest varies based on the number of trees built of.

- We create the forest sequentially from 1 to 30 trees and wefind the prediction accuracy.

```
# Importing matplotlib for plotting
import matplotlib.pyplot as plt

# Initializing empty lists to store the number of estimators
and accuracy values
n_estimators = []
acc = []

# Looping through a range of values for n_estimators from 1 to
30
for i in range(1, 30):
  # Creating a RandomForestClassifier with the current number
of estimators
  rf = RandomForestClassifier(n_estimators=i, random_state=42)

  # Fitting the model on the training data (X_train, y_train)
  rf.fit(X_train, y_train)

  # Predicting the target variable (Survived) on the test data
(X_test)
  pred = rf.predict(X_test)

  # Appending the accuracy score of the predictions to the acc
list
  acc.append(metrics.accuracy_score(y_test, pred))
  # Appending the current number of estimators to the
n_estimators list
  n_estimators.append(i)

# Creating a DataFrame to store the accuracy and number of
estimators for visualization
d = pd.DataFrame({'acc': pd.Series(acc), 'n_estimators':
pd.Series(n_estimators)})

# Plotting the accuracy against the number of estimators
plt.plot('n_estimators', 'acc', data=d, label='acc')

# Adding labels and a legend to the plot
plt.xlabel('n_estimators')
plt.ylabel('accuracy')
plt.legend()
```

Program output:



The forest with 10 trees has the highest accuracy.

```
# Creating a RandomForestClassifier model with a fixed number
of estimators (10) and a random seed for reproducibility
rf_model = RandomForestClassifier(random_state=42,
n_estimators=10)
```

```
# Fitting the model to the training data (X_train, y_train)
rf_model.fit(X_train, y_train)

# Predicting the target variable (Survived) for the test data
(X_test)
pred = rf_model.predict(X_test)

# Printing the accuracy of the predictions by comparing with
the actual labels (y_test)
print(metrics.accuracy_score(y_test, pred))
```

Program output:

```
0.8046511627906977
```

A forest with 10 trees has an accuracy of 80.47%, which is 1.87% higher than the accuracy of the random forest model with default settings.

7.4.3

A random forest is a model, which consists of several individual decision trees.

- True
- False

7.4.4 RandomForest with six trees

Complete the code to build a random forest with six trees. Use the parameter random_state=42 to preserve the randomization.

Set the size of the test set to 20%. As a result, write the model accuracy.

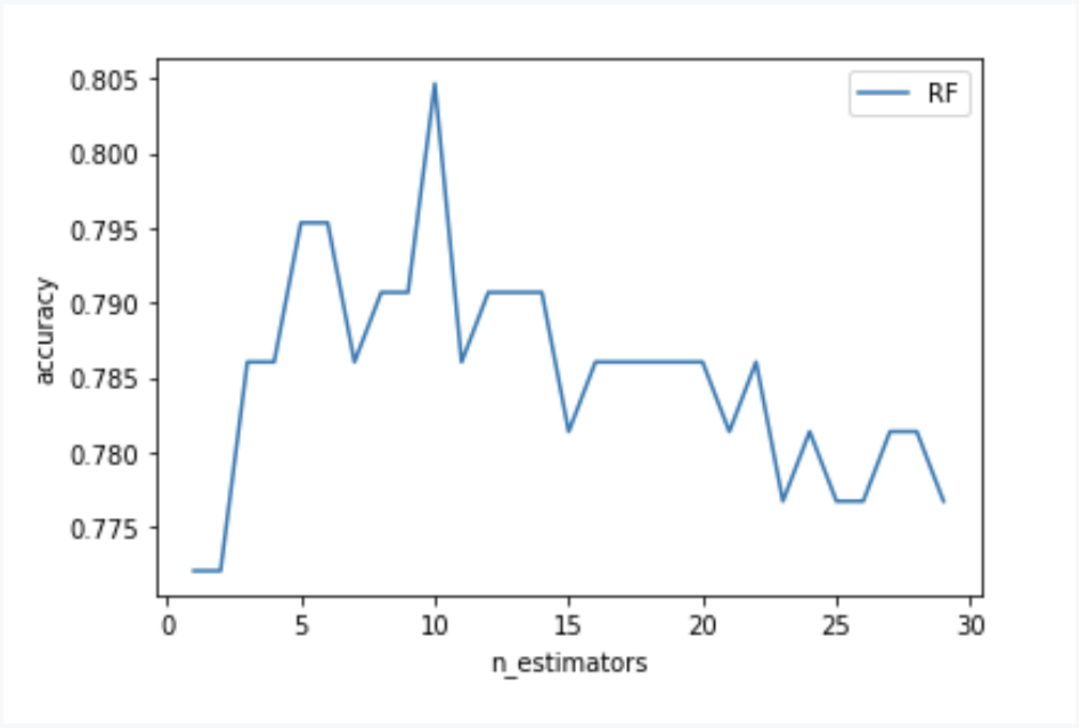
file1.py

```
import pandas as pd
from sklearn.model_selection import train_test_split

data =
pd.read_csv('https://priscilla.fitped.eu/data/pandas/titanic.c
sv')

data = data[['Survived', 'Pclass', 'Sex', 'Age', 'SibSp',
'Parch']]
```

```
data = data.dropna()

data['Sex'] = data['Sex'].replace({'male': 0, 'female': 1})

X = data[data.columns.difference(['Survived'])]
y = data['Survived']

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

7.4.5

How many decision trees would you use in a random forest model based on the following graph?



Regression

Chapter 8

8.1 Regression

📖 8.1.1

Regression is a core technique in supervised learning, particularly in fields like AI and data science. It is used to predict continuous numerical values based on input data. While classification models predict discrete categories, regression models focus on understanding and predicting relationships between variables, making them invaluable for tasks such as forecasting and trend analysis.

In a regression model, we aim to establish a mathematical function that describes the relationship between one or more independent variables (also called predictors or features) and a dependent variable (also called the target or outcome). For instance, a company might use regression to predict sales based on variables like advertising spend, product price, and market conditions. By doing so, the company can understand how changes in these factors influence sales and make more informed decisions.

Regression analysis is widely applied when there is a need to understand the relationship between variables in a dataset. These relationships could be linear (where the dependent variable changes at a constant rate with respect to an independent variable) or non-linear. A key application of regression models is forecasting, where they help predict future values based on historical data. For example, predicting stock market trends or forecasting weather patterns often relies on regression techniques. It can also help in establishing causal relationships, where researchers determine how changes in one variable (e.g., temperature) affect another (e.g., crop yields).

The purpose of regression analysis is primarily twofold: predicting the value of the dependent variable given the independent variables, and understanding the effect of an independent variable on the dependent variable. For example, a business could use regression to predict how changes in advertising spending will influence sales, or to forecast future sales based on historical trends.

📝 8.1.2

What is the primary difference between regression and classification in machine learning?

- Regression predicts continuous values, while classification predicts discrete categories.
- Regression predicts discrete categories, while classification predicts continuous values.
- Both regression and classification predict continuous values.
- Both regression and classification predict discrete categories.

8.1.3

In AI systems, regression is not just a standalone predictive tool but also a foundational component of larger workflows. It aids in deriving insights, optimizing processes, and supporting decision-making. Typical applications are:

- Predicting prices - regression models are used in real estate platforms to predict house prices based on factors like size, location, and age of properties, enabling accurate valuations.
- Estimating probabilities and scores - in e-commerce, regression helps estimate customer lifetime value, enabling businesses to identify high-value customers and allocate resources effectively.
- Forecasting trends - regression supports forecasting tasks, such as predicting sales trends, stock prices, or energy demand, based on historical data and influencing factors.

By capturing the relationships between input variables and continuous outcomes, regression serves as a cornerstone of predictive analytics and enhances the functionality of AI-driven systems. It also provides a stepping stone to more advanced AI techniques, such as neural networks, where regression-like structures are integral to model architectures.

8.1.4

Which of the following are typical applications of regression analysis?

- Predicting future sales based on past sales data.
- Forecasting stock market trends.
- Determining the price of a house based on its features like size and location.
- Categorizing animals based on species.

8.1.5

Classification vs. regression

In machine learning, classification and regression are two fundamental tasks, but they differ significantly in terms of goals and types of predictions they make. The key difference lies in the nature of the predicted output. **Classification** is used when we want to categorize data into predefined classes or labels, such as predicting whether an email is spam or not spam (binary classification), or predicting the digit in an image (such as 0, 1, 2,...,9). These outputs are discrete and non-numerical.

On the other hand, **regression** is used when we need to predict a continuous value. Instead of assigning a class label, regression models output numerical values, which can be integers or decimals. For example, predicting house prices based on various factors like size and location, or predicting the temperature for the next week, are tasks suited for regression models. The purpose of regression is to understand the relationship between the input variables (predictors) and a continuous target variable.

The assumptions we make in classification and regression also differ. In classification, we assume that there is a set of discrete classes, and we aim to find the boundary or decision rule that separates these classes based on the input features. In contrast, regression assumes a continuous relationship between the dependent and independent variables, which allows us to make predictions with a smooth, continuous output.

Thus, the main difference between classification and regression is the nature of the predicted variable: classification predicts labels, whereas **regression predicts continuous values**.

8.1.6

What is the key difference between classification and regression?

- Classification predicts discrete values, while regression predicts continuous values.
- Classification predicts continuous values, while regression predicts discrete values.
- Classification and regression both predict continuous values.
- Classification and regression both predict discrete values.

8.2 Linear regression I.

8.2.1

Linear regression

In linear regression, the goal is to predict a continuous target variable by understanding the relationship between the input variables (features) and the output variable (target). The principle behind linear regression is to sum the effects of each independent variable (feature) on the dependent variable (target), and the result is the predicted value.

This process of summing the effects of variables is called a **linear combination**. The term "linear" refers to the fact that if the value of one feature changes by a fixed amount, the output value will increase (or decrease) by a proportional amount. For example, if you are predicting the price of a house based on its size, location, and age, the regression model might learn that for each additional square meter, the price increases by a fixed amount, and for each year of age, the price decreases by another fixed amount. This relationship can be represented mathematically as a linear equation.

A simple example of linear regression involves predicting the price of items in a shopping basket. Imagine you know the total cost of a basket with various items, such as potatoes, carrots, and a bottle of wine. However, determining the price of each item individually can be tricky with just one basket, as there are many possible combinations of prices that could result in the same total cost. But if you have multiple baskets with different combinations, the model can use the available data to find the best estimate for the price of each item.

In linear regression, the model learns the relationship between the features (such as the quantity of each item) and the total price. It does this by minimizing the error in its predictions, ensuring that the sum of the effects of each variable gives the closest possible prediction to the actual target values.

8.2.2

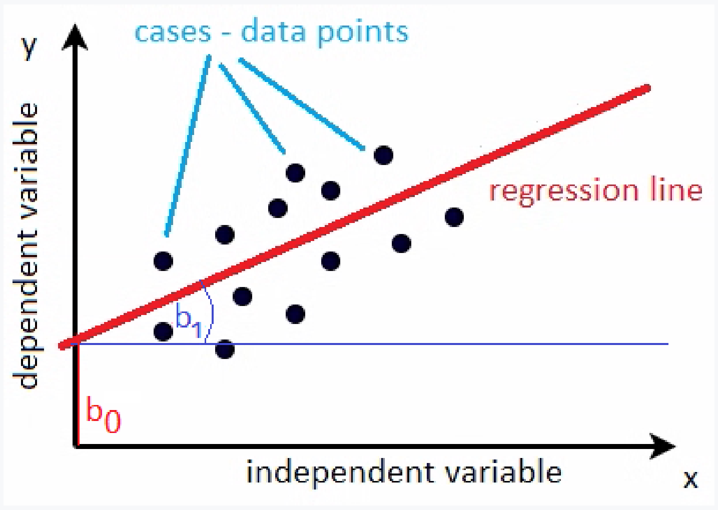
Which of the following statements about linear regression are true?

- Linear regression uses a linear combination of variables to make predictions.
- In linear regression, a change in the input variable results in a proportional change in the output.
- Linear regression predicts a discrete class label.
- The output of linear regression is always an integer.

8.2.3

Linear regression is widely used in machine learning to solve regression problems, where the goal is to predict a continuous output variable based on one or more input variables. It is one of the simplest and most fundamental algorithms used in predictive modeling, and it aims to model the relationship between input variables and a continuous target variable.

In linear regression, we aim to find the best-fitting linear equation that represents the relationship between the input variables (predictors or features) and the output variable (response variable). The key idea behind this method is that the relationship between the input and output variables is assumed to be linear, meaning that the target variable changes in proportion to changes in the input variables.



A simple **linear regression model** has the following equation:

$y = b_0 + b_1 * x$

Where:

- **y** is the **dependent variable** or the target variable (the value we are trying to predict),
- **x** is the **independent variable** or the feature (the input data),
- **b_0** is the **intercept term**, representing the value of y when x is zero,
- **b_1** is the **slope coefficient**, representing the change in y for a one-unit change in x.

The goal of linear regression is to find the best values for b_0 and b_1 such that the line we draw through the data points best fits the actual data. This is achieved by minimizing the errors, which are the differences between the predicted values (y) and the actual values. The line that minimizes these errors is considered the best fit.

8.2.4

What is the primary goal of linear regression?

- To predict a continuous value based on input variables.
- To predict a categorical value.
- To find the relationship between input and output variables in a non-linear fashion.
- To classify data into specific categories.

📖 8.2.5

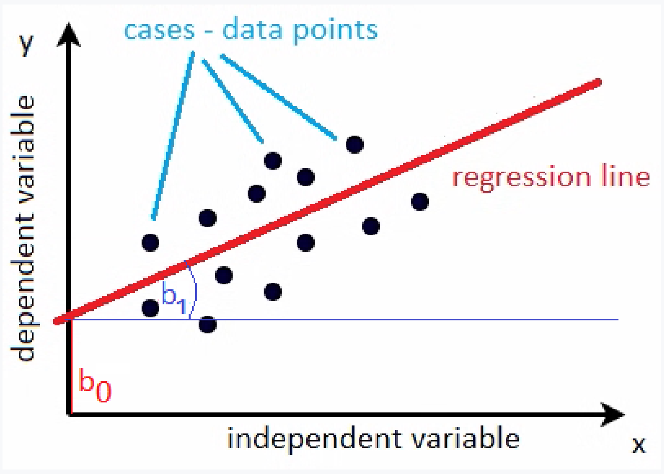
Simple linear regression

In simple linear regression, the relationship between the dependent variable (y) and the independent variable (x) is modeled using a straight line. The equation for this model is:

$$y = b_0 + b_1 * x$$

This equation represents a straight line where:

- **y** is the target variable we want to predict,
- **x** is the feature or input variable,
- **b_0** is the intercept term, which represents the predicted value of y when x is zero,
- **b_1** is the slope coefficient, which measures how much y changes when x increases by one unit.



The purpose of simple linear regression is to estimate the values of **b_0** and **b_1** in such a way that the **predicted values of y** (using this equation) are as close as possible to the **actual observed values of y** in the dataset. This is done by minimizing the **sum of squared errors** between the predicted and actual values. The model searches for the best-fit line that minimizes the discrepancy between the observed data points and the predicted values from the line.

The **slope coefficient (b_1)** is crucial in determining the strength of the relationship between the input variable and the target variable. If b_1 is positive, the relationship is positive, meaning that as x increases, y also increases. Conversely, if b_1 is negative, the relationship is negative, meaning that as x increases, y decreases.

8.2.6

Which of the following statements about simple linear regression are true?

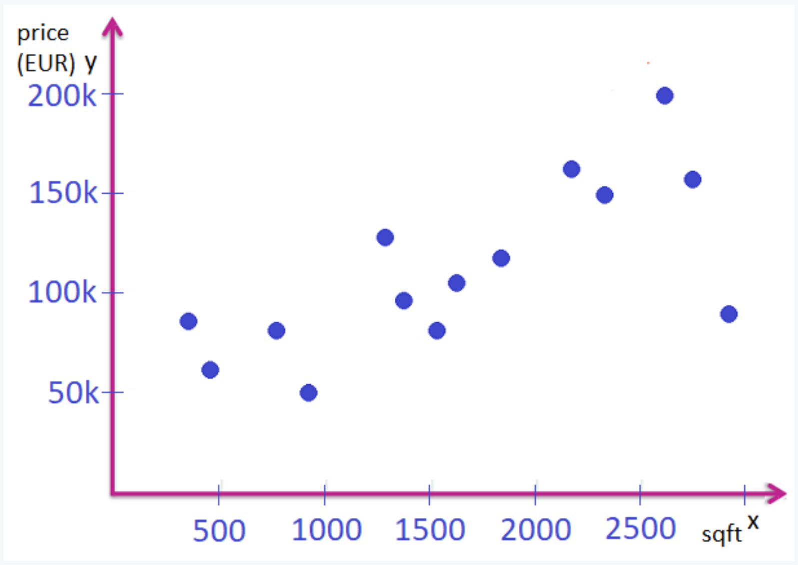
- The goal is to predict a continuous output variable
- The slope coefficient (b_1) determines the change in y for a unit change in x.
- The equation of has only one input variable.
- The intercept term (b_0) represents the change in y for a unit change in x.

8.2.7

Predict house prices

In real estate, one common application of linear regression is predicting the sale price of a house based on its features, such as its square footage. Suppose you want to estimate how much you could sell your house for, and you have a graph of recent home sales over the last two years, with the sale price plotted against the square footage (sqft) of the homes.

In this example, the square footage of a house is the independent variable (input feature), and the sale price is the dependent variable (output). Since no home recently sold has the exact same square footage as your home, we can't predict an exact price based on your home's square footage. However, we can use linear regression to find a general trend in the data, which will allow us to estimate the price for a house with a specific square footage, even if it doesn't have the exact same size as those in the data.



Linear regression will help find the best-fitting line that captures the relationship between square footage and price. This relationship can be represented by the equation:

Price = b_0 + b_1 * sqft

Where:

- **Price** is the dependent variable (the price we want to predict),
- **sqft** is the independent variable (the square footage of the house),
- **b_0** is the intercept, representing the estimated price when the square footage is zero,
- **b_1** is the **slope coefficient, re**presenting how much the price increases for every additional square foot.

When the model is trained on the historical sales data, it will determine the values for **b_0** and **b_1** that minimize the prediction error. Once these values are found, you can input your house's square footage into the equation to predict the sale price.

For example, if the regression model finds that for every additional 100 square feet, the price of the house increases by 15,000 EUR, and your house has 3,500 square feet, the model will estimate the price based on that information.

8.2.8

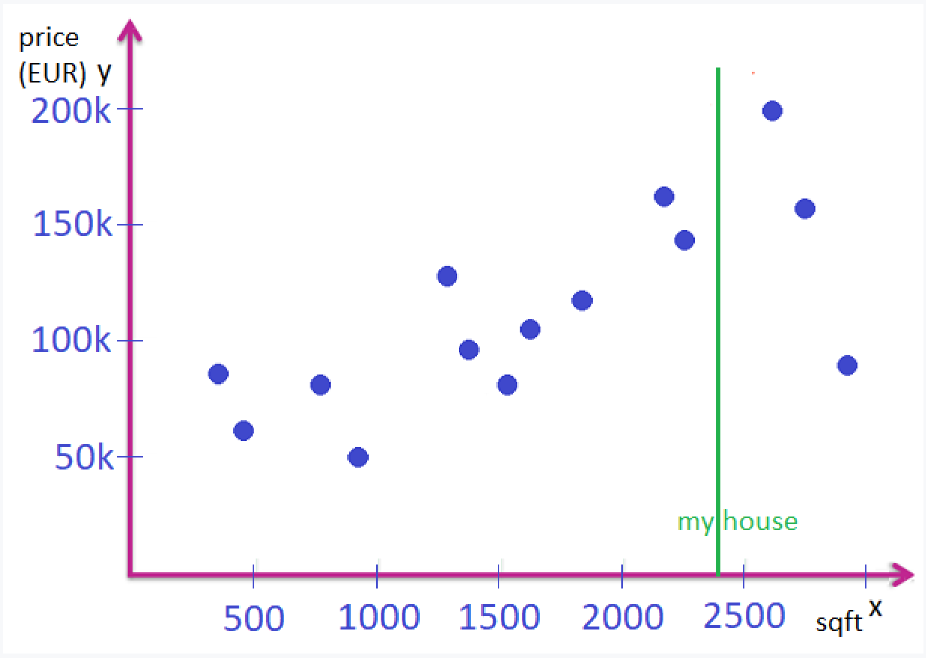
What does the slope coefficient (b_1) represent in a linear regression model for predicting house prices?

- The increase in the sale price for every additional square foot of the house.
- The estimated sale price of the house when square footage is zero.
- The total price of the house.
- The decrease in price for every additional square foot of the house.

8.2.9

Home price estimate

Imagine you own a home with a square footage of 2,350 sqft and want to estimate its sale price. To do this, you gather data on the sale prices of other homes in your neighbourhood, each with different square footage. The question is: how can you determine the correct price for your home?



One simple approach is to find homes with slightly smaller and slightly larger square footage than your home. For example, if homes with 2,300 sqft and 2,550 sqft sold for 148,000 and 192,000 respectively, you could average their prices:

