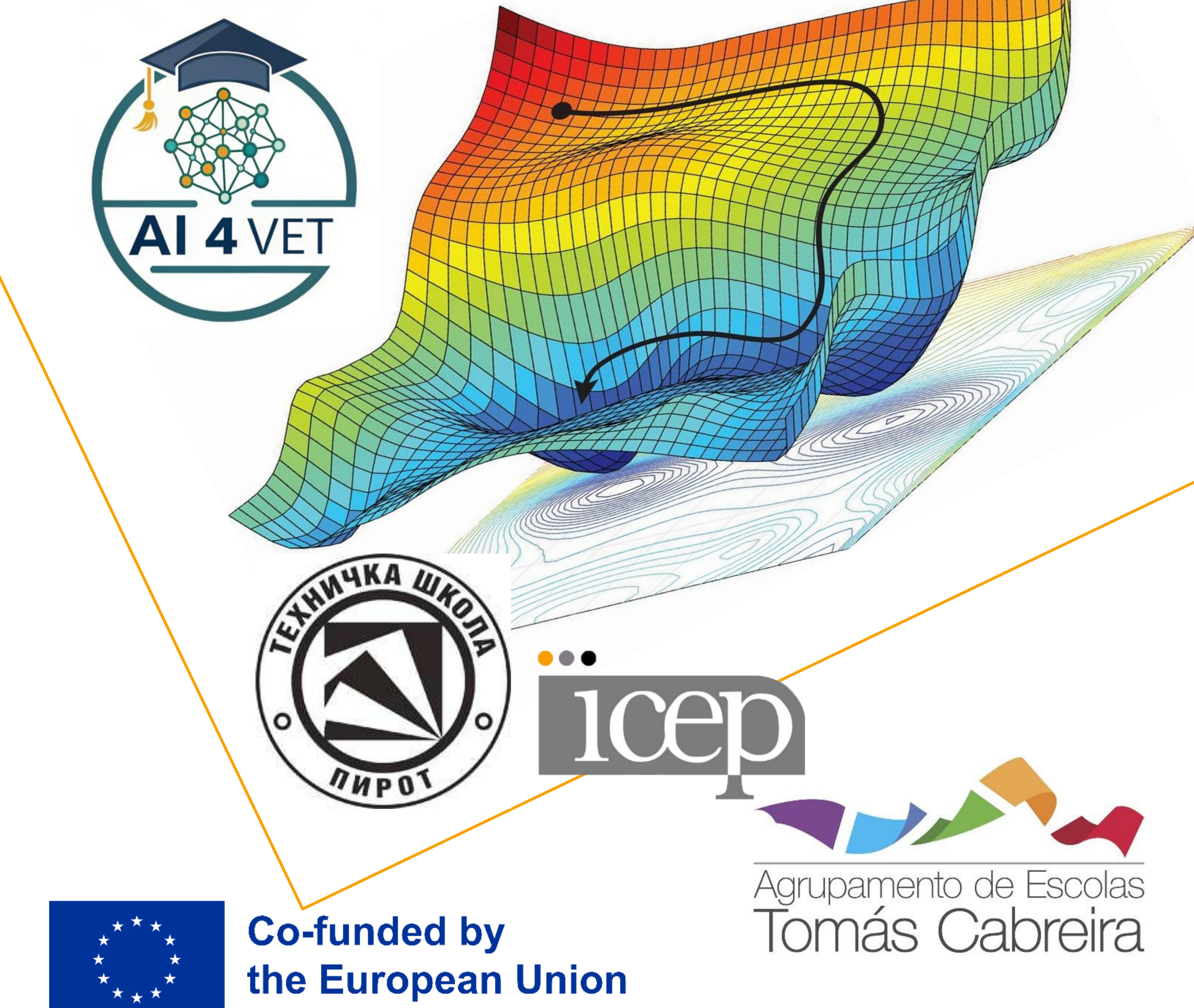


ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING FUNDAMENTALS

Technical school Pirot

KA220-VET - Cooperation partnerships in vocational education and training

Project Title: AI tools for VET schools

Document Date: Jun 2025
This material has been compiled and prepared for purposes of Erasmus project by:
Technical school Pirot (author: Boban Blagojević, co-authors: Bojan Ćirić, Aleksandar Madić)
ICEP (author: Ladislav Mariš, co-author: Adelaida Fanfarova)
Agrupamento de Escolas Tomas Cabreira (author: Sandra Nobre, co-authors: Rui Dias, Gilherme Mota, Carla Lima, Maria Torrinha)
Translated by: Bojana Stojanović

ADDRESS INFORMATION

Takovska 22, Pirot, Serbia
Web: https://book.tsp.edu.rs

Contents 2

1. Artificial Intelligence 4

1.1 The Concept of Artificial Intelligence 4
1.2 Turing's test 5
1.3 Areas of Artificial Intelligence 8
1.4 Regulation of Artificial Intelligence, Legal and Ethical Challenges 14
1.5 Narrow, General, and Superintelligence 16

2. Machine Learning 20

2.1 The Relationship between Artificial Intelligence and Machine Learning 20
2.2 Data-driven programming 23
2.3 Basic concepts of machine learning 28
2.4 A machine learning process 30
2.5 Types of machine learning 33
2.6 Data in Machine Learning 36
2.7 Exploratory Data Analysis (EDA) 41
2.8 Creating a Representation of a Dataset 46
2.9 Training, validation and testing sets 48

3. Training Models 51

3.1 Linear Regression 51
3.2 Gradient Descent 54
3.3 Polynomial regression 60
3.4 Multiple linear regression 66
3.5 Classification, types of classification, and matrix of confusion 68
3.6 Logistic regression 71
3.7 Decision tree 74
3.8 K-Nearest Neighbor (kNN) Algorithm 81
3.9 Hyperparameters 83
3.10 Generalization, under-adaptation, and over-adaptation 85
3.11 Validation, cross-validation 86
3.12 Regularization 87

4.1 Neural Networks 88

4.1 Neural Networks 88
4.2 Training of neural networks 93
4.3 Convolutional Neural Networks (CNN) 95
4.4 Recurrent neural networks 105
4.5 K-Means algorithm 108
Reference 114

1. ARTIFICIAL INTELLIGENCE

Welcome to the topic of **Artificial Intelligence (AI)**! In this section, we will explore the fascinating world of AI, which involves creating systems that can perform tasks typically requiring human intelligence. You will learn about the history and development of AI, the fundamental concepts, and the ethical considerations surrounding its use. We will also discuss the different types of AI, including narrow, general, and superintelligence, and how AI is transforming various aspects of our daily lives.

1.1 The Concept of Artificial Intelligence

Artificial Intelligence (*AI*) is a discipline that deals with the development of programs, which with their capabilities give the impression of intelligent behaviour. These programs are characterized by the ability to spot complex relationships and draw conclusions based on them. We will see that these activities have a foundation in disciplines such as mathematics, computer science, computer science and robotics. Because of its wider distribution, the field is also related to all other disciplines that deal with the understanding of intelligence, such as neuroscience, philosophy and art and its effects on society, such as sociology, law and ethics.

It is important to emphasize that not every program that has some form of *intelligent* behaviour has to be based on artificial intelligence. Let's look at a program that opens a door when entering a building. This functionality can be enabled by proximity sensors, which detect the presence or require that a code be entered that should match the expected code. We could cover both of these scenarios with classical programming techniques by comparing the distance measured by the sensor with some boundary distance, i.e. I'm going to enter a code with the correct code. On the other hand, if it is necessary for an accompanying camera to recognize our face to enter a building, we will, as we will soon see, need the help of artificial intelligence.

History of Artificial Intelligence

The history of artificial intelligence (AI) is marked by significant milestones that reflect the evolution of technology and human understanding of intelligence. From its conceptual beginnings to its current applications, AI has undergone various transformations influenced by research breakthroughs, societal needs, and technological advancements.

Early Foundations (1950s)

The journey of AI began in the 1950s, a decade that laid the groundwork for the field. In 1950, Alan Turing published his seminal paper "Computing Machinery and Intelligence," introducing the Turing test, which aimed to assess a machine's ability to exhibit intelligent behaviour indistinguishable from that of a human. The following year, Marvin Minsky and Dean Edmonds created SNARC, the first artificial neural network (ANN), simulating a network of neurons using vacuum tubes. In 1956, at a workshop organized by John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon, the term "artificial intelligence" was coined. This event is widely regarded as the founding moment of AI as a distinct field.

The Rise of Machine Learning (1960s-1970s)

The 1960s saw the development of early AI programs such as Eliza, a chatbot capable of engaging in simple conversations, and Shakey, the first mobile robot with AI capabilities. However, this period also faced challenges. The limitations of early neural networks were highlighted in 1969 when Marvin Minsky and Seymour Papert published *Perceptrons*, which led to a decline in neural network research in favor of symbolic AI approaches. The 1970s marked an "AI winter," characterized by reduced funding and interest due to unmet expectations. A pivotal report by James Lighthill in 1973 criticized AI research in the UK, leading to significant cuts in government support.

Revival and Expansion (1980s-1990s)

AI experienced a renaissance in the 1980s with the commercialization of Lisp machines and renewed interest in expert systems. This period saw advancements in knowledge representation and reasoning techniques, which allowed for more sophisticated AI applications. The introduction of multi-layer ANN training algorithms has further revitalized neural network research. By the 1990s, AI began to integrate into practical applications such as speech recognition and video processing. IBM's Deep Blue made headlines by defeating world chess champion Garry Kasparov in 1997, showcasing AI's potential in strategic thinking.

Modern Era (2000s-Present)

The 21st century has witnessed an explosion in AI capabilities driven by advancements in machine learning, particularly deep learning. Technologies such as IBM Watson, personal assistants like Siri and Alexa, facial recognition systems, and generative models like GPT have become integral to everyday life. The rise of big data and increased computational power have enabled these systems to learn from vast amounts of information, leading to significant improvements in performance across various domains. Today, discussions around AI also encompass ethical considerations and societal impacts. As AI systems become more prevalent, issues related to privacy, bias, and accountability are increasingly scrutinized.

Key Milestones

- **1950**: Alan Turing proposes the Turing Test.
- **1956**: Dartmouth Conference establishes AI as a field.
- **1966**: ELIZA, an early NLP program, is created.
- **1997**: IBM's Deep Blue defeats Garry Kasparov.
- **2012**: Deep learning breakthroughs with AlexNet winning the ImageNet competition.
- **2020s**: AI becomes integral to various industries, raising ethical and regulatory concerns.

1.2 Turing's test

The Turing test, proposed by Alan Turing (1950), was designed as a thought experiment that would sidestep the philosophical vagueness of the question “Can a machine think?” A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer. Chapter 28 discusses the details of the test and whether a computer would really be intelligent if it passed. For now, we note that programming a computer to pass a rigorously applied test provides plenty to work on. The computer would need the following capabilities:

- natural language processing to communicate successfully in a human language;
- knowledge representation to store what it knows or hears;
- automated reasoning to answer questions and to draw new conclusions;
- machine learning to adapt to new circumstances and to detect and extrapolate patterns.

Turing viewed the physical simulation of a person as unnecessary to demonstrate intelligence. However, other researchers have proposed a total Turing test, which requires interaction with objects and people in the real world. To pass the total Turing test, a robot will need

- computer vision and speech recognition to perceive the world;
- robotics to manipulate objects and move about.

These six disciplines compose most of AI. Yet AI researchers have devoted little effort to passing the Turing test, believing that it is more important to study the underlying principles of intelligence. The quest for “artificial flight” succeeded when engineers and inventors stopped imitating birds and started using wind tunnels and learning about aerodynamics. Aeronautical engineering texts do not define the goal of their field as making “machines that fly so exactly like pigeons that they can fool even other pigeons.”

Can a machine think?

Interestingly, the development of artificial intelligence has been fraught with obstacles.

The question is, "Can a machine think?" In 1950, English mathematician Alan Turing signaled the beginning of the development of the field now known as artificial intelligence. A few years after this great idea, in 1956, eminent scientists John McCarthy, Marvin Minsky, Nathaniel Rochester and Claude Shannon gathered for a conference in Dartmouth, which lasted as long as a month, with the desire to define research goals and protocols in this field. It was then that it was given its official name and for the first time actually presented as an artificial intelligence.

At the time of the advent of artificial intelligence, computers were very different than they are today. They were much smaller in capacity and speed, and much higher in price. Therefore, research in this area required different design solutions, and depended on official support and stable sources of funding.

In the first wave of the development of artificial intelligence, which lasted until the beginning of the seventies of the 20th century, many interesting programs appeared. The first among them was *the program Logic Theorist* written in 1956, which examined the capacities of mathematical logic and deriving conclusions. This program managed to prove 38 of the first 52 theorems listed in the famous book *Principles of Mathematics.* There is also the *ELIZA* program, which could simulate a conversation with users by following simple rules and constructions in English. The motivation for research in the field of communication came from Alan Turing's proposal to declare intelligent machines that can communicate in such a way that they do not give the impression that a person is talking to a machine. This test is now known as the Turing test.

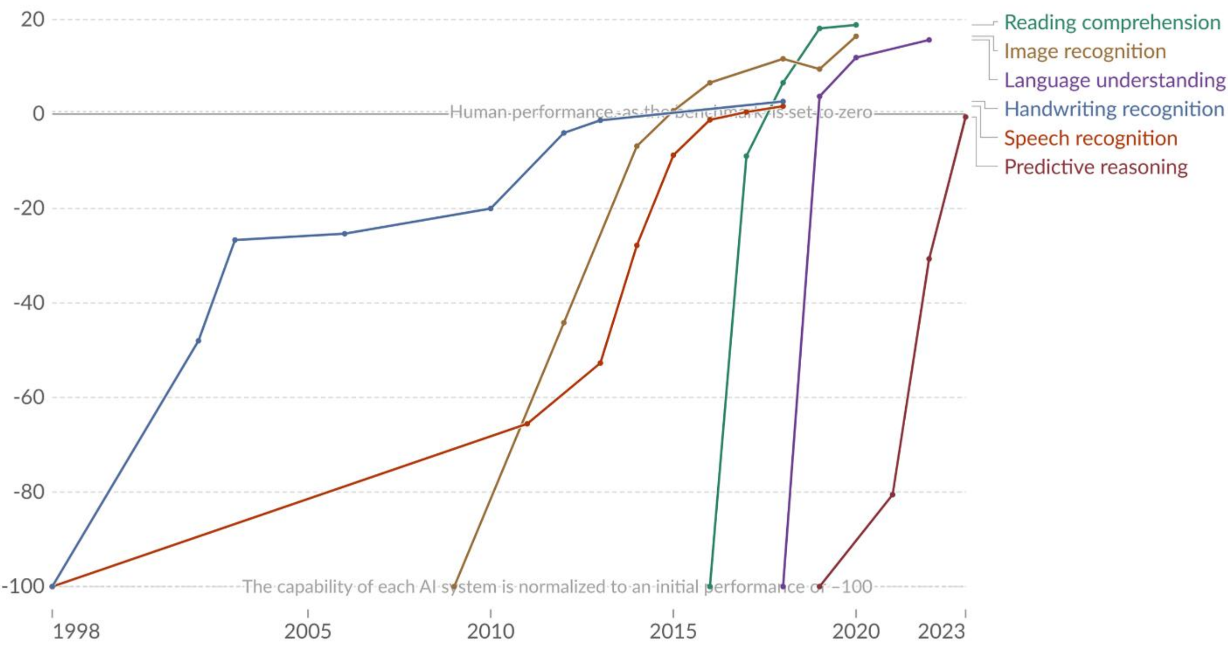
In the period up to the beginning of the 1970s, many ideas emerged that would later be used for breakthroughs in modern artificial intelligence. One such is the idea of the perceptron, the basis of today's neural networks, introduced in 1957 by Frank Rosenblatt. In the optimistic statements of this researcher, the perceptron had the power to learn, make decisions and translate languages, but it took a long time to confirm this.

Due to the lack of funding, the first wave of artificial intelligence development was followed by the so-called first winter of artificial intelligence. This status is partly due to ambitious projects whose results have been lacking due to limited computer capacities and a lack of available data.

One of the interesting and stimulating events in the history of artificial intelligence occurred in 1997 when *IBM's* DeepBlue *system* managed to beat the chess game of world grandmaster Garry Kasparov. The *DeepBlue system* is a representative of the class of so-called expert systems, systems that, on the basis of a database containing a multitude of rules of the form if-then, by applying logical rules, could imitate the reasoning of domain experts and give a correct result. Google's DeepMind's *AlphaGo system had a similar effect on the development of artificial intelligence* almost 20 years later, in 2016, when it defeated world champion Lee Sedol in the game of Go.

In 2011, the ability of machines to find the answer to a question posed in English was demonstrated by *IBM*'s *Watson* system in the *Jeopardy* quiz! *Watson* defeated its two opponents, winners of previous editions of the quiz, by giving the fastest correct answers to the questions asked. Sources who wrote about this event stated that *Watson* was able to process 500 GB of content per second, i.e. about a million books.

On the other hand, the capacity of machines to recognize and distinguish objects in images was demonstrated in 2012 by *the Google X* team, who created a program that can recognize cats in images. This program saw over 10 million images in 3 days and learned to recognize cats. To date, the capacity of recognition systems has been greatly improved and in many applications such systems give more accurate answers than most people. In the image below you can see the development trend systems for handwritten text recognition, speech recognition, image recognition, and two more recent results with a remarkable growth in capabilities are related to language comprehension tasks.



The image is taken from https://ourworldindata.org/brief-history-of-ai

These achievements were also a prelude to a far brighter continuation of the development of artificial intelligence, both because of the availability of the Internet, the web and more data, as well as because of computers whose processing power is incomparably greater than computers in the 1950s. This has also led to a paradigm shift that has been dominant in the field and the transition from logic-based systems to statistics-based systems.

The story of the development of artificial intelligence is also related to robots. Not only in science fiction novels and movies, but also when it comes to the appearance of real robots. In 1950, the American scientist Claude Shannon designed a mouse that could find its way and get out of the maze. In the spirit of Greek mythology, the mouse was named Theseus. In 1966, a team of scientists from the Stanford Research Institute began work on the development of the Shakey robot, the first robot capable of moving and inferring about the environment. The first autonomous vehicle ALVINN (acronym for Autonomous *Land Vehicle In a Neural Network)*, worked on by a team of researchers from Carnegie Mellon University, was constructed in 1989 and successfully covered 145 km traveling at a speed of 110 km per hour among other cars.

1.3 Areas of Artificial Intelligence

In this lesson, we will explore some areas of artificial intelligence. The boundaries between them are not strict, and often the techniques used to solve the problems of one area can be helpful in solving the problems of another area. The real power of artificial intelligence will actually be in connecting all areas.

Computer vision

Computer vision is a field of artificial intelligence that deals with the development of algorithms and tools that give computers the ability to understand the visual world like humans. Such are, for example, the tasks of recognizing objects in images, understanding their relationships, recognizing colours and textures, then recognizing movements, actions and their characteristics. As this field is primarily concerned with the analysis of images and videos, we will also get to know some of the most common tasks of this field.

The task **of image classification** is used to determine what type of object is present in the image. For example, determining whether or not there is a dog in an image is the task of classifying images. **Object Detection** is the task of locating objects and answering the question of where exactly objects are located in an image. Such, for example, is the task of framing the dog and cat that are in the picture below. The Task of **Image Segmentation** is used to determine the exact shape of objects that appear in an image. So, now, the finer separation of the contours of the dog and cat in the third image is an example of segmentation.

The Three Primary Tasks of Computer Vision in Working with Images

All these tasks are very applicable in many disciplines such as autonomous driving, medical image analysis or satellite image analysis, and allow us to search and organize images more easily.

What tasks do the following problems fall for:
To find out if there is a pedestrian in the picture.
Separating the contours of traffic lights, pavement and pedestrians in the picture,
Do you want to know where the sign is in the picture?

When it comes to video processing, the most common tasks are object tracking, action recognition, and positioning.

The Task of Object Tracking, as the name suggests, allows you to track objects in a video in real time. For example, tracking a neighbouring car while driving autonomously and tracking a player's movements during a match are examples of object tracking.

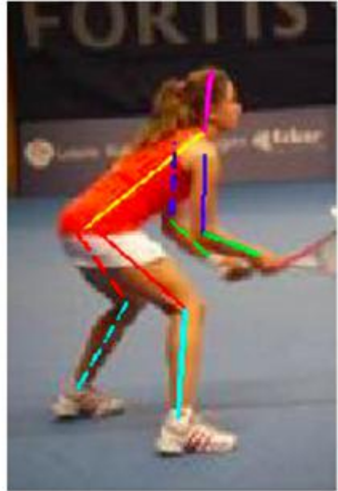
Object tracking

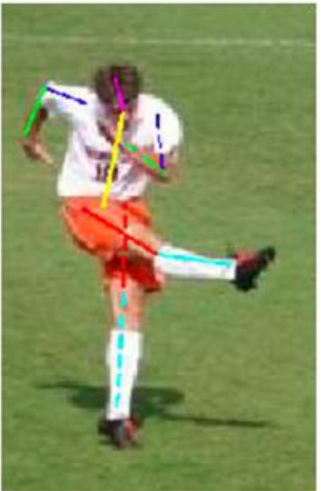
(image taken from https://docs.ultralytics.com/modes/track/)

The ***action recognition*** task is the ability to recognize and name an action that is present in a video, for example, jumping into the water or closing a window. These tasks help us to better understand video content and search for it more efficiently.

Examples of recognizing actions in videos

Pose estimation is a task that deals with recognizing the figures of people in videos in real time and extracting all the key points of their skeleton. These are the most common choroids of the eyes, nose, mouth, shoulders, elbows, waist, hands, knees and feet. These tasks help us with interactive animations, augmented reality modelling, and a variety of other applications.





Object Position Recognition Task: Images from the Leeds Sports Pose Dataset

What task do we need to solve in order to:

- Analyse whether we are sitting correctly,

- Recognize the exit to the pet's yard,

- Do you want to keep track of the customer's movements in the store?

Later, we'll get to the datasets used in computer vision and convolutional network tasks, a special type of neural network used in image and video processing tasks.

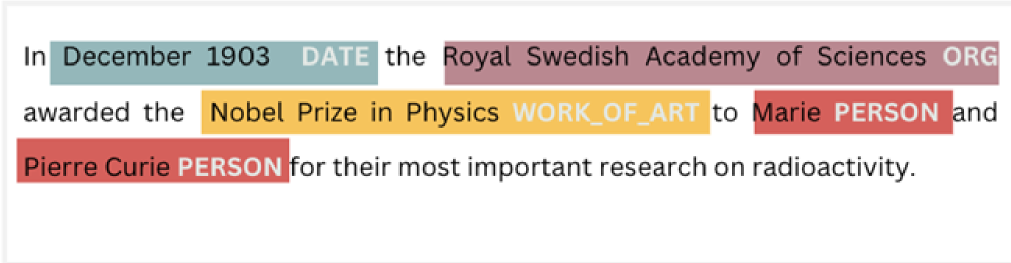
Natural Language Processing

Natural Language Processing *(NLP*) is a field of artificial intelligence that deals with tasks related to the understanding and generation of natural language. As we know, there are over 7,000 languages, and each of them has its own peculiarities in terms of vocabulary, grammatical rules and meanings. Some common tasks encountered in natural language processing will be described below.

Just like in the image classification tasks, in the text **classification** tasks we try to conclude whether the text belongs to a category or not. For example, is it a newspaper article on the subject of sports, is it written in Spanish, is it positive, i.e. contains some complimentary comment, whether it is true or false and the like.

Text classification

Named Entity Recognition is a task related to the recognition of some parts of the text that are relevant for its further analysis. These are usually the names of the people who appear in it, dates, geolocation names or in some professional texts, for example in the field of medicine, symptoms or names of diseases. By a single name, these parts of the text are called entities.



Example of tagging named entities

The Task of **Translation Machine** is to develop tools that allow us to translate content from one language into the content of another language. We will agree that this task is the basis for successful communication and the availability of information, but also that it is complicated because each language and each culture that the language represents has its own peculiarities such as phrases, idioms, slang or sarcasm that are very challenging to translate (how to translate *rack one's brain?*).

Question-answering systems deal with the question of how to find a concrete answer to a given question. They are generalizations of classic information retrieval systems and allow us to get the information we need more easily.

Summarizing all important information from a number of different sources is known as **a summarization** task. Just like in the previous task, the summaries that come with the summarization tasks should make it easier for us to go through a larger amount of content or remind us of important information and details of the content we have read.

Summarization

In addition to tasks related to text and textual content, natural language processing also deals with speech analysis. There are two tasks in particular: *speech-to-text* and vice versa, *text-to-speech*. These two groups of tasks are especially important for the development of personal assistants, programs such as *Siri, Cortana*, or *Alexa*, that can understand voice messages and perform a requested task accordingly, for example, set an alarm or call someone from the phonebook.

What tasks do the following problems fall for:
Extracting the name of the organization in the text,
Finding out who is the author of a book,
What is the meaning of the word Netizen?

Generative Artificial Intelligence (AGI) is a field of artificial intelligence that deals with the generation of content such as images, text, audio, or video. Over the past few years, the breakthroughs in this field have been impressive.

ChatGPT is a program that has made a breakthrough in the field of generating text content. He, in accordance with the user's instructions, the so-called prompts, can generate appropriate text content. It should be remembered that texts generated in this way do not have to be absolutely accurate, they may contain incorrect data, fabricated references or offensive content. Therefore, before use, you should check everything that the program has generated. If you create an account at chat.openai.com, you can try for yourself how *the ChatGPT* program works. Behind *the ChatGPT* program is the *OpenAI* community.

The StableDiffusion *program,* unlike *ChatGPT*, which generates text, generates images based on instructions. For example, all the images listed below were generated by this program. It is open source and can be downloaded from the official GitHub repository with the accompanying code. You can test the model itself at https://stablediffusionweb.com/. Keep in mind that this service is used by a large number of people for free and is sometimes not available. The name of the program itself is a popular technique used in this field.

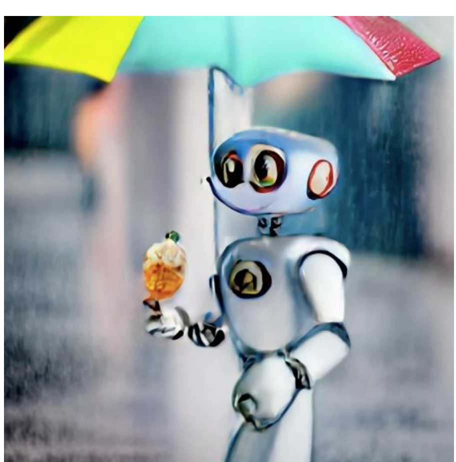
Examples of images generated by StableDiffusion

Often, when generating images, the desired style of the new image can also be selected. This technique is known as **style transfer***.* You can see an example in the picture below.

In addition to images and text, artificial intelligence can also generate audio content. At this link, you can test Meta's *MusicGen program,* by describing in words what kind of music you want to generate and possibly leaving an example for style transfer. Then you can listen to your own content. You can also try with programs that perform style transfer when generating a voice (mimic another person's voice) or compose music yourself based on the what they have already "heard" in the data. One such project is Magenta. A link https://magenta.github.io/listen-to-transformer/ will take you to it.

Ask ChatGPT to create a quiz with questions about artificial intelligence, and then see how many questions you can answer.

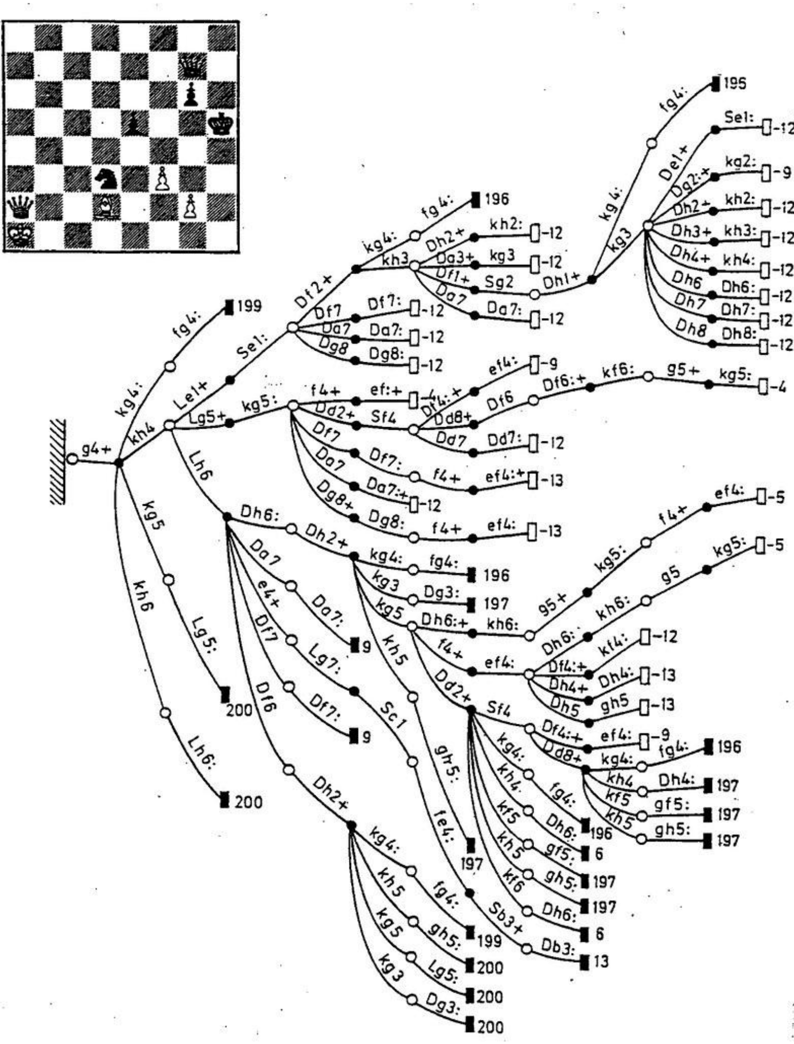
Try giving instructions to StableDiffusion or Dalle-mini that will generate an image like this:



OpenAI's DALL-E program also generates images based on user guidelines. The dalle-mini program is a publicly available version of this program. It is available at https://huggingface.co/spaces/dalle-mini/dalle-mini .

Playing games

One of the first tasks in which artificial intelligence has been tried is the game of chess. With its victory over Grandmaster Garry Kasparov, this area of research has received a lot of sympathy and support from the side of artificial intelligence. Although they have a precise set of instructions and rules, the games are characterized by the property of a combinatorial explosion - a large number of possible choices of actions after a certain number of steps. This further means that the games do not allow for finding solutions by applying common programming techniques in some within a reasonable time. In addition to the game of chess, artificial intelligence has also become famous in the game of go with *the AlphaGo program*, then playing Atari video games and strategies such as Dota, Starcraft and others. At the end of the course, you will learn more about the field called reinforcement learning, which is actively applied in this area.



Combinatorial explosion in the game of chess

Check to see if the game you love uses artificial intelligence in some of its segments.

Robotics

Artificial intelligence often needs to improve the behaviors and capabilities of physical objects such as robots, industrial machines, cars, drones, home appliances or medical devices. Information about the world reaches these objects through voice instructions, camera recordings, or sensor measurements, and their task is to process them and transform them into decisions. The role of artificial intelligence in this sphere is to improve the capabilities of objects and help them adopt intelligent behavior.

1.4 Regulation of Artificial Intelligence, Legal and Ethical Challenges

The rapid advancement of artificial intelligence (AI) technologies has prompted urgent discussions surrounding their regulation. As AI systems become increasingly integrated into various sectors, including healthcare, finance, and transportation, the need for effective legal frameworks and ethical guidelines becomes paramount. This text explores the complexities of regulating AI, highlighting the legal and ethical challenges that arise in this evolving landscape.

The Need for Regulation

AI technologies possess unique characteristics that differentiate them from traditional technologies. They often operate as "black boxes," where their decision-making processes are not transparent even to their developers. This opacity raises significant concerns regarding accountability, bias, and the potential for harm. As AI systems can influence critical areas such as employment decisions, criminal justice outcomes, and public safety, the stakes for effective regulation are high.

Key Drivers for Regulation:

- **Safety and Welfare**: Ensuring that AI systems do not compromise public safety or individual rights is a primary concern. Incidents involving autonomous vehicles or biased algorithms in hiring processes underscore the need for regulatory oversight.
- **Accountability**: Establishing clear accountability mechanisms is essential to address who is responsible when AI systems cause harm or make erroneous decisions.
- **Ethical Use**: As AI systems can perpetuate biases present in training data, regulations must ensure ethical standards are upheld to prevent discrimination and unfair treatment.

Legal Challenges in Regulating AI

The legal landscape surrounding AI regulation is complex and often lagging behind technological advancements. Traditional legal frameworks may not adequately address the nuances of AI systems.

1. Defining Liability

One of the foremost challenges is determining liability when an AI system causes harm. Questions arise regarding whether liability should fall on the developers, users, or the AI itself. For instance, if an autonomous vehicle is involved in an accident, should the manufacturer be held accountable, or does responsibility lie with the owner?

2. Intellectual Property Issues

As AI-generated content becomes more prevalent, questions about intellectual property rights emerge. Who owns the rights to works created by AI? Current laws may not sufficiently cover these scenarios, leading to potential conflicts over ownership and copyright.

3. Data Privacy Concerns

AI systems rely heavily on data for training and operation. The collection and use of personal data raise significant privacy concerns. Regulations must balance the need for data to improve AI capabilities with individuals' rights to privacy and data protection.

Ethical Challenges in Regulating AI

Beyond legal considerations, ethical challenges play a crucial role in shaping AI regulation.

1. Bias and Fairness

AI systems can inadvertently perpetuate biases present in training data, leading to unfair outcomes. Regulators must establish guidelines to ensure fairness and mitigate bias in AI algorithms. This involves not only technical solutions but also a commitment to ethical principles that prioritize equitable treatment.

2. Transparency and Explainability

The "black box" nature of many AI systems complicates transparency efforts. Users and affected individuals often lack insight into how decisions are made. Regulations should promote explainability, ensuring that individuals can understand the rationale behind decisions made by AI systems.

3. Informed Consent

As AI technologies increasingly influence personal decisions—such as credit approvals or job applications—ensuring informed consent becomes critical. Individuals should be aware of how their data is used and how AI impacts decision-making processes affecting them.

Approaches to Regulation

Given these challenges, various approaches to regulating AI have emerged.

1. Risk-Based Frameworks

The European Union has proposed a risk-based approach to AI regulation, categorizing applications based on risk levels—from minimal to unacceptable risks. This framework allows for tailored regulations that address specific concerns associated with different types of AI applications.

2. Continuous Oversight

Regulating AI requires ongoing vigilance due to its rapidly evolving nature. Regulatory bodies must adapt to new developments in technology and continuously assess the impact of existing regulations on society.

3. Collaborative Governance

Effective regulation may involve collaboration between governments, industry stakeholders, and civil society organizations. Engaging diverse perspectives can lead to more comprehensive regulations that consider various interests and ethical considerations.

Conclusion

The regulation of artificial intelligence presents multifaceted legal and ethical challenges that require careful consideration and proactive measures. As AI continues to permeate various aspects of life, establishing robust regulatory frameworks is essential to safeguard public welfare, ensure accountability, and uphold ethical standards. By addressing these challenges through collaborative governance and adaptive regulatory approaches, society can harness the benefits of AI while mitigating its risks effectively.

