FITPED | AI

Natural Language Processing

Jozef Kapusta
Ján Skalka
David Držík
Vladimiras Dolgopolovas
Kirsten Šteflovič
Dominik Halvoník

www.fitped.eu

2024

Funded by the European Union

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

FITPED | AI

Natural Language Processing

Published on

November 2024

Authors

Jozef Kapusta | Teacher.sk, Slovakia

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

David Držík | Constantine the Philosopher University in Nitra, Slovakia

Vladimiras Dolgopolovas | Vilnius University, Lithuania

Kirsten Šteflovič | Constantine the Philosopher University in Nitra, Slovakia

Dominik Halvoník | Constantine the Philosopher University in Nitra, Slovakia

Reviewers

Piet Kommers | Helix5, Netherland

Vaida Masiulionytė-Dagienė | Vilnius University, Lithuania

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

Ivo Písařovic | Mendel University in Brno, Czech Republic

Erasmus+ FITPED-AI

Future IT Professionals Education in Artificial Intelligence

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

Funded by
the European Union

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

ISBN 978-80-558-2230-3

TABLE OF CONTENTS

1 Introduction 6
1.1 Natural language processing 7
1.2 Applications of NLP 9
1.3 Short history 14
2 First Steps 20
2.1 NLP components 21
2.2 NLTK library 25
3 Natural Language Understanding 27
3.1 Speech recognition 28
3.2 Syntax analysis 32
3.3 Semantic analysis 35
3.4 Pragmatic analysis 38
4 Knowledge Acquisition 45
4.1 Knowledge acquisition 46
4.2 Data sources 50
4.3 Data extraction 56
4.4 Data (knowledge) structure 64
5 Knowledge Inferencing 70
5.1 Inference 71
5.2 Applications of inference 78
6 Natural Language Generation 83
6.1 NGL applications I. 84
6.2 NLG applications II. 87
6.3 NLG phases 89
6.4 NLG projects 92
7 Data Sources 98
7.1 Structured and unstructured data 99
7.2 XPath 100
8 Preprocessing 107
8.1 Preprocessing methods 108
8.2 Manual examples 111
8.3 Basic operations 113
8.4 Feature extraction 120
8.5 Project 129

9 WordNet 136
9.1 WordNet 137
9.2 Practical use 142
Project: Synonym recommendation system 146
10 Document Models 148
10.1 Introduction 149
10.2 Boolean document model 152
10.3 Vector model of the document 156
10.4 Bag of words 162
10.5 TF-IDF model 170
11 Document Similarity 176
11.1 Introduction 177
11.2 Application 180
12 Sentiment Analysis 189
12.1 Introduction 190
12.2 Types of sentiment analysis 194
12.3 Sentiment analysis approaches 197
12.4 Practical examples 200
12.5 Available libraries 229
13 Spam Identification 234
13.1 Spam 235
13.2 Spam project 238
14 Large Language Models 252
14.1 Large language models 253
14.2 LLM in practice 256
15 Bibliography and Sources 260
15.1 Bibliography and sources 261

Introduction

Chapter 1

1.1 Natural language processing

📖 1.1.1

Natural language processing (NLP) is a rapidly growing field of artificial intelligence that has transformed the way we communicate with machines. It powers applications such as text generators that produce essays, chatbots capable of human-like conversation, and text-to-image programs that create realistic visuals from descriptions. In addition to these fascinating applications, NLP has expanded to include the analysis of complex systems such as programming languages and biological sequences such as DNA and proteins that exhibit linguistic patterns.

At its core, NLP focuses on enabling computers to process and understand human language, both written and spoken. This ability is essential for creating tools that simplify communication between humans and machines. NLP's revolutionary advances are driven by sophisticated AI models that can decipher the complexity of input text and produce coherent and meaningful outputs. These models are designed to learn from large data sets, allowing them to understand nuance and generate contextually relevant responses.