```
Estimated Price = (148,000 + 192,000) / 2 = 170,000
```

Alternatively, you might average prices from more neighbours add homes with 2,100 sqft, 2,700 sqft, and 2,400 sqft - to get a broader estimate (result is different).

Instead of relying on a few neighbours, you could use all the available data points and fit a linear regression line. This line summarizes the relationship between square footage and sale price, allowing you to predict the price of your home based on its exact size.

8.2.10

What is the average price if we have houses with following prices as neighbours:

145,000; 155,000; 160,000; 200,000?

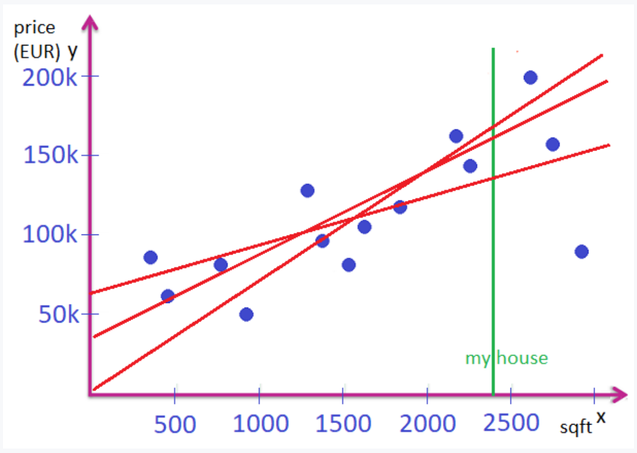
8.3 Linear regression II.

🕮 8.3.1

How to find regression line

In real-world scenarios, we often encounter datasets that involve multiple variables. We have a dataset of houses where we know their sizes (in square feet) and their corresponding selling prices. While it is clear that larger houses generally sell for higher prices, the exact relationship between size and price is not immediately apparent. This is where regression comes in.

To predict the price of a house based on its size you might start by plotting the available data on a graph, with house sizes on the x-axis and prices on the y-axis. The points scatter across the graph, showing variability in house prices even for homes with similar sizes. This variability is expected because house prices are influenced by other factors like location, condition, and market demand. However, the question remains: can we capture a general trend between size and price?



To simplify the relationship and make predictions, we need to draw a line that represents the average relationship between size and price. This line is called a **regression line**. Its purpose is to summarize the data and provide a mathematical formula to predict prices for any given house size, even if the exact size does not exist in the dataset.

But here’s the challenge: **there isn’t just one possible line.** Depending on how we position the line on the graph, we can create many different lines, each suggesting a slightly different relationship between size and price. Some lines might tilt steeply, indicating a sharp increase in price as size increases, while others may be flatter, suggesting a weaker relationship. The question then becomes: **which line best represents the data?**

This step is critical because the choice of the regression line affects the accuracy of our predictions. A poorly chosen line might overestimate prices for some homes and underestimate them for others, leading to unreliable conclusions. Thus, finding the "best" regression line is not just about drawing a line - it’s about ensuring it reflects the underlying relationship between variables **as accurately as possible**.

8.3.2

Why is it important to find the best regression line for a dataset?

- To minimize the differences between actual and predicted values.
- To establish a clear relationship between variables.
- To eliminate errors completely.
- To guarantee predictions are always correct.

8.3.3

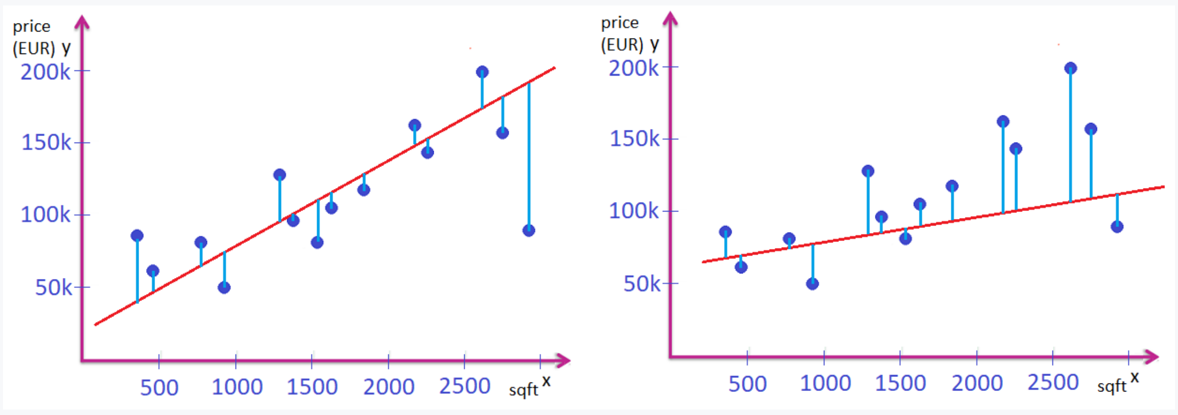
As accurately as possible

Once we recognize that many lines can represent the relationship between house size and price, the next step is to determine which one is the most accurate. But how do we decide what makes a line “accurate”? The answer lies in minimizing the errors between the actual prices and the prices predicted by the line.

For each house in the dataset, the regression line provides a predicted price based on its size. The error, in this context, is the difference between the actual price of the house (as recorded in the dataset) and the predicted price (from the line). Mathematically, this error is called a **residual** and is calculated as:

Residual = Actual Price – Predicted Price

If the regression line is positioned well, the residuals will be small, meaning the line closely matches the data. However, if the line is poorly placed, the residuals will be large, indicating that the predictions deviate significantly from the actual values.



To evaluate how well a line fits the data, we aggregate the residuals for all houses. A simple sum of residuals won't work because positive and negative residuals could cancel each other out. Instead, we square each residual (to eliminate negatives) and calculate the **Residual sum of squares (RSS):**

$$RSS = \sum (ActualPrice - PredictedPrice)^2$$

The RSS measures the total error in the predictions. A smaller RSS indicates a better-fitting line because it means the line minimizes the differences between actual and predicted prices across all houses in the dataset. Therefore, the problem of finding the "best" regression line becomes a mathematical optimization problem: **find the line that minimizes the RSS.**

But solving this isn't just a matter of trial and error. With many potential lines, it would be impractical to test them all manually. Instead, we use a systematic mathematical approach, often leveraging tools from calculus and linear algebra, to identify the optimal values for the line's slope and intercept. This process ensures that the regression line is as accurate as possible, making it a powerful tool for prediction and analysis.

By focusing on minimizing the RSS, we not only determine the best line but also ensure that our predictions for house prices are grounded in the most reliable trend derived from the data.

8.3.4

What does the Residual sum of squares measure in a regression analysis?

- The total error between actual and predicted values.
- The total sum of all predicted prices.
- The average size of all houses.
- The difference between the largest and smallest prices.

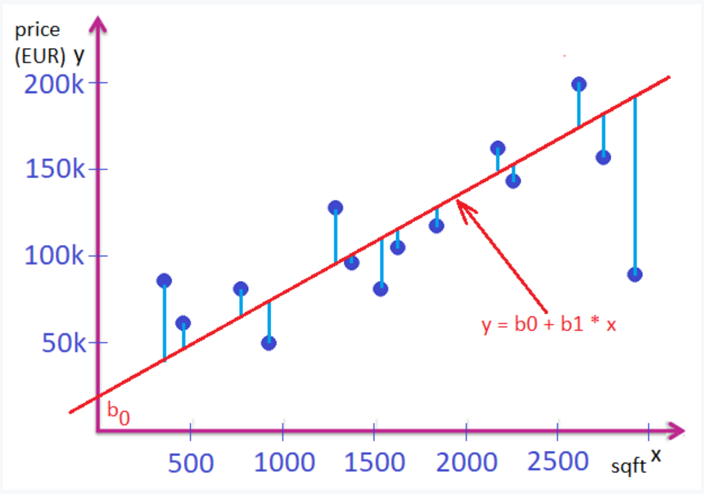
8.3.5

Minimizing the residual sum of squares

Finding the best-fitting line involves minimizing the RSS. The RSS measures the total error between the actual values and the predicted values made by the regression line. Let's explore how to minimize this error systematically.

The Residual sum of squares is given by:

$$RSS = \sum (ActualValue - PredictedValue)^2$$



Each residual represents the difference between the actual value of the dependent variable and the value predicted by the line. Squaring these differences ensures that **all errors are positive and penalizes larger errors more heavily**.

The goal is to minimize the RSS, which means finding the values of the **intercept (b_0)** and the **slope (b_1)** that produce the smallest RSS.

The **predicted value (y)** in linear regression is calculated as:

$y = b_0 + b_1 * x$

The actual value is stored in the dataset and so we calculate RSS as the sum of the squares of the differences between the actual price and the predicted price for all houses in the dataset

$RSS = [price_{house1} - (b_0 + b_1 * square_{house1})]^2 + [price_{house2} - (b_0 + b_1 * square_{house2})]^2 + ... + [price_{housen} - (b_0 + b_1 * square_{housen})]^2$

Optimization

To find the optimal b_0 and b_1, we use a method called Least squares estimation. This involves calculus to compute the partial derivatives of the RSS with respect to b_0 and b_1, setting these derivatives to zero, and solving for the parameters.

An alternative way to minimize the RSS is by using **Gradient descent**, an iterative optimization technique. This method adjusts the parameters gradually to reduce the RSS:

1. Start with initial guesses for b_0 and b_1.

2. Calculate the gradient of the RSS (how RSS changes with respect to each parameter).
3. Update b_0 and b_1 in the direction that reduces the RSS the most.
4. Repeat until the RSS converges to its minimum value.

8.3.6

What does minimizing the residual sum of squares achieve in linear regression?

- Reduces the total prediction error of the regression model.
- Ensures the predicted values exactly match the actual values.
- Makes the regression line pass through all data points.
- Eliminates outliers from the dataset.

8.3.7

Which methods are commonly used to minimize the RSS?

- Least squares estimation using calculus
- Gradient descent optimization
- Ignoring residuals for outliers.
- Trial and error to adjust the line visually.

8.3.8

Least squares method

The Least squares method is a way to find the line that best fits a set of data points. It ensures that the total of all the squared differences (errors) between the actual values and the values predicted by the line is as small as possible.

Imagine you have some points on a graph: each point has an **x**-value (independent variable) and a **y**-value (dependent variable). We want to find a straight line, **$y = mx + b$** (it is the same like previous **$y = b_1x + b_0$**, but for better transparency we use one letter variables), where:

- **m** is the slope of the line (how steep it is),
- **b** is the **y**-intercept (where the line crosses the y-axis).

This line helps us predict **y** for any given **x**, showing the relationship between the two variables.

The formulas to calculate m (slope) and b (intercept) are:

$$m = \frac{n\sum x \cdot y - \sum x \cdot \sum y}{n\sum x^2 - \left(\sum x\right)^2}$$

$$b = \frac{\sum y - m\sum x}{n}$$

Where:

- **n** is the number of data points,
- **∑x** and **∑y** are the sums of all x- and y-values,
- **∑xy** is the sum of the product of each x and y value,
- **∑x²** is the sum of the squares of all x-values.

Procedure:

1. Start with a table that includes columns for x and y values (your data points).
2. Add two more columns: one for xy (multiply each x by its corresponding y) and one for x^2 (square each x-value).
3. Calculate ∑x, ∑y, ∑xy, and $\sum x^2$ by adding up the respective columns.
4. Use the formula for m to calculate the slope.
5. Use the formula for b to find the y-intercept.
6. Plug m and b into the equation y = m * x + b.

Example:

Consider the data:

- x: 1, 2, 3, 4, 5
- y: 2, 5, 3, 8, 7

x	y	xy	x^2
1	2	2	1
2	5	10	4

x	y	xy	x^2
3	3	9	9
4	8	32	16
5	7	35	25

1. Sum the columns:$\sum x = 15$, $\sum y = 25$, $\sum xy = 88$, $\sum x^2 = 55$.
2. Use formula for m = $(n\sum xy - \sum x \sum y) / (n\sum x^2 - (\sum x)^2) = (5 * 88 - 15 * 25) / (5 * 55 - (15)^2) = 65 / 50 = 1.3$
3. Find b =$(\sum y - m\sum x) / n = (25 - (1.3 * 15)) / 5 = 1.1$
4. Write the equation **y = m*x + b = 1.3x + 1.1**

8.3.9

Which of the following is the primary purpose of the least squares method?

- Minimizing the sum of squared errors
- Maximizing the sum of squared errors
- Finding the median of the data
- Maximizing the slope of the regression line

8.3.10

Which statements about the least squares method are true?

- It can be used to predict a dependent variable based on independent variables.
- The sum of the squares of errors is called variance.
- It minimizes the difference between observed and predicted values.
- The slope of the line is determined by trial and error.

8.3.11

What does the slope **m** in the equation **y = mx + b** represent?

- The rate of change of y with respect to x

- The value of y when x=0
- The error in the model
- The intercept of the line

8.3.12

Project: Least Squares Method and Gradient Descent application

Here is a Python program that demonstrates the process of finding the Residual sum of squares (RSS) and minimizing it using both the Least squares method and Gradient descent.

- Dataset - we simulate data points for the independent variable x and dependent variable y.
- RSS calculation - a function calculates the residual sum of squares for given b0 and b1.
- Least squares method computes the optimal b0 and b1 using closed-form formulas derived from calculus.
- Gradient descent iteratively adjusts b0 and b1 to minimize the RSS using a specified learning rate.

```
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

# Example dataset
x = np.array([1, 2, 3, 4, 5])  # Independent variable
y = np.array([1.2, 1.9, 3.2, 4.0, 5.1])  # Dependent variable
```

```
# Function to calculate Residual Sum of Squares (RSS)
def calculate_rss(b0, b1, x, y):
    predicted = b0 + b1 * x
    residuals = y - predicted
    return np.sum(residuals ** 2)

# 1. Least Squares Solution
def least_squares(x, y):
    n = len(x)
    b1 = (np.sum(x * y) - np.sum(x) * np.sum(y) / n) /
(np.sum(x ** 2) - (np.sum(x) ** 2) / n)
    b0 = np.mean(y) - b1 * np.mean(x)
    return b0, b1
```

```
b0_ls, b1_ls = least_squares(x, y)

print(f"Least Squares Solution:")
print(f"Intercept (b0): {b0_ls:.2f}, Slope (b1): {b1_ls:.2f}")
print(f"RSS (Least Squares): {calculate_rss(b0_ls, b1_ls, x,
y):.2f}")
```

Program output:

```
Least Squares Solution:
Intercept (b0): 0.11, Slope (b1): 0.99
RSS (Least Squares): 0.07
```

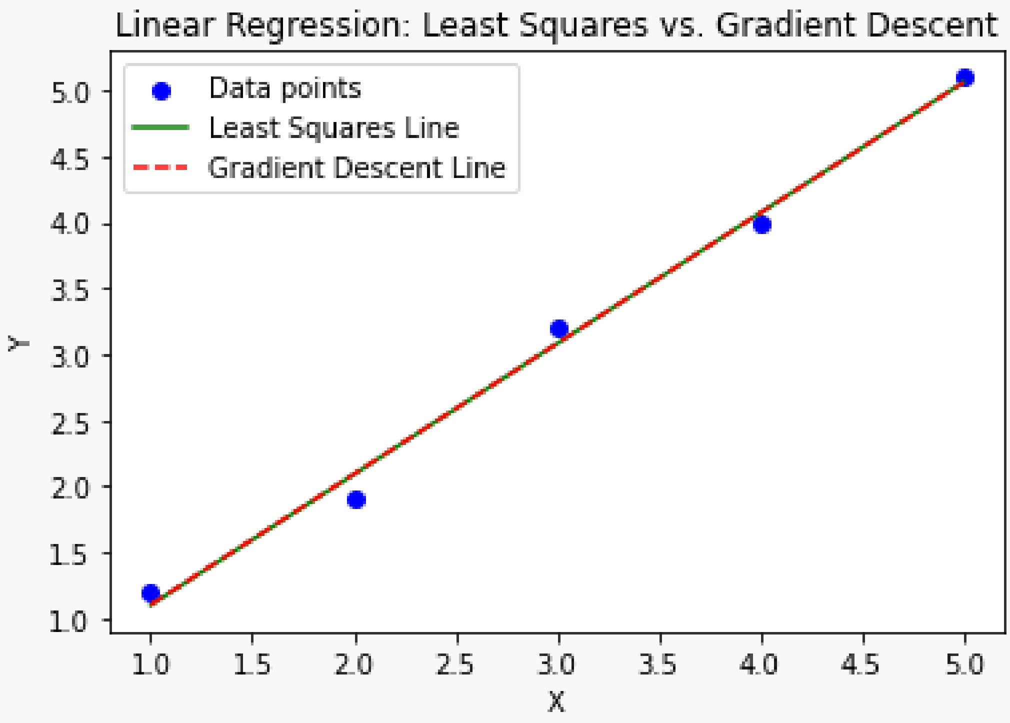
```
# 2. Gradient Descent Solution
def gradient_descent(x, y, learning_rate=0.01,
iterations=1000):
    b0 = 0  # Initial guess for intercept
    b1 = 0  # Initial guess for slope
    n = len(y)

    for _ in range(iterations):
        predicted = b0 + b1 * x
        error = y - predicted

        # Partial derivatives of RSS with respect to b0 and b1
        b0_gradient = -2 * np.sum(error) / n
        b1_gradient = -2 * np.sum(x * error) / n

        # Update parameters using gradient descent
        b0 -= learning_rate * b0_gradient
        b1 -= learning_rate * b1_gradient

    return b0, b1

b0_gd, b1_gd = gradient_descent(x, y, learning_rate=0.01,
iterations=1000)

print(f"\nGradient Descent Solution:")
print(f"Intercept (b0): {b0_gd:.2f}, Slope (b1): {b1_gd:.2f}")
print(f"RSS (Gradient Descent): {calculate_rss(b0_gd, b1_gd,
x, y):.2f}")
```

Program output:

```
Gradient Descent Solution:
Intercept (b0): 0.12, Slope (b1): 0.99
RSS (Gradient Descent): 0.07
```

```
# Visualizing the result
plt.scatter(x, y, color="blue", label="Data points")
plt.plot(x, b0_ls + b1_ls * x, color="green", label="Least
Squares Line")
plt.plot(x, b0_gd + b1_gd * x, color="red", linestyle="--",
label="Gradient Descent Line")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.title("Linear Regression: Least Squares vs. Gradient
Descent")
plt.show()
```

Program output:



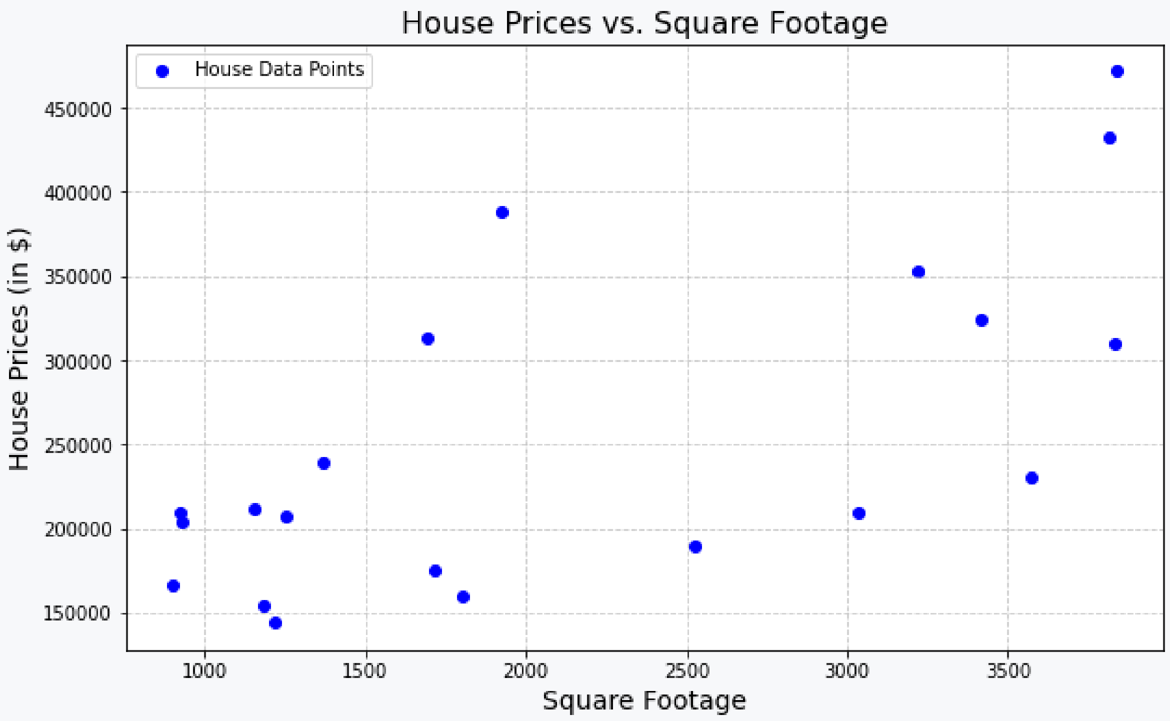
8.3.13

Project: Find price for houses

Solve linear regression for the following dataset and show how it find price for houses with 2500 and 3700 sqft

```
import matplotlib.pyplot as plt

# Data
square_footage = [3419, 1256, 902, 3837, 1926, 1803, 1714,
1371, 3816, 1219,
                  3571, 3833, 3033, 1156, 3218, 2528, 930,
922, 1183, 1695]
house_prices = [324350.89, 207179.35, 166770.06, 471519.89,
387793.16, 160234.44,
                175753.27, 239650.98, 432629.56, 144423.14,
230201.22, 310310.62,
                209227.28, 212224.21, 353152.77, 189225.85,
203751.89, 209049.79,
                154739.81, 313328.74]

# Create scatter plot
plt.figure(figsize=(10, 6))
plt.scatter(square_footage, house_prices, color='blue',
label='House Data Points')

# Labels and title
plt.title("House Prices vs. Square Footage", fontsize=16)
plt.xlabel("Square Footage", fontsize=14)
plt.ylabel("House Prices (in $)", fontsize=14)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()
plt.show()
```

Program output:



Your solution:

```
# write your code
```

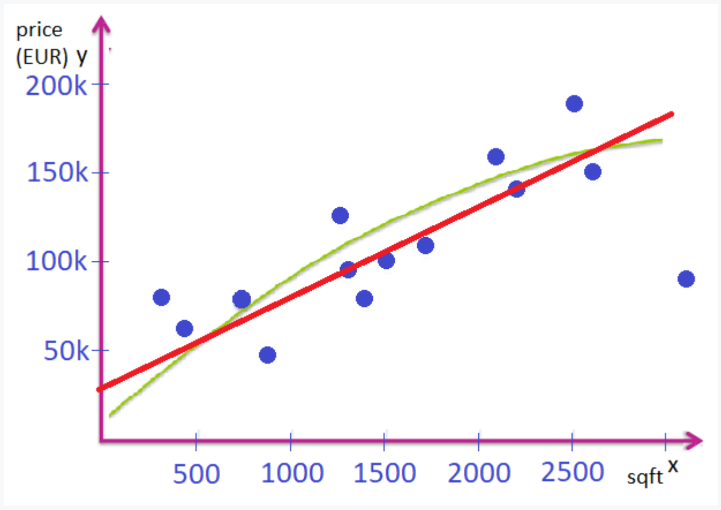
8.4 Advanced linear regression

8.4.1

Quadratic regression

Regression analysis is a powerful tool in predicting relationships between variables. While linear regression captures relationships in a straight line, quadratic regression allows for more complexity by including a squared term. Choosing between these models depends on the nature of your data and the underlying patterns you observe.

Linear regression finds the best straight-line relationship between an independent variable **x** and a dependent variable **y**. This model assumes a constant rate of change: if you increase x by a fixed amount, y changes proportionally. For example, if you're predicting house prices based on square footage, linear regression assumes every additional square foot adds roughly the same amount to the price.



Quadratic regression extends linear regression by adding an $\mathbf{x^2}$ term to the equation, allowing the model to capture curves. This is useful when the relationship between x and y is not constant but varies. For instance, predicting crop yield based on fertilizer use might exhibit diminishing returns: adding small amounts of fertilizer increases yield, but too much could harm the plants. A quadratic regression curve fits this scenario better than a straight line.

The formula for quadratic regression is an extension of linear regression, where the relationship between the independent variable x and the dependent variable y is modeled as a second-degree polynomial. The general form of the quadratic regression equation is:

$$\mathbf{y = b_0 + b_1x + b_2x^2}$$

Where:

- y is the dependent variable (the value we want to predict),
- x is the independent variable (the predictor),
- b_0 is the y-intercept (the value of y when x=0),
- b_1 is the coefficient of the linear term (how much y changes with a unit change in x),
- b_2 is the coefficient of the quadratic term (how the change in x affects y in a non-linear way).

How to compute the coefficients (b_0, b_1, b_2):

1. Create a table of the data points (x, y).
2. Calculate the values of x^2 (square of x) and x^2y (product of x^2 and y).
3. Find the sums:$\sum x,\sum y,\sum x^2,\sum x^2y$
4. Set up the normal equations:

$$\begin{bmatrix} n & \sum x & \sum x^2 \\ \sum x & \sum x^2 & \sum x^3 \\ \sum x^2 & \sum x^3 & \sum x^4 \end{bmatrix} \cdot \begin{bmatrix} b0 \\ b1 \\ b2 \end{bmatrix} = \begin{bmatrix} \sum y \\ \sum xy \\ \sum x^2 y \end{bmatrix}$$

Solve the system of equations to find the coefficients b_0, b_1, b_2.

The quadratic regression model captures the curvature in the data and is useful when the relationship between the variables is not linear.

8.4.2

What type of regression should you use if the relationship between variables changes direction, forming a curve?

- Quadratic regression
- Linear regression
- No regression
- Exponential regression

8.4.3

Project: Quadratic regression

Find the quadratic regression solution for the dataset where home price depends on square footage.

1. Prepare the X matrix:

- Include a column of 1’s for the intercept term (b_0).
- Add x for the linear term and x^2 for the quadratic term.

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv

# Dataset
x = np.array([1200, 1500, 1800, 2000, 2200, 2400, 2600, 2800, 
3000, 3200])  # Square Feet
y = np.array([250000, 300000, 350000, 400000, 450000, 500000, 
550000, 600000, 650000, 320000])  # Home Price
```

```
# Step 1: Prepare matrices for quadratic regression
# X matrix with terms [1, x, x^2]
X = np.column_stack((np.ones(len(x)), x, x**2))
```