1.5 Narrow, General, and Superintelligence

Now that we know the fields of application of artificial intelligence and its reach, we can also introduce concepts such as narrow and general artificial intelligence, superintelligence and the singularity of artificial intelligence.

We have seen that the examples of artificial intelligence programs that we have listed are focused on solving one specific task, for example, playing chess, translating languages, segmenting images, and the like. We say that such programs have **Narrow Artificial Intelligence** (*narrow AI*). In contrast to Artificial Intelligence, there is **General Artificial Intelligence** (*General AI*). The programs of this group should be more approaching to human intelligence, to enable the solution of a multitude of different tasks. According to leading

researchers in the discipline, programs of this type are still in their infancy and will not be developed in the foreseeable future.

Superintelligence is a term that refers to intelligence greater than that of humans. Such intelligence could help us solve problems such as global warming, providing enough food, or finding a cure for cancer. It could, hypothetically, get out of control, continue to improve, and in some ways endanger humanity. **AI singularity** is a theoretical concept denoting the dominance of artificial intelligence over human intelligence and is often encountered as a theme in science fiction movies and books.

Looking at artificial intelligence from this angle is of particular interest to philosophers, historians, and researchers in the field of social sciences. Israeli historian Yuval Noah Harari and Swedish philosopher Nick Bostrom have written about them. You can find a lot of these stories on YouTube.

Try to think of another example of the application of superintelligence. What are the problems that people, despite the development of society, science and technology, still do not know how to solve?

Explore the meaning of the term *existential risk*. How do you view it?

The field of artificial intelligence (AI) encompasses a variety of systems and capabilities, which can be categorized into three primary types: **Narrow Intelligence**, **General Intelligence** and **Superintelligence**. Each category represents a different level of complexity and capability in AI systems, reflecting the ongoing evolution of technology and its potential impact on society. Understanding these distinctions is crucial for grasping the current landscape of AI and its future implications.

Narrow Intelligence (Weak AI)

Narrow intelligence, often referred to as weak AI, is designed to perform specific tasks or solve particular problems. These AI systems excel in their designated functions but lack the ability to operate outside their predefined parameters.

Characteristics of Narrow Intelligence:

- **Task-Specific Functionality**: Narrow AI systems are built to handle specific tasks, such as image recognition, language translation, or playing games like chess. They operate effectively within their limited scope but do not possess general reasoning capabilities.
- **Data-Driven Learning**: These systems rely heavily on large datasets for training. Machine learning algorithms analyze patterns in data to make predictions or decisions based on the information they have been provided.
- **Lack of Self-Awareness**: Narrow AI does not possess consciousness or self-awareness. It operates based on algorithms and predefined rules without understanding the context or implications of its actions.

Examples of Narrow Intelligence:

- **Voice Assistants**: Applications like Siri and Alexa can perform tasks such as setting reminders or answering questions but cannot engage in conversations beyond their programmed capabilities.
- **Recommendation Systems**: Platforms like Netflix and Amazon use narrow AI to suggest content based on user preferences, analyzing past behavior to predict future choices.
- **Autonomous Vehicles**: While self-driving cars utilize complex algorithms to navigate roads, they are still limited to driving tasks and cannot perform other human-like cognitive functions.

General Intelligence (Strong AI)

General intelligence, also known as artificial general intelligence (AGI), refers to a theoretical form of AI that possesses the ability to understand, learn, and apply knowledge across a wide range of tasks at a level comparable to human intelligence.

Characteristics of General Intelligence:

- **Versatile Learning**: AGI can generalize knowledge from one domain to another, allowing it to adapt to new situations without requiring extensive retraining.
- **Reasoning and Problem-Solving**: Unlike narrow AI, AGI can reason through complex problems, understand abstract concepts, and make decisions based on incomplete information.
- **Human-Like Interaction**: AGI systems would be capable of engaging in natural conversations with humans, demonstrating emotional understanding and social awareness.

Current Status of General Intelligence:

As of now, AGI remains largely theoretical. While significant progress has been made in machine learning and neural networks, no existing system possesses the full range of cognitive abilities associated with human intelligence. Researchers continue to explore pathways toward achieving AGI, focusing on developing algorithms that can learn more flexibly and autonomously.

Superintelligence

Superintelligence refers to a hypothetical form of AI that surpasses human intelligence in virtually every aspect. This concept raises profound questions about the future of humanity and the ethical implications of creating machines that could potentially outthink their creators.

Characteristics of Superintelligence:

- **Exponential Growth**: Superintelligent systems would have the capacity for rapid self-improvement, leading to an intelligence explosion where machines could enhance their own capabilities beyond human comprehension.
- **Problem-Solving Abilities**: Such systems could solve complex global challenges—ranging from climate change to disease eradication—more efficiently than any human or collective group.
- **Ethical Considerations**: The development of superintelligence poses significant ethical dilemmas regarding control, safety, and the potential consequences for society. Ensuring that superintelligent systems align with human values is a critical concern.

Theoretical Implications:

The discussion surrounding superintelligence often includes scenarios about the "singularity," a point at which technological growth becomes uncontrollable and irreversible. This concept has sparked debates among scientists, ethicists, and futurists about the potential risks associated with creating entities that could operate independently of human oversight.

Challenges in Advancing Towards General and Superintelligence

While the pursuit of general and superintelligent AI presents exciting possibilities, several challenges must be addressed:

1. **Technical Limitations**: Current AI systems struggle with generalization; they excel in narrow tasks but fail when faced with unfamiliar contexts or problems outside their training data.

2. **Ethical Concerns**: The development of AGI and superintelligence raises ethical questions about autonomy, decision-making authority, and the potential for misuse in military or surveillance applications.
3. **Safety Measures**: Ensuring that advanced AI systems operate safely is paramount. Researchers advocate for robust safety protocols and alignment strategies to prevent unintended consequences.
4. **Public Perception**: Misunderstandings about AI capabilities can lead to unrealistic expectations or fears regarding its impact on society. Public education is essential for fostering informed discussions about AI technologies.

Conclusion

The distinctions between narrow intelligence, general intelligence, and superintelligence highlight the diverse landscape within the field of artificial intelligence. While narrow AI continues to thrive in various applications today, the pursuit of AGI remains an ambitious goal for researchers worldwide. The prospect of superintelligence invites both excitement and caution as society grapples with the implications of creating machines that could surpass human intellect. As advancements continue, addressing ethical considerations and ensuring responsible development will be crucial for harnessing the benefits of AI while mitigating its risks effectively.

2. MACHINE LEARNING

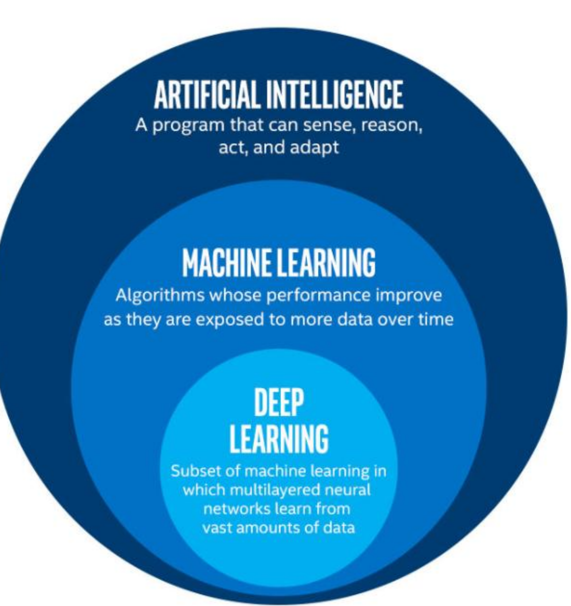
Welcome to the topic of **Machine Learning (ML)**! Machine learning is a subset of AI that focuses on developing algorithms that allow computers to learn from and make predictions based on data. In this section, you will gain a deep understanding of the basic concepts of machine learning, the different types of learning (supervised, unsupervised, semi-supervised, and reinforcement learning), and the importance of data in this process. We will also cover the steps involved in building and validating machine learning models, and how these models can be applied to solve real-world problems.

2.1 The Relationship between Artificial Intelligence and Machine Learning

Artificial intelligence (AI) and machine learning (ML) are two interconnected fields that have attracted significant attention in recent years for their transformative impact on technology and society. Although often used interchangeably, they represent different concepts in the broader landscape of computational intelligence. Understanding their relationship is critical to understanding how modern AI systems work and evolve.

1. **Artificial Intelligence (AI)** is a broad field that encompasses various technologies aimed at creating systems capable of performing tasks that typically require human intelligence. These tasks include problem-solving, understanding natural language, recognizing patterns, and making decisions.

2. **Machine Learning (ML)** is a subset of AI that focuses on developing algorithms that allow computers to learn from and make predictions based on data. Instead of being explicitly programmed to perform a task, ML algorithms use statistical techniques to identify patterns in data and improve their performance over time.



Example:

- **AI Application**: A virtual assistant like Siri or Alexa uses AI to understand and respond to voice commands.
- **ML Application**: A recommendation system on Netflix uses ML to suggest movies and TV shows based on your viewing history.

To visualize the relationship:

- **AI** is the overarching concept of machines being able to carry out tasks in a way that we would consider "smart."
- **ML** is a specific approach to achieving AI, where machines are given data and learn for themselves.

Data-Driven Programming

Data-driven programming is a paradigm where the program's behavior is controlled by data rather than by hard-coded logic. This approach allows for more flexible and adaptable systems. In data-driven programming, the data dictates the flow of the program, which can be particularly useful in applications like data processing, transformation, and analysis.

Examples:

- **AWK** and **sed** for text processing: These tools process text data based on patterns and rules defined in the data itself.
- **XSLT** for transforming XML documents: XSLT uses XML data to define how other XML documents should be transformed.
- **Procmail** for email filtering: Procmail uses rules defined in configuration files to filter and process incoming emails.

Basic Concepts of Machine Learning

Machine learning involves several key concepts:

1. **Supervised Learning**: The algorithm is trained on labeled data, meaning that each training example is paired with an output label. The goal is to learn a mapping from inputs to outputs that can be used to predict labels for new data
    - **Example**: Predicting house prices based on features like size, location, and number of bedrooms. The training data includes houses with known prices.

2. **Unsupervised Learning**: The algorithm is given data without explicit instructions on what to do with it. It tries to find hidden patterns or intrinsic structures in the input data
    - **Example**: Customer segmentation in marketing, where the algorithm groups customers based on purchasing behavior without predefined labels.

3. **Reinforcement Learning**: The algorithm learns by interacting with an environment and receiving feedback in the form of rewards or penalties. It aims to learn a strategy that maximizes cumulative rewards
    - **Example**: Training a robot to navigate a maze, where it receives rewards for reaching the end and penalties for hitting walls.

4. **Key Elements**:
    - **Task**: The problem to be solved.
    - **Experience**: The data used to learn.

  - **Performance**: The measure of how well the algorithm solves the task

Example:

- **Task**: Classifying emails as spam or not spam.
- **Experience**: A dataset of emails labeled as spam or not spam.
- **Performance**: Accuracy of the classification on new, unseen emails.

The Interconnection Between AI and ML

The relationship between AI and ML is fundamentally synergistic. While all ML is a form of AI, not all AI systems utilize machine learning techniques. Here's how they interact:

1. **Foundation of AI**: AI provides the theoretical framework for creating intelligent systems, while ML offers practical techniques for implementing these systems. For instance, natural language processing (NLP), a significant area of AI, relies heavily on ML algorithms to process and understand human language effectively.
2. **Learning Mechanisms**: Traditional AI systems were primarily rule-based, relying on predefined rules and logic to make decisions. However, these systems struggled with uncertainty and variability in data. ML introduced the ability for AI systems to learn from experience, allowing them to adapt to new information and make probabilistic decisions. This advancement is particularly important in dynamic environments like autonomous vehicles, which must navigate complex road conditions.
3. **Applications in Robotics**: The synergy between AI and ML is evident in robotics. Intelligent robots utilize AI for cognitive capabilities while employing ML algorithms for navigation, obstacle avoidance, and task execution. For example, warehouse robots can autonomously navigate their environment using computer vision powered by ML techniques.
4. **Predictive Analytics**: In business applications, predictive analytics leverages both AI and ML to forecast future outcomes based on historical data. AI provides the foundational framework for creating predictive models, while ML enhances these models' ability to learn from data and generate accurate predictions.

Key Techniques in Machine Learning

Machine learning encompasses several techniques that contribute to its effectiveness in enhancing AI applications:

1. **Supervised Learning**: In this approach, algorithms are trained on labeled datasets where the correct output is known. The model learns to map inputs to outputs based on this training data, enabling it to make predictions on new, unseen data.
2. **Unsupervised Learning**: Unlike supervised learning, unsupervised learning involves training algorithms on unlabeled data. The model identifies inherent patterns or structures within the data without explicit guidance on what to look for.
3. **Reinforcement Learning**: This technique involves training an agent through trial-and-error interactions with its environment. The agent receives feedback in the form of rewards or penalties based on its actions, allowing it to learn optimal strategies over time.
4. **Deep Learning**: A subset of ML that utilizes artificial neural networks with multiple layers (deep networks) to analyse complex datasets such as images or audio signals. Deep learning has driven many recent advancements in areas like computer vision and natural language processing.

Real-World Applications

The integration of AI and ML has led to significant advancements across various industries:

- **Healthcare**: AI-powered diagnostic tools leverage ML algorithms to analyze medical images for disease detection or predict patient outcomes based on historical health data.
- **Finance**: In finance, ML models are used for fraud detection by analyzing transaction patterns in real-time, providing banks with tools to mitigate risk effectively.
- **Autonomous Vehicles**: Self-driving cars utilize a combination of AI techniques such as computer vision and decision-making alongside ML algorithms like deep learning to interpret sensory data from cameras and LIDAR systems.
- **Retail**: E-commerce platforms employ ML algorithms for personalized recommendations based on user behaviour and preferences, enhancing customer experience.

Challenges in the Relationship Between AI and ML

Despite their potential benefits, the relationship between AI and ML also presents challenges:

1. **Data Quality**: The effectiveness of ML algorithms heavily relies on the quality of the training data. Poor-quality or biased data can lead to inaccurate predictions or reinforce existing biases within AI systems.
2. **Interpretability**: Many advanced ML models, especially deep learning networks, operate as "black boxes," making it challenging for users to understand how decisions are made. This lack of transparency can hinder trust in AI applications.
3. **Ethical Concerns**: As AI systems increasingly influence critical areas like hiring or law enforcement, ethical considerations regarding fairness, accountability, and transparency become paramount.

Conclusion

The relationship between artificial intelligence and machine learning is foundational to understanding modern technological advancements. While machine learning serves as a powerful tool within the broader field of AI, it is essential to recognize the distinct roles each plays in developing intelligent systems capable of transforming industries and improving lives. As both fields continue to evolve together, addressing challenges related to data quality, interpretability, and ethics will be crucial for harnessing their full potential responsibly.

2.2 Data-driven programming

The Importance of Large Amounts of Data in Machine Learning

In the realm of machine learning (ML), data is often referred to as the "fuel" that powers algorithms and models. The effectiveness and accuracy of machine learning systems heavily depend on the availability of large amounts of high-quality data. This section emphasizes the significance of data in ML, introduces the concept of data-driven programming, and explains fundamental concepts such as datasets, instances, attributes, input variables, and models.

The Role of Data in Machine Learning

Data serves as the foundation upon which machine learning models are built. Without sufficient data, machine learning algorithms cannot learn patterns or make predictions effectively. The importance of large datasets can be summarized in the following points:

1. **Learning from Examples**: Machine learning is fundamentally about learning from examples. Large datasets provide a diverse range of instances that help algorithms identify patterns and relationships within the data.

2. **Improved Accuracy**: Models trained on larger datasets tend to generalize better to unseen data, leading to improved accuracy in predictions. This is particularly crucial in applications such as healthcare diagnostics or financial forecasting, where precision is vital.
3. **Robustness**: A well-rounded dataset that encompasses various scenarios allows models to handle edge cases and anomalies more effectively. This robustness is essential for real-world applications where data can be noisy or incomplete.
4. **Feature Representation**: Large datasets enable the extraction of meaningful features that can significantly enhance model performance. With more data, algorithms can learn complex representations that capture underlying trends.
5. **Mitigating Overfitting**: Having a substantial amount of training data helps prevent overfitting, where a model learns noise instead of the underlying distribution. A larger dataset provides a better approximation of the true data distribution.

Data-Driven Programming

Data-driven programming is an approach that emphasizes the use of data as a primary driver for decision-making and algorithm development. In this paradigm, programmers focus on collecting, analyzing, and utilizing data to inform their coding practices rather than relying solely on predefined rules or logic.

Why We Need Data-Driven Programming

1. **Adaptability**: Data-driven programming allows systems to adapt to changing conditions by continuously learning from new data inputs.
2. **Enhanced Decision-Making**: By leveraging large datasets, developers can create more informed algorithms that lead to better outcomes.
3. **Automation**: Data-driven approaches facilitate automation by enabling machines to learn from historical data and make predictions without human intervention.
4. **Scalability**: As more data becomes available, systems can scale their capabilities without requiring significant changes to their underlying architecture.

Basic Concepts of Machine Learning

1. **Dataset**

A dataset is a structured collection of data used for training machine learning models. It consists of multiple instances (data points) organized into rows and columns, where each column represents an attribute or feature.

2. **Instance**

An instance refers to a single record or observation within a dataset. Each instance contains values for various attributes that describe its characteristics.

3. **Attribute**

Attributes are the individual variables or features that describe an instance in a dataset. They can be numerical (e.g. age, salary) or categorical (e.g. gender, color). Attributes are crucial for defining the input space for machine learning algorithms.

4. **Input Variables**

Input variables are specific attributes used as predictors in a machine learning model. These variables provide the necessary information for the model to learn patterns and make predictions about output variables (targets).

5. **Output Variables**

Output variables are the target values that the model aims to predict based on input variables. For example, in a housing price prediction model, input variables might include square footage and location, while the output variable would be the estimated price.

The Concept of a Model

In machine learning, a model is defined as a mathematical representation that maps given input variables to output variables based on learned patterns from training data. The process involves:

1. **Training**: During training, the model learns from labelled instances in the dataset by adjusting its parameters to minimize prediction errors.
2. **Mapping**: Once trained, the model can take new input variables and generate corresponding output predictions based on its learned mappings.
3. **Evaluation**: The performance of a model is evaluated using metrics such as accuracy, precision, recall, and F1 score on validation or test datasets.

The significance of large amounts of suitable data in machine learning cannot be overstated; it is essential for training effective models capable of making accurate predictions in real-world applications. Data-driven programming emerges as a vital methodology that leverages this abundance of data to inform algorithm development and decision-making processes. Understanding fundamental concepts such as datasets, instances, attributes, input variables, output variables, and models lays the groundwork for further exploration into machine learning techniques and their applications across various domains. As we continue to harness the power of data in machine learning, it is imperative to prioritize quality data collection and management practices to ensure robust and reliable outcomes in AI systems.

We will start this story with a brain teaser. Try to conclude from the numbers you have in the left column (we will call them **inputs**) how they relate to the numbers in the right column (we will call them **outputs)**.

Input	Output
1, 9, 5, 4	19
3, 8, 11	22
6, 7, 9, 2, 2	26
2, 3, 6	11

We assume that this task did not bother you much and that already in the second or third row of the table you came up with the idea that the numbers in the second column, i.e. output, represent the sum of the numbers that are in the left column, i.e. at the entrance. For this task, you also know how to write an algorithm (and by squinting!) that leads you to the solution. For example, In the Python programming language, *we* can

calculate the sum of the elements of the array [1, 9, 5, 4] by initially declaring the sum zero and then adding one element at a time until we reach the end of the array:

```
numbers = [1, 9, 5, 4]
sum = 0
for element in numbers:
  sum = sum + element
```

(The built-in sum function calculates this quickly for us and we actually use it more often.)

Now try your hand at the next brain teaser in which you need to figure out how the inputs and outputs are connected. The inputs are now photos of animals, and the outputs are the numbers 0 or 1.

Input	Output
	1
	0
	0
	1
	0

We’re sure you've got a lot of ideas here. And it's very possible that they're all valid! However, since the ones are next to the photos of the cats, and the zeros are next to the rest of the photos of the animals, we wanted you to conclude that this is a task in which you need to recognize whether there is a cat in the photo or not.

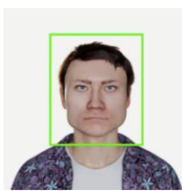
If there is a photo of a cat at the input, there is 1 at the output, and vice versa, if there is no photo of a cat at the input, there is 0 at the output. You will see a little later that this task is called a binary classification task and is very common in the field of machine learning.

Is it possible to create an algorithm to solve this problem? You will agree that it is important for us to be able to distinguish what is in the photos because that way we can make them easier to search and analyze. You can try to make a list of dozens of rules to determine whether there is a cat in the picture or not. Don't forget to consider the background, lighting, angle of view, and the fact that there are over 50 different types of cats.

There are actually a lot of problems like this, in which even if we try hard (we name 1000 rules) we are not able to write algorithms that solve them precisely. Some other examples are translating from one language to another and marking faces in photos. You will agree that these tasks are also important to us because they allow us to understand content written in a language we do not speak or to improve security. That is why we describe such problems **with datasets**, pairs of inputs and expected outputs, and are solved by data-driven programming techniques. In the case of the task of recognizing cats (and any other objects), a suitable set of data can be organized similarly to ours, with images at the inputs and values of 0 or 1 at the output. In the case of a translation task, it can be pairs of sentences in both languages, while in the case of a face marking task, it can be pairs of images, one without marked faces as input and one with marked faces as output. Here is an example.

Input

Output



Datasets that we can use to describe problems can be created through everyday activities. For example, after examining a patient in an electronic health record, a doctor records information about the patient such as age, gender, symptoms, drug allergies (all of these values are inputs), and the code corresponding to his diagnosis (output). Similarly, at airports, for each flight, information is known such as the time of departure, the airline, the type of aircraft, etc. (all these values are inputs) and information about whether that flight was delayed or not (exit).

Datasets can also be created specifically to solve a specific task. For example, the dataset we used to identify cats could have been created by a team of volunteers who looked at the images we had and followed our guidelines, for example, if there is a cat in the image, write 1, otherwise write 0, enter 1 or 0 in the exit column. For domain datasets, for example, recognizing changes in X-rays, medical experts would need to be hired who have the appropriate skills and knowledge to make decisions. A little later, you'll learn more about how datasets are created.

You're probably wondering: but how do we learn the connection between input and output in a data set? Just as there is an area that deals with the development of classical programming algorithms and analysing their properties, there is also an area that deals with the development of data-driven algorithms and the examination of their properties. This is called **machine learning**. *Machine learning* is at the core of all modern areas of artificial intelligence because it is closely related to data and ways of deriving knowledge from data. In the next lesson, machine learning will answer the question you are interested in.

It is important to emphasize that there are other areas that deal with data. Among them, the oldest is certainly **statistics**, a branch of mathematics that deals with collecting data, describing and analysing them,

as well as drawing conclusions from data. Statistical techniques are at the heart of many machine learning algorithms. **Data Science** is a discipline that emerged as a result of the inability of individual disciplines to answer many interesting questions. For example, every company is faced with the question of how to improve its services. To do this, the company can analyse user comments on social media or sales sites. In order to process comments, it is necessary to collect them in one place and store them in a database, then organize them, for example, separate positive and negative comments, and then analyse each of these sets in a finer way to determine what users evaluate as negative or positive, for example, a specific product model or some functionality. This information should be shared with the company's management so that they can decide what to do next. The answer to the initial question, you notice, is long and requires knowledge and work with tools for downloading content from the web (the so-called scrapers), working with databases, natural language processing, as well as data display techniques that would be most informative to domain experts. In this and other data science application examples, machine learning is an indispensable part of the knowledge pathway.

Before we go any further, let's summarize what problem solving looks like with classical programming and data-driven programming.

When we solve a problem using classical programming techniques, for example, finding the largest element in a sequence of numbers, we first think about the problem and the spatial and temporal constraints we have, then we design an algorithm to solve it (for example, merge sorting), and then we program it in a programming language and save the code. We check the accuracy of the implementation on a few random entries until we make sure that everything works exactly as we expect. When we need to sort a new string, we can use the program we wrote, run it and get the appropriate solution.

When we rely on data-driven programming, we initially have only data, for example, thousands of pairs of inputs and outputs. Again, it is wise to think about the problem first. We do this now by getting to know the dataset. This is done by the exploratory analysis techniques that we will discuss later in the course, which can give us an idea of what form to look for a solution. Then, instead of devising an algorithm to solve a problem, we **design an algorithm to learn how to solve a problem**. This would mean that if we need to learn the connection between input and output and change the data set, this algorithm can again find the best connection between them. The connection that exists between input and output depends on the data (it is not static) and therefore we need to learn it, and we don't say it directly. When we design and program an algorithm like this, we need to use the data to see how well it works. If we are not satisfied with the results, we need to take a step back and fix the algorithm or go back to the very beginning and check if there is anything else in the data that can be important for solving the problem. Unlike classical programming, this iterativeness is very present in data-driven programming.

This embedded question is incorrectly configured.

2.3 Basic concepts of machine learning

The concepts that we will introduce in this lesson are the basic concepts of machine learning. They will help you follow the topics discussed below, and you will learn more about each of them.

In terms of machine learning, the collection of data we have is called a **data set**. It can be some fine tabular records, similar to those we encounter in databases or *Excel* files, but also some group of satellite images or audio clips. One specific element of a data set is called an **instance**. So, one particular row in a log table or one particular satellite image are examples of instances. The number of instances in a data set can determine the choice of learning algorithm because some algorithms require more data than others.

In instances there are **attributes**, properties that we use to describe the data. If we imagine that it is a tabular record of earthquake occurrences, the date and time of occurrence, latitude, longitude, earthquake strength, level of destruction and other important data can appear as attributes. Attributes are equally called **features**. A little later you will learn what all kinds of attributes exist and what we need to take care of. The attributes on the basis of which we need to learn how to solve a task are called **input variables** (inputs), and those that need to be taught **output variables**. Thus, the date and time of an earthquake, its geo-coordinates and its magnitude can be input variables in the task of determining the devastation of an earthquake. The devastation of an earthquake is also present as an attribute in the data set, so it would be an output variable. Sometimes we will use less formal terms such as **input** and **output**. It is important to note that it is the task that dictates what will be input variables and what will be output variables.

What could be the attributes of a satellite image?

Answer: *You will also agree that for satellite images, we can introduce attributes such as location, date, and time of creation. We can also introduce attributes that describe the satellite that took the image. However, none of these attributes directly describe what the satellite image contains. Think about this topic until we get to the lesson that covers it.*

We have said that the goal of machine learning algorithms is to determine the mapping of given inputs to given outputs. Now we can be more precise and say that the goal of machine learning is to determine the mappings of given input variables to given output variables. These mappings are called models.

The concept that we associate with mapping is a function. In math class, you've heard a lot about functions such as input-to-output mappings. For example, the function of one variable $y = 2x + 4$ maps the input $x = 5$ to the value $y = 14$, while the function of multiple variables $y = 2x_1 - 3x_2 + x_3 + 5$ maps the input $(x_1, x_2, x_3) = (1, -1, 3)$ to the value $y = 13$. The variables that appear in the functions are related to the values of the attribute. Thus, x in the first function can represent the square footage of the property, while x_1, x_2, x_3 in the second function can represent the values of attributes such as latitude, longitude, and earthquake strength. In math class, you've heard that there are different classes of functions (linear, polynomial, trigonometric, exponential, logarithmic), and that each of them is characterized by some special properties such as continuity, monotony, or convexity. All this knowledge is welcome when looking for the right model.

The complexity of a function is something that we will not formally introduce. You will understand that some functions are simpler than othe. Simple functions are more rewarding for work and easier to understand, but they do not give us much freedom to describe some of the more unusual relationships between the attributes themselves and the outputs. On the other hand, complex functions are complex for a reason, so it can be difficult for us to keep track of some of their mathematical behaviors that can affect learning. We try to strike a balance between complexity and what we know about data and what we want to learn.

In models, as we have seen in the introductory example of real estate pricing, **parameters** such as k and n can appear. Such models are called **parametric models** and the task of determining the right model is reduced to the task of determining the best values of the parameters. In the linear model, only two parameters appeared in the real estate pricing task, while modern models, those that are based on neural networks have millions or billions of parameters. We will see that there are also slightly different **non-parametric models,** whose forms are expressed differently.

The process of finding a model is called **model training**. If there are unknown parameters in the model, we need to determine their values during training. That is our goal.

In the dataset used to train models, inaccurate, or contradictory values can also be found. That is why models are never absolutely correct. This brings us to another important concept in machine learning theory: the **loss function**. The error function tells us how wrong the model is. We actively use its values during model training and strive for those model configurations that lead us to the lowest value of the error function. In the case of parametric models, as was the case in the introductory example with real estate, the goal is to determine those parameter values for which the value of the error function is the lowest.

When we train a machine learning model, we need to assess how good it actually is for application in practice. This is what the so-called **quality measures serve** us - each of them is adapted to a specific learning task and the domain in which the model will be applied. It is important to emphasize that, in general, the error function and quality measures differ. Both aim to give us information about how good the model is, The error function does this during model training, while quality measures do this after model training. The error function is closely tied to the model, while quality measures are designed so that they can be understood by both users and domain experts. If the correct quality values are not obtained, the model must be repaired. Below, we'll talk about what that means and how it can be achieved. The whole process of testing the quality of a model and calculating its quality measures is called **model testing**.

Typically, the values calculated and generated by a trained model are called **predictions**. Thus, the price of a new property or the assessment of the devastation of an earthquake are examples of model predictions. This is why we are talking about predictions in the world of artificial intelligence. It is clear to you that these predictions are by no means random, but very well-founded and data-based. The application of the model itself is also called **inference**.

All the terms that are emphasized are important concepts of machine learning and are always present in the literature on machine learning and its applications. That is why it is important that they are clear to you and that you understand what role they play in the development of a model.

2.4 A machine learning process

The machine learning process is a systematic approach to developing algorithms that enable computers to learn from data and make predictions or decisions without being explicitly programmed. This process encompasses several stages, each critical to building effective machine learning models. Understanding these stages helps practitioners navigate the complexities of machine learning and enhances the likelihood of successful outcomes.

1. Data Collection

The first step in the machine learning process is data collection, which involves gathering relevant data that will be used to train the model. The quality and quantity of the data collected directly influence the model's performance. Data can be sourced from various origins, including:

- **Public Datasets**: Repositories like Kaggle, UCI Machine Learning Repository, and government databases offer pre-collected datasets for various applications.
- **Web Scraping**: Automated tools can extract data from websites to gather information not readily available in structured formats.
- **Surveys and Experiments**: Custom data collection through surveys or controlled experiments can yield specific datasets tailored to a particular problem.

The outcome of this step is a coherent dataset that represents the problem domain, which will serve as the foundation for subsequent steps in the machine learning process.

2. Data Preparation

Once data is collected, it must be prepared for analysis. This preparation phase involves several critical tasks:

- **Data Cleaning**: Raw data often contains errors, duplicates, or missing values. Cleaning the data is essential to ensure accuracy and reliability. Techniques include removing duplicates, correcting inconsistencies, and filling in or removing missing values.
- **Data Transformation**: This step may involve normalizing or standardizing data to ensure that all features contribute equally to the model's performance. Data types may also need conversion (e.g., converting categorical variables into numerical formats).
- **Data Splitting**: The dataset is typically divided into training and testing sets. A common split ratio is 80% for training and 20% for testing, ensuring that the model can be evaluated on unseen data.

Data preparation is often one of the most time-consuming aspects of the machine learning process but is crucial for developing an effective model.

3. Feature Selection and Engineering

Feature selection and engineering are pivotal steps that determine which attributes of the data will be used in training the model:

- **Feature Selection**: This involves identifying the most relevant features that contribute to predicting the target variable. Techniques such as correlation analysis, recursive feature elimination, or using algorithms like LASSO can help in selecting important features while discarding irrelevant ones.
- **Feature Engineering**: In this phase, new features may be created from existing ones to improve model performance. This could include polynomial features, interaction terms, or aggregating features based on domain knowledge.

Effective feature selection and engineering can significantly enhance a model's predictive power by ensuring that it focuses on the most informative aspects of the data.

4. Model Selection

With prepared data and selected features, the next step is choosing an appropriate machine learning algorithm. The choice of algorithm depends on several factors:

- **Type of Problem**: Different algorithms are suited for different tasks—classification (e.g., logistic regression, decision trees), regression (e.g., linear regression), clustering (e.g., k-means), or reinforcement learning.
- **Data Characteristics**: The nature of the data (e.g., size, dimensionality) influences algorithm choice. For instance, deep learning models may be preferred for large datasets with complex patterns.

Selecting an appropriate model lays the groundwork for effective training and evaluation.

5. Model Training

Model training involves feeding prepared data into a selected algorithm to allow it to learn patterns within the dataset:

- **Training Process**: During training, the algorithm adjusts its internal parameters based on input features and corresponding target outputs. This iterative process continues until a stopping criterion is met (e.g., a specified number of epochs or convergence).

- **Evaluation Metrics**: It's essential to define success metrics before training begins. Common metrics include accuracy for classification tasks, mean squared error for regression tasks, and F1-score for imbalanced datasets.

Training aims to develop a model that generalizes well to new data rather than simply memorizing the training set.

6. Model Evaluation

After training, evaluating the model's performance on unseen test data is crucial:

- **Testing**: The model is assessed using the test dataset that was not involved in training. This evaluation provides insights into how well the model generalizes to new instances.

- **Performance Metrics**: Depending on the problem type, various metrics can be employed:
    - For classification tasks: accuracy, precision, recall, F1-score.
    - For regression tasks: R-squared, mean absolute error (MAE), mean squared error (MSE).

This evaluation helps identify areas where the model excels or needs improvement.

7. Hyperparameter Tuning

Hyperparameter tuning involves optimizing parameters that govern the training process but are not learned from data directly:

- **Grid Search and Random Search**: These techniques systematically explore combinations of hyperparameters to find optimal settings that enhance model performance.

- **Cross-Validation**: Implementing cross-validation during hyperparameter tuning helps ensure that performance improvements are consistent across different subsets of data.

Fine-tuning hyperparameters can lead to significant improvements in model accuracy and robustness.

8. Deployment

Once a satisfactory model has been trained and evaluated, it can be deployed into production:

- **Integration**: The final model needs to be integrated into existing systems where it will provide predictions or insights based on real-time data.

- **Monitoring**: Continuous monitoring post-deployment is essential to ensure that the model maintains its performance over time as new data becomes available.

Deployment marks the transition from development to practical application, emphasizing real-world utility.

Conclusion

The machine learning process encompasses a series of structured steps that guide practitioners from initial data collection through deployment of predictive models. Each stage—data collection, preparation, feature selection, model selection and training, evaluation, hyperparameter tuning, and deployment—plays a vital role in developing effective machine learning solutions. By understanding this process thoroughly, practitioners can improve their ability to create robust models that deliver valuable insights across various applications in today's data-driven world.