The impact of NLP is not limited to improving user experiences; it extends to solving real-world problems. Whether through the creation of interactive digital assistants or the analysis of scientific data, NLP is reshaping industries and expanding the potential of AI.

📝 1.1.2

Which of the following is an example of NLP in action?

- Generating text-based essays
- Recognizing faces in images
- Calculating mathematical formulas
- Predicting weather patterns

📖 1.1.3

NLP is a specialized field of computer science focused on enabling machines to understand and manipulate human language. Unlike general computational linguistics, which aims to study the theoretical principles of language, NLP is an applied discipline that develops tools and technologies for practical tasks. These tasks include language translation, document summarization, and even writing original content based on user input. NLP goes beyond simple text processing by capturing linguistic structures and meanings.

NLP includes two primary subfields: natural language understanding (NLU) and natural language generation (NLG). NLU is concerned with interpreting and analyzing the meaning of text, identifying sentiment, and extracting valuable

insights. NLG, on the other hand, focuses on generating coherent text output from given data, simulating human-like writing abilities.

It is important to note that NLP is different from – but complementary to – speech recognition. While NLP processes text and its own meaning, speech recognition converts spoken words into text, making the data available for further NLP-based processing. This combination supports technologies such as voice-activated assistants, automated transcription tools and other.

1.1.4

Select features of Natural language processing:

- Translates languages
- Summarizes text content
- Recognizes spoken words directly
- Edits video sequences

1.1.5

NLP has become an integral part of our daily lives and underpins many of the technologies we rely on. Virtual assistants like Alexa and Siri rely on NLP to process user queries and provide accurate answers. Similarly, advanced AI models like GPT-x excel at creating content on a variety of topics and powerful chatbots that engage users in meaningful conversations. Retailers and healthcare providers are using NLP to improve customer experiences, whether through chatbots or by analyzing medical records to extract useful information.

NLP plays a critical role in improving search engines. For example, Google uses NLP to analyze search queries and efficiently deliver relevant results. Social media platforms like Facebook also use NLP to detect and filter harmful content such as hate speech, promoting a safer online environment. The increasing sophistication of NLP is enabling its application in areas such as education, finance, and scientific research, solving unique challenges, and increasing productivity.

Despite these successes, NLP systems are far from perfect. Challenges such as bias, inconsistency, and unpredictable behavior persist. However, these limitations also open the door to innovation.

1.1.6

Natural language processing is used in _____ to provide relevant search results, in _____ to enhance customer experiences, and in _____ to analyze harmful content.

- search engines
- chatbots
- moderation systems

📖 1.1.7

While NLP has made remarkable progress, it also faces significant challenges that highlight its limitations. One major issue is bias in language models, which can lead to unethical or misleading outputs. This bias often stems from unbalanced training data that reflects societal prejudices. Another issue is the occasional incoherence in the generated text, where models fail to maintain logical flow or relevance in their responses.

However, these challenges serve as opportunities for innovation. Researchers are actively working to create more balanced datasets and improve model architectures to reduce bias. Techniques such as learning and fine-tuning are used to improve model performance, ensuring greater accuracy and context awareness. Addressing these limitations is crucial to achieve wider acceptance and ethical application across sectors.

The potential for NLP remains enormous. From supporting accessibility for differently-abled individuals to automating complex tasks in scientific research, NLP continues to redefine what is possible with AI. As the technology evolves, the focus shifts towards creating more inclusive, equitable, and robust NLP systems that meet the needs of a global audience.

1.1.8

What is one major challenge faced by NLP systems?

- Incoherent text generation
- Lack of computing power
- Limited storage capacity
- Poor programming interfaces

1.2 Applications of NLP

📖 1.2.1

Natural language processing is at the heart of a variety of modern technologies. Its applications range from answering questions to enabling fluid conversations with users. NLP achieves these feats by analyzing and generating human language or even translating one language into another. By using computational techniques, NLP transforms raw text into meaningful insights or actionable results in many fields.