2. Normal equation - use the formula b = $(X^T X)^{-1}$ to calculate the coefficients.

```
# Step 2: Solve for coefficients using the normal equation:
Beta = (X^T X)^-1 X^T y
XT = X.T
beta = inv(XT @ X) @ (XT @ y)
```

Computes b_0 (intercept), b_1 (linear term), and b_2(quadratic term).

```
# Step 3: Extract coefficients
b0, b1, b2 = beta
print(f"Coefficients:\nIntercept (b0): {b0}\nLinear term (b1):
{b1}\nQuadratic term (b2): {b2}")
```

Program output:

```
Coefficients:
Intercept (b0): -518643.6228500223
Linear term (b1): 766.5431555285711
Quadratic term (b2): -0.14166684850999348
```

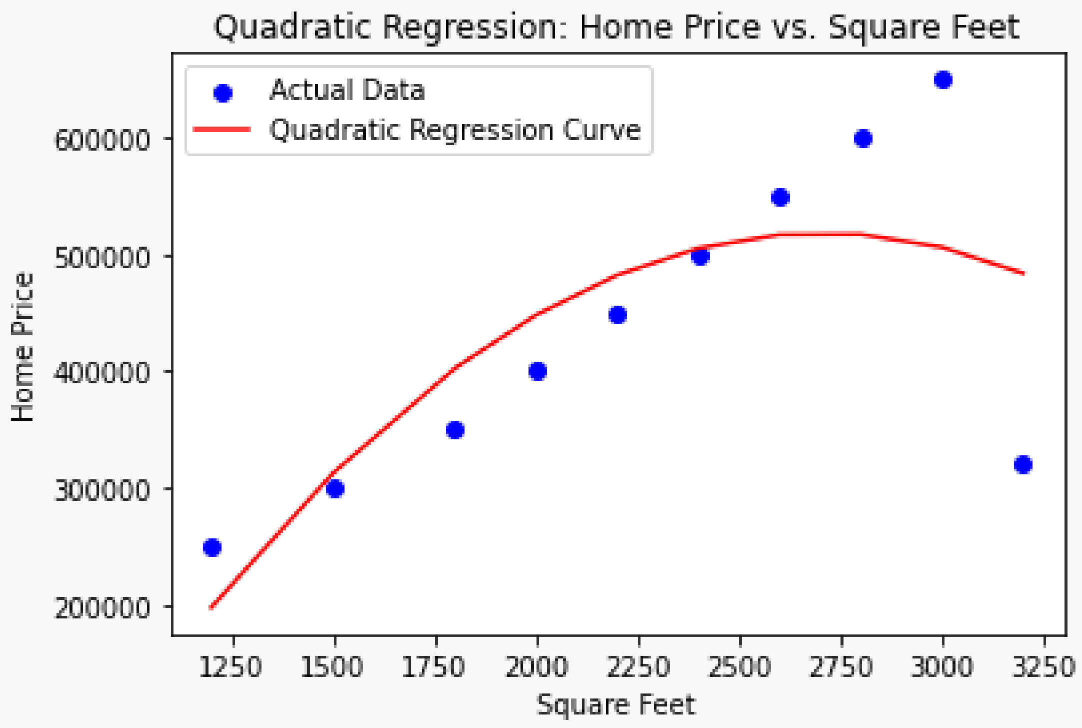
Compute y-values using the quadratic equation.

```
# Step 4: Predicted values
y_pred = b0 + b1 * x + b2 * x**2
```

The scatter plot shows the original data, and the red curve shows the quadratic regression fit.

```
# Step 5: Plot the data and the quadratic regression curve
plt.scatter(x, y, color='blue', label='Actual Data')
plt.plot(x, y_pred, color='red', label='Quadratic Regression
Curve')
plt.xlabel('Square Feet')
plt.ylabel('Home Price')
plt.title('Quadratic Regression: Home Price vs. Square Feet')
plt.legend()
plt.show()
```

Program output:



8.4.4

Polynomial regression

Polynomial regression generalizes quadratic regression by including higher powers of x, such as x^3, x^4 and so on. This flexibility enables the model to capture more complex patterns, but it comes with challenges like overfitting.

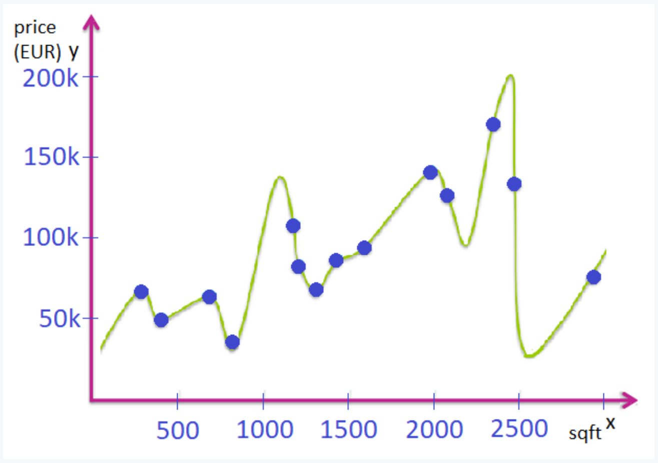
In polynomial regression, the relationship between x and y is modeled as

$$y = b_0 + b_1x + b_2x^2 + \ldots + b_nx^n$$

Where:

- **y** is the dependent variable (the value we want to predict),
- **x** is the independent variable (the predictor),
- **$b_0, b_1, b_2, \ldots, b_n$** are the coefficients that need to be determined,
- **n** is the degree of the polynomial, indicating how many powers of x are included in the model.

Increasing the degree of the polynomial increases the model's complexity, allowing it to adapt to more intricate datasets.



How to compute polynomial regression:

1. Create a table of the data points (x,y).
2. For a polynomial of degree **n**, create additional columns with $x^2, x^3, \ldots, x^n$.
3. Use the least squares method to minimize the error. This involves solving the following normal equations:

$$\mathbf{X^T X b = X^T y}$$

Where:

- **X** is the matrix of input variables (with powers of x),
- **b** is the vector of coefficients $\mathbf{[b_0, b_1, b_2, \ldots, b_n]}$,
- **y** is the vector of observed values.
- The solution to this matrix equation gives the coefficients $\mathbf{b_0, b_1, b_2, \ldots, b_n}$.

Example for a polynomial of degree 3

For a cubic polynomial regression (degree = 3), the equation is:

$$y = b_0 + b_1x + b_2x^2 + b_3x^3$$

You would need to calculate the sums of x, x^2, x^3, x^4, x^5, x^6, and the corresponding sums of y, xy, x^2y, x^3y, and solve the system of equations for the coefficients.

While polynomial regression offers flexibility, it risks overfitting the data, meaning the model becomes too tailored to the training set and performs poorly on new data.

- Polynomial regression can capture more complex relationships compared to linear regression.
- The degree of the polynomial n must be chosen carefully, as a very high degree can lead to overfitting, where the model fits the noise in the data rather than the actual relationship.

8.4.5

Which of the following are true about polynomial regression?

- It can model more complex patterns than linear regression.
- It risks overfitting if the degree is too high.
- It is always better than linear regression.
- It can only handle small datasets.

8.4.6

Project: Polynomial regression

Find the polynomial regression solution for the dataset where home price depends on square footage.

1. Prepare the data

- Input dataset contains x ss square footage of homes, and y as prices of homes.
- **Transform input features by u**se **PolynomialFeatures(degree=4)** from **sklearn.preprocessing** to generate polynomial terms x, x^2, x^3, x^4.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Dataset
x = np.array([1200, 1500, 1800, 2000, 2200, 2400, 2600, 2800,
3000, 3200]).reshape(-1, 1)  # Square Feet
y = np.array([250000, 300000, 180000, 400000, 450000, 250000,
550000, 600000, 650000, 480000])  # Home Price

# Step 1: Transform the input features for polynomial
regression (degree 4)
poly = PolynomialFeatures(degree=4)
x_poly = poly.fit_transform(x)
```

2. Fit the transformed features x into a LinearRegression model.

```
# Step 2: Train a linear regression model with transformed
features
model = LinearRegression()
model.fit(x_poly, y)
```

Generate predictions - use the trained model to predict y-values based on polynomial features.

```
# Step 3: Generate predictions
y_pred = model.predict(x_poly)
```

Display the coefficients for the polynomial regression equation: $y=b_0+b_1x+b_2x^2+b_3x^3+b_4x^4$

```
# Step 4: Extract coefficients for the polynomial equation
coefficients = model.coef_
intercept = model.intercept_
print("Polynomial Regression Equation Coefficients:")
print(f"Intercept: {intercept}")
for i, coeff in enumerate(coefficients[1:], start=1):
    print(f"x^{i}: {coeff}")
```

Program output:

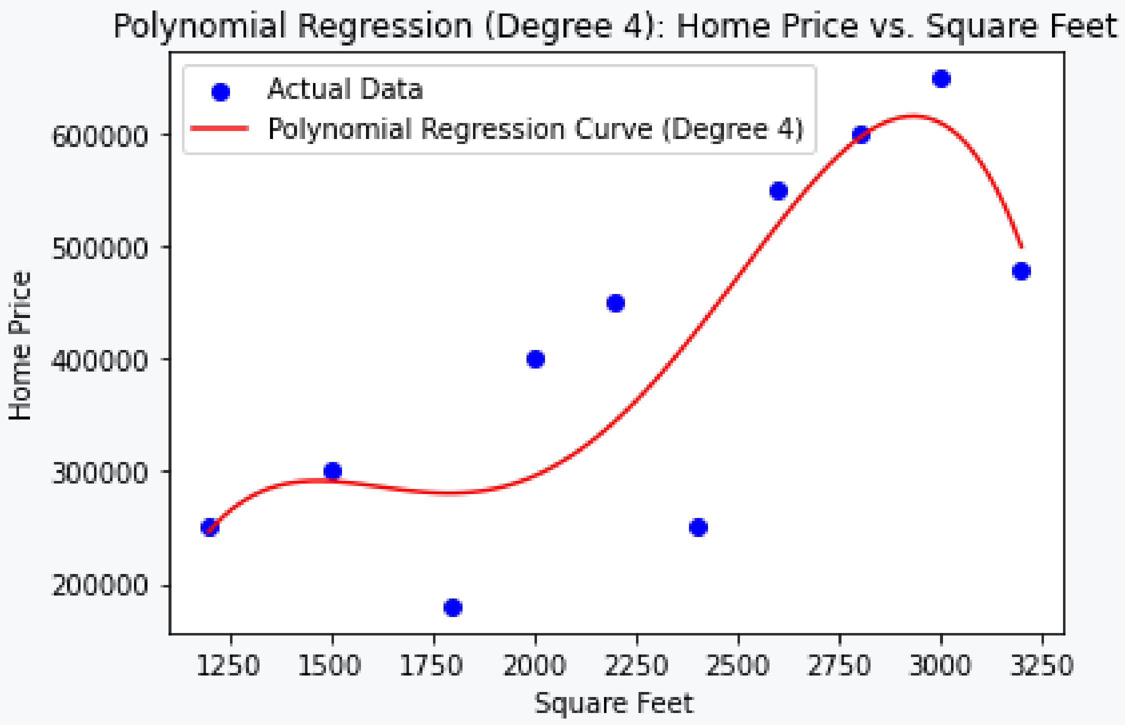
```
Polynomial Regression Equation Coefficients:
Intercept: -5085537.153200174
x^1: 11655.348876638145
x^2: -9.207415285868633
x^3: 0.0031174453504840827
x^4: -3.7746212910860777e-07
```

Create a scatter plot for the original data points and overlay the fitted polynomial regression curve (red line).

```
# Step 5: Plot the data and the polynomial regression curve
plt.scatter(x, y, color='blue', label='Actual Data')
x_line = np.linspace(min(x), max(x), 100).reshape(-1, 1)
y_line = model.predict(poly.transform(x_line))
plt.plot(x_line, y_line, color='red', label='Polynomial
Regression Curve (Degree 4)')
```

```
plt.xlabel('Square Feet')
plt.ylabel('Home Price')
plt.title('Polynomial Regression (Degree 4): Home Price vs.
Square Feet')
plt.legend()
plt.show()
```

Program output:

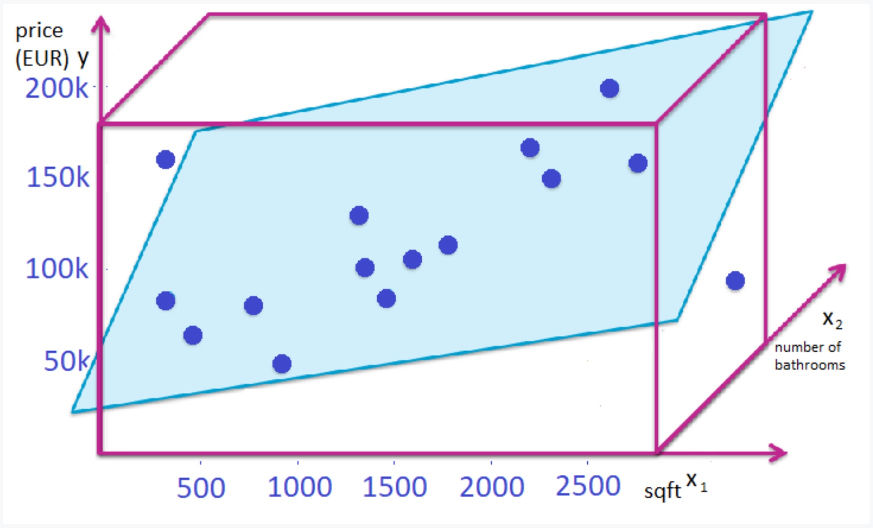


8.4.7

Multivariable regression

Most real-world problems involve more than one independent variable. Multivariable regression (or multiple regression) extends linear regression to consider multiple predictors, allowing us to better understand and predict outcomes.

If we return to the example of estimating the price of a house, we must admit that square meters are not a completely unambiguous and sufficient predictor for determining the price. We can add, for example, the number of bathrooms. In this case, the visualization of the data set will look like in the figure, and the result will not be a regression line, but a plane - in general, it exists in n-dimensional space.



The equation for multivariable regression is:

$$\mathbf{y = b_0 + b_1x_1 + b_2x_2 + \ldots + b_nx_n}$$

where

- $x_1, x_2, \ldots, x_n$ are independent variables (predictors),
- $b_1, b_2, \ldots, b_n$ are their weights respective coefficients.

For example, completed predicting house prices might involve square footage, number of bedrooms, and location as predictors.

Multivariable regression captures the combined effect of multiple factors, but it requires careful data preparation. Variables must be independent of each other (no multicollinearity), and their scales might need normalization to avoid bias.

How to compute the coefficients

1. Your dataset should have multiple predictors (independent variables) and one dependent variable.
2. Organize the data into a matrix: Let **X** be an n × k matrix of the independent variables, where n is the number of observations, and k is the number of independent variables.
3. Let y be an n × 1 vector of the dependent variable.
4. The formula to compute the coefficients $\mathbf{b = [b_0, b_1, \ldots, b_k]^T}$ is derived using matrix algebra, and is given by:

$$\mathbf{b = (X^TX)^{-1} X^T y}$$

Where:

- X^T is the transpose of matrix X,
- X^TX is the matrix product of the transpose of X and X,
- $(X^TX)^{-1}$ is the inverse of the matrix X^TX,
- X^Ty is the matrix product of the transpose of X and the vector y.

The equation $y = b_0 + b_1x_1 + b_2x_2 + \cdots + b_kx_k$ represents a hyperplane in a k-dimensional space. Each independent variable contributes to determining the value of y, and the coefficients $b_1, b_2, ...b_k$ represent the influence of each variable on the prediction.

Example

We have following dataset with square feet, number of bathrooms and home price:

Square Feet (x1)	Number of Bathrooms (x2)	Home Price (y)
1200	2	250000
1500	2	300000
1800	3	350000
2000	3	400000
2200	4	450000
2400	4	500000
2600	5	550000
2800	5	600000
3000	6	650000
3200	6	700000

We need to calculate the following values:

- $\sum x_1$, $\sum x_2$, $\sum y$
- $\sum x_1^2$, $\sum x_2^2$, $\sum x_1x_2$
- $\sum x_1y$, $\sum x_2y$

Let's start with the basic calculations for the dataset:

x1	x2	y	$x1^2$	$x2^2$	x1x2	x1y	x2y
1200	2	250000	1440000	4	2400	300000000	500000
1500	2	300000	2250000	4	3000	450000000	600000
1800	3	350000	3240000	9	5400	630000000	1050000
2000	3	400000	4000000	9	6000	800000000	1200000
2200	4	450000	4840000	16	8800	990000000	1800000
2400	4	500000	5760000	16	9600	1200000000	2000000
2600	5	550000	6760000	25	13000	1430000000	2750000
2800	5	600000	7840000	25	14000	1680000000	3000000
3000	6	650000	9000000	36	18000	1950000000	3900000
3200	6	700000	10240000	36	19200	2240000000	4200000

Now, calculate the following sums:

- $\sum x_1$ = 1200 + 1500 + 1800 + 2000 + 2200 + 2400 + 2600 + 2800 + 3000 + 3200 = 24500
- $\sum x_2$ = 2 + 2 + 3 + 3 + 4 + 4 + 5 + 5 + 6 + 6 = 40
- $\sum y$ = 250000 + 300000 + 350000 + 400000 + 450000 + 500000 + 550000 + 600000 + 650000 + 700000 = 4800000
- $\sum x_1^2$ = 1440000 + 2250000 + 3240000 + 4000000 + 4840000 + 5760000 + 6760000 + 7840000 + 9000000 + 10240000 = 53280000
- $\sum x_2^2$ = 4 + 4 + 9 + 9 + 16 + 16 + 25 + 25 + 36 + 36 = 180
- $\sum x_1x_2$ = 2400 + 3000 + 5400 + 6000 + 8800 + 9600 + 13000 + 14000 + 18000 + 19200 = 108400
- $\sum x_1y$ = 300000000 + 450000000 + 630000000 + 800000000 + 990000000 + 1200000000 + 1430000000 + 1680000000 + 1950000000 + 2240000000 = 13800000000
- $\sum x_2y$ = 500000 + 600000 + 1050000 + 1200000 + 1800000 + 2000000 + 2750000 + 3000000 + 3900000 + 4200000 = 24650000

The general formula for the multivariate regression is:

$y = b_0 + b_1x_1 + b_2x_2$

To solve for b_0, b_1, and b_2, we use the following normal equations:

$$b_1 = \frac{n\sum x_1 y - \sum x_1 \sum y}{n\sum x_1^2 - \left(\sum x_1\right)^2}$$
$$b_2 = \frac{n\sum x_2 y - \sum x_2 \sum y}{n\sum x_2^2 - \left(\sum x_2\right)^2}$$
$$b_0 = \frac{\sum y - b_1 \cdot \sum x_1 - b_2 \cdot \sum x_2}{n}$$

Substitute the values into these formulas:

- b_1 = approx -30.2
- b_2 = 272500
- b_0 = −238900

Now, the regression equation is:

$y = -238900 + (-30.2) \cdot x_1 + 272500 \cdot x_2$

Where:

- y is the predicted home price,
- x_1 is the square footage,
- x_2 is the number of bathrooms.

8.4.8

What is the primary benefit of multivariable regression compared to simple linear regression?

- It considers multiple factors simultaneously.
- It always predicts better than other models.
- It is faster to compute.
- It doesn't require data preparation.

8.4.9

Project: Multivariate regression

To find the correct solution for the given dataset using multiple regression (with two independent variables: square footage and number of bathrooms), we can use the normal equation for multiple linear regression.

We will calculate the regression coefficients b_0 (intercept), b_1 (coefficient for square footage), and b_2 (coefficient for the number of bathrooms) based on the given dataset.

1. We will set up the matrix form for multiple linear regression:

Y = X*b

Where:

- Y is the dependent variable (home prices).
- X is the matrix of independent variables (square footage and number of bathrooms).
- b is the vector of regression coefficients.

2. We will compute $b = (X^T X)^{-1}$ using the normal equation.