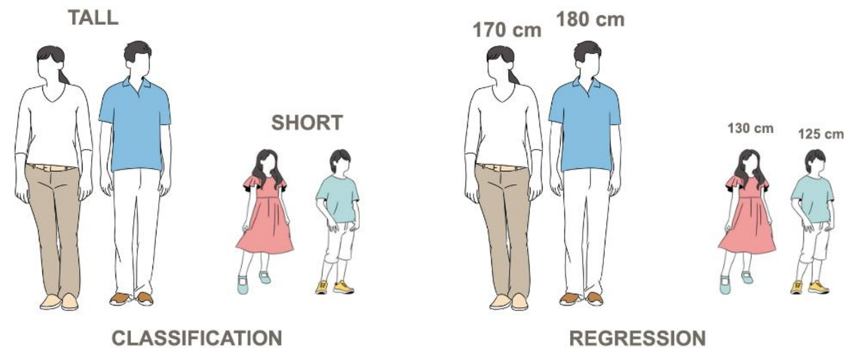
2.5 Types of machine learning

In this lesson, we will introduce the basic types of machine learning: supervised machine learning, unsupervised machine learning, and reinforcement learning. All of them are hidden behind tasks such as predicting precipitation, recommending movies to watch or playing games. Each of these areas will be discussed in more detail later in the course. We will also talk about some current areas of research, such as learning through the transfer of knowledge.

Supervised Machine Learning

Most of the examples we've seen so far are actually examples of **supervised machine learning**. Supervised machine learning involves algorithms that fit perfectly with what we've been talking about so far and help us learn how to map one set of values to another. Therefore, it is necessary that in the data set to which they are applied, in addition to the values of the attribute, we also know the values of the target variable.

The two main tasks of supervised machine learning are **regression** and **classification**. In both regression and classification tasks, we want to learn to predict values, but in regression tasks, values can be arbitrary, and in the case of classification, they can be from a predefined finite set of values. Thus, regression tasks are suitable for predicting the value of temperature, the price of a product (the task of determining the price of real estate that we encountered in the introductory part is an example of a regression task), the devastation of an earthquake, and the like. On the other hand, determining whether mail is undesirable or desirable or determining the genre of a film are classifying tasks because the set of values we have on the other side is finite - mail can be either desirable or undesirable (two values) while a genre can be, say, comedy, drama, action, or thriller (four values). A little later, we will introduce a more precise definition of each of these tasks.



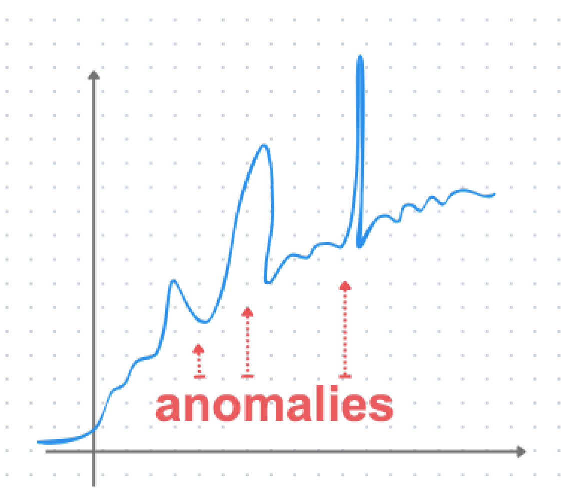
Classification and regression. Determining whether someone is tall or short is the task of classification. Determining the exact height is the task of regression.

Unsupervised Machine Learning

We use unsupervised machine learning in tasks that need to examine the structure of a data set. For example, if we analyse the purchases of consumers of one store, it can be interesting to notice products that are often bought together in order to distribute them more finely in the store, improve the offer, but also profits. In the same way, user comments can be analysed and grouped and the services or features that users are talking about can be insighted. Tasks of this type, in which we want to see groups among the data, are called **clustering**. Later in the course, you will learn about the k-mean algorithm, the most well-known clustering algorithm.

Clustering

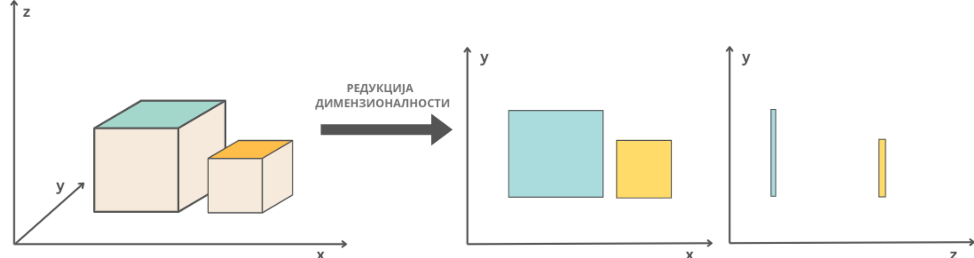
Spotting instances of data that are somehow different from others also falls under the tasks of unsupervised machine learning. Thus, spotting atypical measurements of a factory's sensors can be a signal to initiate additional security procedures. Similarly, spotting atypical banking transactions, for example, from a remote location or in some unusual amount, can be hints of fraud. This task of unsupervised machine learning is called **anomaly detection**.



Anomaly detection

Unsupervised machine learning also deals with the tasks **of reducing dimensionality.** Often, for the purposes of graphical representation of data, we need to move from a larger number of attributes to a smaller number of attributes, for example, two or three. It is clear that during this transformation, some information from the initial data set is lost, but, on the other hand, the ability to display data and perhaps a better insight into some regularities is gained. Smaller dimensionality of data (fewer attributes) is desirable and because of the faster execution of algorithms and less memory complexity, which can be especially important if we have

limited resources to work with. Some of the most commonly used algorithms for dimensionality reduction are principal component analysis. *principal component analysis (PCA*) and t-SNE.

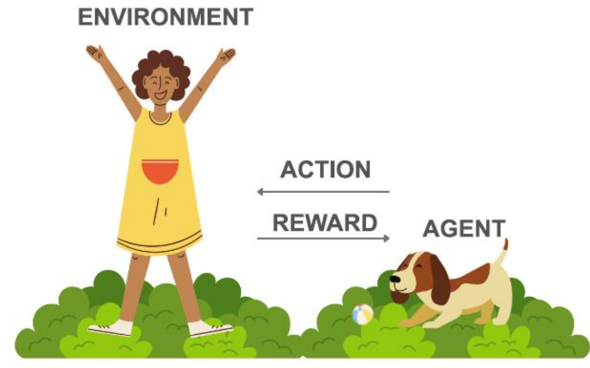


The Meaning of Dimensionality Reduction: Two Cuboids and Their Projections from Three-Dimensional to Two-Dimensional Space

Interestingly, in unsupervised machine learning tasks, it is not necessary to know the values of the target variable. Clustering, anomaly detection, and dimensionality reduction are performed only on the basis of attribute values.

Reinforcement learning

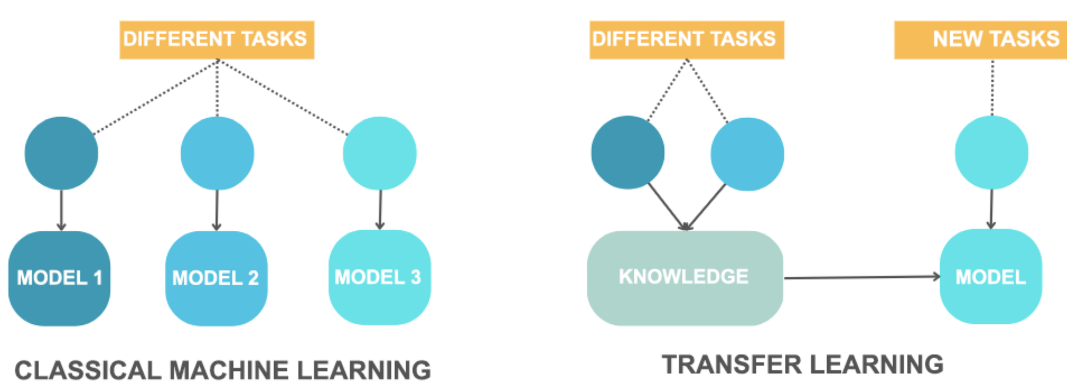
Surely you have seen many times how to train a dog. When he is given a task, for example, to bring a ball from the other end of the yard, the reward in the form of a cookie when he brings it will motivate the dog to perform this task even more successfully and joyfully next time. This idea also lies at the basis of learning by reinforcement. **Reinforcement learning** is an area of machine learning used in tasks such as playing games or driving autonomously. It is characterized by the existence of an environment that has its own states, an agent that can perform a certain set of actions and the concept of reward. The goal is for the agent in a given environment, whose states are changing, to choose (learn) the sequence of actions that allows him the greatest reward. In the context of the introductory example, the yard is the environment. Its states can be a ball at the end of the yard or a neighbour's cat in a tree. A dog is an agent, and the set of actions it can take is to run, to sit, to go to sleep. The reward can be a number of cookies or nothing. If the dog chooses the right sequence of actions (run, find it and return it) to a change of environment, for example, the appearance of a ball, he will be able to win the biggest prize.



You will learn more about this type of learning at the end of the course.

New directions of learning

When we need to master a new task, for example, to learn to ride a scooter, we don't start from scratch. All the knowledge and skills we have acquired in some other tasks, for example, playing basketball, cycling, and even perseverance and patience in tasks that were not our favourite, such as tidying up the basement, help us to master it better. This idea is the basis of **transfer learning**. That's why you can often hear people talking about models that have been used as the basis for the development of some other model. Such models are first trained on some general data sets and tasks, and then retrained, i.e. They can also be used to solve a very specific task. For example, the *GPT* language model was used as a basis for the development of *the ChatGPT model*, which had previously performed well in the tasks of generating summaries, shortened versions of text, and answering questions.



The idea of learning through the transfer of knowledge

Knowledge transfer techniques can be combined with all the above-mentioned types of learning. They are especially important to us when the training datasets for a specific task are not large enough or when we are developing a model for a specific domain.

2.6 Data in Machine Learning

In the AI community, you often hear two sayings: "data is the new gold" and "garbage in, garbage out." These remind us of the value of data for understanding and modeling phenomena and the importance of creating high-quality datasets. Let's explore these topics.

Today, almost all activity domains generate large amounts of data: information about videos we've watched online, products we've purchased, friends we've connected with on social media, as well as information about doctor visits, weather conditions in our city, or traffic conditions recorded by relevant institutions. All this data can be used to better understand the environment in which it is generated.

Just like in the story of databases, in machine learning we describe important entities and events whose behaviours we want to model using attributes (also called features). For example, a movie can be described by its title, genre, year of release, production company, budget, profit, synopsis, director's name, and main actors' names. Choosing the right attributes to track and record when collecting data is not an easy task because we don't know in advance which attributes will be most useful for the task we want to solve in the future. For instance, if we want to use data to predict a movie's profit (a regression task), information about

actors and the production company might be more useful, while for determining the movie's genre (a classification task), the synopsis might be more helpful. In more complex domains, these choices come with even more dilemmas and challenges.

Due to the need to use data for a wide range of applications, we might consider collecting as many attribute values as possible. While this idea is valid in some situations, generally, we must remember that large amounts of data require appropriate storage, hardware to support their processing, and a team of experts with the necessary skills and knowledge to perform these tasks. Therefore, such choices can be costly and require special planning. It's also important to note that analysing and understanding large amounts of data is challenging and requires appropriate technical competencies, such as data visualization techniques. Additionally, many domains involving private and sensitive data must consistently follow regulations and ethical guidelines on data collection, which imposes further restrictions on attribute selection and storage possibilities. Thus, the task of collecting data and creating high-quality datasets is challenging and demanding, requiring careful organization.

In the upcoming lessons, we will see that each attribute is defined by its type and value set, and these properties affect how we prepare the data. Ultimately, machine learning algorithms can only be applied to numerical values. The number of attributes and their properties also influence the choice of machine learning algorithm.

Advanced machine learning algorithms, such as neural networks, can identify important attributes for solving a task on their own. This relieves us from thinking about attribute selection and combinations. This is particularly useful when working with complex data like images or textual content, where defining and extracting attributes is not always intuitive. These algorithms can work with raw data.

- What do you find challenging about data collection in the domain that interests you? It could be sports, a scientific discipline, a social phenomenon, or anything else.
- Do you have any concerns or reservations about data collection and processing?
- What is most important to you personally in the data collection process?

Popular datasets

It may surprise you, but datasets can be popular too! Some of them are known for being used in the first machine learning tasks, while some have achieved their popularity through persistent community engagements to expand and complement them. As different datasets track different domains of AI, here we will use this as a criterion for grouping and displaying them. Namely, we will get to know sets that contain images, textual data, audio archives and videos. A large number of libraries used in the field of machine learning make it possible to quickly and easily load the sets we are going to discuss.

Computer vision

MNIST

Certainly, one of the most popular sets in the field of computer vision is **MNIST**, a set of images of handwritten numerals. Its development was started by the US National Institute of Standards and Technology *(NIST)* back in 1998. All images are 28x28 pixels, black and white, and there is a total of 70,000 of them: 60,000 images make up the training set and 10,000 images make the test set. In the image you can see some of the digits from this data set.

Some of the figures of the MNIST conference

The MNIST set is used to train multiclass classifiers, most often in combination with convolutional neural networks, which you will hear more about later in the course.
For each digit of the MNIST set, one class is provided. Think about which digits are potentially problematic to distinguish (for example, the digits 1 and 7 may resemble each other), and then try to find some examples on the web.

ImageNet
The images in the **ImageNet** set represent images of general objects: computers, windows, airplanes, seedlings, tropical animals and various other entities. Interestingly, these images are organized into related groups (so-called synsets) between which the parent-child relationship applies. For example, all sailing ships belong to one group (one synset), in the hierarchy below them there are groups of gliders and trimarans, while in the hierarchy above there are groups of watercraft, vessels and vehicles. In the picture, in the bottom row, you can trace this hierarchy: at the bottom of it are the trimarans, and at the top of the vehicle. In the top row are synsets that refer to dogs and some of their categorizations.

Example of an ImageNet image

The collection currently contains about 14 million images and over 21,000 synsets. It is used in a variety of image classification and object detection tasks in images.
The official website of the ImageNet conference is https://www.image-net.org/index.php. Researchers from Stanford and Princeton universities are actively working on its development.

Try to find out which group a computer belongs to in the ImageNet set and which groups are in the hierarchy below and above.

COCO

The **COCO** dataset (acronym for *Common Objects in Context*) is used in tasks of object detection, image segmentation, and automatic association of titles to images. It was created by Microsoft and shared with the community in 2015.

An image of the COCO set with marked recognized objects: planes, trucks and cars

The set can be viewed interactively on the official website: for each image there is a URL from which the image was taken, several titles associated with the image, and then a series of icons corresponding to the recognized objects. The number of images in the dataset is 330,000 and contains 80 categories of objects with over 1.5 million instances. The link to the search section on the site is https://cocodataset.org/#explore.

Natural Language Processing

IMDB

If you like to watch movies and TV shows, you will be interested in **the IMDB** dataset, which contains user reviews from the popular IMDB platform. For each view in this dataset, it is also known whether it is positive or negative, i.e. whether it primarily contains something praiseworthy and good about the film or some criticism and objection. When it comes to datasets that contain textual content, it is always important to emphasize in which language they are written. The IMDB dataset contains views that are in English with a total of 50,000 views, 25,000 positive and 25,000 negative views. Below you can see a positive and a negative entry in this data set.

Review	Sentiment 0-negative 1-positive
Don't even ask me why I watched this! The only excuse I can come up with that I was sick with Bronchitis and too weak to change the channel. :) It's too terrible for words, the movie that is, not the Bronchitis.	0
this movie is the best movie ever it has a lot of live action It's just great everyone should watch it and the actor are great the location is Rome Italy thats the best place ever the actors are great	1

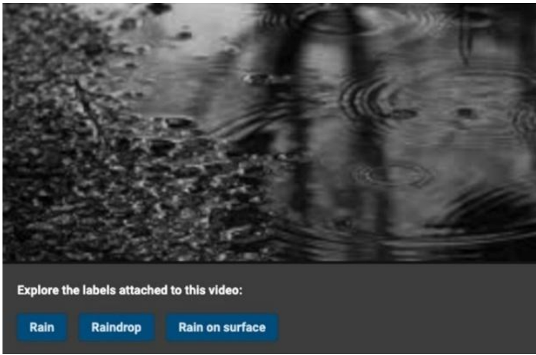
Examples of positive and negative reviews of the IMDB set

The IMDB dataset is used in sentiment analysis tasks - recall that these are tasks in which it is necessary to recognize an emotion or attitude present in the text. Since the set contains only information whether the review is positive or negative, the task of sentiment analysis in the IMDB set is approached as a problem of binary classification. In general, the sentiment scale can be finer and includes ratings such as very positive, positive, neutral, negative, or very negative.

Sound processing

AudioSet

An AudioSet is a dataset that contains 10-second snippets of video from YouTube. Each of these snippets is associated with the characteristics of the sounds heard in them. The set was created by Google and contains over 2 million clips with a total duration of 5.8 thousand hours.



An example of a video clip with the associated audio annotations it contains

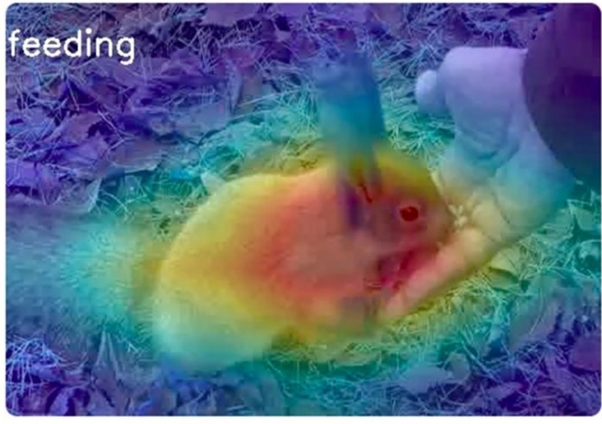
The official website of the conference provides an overview of examples and insight into the organization of the conference. 632 different categories are used, such as the sounds of musical instruments, the sound of the wind, the sound of man, noise, etc. You can visit the address https://research.google.com/audioset/index.html and listen to some more examples. The conference itself was created with the idea of supporting the development of sound recognition algorithms.

Video Processing

Moments in Time

Moments in Time is a dataset that is being developed with the idea of helping artificial intelligence systems learn to recognize actions and events. This set currently contains one million videos of 3 seconds in length in which activities are marked. The videos contain people, animals, objects and natural phenomena. Just some of the events that are covered are dancing, exercise, climbing a tree, jumping into the water and sleeping.

The Moments in Time gathering is being developed by a team from the Massachusetts Institute of Technology (MIT) and on the official website of the project you can see some more examples of videos and recognized actions. The link to the official website is http://moments.csail.mit.edu/.



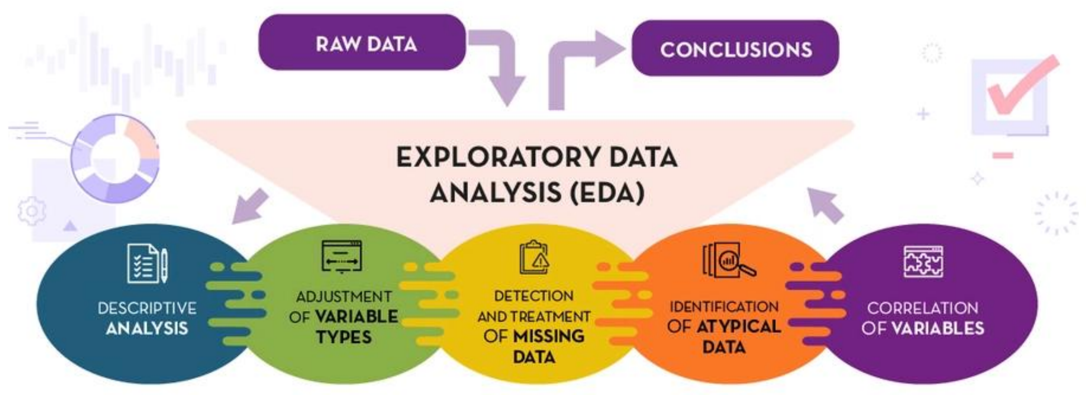
A video in which a man is recognized as feeding a rabbit

© 2025 https://book.tsp.edu.rs

2.7 Exploratory Data Analysis (EDA)

Encountering a new set of data is like a trip to a new place. You need to research it carefully, find out where everything is and what connections exist between the different parts. In this section, you will learn a few techniques that will help you in our adventure with data.

We start every machine learning task by getting to know the data set. If we use tabular data, we are interested in what attributes appear, what values they have, and whether some of them may be related. When we work with other types of data, such as text, we are usually interested in whether all the texts are written in the same language and how long they are. Since no data set is perfect, in the analysis, we try to find potential duplicates and some atypical entries. All of these tasks are called **exploratory data set analysis.** *Exploratory Data Analysis* (EDA). Its goal is to help us, through a diverse set of tasks, to get to know the data set better and to make further decisions regarding data preparation more informed. Given the importance of data in the next steps (remember the saying "garbage in, garbage out"), we try to devote enough time to exploratory analysis of the dataset and move to the next step only when we are sure that we understand the data.

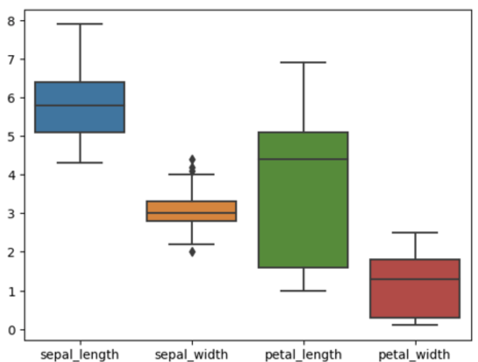


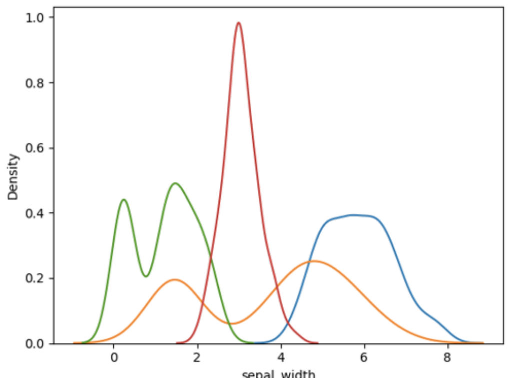
Exploratory Data Analysis Tasks

Attribute analysis

Since attributes are used to describe a wide variety of properties, their types and ranges of values vary. The two large groups of attributes that we encounter are **numerical** (quantitative) and **categorical** (qualitative) attributes. Numerical attributes have, as the name implies, numerical values. Such are, for example, the height of the player, the distance from the airport, the number of pets, the outside temperature, the number of ice creams sold, the concentration of glucose in the blood and many others. For these attributes, we usually look at the value ranges, the highest and lowest values, the average value, the median, as well as the distribution itself. We call all these **analyses - descriptive analyses,** because they help us describe the quantity to which an attribute is associated.

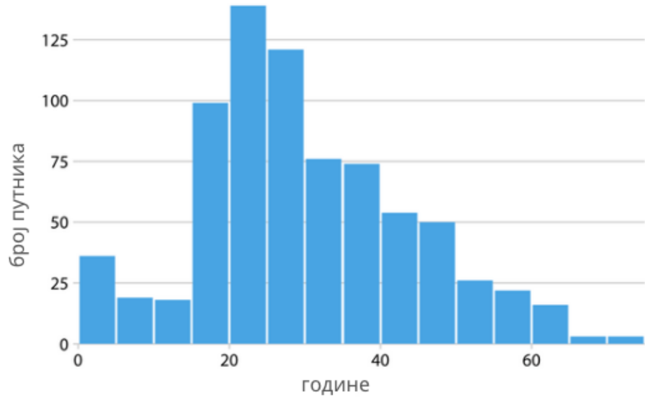
© 2025 https://book.tsp.edu.rs





Examples of some descriptive analyses of the attributes of the Iris dataset

Categorical attributes are a type of attribute that can have a finite set of values. Such attributes are, for example, the color of the car, the type of clothing, the sex of the patient, the current season, and others. These attributes are usually represented by strings or equivalent numeric codes. For example, the month of the year can be listed as the name "February" or as the number "two" (because February is the second month of the year). It is important to note that even if we use numeric codes to represent these attributes, it makes no sense to calculate values such as average or maximum because these values are not inherently numeric. For them, we usually analyse what values they can take and how often they appear, and we show these conclusions using graphs.



An example of the analysis of the attribute "year" in the set Titanic

The Unification of Values

In the course of analyzing the data, we can find that the attribute values are not uniformly set. For example, color names may be spelled inconsistently, sometimes in lowercase and sometimes uppercase letters, or dates may be given in different formats such as day-month-year and year/month/day. In order to be able to carry out the task of analysis correctly, it is desirable to unify these values, i.e. Let's reduce them to the same way of representation. Usually there is a way that is more desirable or useful, but it also happens that the elections are completely equal.

Missing values

When analysing a data set, we may notice that the values of some attributes are missing. This may be due to carelessness in data entry or simply unavailability of information. Such values in the data set are called missing values.

name and surname	year	work experience	revenues
Marco	48	22	83
Ivan	67	30	110
Ana	34	10	
Peter		4	
Milan	21		85

An example of a set with missing values

The simplest step we can take when we notice missing values is to delete either the attributes (columns of the data set) or the instances (types of data set) in which they appear. For example, if we don't know the value of an attribute for more than 50% of instances of a data set, it makes sense to delete it. If, on the other hand, we have only a few instances in which the value of the attribute is missing, it is best to delete the instances and keep the attribute. However, these decisions are not always easy. For example, it can happen that different attribute values are missing in different instances, so we delete and ignore a significant number of instances in this way, which can be problematic if we do not have a large set of data. That is why it makes sense to consider some more options in working with missing values.

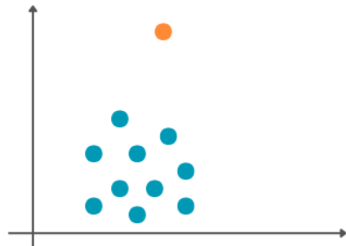
If the missing attribute is numeric, for example, the distance to the airport or the height of the player, we can replace the missing values with the average value of the known values. The argument we have for this choice is that we will use the information that already exists in the dataset and that we will not change much about some other attribute properties. On the other hand, if we are talking about categorical attributes, such as the color of the car or the country of manufacture, which can have a finite set of values, we can replace the missing value with the most common value. Another option that is valid for both numeric and categorical attributes is the use of random values - so we can replace the missing color with a random color from a possible set of colors, and the missing height of a player with a value from the range of the smallest and highest height in the set. In all cases, we must be careful because changes in the data can affect the success of the model and the results we obtain. It is also very important at what point we make these repairs. We'll talk about that later on.

Duplicates

The presence of duplicates in the dataset can affect the generalization power of the model. That is why it is always convenient to check whether there are any data that are repeated or very similar. When it comes to tabular data, duplicates can be found by directly comparing attribute values. When working with different types of data, we usually need more advanced techniques. For example, duplicate images can be symmetrical, like in a mirror, either horizontally or vertically. It's the same with textual data. Two news stories may contain the same announcement (reported by some news agencies) with slightly different headlines, so in terms of direct character comparison, they are different, yet the same.

Spotting exceptions

Noticing data that is in some way different from the rest allows us to spot errors in the data or discover new, atypical behaviours. Such data is called exceptions or *outliers*. The distance from the airport, which is -1.2 km, would be a discrepancy figure because we expect the distance to be a positive value. That way we could spot the mistake and correct it. On the other hand, a temperature of 45 °C is also an unusual value, but a real one due to climate change and perhaps very useful as information for taking certain steps and actions.

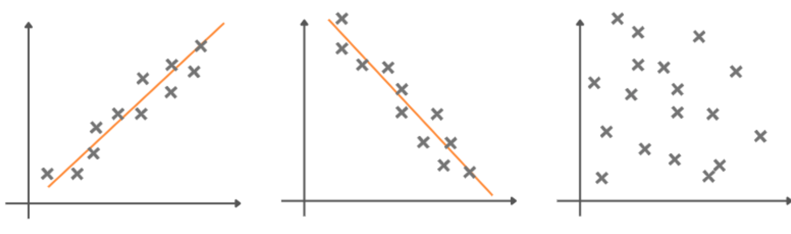


Graphical representation of the discrepancy

Discrepancies can also affect the outcome of machine learning algorithms. That is why, once they have been spotted and processed, it is important to decide whether they should be kept or deleted.

Attribute correlation

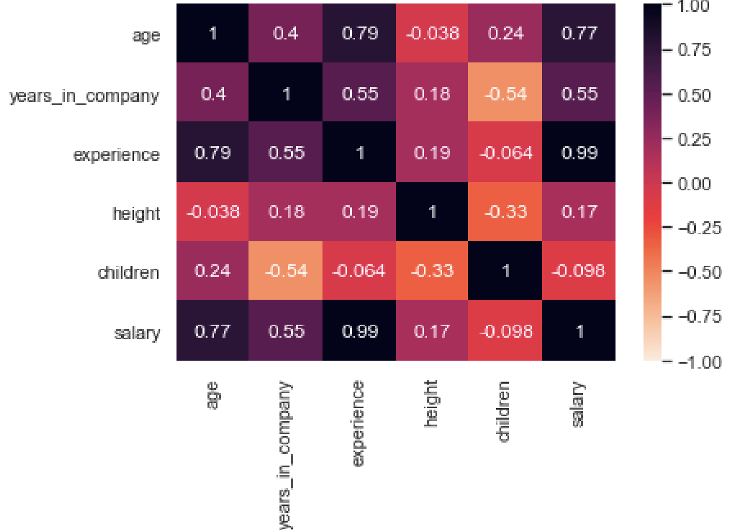
Attributes can be related to each other. We can see the connection if we draw a graph that has the value of one attribute along the x-axis and the value of another attribute along the y-axis. For example, we can track pairs of attributes outdoor temperature and the number of ice creams sold, outdoor temperature and electricity consumption and outdoor temperature and the number of books in the library. Let each of these pairs correspond to a graph like in the picture below. We can notice that the increase in temperature is accompanied by an increase in the number of sold Ice cream. If the increase in the value of one attribute follows the increase in the value of another attribute, we say that they are **positively correlated**. On the graph we can also notice that this dependence is linear, i.e. that it follows an imaginary line that passes through a set of points. On the other hand, it seems that the situation with the outside temperature and the consumption of current is somewhat different, i.e. that the decrease in temperature is accompanied by a higher consumption of electricity, probably due to the use of heaters. Attributes in which an increase in the value of one attribute is followed by a decrease in the value of another attribute are said to be **negatively correlated**. From the graphic, we can conclude, again, that this kind of correlation is linear. The third graph, which shows the external temperature and the number of books in the library, does not indicate any, at least not obvious, regularity between the attributes. We can certainly conclude that these attributes are not linearly correlated.



Attribute association graphics

To measure the linear relationship of attributes, we can also use different types of coefficients that are established in the domain of mathematical statistics. One such coefficient is Pearson's correlation coefficient. Its values range from -1 to 1 and indicate both the direction and the strength of the connection. Coefficient values closer to -1 indicate negative correlation, coefficient values closer to 1 indicate positive correlation, and values around zero indicate the absence of linear correlation.

It is common for the values of correlation coefficients between attributes to be displayed graphically in the form of a so-called heat map. Each square in this map corresponds to one pair of attributes and its color is adjusted to the value of the correlation coefficient. The column located on the side of this map connects the values and shades of colors. By observing this map, we can easily see the correlations in the data. The figure below shows the pairs of attributes of one data set that combines information about the Employees. Although we know little about this set, we can conclude that experience and the number of years (age attribute) best track *salary values*. We can also see that there is a correlation between the number of years (the age attribute) and the experience attribute.



Heat map with values of the correlation coefficient

Spotting the attributes that are related allows us, first of all, to better understand the domain to which the data refers. Some connections can be expected, while others can bring us new knowledge. By deleting attributes that are linked, we can reduce the dimensionality of the data set. In this way, we can speed up the work of some algorithms and understand the results more easily. There are also machine learning algorithms that don't behave well if there are associations in the dataset - deleting attributes for which this applies can improve the success of the algorithm.

This lesson is paired with the Jupyter notebook 03-exploratory-data-analysis.ipynb. If you want to practice the tasks we have described, click on the link and then on the button Open in Colab to open the content in *the Google Colab environment*. If you view the notebooks on your local machine, find the notebook with the same name among the contents and run it. For more detailed instructions, see the *Hands-on zone* section and the lesson *Jupyter exercise notebooks*.

In the Jupyter notebook, using the functions of the *Pandas library*, the data of the Titanic set was analysed. This set contains information about passengers who were on the famous ship Titanic when in 1912, sailing in the Atlantic Ocean, it hit an iceberg and was shipwrecked.

2.8 Creating a Representation of a Dataset

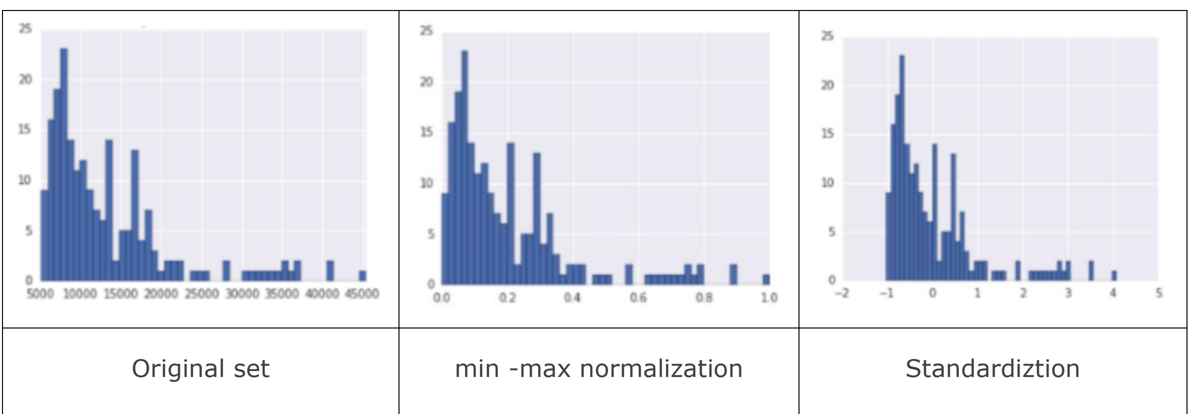
After exploratory analysis of the dataset, we can make decisions about which attributes and instances to discard. The remaining dataset needs to be prepared so that we can apply some of the machine learning algorithms to it. Depending on the type of attribute and the set of values it contains, you can read about some preparation techniques below. In the next lesson, you'll learn when it's the perfect time to do it.

Preparation of numerical attributes

When working with numerical attributes, we encounter quantities that are expressed on different scales of values. For example, in a single set of medical data, there may be laboratory analyses with values ranging from 0 to 1 and information about the patient's height expressed in centimeters, from 100 to 250, which is significantly higher. Many machine learning models are sensitive to the presence of these attributes and therefore take longer to find a solution. It's not easy to understand the results that come with this kind of work. That's why when working with numeric data, we use normalization techniques that help us bring a set of attribute values to the same ranges of values.

One such normalization is *min-max* normalization. To clarify how it is performed, suppose that we need to normalize the value of the attribute X to express the height of the patients. Let Xmax = 180 represent the maximum height of the patients and Xmin = 110 the smallest. The normalization of *min-max* is carried out by applying the formula (x − Xmin)/(Xmax − Xmin) to each value of the attribute x. If x = 165, the new normalized value will be x′= (165 − 110)/(180 − 110) = 0.786. In this way, the value of the attribute is reduced to a range from 0 to 1.