Sentiment analysis

An important area is sentiment analysis, where the emotional tone of text is classified as positive, negative, or neutral. For example, online retailers use sentiment analysis to understand customer reviews, while mental health

researchers analyze social media comments to detect signs of psychological distress. This task involves using various methods, such as word patterns or advanced deep learning models, to evaluate emotional contexts.

Another key task is toxicity classification, which goes beyond general sentiment analysis and identifies harmful or offensive content. These models detect specific categories such as threats, obscenities, and hate speech. By using toxicity classification, platforms ensure a safer online environment by filtering out hostile language.

1.2.2

What is the primary goal of sentiment analysis in NLP?

- To classify the emotional tone of a text
- To translate text from one language to another
- To extract named entities from text
- To generate summaries of documents

1.2.3

Machine translation

Machine translation (MT) is a fundamental application of NLP that automates the process of translating text from one language to another. This technology bridges language barriers and enables communication between speakers of different languages. Popular tools like Google Translate have revolutionized global communication and enabled multilingual interactions for users around the world. Machine translation works by taking input text in a source language, processing it with advanced algorithms, and generating output text in the target language.

MT systems use methods ranging from rule-based approaches to modern deep learning models. Early systems relied on predefined grammatical and linguistic rules. However, with the advent of neural networks, statistical and neural machine translation (NMT) has become the standard. NMT uses sequential models to learn patterns and contexts from vast amounts of bilingual text data. This allows modern MT tools to handle nuances such as idioms, slang, and cultural expressions. Despite these advances, challenges such as preserving context and tone remain significant.

The impact of MT goes beyond everyday conversations. Social media platforms use MT to translate posts and comments for global audiences. Companies deploy it for business communications, enabling seamless collaboration across countries. It also plays a key role in education by providing access to learning materials in multiple languages. As MT continues to evolve, its integration into more aspects of everyday life demonstrates its potential to promote inclusivity and understanding.

1.2.4

Which of the following is a key advantage of modern Neural machine translation over traditional rule-based systems?

- Can understand and translate context better
- Requires no data for training
- Ignores idiomatic expressions
- Translates only technical texts

1.2.5

Spam detection

Spam detection is a critical NLP application designed to improve user experience by filtering out unwanted emails. This binary classification task evaluates whether an email is legitimate or spam by analyzing its content, subject, and sender information. Platforms like Gmail rely on spam detection to ensure that inbox stays clean and safe.

Spam detectors are trained on large email datasets, allowing them to recognize patterns associated with spam messages. Advanced systems use deep learning models to improve accuracy, reduce false positives, and reduce false negatives. In addition to email, spam detection is used in messaging apps and social media platforms to block spam ads and phishing attempts.

1.2.6

Which of the following are common categories for email classification in spam detection systems?

- Spam
- Non-spam
- Unread
- Trash

1.2.7

Grammar error correction

Grammar error correction (GEC) improves the clarity and accuracy of written communication by identifying and correcting grammatical errors. It is a sequential NLP task where ungrammatical input is transformed into grammatically accurate output. Tools like Grammarly and Microsoft Word integrate GEC models to help users improve their writing.

In addition to personal use, GEC is invaluable in academic and professional settings. For example, students can improve the quality of their essays using GEC feedback, while businesses use it to improve their official communications. Modern systems use advanced machine learning techniques to detect subtle errors such as incorrect subject-verb agreement or misplaced modifiers.

📝 1.2.8

What is the main goal of grammatical error correction?

- To enhance the grammatical quality of text
- To detect hate speech
- To improve essay grading automation
- To classify emails as spam or non-spam

🕮 1.2.9

Information retrieval

Information retrieval (IR) is a fundamental NLP task that identifies documents relevant to a user's query. Search engines like Google use IR to process billions of web pages and retrieve results that match the intent of the query. IR focuses on finding relevant documents or information from a large dataset based on a user's query. It is the backbone of search engines and recommendation systems.

IR involves two key processes:

- Indexing - storing data in a structured way to facilitate quick access.
- Matching - identifying documents relevant to the query using similarity scores or vector space models.