```
import numpy as np

# Given data: [Square Feet (x1), Number of Bathrooms (x2),
Home Price (y)]
data = [
    [1200, 2, 250000],
    [1500, 2, 300000],
    [1800, 3, 350000],
    [2000, 3, 400000],
    [2200, 4, 450000],
    [2400, 4, 500000],
    [2600, 5, 550000],
    [2800, 5, 600000],
    [3000, 6, 650000],
    [3200, 6, 700000]
]

# Step 1: Extract X and Y
```

```
X = np.array([ [1, row[0], row[1]] for row in data ])  # Add 1
for intercept
Y = np.array([ row[2] for row in data ])

# Step 2: Calculate (X^T X)^-1 X^T Y
X_T = X.T  # Transpose of X
X_T_X = np.dot(X_T, X)  # X^T * X
X_T_X_inv = np.linalg.inv(X_T_X)  # Inverse of (X^T * X)
X_T_Y = np.dot(X_T, Y)  # X^T * Y

# Step 3: Compute the coefficients (beta)
beta = np.dot(X_T_X_inv, X_T_Y)

# Step 4: Output the results (intercept, coefficient for x1,
and coefficient for x2)
intercept = beta[0]
coef_x1 = beta[1]  # Coefficient for square footage
coef_x2 = beta[2]  # Coefficient for number of bathrooms

print(f"Intercept (b0): {intercept}")
print(f"Coefficient for Square Feet (b1): {coef_x1}")
print(f"Coefficient for Number of Bathrooms (b2): {coef_x2}")

# Step 5: Predict prices for given square footage and number
of bathrooms
def predict_price(x1, x2):
    return intercept + coef_x1 * x1 + coef_x2 * x2

# Example: Predict price for a house with 2500 sqft and 4
bathrooms
predicted_price = predict_price(2500, 4)
print(f"Predicted Price for 2500 sqft, 4 bathrooms:
{predicted_price}")
```

Program output:

```
Intercept (b0): -30769.23076920415
Coefficient for Square Feet (b1): 192.30769230769954
Coefficient for Number of Bathrooms (b2): 17307.692307692378
Predicted Price for 2500 sqft, 4 bathrooms: 519230.7692308142
```

8.5 Logistic regression

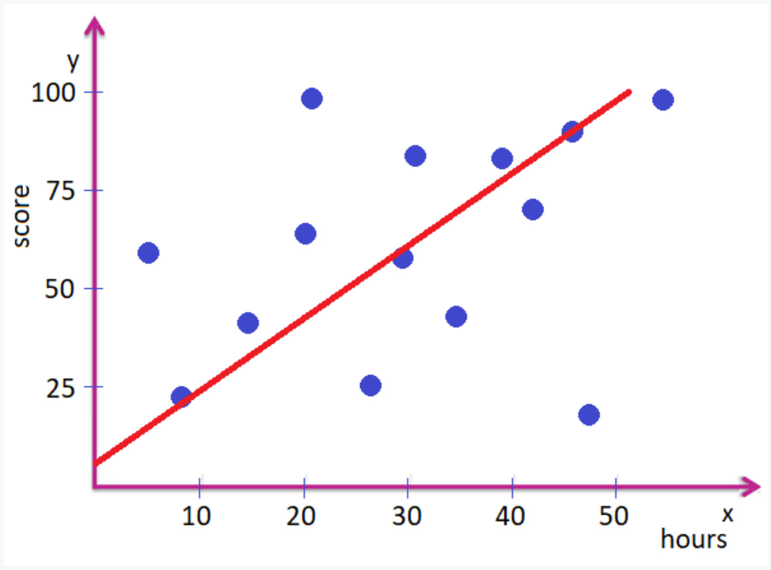
🕮 8.5.1

Imagine we want to study the relationship between features like **hours studied** and **exam success**. A common approach is to model this relationship using linear regression. Here, we assume that the dependent variable, such as the exam score, is related to the independent variable (hours studied) through a straight-line equation.

For example:

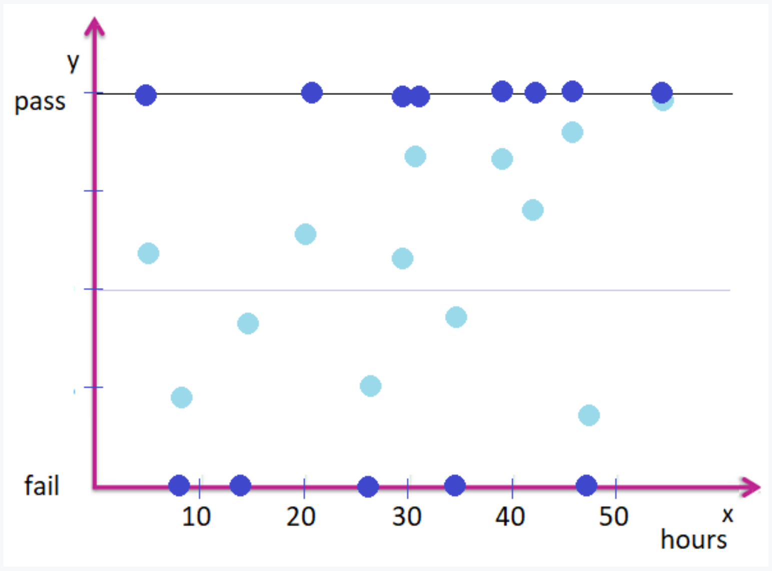
Exam score = m * (Hours studied) + b

By learning the slope (m) and intercept (b) from the data, we can predict the exam score for any number of hours studied.



Now imagine we aren't interested in predicting the exact exam score. Instead, we want to know whether a student will "pass" or "fail." To do this, we can take the output of the linear regression and assign it to one of two classes:

- Pass - if the predicted score is greater than a certain threshold (e.g., 50).
- Fail - if the predicted score is less than or equal to the threshold.



This simple transformation turns the continuous output of linear regression into a binary prediction about success or failure.

What if we could do more than just assign a class? Imagine a student studies for 10 hours, and the model predicts that they will pass. Wouldn't it be helpful to also know **how confident** the model is in this prediction? For instance, does the model believe the student has a 90% chance of passing, or is it closer to 55%? This idea of attaching probabilities to predictions can provide richer insights and guide better decisions.

8.5.2

Why is linear regression commonly used to model the relationship between two variables?

- It models relationships with a straight-line equation.
- It assumes random relationships.
- It ignores independent variables.
- It only predicts binary outcomes.

8.5.3

When using a linear model, how can we classify predictions into two classes?

- By setting a threshold value to split the predictions.
- By ignoring the continuous outputs.
- By always assuming the output is 1.
- By multiplying the prediction by a constant factor.

8.5.4

The limits of linear models for binary predictions

Consider scenarios where we want binary outcomes:

- Health for predicting whether a patient has a disease (yes or no).
- E-commerce to decide if a customer will buy a product (buy or not buy).
- Education to determine if a student will pass a course (pass or fail).

For these situations, we use a binary response variable, Y:

- Y = 1 if "yes"
- Y = 0 if "no"

We aim to model the **probability** that Y=1. For example, what is the likelihood that a customer will buy a product?

At first glance, it might seem logical to use linear regression to predict probability. However, linear regression has limitations:

- The predicted probabilities can go **below 0** or **above 1**, which doesn't make sense in a real-world context.
- Probabilities must always stay within the interval [0, 1].
- If we predict a probability of -10%, what does that mean?
- If the predicted probability is 200%, it's equally nonsensical.

To fix this problem, we need a model that:

1. Keeps probabilities within the interval [0, 1].
2. Provides us with meaningful confidence levels for predictions.

8.5.5

Why is it inappropriate to use linear regression to predict probabilities?

- Linear regression can produce values outside the range [0, 1].
- Probabilities are always fixed.
- Linear regression ignores input variables.
- Probabilities require exact values, not approximations.

8.5.6

What does a probability of 55% indicate in a binary prediction model?

- A moderate likelihood of success.
- A guarantee of failure.
- An invalid prediction.
- A probability outside the acceptable range.

8.5.7

Predicting outcomes with confidence

Let's imagine a company trying to predict whether a customer will buy their new product. They collect data on several features, such as income and online shopping habits, and build a simple linear regression model.

For example:

Likelihood of purchase = 0.2 * Income – 0.5

Using this model, they predict:

- A customer with an income of 5,000/month will have a likelihood of 0.5 (50%).
- Another customer with 10,000/month will have a likelihood of 1.5 (150%).

Problem with linear predictions

The issue becomes obvious: probabilities greater than 100% (or less than 0%) are meaningless in real life. A value of 150% cannot represent the likelihood of a purchase. Similarly, a negative likelihood, say -10%, doesn't make sense.

A better approach is to use probability model

What if instead of a direct linear relationship, we use a model that squashes predictions into a range between 0 and 1? For example:

- A customer with a 5,000/month income might have a probability of 55% to make a purchase.
- Another customer with 10,000/month income might have a probability of 90%.

These probabilities provide clearer, actionable insights for marketing campaigns. Such a model gives us two things:

1. Predictions that always stay between 0 and 1.
2. A confidence level (e.g., "90% sure the customer will buy").

8.5.8

What is the main problem with using linear regression for binary outcomes like "buy" or "not buy"?

- It can produce values outside the range [0, 1].
- It always predicts the wrong class.
- It requires more data than binary models.
- It only works for three or more classes.

8.5.9

Transforming Linear regression to Logistic regression

Suppose we have a linear regression model to predict whether a customer will buy a product. The model is given as:

$z = 1.5 * \text{Income} - 0.5 * \text{Age} + 2$

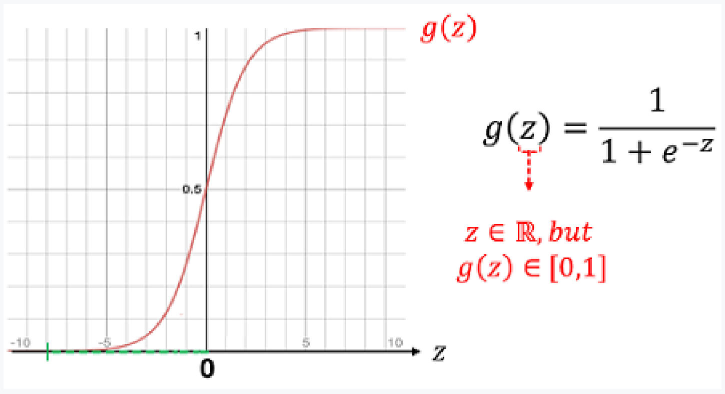
- where, z is the raw score. Based on z, we want to predict whether the customer will buy (1) or not buy (0).

The values of z can range from $-\infty$ to $+\infty$. However, probabilities must always fall between 0 and 1. Linear regression cannot directly handle this, so we need a transformation.

The logistic function transforms z into a probability P (g(z)) that lies between 0 and 1:

$P = 1 / (1+e^{-z})$

This function maps any z value into the range (0, 1). It is also known as the **sigmoid function**.



The probability P represents the likelihood of the event occurring (e.g., the customer buying the product).

For our model:

P=1 / (1+e$^{-(1.5 \cdot \text{Income} - 0.5 \cdot \text{Age} + 2)}$)

This probability P now gives us a clearer measure of how likely a customer is to buy based on their income and age.

Let’s calculate the probability for a customer with:

- Income = 50 (in thousands)
- Age = 30

First, compute z:

- z = 1.5*50 − 0.5*30 + 2 = 75 − 15 + 2 = 62

Now apply the logistic function to find P:

- P = 1 / (1 + e^{-62})
- For large positive values of z, e^{-z} becomes very small, so P≈1 / (1 + 0) = 1
- This means there’s a very high probability (almost 100%) that this customer will buy the product.

Decision-making

- If P≥0.5, we classify the customer as "likely to buy."
- If P<0.5, we classify the customer as "unlikely to buy."

The transformation from linear regression to logistic regression allows us to interpret predictions as probabilities. Instead of just a raw score, we now get a likelihood that can guide decisions in real-world scenarios.

8.5.10

What does a probability of 90% mean in the context of predicting customer subscriptions?

- The model is highly confident the customer will subscribe.
- The model guarantees the customer will subscribe.
- The customer has already subscribed.
- The customer won't subscribe.

8.5.11

Why is it better to predict probabilities instead of just using a score?

- Probabilities provide a measure of confidence.
- Probabilities eliminate incorrect predictions.
- Probabilities don't require labeled data.
- Probabilities ensure perfect predictions.

8.5.12

Project: Buy or not

We are working for a marketing team that wants to predict whether a customer will purchase a product based on their annual income and age. The dataset contains binary outcomes:

- y=1, if the customer purchased the product.
- y=0, if the customer did not purchase.

Using logistic regression, we will predict the probability of purchase for a given customer and classify them.

1. Dataset preparation

- The dataset contains features **Income** and **Age**, along with a binary target variable **Purchased**.
- A synthetic formula is used to create the target variable y based on a linear combination of Income and Age with added noise.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

# Step 1: Create the Dataset
np.random.seed(42)  # For reproducibility
data = {
    'Income': np.random.randint(30000, 100000, size=100),  #
Annual income in money
    'Age': np.random.randint(20, 60, size=100),           #
Age in years
}
# Binary target variable (purchase or not)
data['Purchased'] = np.where(
    (0.05 * (data['Income'] - 50000)) - (0.03 * (data['Age'] -
30)) + np.random.normal(0, 2, 100) > 0, 1, 0
)

df = pd.DataFrame(data)
```

2. Split and train

- Split the dataset into training (80%) and testing (20%) subsets.
- A logistic regression model is fit on the training data using **Income** and **Age** as predictors.

```
# Step 2: Split the Dataset
X = df[['Income', 'Age']]
y = df['Purchased']

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

# Step 3: Train Logistic Regression Model
```

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

3. Predictions are made on the test dataset. Probabilities for class y=1 (purchase) are also computed.

```
# Step 4: Make Predictions
y_pred = model.predict(X_test)
y_pred_prob = model.predict_proba(X_test)[:, 1]  #
Probabilities for class 1
```

4. Model evaluation - the accuracy, confusion matrix, and classification report.

```
# Step 5: Evaluate the Model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred)

# Display Results
print("Accuracy:", accuracy)
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", report)
```

Program output:

```
Accuracy: 1.0

Confusion Matrix:
 [[ 6  0]
 [ 0 14]]

Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00         6
           1       1.00      1.00      1.00        14

    accuracy                           1.00        20
   macro avg       1.00      1.00      1.00        20
weighted avg       1.00      1.00      1.00        20
```

For a customer with specific values for **Income** and **Age**, the model predicts whether they will purchase the product and the associated probability.

```
# Example Prediction
new_customer = pd.DataFrame({'Income': [70000], 'Age': [35]})
predicted_class = model.predict(new_customer)
predicted_prob = model.predict_proba(new_customer)[:, 1]

print("\nFor a customer with Income $70,000 and Age 35:")
print(f"Predicted Class: {predicted_class[0]} (1 = Purchase, 0
= No Purchase)")
print(f"Probability of Purchase: {predicted_prob[0]:.2f}")
```

Program output:

```
For a customer with Income $70,000 and Age 35:
Predicted Class: 1 (1 = Purchase, 0 = No Purchase)
Probability of Purchase: 1.00
```

Clustering

Chapter 9

9.1 K-means clustering

9.1.1

Clustering is an essential machine learning technique used to group data points based on their similarity. Unlike classification, clustering is an **unsupervised learning** method, meaning it does not rely on labeled data. Instead, the algorithm identifies inherent patterns or structures in the data, forming clusters that group similar items together. This is particularly useful in real-world applications like customer segmentation, document classification, and image compression.

The key goal of clustering is to minimize the **intra-cluster distance** (distance within the same cluster) and maximize the **inter-cluster distance** (distance between clusters). The two most popular clustering algorithms are **K-Means** and **Hierarchical Clustering**. Each approach has strengths, depending on the dataset's structure and the problem you aim to solve.

To better understand clustering, imagine you're tasked with organizing books in a library. Without knowing the categories beforehand, you might group them by genre, size, or cover color. Similarly, clustering groups data points by their features, uncovering natural groupings in the dataset.

9.1.2

What type of learning does clustering belong to?

- Unsupervised learning
- Supervised learning
- Reinforcement learning
- Semi-supervised learning

9.1.3

What is the primary goal of clustering?

- To group similar data points together.
- To label data points.
- To classify new data points.
- To predict numerical outcomes.

9.1.4

Why is clustering important? Consider a marketing example: a company wants to create targeted campaigns for its customers. Without prior knowledge, grouping customers based on their purchasing habits, demographics, or browsing history can reveal distinct segments like "frequent buyers" or "budget-conscious customers." These insights drive better decision-making.

Clustering also finds applications in fields like biology (gene grouping), astronomy (galaxy classification), and cybersecurity (malware detection). By uncovering hidden patterns, clustering enables more effective resource allocation and strategy formulation. Moreover, its unsupervised nature makes it versatile, especially when labeled data is unavailable or expensive to obtain.

Clustering doesn't always give perfectly defined groups. It's important to remember that the quality of the clusters depends on the algorithm used and the dataset. Evaluation metrics like the **Silhouette Score** or **Davies-Bouldin Index** help determine how well the clustering performed.

9.1.5

Select all correct applications of clustering.

- Customer segmentation.
- Malware detection.
- Predicting stock prices.
- Calculating the area of a triangle.

9.1.6

K-Means clustering

K-Means is one of the most widely used clustering algorithms due to its simplicity and efficiency. The idea is to partition the dataset into **k** clusters, where each data point belongs to the cluster with the nearest **centroid** (average of all points in that cluster). The process involves the following steps:

1. Initialization - select k initial centroids randomly.
2. Assign each data point to the nearest centroid, forming k clusters.
3. Calculate the new centroid for each cluster by averaging the data points within it.
4. Iterate steps 2 and 3 until centroids stabilize (no significant change).

K-Means requires choosing the value of k beforehand. Techniques like the **Elbow Method** help determine the optimal k by analyzing the sum of squared distances between data points and their centroids. The ideal k minimizes this value without overfitting.

Example

To understand K-Means better, let's go through an example. Suppose we have 6 points: (2, 3), (3, 3), (6, 8), (7, 9), (1, 2), and (8, 9), and we want to divide them into 2 clusters (k=2):

1. Initialize centroids - the simplest method is to randomly select k points from the dataset to serve as the initial centroids. These points are chosen without any specific pattern, and the algorithm then starts from these points - start with **(3, 3) as Centroid 1** and **(8, 9) as Centroid 2**.
2. Compute the Euclidean distance of each point to the centroids. The Euclidean distance between two points (x_1, y_1) and (x_2, y_2) in a 2D space is calculated using the formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Where:

- d is the distance between the two points.
- (x_1,y_1) and (x_2,y_2) are the coordinates of the points.

Assign each point to the nearest centroid:

Point (2, 3):

- Distance to Centroid 1 (3, 3): d = sqrt ($(3-2)^2+(3-3)^2$) = 1
- Distance to Centroid 2 (8, 9): d = sqrt ($(8-2)^2+(9-3)^2$) = 8.49
- belogs to **Centroid 1**

Point (3, 3):

- Distance to Centroid 1 (3, 3): d = sqrt ($(3-3)^2+(3-3)^2$) = 0
- Distance to Centroid 2 (8, 9): d = sqrt ($(8-3)^2+(9-3)^2$) = 7.81
- belogs to **Centroid 1**

Point (6, 8):

- Distance to Centroid 1 (3, 3): d = sqrt $((6-3)^2+(8-3)^2)$ = 5.83
- Distance to Centroid 2 (8, 9): d = sqrt $((8-6)^2+(9-8)^2)$ = 2.24
- belogs to **Centroid 2**

Point (7, 9):

- Distance to Centroid 1 (3, 3): d = sqrt $((7-3)^2+(9-3)^2)$ = 7.21
- Distance to Centroid 2 (8, 9): d = sqrt $((8-7)^2+(9-9)^2)$ = 1
- belogs to **Centroid 2**

Point (1, 2):

- Distance to Centroid 1 (3, 3): d = sqrt $((1-3)^2+(2-3)^2)$ = 2.24
- Distance to Centroid 2 (8, 9): d = sqrt $((8-1)^2+(9-2)^2)$ = 9.9
- belogs to **Centroid 1**

Point (8, 9):

- Distance to Centroid 1 (3, 3): d = sqrt $((8-3)^2+(9-3)^2)$ = 7.81
- Distance to Centroid 2 (8, 9): d = sqrt $((8-1)^2+(9-2)^2)$ = 0
- belogs to **Centroid 2**

Update centroids

For each cluster, calculate the mean of its points:

- Cluster 1 centroid: (2 + 3 + 1) / 3,(3 + 3 + 2) / 3 = (2, 2.67)
- Cluster 2 centroid: (6 + 7 + 8) / 3,(8 + 9 + 9) / 3 = (7, 8.67)

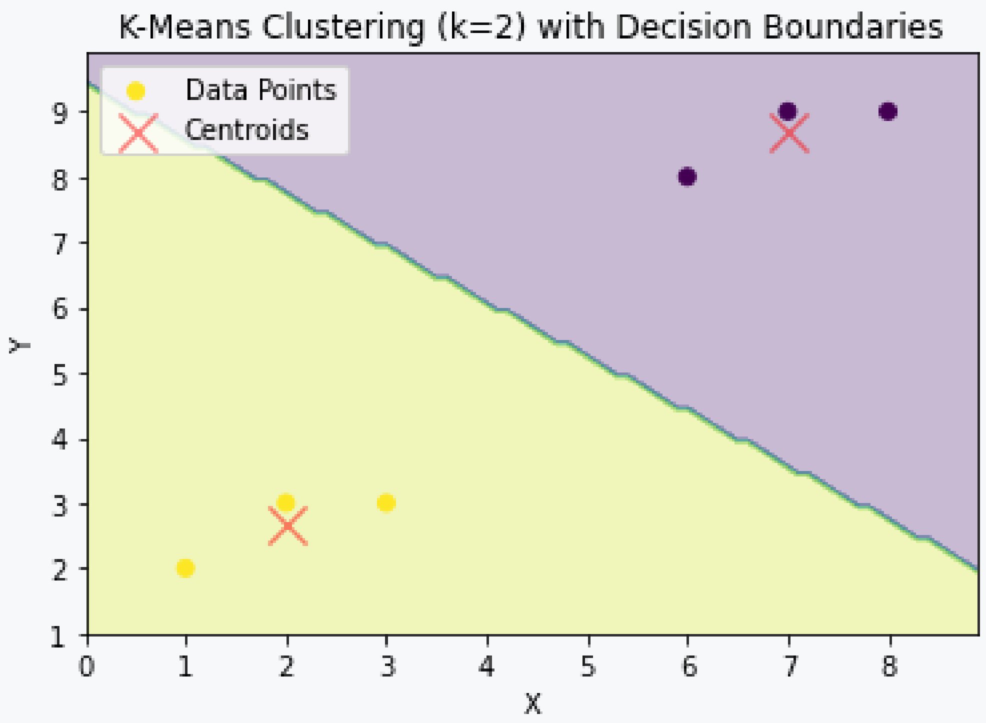
Reassign points by recalculate centroids until stable - if the cluster membership of points does not change during the recalculation, it means that they are stable

Solution by code:

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# Define the data points (6 points in 2D space)
data_points = np.array([[2, 3], [3, 3], [6, 8], [7, 9], [1,
2], [8, 9]])

# Initialize the KMeans algorithm with 2 clusters (k=2)
kmeans = KMeans(n_clusters=2)

# Fit the KMeans algorithm to the data points
kmeans.fit(data_points)

# Get the centroids of the clusters
centroids = kmeans.cluster_centers_

# Get the labels (which cluster each point belongs to)
labels = kmeans.labels_

# Create a grid of points to visualize decision boundaries
x_min, x_max = data_points[:, 0].min() - 1, data_points[:,
0].max() + 1
y_min, y_max = data_points[:, 1].min() - 1, data_points[:,
1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
np.arange(y_min, y_max, 0.1))

# Predict the cluster for each point in the grid
Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot the decision boundary by filling the background with
the predicted clusters
plt.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')

# Plot the points and the centroids
plt.scatter(data_points[:, 0], data_points[:, 1], c=labels,
cmap='viridis', marker='o', label='Data Points')

# Plot the centroids
```

```
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', s=200,
alpha=0.5, marker='x', label='Centroids')

# Add labels and title
plt.xlabel('X')
plt.ylabel('Y')
plt.title('K-Means Clustering (k=2) with Decision Boundaries')
plt.legend()