One of the most important aspects of normalization is standardization. It involves centering the attribute value around zero and scaling to unit variance. To clarify how it is carried out, we can again assume that we need to normalize the value of the X attribute, which expresses the height of patients. Let now Xmean = 153.2 the mean height in the data set and σ = 40.23 the standard deviation. Standardization is accomplished by applying the formula (x − Xmean)/σ to each value of the attribute x. The new standardized value for a patient whose height is x = 165 is now x′= (165 − 153,2)/40.23 = 0.293.

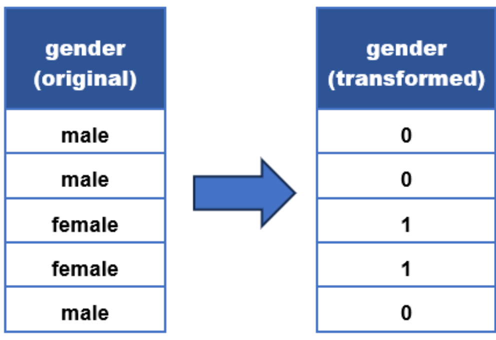


Original set	min -max normalization	Standardiztion

The Effect of Normalization and Standardization on a Data Set

Preparation of categorical attributes

Since machine learning algorithms can only be applied to numbers, categorical attributes require special preparation. We have said that they represent quantities that have a finite number of values and that they often appear in the form of string. Some of the examples we have mentioned are the name of the color, the sex of the patient, and the month of the year.
If an attribute has only two values, for example, representing the sex of a patient, its values are usually mapped to the numbers 0 or 1. For example, the value "female" can be mapped to the number 1, and the value "male" to the number 0. These attributes are otherwise called binary attributes.



gender (original)
male
male
female
female
male

gender (transformed)
0
0
1
1
0

Example of value mapping

For attributes that can have multiple values, we use *one-hot* coding. To clarify its meaning, we can look at an attribute that represents a color, which can have three values: red, yellow, and green. The idea is to represent the default color attribute using three new attributes, each of which will correspond to one of the values that the color can take: red, yellow and green (look at the picture, this was a complicated sentence). This further means that we will transform each of the values of the initial attribute into a triplet of values, namely the value *of red* into a triplet of *1, 0, 0*, the value *of yellow* into a triplet of *0, 1, 0, and* the value *of green* into a triplet *of 0, 0, 1.* The triplets, as we can see, consist of zeros and exactly one unit in the column that corresponds to the value of the attribute.

COLOR
red
red
yellow
green
yellow

RED	YELLOW	GREEN
1	0	0
1	0	0
0	1	0
0	0	1
0	1	0

Example of one-hot mapping

Representation of the dataset

After the step of transforming attributes, we arrive at the final form of data that we can use to run learning algorithms. This final form is called **the representation of the data set.** In the story so far, we have covered, first of all, how to arrive at the representation of tabular data. And for all other types of data such as images, audios, text, video-content, but also complex structures such as graphs, we need to create appropriate representations. In the section on neural networks, we will learn about some other ways of creating representations.

2.9 Training, validation and testing sets

In this lesson, you will learn about training, validation and testing sets. You can think of a training set as literature from which a machine learning model learns, while you can think of a test set as a control task that checks how well the model has learned and understood the required material. A validation set is used in the process of learning a model, and you can think of it as a set of auxiliary tests that check how ready the model is for control and based on the results of which the model can improve the way and success of learning.

After the data is analysed and the appropriate instances and attributes are selected, the set of all data is divided into **a Training set** (trainable) and **Testing set**. As you can guess from the name of the set, the training set is used to train the machine learning model itself. A selected algorithm is applied to it and creates the model itself. A test suite is used to test the model, i.e. calculation of suitable measures of model quality. Thanks to him, one can objectively assess how well the model has learned the necessary task. Typically, we also use a portion of the baseline dataset data to create **a validation set**. A validation kit is used to track the process of training a model and determine some model configurations that lead to better quality measures. These topics will be discussed later in the course.

Dividing a dataset into a training set, a validation set, and a testing set

Training, validation, and testing sets are typically created by randomly dividing an underlying dataset. First, we define how large these sets should be, and then randomly select the instances that will be found in each of them. Typically, the training set is the largest, while the testing and validation sets are smaller because we want to have enough data to train the model, but also enough data to adequately evaluate its performance. The practice is that the size ratios of these sets are expressed in proportion. For example, you will often find the ratio of the training set to the test set expressed as 2:1, which would mean that two-thirds of the starting set is a training set and one-third is a test set. Similarly, a ratio of 2:1:1 would mean that two-quarters (i.e., one-half) of the baseline set would be used as a training kit and one-quarter each as a validation and testing set.

While it's convenient that train, validation, and testing sets are created randomly, it would still mean some insight into how this division was done. For example, when we want to replicate an experiment or allow others to run it on their own (this is an important property of experiments and is called reproducibility), it's preferable to use the same training sets, validation and testing. Similarly, when we solve a problem, we are not immediately sure what is best to do, so we try a larger number of algorithms and create a larger number of models. For the sake of fairness of comparison, it would mean that we create all models over the same training set and evaluate over the same test set. That's why it's a good idea to set a parameter at the level of the library that affects the randomness of division (usually called *random seed* and has the same purpose as setting seeds in a random number generator) or simply split the data from scratch and continue to use it

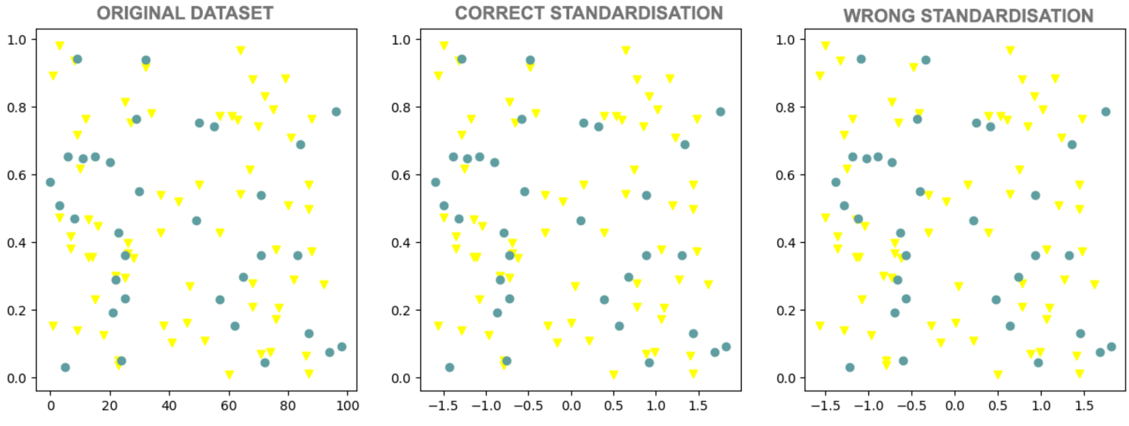
consistently. Some commonly used datasets have these predefined divisions into a set for training, validation, and testing (for example, you can look at the *MNIST* set).

An important property that train, validation, and testing suites need to meet is that they be disjointed. This means that each instance of the source dataset when creating training, validation, and testing sets must belong to exactly one of these sets, and there must be no intersections and shared instances. It should be remembered that machine learning models are expected to generalize well, i.e. that they behave well for new instances that the model has not had the opportunity to meet in the training set. If the training sets and test sets overlap, we will not be able to objectively assess whether the model actually learns or memorizes. remembers information from the training set. The same is true of the relationship between the training set and the validation set: the function of the validation set is to help select the configurations that will make the learning as successful as possible. If these sets overlap, we will not be able to objectively and impartially evaluate the behavior of the model and select suitable configurations.

The requirement that training, validation and testing sets should be disjoint also means that information from one of the sets must not spill over into the other sets. This is especially important when applying preprocessing and set preparation techniques. Let's consider the following example. We mentioned that due to the sensitivity of the machine learning model to the value of attributes, standardization of numerical attributes is often performed. One can approach this transformation by calculating the mean and standard deviation based on the entire set, and then using it, in turn, to standardize the training and testing sets. In order not to overflow information, the correct sequence of these steps is actually as follows:

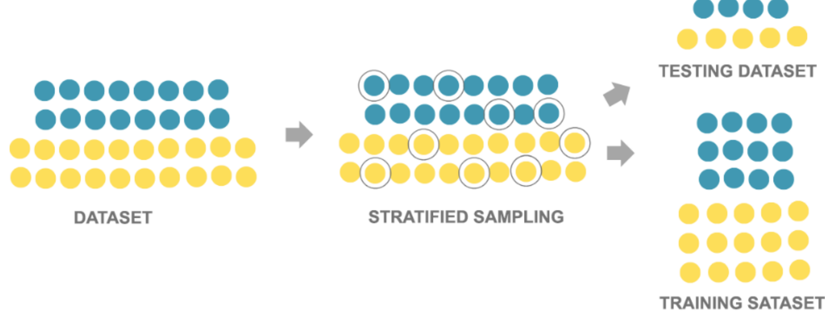
- dividing the dataset into a training and testing suite,
- calculation of the mean and standard deviation only on the training set,
- transformation of the training set using calculated values,
- Transform the test set using the values computed over the training set.

Because of the desire not to make mistakes, one may also think as follows: after dividing the initial dataset into a training set and a test set, I will perform a separate standardization of the training set and the test set. And this approach, although more cautious, is not correct because it leads to the modification of the test set. In the lower left image, the yellow triangles represent the instances of the training set, and the blue circles represent the instances of the test set. The image in the middle represents these instances after correct standardization (you can carefully compare the images and the arrangement of the points - the scale along the x-axis has changed due to standardization, everything else has remained the same). In the image on the right you can see the instances after the standardization has been done separately over the training set and the test set - the spatial arrangement has changed quite a bit now.



Examples of Right and Wrong Standardization

When dividing the baseline dataset, it would be ideal to preserve the proportions relative to the attribute values and the value of the target variable. For example, if the ratio of male to female patients in the medical dataset is 4:5, it would be ideal that, after the division, the ratio of patients in the training set and in the test set should be approximately 4:5. Techniques that allow this type of division are called **stratification** techniques. However, due to the number of attributes and their combinations, in practice this is often not a realistic requirement, so most often it is insisted on proportionality in relation to the values of the target variable. We will discuss this topic separately in the context of the task of classification.



Stratified Training and Testing Assemblies

3. TRAINING MODELS

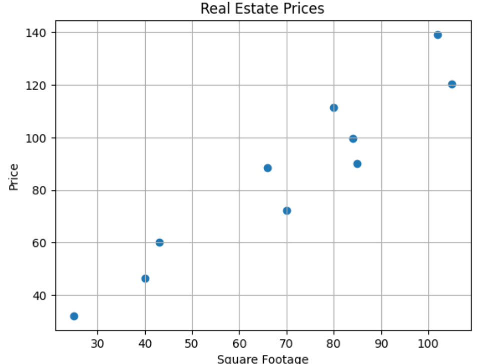
Welcome to the topic of **Training Models**! This section delves into the various models and techniques used in machine learning to solve different types of problems. You will learn about linear regression, classification, decision trees, and the k-nearest neighbors algorithm, among others. We will discuss the characteristics, advantages, and disadvantages of each model, and the importance of model validation to ensure accuracy and reliability. Through practical exercises, you will apply these models to real datasets and interpret the results.

3.1 Linear Regression

Let's go back to the example we used to introduce the data-driven programming paradigm. In the dataset, we had information about the square footage of the properties and their prices, and our task was to learn the connection between these values so that we could estimate prices for new properties.
For the sake of simplicity, let's have a training set that contains 10 instances, which are listed in the table below:

Square Footage (m^2)	Real estate Prices (1000€)
43	60
25	32.1
66	88.4
80	111.4
105	120.32
70	72.1
40	46.3
85	90.1
84	99.6
102	139.2

We can also display these images graphically. Along the x-axis, we will set the values of square footage, along the y-axis the values of real estate prices and mark the pairs of values with blue circles.



Let us choose for the model a function that relates the real estate square footage x and the real estate prices y with the equation $y = \beta_0 + \beta_1 x$, where β_0 i β_1 represent unknown parameters. This is the so-called **linear model,** and since we use it to solve the regression problem, we also call it **a linear regression model.** Note that this is actually a line equation $y = kx + n$, where the coefficient of the line line is denoted by β_1 and the free term is denoted by β_0. The motivation for introducing this model lies in the fact that the dots follow the imaginary diagonal of the quadrant, perhaps slightly lowered.

This section is paired with Jupyter Volume 05-1-linear_regression.ipynb. To follow the content further, click on the link and then on the button Open in Colab to open the content in *Google Colab.* If you are viewing the notebooks on your local machine, find the notebook with the same name among the contents and run it. For more detailed instructions, see the *Hands-on Zone* section and the *Jupyter Exercise Notebook* lesson.

Your task is to load the real estate data set in the companion notebook and select the values of the parameters β_0 and β_1 that you think best match this data. You can adjust them by moving the sliders left and right. Remember the values you have chosen and what ideas guided you when determining the parameters.

You've probably tried to get the right one that comes as close as possible to the given points and makes as few deviations as possible. You were equally satisfied with some parameter choices, while some were really bad. And from the point of view of machine learning, we try to find the values of the parameters β_0 and β_1 for which we make the smallest error, but we have to define exactly what the error actually is. Here's how we're going to do it.

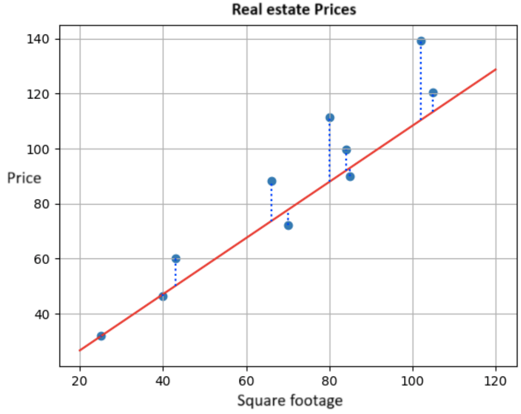
Suppose the selected values are $\beta_0 = 6.3$ i $\beta_1 = 1.02$. Now let's expand the data table with a column with the values calculated by this linear regression model for the quadrature values we have. From the angle of the model, we represent them with the size x.

Square Footage (m^2)	Real estate Prices (1000€)	Model price y=6.3+1.02x (1000€)
43	60	50.16
25	32.1	31.8
66	88.4	73.62
80	111.4	87.9
105	120.32	113.4
70	72.1	77.7
40	46.3	47.1
85	90.1	93
84	99.6	91.98
102	139.2	110.34

The difference between the values that are expected (known in the data set) and the values that we have calculated (remember to call them predictions) is an error. Now let's calculate all the errors and record them in the table.

Square Footage (m^2)	Real estate Prices (1000€)	Model price y=6.3+1.02x (1000€)	Model error
43	60	50.16	9.84
25	32.1	31.8	0.3
66	88.4	73.62	14.78
80	111.4	87.9	2.53
105	120.32	113.4	6.92
70	72.1	77.7	-5.6
40	46.3	47.1	-0.8
85	90.1	93	-2.9
84	99.6	91.98	7.62
102	139.2	110.34	28.86

To make it easier to track the behavior of the errors, in the image below, their values are shown by blue dotted lines.



In order to get an idea of the total error of the model, it is unwise to add the individual errors, since some error values are positive and some values are negative. Therefore, we can square them and add them together - this will give us stronger information about the size of the error, regardless of whether it is positive or negative. If we divide the resulting sum by the number of instances in the set, Let's get an idea of the average error of the model. In our case, it is:

$$(9.84^2 + 0.32^2 + 14.782^2 + 23.52^2 + 6.92^2 + (-5.6)^2 + (-0.8)^2 + (-2.9)^2 + 7.62^2 + 28.86^2)/10 = 184.687$$

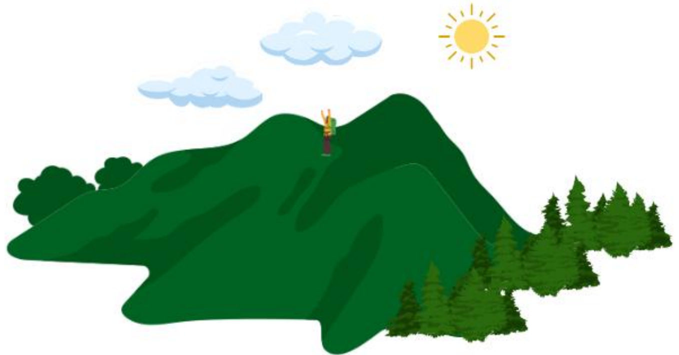
The error of the linear regression model calculated in this way is called the **mean square error** (*MSE*). For fixed values of the parameters β_0 and β_1, the calculation procedure we have described can be abbreviated by the formula $\frac{1}{N}\sum_{i=1}^{N}\left(y_i - (\beta_0 + \beta_1 x_i)\right)^2$. In it, the pairs x_i, y_i correspond to the individual instances, the square footage of the properties x_i and their prices y_i, and the number n indicates the total number of instances. That's 10 in our case. The expression figured in the sum represents the difference between the expected yi and the calculated $\beta_0 + \beta_1 x_i$ values.

The squared error is the error that we always pair with the linear regression model, and that we want to minimize as much as possible by choosing the right parameters β_0 and β_1. From the experience of setting the parameters, you have seen that this is not a very easy task. Fortunately, there are mathematical techniques that can help us with this. To figure out how to do this, let's move on to the next lesson on gradient descent.

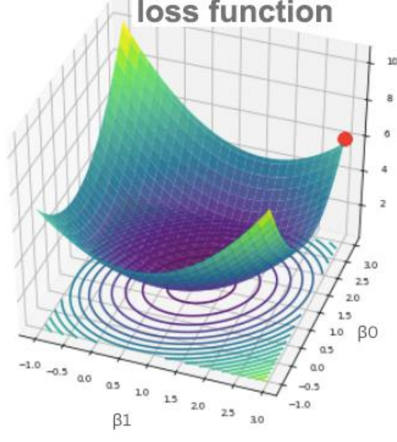
3.2 Gradient Descent

In this lesson, we will learn about gradient descent, a technique that helps us find the parameters for which the mean square error function has the smallest value. This technique can also be applied to other functions under certain conditions.

You'll have to imagine something again - this time you're on top of a beautiful mountain. That is not so difficult! The trouble is that now follows the task: to get down to the foot quickly! One way to do this is to first look around and see which direction in your area the mountain is steepest - keep in mind that you need to go down very quickly! Then you can take a careful step in that direction and then stop and look around again. Again, you can see which direction the mountain is the steepest in your area, take a step in that direction and stop. It is clear to you that you can continue to repeat this order of observation, choice of direction and lunge until you reach the foot. There is a refreshment waiting for you for a successfully completed task!



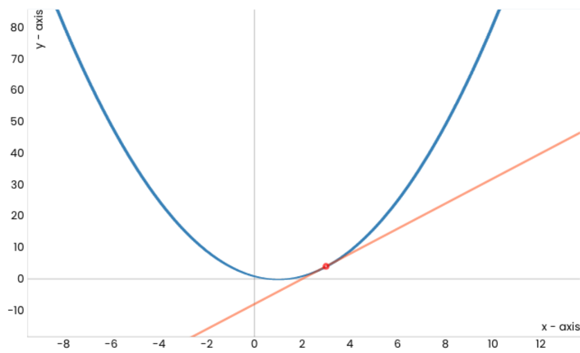
A little physical activity in the middle of a linear regression story doesn't hurt, but you sense that there's something else. The mean square error function depends on the choice of the parameters of β_0 and β_1- for different combinations of β_0 and β_1 values, we get different error values. If we plot a graph of this function, for example, along the x-axis we record the values of β_0, along the y-axis we record the values of β_1, and along the z-axis we record the error values, we get a graph that looks like the one in the figure below. If we mark a random selection of the parameters β_0 and $\beta_1$with a red dot, in order to get to the point for which the error value is smallest, we really have to go down to the foot of this surface. That is why the "technique" that we developed in the previous example is very relevant. We just need to figure out how to find the steepest directions of descent. Functions will help us with this



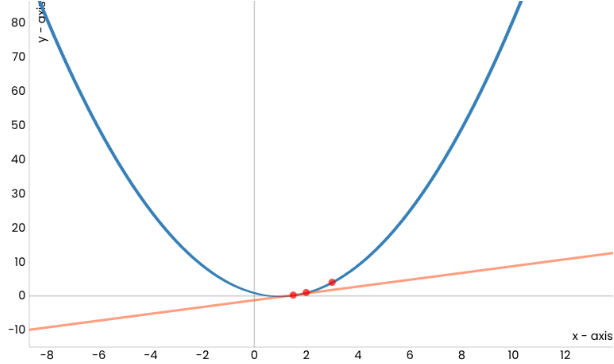
Graph of the Mean Square Error Function

The smallest value of a function is called the minimum.

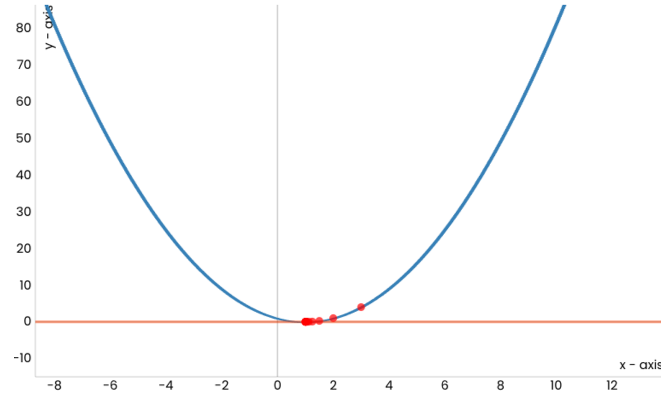
Now let's look at the quadratic function $f(x) = (x-1)^2$ whose graph is shown in the figure below, and try to reach its minimum with the descent technique - it is at the point x=1 and is 0.



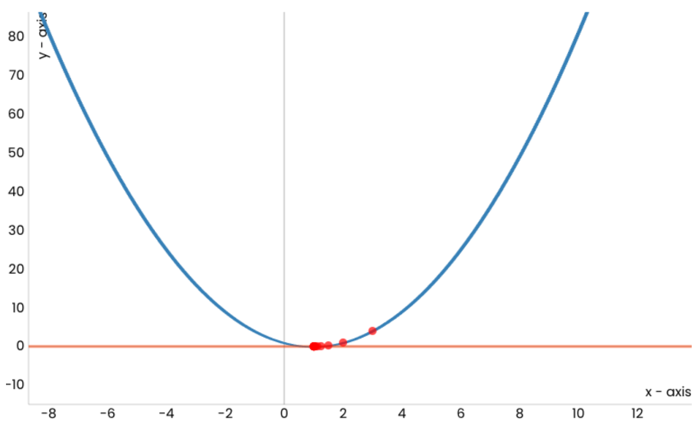
Notice the red dot corresponding to x=3 (randomly chosen) that marks the starting position of the movement towards the minimum of this function. It seems that the orange line marks the steepest direction along which we can start the descent. Interestingly, this line actually represents the tangent of our function at the point x=3. If we take a step along this line, we will find ourselves at a new point. Let's mark its value in red and display it on the graph. It's a little closer to the expected minimum.



Now we can repeat the process: let's draw a tangent at a new point, and then take a step along that line.



After a certain number of steps, this procedure will bring us to the minimum function, i.e. to the point x = 1.



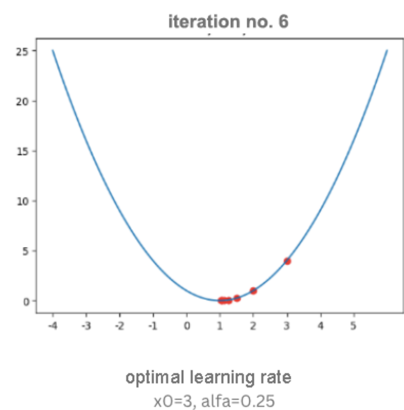
This section is paired with Jupyter Notebook 05-2-gradient_descent.ipynb. To follow the content further, click on the button Open in Colab d to open the content in *Google Colab.* If you are viewing the notebooks on your local machine, find the notebook with the same name among the contents and run it. For more detailed instructions, see the *Hands-on Zone* section and the *Jupyter Exercise Notebook* lesson.

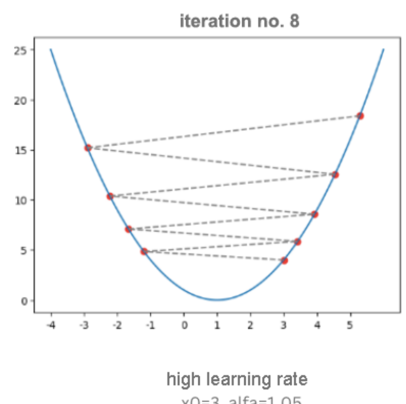
In the notebook that accompanies this material, you can start the animation yourself and make sure that it is so.

Before we go into more detail about the procedure we have described, let's recall what these real tangents are. For a fixed point x, the coefficient of the direction of the tangent at x is equal to the value of the first derivative of the function at x. The first derivative of our function is the function $f'(x) = 2x - 2$ and at the initial point x = 3 the value of the derivative is $f'(x) = 4$. This means that the tangent has the equation $y = 4x - 8$ (the number -8 is obtained from the condition that this line must contain a point (3, 4)). That is why we can also say that the tangent has a direction corresponding to the derivative of a function at a certain point, and for the movement itself in that direction that the movement along the direction of the derivative is at that point. Now the dilemma is whether we move along or down, i.e. Are we going in the opposite direction, or are we going in the opposite direction? Well, since we want to go down to the minimum, we need to follow the direction opposite to the direction of the derivative of the function.

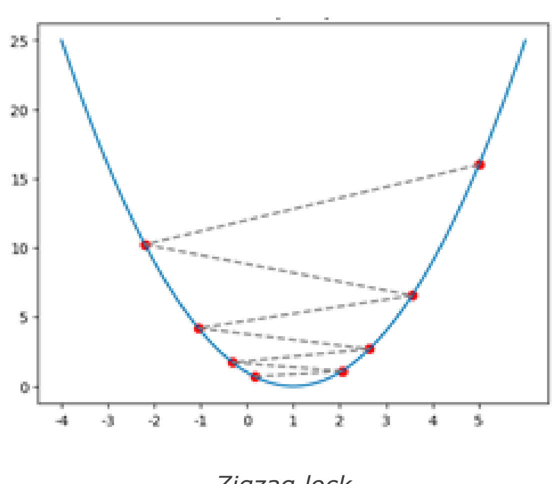
If we now denote the starting point with x_0, we get a new point x_1 by taking a step along the direction of the derivative of the function at the point x_0. by taking a step along the direction of the derivative of the function at the point x_1 we calculate the value of the new point $x_1 = x_0 - \alpha f'(x_0)$. Since we repeat the procedure, we calculate the value of the point x_2 as $x_2 = x_1 - \alpha f'(x_1)$ and continue with the calculations $x_3 = x_2 - \alpha f'(x_2), x4 = x_3 - \alpha f'(x_3)$, … Repeat the procedure until for two consecutive values, say x_{34} and x_{35}, values of the function are close enough, i.e. while the absolute value of the difference $f(x_{35}) - f(x_{34})$ not less than some predetermined precision, say 0.001. Thus, computationally, we can approach the concept of convergence in mathematics.

The value ɑ we introduced is called the **learning rate** and represents a very important parameter of the algorithm that we have described. If the values for ɑ are very small, it will take us a long time to reach the minimum. On the other hand, if the values for ɑ are very high, it can happen that we skip the minimum or fall into a zigzag trap by constantly jumping around it! Look at the image below!





The Impact of Learning Step Choices



Zigzag lock

Be sure to check both of these behaviours yourself in the companion notebook using the different settings for the learning step in the animation.

The algorithm we have described is called **Gradient Descent** and despite its simplicity, it is one of the most important algorithms in machine learning because it makes it possible to find the smallest value of an error function. There are many details about this algorithm that we won't go into about the properties of the functions to which this algorithm can be successfully applied, the numerical calculation of the derivative, and the selection of learning steps. All of them must be considered during the practical application of the algorithm.

The algorithm itself is not unpleasant to program, so we will embark on an adventure. We need a function f, which will calculate the value of a given function, and a function f which will calculate the value of the derivative of a given function. We need to define both the alpha learning step and the stop criteria: we will suspend the procedure when the values of the function in two successive iterations are close enough (the difference between their values is less than some predetermined epsilon accuracy) or when we reach a finite number of iterations (we also need to make sure in cases of inappropriate learning step choices).

Follow the code block. The algorithm started by setting a starting point. Since the point at which we move by the gradient descent algorithm is the starting point of the next step, we use the x_old and x_new markers to mark them in successive steps. The report that we create at the end of the function contains information about whether the algorithm has stopped, how many steps it takes, i.e. iteration needed and what value it found.

```
def gradient_descent(f, f_derivative, x, alpha, epsilon, max_iterations):
    # set the initial value for x
    x_old = x
    # in each iteration...
    for i in range(0, max_iterations):
        # calculate the current value for x
        x_new = x_old - alpha * f_derivative(x_old)
        # and then check if the stopping criterion is met
        if np.abs(f(x_new) - f(x_old)) < epsilon:
            break
        # if the criterion is not met, prepare x for the next iteration
        x_old = x_new
    # at the end of the whole process, prepare a report with information:
    # whether the algorithm stops,
    # how many iterations it lasted,
    # and what value of x was found
    report = {}
    report['stops'] = i != max_iterations
    report['number_of_iterations'] = i
    report['x_min'] = x_old

    return report
```

The function and its derivation can be defined by the following Python blocks:

```
def f(x):
   return (x-1)**2
def f_izvod(x):
   return 2*x-2
```

After running the gradient_descent function for the values of the arguments x0 = 3, alpha = 0.1, epsilon = 0.001, and max_number_of_iterations = 100, we get that the minimum of the function is 1.0048, which we can confirm. You can also run the code yourself and make sure that you get the result. Don't forget to examine how the results change if other argument values are selected.

Now we can return to the problem of finding the parameter β_0 i β_1 linear regression for which the value of the mean square error should have the smallest value. The mean square error function is a function of two variables - it depends on both the value of the parameter β_0 and the value of the parameter β_1. When working with functions of multiple variables, in general with n variables $x_1, x_2, x_3, \ldots, x_n$, the derivative that we used in the gradient descent algorithm is generalized by a vector of partial derivatives - for each of the variables we calculate the derivatives individually. For example, for a function $\frac{1}{2}(x_1^2 + 10x_2^2)$, the derivative of the variable x_1 is obtained by declaring the variable x_2 as a constant and then applying the standard rules for calculating the derivative that bring us to $\frac{1}{2} \cdot 2 \cdot x_1 = x_1$. On the other hand, the derivative of the variable x_2 is calculated by declaring the variable x_1 as a constant and applying the standard rules for calculating the derivative. Now we get $\frac{1}{2} \cdot 10 \cdot 2 \cdot x_2 = 10 \cdot x_2$. Now we get that the vector of the derivative by individual variables (such derivatives are called partial) is the vector $[x_{1,} 10 \cdot x_2]$. In mathematics, and even in machine learning, these vectors are called **gradients**, hence the name of the algorithm itself. To denote gradients, we use the symbol

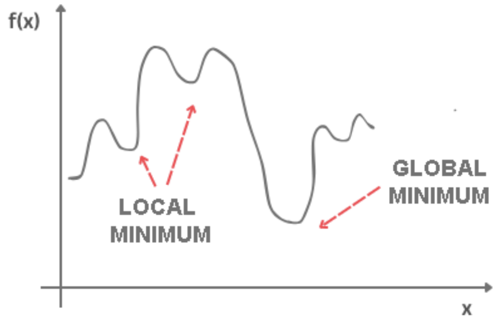
triangle down, ∇ called nabla. Thus, the precise notation of the gradient of the starting function f' would be $\nabla f(x_1, x_2) = [x_1, 10 \cdot x_2]$ and would allow us to keep track of which directions of the lead we should move individually during the descent.

The other steps of the gradient descent algorithm do not differ much in the case of multivariate functions: we expect the algorithm to stop after the desired accuracy has been achieved, or after a certain number of iterations have been performed.

Now that we understand how gradient descent works for functions of multiple variables, let's go back to calculating the parameters β_0 i β_1. We said that the equation of the mean square error is $\frac{1}{N}\sum_{i=1}^{N}\left(yi - (\beta_0 + \beta 1xi)\right)^2$. Since this is a function for which we need to find the minimum, if we roll up our sleeves and check, we get that the derivative of the mean square function by β_0 is exactly $\frac{2}{N}\sum_{i=1}^{N}(\beta_0 + \beta_1 x_i - y_i)$ and the derivative by $\frac{1}{N}\sum_{i=1}^{N}(\beta_0 + \beta_1 x_i - y_i) \cdot x_i$. These derivatives indicate which directions we should move along and how much we should correct the values for β_0 i $\beta_1$in each step of the gradient descent iteration.

In the notebook, you can also see how these values are calculated through code, and then go through the entire process of custom gradient descent. For the real estate set we have introduced, we will arrive at the values of $\beta_0 = 2.056$ and $\beta_1 = 1.198$.

We have said that there are certain prerequisites that a function needs to satisfy in order for its minimum to be found by the gradient descent technique (it is necessary for the function to be differentiable). It is also important to know that in general a *local minimum is reached in this way*. For example, the function in the figure below has several local minimums and only one *global minimum*. In some cases, for example, when a function is convex, the local and global minimums coincide, so we always arrive at the desired solution, the global minimum. The squared error function is convex by the parameters β_0 i β_1.



Local and global minimums.

The field of mathematics that deals with finding the maximum and minimum values of functions (we call them optimums by one name) is called **mathematical optimization.** Gradient descent is only one algorithm from the palette of this field.

3.3 Polynomial regression

What if your data is actually more complex than a simple straight line? Surprisingly, you can actually use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called Polynomial Regression.

Polynomial Regression is a regression algorithm that models the relationship between a dependent(y) and independent variable(x) as nth degree polynomial. The Polynomial Regression equation is given below:

$$y = \beta_0 + \beta_1 x_1^1 + \beta_2 x_1^2 + \beta_3 x_1^3 + \cdots \ldots \beta_n x_1^n$$

It is also called the special case of Multiple Linear Regression in ML. Because we add some polynomial terms to the Multiple Linear regression equation to convert it into Polynomial Regression.

It is a linear model with some modification in order to increase the accuracy.

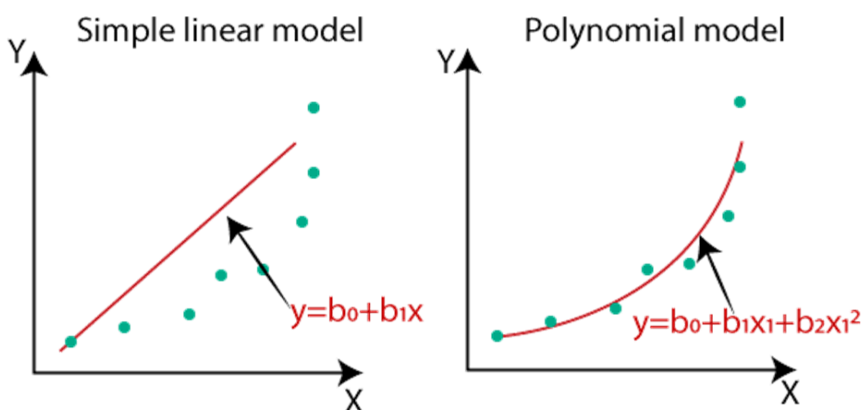
The dataset used in Polynomial regression for training is of non-linear nature.

It makes use of a linear regression model to fit the complicated and non-linear functions and datasets.