Modern IR systems even integrate multimodal models to analyze text, images, and videos, making searches more comprehensive.

Summarization is another powerful application of NLP that reduces the length of text while preserving its essential meaning. Summarization can be either extractive, which extracts key phrases directly from the original text, or abstract, which transcribes information in a condensed form. Summarization is widely used in journalism to produce article summaries, in education for condensing academic papers, and in AI systems like chatbots that provide quick overviews.

Many summarizer tools generate concise summaries while ensuring factual accuracy. Summarization makes it easier to consume large amounts of text, from research papers to long reports.

1.2.10

Which of the following describe summarization techniques?

- Extractive summarization selects key sentences from the original text
- Abstractive summarization rewrites the content in a condensed form
- Information retrieval indexes and matches documents
- Summarization classifies emotional tone in text

1.2.11

Question answering

Question answering (QA) is a field within natural language processing that focuses on creating systems capable of understanding and responding to human questions in natural language. These systems aim to provide accurate and relevant answers from a wide range of sources, such as text documents, structured databases, or knowledge graphs. QA systems have applications in virtual assistants, search engines, customer support, and education.

QA systems can be classified into different types based on their approach. **Open-domain** QA systems answer questions from unstructured text sources, such as articles or web pages, while **closed-domain** QA systems specialize in a specific domain or data set, such as medical research or legal documents.

QA can also be categorized as **extractive**, where the answer is taken directly from the text, or **generative**, where the system creates the answer based on the input.

Modern QA systems generally include the following components:

- **Question processing** identifies the type of question and relevant keywords. For example, a “what” question may indicate that the system should retrieve factual information, while a “why” question may contain explanations.
- **Document retrieval** selects documents or text passages that are likely to contain the answer.
- **Answer extraction or generation** finds or formulates the most appropriate answer from the selected text or data set.

Many QA systems use machine learning and deep learning techniques. Models such as BERT, RoBERTa, and GPT are widely used to understand context and extract information. In more advanced systems, transformers and attention mechanisms allow them to better handle long-range dependencies in text, improving accuracy and relevance.

Despite progress, QA systems face challenges such as handling ambiguous questions, contextualizing multiple references, and generating answers in creative or non-standard formats.

1.2.12

Which of the following are key components of a Question Answering system?

- Question Processing
- Answer Generation or Extraction
- Text Translation
- Grammar Correction

1.3 Short history

1.3.1

Before 1960

NLP has a rich history, beginning in the 17th century with foundational ideas by Gottfried Wilhelm Leibniz and René Descartes. Their work explored relationships between words and languages, forming the groundwork for machine translation technologies.

Early Foundations (1933-1950):

- Georges Artsrouni filed the first patent related to machine translation in 1933.
- In 1950, Sir Alan Turing published *Computing Machinery and Intelligence* and introduced the Turing Test, marking a significant leap in evaluating machine intelligence.

Initial Machine Translation Efforts (1952-1956):

- The first and second International Conferences on Machine Translation used rule-based and stochastic techniques.
- The 1954 Georgetown-IBM experiment successfully translated over 60 Russian sentences into English automatically, sparking optimism for rapid advancements in the field.

Linguistic Breakthrough and Setbacks (1957-1966):

- In 1957, Noam Chomsky proposed universal grammar, revolutionizing linguistics and influencing NLP.

However, the ALPAC report in 1966 highlighted slow progress in AI and machine translation, initiating the first AI winter.

1.3.2

What was the significance of the Georgetown-IBM experiment in the history of NLP?

- It marked the first successful automatic translation of Russian sentences into English.
- It inspired optimism about rapid advances in machine translation.
- It demonstrated that the machine translation problem could be solved within a few years.
- It was led by Noam Chomsky.

1.3.3

Early AI on NLP (1960-1970)

The 1960s and 1970s marked significant progress in Natural Language Processing (NLP) as researchers explored its potential in knowledge representation and question-answering systems. This era laid the foundation for linking human-computer interaction with AI techniques, though challenges remained.