# Display the plot
plt.show()
```

Program output:



9.1.7

What is the purpose of centroids in K-Means clustering?

- To represent the center of a cluster.
- To initialize the number of clusters.
- To label data points.
- To calculate the distance between clusters.

9.1.8

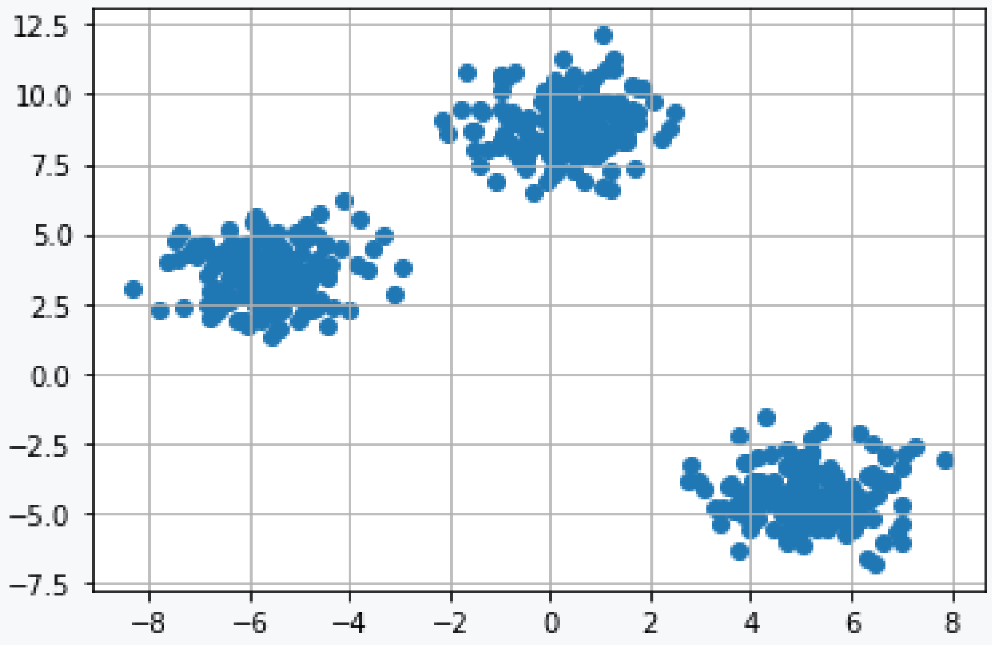
Project: Understanding clusterisation

(by https://www.geeksforgeeks.org/k-means-clustering-introduction/)

Lets go to create 3 groups of values as random points

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

X,y = make_blobs(n_samples = 500,n_features = 2,centers =
3,random_state = 23)

fig = plt.figure(0)
plt.grid(True)
plt.scatter(X[:,0],X[:,1])
plt.show()
```

Program output:



Initialize random centers

- The code initializes three cluster centers. It sets random cluster centers within the specified range and creates an empty list of points for each cluster.

```
# Number of clusters
k = 3

# Initialize an empty dictionary to store clusters
clusters = {}

# Set a random seed for reproducibility
np.random.seed(23)

# Iterate through each cluster to initialize its center
for idx in range(k):
    # Generate random points for the center within a specified
range
    center = 2 * (2 * np.random.random((X.shape[1],)) - 1)

    # Create an empty list to store the points of the cluster
    points = []

    # Define the cluster with its center and an empty points
list
    cluster = {
        'center': center,
        'points': []
    }

    # Add the cluster to the clusters dictionary
    clusters[idx] = cluster

# Output the initialized clusters
print(clusters)
```

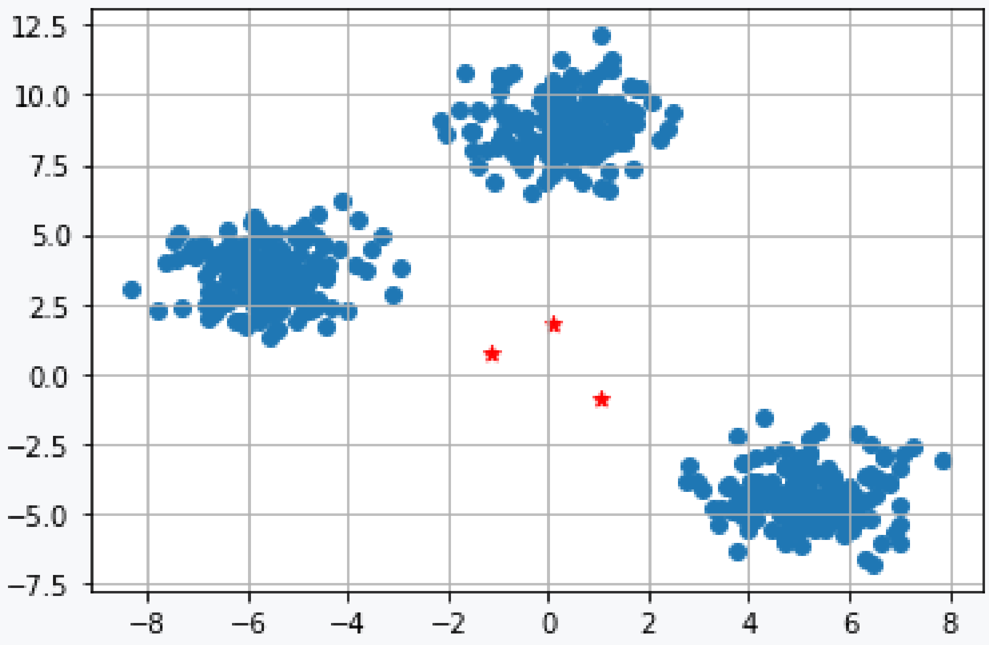
Program output:

```
{0: {'center': array([0.06919154, 1.78785042]), 'points': []},
1: {'center': array([ 1.06183904, -0.87041662]), 'points':
[]}, 2: {'center': array([-1.11581855,  0.74488834]),
'points': []}}
```

Plot the random initialize center with data points

```
plt.scatter(X[:,0],X[:,1])
plt.grid(True)
for i in clusters:
    center = clusters[i]['center']
    plt.scatter(center[0],center[1],marker = '*',c = 'red')
plt.show()
```

Program output:



Define Euclidean distance

```
def distance(p1,p2):
    return np.sqrt(np.sum((p1-p2)**2))
```

Functions assign data points to the nearest cluster center, and updates cluster centers based on the mean of assigned points in K-means clustering.

```
# Assign each point to the nearest cluster based on distance
def assign_clusters(X, clusters):
    for idx in range(X.shape[0]):
        dist = []  # List to store distances to all cluster
centers

        curr_x = X[idx]  # Current point

        # Calculate the distance to each cluster center
        for i in range(k):
            dis = distance(curr_x, clusters[i]['center'])
            dist.append(dis)
        # Assign the point to the nearest cluster
        curr_cluster = np.argmin(dist)  # Index of the cluster 
with minimum distance
        clusters[curr_cluster]['points'].append(curr_x)  # Add 
point to corresponding cluster

    return clusters

# Recalculate the cluster centers based on assigned points
def update_clusters(X, clusters):
    for i in range(k):
        points = np.array(clusters[i]['points'])  # Points 
assigned to the cluster

        # If there are points assigned to the cluster, update 
the center
        if points.shape[0] > 0:
            new_center = points.mean(axis=0)  # Calculate the 
new center as the mean of points
            clusters[i]['center'] = new_center  # Update the 
cluster center

            clusters[i]['points'] = []  # Reset the points 
list for the next iteration

    return clusters
```

The function to Predict the cluster for the datapoints

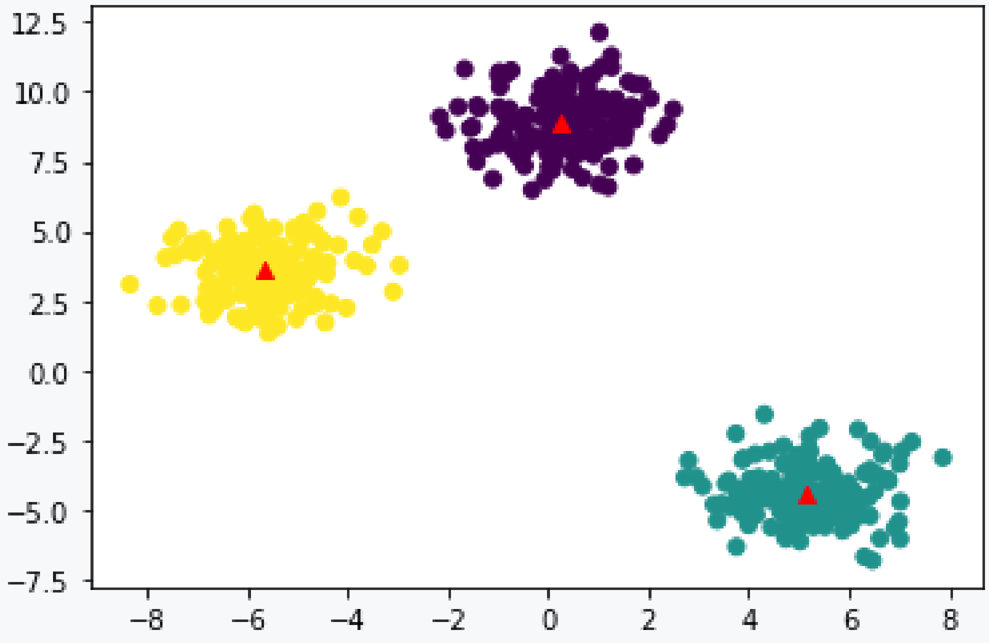
```
# Function to predict the cluster for each point based on the 
nearest centroid
def pred_cluster(X, clusters):
    pred = []  # List to store the predicted cluster for each 
point

    # Loop through each point in the dataset
    for i in range(X.shape[0]):
        dist = []  # List to store the distances to each 
cluster center

        # Calculate the distance to each cluster center
        for j in range(k):
            dist.append(distance(X[i], clusters[j]['center']))
        # Assign the point to the nearest cluster based on the
minimum distance
        pred.append(np.argmin(dist))   # The cluster index with
the minimum distance

    return pred
```

Process - assign, update, predict

```
# Assign clusters to the points based on the current centroids
clusters = assign_clusters(X, clusters)

# Update the centroids based on the current points in each
cluster
clusters = update_clusters(X, clusters)

# Predict the cluster for each point based on the nearest
centroid
pred = pred_cluster(X, clusters)
```

```
plt.scatter(X[:,0],X[:,1],c = pred)
for i in clusters:
    center = clusters[i]['center']
    plt.scatter(center[0],center[1],marker = '^',c = 'red')
plt.show()
```

Program output:



The plot shows data points colored by their predicted clusters. The red markers represent the updated cluster centers after the K-means clustering algorithm.

9.2 Normalisation

9.2.1

Importance of data normalization

In machine learning, data normalization is crucial when features in a dataset have different scales. For instance, one feature could range from 0 to 1000 while another ranges from 0 to 1. Without normalization, algorithms that calculate distances might assign greater importance to the larger-scale feature, distorting the results.

Normalization scales all features to a similar range, usually between 0 and 1 or with zero mean and unit variance (standardization). This process ensures that all features contribute equally to the analysis.

Clustering algorithms, such as K-Means, use distances to group data points. Without normalization:

- Features with larger scales dominate distance calculations.
- The clustering output may incorrectly group data based on the dominant feature.

With normalized data:

- Each feature contributes equally.
- Clusters better represent the inherent structure of the data.

9.2.2

Why is data normalization important in clustering?

- It ensures that features contribute equally to distance calculations.
- It makes the data more complex.
- It reduces the number of clusters.
- It improves the algorithm's speed.

9.2.3

What are the benefits of data normalization?

- Prevents larger-scale features from dominating.
- Improves clustering accuracy.
- Eliminates the need for feature selection.
- Guarantees fewer clusters.

9.2.4

Normalization methods

Normalization methods transform data to a specific scale or distribution. Below are common normalization techniques, their purposes, and how to implement them in Python.

1. Min-max normalization

Scales data to a range, typically [0, 1]. It is sensitive to outliers. x is element, X is set.

$x_{new} = (x_{original} - \min(X)) / (\max(X) - \min(X))$

When to use:

- When features need to be scaled within a specific range.
- Suitable for machine learning models sensitive to the magnitude of data (e.g., K-Means, Gradient Descent).

```
from sklearn.preprocessing import MinMaxScaler
import numpy as np

data = np.array([[1, 2], [3, 4], [5, 6]])
scaler = MinMaxScaler()
normalized_data = scaler.fit_transform(data)
print(data)
print(normalized_data)
```

Program output:

```
[[1 2]
 [3 4]
 [5 6]]
[[0.  0. ]
 [0.5 0.5]
 [1.  1. ]]
```

2. Standardization (Z-score normalization)

Scales data to have a mean (μ) of 0 and a standard deviation (σ) of 1.

$x_{new} = (x_{original} - \mu) / \sigma$

When to use:

- When features need to be standardized (e.g., Principal Component Analysis).
- Robust to outliers compared to Min-Max normalization.

```
from sklearn.preprocessing import StandardScaler
import numpy as np

data = np.array([[1, 2], [3, 4], [5, 6]])
scaler = StandardScaler()
standardized_data = scaler.fit_transform(data)
print(data)
print(standardized_data)
```

Program output:

```
[[1 2]
 [3 4]
 [5 6]]
[[-1.22474487 -1.22474487]
 [ 0.          0.        ]
 [ 1.22474487  1.22474487]]
```

3. Max absolute scaling

Scales each feature by its maximum absolute value. The range is [-1, 1].

$x_{new} = x_{original} / \max(|X|)$ or $x_{new} = x_{original} / \max(abs(X))$

When to use:

- For datasets with positive and negative values where the magnitude is important.

```
from sklearn.preprocessing import MaxAbsScaler
import numpy as np

data = np.array([[1, -2], [3, -4], [5, -6]])
scaler = MaxAbsScaler()
```

```
max_abs_scaled_data = scaler.fit_transform(data)
print(data)
print(max_abs_scaled_data)
```

Program output:

```
[[ 1 -2]
 [ 3 -4]
 [ 5 -6]]
[[ 0.2        -0.33333333]
 [ 0.6        -0.66666667]
 [ 1.         -1.        ]]
```

9.2.5

What is the typical range of values after applying Min-Max normalization?

- [0, 1]
- [-1, 1]
- [0, ∞]
- No fixed range

9.2.6

Which of the following are characteristics of Min-Max normalization?

- It scales data to a specific range
- It is sensitive to outliers
- It is robust to outliers
- It always centers the data at 0

9.2.7

After Z-score normalization, what are the mean and standard deviation of the data?

- Mean = 0, Standard deviation = 1
- Mean = 1, Standard deviation = 1
- Mean = 0, Standard deviation = 0
- Mean = 1, Standard deviation = 0

9.2.8

Which of the following are true about Z-score normalization?

- It centers the data around zero
- It standardizes data to have unit variance
- It is robust to outliers
- It requires the data to be strictly positive

9.2.9

What is the primary characteristic of Max absolute scaling?

- Scales values based on their maximum absolute value
- Scales values to a range of [0, 1]
- Centers the data at zero
- It only works for positive values

9.2.10

Which of the following are true about Max absolute scaling?

- It scales values between [-1, 1]
- It maintains the sparsity of data
- It is robust to outliers
- It centers the data at zero

9.2.11

Logarithmic normalization

Logarithmic normalization is a preprocessing technique often used to handle data with a wide range of values. Many real-world datasets exhibit skewed distributions, where a few values dominate the range. This can cause issues in clustering, as algorithms like K-Means rely on Euclidean distance, which is sensitive to large variations. Logarithmic normalization transforms such data into a compressed scale by applying a log function, reducing the effect of extreme values and making distributions more manageable for clustering.

For example, raw data like **[1, 10, 100, 1000]** would be transformed into **[0, 1, 2, 3]** using logarithmic normalization (logarithm base 10).

The logarithmic normalization is important for clustering because:

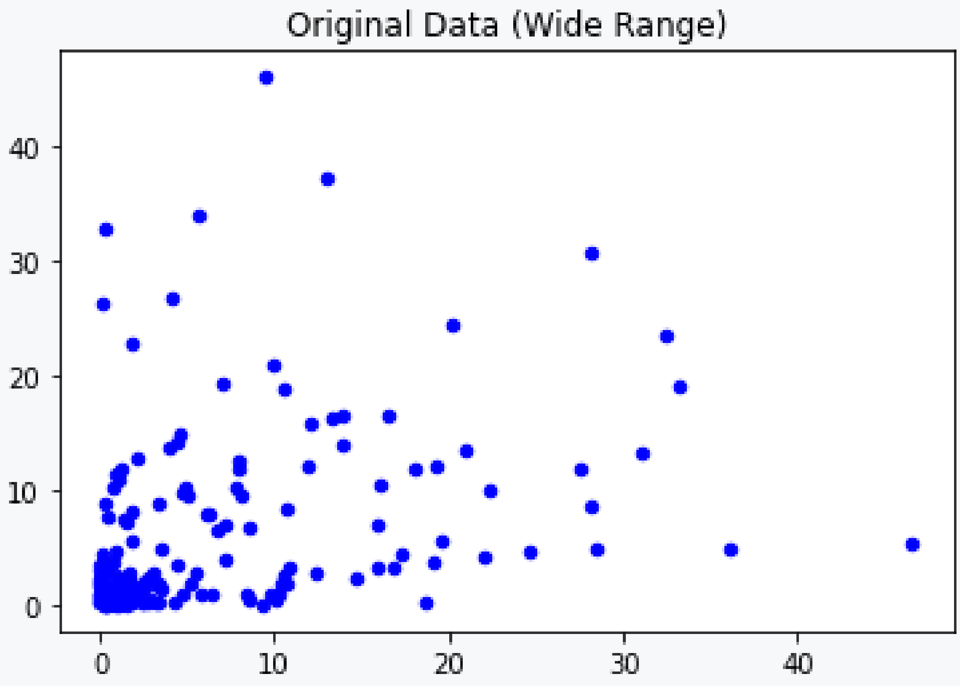
- It corrects the disproportionate impact of outliers, ensuring that the clustering algorithm focuses on meaningful patterns.
- By reducing the data range, it helps clustering algorithms form compact and distinct clusters.
- Log-transformed data often aligns better with the assumptions of statistical or machine learning models.

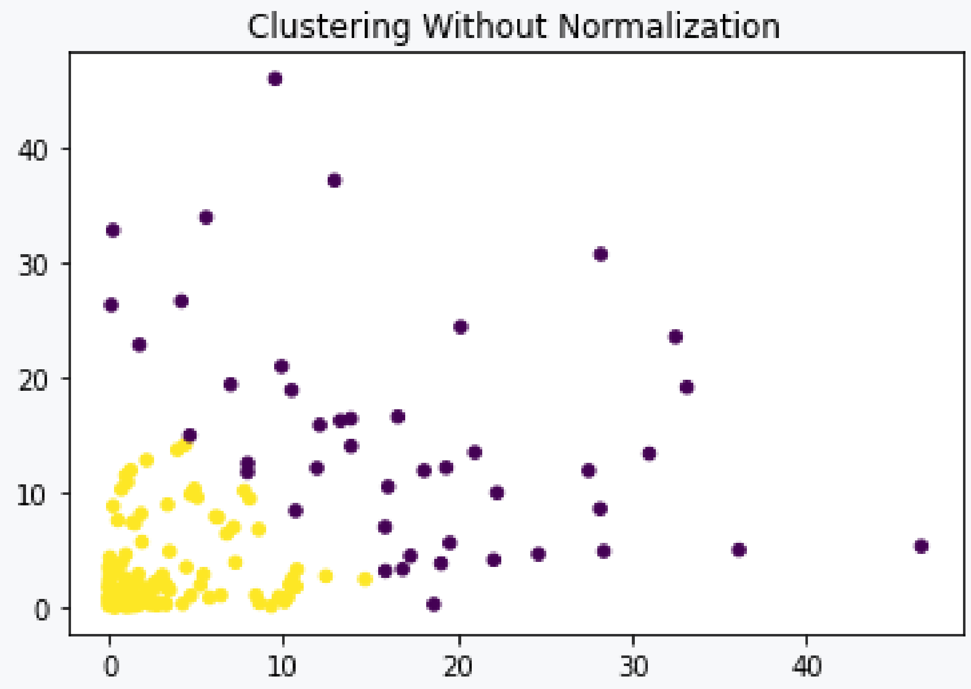
Consider a dataset with features that differ greatly in magnitude. Without normalization, smaller-scale features might get overshadowed by larger ones, leading to poor clustering performance. Consider a dataset with income ranging from $10,000 to $1,000,000. Without normalization, clustering may fail to recognize patterns among lower-income data. After applying logarithmic normalization, clusters can form based on proportional differences rather than absolute values.

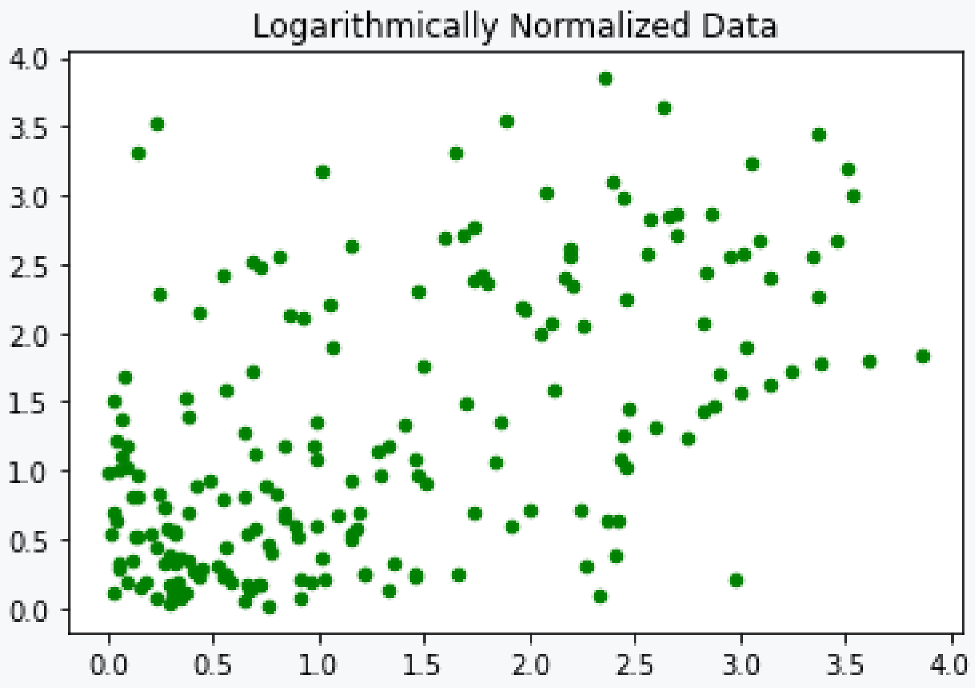
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import FunctionTransformer

# Generate data
np.random.seed(42)
X = np.concatenate([
    np.random.exponential(scale=1.0, size=(100, 2)),  #
Cluster 1
    np.random.exponential(scale=10.0, size=(100, 2))  #
Cluster 2
])

# Visualize original data
plt.scatter(X[:, 0], X[:, 1], s=20, color='blue')
plt.title("Original Data (Wide Range)")
plt.show()

# Clustering without normalization
kmeans_no_norm = KMeans(n_clusters=2, random_state=42).fit(X)
plt.scatter(X[:, 0], X[:, 1], c=kmeans_no_norm.labels_,
cmap='viridis', s=20)
plt.title("Clustering Without Normalization")
plt.show()