Hence, "In Polynomial regression, the original features are converted into Polynomial features of required degree (2,3,..,n) and then modelled using a linear model."

The need of Polynomial Regression in ML can be understood in the below points:

- If we apply a linear model on a **linear dataset**, then it provides us a good result as we have seen in Simple Linear Regression, but if we apply the same model without any modification on a **non-linear dataset**, then it will produce a drastic output. Due to which loss function will increase, the error rate will be high, and accuracy will be decreased.
- So for such cases, **where data points are arranged in a non-linear fashion, we need the Polynomial Regression model**. We can understand it in a better way using the below comparison diagram of the linear dataset and non-linear dataset.



- In the above image, we have taken a dataset which is arranged non-linearly. So if we try to cover it with a linear model, then we can clearly see that it hardly covers any data point. On the other hand, a curve is suitable to cover most of the data points, which is of the Polynomial model.

- Hence, *if the datasets are arranged in a non-linear fashion, then we should use the Polynomial Regression model instead of Simple Linear Regression.*

Note: A Polynomial Regression algorithm is also called Polynomial Linear Regression because it does not depend on the variables, instead, it depends on the coefficients, which are arranged in a linear fashion.

Simple Linear Regression equation	$y = \beta_0 + \beta_1 x_1$
Multiple Linear Regression equation	$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + + \cdots \ldots \beta_n x_n$
Polynomial Regression equation	$y = \beta_0 + \beta_1 x_1^1 + \beta_2 x_1^2 + \cdots \ldots \beta_n x_1^n$

Implementation of Polynomial Regression using Python:

Here we will implement the Polynomial Regression using Python. We will understand it by comparing Polynomial Regression model with the Simple Linear Regression model. So first, let's understand the problem for which we are going to build the model.

Problem Description: There is a Human Resource company, which is going to hire a new candidate. The candidate has told their previous salary 160K per annum, and the HR have to check whether they are telling the truth or bluff. So to identify this, they only have a dataset of his previous company in which the salaries of the top 10 positions are mentioned with their levels. By checking the dataset available, we have found that there is a **non-linear relationship between the Position levels and the salaries**. Our goal is to build a **Bluffing detector regression** model, so HR can hire an honest candidate. Below are the steps to build such a model.

Position	Level(X-variable)	Salary(Y-Variable)
Business Analyst	1	45000
Junior Consultant	2	50000
Senior Consultant	3	60000
Manager	4	80000
Country Manager	5	110000
Region Manager	6	150000
Partner	7	200000
Senior Partner	8	300000
C-level	9	500000
CEO	10	1000000

Steps for Polynomial Regression:

The main steps involved in Polynomial Regression are given below:

- Data Pre-processing
- Build a Linear Regression model and fit it to the dataset
- Build a Polynomial Regression model and fit it to the dataset
- Visualize the result for Linear Regression and Polynomial Regression model.
- Predicting the output.

Data Pre-processing Step:
The data pre-processing step will remain the same as in previous regression models, except for some changes. In the Polynomial Regression model, we will not use feature scaling, and also we will not split our dataset into training and test set. It has two reasons:

- The dataset contains very less information which is not suitable to divide it into a test and training set, else our model will not be able to find the correlations between the salaries and levels.
- In this model, we want very accurate predictions for salary, so the model should have enough information.

The code for pre-processing step is given below:

```
# importing libraries
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd
#importing datasets
data_set= pd.read_csv('Position_Salaries.csv')
#Extracting Independent and dependent Variable
x= data_set.iloc[:, 1:2].values
y= data_set.iloc[:, 2].values
```

Building the Linear regression model:

Now, we will build and fit the Linear regression model to the dataset. In building polynomial regression, we will take the Linear regression model as reference and compare both the results. The code is given below:

```
#Fitting the Linear Regression to the dataset
from sklearn.linear_model import LinearRegression
lin_regs= LinearRegression()
lin_regs.fit(x,y)
```

In the above code, we have created the Simple Linear model using **lin_regs** object of **LinearRegression** class and fitted it to the dataset variables (x and y).

Output:

Out[5]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

Building the Polynomial regression model:

Now we will build the Polynomial Regression model, but it will be a little different from the Simple Linear model. Because here we will use **PolynomialFeatures** class of **preprocessing** library. We are using this class to add some extra features to our dataset.

```
#Fitting the Polynomial regression to the dataset
from sklearn.preprocessing import PolynomialFeatures
poly_regs= PolynomialFeatures(degree= 2)
x_poly= poly_regs.fit_transform(x)
lin_reg_2 =LinearRegression()
lin_reg_2.fit(x_poly, y)
```

In the above lines of code, we have used **poly_regs.fit_transform(x)**, because first we are converting our feature matrix into polynomial feature matrix, and then fitting it to the Polynomial regression model. The parameter value (degree= 2) depends on our choice. We can choose it according to our Polynomial features.

After executing the code, we will get another matrix **x_poly**, which can be seen under the variable explorer option:

	0	1	2
0	1	1	1
1	1	2	4
2	1	3	9
3	1	4	16
4	1	5	25
5	1	6	36
6	1	7	49
7	1	8	64
8	1	9	81
9	1	10	100

Next, we have used another LinearRegression object, namely **lin_reg_2**, to fit our **x_poly** vector to the linear model.

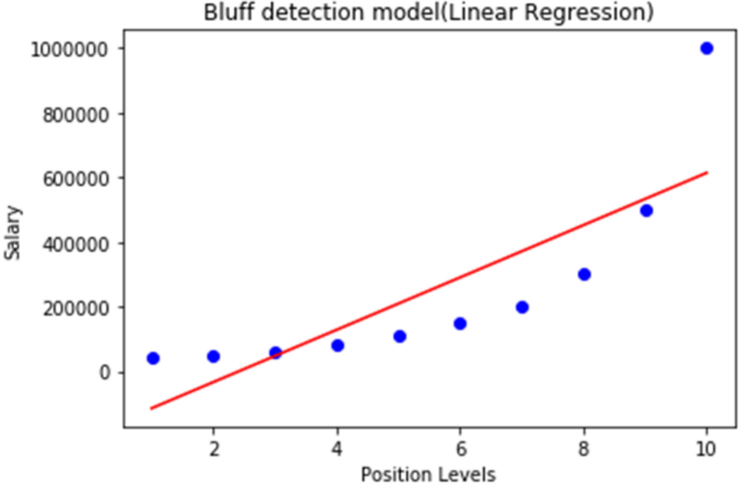
Output:

Out[11]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

Visualizing the result for Linear regression:

Now we will visualize the result for Linear regression model as we did in Simple Linear Regression. Below is the code for it:

```
#Visulaizing the result for Linear Regression model
mtp.scatter(x,y,color="blue")
mtp.plot(x,lin_regs.predict(x), color="red")
mtp.title("Bluff detection model(Linear Regression)")
mtp.xlabel("Position Levels")
mtp.ylabel("Salary")
mtp.show()
```



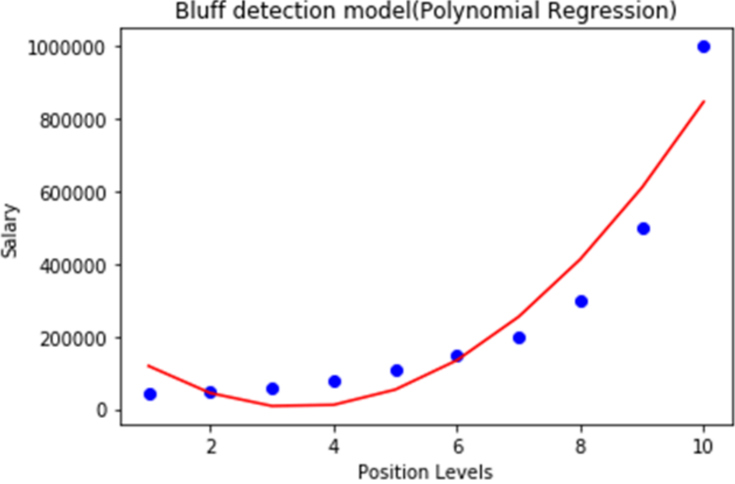
In the above output image, we can clearly see that the regression line is so far from the datasets. Predictions are in a red straight line, and blue points are actual values. If we consider this output to predict the value of CEO, it will give a salary of approx. 600000$, which is far away from the real value.

So we need a curved model to fit the dataset other than a straight line.

Visualizing the result for Polynomial Regression

Here we will visualize the result of Polynomial regression model, code for which is little different from the above model.

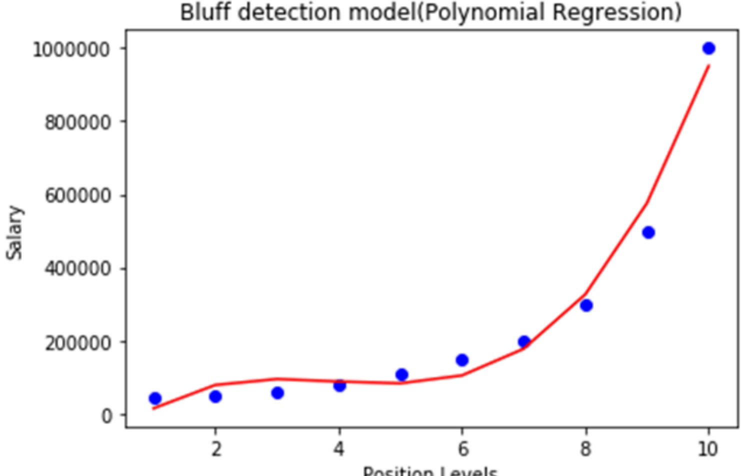
In the above code, we have taken lin_reg_2.predict(poly_regs.fit_transform(x), instead of x_poly, because we want a Linear regressor object to predict the polynomial features matrix.



As we can see in the above output image, the predictions are close to the real values. The above plot will vary as we will change the degree.

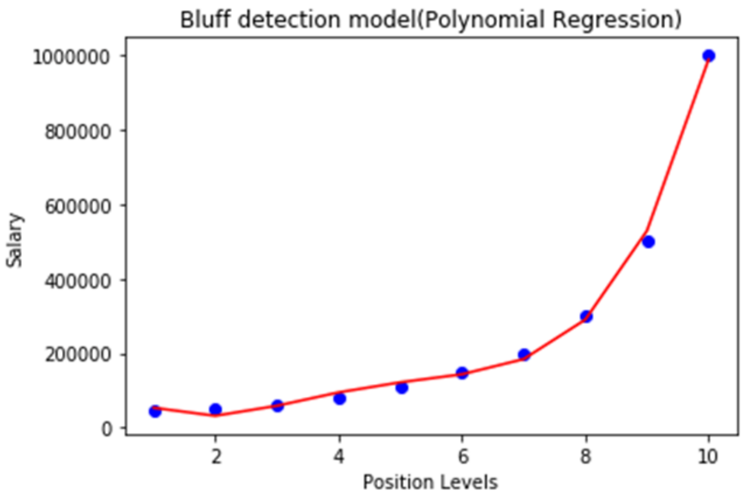
For degree= 3:

If we change the degree=3, then we will give a more accurate plot, as shown in the below image.



So as we can see here in the above output image, the predicted salary for level 6.5 is near to 170K$-190k$, which seems that future employee is saying the truth about his salary.

Degree= 4: Let's again change the degree to 4, and now will get the most accurate plot. Hence we can get more accurate results by increasing the degree of Polynomial.



Predicting the final result with the Linear Regression model:

Now, we will predict the final output using the Linear regression model to see whether an employee is saying truth or bluff. So, for this, we will use the **predict()** method and will pass the value 6.5. Below is the code for it:

```
lin_pred = lin_regs.predict([[6.5]])
print(lin_pred)
```

Output:

[330378.78787879]

Predicting the final result with the Polynomial Regression model:

Now, we will predict the final output using the Polynomial Regression model to compare with Linear model. Below is the code for it:

```
poly_pred = lin_reg_2.predict(poly_regs.fit_transform([[6.5]]))
print(poly_pred)
```

Output:

[158862.45265153]

As we can see, the predicted output for the Polynomial Regression is [158862.45265153], which is much closer to real value hence, we can say that future employee is saying true.

3.4 Multiple linear regression

In the introductory story about datasets, we saw that a larger number of attributes are used. However, in the story of linear regression, we used only one attribute (the square footage of the property). You're probably wondering what we do when we have multiple attributes and whether we can then apply a linear regression model.

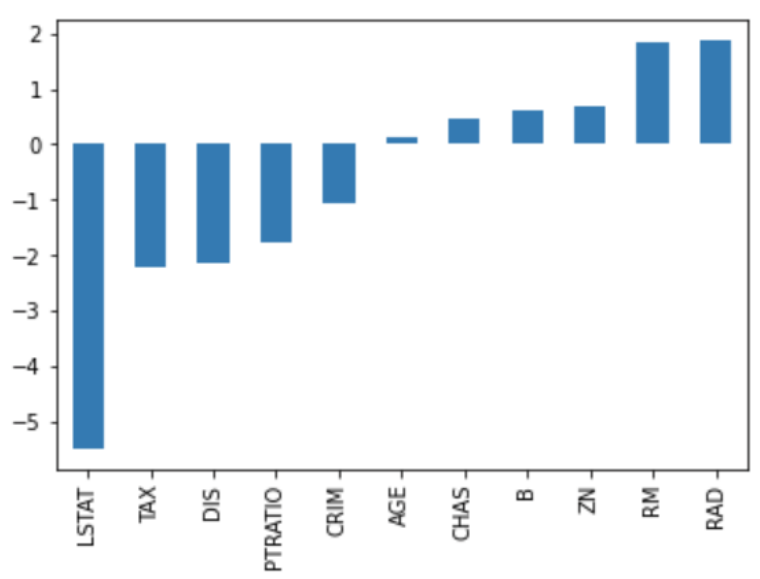
The linear regression model that is adapted to this scenario is called **multiple** linear regression and has the form $y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \cdots + \beta_n X_n$. Don't be confused by this long expression - now the values $X_1, X_2, X_3, \ldots, X_n$ represent the individual attributes and the parameters $\beta_0, \beta_1, \beta_2, \beta_3, \ldots, \beta_n$ ..., are the parameters of the model. Behind this generalization is again the idea of a linear relationship between the individual attributes and the target variable.

The goal of learning is to determine the values of the parameters $\beta_0, \beta_1, \beta_2, \beta_3, \ldots, \beta_n$ and thus get an idea of the dependencies. We arrive at them in the same way as with the linear regression that we have come to know (we also say that it is simple): by minimizing the mean square error on the training dataset. The gradient descent technique can be generalized to suit this task setup and can help us find the set of values $\beta_0, \beta_1, \beta_2, \beta_3, \ldots, \beta_n$ for which the mean square error is the smallest.

In the case of a single-attribute linear regression model, we could also imagine the meaning of the parameters β_0 and β_1: they determined the displacement and slope of the line passing through the data set. Thus, they showed us the strength of the linear dependence between the input and output variables, i.e. how much the value of the output variable y changes when we change the attribute x by 1. Now that we have more parameters, it is natural to wonder what meaning we can give them. They have the same kind of dependency. If we imagine that only β_0 and β_2: are non-zero parameters, then the relationship between the target variable y and the attribute X2 is represented by the equation $y = \beta_0 + \beta_2 X_2$, tj. Linear and the same tells us how much the value for the target variable y will change and in which direction when we change the value for X_2 by 1.

Given that the parameters summarize the knowledge from the data set for us, in the case of multiple linear regression, larger parameter values indicate a greater significance of an attribute on the value of the target variable. In order to be able to track this property, we usually plot the values of the calculated parameters with a column graph. The figure below shows the parameter values of a model that uses a real dataset to

predict real estate prices (the popular Boston real estate). Without going into much detail about this set, we can immediately notice that the LSTAT attribute has the most, and negatively, on the value of the target variable, while the RM and RAD attributes have a positive effect, almost equally. Graphics of this type, which can give us some idea of the influence of attributes, are called **feature importance graph**.



Graph of the importance of multiple regression attributes

Another detail that should be emphasized, so as not to surprise you later, concerns linearity. The linear regression model is **linear in terms of parameters.** This means that a model whose form $y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3$ in which the degrees of attribute values are figured, would be run as a linear model. It is similar for the model $y = \beta_0 + \beta_1 log(X)$, in which the logarithm of the attribute value is figured. You can think of these perhaps unexpected attribute roles as transformations that fix the linear relationship between the attribute and the target variable.

3.5 Classification, types of classification, and matrix of confusion

Believe it or not, you've been asked this many times before. When you're tidying up a room and separating pieces of paper to keep or throw away, or when you're sorting your photos into those from an excursion, aunt's birthday, or a trip with friends, you're actually doing a classification task: you have groups in mind, and when you look at a piece of paper or a photo, you're deciding which group you belong to. And many of the programs you encounter do the task of classification. For example, your email client distinguishes between desirable and unsolicited mail, and thanks to this feature, you manage to avoid many traps and scams on the Internet. Also, on social media, you often get recommendations for connecting with new people - a program that is behind the social network actively evaluates whether a person is a potential friend of yours or not (it usually follows your friends' friends and gets ideas). Since we have no doubt that you are an expert in organizing the room and files on your computer, let's learn how these skills are mastered by programs!

Types of classification

At the very beginning, it is important to emphasize that not all classifications are the same. Therefore, we will first find out what classifications exist and what characterizes them. Examples of sorting pieces of paper or sorting mail are examples **of binary classification** because we have only two groups: pieces of paper to be thrown away and pieces of paper to be stored, i.e. desirable and undesirable mail. Groups in the world of machine learning are called **classes** so we will continue to stick to that date. In order to be able to distinguish between classes, we associate them with names that approximate what they actually contain. For example, "slips of paper" and "junk mail" are clear enough names. Names are often specified by labels, which appear in the dataset to which the classification task is applied.

If we have more than two classes, we are talking about **a multiclass classification** task. For example, such is the task of sorting photos by events where each event can represent one class. We can create three directories, i.e. three classes, give them the names "excursion", "aunt's birthday" and "trip", and then assign each of the photos to one of these classes by putting it in the appropriate directory.

We can think about different types of classification based on the criteria of affiliation. For example, one e-mail can be either desirable or undesirable, it cannot belong to both the class of desirable and undesirable e-mails at the same time. It is similar with the photographs and classes that we have introduced. On the other hand, one newspaper article can be simultaneously on the topic of culture, travel and food, so we can associate it with a greater number of classes - the one that represents culture, the one that represents travel and the one that represents food. Since in this case the instances have more features, i.e. labels, we call this type **of classification multilabelary classification.** Although it is very interesting and useful, we will not cover the multilabelar classification with further contents, but we will focus on binary and multiclass classification.

What kind of classification do you think the following tasks belong to:

- sorting garbage for recycling,
- determining the correctness of the program,
- determining the language of the document,
- checking the validity of a bank transaction.
- next word suggestion when typing an SMS message

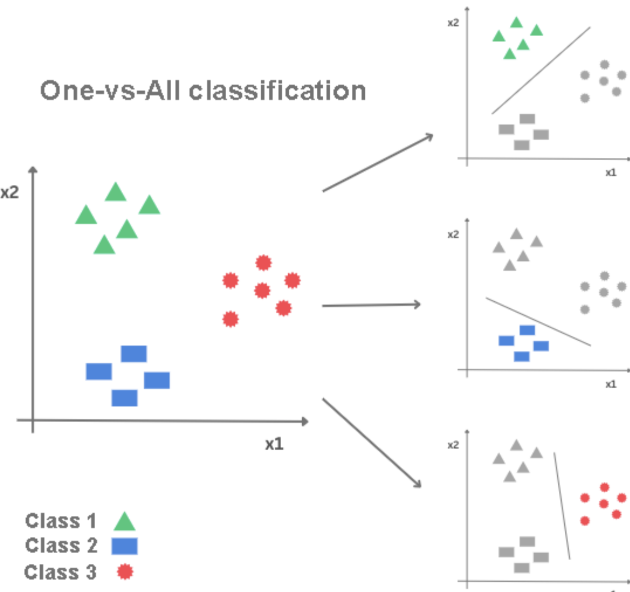
Classification from the point of view of machine learning

When we think about the classification task from the point of view of machine learning, we are interested in discrete mappings, i.e. mappings that can assign one of finite values to input variables. Most often, the number of classes is smaller, expressed in a single digit number, but you can also recall the *ImageNet* set and the image classification competition in which 1000 classes are used. Variables that can take a finite number of values are called categorical, so we can talk about classification. It's like a mapping that is characterized by a categorical target variable.

$$F(x) = f(x) = \begin{cases} 0, & x < 0 \\ \frac{1}{2}, & 0 \leq x < 1 \\ 1, & x > 1 \end{cases}$$

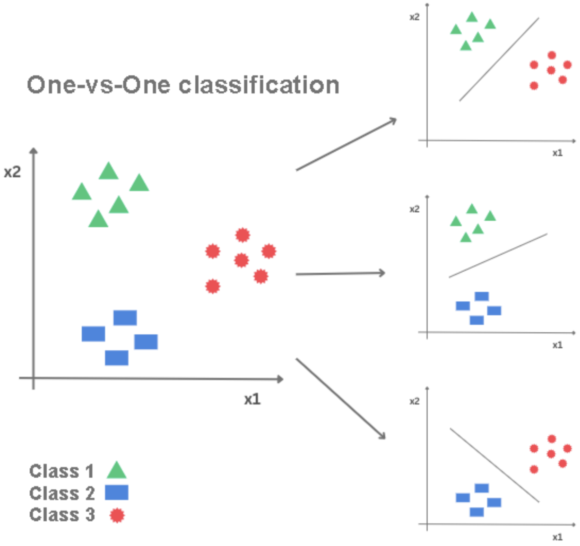
An example of a discrete function.

Since classification is a supervised machine learning task, the dataset used to train the model contains pairs of inputs and expected outputs. The inputs can be images, text messages, or tabular data, and all the guidelines discussed in the lesson on preparing a dataset apply to their preparation. The output is always the name of the class. Although we introduced class names in order to make it easier to follow the classification task, when we get to the part of preparing the data, we need to transform it into numerical values as well. Here we can be guided by the preparations that we have discussed for working with categorical attributes: mapping a set of values or *one-hot* coding.
In the case of binary classification, we usually map the class names to the values 0 and 1. For example, the occurrence of the class name "junk mail" is replaced by the value 0, and the occurrence of the name "junk mail" by the value 1. Often, instances that have a label of 0 are said to belong to a **negative class**, and instances that have a label of 1 are said to belong **to a positive class.**
When it comes to multiclass classification, we use *one-hot coding to prepare the target variable.* For example, for the task of sorting photos by events, we will transform the outputs into vectors of length three because we have exactly three classes: "excursion", "aunt's birthday", and "trip". Next, we will assign to each of these values a vector that has a one at the appropriate position, and zero at all remaining positions. That will, In order, be the values of (1, 0, 0), (0, 1, 0) and (0, 0, 1). Here it is important to consistently adhere to the chosen order of classes.
Below, we will get to know the two algorithms used for the binary classification task. The problem of multiclass classification can be solved through specially designed algorithms, but also through several associated binary classifiers. We will bring closer two such techniques called "one against all" and "one on one".
Let's imagine that we have three classes: red, green and blue. The "one against all" approach implies that we need to learn three binary classifiers: one that distinguishes the green class from the rest (unions of the red and blue classes), one that distinguishes the blue class from the rest (unions of the green and red classes), and one that distinguishes the red class from the rest (unions of the green and blue classes). When we need to classify a new instance, we run each of the three binary classifiers and apply the principle of highest confidence to the results obtained: the instance joins the class whose classifier is the safest. We'll see soon how the safety of the classifier is assessed.



One-vs-All classification

Again, imagine that we have three classes: red, green, and blue. The one-to-one approach involves training binary classifiers that can distinguish between each of the class pairs: red and green, green and blue, and red and blue. In general, if we have n classes, the number of binary classifiers we need to train is $\frac{n\cdot(n-1)}{2}$. When we need to classify a new instance, we run each of the learned classifiers and apply the principle of majority voting to the results obtained: the instance joins the class for which the largest number of classifiers vote.

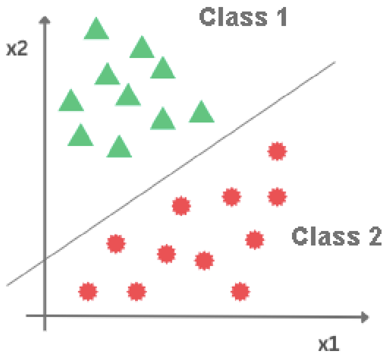


One-vs-One classification

3.6 Logistic regression

Logistic regression is a well-known algorithm used to create binary classification models. In addition to knowing which class an instance belongs to, it also calculates the probability of belonging to that class.

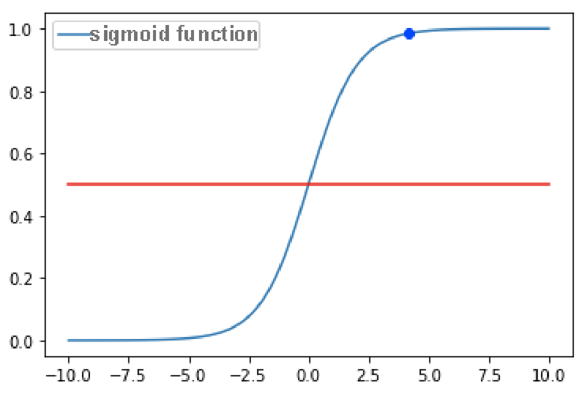
Let's imagine that we have a data set with two attributes, X_1 i X_2, and that the instances of this set are shown as in the figure below. Along the x-axis, the X_1, attribute is represented, along the y-axis, the X_2, attribute, and the color of the dots indicates the class to which each of these instances belongs. You will agree that a linear model that determines a line in a plane could help us solve the classification problem by separating the classes - one below this line and the other above. In order to be able to conclude in this way, we will benefit from the sigmoid function.



The sigmoid function is a popular function in the machine learning story. It is determined by the equation

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Her picture looks like the one in the picture below.



Graph of sigmoid function

We can immediately notice that this function takes a range of values from 0 to 1. The smaller the values of x, the closer the value of this function is to 0 and, similarly, the larger the value of x the closer the value of the sigmoid function is 1. For x= 0, the value of the sigmoid function is 0.5. If we declare this value as a threshold and introduce the rules:

1. If the value of the sigmoid function is greater than or equal to 0.5, associate x with a positive class
2. If the value of the sigmoid function is less than the threshold of 0.5, associate x with the negative class

we will get a function suitable for the classification task.

It also seems to us that the higher the values of x, the more convincing the decision to associate x with a positive class, because we significantly exceed the threshold value. It also seems that the smaller the values of x, the more plausible the decision to join x to the negative class, because we are significantly below the threshold value. For values of x, which are around zero, these arguments are weaker. Therefore, the sigmoid function can also be associated with the interpretation of the probability of belonging to a class.

If we connect the sigmoid function and the equation of the linear model, we get the equation of the logistic regression model, which in general is

$$y = \sigma(X_1, X_2, \dots, X_n) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \dots + \beta_n X_n)}}.$$

The arguments $X_1, X_2, \dots, X_n$ denote attributes in the data set, while its uu values range from 0 to 1 and, as we have seen, make sense for the classification task. To this equation we can also add the following geometric interpretation: the data is classified either below or above the "line" that is determined by the linear relationship equation that we initially imagined.

If we have exactly one attribute, the "right" we are referring to is the real thing. If we have exactly two attributes, the "right" is actually flat in space. If we have more than two attributes, they are "true", in mathematical language, hyperplane.

Cross-entropy

The error function that characterizes logistic regression is called **cross-entropy**. Let's first get to know the intuition that lies behind this function, and then let's get to know its mathematical form.

We have said that we interpret the value calculated by the logistic regression model as the probability of belonging to one of the classes, and that we are guided by the rule that if this value exceeds the threshold of 0.5, we interpret it as belonging to a positive class, and if this value is less than 0.5, we interpret it as belonging to a negative class. If the probability value is 0.5, it is interpreted as belonging to a positive class.

We calculate the error function on the training set. In it, we know for each instance what the exact characteristics are, so we can always compare them with the characteristics that he calculated, i.e. I've joined the model.

Suppose that for three instances belonging to a positive class, the logistic regression model calculated the values 0.94, 0.56, and 0.3 respectively. In the first case, the value is close to the unit, so it indicates a certain decision of the model. In the second case, this value is smaller and closer to the classification threshold, but sufficient for a good decision of the model. In the third case, the value is below the threshold and would cause the model to make an error. When designing an error function, we want to penalize more model calculations that deviate more from the value of 1 for positive instances, i.e. to make their contributions to the overall error of the model greater. One such function that satisfies the required property is $-log(x)$, the graph of which is shown in the figure below. We need a minus sign for the error to get a positive value because the logarithm is negative for the values of the function argument that are from 0 to 1. In the graph we can also see that the values of the function are small for arguments closer to 1, i.e. That the values of the function are greater for arguments that are closer to zero. So now, in order, the contributions to the total error of the extracted instances will be $-log(0.94) = 0.062, -log(0.56) = 0.579, -log(0.3) = 1.203$ and exactly the size ratio we wanted. We can also record them in a table, the way we did and in the linear regression task. In the first column, we will place the class marker (the exact value), in the second column the probability p calculated by the model, and in the third column we will enter the value $-log(p)$. Note that the column name says $-y * log(p)$, but since $y = 1$ this is the same as $-log(p)$.

contributions to the overall error	class marker	probability p	-y*log(p)
	1	0.94	0.062
	1	0.56	0.579
	1	0.3	1.203

Let us now select three instances of the negative class and discuss the expectations we have of the error function in their case. Let the probabilities, respectively, calculated by the logistic regression model be 0.03,0.48 and 0.74. Now, in the first case, the value of the model is close to zero, so it indicates a certain decision to belong to the negative class. In the second case, this value is close to the classification threshold, but it isbelow it, so again it is enough for the model to decide on a negative class. In the case of the third instance, the probability value is over the threshold, so the model will err and classify the instance as positive. What we expect from the error function for negative instances is that their share of the total error is as high as possible the farther away they are from zero. One such function that satisfies this property is $-log(1-p)$ and its graph is shown in the figure below. Again, we use a function with a minus sign to make the error value positive. We can now write the values of this function in a table. The first column now contains the instances with a value of 0, the second column contains the probabilities p that the model calculated, and the last column contains the values of the error functions $-log(1-p)$. Since y = 0 for all instances, the symbol in the column name $-(1-y)*(1-p)$ does not change anything.

contributions to the overall error	class marker	probability p	-(1-y)*log(1-p)
	0	0.03	0.030
	0	0.48	0.654
	0	0.74	1.347

The total value of the cross-entropy functions is obtained when the error contributions of all positive and all negative instances are added together (similar to what we did in the linear regression and mean-squared error problem). This is written in the form

$$-\sum_{i=1}^{N}\left(y_i \cdot log(p_i) + (1-y_i) \cdot log(1-p_i)\right)$$

where the first factor sums the contributions of the errors of the positive instances, and the second factor contributes the errors of the negative instances. The value of yi is the exact characteristic of the class in the training set, and pi is the probability calculated by the logistic regression model. This error is called **binary crossentropy**.

The values of unknown β parameters in the logistic regression model are found by selecting the parameter value for which the cross-error function is the smallest. The gradient descent technique can help us in this case as well.

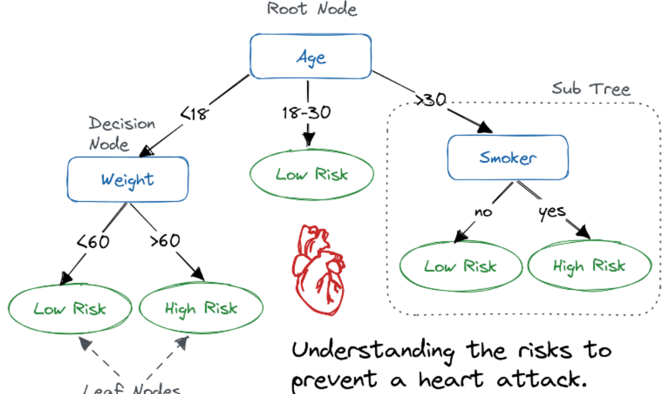
Now let's get to know a slightly different classification algorithm.

3.7 Decision tree

Decision Trees are versatile Machine Learning algorithms that can perform both classification and regression tasks, and even multioutput tasks. They are very powerful algorithms, capable of fitting complex datasets.

A decision tree is a flowchart-like tree structure where an internal node represents a feature (or attribute), the branch represents a decision rule, and each leaf node represents the outcome.

The topmost node in a decision tree is known as the root node. It learns to partition on the basis of the attribute value. It partitions the tree in a recursive manner called recursive partitioning. This flowchart-like structure helps you in decision-making. It's visualization like a flowchart diagram which easily mimics the human level thinking. That is why decision trees are easy to understand and interpret.



A decision tree is a white box type of ML algorithm. It shares internal decision-making logic, which is not available in the black box type of algorithms such as with a neural network. Its training time is faster compared to the neural network algorithm.

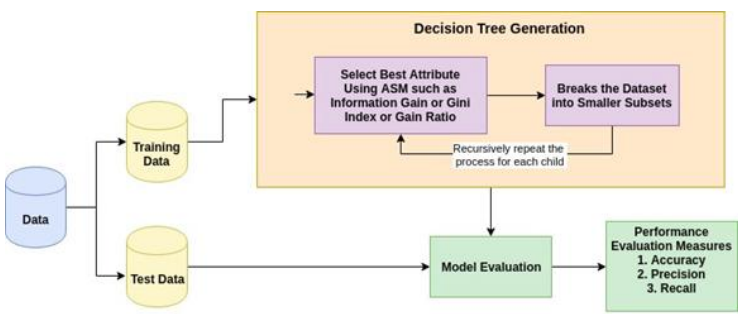
The time complexity of decision trees is a function of the number of records and attributes in the given data. The decision tree is a distribution-free or non-parametric method which does not depend upon probability distribution assumptions. Decision trees can handle high-dimensional data with good accuracy.

How Does the Decision Tree Algorithm Work?

The basic idea behind any decision tree algorithm is as follows:

1. Select the best attribute using Attribute Selection Measures (ASM) to split the records.
2. Make that attribute a decision node and break the dataset into smaller subsets.
3. Start tree building by repeating this process recursively for each child until one of the conditions will match:
    - All the tuples belong to the same attribute value.

- There are no more remaining attributes.
- There are no more instances.



Attribute Selection Measures

Attribute selection measure is a heuristic for selecting the splitting criterion that partitions data in the best possible manner. It is also known as splitting rules because it helps us to determine breakpoints for tuples on a given node. ASM provides a rank to each feature (or attribute) by explaining the given dataset. The best score attribute will be selected as a splitting attribute. In the case of a continuous-valued attribute, split points for branches also need to define. The most popular selection measures are Information Gain, Gain Ratio, and Gini Index.

Information Gain

Claude Shannon invented the concept of entropy, which measures the impurity of the input set. In physics and mathematics, entropy is referred to as the randomness or the impurity in a system. In information theory, it refers to the impurity in a group of examples. Information gain is the decrease in entropy. Information gain computes the difference between entropy before the split and average entropy after the split of the dataset based on given attribute values. ID3 (Iterative Dichotomiser) decision tree algorithm uses information gain.

$$Info(D) = -\sum_{i=1}^{m} pi \cdot log_2 pi$$

Where Pi is the probability that an arbitrary tuple in D belongs to class Ci.

$$Info_A(D) = \sum_{j=1}^{V} \frac{Dj}{D} X Info(D_j)$$

$$Gain(A) = Info(D) - Info_A(D)$$

Where:

- Info(D) is the average amount of information needed to identify the class label of a tuple in D.
- |Dj|/|D| acts as the weight of the jth partition.
- InfoA(D) is the expected information required to classify a tuple from D based on the partitioning by A.