- BASEBALL System (1961) was designed as a domain-specific Q&A system to answer baseball-related questions using basic natural language processing techniques.It was focused on human-computer interaction but was limited by restrictive inputs and basic language processing.
- AI-based Question-Answering Systems (1968) was an advanced system created by Marvin Minsky that used an inference engine to provide knowledge-based interpretations of questions and answers. It utilized an AI-driven inference engine for knowledge-based interpretation of queries.
- Augmented Transition Network (ATN) (1970) was proposed by William A. Woods, this system represented natural language inputs with structured networks to improve understanding and processing of linguistic information. Represented natural language inputs more effectively, advancing linguistic structures in AI.

Despite these advancements, early expert systems struggled to meet practical expectations. Their limitations contributed to the second AI winter, a period of reduced funding and interest in AI research.

1.3.4

Which advancements in NLP were made during the 1960s and 1970s?

- The BASEBALL system was an early domain-specific Q&A system.
- Marvin Minsky introduced an AI-based question-answering system in 1968.
- William A. Woods developed a universal translation engine for AI.
- Expert systems during this period completely met user expectations.

📖 1.3.5

Grammatical logic phase (1970-1981)

In the development of NLP, the grammatical logic phase marked a significant shift towards understanding and processing human language using advanced techniques. During this period, researchers focused on **knowledge representation**, **programming logic**, and **reasoning in AI**. These efforts aimed to make computer systems capable of comprehending and generating natural language more effectively.

One of the major advancements was the introduction of **SRI's Core language engine**, a powerful sentence processing tool. This system could parse complex sentences, breaking them down into meaningful components for analysis. It laid the groundwork for modern parsers, which are essential for tasks like grammar checking and natural language understanding.

Another breakthrough was the development of **Discourse representation theory (DRT)**. This theory introduced a pragmatic way to represent and interpret discourse - how sentences connect to each other in a logical way. DRT became instrumental in designing chatbots and Q&A systems that could hold coherent conversations by understanding the context of questions and answers.

However, progress in the 1980s faced significant challenges due to the limitations of computational power. Despite these hurdles, researchers focused on expanding **lexicons**, or structured collections of words and their meanings. These lexicons were essential for enabling NLP systems to understand a broader vocabulary and perform more sophisticated language tasks.

Most important facts for this period are:

- Researchers prioritized **knowledge representation** and **logical reasoning** during this phase.
- Tools like **SRI's Core language engine** and **Discourse representation theory** revolutionized sentence and discourse processing.
- Efforts were hindered by limited computational resources, but the expansion of **lexicons** enabled steady progress in NLP development.

📝 1.3.6

Which of the following advancements emerged during the grammatical logic phase of NLP?

- SRI's Core Language Engine
- Discourse Representation Theory
- Expanded lexicons to increase vocabulary coverage
- Neural Network-based translation tools

📖 1.3.7

NLP advances through machine learning (1981-2000)

The late 20th century marked a transformative period in NLP, driven by breakthroughs in machine learning and improved computing capabilities. During this era, researchers moved beyond rule-based and stochastic methods to embrace machine learning as a more efficient approach to language processing.

A key innovation was the Hopfield network, introduced by Professor John Hopfield. This neural network model demonstrated how machine learning could identify patterns in data, revolutionizing NLP and paving the way for neural approaches to language processing.

Advances in computing power and memory in the 1980s allowed researchers to combine Chomsky's linguistic theories with machine learning, improving understanding of grammar and syntax. This laid the foundation for more sophisticated language models and tools.

Lexical and corpus methods emerged during the same period. These techniques focused on creating grammars tied to specific words and using them in real-world contexts, thereby increasing the naturalness of NLP systems.

📝 1.3.8

Which of the following are milestones in NLP during the late 1980s and early 2000s?