# Logarithmic normalization
```

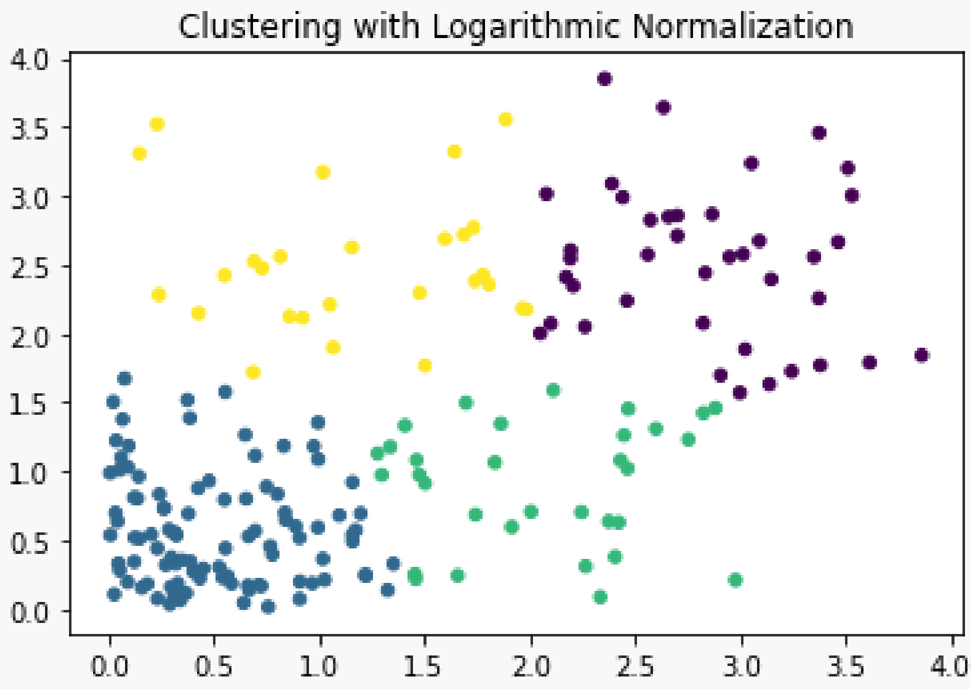
```
log_transformer = FunctionTransformer(np.log1p, validate=True)
X_log = log_transformer.transform(X)

# Visualize log-transformed data
plt.scatter(X_log[:, 0], X_log[:, 1], s=20, color='green')
plt.title("Logarithmically Normalized Data")
plt.show()

# Clustering with normalization
kmeans_log_norm = KMeans(n_clusters=4,
random_state=42).fit(X_log)
plt.scatter(X_log[:, 0], X_log[:, 1],
c=kmeans_log_norm.labels_, cmap='viridis', s=20)
plt.title("Clustering with Logarithmic Normalization")
plt.show()
```

Program output:




Clustering Without Normalization
40
30
20
10
0
0
10
20
30
40


Logarithmically Normalized Data
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0



9.2.12

What is the primary purpose of logarithmic normalization in clustering?

- To handle skewed data
- To increase data range
- To remove all outliers
- To split data into predefined clusters

9.2.13

Which of the following are true about logarithmic normalization?

- It compresses wide-ranging data
- It is suitable for skewed distributions
- It removes noise completely
- It works only with integer data

9.3 Cluster quality

9.3.1

Elbow method

When performing clustering on a dataset, one of the most critical questions is: "How many clusters should we choose?" Choosing the optimal number of clusters is essential because the wrong choice can lead to poor model performance and unreliable insights. A popular and simple method for identifying the ideal number of clusters is the **Elbow method**. This method is visual and relies on plotting the sum of squared distances (inertia) from each point to its assigned cluster center for various numbers of clusters.

To apply the Elbow method, we begin by fitting the clustering algorithm (e.g., K-means) with a range of cluster numbers, such as from 1 to 10. After fitting the algorithm for each cluster number, we calculate the **inertia** (also called the within-cluster sum of squares). Inertia measures how tightly the data points are clustered around the center. The smaller the inertia, the better the data points fit within their respective clusters.

The key to the Elbow method lies in the plot of inertia against the number of clusters. As the number of clusters increases, the inertia decreases because more clusters generally result in better fits. However, after a certain point, adding more clusters results in only marginal improvements in inertia. The "elbow" of the plot is the point where the rate of decrease slows down. This elbow indicates the ideal number of clusters, as beyond this point, adding more clusters does not significantly improve the clustering.

For example, if we plot inertia for cluster counts from 1 to 10 and notice a sharp drop from 1 to 4 clusters, followed by a slow and steady decrease, the "elbow" is at 4. Thus, 4 would be the optimal number of clusters for that dataset. This method is widely used because of its simplicity, although it may not always work perfectly in more complex datasets.

Example

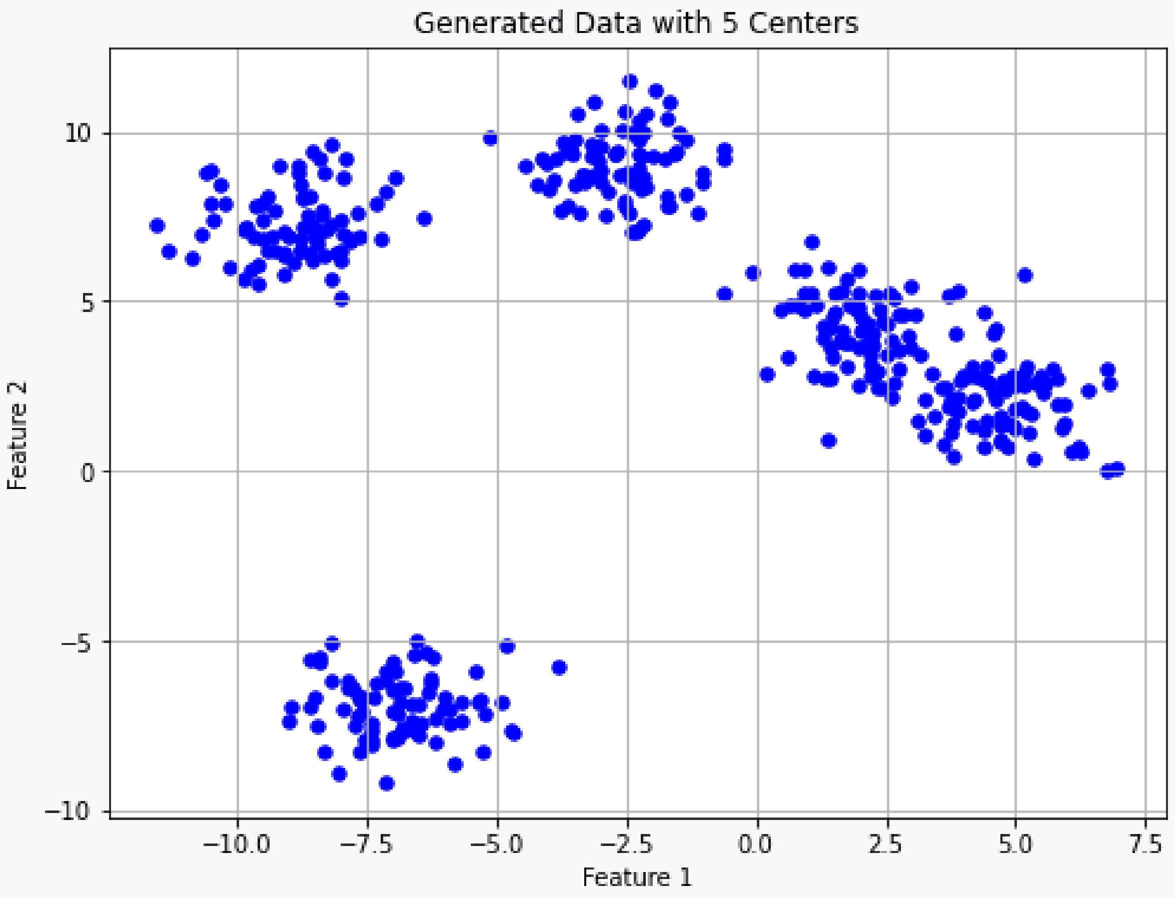
For different values of **k** (from 1 to 10), K-means is run on the data, and the inertia (sum of squared distances between data points and their assigned cluster centers) is recorded.

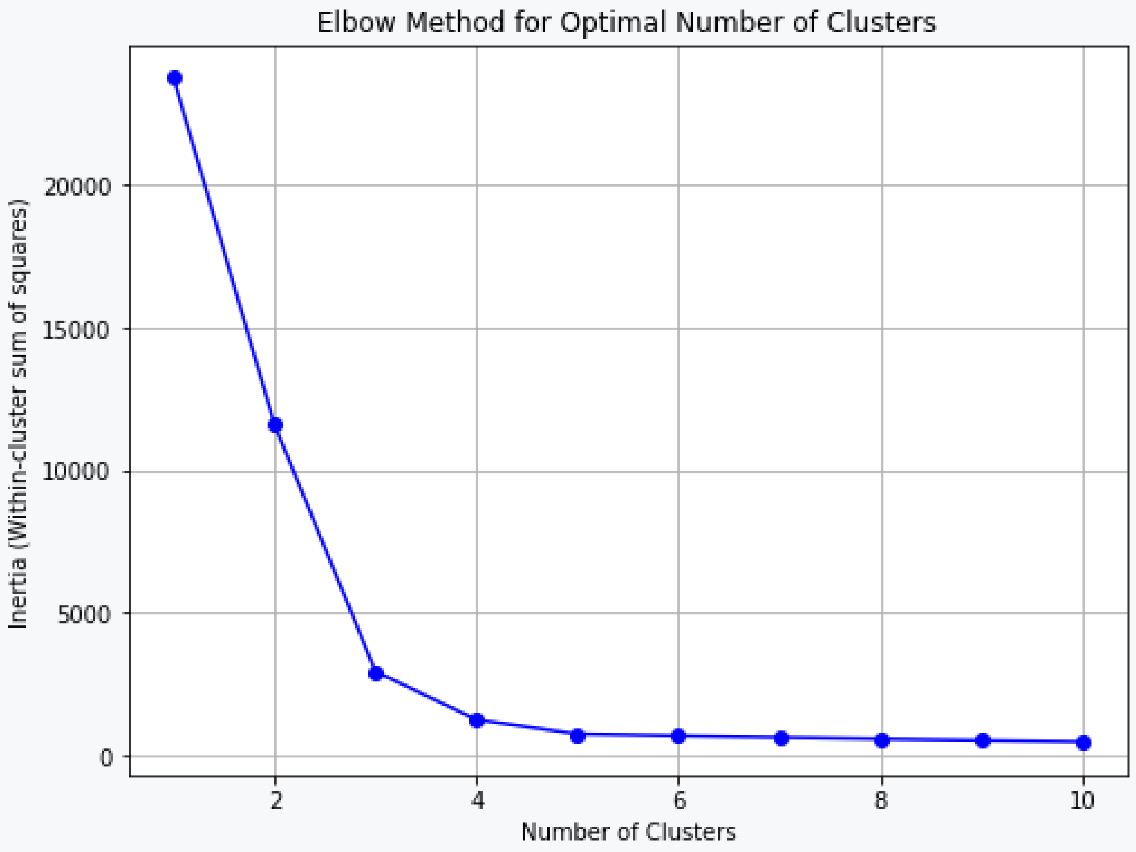
The inertia values are plotted against the number of clusters, and the "elbow" can be visually inspected to determine the optimal number of clusters.

The plot will display a curve showing the inertia for different numbers of clusters. The "elbow" point, where the decrease in inertia slows down, suggests the optimal number of clusters. For the synthetic data with 5 centers, this will likely be around k=5, but sometimes are other values correct (in this case it is 3 or 4).

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Generate random data with 5 centers (clusters)
X, _ = make_blobs(n_samples=400, centers=5, random_state=42)

# Plot the generated data
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], s=30, color='b', label='Data
points')
plt.title('Generated Data with 5 Centers')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()

# Use the elbow method to find the optimal number of clusters
inertia = []

# Try different numbers of clusters from 1 to 10
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X)
    inertia.append(kmeans.inertia_)

# Plot the elbow method graph
plt.figure(figsize=(8, 6))
plt.plot(range(1, 11), inertia, marker='o', color='b')
plt.title('Elbow Method for Optimal Number of Clusters')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia (Within-cluster sum of squares)')
plt.grid(True)
plt.show()
```

Program output:





9.3.2

What does the "elbow" in the Elbow method plot represent?

- The point where the rate of decrease in inertia slows down
- The point where inertia starts to increase rapidly
- The point where the number of clusters reaches its maximum
- The point where the clusters are not distinguishable

9.3.3

Which of the following are true about the Elbow method?

- It helps identify the optimal number of clusters by plotting inertia against cluster numbers
- The "elbow" in the plot indicates the ideal number of clusters
- It requires calculating the Silhouette score for each cluster count
- Adding more clusters after the elbow significantly improves the model's performance

9.3.4

Silhouette score

Another effective method for determining the optimal number of clusters is by using the **Silhouette score**. Unlike the Elbow method, which only measures the within-cluster distance, the Silhouette score takes both cohesion and separation into account. The cohesion measures how close the data points in a cluster are to one another, while the separation measures how distinct a cluster is from others. A high Silhouette score means that the clusters are well-separated and tightly packed, indicating a good clustering result.

The Silhouette score is calculated for each data point, and its overall value is the average of all individual scores. It ranges from -1 to 1, where:

- A score close to **1** indicates that the data points are well-clustered, with good separation from other clusters.
- A score close to **0** indicates that the data points are on or near the decision boundary between clusters.
- A score close to **-1** suggests that the data points might have been assigned to the wrong cluster.

To use the Silhouette Score to determine the best number of clusters, we compute the score for a range of cluster numbers (e.g., from 2 to 10 clusters). After plotting the scores for each cluster number, we select the number of clusters that maximizes the Silhouette score. This method is valuable because it not only considers how well the data fits the chosen clusters but also ensures that clusters are well-separated.

For example, if we test different cluster numbers and find that the Silhouette score peaks at 3 clusters, we conclude that 3 is the best choice for the dataset. This method is particularly useful when the dataset has well-defined clusters and can be applied to various clustering algorithms like K-means, DBSCAN, or hierarchical clustering.

By using it to compare different cluster counts, we can make more informed decisions about the optimal number of clusters for our data.

Example

The silhouette score is calculated for different numbers of clusters (from 2 to 10). The silhouette score measures how well each point fits within its cluster, with higher values indicating better-defined clusters.

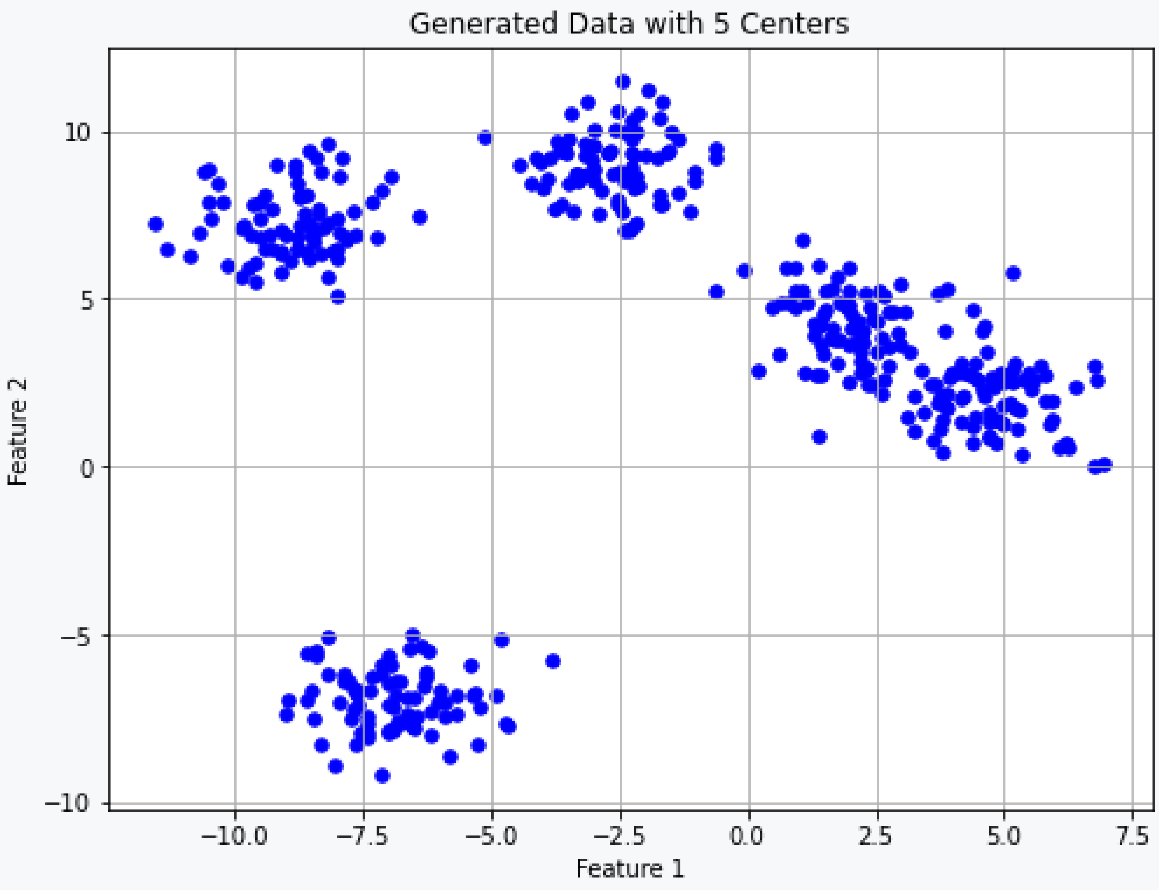
The best number of clusters is determined by selecting the k with the highest silhouette score - we can print values or visualise differences between various numbers of clusters.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.metrics import silhouette_score,
silhouette_samples

# Generate random data with 5 centers (clusters)
X, _ = make_blobs(n_samples=400, centers=5, random_state=42)

# Plot the generated data
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], s=30, color='b', label='Data
points')
plt.title('Generated Data with 5 Centers')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.show()
```

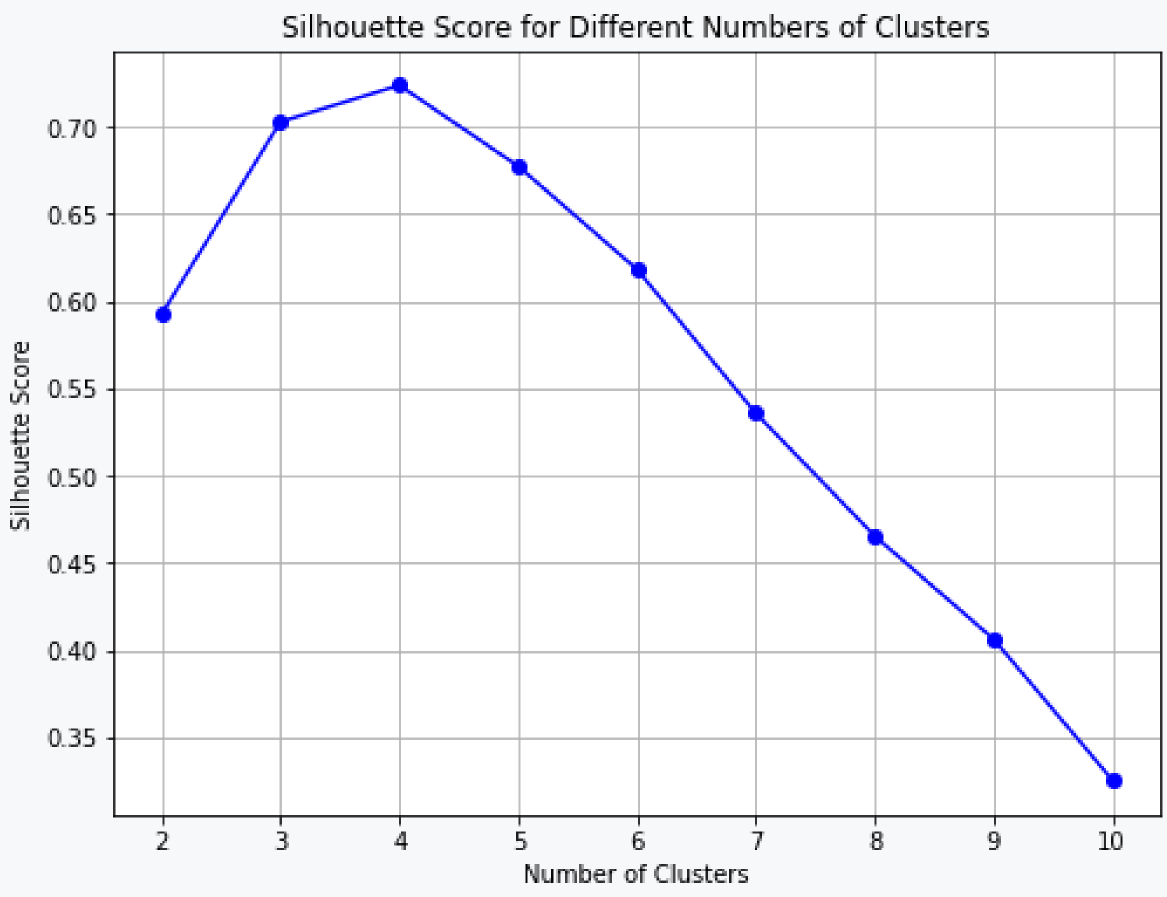
```
# Initialize variables to track the best silhouette score and
number of clusters
best_silhouette_score = -1
best_n_clusters = 0
silhouette_scores = []

# Try different numbers of clusters from 2 to 10 (silhouette
score is not defined for 1 cluster)
for k in range(2, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X)

    # Calculate silhouette score for the current k
    score = silhouette_score(X, kmeans.labels_)
    silhouette_scores.append(score)

    # Track the best silhouette score and the corresponding
number of clusters
    if score > best_silhouette_score:
        best_silhouette_score = score
        best_n_clusters = k

# Print the best number of clusters and corresponding
silhouette score
print(f"Best number of clusters: {best_n_clusters}")
print(f"Best silhouette score: {best_silhouette_score:.4f}")

# Plot silhouette scores for each k
plt.figure(figsize=(8, 6))
plt.plot(range(2, 11), silhouette_scores, marker='o',
color='b')
plt.title('Silhouette Score for Different Numbers of
Clusters')
plt.xlabel('Number of Clusters')
plt.ylabel('Silhouette Score')
plt.grid(True)
plt.show()