The attribute A with the highest information gain, Gain(A), is chosen as the splitting attribute at node N().

Gain Ratio

Information gain is biased for the attribute with many outcomes. It means it prefers the attribute with a large number of distinct values. For instance, consider an attribute with a unique identifier, such as customer ID, that has zero info(D) because of pure partition. This maximizes the information gain and creates useless partitioning.

C4.5, an improvement of ID3, uses an extension to information gain known as the gain ratio. Gain ratio handles the issue of bias by normalizing the information gain using Split Info. Java implementation of the C4.5 algorithm is known as J48, which is available in WEKA data mining tool.

$$Info_A(D) = \sum_{j=1}^{V} \frac{Dj}{D} \times \log_2\left(\frac{|D_j|}{|D|}\right)$$

Where:

- $\frac{|D_j|}{|D|}$ acts as the weight of the jth partition.
- v is the number of discrete values in attribute A.
- The gain ratio can be defined as

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}$$

The attribute with the highest gain ratio is chosen as the splitting attribute (Source).

Gini index

Another decision tree algorithm CART (Classification and Regression Tree) uses the Gini method to create split points.

$$Gini(D) = 1 - \sum_{i=1}^{m} Pi^2$$

Where pi is the probability that a tuple in D belongs to class Ci.

The Gini Index considers a binary split for each attribute. You can compute a weighted sum of the impurity of each partition. If a binary split on attribute A partitions data D into D1 and D2, the Gini index of D is:

$$\mathrm{Gini}_A(D) = \frac{|D_1|}{|D|}\mathrm{Gini}(D_1) + \frac{|D_2|}{|D|}\mathrm{Gini}(D_2)$$

In the case of a discrete-valued attribute, the subset that gives the minimum gini index for that chosen is selected as a splitting attribute. In the case of continuous-valued attributes, the strategy is to select each pair of adjacent values as a possible split point, and a point with a smaller gini index is chosen as the splitting point.

$$\Delta Gini(A) = Gini(D) - Gini_A(D)$$

The attribute with the minimum Gini index is chosen as the splitting attribute.

Decision Tree Classifier Building in Scikit-learn

Importing Required Libraries

Let's first load the required libraries.

```
# Load libraries
import pandas as pd
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
```

Loading Data

Let's first load the required Pima Indian Diabetes dataset using pandas' read CSV function. You can download the Kaggle data set to follow along.

```
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age', 'label']
# load dataset
pima = pd.read_csv("diabetes.csv", header=None, names=col_names)
pima.head()
```

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Feature Selection

To understand model performance, dividing the dataset into a training set and a test set is a good strategy.

Let's split the dataset by using the function train_test_split(). You need to pass three parameters features; target, and test_set size.

```
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
# 70% training and 30% test
```

Building Decision Tree Model

```
# 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)
```

Evaluating the Model

Let's estimate how accurately the classifier or model can predict the type of cultivars.
Accuracy can be computed by comparing actual test set values and predicted values.

```
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

Accuracy: 0.6753246753246753

We got a classification rate of 67.53%, which is considered as good accuracy. You can improve this accuracy by tuning the parameters in the decision tree algorithm.

Visualizing Decision Trees

You can use Scikit-learn's *export_graphviz* function for display the tree within a Jupyter notebook. For plotting the tree, you also need to install graphviz and pydotplus.

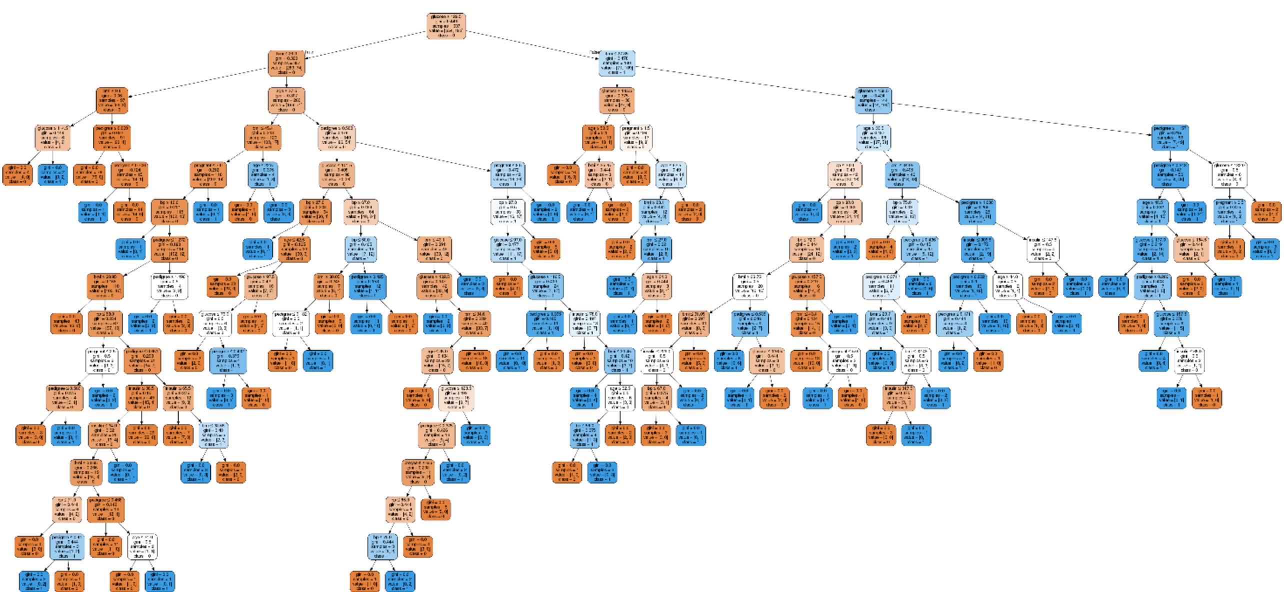
pip install graphviz

pip install pydotplus

The *export_graphviz* function converts the decision tree classifier into a dot file, and pydotplus converts this dot file to png or displayable form on Jupyter.

```
from sklearn.tree import export_graphviz
from sklearn.externals.six import StringIO
from IPython.display import Image
import pydotplus

dot_data = StringIO()
export_graphviz(clf, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True,feature_names =
feature_cols,class_names=['0','1'])
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('diabetes.png')
Image(graph.create_png())
```



In the decision tree chart, each internal node has a decision rule that splits the data. Gini, referred to as Gini ratio, measures the impurity of the node. You can say a node is pure when all of its records belong to the same class, such nodes known as the leaf node.

Here, the resultant tree is unpruned. This unpruned tree is inexplicable and not easy to understand. In the next section, let's optimize it by pruning.

Optimizing Decision Tree Performance

- **criterion : optional (default="gini") or Choose attribute selection measure.** This parameter allows us to use the different-different attribute selection measure. Supported criteria are "gini" for the Gini index and "entropy" for the information gain.

- **splitter : string, optional (default="best") or Split Strategy.** This parameter allows us to choose the split strategy. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

- **max_depth : int or None, optional (default=None) or Maximum Depth of a Tree.** The maximum depth of the tree. If None, then nodes are expanded until all the leaves contain less than min_samples_split samples. The higher value of maximum depth causes overfitting, and a lower value causes underfitting (Source).

In Scikit-learn, optimization of decision tree classifier performed by only pre-pruning. Maximum depth of the tree can be used as a control variable for pre-pruning. In the following the example, you can plot a decision tree on the same data with max_depth=3. Other than pre-pruning parameters, you can also try other attribute selection measure such as entropy.

```
# Create Decision Tree classifer object
clf = DecisionTreeClassifier(criterion="entropy", max_depth=3)
# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)
#Predict the response for test dataset
y_pred = clf.predict(X_test)
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

Accuracy: 0.7705627705627706

Well, the classification rate increased to 77.05%, which is better accuracy than the previous model.

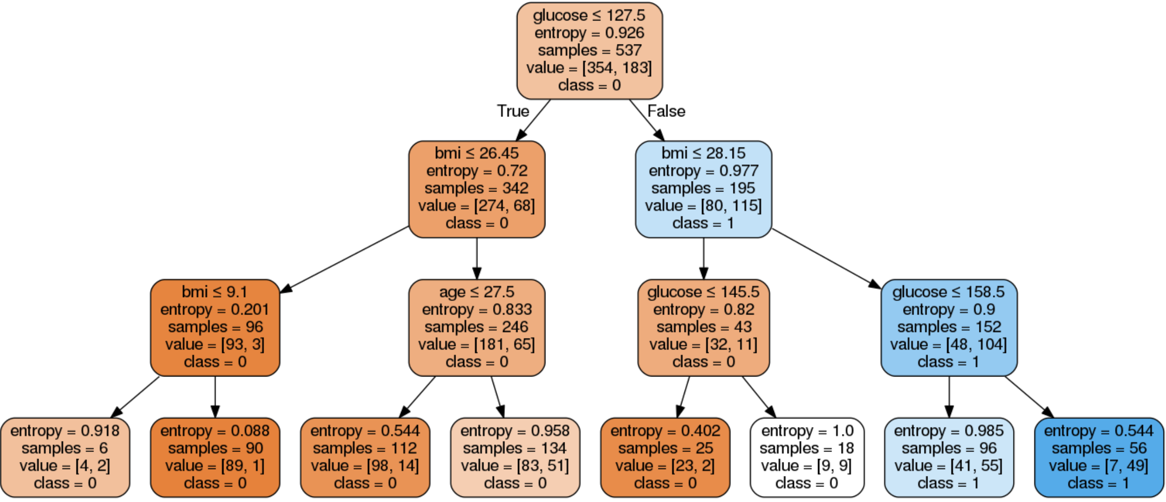
Visualizing Decision Trees

Let's make our decision tree a little easier to understand using the following code:

```
from six import StringIO from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus
dot_data = StringIO()
export_graphviz(clf, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True,                  feature_names =
feature_cols,class_names=['0','1'])
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('diabetes.png')
Image(graph.create_png())
```

Here, we've completed the following steps:

- Imported the required libraries.
- Created a `StringIO` object called `dot_data` to hold the text representation of the decision tree.
- Exported the decision tree to the `dot` format using the `export_graphviz` function and write the output to the `dot_data` buffer.
- Created a `pydotplus` graph object from the `dot` format representation of the decision tree stored in the `dot_data` buffer.
- Written the generated graph to a PNG file named "diabetes.png".
- Displayed the generated PNG image of the decision tree using the `Image` object from the `IPython.display` module



As you can see, this pruned model is less complex, more explicable, and easier to understand than the previous decision tree model plot.

Decision Tree Pros

- Decision trees are easy to interpret and visualize.
- It can easily capture Non-linear patterns.
- It requires fewer data preprocessing from the user, for example, there is no need to normalize columns.
- It can be used for feature engineering such as predicting missing values, suitable for variable selection.
- The decision tree has no assumptions about distribution because of the non-parametric nature of the algorithm. (Source)

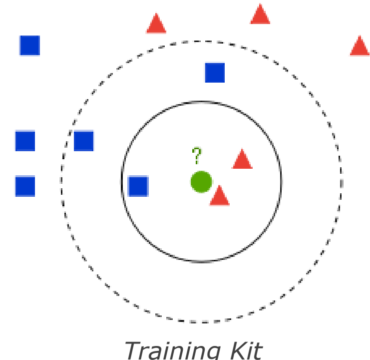
Decision Tree Cons

- Sensitive to noisy data. It can overfit noisy data.
- The small variation (or variance) in data can result in the different decision tree. This can be reduced by bagging and boosting algorithms.
- Decision trees are biased with imbalance dataset, so it is recommended that balance out the dataset before creating the decision tree.

3.8 K-Nearest Neighbor (kNN) Algorithm

There are also non-parametric machine learning models. The model obtained using the k-nearest neighbor algorithm is just like that. Let's find out how it works!

Let our training set consist of pairs of numbers (x_1, x_2) and the corresponding class names. Pairs can be represented as points in the plane, where the first coordinate x_1 denotes the value on the x-axis and the second coordinate x_2 denotes the value on the y-axis. In practice, the values of x_1 and x_2 are always associated with some specific attributes, for example, temperature and humidity, but now we can think of them as some general values. Each pair of numbers belongs to one of two classes: red triangles or blue squares. Since there are only two classes, you can assume that it is a binary classification. Now imagine that the green circle represents a new instance, a new pair of numbers, for which we need to determine which class it belongs to: whether it is a red triangle or a blue square.



Training Kit

The k-nearest neighbor algorithm is a classification algorithm that says that we first fix the number of neighbors (surrounding instances) k to a specific value and then determine how many of the k-closest neighbors there are red and blue: the red neighbor is an instance belonging to the red class, and the blue neighbor is an instance belonging to the blue class. For example, if we fix the number k to 3, the three closest neighbors of the green circle are inside a solid circle. There are two red triangles and one blue square.

Further, the k-nearest neighbor algorithm says that the new instance, a new pair of dots is added to the class of the more numerous neighbor: if the red neighbors are more numerous, we say that the new instance belongs to the red class, and, similarly, if the blue neighbors are more numerous, we say that the new instance belongs to the blue class. You can also think of this as the saying "you can tell a man by the company" in the world of machine learning.

In our example, when the value of k is fixed at 3, we conclude that we should associate the green circle with the red class because we have two red neighbors and one blue neighbor.

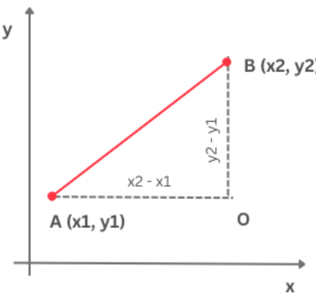
Let's see what happens if we fix the number k at 5. In the figure this neighborhood is shown by a dashed circle. Since there are now three blue squares and two red triangles, the conclusion would be that the green circle should be joined to the blue class.

This section is paired with Exercise 8 and Jupyter Notebook 08-k-nearest_neighbors.ipynb. To follow the content further, click on the link and then on the button Open in Colab to open the content in *Google Collab*. If you are viewing the notebooks on your local machine, find the notebook with the same name among the contents and run it. For more detailed instructions, see the *Hands-on Zone* section and the *Jupyter Notebook Practice Lesson*.

The accompanying material contains the aforementioned set of points and an application in which you can examine what will happen if you choose a different value of the number k. Since the algorithm needs to decide which neighbors there are more, it is wise to choose odd values of the number k.

Note that in addition to the number of neighbors k, the result of the algorithm also depends on how we measure the distances to the neighbors! To find the nearest neighbors, we need to somehow measure the distance to them.

Until now, we have encountered a distance called the Euclidean distance. The Euclidean distance between the points A and B is calculated as the length of the lengths connecting the points A and B. For example, for the points A =(0,0) i B =(3,4) the Euclidean distance is calculated as $\sqrt{(3-0)^2+(4-0)^2} = 5$



Euclidean distance

There are many other distances as well. For example, you may be intrigued by the Manhattan distance. Unlike the Euclidean distance, which calculates the "hypotenuse" of a triangle defined by the points A and B and O (if we follow the previous figure), the Manhattan distance calculates the sum of the "legs" of this triangle. For points A and B, the value of the Manhattan distance would be $|\, 3-0 \,| + |\, 4-0 \,| = 7$.

Which distance we choose depends on the nature of the task and the meaning that the attributes we are working with have. In general, we can try more distances and choose the one for which we get the best results. We will talk about this later. It is important to note that a function must satisfy certain mathematical properties in order to be declared a distance, so not every function can be helpful.

Just like other machine learning algorithms, the x-nearest neighbor algorithm is trained over the training set. It is interesting to note that the learning phase in this algorithm is actually reduced to storing the dataset only. In other algorithms, such as linear regression or logistic regression, we have seen that in this phase, the values of some parameters that appear in the model are calculated by looking for the minimum error function. The x-nearest neighbor algorithm is not like that. The mapping we learn It's not about a specific function, it's about the data itself and the steps that need to be taken. That is why it is common to call models that have this property **nonparametric models**.

The k-nearest neighbor algorithm performs all the work during the application, i.e. Decide which class the new instance belongs to. When we need to classify a new instance, we first calculate the distance of the new instance from all instances in the training dataset. Next, we sort these distances from smallest to largest. We keep the first k distances (because they are the distances to k closest neighbors) and choose instances from the training set to which they refer. We continue to monitor what is happening in the space of their landmarks and look for the most numerous landmarks, i.e. the largest class. As we saw in the introductory example, the new instance should be associated with the class that is the most numerous.

This algorithm is easy to implement, so let's roll up our sleeves and get started!

Let's imagine that we are working with a set of data that we have used so far and that each instance has a form (x1, x2), where the mark is the value 0 for red or 1 for blue.

To measure the distance between instances, we will use the euclidean_distance function, which is defined by the following block of code:

```
def euclidean_distance(instance1, instance2):
  return np.sqrt((instance1[0]-instance2[0])**2 + (instance1[1]-instance2[1])**2)
```

The x-nearest neighbor algorithm itself is represented by the following block of code:

```
def kNN(k, instances, new_instance, classes={0: 'red', 1: 'blue'}):
# first, calculate the distances between the new instance and all instances in the dataset
  distances = [euclidean_distance(instance, new_instance) for instance in instances]
  # then sort the distances, extract the k smallest ones and the corresponding instances
  # declare them as neighbors
  neighbors = np.argsort(distances)[0:k]
  # then read the labels of the neighbors and count them
  neighbor_labels = [instances[neighbor][2] for neighbor in neighbors]
  label_counts = np.bincount(neighbor_labels)
  # the label of the new instance will be the label of the most frequent neighbor
  label = np.argmax(label_counts)
  return classes[label]
```

In it, as we have discussed, we carry out the following steps:

1. We calculate the distance from the new instance to all instances in the data set.
2. And then we put them in the middle of the night, and then we put them in the dark.
3. And we are the ones who have the right to be the neighbors.
4. In the set of isolated neighbors, we count the most numerous,
5. We conclude that the new instance belongs to the class of the most numerous neighbor.

3.9 Hyperparameters

We have seen that in the x-nearest neighbor algorithm it is necessary to fix the value of the number k in advance, and that different choices lead to different conclusions. How do we know which value to choose? This question follows all other machine learning algorithms in which some values appear that we need to define in advance. Such values are called **hyperparameters** or **metaparameters.**

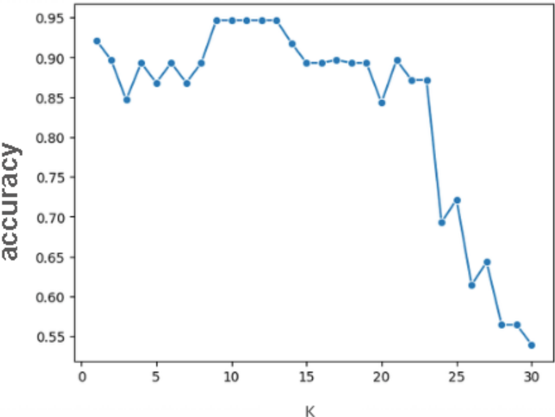
We mentioned that when dividing a dataset, we always single out a training set, a testing set, and a validation set. We have not used the validation set so far. In fact, we need it whenever there are some hyperparameters in our learning algorithm whose best value we need to determine. The story we're going to share applies to all algorithms, but we're going to continue to use the k-nearest neighbor algorithm.

Let's go back to the question of how to choose the best value of the hyperparameter k. It is natural to think: we will try multiple values, for example, all numbers from 1 to 10, and then we will choose the best value! We're going to do that, but we're going to be very careful about where we try how good our choice is. If we do this on a test suite, we will be breaking the golden rule of machine learning about the strict separation of the test suite and model development: we will use the test suite to decide what the best value of the

hyperparameter k is, and then, when we train the model, we will again use the test suite to evaluate how good it is! You'll agree that it doesn't make much sense!

It is correct to do the following: we will test which hyperparameter values are best on the validation set. This set does not share information with either the training set or the test set, so it will contribute to the objectivity of our conclusions. Now that we have established this, we can get to work determining the best value of the hyperparameter k.

For each of the values of the hyperparameter k that we want to test, we will separately train the model on the training set and calculate its quality measure on the validation set. In this case, it's accurate. The obtained values can be displayed graphically by placing different values of the parameter k along the x-axis, and accuracy values along the y-axis. The value of the hyperparameter k for which we get the best value of the quality measure on the validation set is the value of the hyperparameter we are looking for. This is usually seen in the graph as the region where the values are the highest.



Demonstrating the accuracy of the model on the validation set

Based on the previous graph, we can see that the optimal values of the hyperparameter k are actually 9, 10, 11, 12, and 13 because they all result in the same, highest accuracy of the model.

Similar graphics can be drawn for hyperparameter values and error functions. Then we set different values of the hyperparameter along the x-axis, and the values of the error function along the y-axis. Now it is important to note the values of the hyperparameter for which the error function is smallest.

When multiple hyperparameters are present in a learning algorithm, the goal is to find the best combination of hyperparameters. We also determine it based on the validation set by tracking the success of the model and hunting for the combination that gives the best value of the quality measure (or, equally, tracking the error of the model and hunting for the combination that gives the least value of error). The trouble is that this process can be quite slow and computationally demanding for a large number of hyperparameters: for example, if we want to examine 10 different values of k and 3 different distance functions, we actually have 10 x 3 = 30 different combinations, so we have to train and evaluate 30 different models.

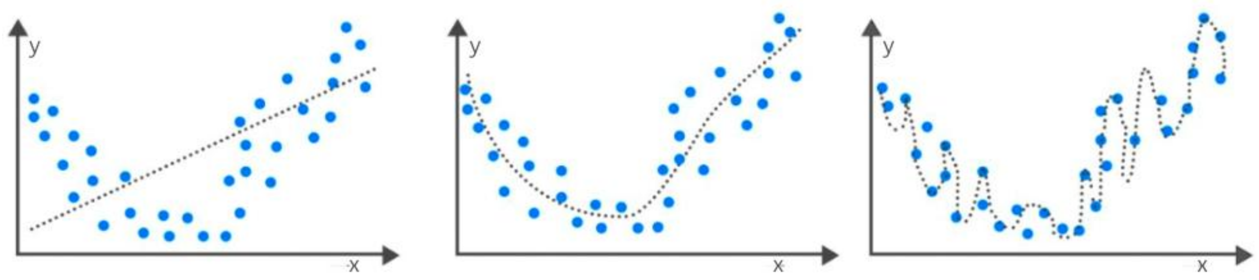
3.10 Generalization, under-adaptation, and over-adaptation

In this volume, we will learn about the concepts of generalization, over adaptation, and under adjustment that are often encountered in the story of machine learning.

Imagine that Mike, Ana and Luka are preparing for a math exam and that they all use the same collection. Mike was slacking off and only practicing the tasks only slightly, Ana was diligent and practiced carefully and comprehensively all week, while Luka decided to memorize the tasks. Can you guess who did the best in the test? Of course, Ana!

A collection of problems can be thought of as an abstract set of data consisting of inputs (task texts) and outputs (solutions). A machine learning model can, like Pera, learn only a few connections in the data and make a lot of mistakes in practice. Such a property of the model is called **underfitting**. The model can also overdo the level of detail, like Luke, and lose the power to handle some new data. Such a property of the model is called **overfitting**. It would be best if the model adopted the right information and could, like Anna, successfully solve both familiar and some new tasks. This property of the model is called **generalization**.

An example of under-adaptation and over-adaptation can be illustrated by the following figure. Imagine that along the x-axis are the values of an attribute, along the y-axis are the values of the target variable, and that the dashed line shows the model. The model on the left is not the best choice considering the arrangement of the dots, it seems too simple. The data looks more like a "glass," so a square model that has this form might be a better solution. We can see him in the middle picture. In the image on the right, we see a model that consistently follows every point in the data set and is completely adapted to it.



Example of under-adaptation and over-adaptation

The task of finding the optimal model and balancing between underfitting and overfitting is not easy. Fortunately, the field of machine learning defines the protocols and techniques that we can use to keep track of each of these situations. For example, large differences in the performance of the model on the training set and the test set indicate that the model has been readapted. This is usually due to choosing a more complex model than necessary (as in the upper right image) or training the model for a longer time. On the other hand, low values of quality measures in both the training set and the test set indicate that the model has not learned enough from the data, is too simple (as in the upper left image), or needs more attributes.

Good generalization is a property that allows machine learning models to be successfully applied in practice. Only a small part of the data that is available is used to train them, and yet, we expect them to behave well during the application and over new data that they have never encountered. That is why it is important that the datasets are representative, i.e. they are both sufficiently rich and diverse to suit the problem being addressed, as well as careful monitoring of possible under-adaptations and over-adaptations of the model.

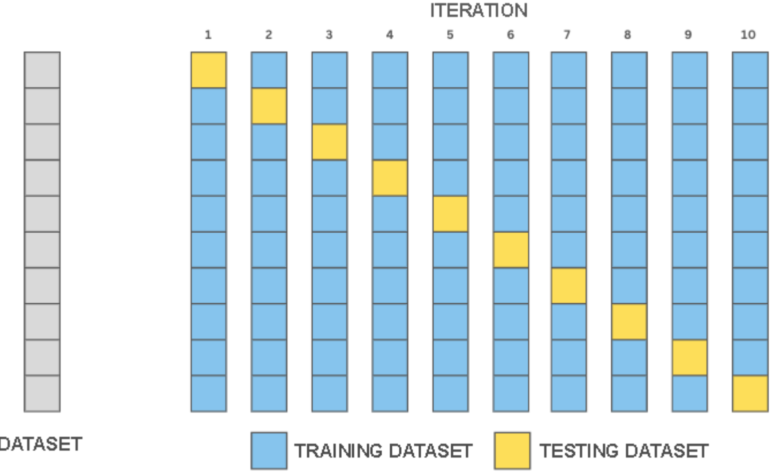
© 2025 https://book.tsp.edu.rs

3.11 Validation, cross-validation

We've said many times that before applying a machine learning algorithm, data is divided into a training set and a testing set (we include the validation set when we need it). We also mentioned that the division process is random. You may have wondered if some different divisions, compared to the ones we have chosen, would lead to different results of the model's work. Perhaps it is for a specific division of the data set that we get more optimistic results or drastically worse. And it's a kind of adjustment.

Whenever the sizes of the datasets and the selected algorithms allow, it is desirable to actually perform multiple divisions of the initial dataset into a training set and a testing set so that each instance in the dataset gets the opportunity to be found in both sets. One such procedure that we will describe is called **cross-validation**. In the example, we will use a linear regression algorithm, but the story is general and applies to all algorithms.

Let's divide the dataset into 10 parts as in the figure below. In the first step, we extract the first part of the test set and keep the remaining nine parts for training. To make it easier for you to follow, the test set is colored yellow in the image, and the training sets are colored blue. Now let's train the first linear regression model on the training set and calculate the value of its mean square error on the test set. The resulting value can be marked with MSE_1. In step two, we separate the second part of the test set, and the remaining nine parts for the training set. Now in the picture the second part is painted yellow, and the remaining parts are painted blue. Let's retrain the linear regression model on the training set (that's the second model now) and calculate the value of its mean square error on the test set. Now let's mark this value with MSE_2. Let's continue this process until we get to the last, tenth, part: now we will keep it as a test set, and we will use the remaining parts to train the model. We will train the tenth linear regression model on it and then calculate the mean square error MSE_{10} on the test set.



Cross-validation with 10 layers

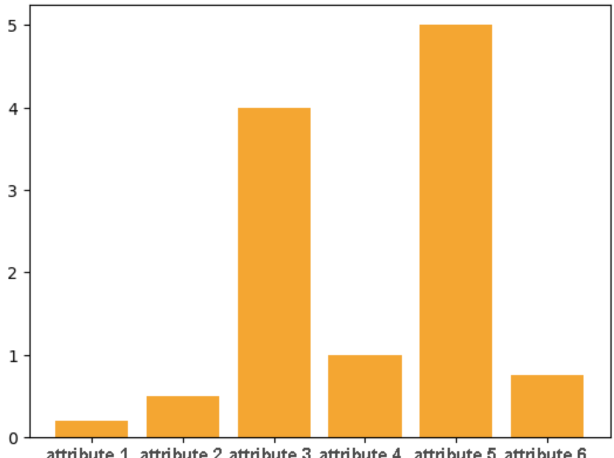
Since we have 10 different divisions of the data set, we also have 10 different values of the mean square error. The average of the obtained values $(MSE_1 + MSE_2 + ... + MSE_{10})/10$ actually best indicates how our model behaves and helps us solve the dilemmas we had at the beginning regarding the impact of division on the success of the model. What is not very clear is which of the 10 different models we have at our disposal should we choose. The slightest mistake or some other? In fact, we should now train a new model over the entire dataset and continue to use it we approximate its behavior and rate it with the behaviors of each of the 10 trained models.

This process is called 10-layer cross-validation*10-fold cross validation*. In practice, 3- and 5-layer divisions are also used, and the choices depend on the size of the dataset and the type of algorithms used. Also, there is a division in which the number of layers corresponds to the number of instances in the data set, called *leave-one-out cross validation*.

3.12 Regularization

Regularizations are another set of techniques that can be used to control model refits. Their main goal is to prevent complex models, which help us learn a richer set of dependencies in data, from becoming over-adapted.

We will introduce regularization using the example of a linear regression model. Suppose we have trained the model and we have obtained the values of the parameters, the graphical representation of which looks like in the figure.



The parameters that are the largest (absolute) in terms of their value are also the most important for model predictions. In the figure, these are the parameters that correspond to attributes 3 and 5 and their values, as we can see, are significantly higher than the values of the other parameters. In this sense, these attributes can ignore the impact of the remaining attributes on the prediction values, so we can interpret this behaviour of the model as a form of data readjustment.

That is why it is desirable, to some extent, to limit the values of the parameters - we want the model to learn the parameters and that they reflect the properties of the data, but we also want to monitor their value in order to prevent overadjustment. This technique is called **regularization**. In the context of linear regression, we can do this by adding the sum of the squares of the parameters to the mean square error of the model:

$$\frac{1}{N}\sum_{i=1}^{N}\left(y_i - (\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n)\right)^2 + \lambda(\beta_1^2 + \beta_2^2 + \cdots + \beta_n^2)$$

The value of λ in the expression is a hyperparameter that affects the strength of the regularization. If its value is 0, the regularization will have no effect. By giving some non-zero values, we balance the learning determined by the mean square error and the readjustment measured by the values of the sum of the squares of the parameters. Squares are there for technical reasons, first to prevent the values of the coefficients from being suppressed against each other, and then to preserve the properties of the error function for the application of the optimization algorithm. Such an extended form of linear regression supplemented by a regularization term is called ridge regression.

A little later, we will return to the story of regularization when we introduce neural networks. They are very complex models, so they can often be re-adapted to the data. We'll also see how we can follow it.

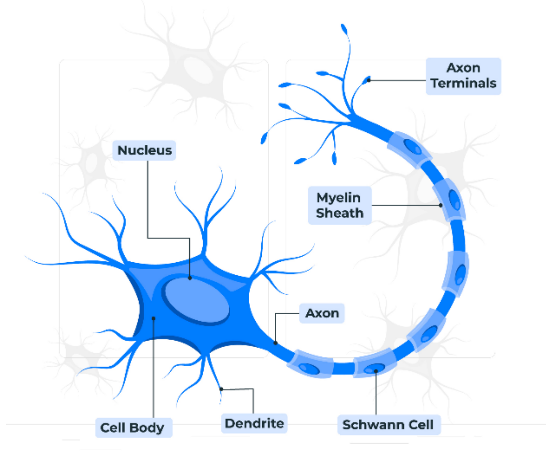
4.1 NEURAL NETWORKS

Welcome to the topic of **Neural Networks**! Neural networks are a key component of deep learning, a subset of machine learning that mimics the structure and function of the human brain. In this section, you will learn about the basic elements of neural networks, including neurons, layers, and activation functions. We will explore different types of neural network architectures, such as convolutional and recurrent neural networks, and their applications in solving complex problems. You will also gain hands-on experience in training and testing neural networks using popular tools and frameworks.

4.1 Neural Networks

In this lesson, we will learn about neural networks, a special group of machine learning algorithms. We owe them a lot of interesting breakthroughs in the world of artificial intelligence.

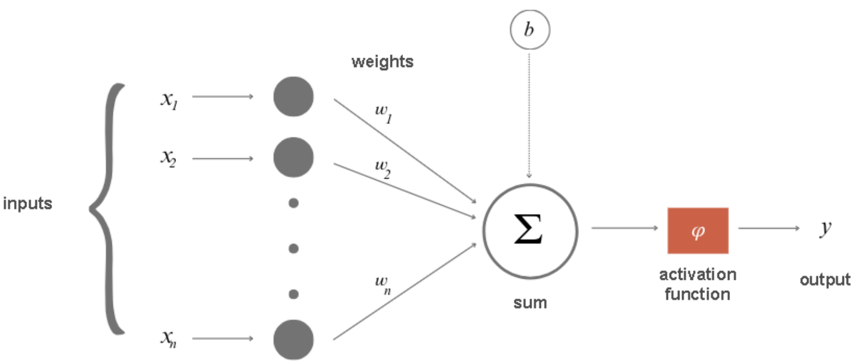
You know from biology classes that the cell is the basic unit of structure and function of all living things. From the point of view of artificial intelligence and learning, the most interesting are brain cells. These are called neurons. Neurons consist of a body with a nucleus and longer and shorter extensions, called axons and dendrites. Extensions allow neurons to connect with other neurons. These connection points of neurons are called synapses. They allow the signals, i.e. electrical impulses generated by one neuron are transmitted to another neuron. Interestingly, one neuron can be connected to millions of other neurons. This means that it receives and processes signals coming from a multitude of other neurons and, based on its internal mechanisms, fine-tunes the signal that it sends to other neurons. It is common for this condition to be called the state of neuronal activation. It lasts only a fraction of a second, but it allows subtle calculations to be made and generates a signal that is transmitted throughout the nervous system.



The neuron we encounter in artificial intelligence is a mathematical abstraction of the brain's neurons. It is described as a function of multiple variables $f(x_1, x_2, \ldots, x_n)$, where each of the variables $x_1, x_2, \ldots, x_n$ corresponds to a single signal that reaches the neuron. Since not all signals are equally important for neuronal activity, they are joined by weights $w_1, w_2, \ldots, w_n$ that should indicate their importance. Higher values of these numbers

© 2025 https://book.tsp.edu.rs

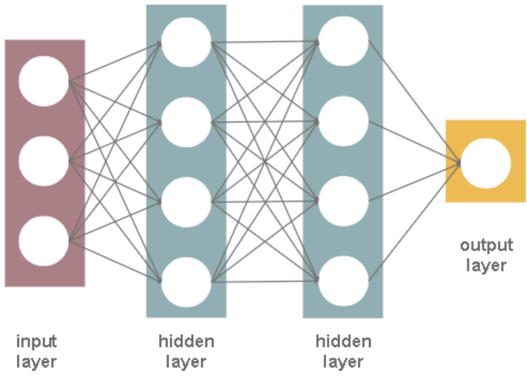
indicate that the signal is more important, and lower values indicate that the signal is less important. Thus, the total stimulation of neurons corresponds to the weight sum $w_1x_1 + w_2x_2 + \cdots + w_nx_n$. In order to influence additional neuronal behaviors, one free term b is added to this sum, so that the total stimulation of the neuron is actually $w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$. This is then passed on to the so-called activation function φ, which has the task of calculating the output of the neuron. Depending on the choice of activation function, the output values that are obtained will also depend. If we now write everything down systematically, we get that for the received signals $x_1, x_2, \ldots, x_n$, the output of the neuron is $y = \varphi(w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$. You can also follow the procedure we have described in the illustration below



Mathematical abstraction of neurons

> Let's take a closer look at the meaning of parameter b. A natural neuron is characterized by the so-called activation threshold - if the total signal received by the neuron is higher than the value of the activation threshold, it is activated, processes the signal and forwards the result of the processing further to other neurons. A similar role in the mathematical model of neurons is played by parameter b. If the total signal is higher than the activation threshold b, i.e. If $w1x1 + w2x2 + \cdots + wnxn > b$, the neuron will be activated. Therefore, parameter b leaves us with the possibility of influencing additional behaviors of neurons. The expression $w_1x_1 + w_2x_2 + \cdots + w_nx_n > b$ can also be written as $w_1x_1 + w_2x_2 + \cdots + w_nx_n - b > 0$, and in this sense the parameter b is also an integral part of the sum.