- Introduction of Hopfield Networks for machine learning
- Integration of linguistic theories with machine learning
- Focus on rule-based systems for language processing

📖 1.3.9

NLP inovations (2000-2010)

The decade from 2000 to 2010 marked a significant leap forward, as machine learning and computing power advanced rapidly. The focus shifted to the use of statistical models and large datasets, leading to robust NLP applications across a variety of industries.

NLP research during this period was dominated by machine learning models. Algorithms such as support vector machines (SVMs) and decision trees were widely adopted for tasks such as text classification, sentiment analysis, and spam detection. The availability of labeled datasets allowed these methods to achieve significant accuracy in predicting and categorizing text data.

Statistical Language Models and Applications

Statistical language models have become the foundation of NLP tasks such as machine translation, speech recognition, and information retrieval. A notable development was Google Translate, introduced in 2006, which applied statistical translation techniques to break down language barriers. Similarly, latent Dirichlet allocation (LDA) emerged as a popular topic modelling algorithm that enables automated understanding of document topics.

The emergence of advanced applications

The decade saw the development of chatbots and question-and-answer systems that could interact more naturally with users. This era saw the early successes of Siri, Apple's virtual assistant (introduced in 2010), which used machine learning and NLP to process and answer users' questions.

IBM DeepQA and Watson

The IBM DeepQA project, launched in 2006, was a groundbreaking innovation of its time. Its goal was to develop a question-answering system capable of accurately understanding and answering complex questions. This project culminated in Watson, a system that later gained worldwide recognition for its success on the Jeopardy! game show.

Despite these advances, researchers faced challenges such as limited computing power, lack of context awareness, and difficulty understanding semantics.

1.3.10

Which advancements in NLP occurred between 2000 and 2010?

- Introduction of Google Translate using statistical methods
- Development of Siri as a virtual assistant
- Deep learning driving most NLP models
- IBM's DeepQA project leading to Watson

1.3.11

Big data, and deep learning (2010-present)

The period from the 2010s to the present has been transformative due to the rise of AI, big data, and deep learning. These advancements have propelled NLP research and applications to new heights, enabling systems to understand, generate, and interact with human language with unprecedented accuracy.

The integration of **cloud computing** revolutionized data accessibility and processing power. Mobile computing further expanded the reach of NLP, enabling its application in devices such as smartphones, virtual assistants, and IoT devices.

Big data allowed models to train on vast amounts of textual data, enhancing their ability to understand and predict human language.

The development of **deep learning techniques**, particularly **recurrent neural networks (RNNs)** and **long short-term memory networks (LSTMs)**, significantly improved the ability of NLP systems to process sequential data. These networks allowed machines to understand the context of words in sentences, capturing relationships between past and future words in text. This breakthrough enabled complex applications such as language translation, summarization, and sentiment analysis.

Contributions by Tech Giants

Companies like **Google, Amazon, and Facebook** spearheaded NLP innovations. For instance:

- **Google Translate** evolved into a highly effective translation service powered by neural networks.
- **Amazon Alexa** and **Google Assistant** brought virtual assistants into everyday life, leveraging NLP for voice-based interaction.
- **Facebook** improved content moderation and personalized recommendations using advanced NLP models.

Deep neural networks facilitated the creation of **Q&A chatbots**, **autonomous vehicles**, and **natural language generation (NLG)** systems. These innovations demonstrated NLP's potential to solve real-world problems, such as improving accessibility, enabling smarter customer support, and powering search engines.

The combination of NLP with big data and AI continues to push boundaries. Recent trends include:

- The development of **transformers** like GPT and BERT, which have revolutionized NLP by providing state-of-the-art performance in tasks like summarization, translation, and question answering.
- Efforts to create **ethically responsible AI systems** to reduce bias in NLP models.

1.3.12

Which developments significantly advanced NLP during the 2010s and beyond?