# Visualize the clustering quality with the best number of
clusters
best_kmeans = KMeans(n_clusters=best_n_clusters,
random_state=42)
best_kmeans.fit(X)
```

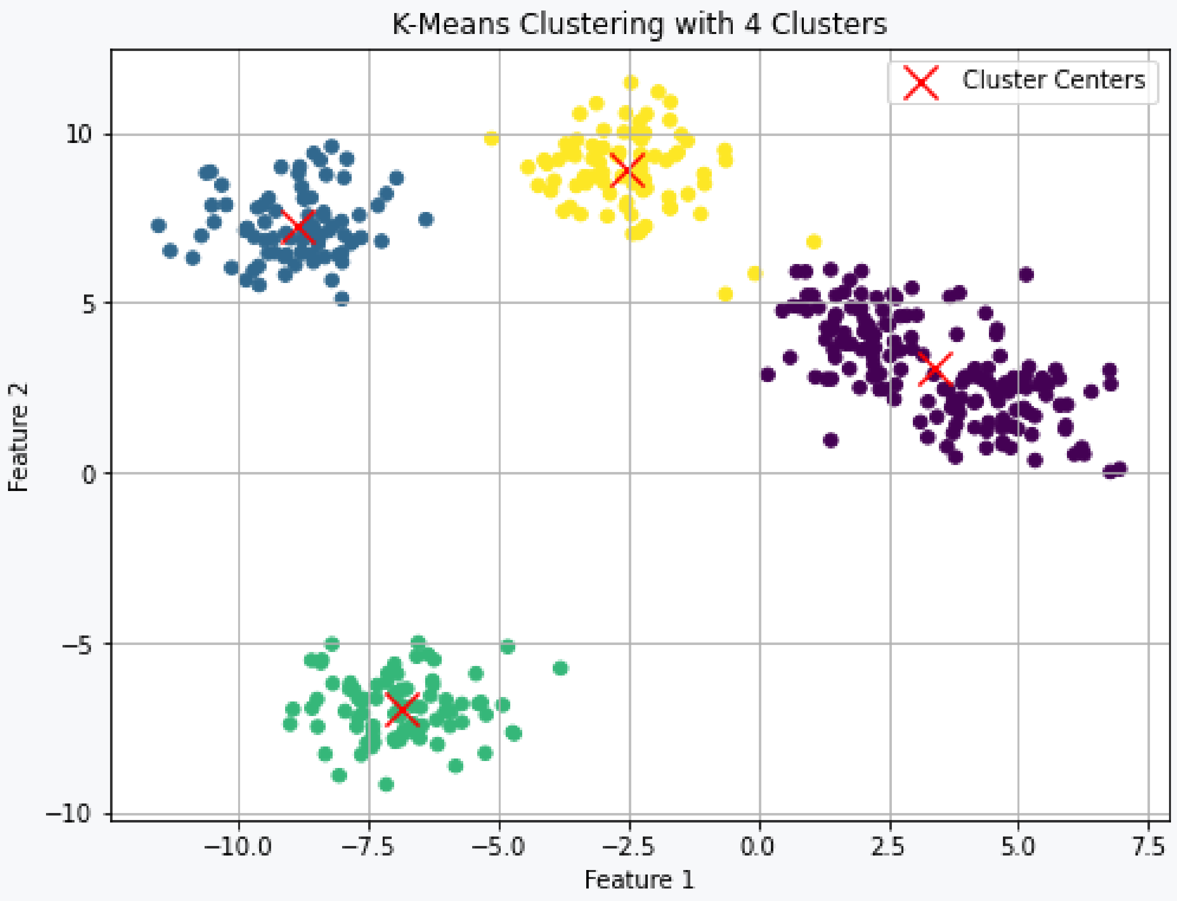
```
# Plot the data points with the cluster centers
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=best_kmeans.labels_,
cmap='viridis', s=30)
plt.scatter(best_kmeans.cluster_centers_[:, 0],
best_kmeans.cluster_centers_[:, 1],
            marker='x', color='red', s=200, label='Cluster
Centers')
plt.title(f'K-Means Clustering with {best_n_clusters}
Clusters')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.legend()
plt.show()
```

Program output:



```
Best number of clusters: 4
Best silhouette score: 0.7235
```


Silhouette Score for Different Numbers of Clusters
Silhouette Score
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
2
3
4
5
6
7
8
9
10
Number of Clusters


K-Means Clustering with 4 Clusters
Cluster Centers
Feature 2
10
5
0
−5
−10
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
Feature 1

The silhouette score plot showed the silhouette scores for different numbers of clusters. The optimal number of clusters corresponds to the peak in the silhouette score. The final plot will show the clustering results for the best number of clusters, with the cluster centers marked in red. The clusters should be well-separated if the silhouette score is high.

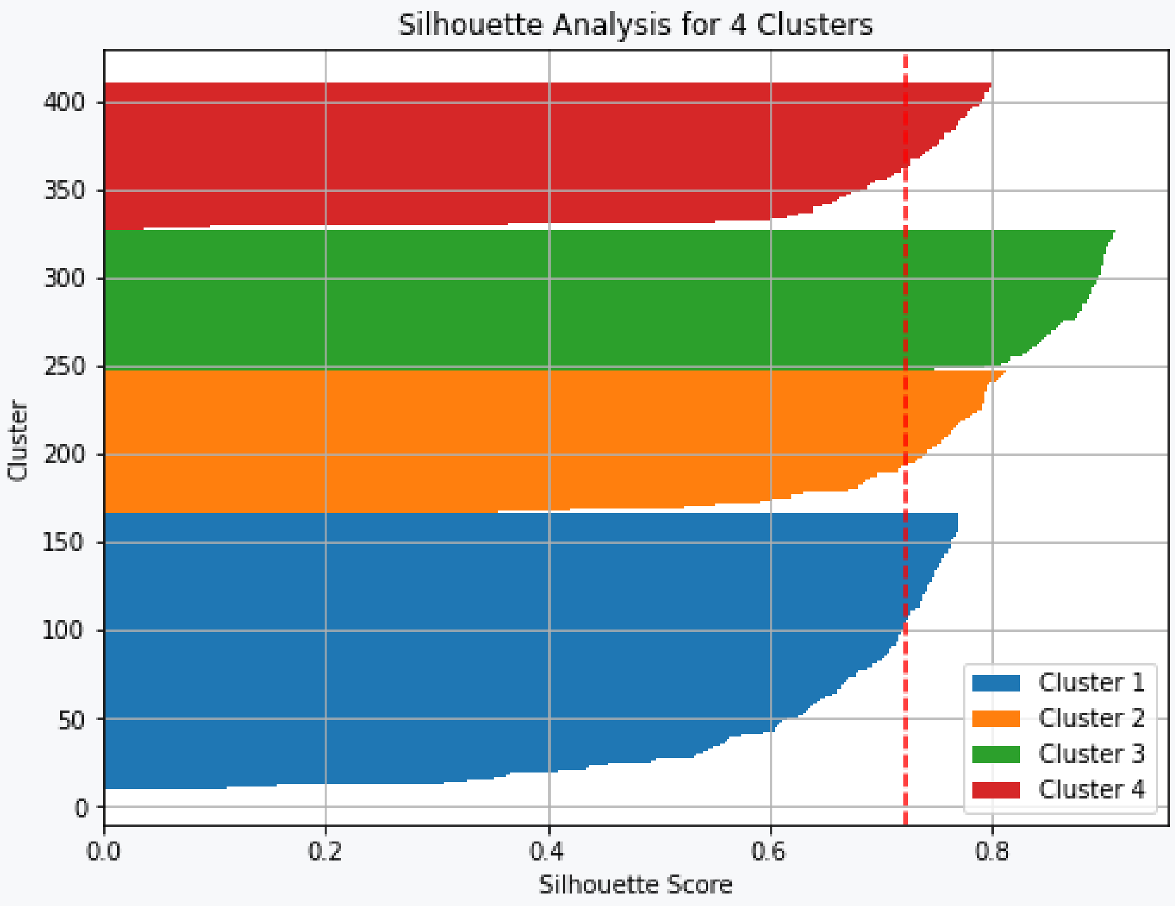
Silhouette analysis

Silhouette analysis for visualizing how well the data points fit within their clusters, uses the **silhouette_samples** function from **sklearn.metrics**. It provides a silhouette score for each individual data point, which can be plotted to visualize the quality of clustering and amount of elements for each point.

```
# Silhouette Analysis: plot silhouette scores for each sample
in the dataset
silhouette_avg = silhouette_score(X, best_kmeans.labels_)
sample_silhouette_values = silhouette_samples(X,
best_kmeans.labels_)

# Plot the silhouette analysis
plt.figure(figsize=(8, 6))
y_lower = 10
for i in range(best_n_clusters):
    # Aggregate the silhouette scores for samples belonging to
cluster i
    cluster_silhouette_values =
sample_silhouette_values[best_kmeans.labels_ == i]
    cluster_silhouette_values.sort()

    # Determine the size of the cluster
    size_cluster = cluster_silhouette_values.shape[0]

    # Plot the silhouette scores for the cluster
    plt.barh(range(y_lower, y_lower + size_cluster),
cluster_silhouette_values,
             height=1.0, edgecolor='none', label=f'Cluster
{i+1}')
    y_lower += size_cluster

# Add a vertical line for average silhouette score
plt.axvline(x=silhouette_avg, color="red", linestyle="--")

# Add labels and title
plt.title(f'Silhouette Analysis for {best_n_clusters}
Clusters')
```

```
plt.xlabel('Silhouette Score')
plt.ylabel('Cluster')
plt.grid(True)
plt.legend(loc='best')
plt.show()
```

Program output:

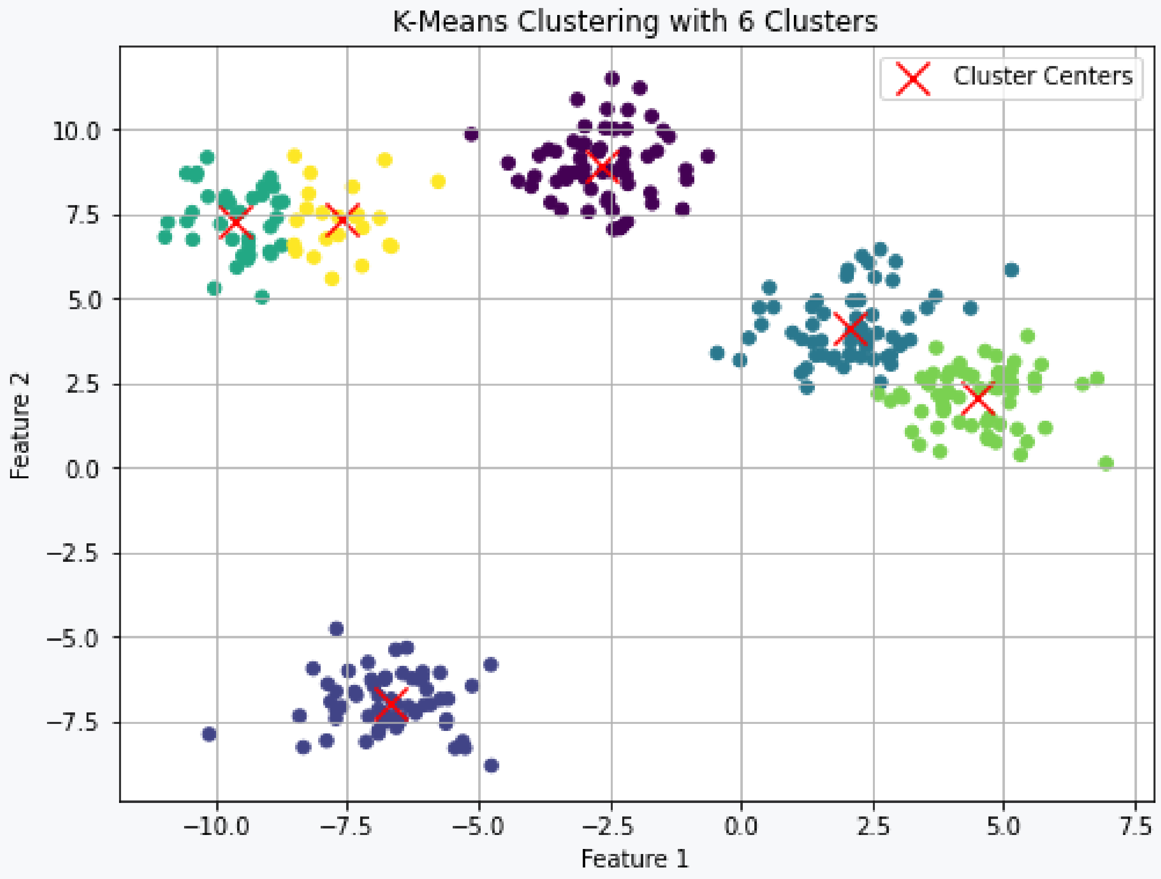


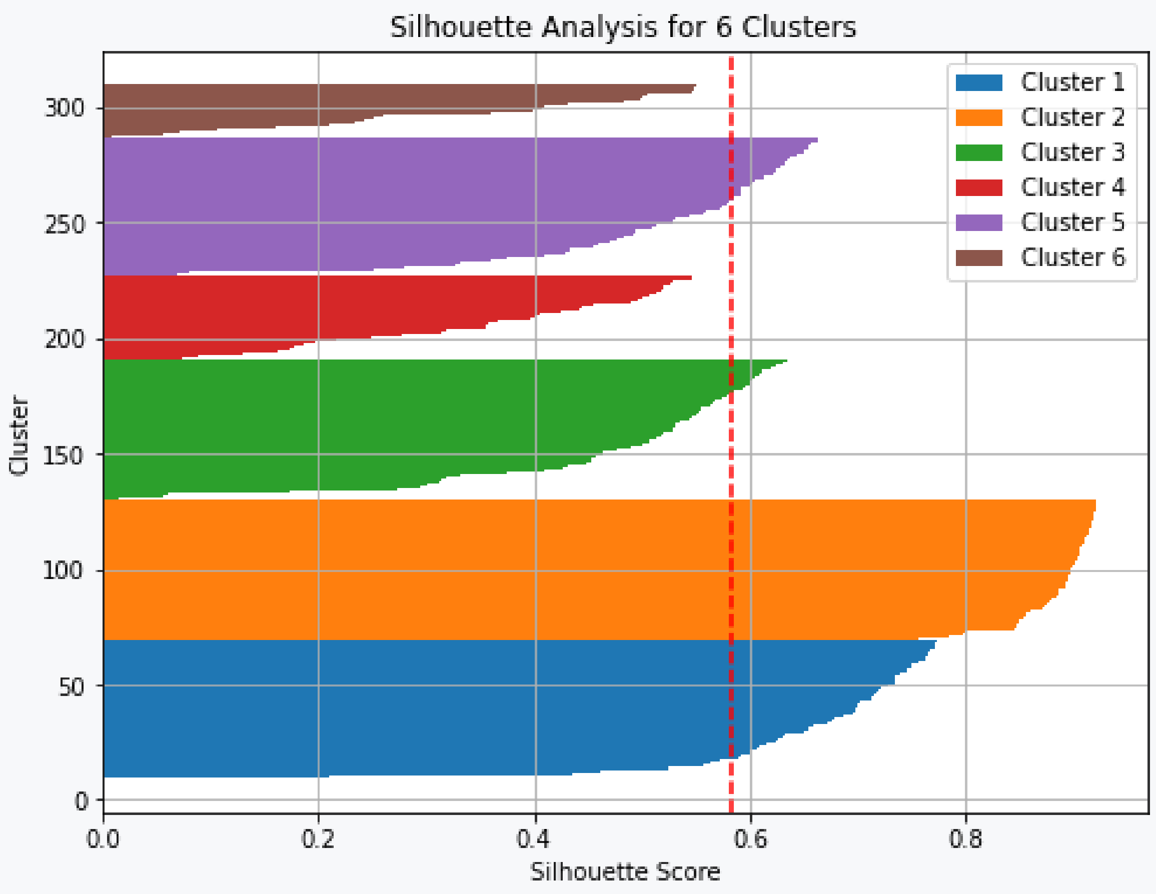
For experimentation with other number of clusters you can use following code:

```
other_n_clusters = 6

# Visualize the clustering quality with the best number of
clusters
other_kmeans = KMeans(n_clusters=other_n_clusters,
random_state=42)
other_kmeans.fit(X)

# Plot the data points with the cluster centers
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=other_kmeans.labels_,
cmap='viridis', s=30)
plt.scatter(other_kmeans.cluster_centers_[:, 0],
other_kmeans.cluster_centers_[:, 1],
            marker='x', color='red', s=200, label='Cluster
Centers')
plt.title(f'K-Means Clustering with {other_n_clusters}
Clusters')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True)
plt.legend()
plt.show()

# Silhouette Analysis: plot silhouette scores for each sample
in the dataset
silhouette_avg = silhouette_score(X, other_kmeans.labels_)
sample_silhouette_values = silhouette_samples(X,
other_kmeans.labels_)

# Plot the silhouette analysis
plt.figure(figsize=(8, 6))
y_lower = 10
for i in range(other_n_clusters):
    # Aggregate the silhouette scores for samples belonging to
cluster i
    cluster_silhouette_values =
sample_silhouette_values[other_kmeans.labels_ == i]
    cluster_silhouette_values.sort()

    # Determine the size of the cluster
    size_cluster = cluster_silhouette_values.shape[0]

    # Plot the silhouette scores for the cluster
    plt.barh(range(y_lower, y_lower + size_cluster),
cluster_silhouette_values,
             height=1.0, edgecolor='none', label=f'Cluster
{i+1}')
    y_lower += size_cluster

# Add a vertical line for average silhouette score
plt.axvline(x=silhouette_avg, color="red", linestyle="--")

# Add labels and title
```

```
plt.title(f'Silhouette Analysis for {other_n_clusters}
Clusters')
plt.xlabel('Silhouette Score')
plt.ylabel('Cluster')
plt.grid(True)
plt.legend(loc='best')
plt.show()
```

Program output:





9.3.5

What does a Silhouette score close to 1 indicate?

- The clusters are well-separated and tightly packed
- The clusters are poorly separated
- The data points are close to the decision boundary
- The clusters overlap significantly

9.3.6

Which statements are true about the Silhouette score?

- A Silhouette score close to 0 suggests that the data points are on the boundary between clusters
- The Silhouette score measures both cohesion and separation of clusters
- The Silhouette score ranges from 0 to 10
- A Silhouette score close to -1 indicates that the data points are well-clustered

9.4 Clustering algorithm types

📖 9.4.1

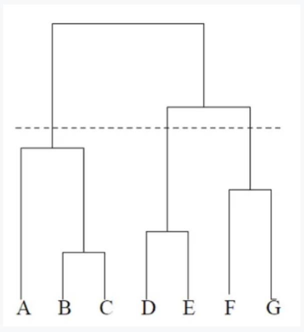
Hierarchical clustering

Hierarchical clustering is a method that builds a hierarchy of clusters. This method works by recursively merging or splitting clusters. The result is a tree-like structure, often represented as a dendrogram. This approach is particularly useful when we need to understand the relationships between clusters and have no prior knowledge of how many clusters to expect. For example, in biology, hierarchical clustering can be used to understand the relationship between different species based on their characteristics.

There are two main approaches in hierarchical clustering:

- Agglomerative hierarchical clustering starts by treating each data point as its own cluster and then successively merges the closest clusters.
- Divisive hierarchical clustering works in the opposite direction, starting with one large cluster and iteratively splitting it into smaller clusters.

To build the hierarchy, the similarity between clusters is computed at each step. This similarity can be measured using various metrics like Euclidean distance or cosine similarity. The hierarchical process is represented as a dendrogram, where the leaves represent the individual data points and the nodes represent the merged clusters. The height of the branches in the dendrogram indicates the similarity between clusters. A cut can be made at any level to choose the desired number of clusters.



Hierarchical clustering is useful when we need a comprehensive view of how data points are related to one another. It is particularly helpful when the number of clusters is not known in advance or when we want to understand the hierarchy of data. For example, in market segmentation, hierarchical clustering can reveal the hierarchy of consumer preferences.

9.4.2

Which of the following is true about hierarchical clustering?

- It creates a tree-like structure called a dendrogram.
- It requires the number of clusters to be defined in advance.
- It always produces a flat clustering solution.
- It splits clusters starting from individual data points.

9.4.3

Hierarchical agglomerative clustering

Hierarchical agglomerative clustering is the bottom-up approach, is a clustering method that doesn't require specifying the number of clusters in advance. It starts by treating each data point as a singleton cluster and repeatedly merges the closest clusters until all data points belong to a single cluster. This approach provides a more informative structure than the unstructured set of clusters from flat clustering.

Algorithm steps:

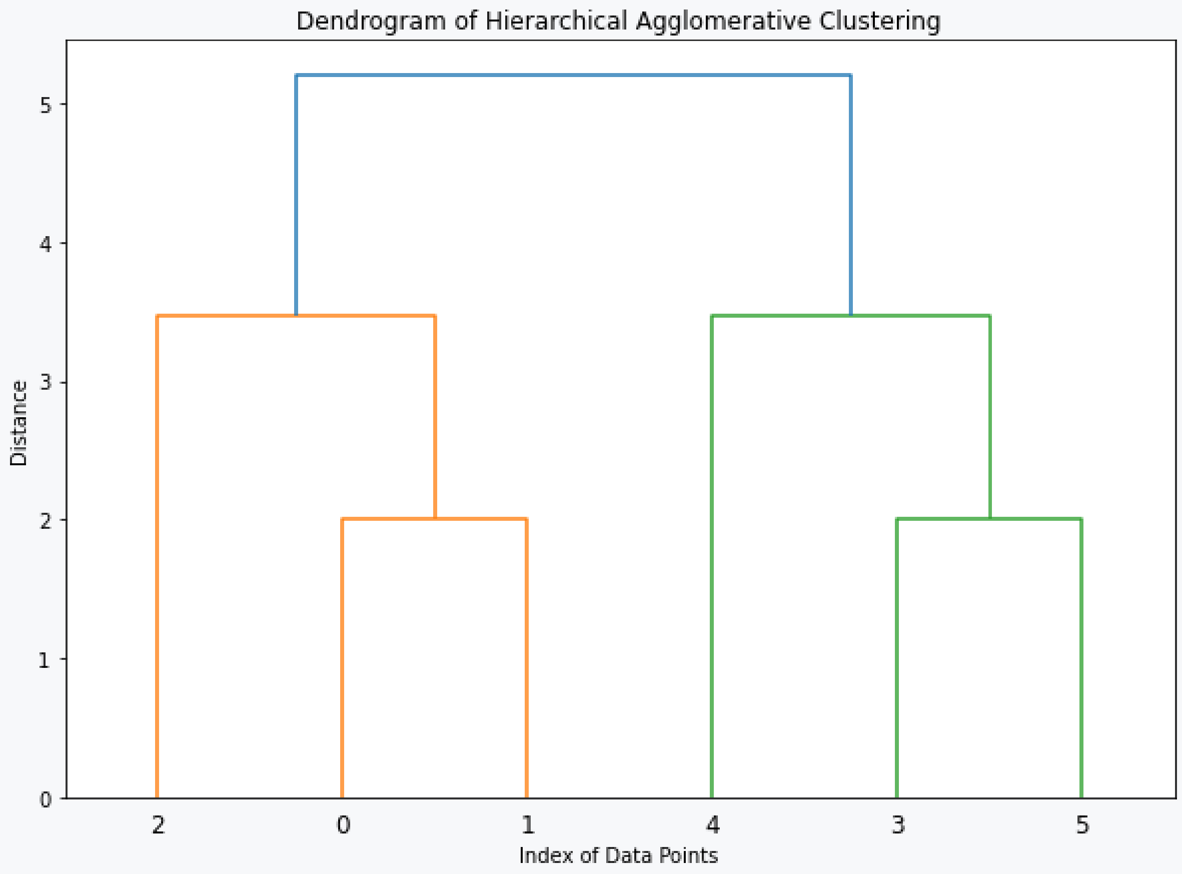
1. Start with a dataset (d_1, d_2, ..., d_n) of size n.
2. Compute the distance matrix: for each pair of points (d_i, d_j), compute their distance (distance[d_i, d_j]). As the matrix is symmetric, calculate only the lower half.
3. Initially, treat each data point as a singleton cluster.
4. Repeatedly merge the two closest clusters and update the distance matrix.
5. Continue merging until only one cluster remains.

This process forms a hierarchy that can be visualized as a dendrogram, providing insight into the data's structure.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage

# Randomly chosen dataset
X = np.array([[1, 2], [1, 4], [1, 0],
              [4, 2], [4, 4], [4, 0]])

# Perform hierarchical/agglomerative clustering
```

```
clustering = AgglomerativeClustering(n_clusters=None,
distance_threshold=0)
clustering.fit(X)

# Create a linkage matrix using scipy
Z = linkage(X, 'ward')

# Plot the dendrogram to visualize the clustering steps
plt.figure(figsize=(10, 7))
dendrogram(Z)
plt.title("Dendrogram of Hierarchical Agglomerative
Clustering")
plt.xlabel("Index of Data Points")
plt.ylabel("Distance")
plt.show()

# Optionally print the class labels
print(clustering.labels_)
```

Program output:



```
[5 4 3 1 2 0]
```

In the previous code using AgglomerativeClustering from scikit-learn, the algorithm uses Euclidean distance by default, and the complete linkage method is typically used for merging clusters unless specified otherwise. Therefore, the similarity (or distance) is based on the Euclidean distance between the clusters, and the algorithm merges the closest clusters iteratively.