When neurons are connected to each other, we have **a neural network**. A neural network usually consists of **layers**, especially associated groups of neurons.



Neural network layers

The input layer is a layer that is located at the input of a neural network. The input signals x1, x2,..., xn of this layer are related to the values of the attributes we have in the data set, and thus we approach the practical application of neural networks. For example, if we have a data set containing three attributes, temperature, humidity and atmospheric pressure, the input layer will have three neurons: the first will correspond to the first attribute, temperature, the second will correspond to the second attribute, humidity, and the third neuron to the third attribute, i.e. the atmospheric pressure. For one particular instance of the data set with the values of temperature, humidity and atmospheric pressure amounting to, respectively, 19 °C, 77% and 1011.2 mb, we will have the values of the signal x1 = 19, x2 = 77 and x3 = 1011.2. In the spirit of the previous story, the first neuron of the input layer receives and processes only the signal x1 by passing it through without any modification (this is possible for the selection of the activation function φ(x)= x and the value w_1 = 1 and b = 0). The other two neurons and their signals x_2 and x_3 are also valid. This would mean that the input layer allows us to enter the network.

The output layer is a layer that is located at the output of a neural network. As you can guess, it allows us to read the results that the neural network has calculated for us. Depending on the task being solved, the number of neurons in this layer will also depend

In regression tasks, since we expect a single numerical value as a result (amount of precipitation or something similar), one neuron is enough. Its outcome should correspond to the prediction that we expect. For the classification task, let's consider binary classification and multiclass classification separately. Since binary classification expects two values, 0 or 1, your first thought may be that we need two neurons. However, if you think about it, you will notice that even one neuron is enough: if its output exceeds a threshold, a predefined value, we can take it as a result of 1, or, otherwise, as a result of 0. In the case of a multiclass classification, we can have several classes, so it is practical to introduce one neuron for each class.

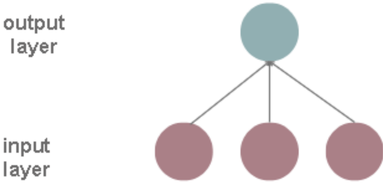
You will agree that in a multiclass classification task, we expect all the outputs of the output layer neurons to be 0, except for one that has a value of 1 - so we will know exactly what class it is.

We call the layers of the neural network between the input and output layers **hidden layers**. Neural networks that have more than one hidden layer are commonly referred to as **Deep Neural Networks**. This is where the name **deep learning** comes from. *Deep Learning* is the area of machine learning that studies them. The name **shallow learning is** for more classical forms of learning.

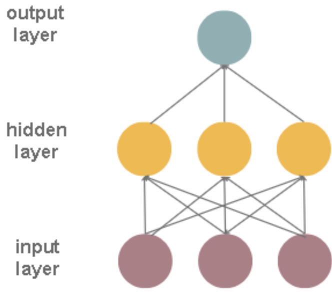
Fully Connected Neural Networks are networks in which each neuron of the previous layer is connected to each neuron of the next layer. The image showing the layers of the neural network also shows one fully connected neural network because all the neurons of the input layer are connected to all the neurons of the first hidden layer, then all the neurons of the first hidden layer are connected to all the neurons of the second hidden layer, and finally, all the neurons of the second hidden layer are connected to all the neurons (only one in our picture) of the output layer. The ways in which the neurons of the layers are connected to each other determines the architecture of the neural networks and some specific properties of the networks that further determine in which areas they can be used. In the next lesson, we're going to get to know some of them.

Now let's consider what we actually got from the introduction of neurons and neural networks. Suppose we have three attributes $x_1, x_2 \ and \ x_3$. The linear relationship between an attribute and a target variable is mathematically described by the equation $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3$. If, instead of the parameters β, we write w a instead of β0, write b and move it to the end, we actually get the weight sum $w_1 x_1 + w_2 x_2 + w_3 x_3 + b$ calculated by one neuron for the signals it receives. This means that if there were no activation function, \(φ\) and neuron would model a linear relationship between attributes (signals) and outputs. This

can also be graphically represented by a network consisting of only an input layer with three neurons and an output layer with one neuron, as in the figure below



If the activation function didn't exist, from a dependency modeling perspective, would adding a new hidden layer make a difference? Let it be a layer of yellow color in the next picture.

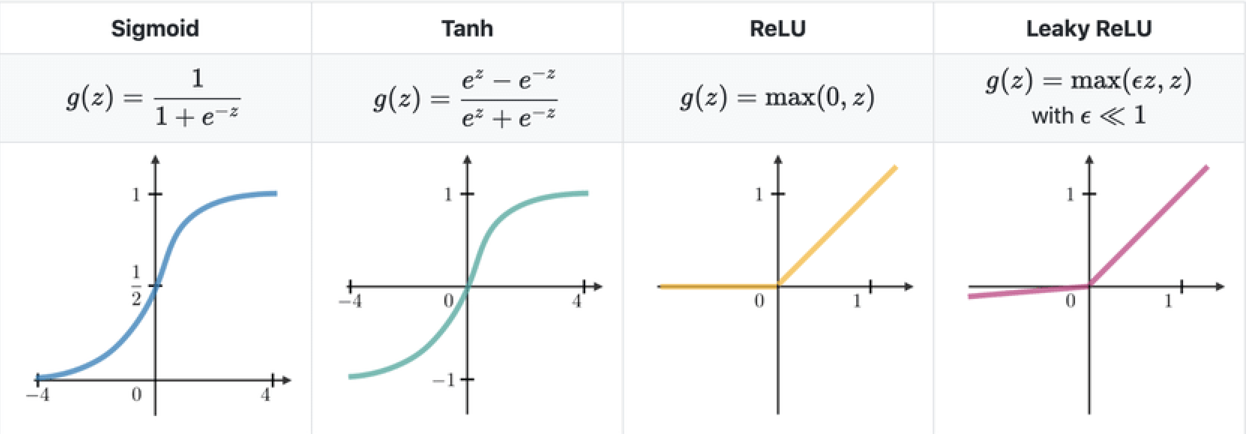


Now each neuron of the hidden layer calculates some linear combination of attributes, and the neuron of the output layer calculates some linear combination of values of the hidden layer. This would mean that our output layer neuron is again calculating some linear combination of attributes, and that we haven't moved much from representing some more complex relationships between attributes and outputs. In addition, we wouldn't move even by adding 100 hidden layers - we'd always be modeling a linear dependence.

That's why the inclusion of an activation function in the calculations of neurons significantly changes the set of possibilities we have. If we use a nonlinear activation function, we will be able to model some nonlinear relationships between the attribute and the target variable. Thus, the existence of a nonlinear activation function in the hidden layer from the previous example allows the output layer neuron to now compute some nonlinear combination of attributes. In this light, adding new layers makes much more sense. By combining the nonlinearities of multiple layers, we can model complex relationships between attributes and outputs.

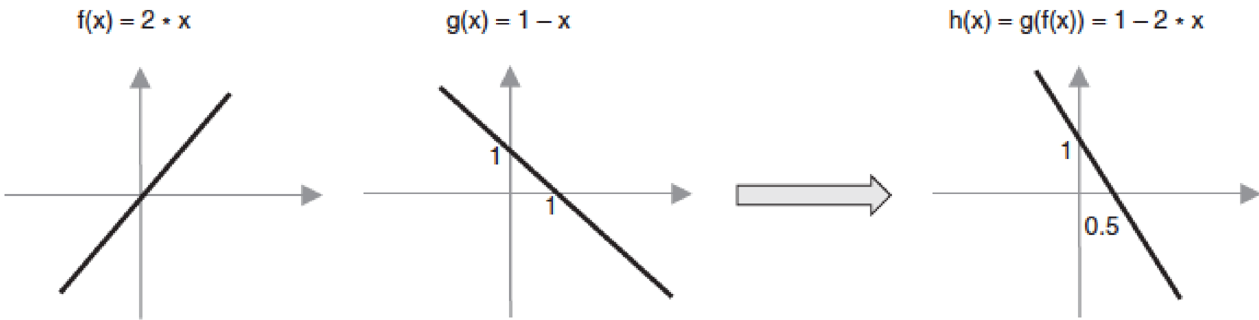
In order to fit all the cubes together, it remains to be discussed what are the nonlinear activation functions that are popular in machine learning. These are the sigmoid function that we got to know in the story of logistic regression, hyperbolic tenges, *Rectified Linear Unit (ReLU*) and Leaky Rectified Linear Unit (Leaky ReLU). The formulas by which these functions are calculated and their graphs are shown in the figure below. As you can see, these functions are not really linear - their graphics are not real.

© 2025 https://book.tsp.edu.rs

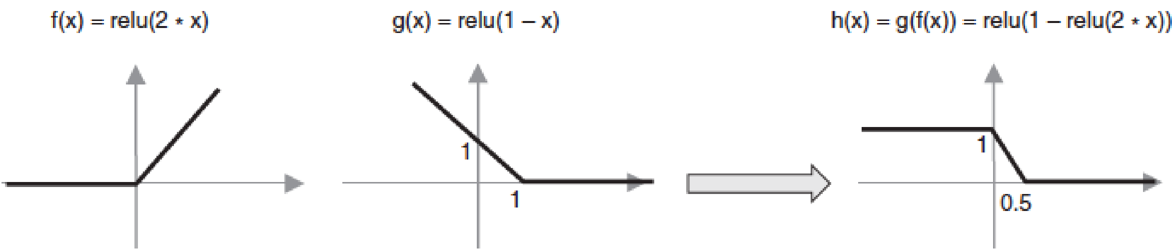


The Most Common Choices of Activation Functions

To complete the story of combining different activation functions, let's look at the functions f(x)= 2x and g(x)= 1 − x. We can see that both functions of a linear function are a variable. By combining them, the composition of the functions, we obtain the function g(f(x))= 1 − 2x, which is also a linear function of a variable. You can see the graphics of all three functions in the image below.



Let us now consider the functions f(x)= ReLU(2x) and g(x)= ReLU(1 − x), which differ from the previous functions in that they feature the activation function of a rectified linear unit. Therefore, both functions are nonlinear. By combining them, i.e. By composing them, we get the function g(f(x))= ReLU(1 − ReLU(2x)), which is also nonlinear, and which has a new "form": it allows us to express a slightly different relationship between the input variable and the output.



The choice of the appropriate activation function depends on the nature of the task and some of the properties that the neural network should have during training. How this is done, we will explain in the next lesson.

4.2 Training of neural networks

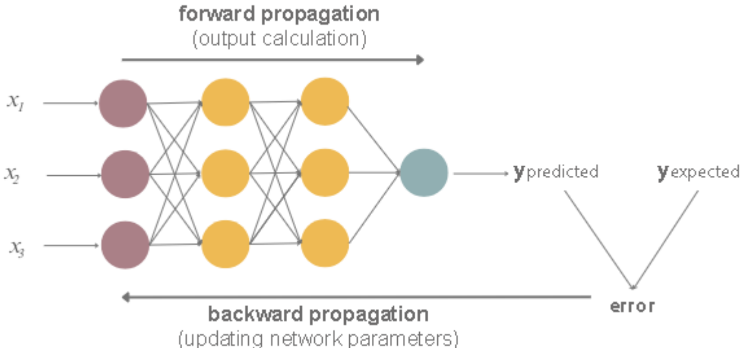
As we have seen, each neuron of a neural network is connected in some way to other neurons in the network. These connections are described by weights w, which actually represent the parameters of the neural network that need to be learned during training. The number of parameters in a neural network is generally large. Let's say, for a fully connected neural network that has 5 neurons in the input layer, one hidden layer with 10 neurons and an output layer with 3 neurons, the number of parameters to be learned is 93. In practice, neural networks have thousands and millions of parameters, even billions! That is why a large amount of data is needed to train them.

Is the number of parameters in a fully connected neural network that has 5 neurons in the input layer, one hidden layer with 10 neurons and an output layer with 3 neurons 93?

The input layer does not contain unknown parameters - it only leaks data into the network. Each neuron of the hidden layer is connected to each neuron of the input layer, which means that each of these connections has 5 parameters and one free member. This is a total of 10x5 + 10x1 = 60 parameters. Each neuron of the output layer is connected to each neuron of the hidden layer, which means that each of these connections has 10 parameters and one free member. This is a total of 3x10 + 3x1 = 33 parameters. When we add up the two values, there are 93 parameters.

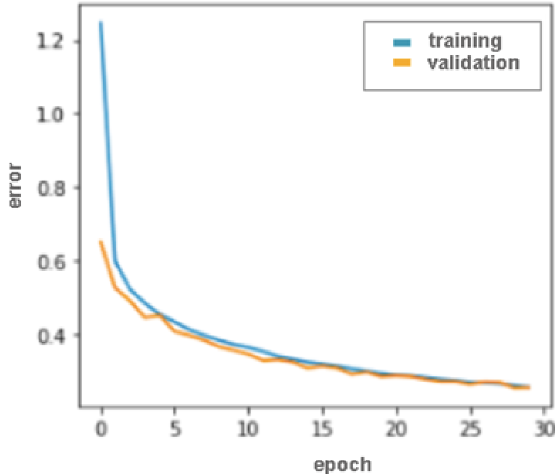
Neural networks, like other models, are trained on a training set and evaluated on a test set. Because neural networks are complex models that can learn complex relationships between attributes and outputs, they can easily be readapted to data. That is why we always use a validation set during the training of the network. It helps us to follow the course of training more finely and notice overadjustments and other undesirable properties of the model earlier.

In the introduction of the course, we said that the unknown parameters of the model are determined by defining the error function, and then applying some optimization techniques (which include gradient descent) with the aim of finding those parameter values for which the error function is the smallest. This protocol follows the story of neural networks, but the error values are not calculated for individual instances, but for groups of instances. The motivation for this design is, First of all, working with a large amount of data and the need to parallelize and speed up the whole process. That is why all the data in the training set is first divided into **packets** (*batch*) of equal sizes. The packets are then passed through the network, one by one, and the value of the error function is calculated for them by comparing the expected and obtained values of the target variable. Then, in proportion to their contributions, the error values of the neural network are updated by going backwards through the network. The described process of updating network parameters is called **backpropagation** and allows us to refine the parameter values in iterations and reach the optimal parameter values. Otherwise, we're going to start at the beginning.



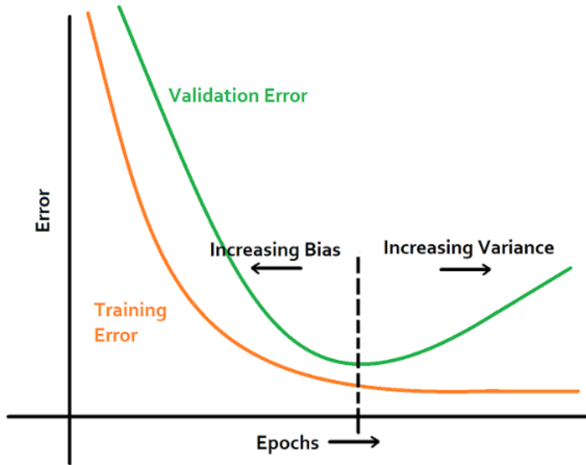
One passage through the entire data set, i.e. one processing of all packets of the training set, is called an **epoch.** Neural networks are trained in several epochs. After one epoch is completed, the data is "shuffled", then divided into packets again and passed through the network. In which epoch the model will be trained, depends on the success of the training and the available resources. Due to working with large amounts of data, Networks need specialized hardware that can parallelize computations (for example, graphics cards or tensor cards), so training networks is often both expensive and time-consuming.

This way of training the network through epochs allows us to follow the course of training more finely. At the end of each epoch, the model error on the training set and the model error on the validation set are calculated. These two values are then displayed on a graph that shows the epoch ordinal number along the x-axis and the error value along the y-axis. You can see one such graphic in the picture below. Good training is characterized by a comparative decrease in these values to a satisfactory error value - the closer we are to zero, the better the model. Recall that this conclusion is based on the fact that the validation set contains data that is separate from the training set and that the network sees for the first time.



If we notice that the values of the error function on the training set are decreasing and on the validation set are increasing, we conclude that the model is refitting and we stop training. Next, we have two options. If the values of the model error function in the epoch before the observed model refitting were satisfactory, we can keep that version of the model for further testing on the test set (usually, during the training of the network, several versions of the model are saved with the idea of using them for this purpose, or to be used if it is necessary to stop and continue the training process). Otherwise, we have to try a slightly different network architecture or a slightly different set of its hyperparameters. Given that each layer of the network has its own

settings (number of neurons, activation function, initial set of parameters), that the layers can be connected in different ways, that we have to simultaneously monitor all the settings of the optimization algorithm, for example the gradient descent and its learning step, and that some expectations in terms of quality measures need to be met, training the network is a challenging and complex task. That is why it is said to represent *the art of training*.



Monitoring of neural network readaptation based on error function value graphs on the training set and validation set

4.3 Convolutional Neural Networks (CNN)

Learning to Represent Data

Neural networks can help us isolate some abstract attributes in data and learn representations that are suitable for solving problems.

In the examples we have used so far, we have most often relied on the existence of some set of attributes in the data set. Indeed, a large number of domains generate data that is of this form, with attributes in columns and instances in individual rows. As we saw in the introductory part of the data preparation story, even when we have these attributes, it's not the most intuitive to decide which attributes to choose to create a model. This put us in a position to try different combinations or devise techniques that can help us in the selection of attributes. Due to the complexity of the functions they model, neural networks boast the ability to learn to filter and group the attributes that matter.

This property of neural networks is especially important when working with non-tabular data - we have raised the question many times how to represent, for example, images, text data or audio recordings. Although we have knowledge of these formats, it is difficult for us to describe exactly what they contain in a concise and usable way. This, among other things, has motivated us to apply the data-driven programming paradigm. Neural networks can (and we will soon see) learn some abstract attributes from data in their source form that are useful for successfully solving problems.

Below, we will introduce convolutional neural networks that are used primarily in working with images and video and to learn visual attributes of input, and then recurrent neural networks and transformers, types of neural networks that are used to learn the attributes of sequential data such as text or sound.

Convolutional Neural Networks

Convolutional neural networks are a type of neural network that is primarily used in the field of computer vision to work with images and video content.

Black and white images are represented by pixel matrices. The number of types of this matrix corresponds to the height of the image, while the number of columns of this matrix corresponds to its width. The individual pixel values are numbers ranging from 0 to 255, where 0 is black and 255 is white. All the values in between represent some shade of grey. Can you guess what's behind the image that is represented by the pixel matrix below? If you step back far enough and draw contours along the darker shades, you might be able to guess what's in the picture. It helps that in the machine learning community, images of cats are a fairly common choice.

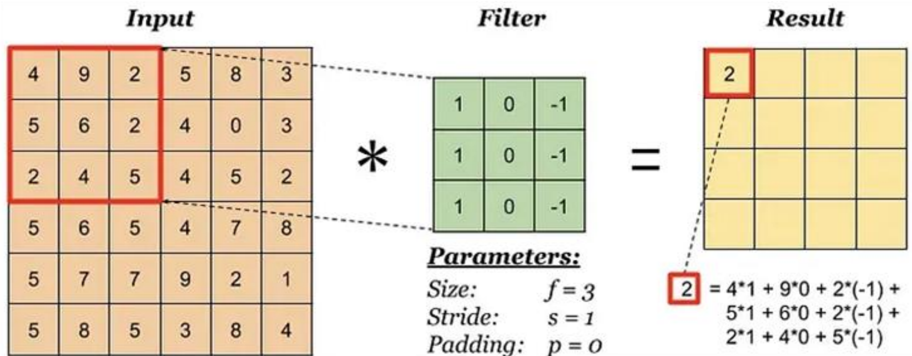
```
[[  9   1  29  70 114  76   0   8   4   5   5   0 111 162   9   8  62  62]
 [  3   0  33  61 102 106  34   0   0   0   0  49 182 150   1  12  65  62]
 [  1   0  40  54 123  90  72  77  52  51  49 121 205  98   0  15  67  59]
 [  3   1  41  57  74  54  96 181 220 170  90 149 208  56   0  16  69  59]
 [  6   1  32  36  47  81  85  90 176 206 140 171 186  22   3  15  72  63]
 [  4   1  31  39  66  71  71  97 147 214 203 190 198  22   6  17  73  65]
 [  2   3  15  30  52  57  68 123 161 197 207 200 179   8   8  18  73  66]
 [  2   2  17  37  34  40  78 103 148 187 205 225 165   1   8  19  76  68]
 [  2   3  20  44  37  34  35  26  78 156 214 145 200  38   2  21  78  69]
 [  2   2  20  34  21  43  70  21  43 139 205  93 211  70   0  23  78  72]
 [  3   4  16  24  14  21 102 175 120 130 226 212 236  75   0  25  78  72]
 [  6   5  13  21  28  28  97 216 184  90 196 255 255  84   4  24  79  74]
 [  6   5  15  25  30  39  63 105 140  66 113 252 251  74   4  28  79  75]
 [  5   5  16  32  38  57  69  85  93 120 128 251 255 154  19  26  80  76]
 [  6   5  20  42  55  62  66  76  86 104 148 242 254 241  83  26  80  77]
 [  2   3  20  38  55  64  69  80  78 109 195 247 252 255 172  40  78  77]
 [ 10   8  23  34  44  64  88 104 119 173 234 247 253 254 227  66  74  74]
 [ 32   6  24  37  45  63  85 114 154 196 226 245 251 252 250 112  66  71]]
```

You can also check the next block and see what the picture is.

Just as it is difficult for us humans to understand what is in an image represented by a group of numbers, so it is with convolutional neural networks. They begin their analysis of the image by first recognizing some simple elements such as horizontal and vertical lines, and then combining them further and adding complexity until they arrive at some complex descriptions of the image that can help them solve the task at hand. Now the natural question is how start from simple contours and build on them to obtain more complex structures. The answer will be given by **the convolutional** operator.

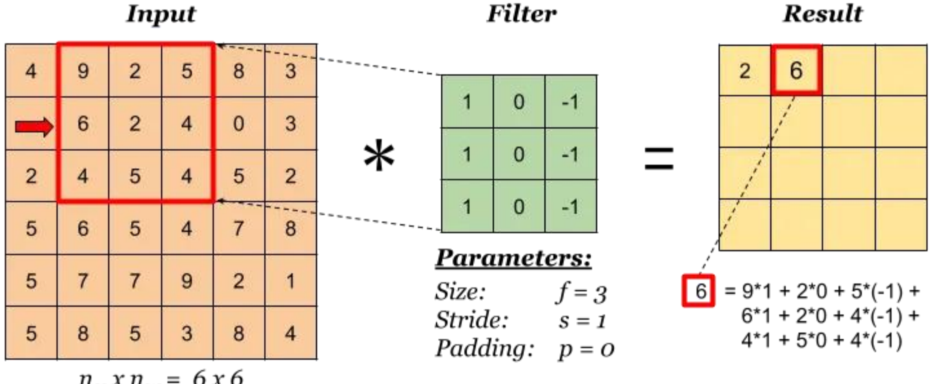
The easiest way to introduce the convolution operator is through illustrations. Let's imagine that we have a matrix of 6x6 pixels at the input that represents the image, and a small matrix, the so-called filter, with dimensions of 3x3 pixels. Let these be the matrices shown in the image below. We start applying the convolution operator over the image (we will mark it with ✷) by overlapping the part of the image in the upper

left corner with the filter, then multiply the individual values and write the sum of the values into a new matrix. This matrix will be the result of the application of the convolutional operator.



Convolution - Step 1

We will continue to apply the convolution operator: we will overlap the filter with the part of the image located in the upper left corner, but so that it is now shifted one pixel from the left edge, i.e. compared to the previous position. Again, we will multiply the individual values, add them together, and write them into the resulting matrix.



Convolution - Step 2

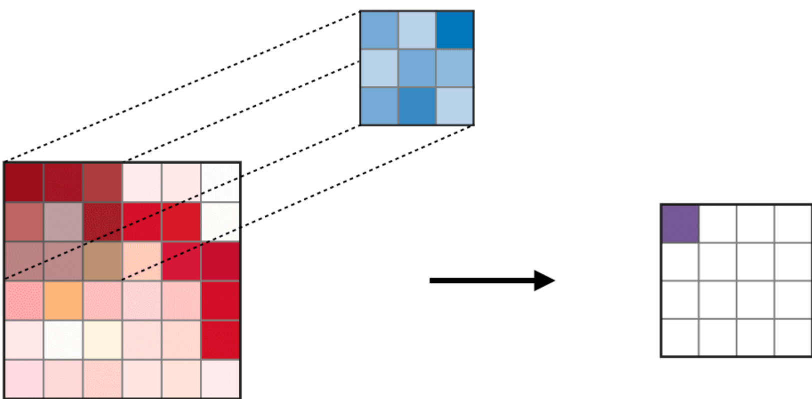
The filter can be moved one position to the right all the way to the edge. Then we need to lower it one position down and put it back right next to the edge. Then we can continue the process until we reach the lower right corner. As a result of this operation, we will get a matrix of dimensions of 4x4 pixels, the values of which are shown in the image below.

2	6	-6	5
0	4	0	-1
-5	0	3	6
-2	5	0	3

Convolution - the result

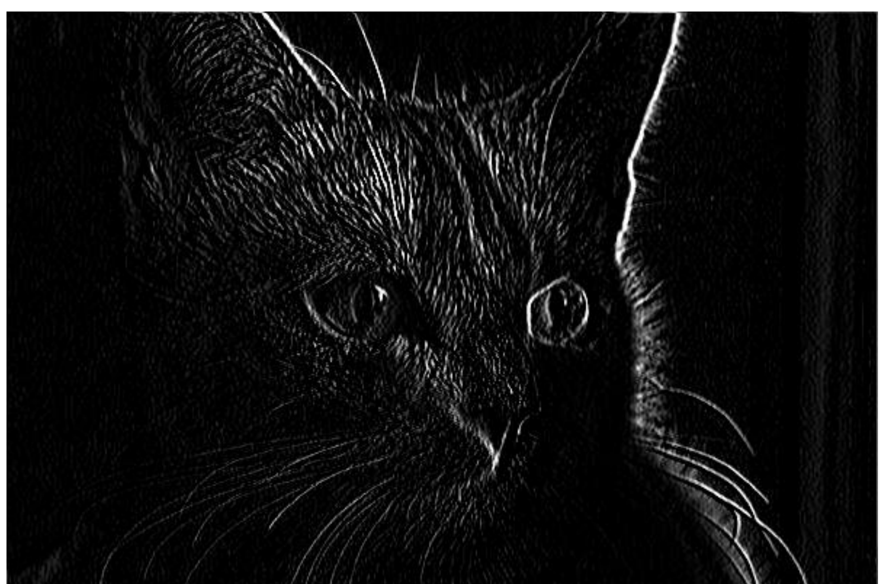
How much the filter will be moved in each iteration is defined by a hyperparameter called a scroll. *stride*). In our case, the offset had a value of 1 because we moved the filter one position to the right, that is, when it should have been one position down. In order to be able to influence the dimensions of the resulting matrix when applying the convolutional operation (usually we want to preserve the dimensions that correspond to the input matrix), we can add a frame around the starting image. It is usually a block of zeros or ones or numbers whose values correspond to the values of the nearest pixel in the image. It's called an extension. *padding*) and its width are always emphasized when applying the convolutional operation. It is especially important to us if the characteristics we want our model to learn are close to the edge of the image.

The animation below illustrates the entire process of applying filters to the image, i.e. On top of her matrix. As you can see, a size 1 offset and a size 0 extension were used.



Animation of the convolutional operation

If we apply the filter from the previous example using the convolution operation on the initial image of the kitty, we get the image below. You can see that all the vertical lines that appear in the picture are highlighted.



Vertical Edge Extraction

Are you surprised that the top filter detected vertical edges?

Here's an explanation. Look at the picture from left to right. The first time you move from the light part of the picture to the dark part of the picture, you actually see a vertical edge. Let's now apply a filter from left to right with a convolutional operation. The highest result of an iteration of the convolution will be when the right side of our filter (the column numbered -1) is positioned exactly on the vertical edge. Since the edge is dark, the values that correspond to that color are small because the black color is represented by zero. The values to the left of the edge are light, so the numbers that correspond to those colors are larger (the value for white is 255). When the small values corresponding to the black color of the edges are multiplied by -1, i.e. with the right part of the filter, and the large values, which correspond to the light colors to the left of the edge, are multiplied by 0 and 1, i.e. The result is a higher value than if the filter were found anywhere else where there is no filter.

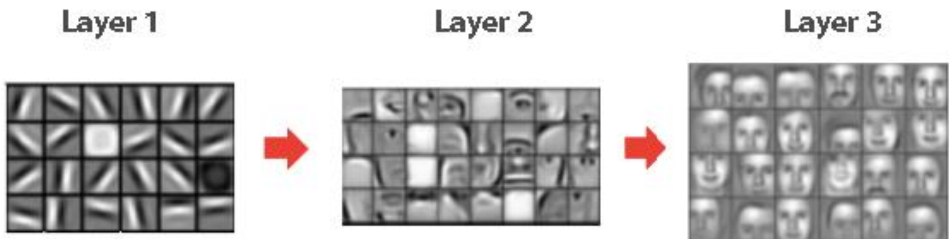
How would you separate the horizontal edges in the picture? Which filter would you use?

All you have to do is rotate the filter that separates the vertical edges! Does that make sense to you?

Now that we know how to separate vertical and horizontal edges, we can combine in a variety of ways to single out lines that are not just horizontal and vertical. By further combining these results, we can even extract spherical contours. This is what we meant when we said that we start with clear, easy-to-learn characteristics, and then build step by step on the complexity of the characteristics we can learn.

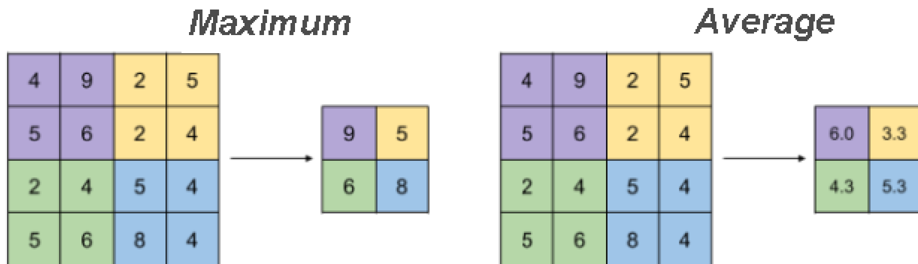
The layers of a neural network characterized by the application of the convolutional operator are called **convolutional layers**. While the pioneers in the field of computer vision created filters manually, the goal of training convolutional neural networks is to learn for themselves the values that figure in them.

In the image below, you can see the representations of the learned filters on the layers of a convolutional neural network that recognizes faces. On the lowest layer, these are some horizontal, vertical and diagonal lines, on the second layer these are already outlines that correspond to parts of the face such as the nose, eyes and mouth, while on the third layer these are filters that correspond to the contours of the face.



Classification process of passing through successive layers of the image

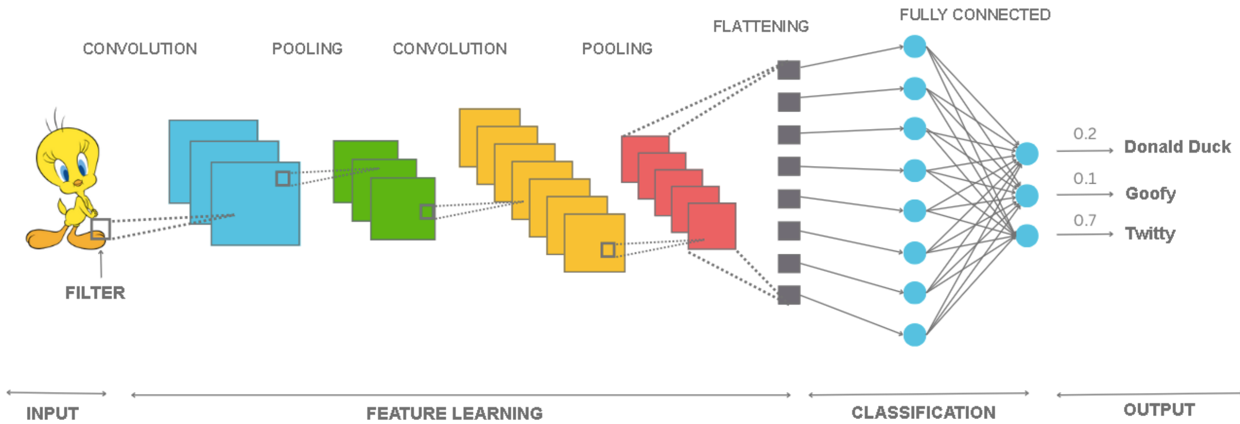
In addition to the convolutional operation, convolutional networks are also characterized by the ***pooling***. As the name suggests, the goal of this operation is to pool, i.e. consolidate the entrances. In the image below, you can see two types of pooling operators: the one that uses the maximum and the one that uses the average to pool the information. Just like the convolutional operator, this operator is applied to blocks of input by noticing the block and performing the necessary calculation on it. The value obtained in this way is entered into a new matrix. In the figure, both operators are applied to 2x2 blocks. Intuitively speaking, we emphasize the most dominant part with maxima, while by calculating the average, we take into account the contribution of all parts.



By applying the pooling operation, we get the ability to reduce the dimension of the input, but at the same time retain some of the information that is contained. In the image you can see that by applying the pooling operator, we have reduced the matrix from 4x4 to 2x2. Why do we need dimension reduction? As a result, the network needs to give us a specific answer, let's say whether there is a cat in the picture or not, for which we need a small number of neurons.

The layers of a neural network that are characterized by the application of the pooling operator are called *pooling layers*. There are no additional parameters in them that the network needs to learn, but, as we have seen, they help us control the dimensions of the matrices we work with.

Now that we know what the building blocks of a convolutional neural network are, let's see how we can connect them and get a functional model that can help us solve the classification task. Let's imagine that we need to solve a multi-class classification task in which for each image of a cartoon character we need to determine whether it is Tweety, Goofy or Dacia Duck. Let's look at an illustration of the architecture of a deep convolutional neural network that we have chosen to solve this task and discuss what is the motivation for its creation.



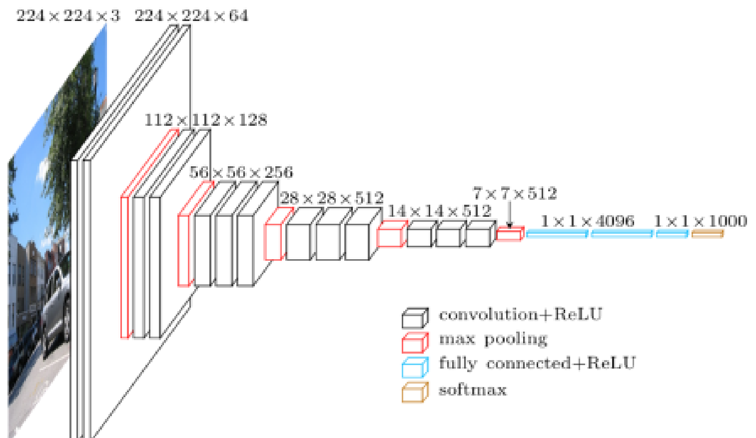
At the beginning of the network there is an input layer that contains pixels of the image. This is followed by a convolutional layer (a blue block). The goal of this layer is to use the convolution operator to extract the first set of abstract attributes. After that, we place a pooling layer (green block), followed by another convolutive layer (orange block) and a pooling layer (red block). In practice, the convolution and pooling layers are combined and often located next to each other because the pooling layers are further pooled, i.e. summarizes what the convolutional layers have learned. The second convolutional layer allows us to apply a second convolutional operator to attributes already extracted by the first convolutional layer and form more complex image attributes. This block of layers is followed by a layer (grey block) that has the task of "correcting" the matrix (or, more precisely, the tensor) that we have received up to that point and repackaging its values so that they are all next to each other in one array. Layers for this purpose are called straightening layers.

flattening layers). After correction, we can further build on a fully connected neural network. This network will now have abstract attributes as inputs that a combination of convolutional layers and pooling layers learns for it. In addition to the corrected input layer, we also see a hidden layer in the image as well as an output layer in which there are exactly three neurons - one for each of the cartoon characters. The output values of these neurons correspond to the probability that the input image belongs to the class they represent. For the image of Twitter that we have at the input, we can notice that the output value of the third neuron, which corresponds exactly to that class, is the largest and is 0.7.

Now we can also see what the architecture of *the VGGNet* network looks like, a popular convolutional network that is actively used in practice.

VGGNet is a deep convolutional neural network developed by the Oxford team Visual *Geometry Group* (hence the name *VGGNet*). At the prestigious Large *Scale Visual Recognition Challenge*, held in 2014, this network proved to be the best in solving the problem of localization of objects in an image and as the second in a row in the problem of classifying objects from an image. Over 1.2 million images were used for the classification task of the ImageNet set that we have come to know and the classification into a possible 1000 classes. To train this network it took between 15 and 20 days using 4 graphics cards (the best at the time) *NVIDIA Titan Black*. Before it, the best in these tasks was the *AlexNet network*, which is notable for introducing the practice of using graphics cards to train neural networks and allow the further development of deep learning.

The architecture of the *VGG-16* network, a version of the network with 16 layers, is shown in the image below. As we can see, a color image measuring 244x244 pixels is expected at the input, and the network combines convolutional layers and pooling layers (with maximum) and ends up with a fully connected neural network with 1000 neurons at the output, where each neuron corresponds to one particular class of *the ImageNet* set. The network itself has 138 million parameters and requires about 500 MB of memory to store them.



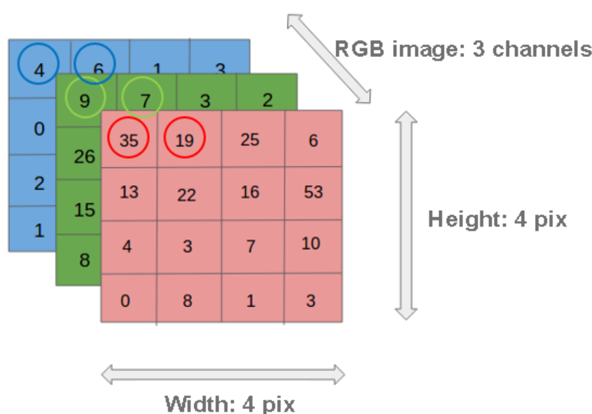
VGG-16 Network Display

Increasing the number of layers of the convolutional part of the network usually gives better results in practice. However, the number of layers cannot increase indefinitely. Not only because of the limitations of resources, time, and cost, but also because of the mathematical properties of deep neural networks, which further make it difficult to apply the backward propagation algorithm and train the network itself.

If you are interested in this mathematical problem, you can try to read more about disappearing and exploding gradients, especially in the part of convolutional networks and residual connections.

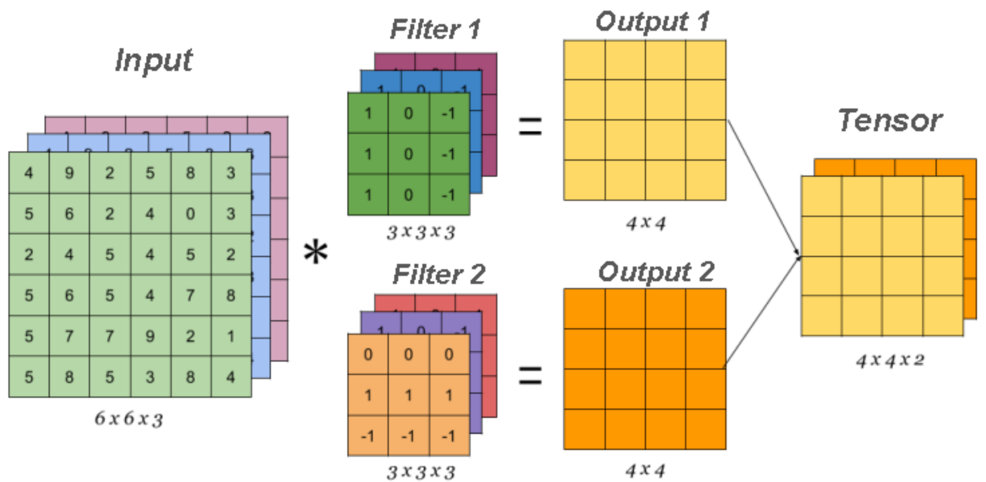
Before we get into the task of working with convolutional neural networks, let's take a look at the issue of working with color images. We haven't mentioned them so far.

When we need to present a color image, one that uses the RGB color format and displays all colors as combinations of red, green and blue, we use three matrices. One matrix is provided for each of the colors. The number of matrices we use to display images is called **channels**. Thus, black and white images have only one channel, while color images have three channels.



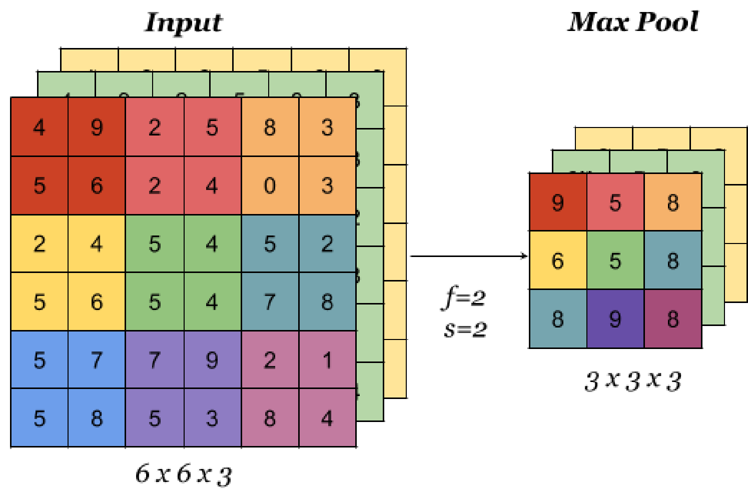
Display of images that use the RGB color format

The presence of colors affects the performance of the convolutional operation by adjusting the number of channels of the filter to the number of channels of the image to which we are applying it. Next, you need to pair each of the filter channels with the image channel (red with red, blue with blue, and green with green) and perform the convolutional operation as if you were working with a single channel. Then we need to add the matrices that we get in this way and declare the resulting matrix as the final result. In the image below, we can see two filters in a convolutional layer that are applied over the color input image. As a result of applying each of these filters, we will get separate matrices that, when "merged", represent the final result of the convolutional layer. In practice, multiple filters are usually placed at the level of one convolutional layer, so tensors are obtained as results. Convolutional operations are then applied to these tensors in the same way - only care is taken to ensure that the number of filter channels corresponds to the dimension of the tensor (let's say, for the next application in the example we considered, this would be the number 2) and that the corresponding input channel is paired with the corresponding filter channel.



Applying the Convolutional Operator to Multi-Channel Inputs

As for the pooling operation, it is applied over each channel of the input image. For example, if the input image has 3 channels, the pooling operation will be applied to each channel separately. This also means that the pooling operation preserves the number of channels at the time of deployment. In the picture below you can see an illustration of this process.



Applying pooling Operators to Multi-Channel Inputs

This section is paired with the Jupyter Notebook [11-VGG-16_network_and_classification.ipynb](). To follow the content further, click on the link and then click [Open in Colab] on the button to open the content in *Google Colab.* If you are viewing the notebooks on your local machine, find the notebook with the same name among the contents and run it. For more detailed instructions, see the *Hands-on Zone* section and the *Jupyter Notebooks lesson for practice*.

Now let's try out how the *VGG-16 convolutional network* really works! Don't forget to follow the notebook with the code while reading the lesson.

Once trained, a neural network model can be shared with the community by dividing the parameters that figure in it. In this example, we will use the model that is available in the *Keras* library. The *Keras* library is an open source library widely used in the machine learning community. In order to take advantage of the *VGG-16 network model,* we need to execute the following two commands:

```
from tensorflow.keras.applications import VGG16
model = VGG16(weights = 'imagenet')
```

We can read the information about the model we have loaded using the model.summary() function. Its result is a description of the network layers followed by information about the input sizes that these layers expect. Now you can execute the command:

```
model.summary()
```

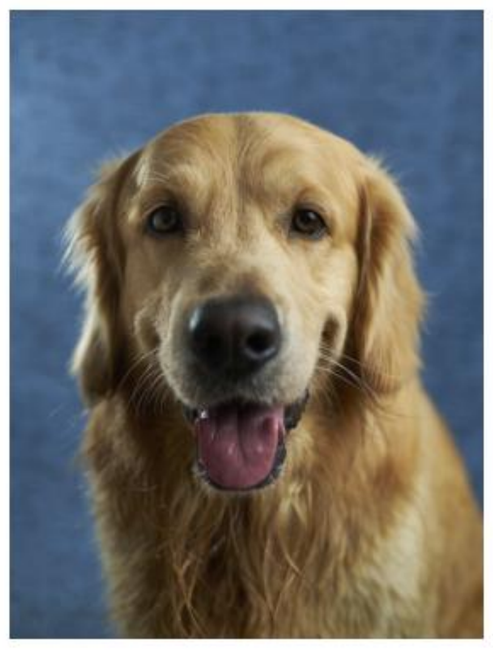
Don't be confused if you don't understand all the details that are displayed after executing this command. It's important to know that the input is expected to be an image with dimensions of 224x224 pixels in color (that's why 224, 224, 3 is listed next to the input layer) and that you have one of 1000 classes at the output. You can also compare the print with the image of the *VGG-16* we discussed to reveal more information.

It is important to emphasize that *we will not train the VGG-16* network - we will use only a trained model. Therefore, we must not change the parameters of the model during operation - each has its own contribution. The total number of model parameters that we can read in the network summary is just over 138 million.

The idea is that the image on which we are going to test the model will be an arbitrary image from the web. To do this, we will use a number of standard Python libraries. To default the URL function upload_image will help us drag the image we want.

```
def load_image(url_path):
        response = request.urlopen(url_path, context=ssl_context).read()
        return Image.open(BytesIO(response))
```

For testing, we have chosen a picture of a Golden Retriever from address https://unsplash.com/photos/x5oPmHmY3kQ, which can be used freely. You can choose the image you want! It is important to keep in mind that the class of the object in the picture must be known to the model. Since *the VGG-16 model has been* trained on over 1.2 million images, it knows a lot of classes, as many as 1000 different ones. The Golden Retriever is one of them. If we give the model a picture with an object that it does not know, It will give us predictions of the classes whose images most closely resemble ours. We'll see in the end which classes resemble the Golden Retriever.



The hero of the story of the VGG-16 model

Since the image that needs to be forwarded to the model needs to be specially prepared, we will do the following:

set its dimensions to 224x224 and tell it to use three RGB color channels:

```
test_image = test_image.resize((224, 224))
test_image = test_image.convert('RGB')
```

Transform the image into a suitable matrix format:

```
matrix_form_test_image = image.img_to_array(test_image)
```

make a package that contains our picture:

```
batch = np.expand_dims(matrix_form_test_image, axis=0)
```

Perform numerical image preprocessing in the form of normalization:

```
test_image_batch = preprocess_input(batch)
```

Only in this way can we pass on the model for classification. The function that will help us is (expectedly) called predict.

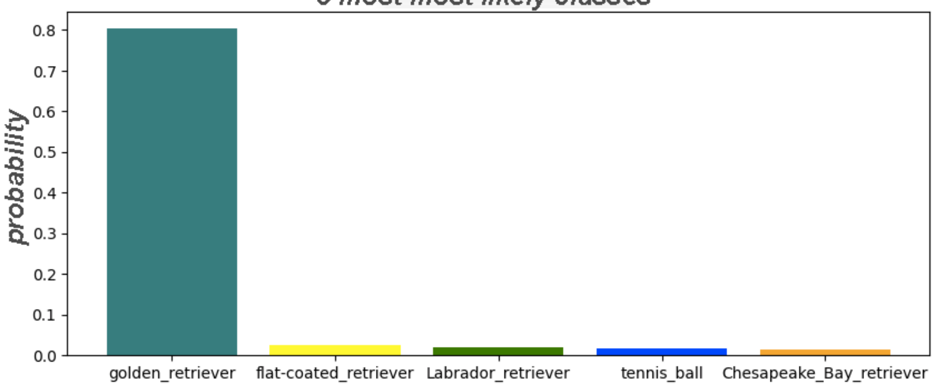
```
model_predictions = model.predict(test_image_batch)
```

The variable model_predictions in which we store the predictions of the model is an array of length 1000 and contains the probabilities of belonging to each of the 1000 classes that the model recognizes. To extract the class to which our image belongs, we can use the decode_predictions function, which will return the probabilities and names for the 5 most probable classes. This will give us an insight into how secure the model is when classifying. After executing the next command, we will get information about the most likely classes.

```
most_likely_classes = decode_predictions(model_predictions)[0]
```

When we graphically represent these predictions with the code listed below, we get a graph with bars that allows us to more easily analyze the results.

```
class_names = [item[1] for item in most_likely_classes]
class_probabilities = [item[2] for item in most_likely_classes]
plt.figure(figsize=(10, 4))
plt.bar(class_names, class_probabilities, color=['teal', 'yellow', 'green', 'blue', 
'orange'])
plt.title('Top 5 Most Likely Classes')
plt.ylabel('Probability')
plt.show()
```

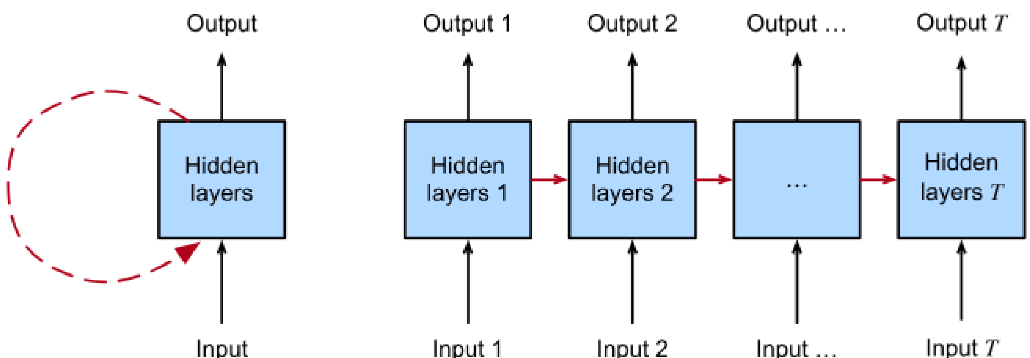


As we can see, the model predicted with great certainty (probability is 0.804) that the image we chose was that of a golden retriever. Some of the other classes that the model has taken into account are some other types of retrievers. Oddly enough, a tennis ball also appeared in the list of results. Probably because in the training set there are also images in which retrievers run after tennis balls. This behavior of the model should be further investigated in practice.

4.4 Recurrent neural networks

Recurrent Neural Networks (Neural Networks) are a type of neural networks that are primarily used to process sequential data. Sequential data or sequences are made up of elements that follow one another. Such are, for example, textual data (elements are individual words), audio recordings (elements are individual samples), time series (elements are individual measurements), genetic sequences (elements are individual

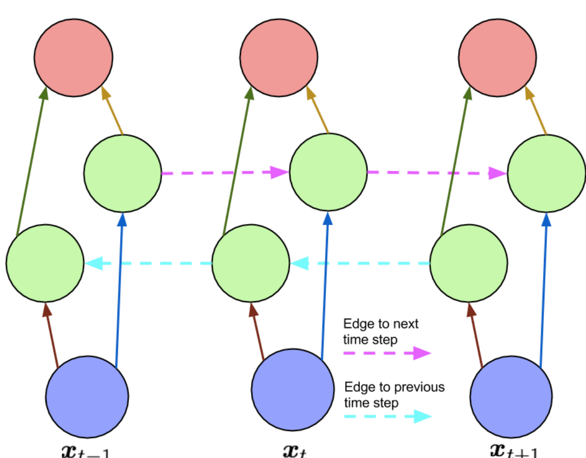
nucleotides), and many others. Recurrent networks process sequence element by element. In order to be able to process an element at position *t*, all the elements that precede it must be processed, and in order for the elements of the sequence to be connected into a single whole, the values of the hidden layers are divided between the processing of successive input elements. This is usually shown graphically as in the figure below.



Recurrent neural network

(image taken from https://d2l.ai/chapter_recurrent-neural-networks/index.html)

Recurrent neural networks are hypothetically capable of processing infinitely long sequences: element by element. However, when training such networks, it has been observed that they forget. If the sequences are too long, the network begins to forget what it saw at the beginning and stores the recently seen information at the level of hidden layers. This observation led to the design of special neurons called LSTM (LSTM - *Long Short Term Memory)* and GRU (*Gated Recurrent Unit)*, and we will not go into details due to its complexity. A solution to this problem is two-way recurrent neural networks (*Bidirectional Recurrent Neural Networks*). In these networks, on the one hand, the sequence is processed from beginning to end, and on the other hand, from end to beginning. The input representation of the individual elements represents the contiguous representations of these passages, illustrated as in the figure below.



Two-way recurrent neural network - sequential elements

(image taken from https://www.arxiv-vanity.com/papers/1506.00019/)

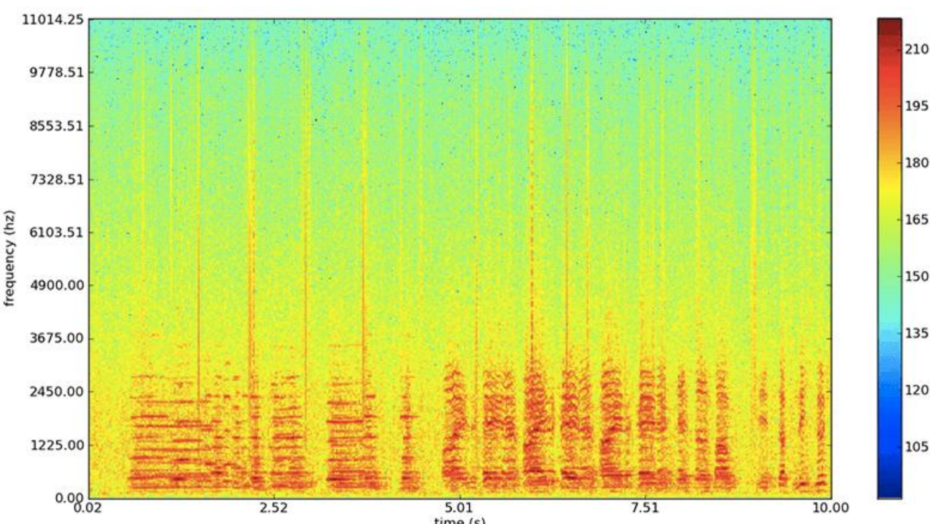
There are several popular architectures of recurrent neural networks. In the table below, we'll briefly go through some of the most popular examples by showing them graphically in the left column and describing the network and application areas in the right column.

ARCHITECTURE	CLARIFICATION AND EXAMPLES OF APPLICATION
	This type of network corresponds to tasks in which the input is a sequence and the output is a vector representation of a fixed length. Networks of this type are called encoders) and the obtained vectors of fixed lengths by context. The tasks in which we encounter this type of networks are various classification tasks such as the classification of audio tracks or the classification of text.
	Unlike the previous example, the input for this type of network is a vector representation of a fixed length and the output is a sequence. This type of network is called **decoders**). The tasks in which we encounter decoders are the generation of image titles.
	This type of network is a combination of the previous two types and is called **encoder-decoder** architecture. The task of the encoder is to create a representation (context) based on the input sequence that the decoder can use to generate a new output sequence. This type of network is encountered in machine translation or abstract generation tasks.
	This type of network allows the generation of outputs for each element of the input. As you can see, there are sequences at both the entrance and the exit. The tasks in which we encounter this type of mesh are, for example, the tasks of tagging (marking) individual elements.

One major drawback of recurrent neural networks is the inability to parallelize: in order to process an element at position t, all the elements that precede it must be processed. That is why training neural networks requires much more time and resources than training the convolutional neural networks that we got to know in the previous lesson. These circumstances have led to the emergence of the attention mechanism and transformers, Neural networks that will be discussed in more detail in the next lesson.

Audio recordings can also be processed using convolutional neural networks. Namely, an audio recording can be divided into fragments, shorter pieces that last a few seconds, and then spectrograms can be created for

each part. A spectrogram is a graphical representation of all the frequencies of sound present in an audio recording. The resulting images can then be passed as inputs to convolutional neural networks and used for audio analysis

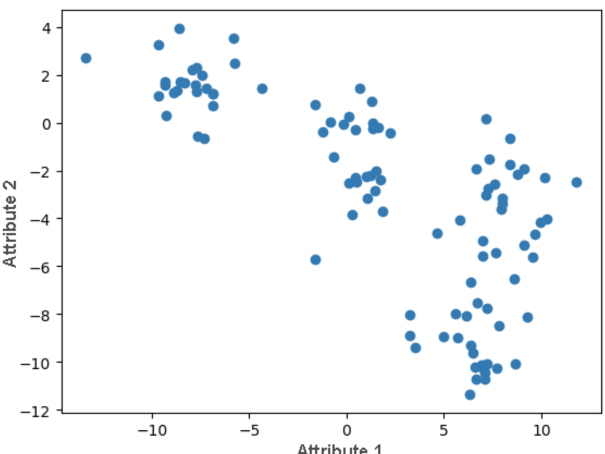


An example of a spectrogram

4.5 K-Means algorithm

The K-mean is an unusual name for an algorithm. Read on to find out what's behind this choice.

To make it easier to follow the story of the x-mean algorithm, we will use the dataset shown in the figure below. It consists of 100 pairs of points, imagine that these are the values of some two numerical attributes.



This section is paired with the Jupyter Notebook 12-k-means.ipynb. To follow the content further, click on the link and then click Open in Colab on the button to open the content in the *Google Colab environment*. If you are viewing the notebooks on your local machine, find the notebook with the same name among the contents and run it. For more detailed instructions, see the *Hands-on Zone* section and the *Jupyter Exercise Notebook* lesson.

In the accompanying material, you can generate all the images and animations yourself.

The **k-mean** algorithm should find *k* clusters in the dataset. The clusters that this algorithm looks for are determined **by the centroid**, an instance that represents the center of the cluster.

The initial step of the algorithm is the initialization step. In it, we need to select randomly *x* centroids. Then we need to repeat the following steps:

For each instance, we need to calculate the Euclidean distance to each of *the x* centroids, and then associate the instance with the cluster whose centroid it is closest to. Once we have arranged all the instances, we move on to the next step.

For each of the *x* clusters, we need to select new centroids. We do this by calculating the average of the instances that are in each of the clusters and declaring this value as the new centroid. After that, we go back to step 1 again.

The clustering algorithm ends when the values of the cluster centroids stabilize. This would mean that in two successive iterations we get centroids that differ very little, less than some predetermined accuracy.

The k-mean algorithm itself is not unpleasant to program, so we will do it together. Before that, let's consider some technical details:

One instance of a data set is a pair of numbers, say (2, -3). This means that the centroid will also have a pair of numbers and will have two coordinates;

If (10, 2) and (4, -4) are two instances of a data set, we will calculate the instance representing their average as (10 + 42, 2 − 42) = (7, -1);

If (0, 0) and (3, 4) are two instances of a data set, we will calculate the Euclidean distance between them as (3 − 0)2 + (4 − 0)2.

In the dataset, we will look for four clusters. We will consider why we chose this issue a little later. Now let's get ready to program the algorithm.

The variable k denotes the number of clusters. K = 4.

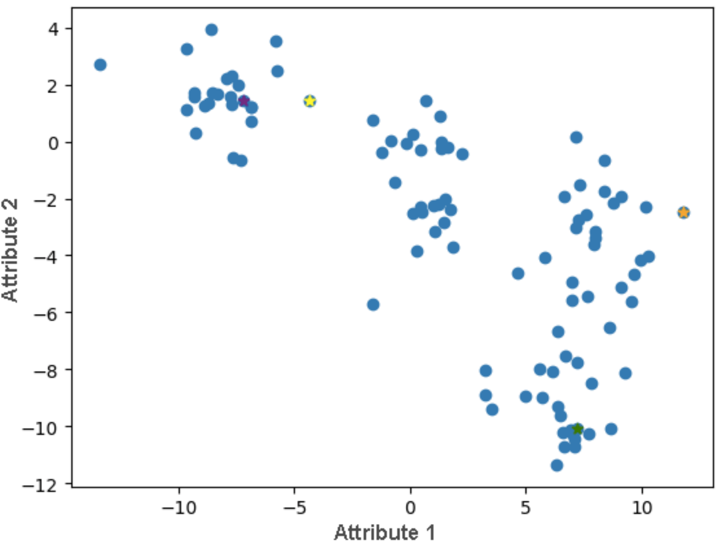
We will denote the centroids of the cluster with the centroid variable. Since we have *k* of the cluster, this will be an array of length *k*. One centroid is, as we said, one pair of numbers, so the elements of this array will be pairs of numbers.

In the process of clustering, we need to keep track of which cluster we associate with which instance. That's why we'll use labels to tag clusters, similar to the classification tasks. These can be values 0, 1, 2 and 3. In general, some values 0, 1, 2,..., k-1. We're going to keep all the labels in a series of cluster labels.

Now let's introduce the function generate_centroids(X, k) that generates the initial centroids. Its arguments are the set of instances X and the number of clusters k, and the function itself randomly selects k numbers from the interval from 0 to 100 and returns the instances that are in those positions.

```
def generate_centroids(X, k):
    N = X.shape[0]
    indices = np.random.randint(low=0, high=N, size=k)
    return X[indices]
```

In the figure below, the generated centroids are shown. Each of them is in the color of the cluster it represents.

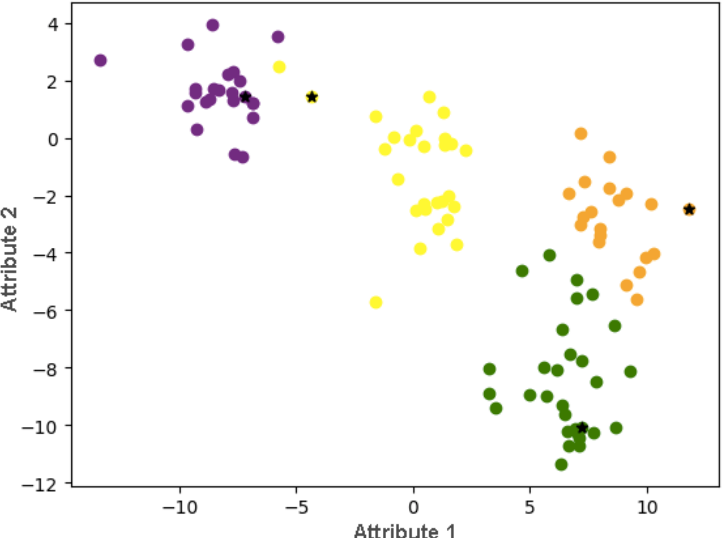


Initial values of centroids

Now let's write a function divide_data(X, centroids, k) to divide a set of instances into clusters. This function has as arguments the set of instances X, the current centroid centroids, and the number of clusters k. For each instance, we calculate the value of the distance to each centroid, then select the centroid that is closest and conclude that the instance belongs to the cluster it specifies.

```
def divide_data(X, centroids, k):
  # initialize the list of cluster labels
  cluster_labels = []
  # iterate through the dataset instance by instance
  for x in X:
    # initialize the list of distances to centroids
    distances_to_centroids = []
    # then for each centroid ...
    for centroid in centroids:
      # ... calculate the distance between the instance and the centroid
      d = calculate_distance(x, centroid)
      #... and add it to the list of distances
      distances_to_centroids.append(d)
    # when we have visited all centroids,
    # choose the centroid closest to the instance x
    label = np.argmin(distances_to_centroids)
    # conclude that the instance belongs to the cluster
    # determined by that centroid
    cluster_labels.append(label)
  # the result of the function is an array of cluster labels
  return np.array(cluster_labels)
```

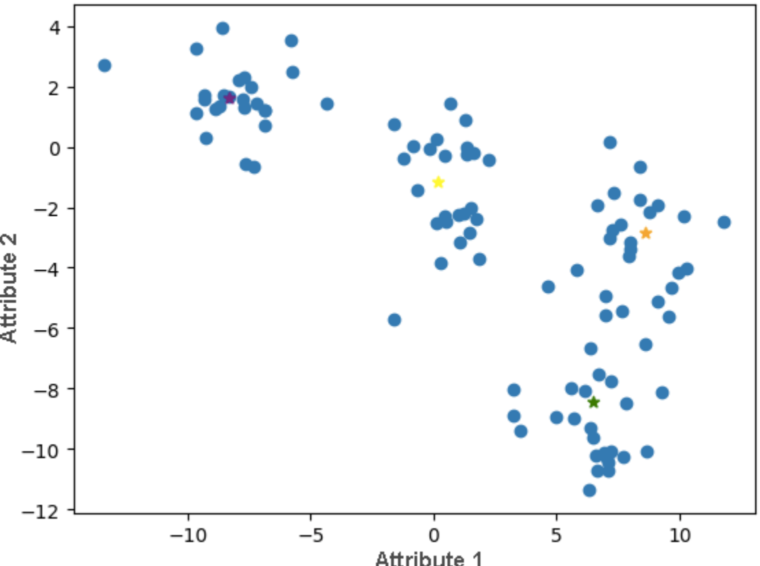
In the image below, you can see the first iteration of the division of instances into clusters.



Now let's write a function calculate_new_centroids(X, cluster_labels, k) that can calculate the values of new centroids based on the current division of instances into clusters. Its arguments are the set of instances X, the current instances of label_cluster, and the number of clusters k. For each of the clusters, this function should extract the instances that belong to it and then calculate their average.

```
def calculate_new_centroids(X, cluster_labels, k):
  # initialize the list of new centroids
  new_centroids = []
  # for each cluster
  for i in range(0, k):
    #... extract the instances that belong to it
    instance_indices = cluster_labels == i
    instances_in_cluster = X[instance_indices]
    # then calculate the new centroid value
    # by averaging all instances in the cluster
    new_centroid = np.average(instances_in_cluster, axis=0)
    # add the calculated new centroid to the list of all centroids
    new_centroids.append(new_centroid)
  # the result of the function is an array of new centroids
  return np.array(new_centroids)
```

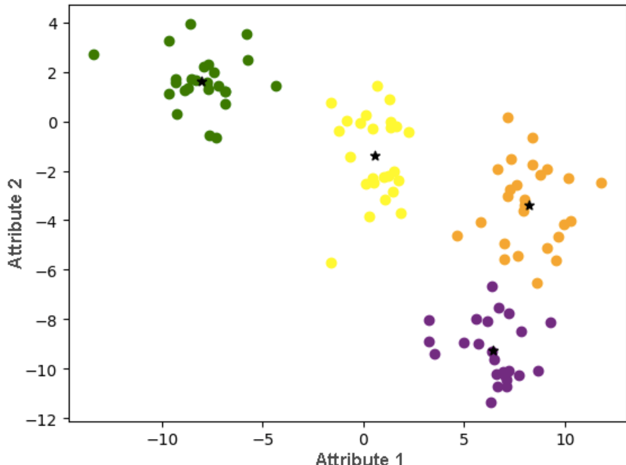
The new centroids are now shown in the image below. You will notice that the centroids of the yellow and purple clusters have "separated".



It remains to consolidate the tasks of individual steps into a function that will repeat them a sufficient number of times. It will be a function execute_clustering (X, k, epsilon = 1e-4, number_of_ iterations = 300) in which X represents the set of instances, k the number of clusters, the epsilon closeness that needs to be satisfied by the centroids of the cluster in order for the algorithm to stop. There is also a maximum number of iterations max_iterations which we additionally provide a stop criterion.

```
def execute_clustering(X, k, epsilon=1e-4, max_iterations=300):
  # step of initializing centroids
  centroids = generate_centroids(X, k)
  # in each iteration of the loop
  for i in range(0, max_iterations):
    # step 1: dividing instances into clusters
    cluster_labels = divide_data(X, centroids, k)
    # step 2: calculating new centroids
    new_centroids = calculate_new_centroids(X, cluster_labels, k)
    # checking stopping criteria
    # if they are met, we stop the algorithm
    if np.linalg.norm(new_centroids - centroids) < epsilon:
      break
    # otherwise, we move to the next iteration
    centroids = new_centroids.copy()
  # the result of the function is the final cluster labels and centroid values
  return cluster_labels, new_centroids
```

Executing this function also brings us to the final division of the set of instances into clusters, which is shown in the figure below.



In the accompanying code book, you can also see an animation that follows this division. Some steps rely on random decisions (for example, if an instance is equally close to multiple centroids), so don't be confused if some values differ slightly.

Reference

Curriculum: Fundamentals of Artificial Intelligence and Machine Learning

https://petlja.org/sr-Latn-RS/kurs/11203/0

https://scikit-learn.org/

„Machine Learning and Artificial Intelligence", Ameet V Joshi

„Deep Learning", Ian Goodfellow Yoshua Bengio Aaron Courville

„Machine Learning For Absolute Beginners", Oliver Theobald

„Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow", Aurélien Géron

"Artificial Intelligence A Modern Approach", Stuart J. Russell and Peter Norvig

„Explorations in Artificial Intelligence and Machine Learning", A CRCPress FreeBook

"The Hundred-Page Machine Learning Book", Andriy Burkov

"Machine Learning in Action", Peter Harrington

"Machine Learning A Probabilistic Perspective", Kevin P. Murphy

"Introduction to Machine Learning with Python", Andreas C. Müller and Sarah Guido

"Machine Learning Yearning", Andrew Ng

"Deep Learning with TensorFlow 2 and Keras", Antonio Gulli, Amita Kapoor, Sujit Pal

"NeuralNetworksandDeepLearning", CharuC.Aggarwal