9.4.4

Hierarchical divisive clustering

Hierarchical divisive clustering is the top-down approach, is a clustering algorithm that recursively divides a large cluster into smaller sub-clusters. Unlike agglomerative clustering, which starts with individual data points and merges them, divisive clustering begins with the entire dataset as one large cluster and splits it iteratively into smaller clusters until every data point is isolated in its own singleton cluster.

Algorithm:

1. Start with one cluster containing all data points.
2. Split the cluster based on some criteria to create two smaller clusters.
3. Recursively split the resulting clusters until all the data points are in individual clusters.

This algorithm does not require you to specify the number of clusters, as it continues to divide the data until each point is isolated. The number of final clusters depends on the stopping criterion used, such as a maximum depth of splits or a minimum cluster size.

To divide clusters, we need to determine how to split them. The key challenge here is defining a **distance measure** between clusters. Different methods of measuring inter-cluster distance will result in different ways of dividing the data. Some common distance metrics include:

1. **Min distance (single linkage)** is defined as the minimum distance between any pair of points, one from each cluster. This method tends to produce long, thin clusters, as it merges clusters based on the closest pair of points.
2. **Max distance (complete linkage)** is the maximum distance between any two points, one from each cluster. This method results in more compact clusters, as it considers the farthest points in each cluster.
3. **Group average (average linkage)** is the average distance between all pairs of points, one from each cluster. This method is a compromise between single and complete linkage, creating clusters that are balanced in size and density.
4. **Ward's method** is based on minimizing the increase in squared error when two clusters are merged. It prioritizes merging clusters that lead to the

smallest increase in variance and tends to create more compact and spherical clusters.

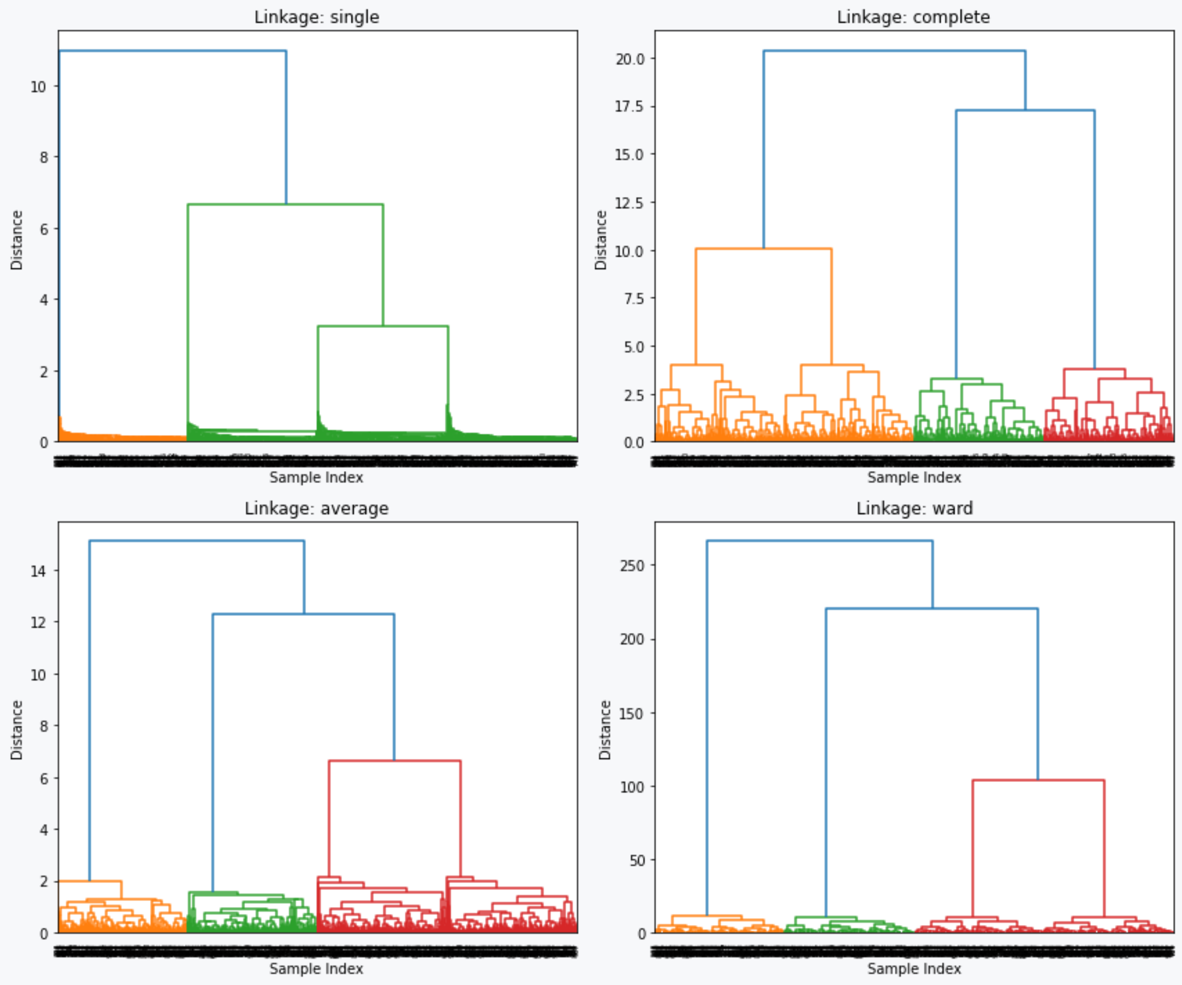
Each of these methods can affect the final clustering result. For instance, the Min distance method might produce elongated clusters, while Ward’s method creates more compact clusters.

To compare the results of the four distance metrics mentioned (Min distance, Max distance, Group average, and Ward's method) in hierarchical divisive clustering, we'll use **AgglomerativeClustering** from scikit-learn with different linkage methods. I'll generate a synthetic dataset with enough size and complexity to show the differences in clustering

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from scipy.cluster.hierarchy import dendrogram, linkage

# Create a synthetic dataset with 1000 samples and 2 features
X, _ = make_blobs(n_samples=1000, centers=4, cluster_std=0.60,
random_state=42)

# Function to perform hierarchical clustering and plot the
dendrogram
def plot_dendrogram(X, linkage_methods):
    plt.figure(figsize=(12, 10))

    for i, linkage_method in enumerate(linkage_methods):
        # Perform linkage for hierarchical clustering
        Z = linkage(X, method=linkage_method)

        plt.subplot(2, 2, i + 1)
        dendrogram(Z)
        plt.title(f"Linkage: {linkage_method}")
        plt.xlabel("Sample Index")
        plt.ylabel("Distance")

    plt.tight_layout()
    plt.show()

# List of different linkage methods
linkage_methods = ['single', 'complete', 'average', 'ward']
```

```
# Plot dendrograms for all the methods
plot_dendrogram(X, linkage_methods)
```

Program output:



9.4.5

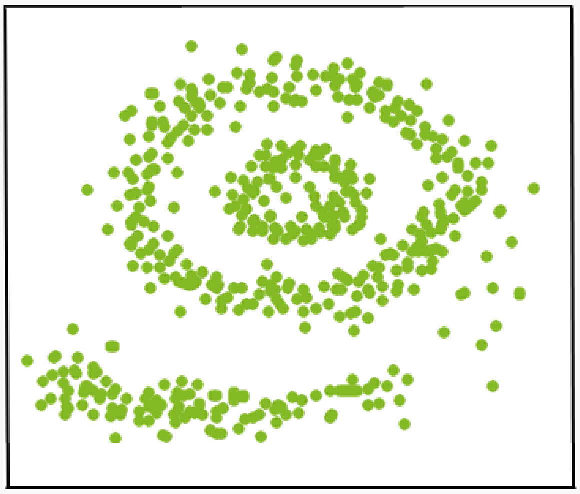
Which of the following distance metrics will likely create more compact clusters in Divisive hierarchical clustering?

- Ward’s method
- Min distance
- Max distance
- Group average

9.4.6

DBSCAN

DBSCAN is a density-based clustering algorithm that groups together closely packed points and labels points in low-density regions as noise. Partitioning methods (like K-means) and hierarchical clustering work to find spherical clusters or convex clusters.They are only suitable for compact and well-separated clusters. In addition, they are also seriously affected by the presence of noise and outliers in the data. DBSCAN is widely used in tasks like spatial data analysis, anomaly detection, and environmental monitoring.



- source: https://www.geeksforgeeks.org/dbscan-clustering-in-ml-density-based-clustering/

The figure above shows a data set containing non-convex shape clusters and outliers. Given such data, the k-means algorithm has difficulties in identifying these clusters with arbitrary shapes.

DBSCAN requires two parameters:

- **eps** defines the neighborhood around a data point. If the distance between two points is less than or equal to ‘eps’, they are considered neighbors. A small **eps** value may label many points as outliers, while a large **eps** value may cause clusters to merge, resulting in most points being grouped together. One way to determine an appropriate **eps** value is by using the k-distance graph.
- **MinPts** is the minimum number of neighbors (data points) required within the **eps** radius. Larger datasets typically require a higher **MinPts** value. As a general rule, **MinPts** should be at least **D+1**, where **D** is the number of dimensions in the dataset. The minimum **MinPts** value should be 3 or higher.

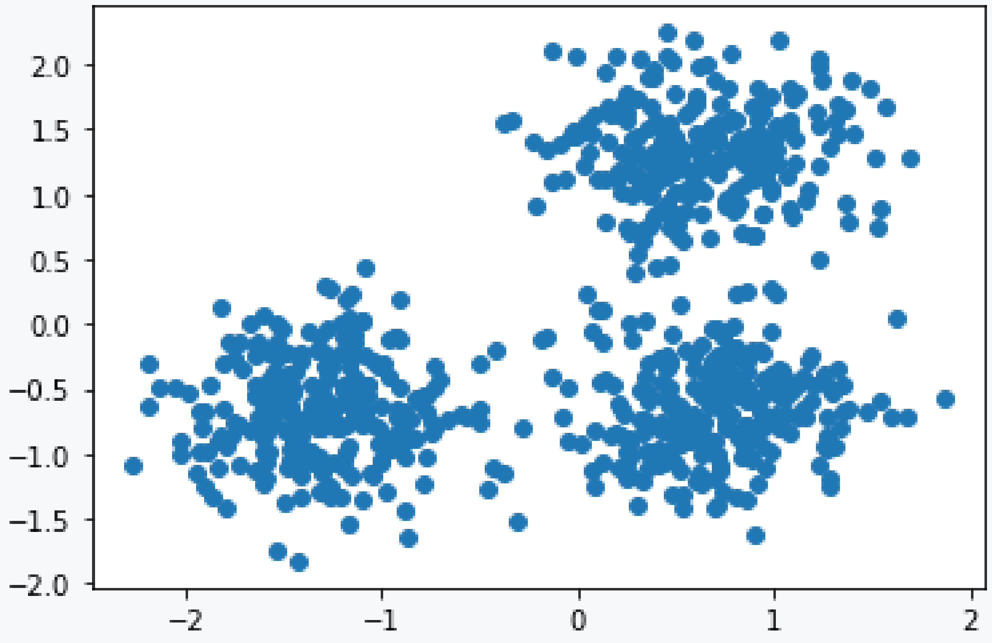
The algorithm starts with an arbitrary point and checks if it has enough neighbours within ε. If it does, a new cluster is created, and the process continues with the neighbours. Points that do not meet the density requirement are considered noise.

DBSCAN is best used when clusters have irregular shapes or varying densities. It is especially useful when you have data with noise or outliers, such as geographical data points, where some regions are densely populated, and others are sparse.

We will generate dataset with noise values (https://scikit-learn.org/1.5/auto_examples/cluster/plot_dbscan.html):

```
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler

centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(
    n_samples=750, centers=centers, cluster_std=0.4,
random_state=0
)

X = StandardScaler().fit_transform(X)
import matplotlib.pyplot as plt

plt.scatter(X[:, 0], X[:, 1])
plt.show()
```

Program output:



In DBSCAN, the **labels_** attribute stores the cluster labels assigned to each data point. Each data point is assigned a specific label indicating which cluster it belongs to.

- Points that are part of a cluster are assigned a unique label (e.g., 0, 1, 2, etc.).
- Points that do not belong to any cluster, often due to being too far from other points (i.e., outliers), are given the label **-1**.

The **labels_** attribute allows you to see which points belong to clusters and which are considered noise or outliers.

```
import numpy as np

from sklearn import metrics
from sklearn.cluster import DBSCAN

# Apply DBSCAN algorithm
db = DBSCAN(eps=0.3, min_samples=10).fit(X)

# Access the labels assigned to each data point
labels = db.labels_

# Number of clusters (excluding noise labeled as -1)
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)

# Number of noise points (those labeled as -1)
n_noise_ = list(labels).count(-1)

# Print the estimated number of clusters and noise points
print("Estimated number of clusters: %d" % n_clusters_)
print("Estimated number of noise points: %d" % n_noise_)
```

Program output:

```
Estimated number of clusters: 3
Estimated number of noise points: 18
```

```
print(f"Homogeneity: {metrics.homogeneity_score(labels_true,
labels):.3f}")
print(f"Completeness: {metrics.completeness_score(labels_true,
labels):.3f}")
print(f"V-measure: {metrics.v_measure_score(labels_true,
labels):.3f}")
```

```
print(f"Adjusted Rand Index:
{metrics.adjusted_rand_score(labels_true, labels):.3f}")
print(
    "Adjusted Mutual Information:"
    f" {metrics.adjusted_mutual_info_score(labels_true,
labels):.3f}"
)
print(f"Silhouette Coefficient: {metrics.silhouette_score(X,
labels):.3f}")
```

Program output:

```
Homogeneity: 0.953
Completeness: 0.883
V-measure: 0.917
Adjusted Rand Index: 0.952
Adjusted Mutual Information: 0.916
Silhouette Coefficient: 0.626
```

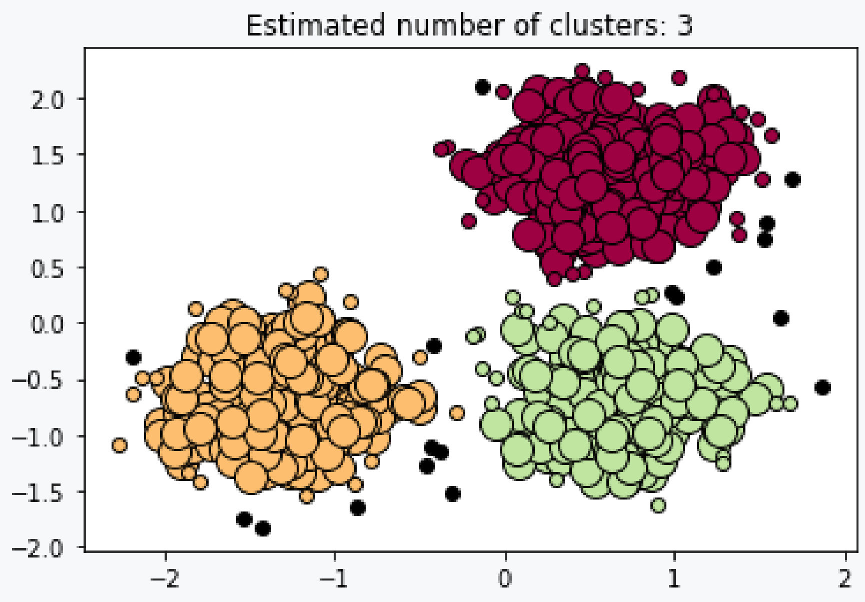
The data points are classified into three categories: core samples, non-core samples, and noise points. These categories help determine the structure of clusters. Here's an explanation of each type:

- Core samples are considered to be part of a dense region in the dataset. The algorithm expands clusters from these core points by including neighbouring points that meet the density criteria.
- Non-core samples are assigned to the cluster of the core sample that is nearest to them.
- Noise points is a data point that does not belong to any cluster. This is identified by DBSCAN if the point does not have enough neighbouring points and are labeled as -1 in the DBSCAN algorithm.

In the following visualization core samples are shown as large colored dots representing the cluster they belong to. Non-core samples are shown as small colored dots, also color-coded according to their assigned cluster. Noise points are shown as black dots, as they are not part of any cluster.

```
unique_labels = set(labels)
core_samples_mask = np.zeros_like(labels, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True

colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1,
len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = labels == k

    xy = X[class_member_mask & core_samples_mask]
    plt.plot(
        xy[:, 0],
        xy[:, 1],
        "o",
        markerfacecolor=tuple(col),
        markeredgecolor="k",
        markersize=14,
    )

    xy = X[class_member_mask & ~core_samples_mask]
    plt.plot(
        xy[:, 0],
        xy[:, 1],
        "o",
        markerfacecolor=tuple(col),
        markeredgecolor="k",
        markersize=6,
    )

plt.title(f"Estimated number of clusters: {n_clusters_}")
plt.show()
```

Program output:



9.4.7

What is DBSCAN primarily used for?

- Clustering in datasets with noise and outliers.
- Finding spherical clusters.
- Dividing data into equal-sized clusters.
- Defining the number of clusters in advance.

9.4.8

Gaussian mixture models

Gaussian mixture models (GMM) is a probabilistic model for clustering that assumes the data points are generated from a mixture of several Gaussian distributions. GMM provides a more flexible approach than K-means by considering the covariance of data points within each cluster. It is widely used in image processing, speech recognition, and anomaly detection.

GMM uses the expectation-maximization (EM) algorithm to estimate the parameters of the Gaussian distributions (mean, covariance, and weight). In the E-step, the probability that each point belongs to each cluster is computed, and in the M-step, the parameters of the Gaussians are updated based on the probabilities from the E-step.

Algorithm:

1. **Expectation (E-step)** calculate the probability of each data point belonging to each Gaussian distribution.
2. **Maximization (M-step)** update the parameters (mean, covariance, and weight) of each Gaussian distribution based on the current probabilities.
3. **Iterate** repeat the E-step and M-step until convergence.

GMM is useful when the clusters have different shapes and sizes and when you want a probabilistic interpretation of the clustering. It is ideal for problems where the data can be modeled by a mixture of normal distributions, such as in speech or image recognition.

9.4.9

Which of the following is true for Gaussian mixture models?

- They estimate the parameters of multiple Gaussian distributions.
- They assume clusters are spherical.
- They only work with flat data structures.
- They always produce hard clustering.

Resources

Chapter 10

10.1 Bibliography

🕮 10.1.1

Literature

- Dan Roth: Applied Machine Learning (CIS 519/419) - https://www.seas.upenn.edu/~cis5190/fall2020/assets/lectures/lecture-1/Lecture1-intro.pptx
- Emily Fox, Carlos Guestrin: Machine Learning Specialization, University of Washington https://www.coursera.org/specializations/machine-learning
- Eric Eaton: Introduction to Machine Learning (CIS 419/519) - https://www.seas.upenn.edu/~cis5190/fall2017/lectures/01_introduction.pdf
- Harikrishnan N B: Confusion Matrix, Accuracy, Precision, Recall, F1 Score - https://medium.com/analytics-vidhya/confusion-matrix-accuracy-precision-recall-f1-score-ade299cf63cd
- https://corporatefinanceinstitute.com/resources/data-science/regression-analysis/
- https://edu.ukf.sk/course/view.php?id=5334
- https://medium.com/@novus_afk/understanding-logistic-regression-a-beginners-guide-73f148866910
- https://medium.com/@satyarepala/understanding-logistic-regression-a-step-by-step-explanation-9a404344964b
- https://medium.com/analytics-vidhya/a-comprehensive-guide-to-logistic-regression-e0cf04fe738c
- https://medium.com/analytics-vidhya/understanding-logistic-regression-b3c672deac04
- https://medium.com/analytics-vidhya/understanding-the-linear-regression-808c1f6941c0
- https://medium.com/data-science-group-iitr/logistic-regression-simplified-9b4efe801389
- https://scikit-learn.org/1.5/auto_examples/cluster/plot_dbscan.html
- https://utsavdesai26.medium.com/linear-regression-made-simple-a-step-by-step-tutorial-fb8e737ea2d9
- https://www.analyticsvidhya.com/blog/2021/08/conceptual-understanding-of-logistic-regression-for-data-science-beginners/
- https://www.cuemath.com/data/least-squares/
- https://www.datacamp.com/tutorial/understanding-logistic-regression-python
- https://www.geeksforgeeks.org/dbscan-clustering-in-ml-density-based-clustering/
- https://www.geeksforgeeks.org/hierarchical-clustering/
- https://www.geeksforgeeks.org/k-means-clustering-introduction/
- https://www.kaggle.com/code/prashant111/logistic-regression-classifier-tutorial
- https://www.scribbr.com/statistics/simple-linear-regression/

- J. Skalka and M. Valko, "Rapid Guessing Behavior Detection in Microlearning: Insights Into Student Performance, Engagement, and Response Accuracy," in *IEEE Access*, vol. 12, pp. 157996-158024, 2024, https://doi.org/10.1109/ACCESS.2024.3485505
- Pavol Návrat et al: Artificial Intelligence. STU in Bratislava, 2002, Bratislava, 393 pages, ISBN 80-227-1645-6.
- Rotem Dror: Evaluation - https://www.seas.upenn.edu/~cis5190/fall2018/assets/lectures/lecture-3/03-eval.pptx
- S. Tahsildar - Gini Index For Decision Trees - https://blog.quantinsti.com/gini-index/
- Sruthi E R: Understanding Random Forest - https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/
- StatQuest: Decision and Classification Trees, Clearly Explained!! - https://www.youtube.com/watch?v=_L39rN6gz7Y
- StatQuest: Random Forests Part 1 - Building, Using and Evaluating - https://www.youtube.com/watch?v=J4Wdy0Wc_xQ
- StatQuest: Random Forests Part 2 - Missing data and clustering - https://www.youtube.com/watch?v=sQ870aTKqiM

10.1.2

Statement regarding the use of Artificial Intelligence in content creation

This content has been developed with the assistance of artificial intelligence tools, specifically ChatGPT, Gemini, and Notebook LM. These AI technologies were utilized to enhance the text by providing suggestions for rephrasing, improving clarity, and ensuring coherence throughout the material. The integration of these AI tools has enabled a more efficient content creation process while maintaining high standards of quality and accuracy.

The use of AI in this context adheres to all relevant guidelines and ethical considerations associated with the deployment of such technologies. We acknowledge the importance of transparency in the content creation process and aim to provide a clear understanding of how artificial intelligence has contributed to the final product.

PRISCILLA

priscilla.fitped.eu