- Cloud computing and big data
- RNNs and LSTMs for sequential data
- Emergence of GPT and BERT models
- Rule-based systems as the primary approach

First Steps

Chapter 2

2.1 NLP components

📖 2.1.1

In the realm of NLP, there are three major components that work together to allow machines to process and generate human-like language. These components are:

- Natural language understanding (NLU) - breaks down language into syntax, semantics, and pragmatics to understand meaning.
- Knowledge acquisition and inferencing (KAI) - systems generate responses based on recognized language, overcoming the limitations of traditional rule-based systems.
- Natural language generation (NLG) - takes the machine's understanding and converts it into a coherent, natural language response, often in the form of speech or text.

Each plays a crucial role in ensuring that machines can interpret, process, and respond to human language in a way that is both meaningful and contextually appropriate.

📝 2.1.2

Which of the following components is responsible for converting the machine's understanding of language into human-readable text or speech?

- Natural Language Generation (NLG)
- Natural Language Understanding (NLU)
- Knowledge Acquisition and Inferencing (KAI)
- Natural Language Processing (NLP)

📖 2.1.3

Natural language understanding (NLU)

NLU is a core component in NLP that enables machines to comprehend human languages. It serves as the foundation for machines to process text or speech data and extract meaningful insights. NLU involves three primary layers of analysis:

- **Syntax analysis** focuses on the grammatical structure of a sentence. It identifies how words are related and ordered within a sentence, helping machines understand the structure of the sentence and its components, such as subjects, verbs, and objects. The goal is to parse the sentence into its syntactic elements and recognize the relationships between them. For example, in the sentence *"The cat sat on the mat"*, syntax analysis would help identify *"The cat"* as the subject and *"sat"* as the verb.
- **Semantic analysis** delves into the meaning of words and phrases within a sentence. It aims to determine how individual words contribute to the overall

meaning of the sentence. This process involves understanding word meanings, word senses, and their relationships to one another in context. For example, the word *"bank"* can have different meanings depending on context (e.g., a financial institution or the side of a river). Semantic analysis helps the machine resolve such ambiguities by using context.
- **Pragmatic analysis** looks at the broader context in which language is used. It focuses on understanding the intent behind a sentence and how it fits into a conversation or discourse. This involves considering factors like tone, prior conversation history, and cultural nuances, which are important for correctly interpreting meaning in communication. For example, the sentence *"Can you pass the salt?"* is a request, not a literal question about the ability to pass the salt. Pragmatics helps the machine recognize this intention.

By combining these layers, NLU enables machines to understand not just the individual components of language, but also the deeper meaning and intent behind what is being said. This approach allows machines to accurately interpret human language and prepare it for further processing, such as response generation or decision-making.

📝 2.1.4

Which of the following layers of NLU is responsible for determining the meaning of words and phrases in a sentence?

- Semantic Analysis
- Syntax Analysis
- Pragmatic Analysis
- Text Classification

🕮 2.1.5

Knowledge acquisition and inferencing (KAI)

KAI systems are designed to generate accurate and context-appropriate responses after a machine has successfully understood human language through Natural language understanding. KAI involves two key processes:

- **Knowledge acquisition - i**n this phase, the system gathers relevant information that will help it form a response. This could involve pulling information from databases, knowledge bases, or learning from previous interactions. The goal is to collect all the data that is pertinent to the current task or conversation. For example, in a customer service chatbot, knowledge acquisition might involve accessing product details or customer support documentation to provide a helpful answer.
- **Inferencing** - once the relevant information is gathered, inferencing takes place. This process involves drawing logical conclusions from the acquired knowledge. It requires the system to apply reasoning to make decisions or offer responses based on the information it has. This can be particularly

challenging in complex conversations, as the system must handle ambiguity, context, and multiple possible interpretations. For example, if a customer asks about the return policy for a product, the system must infer whether they are asking about general returns or a specific item.

The main challenge for KAI systems is that **human conversation** is often **complex** and full of nuance. Unlike traditional **rule-based systems**, which rely on simple "if-then-else" logic for queries and responses, KAI systems must navigate the subtleties of real-world conversations. Simple rule-based approaches are often insufficient because they cannot handle unexpected situations or understand the deeper meaning behind certain words or phrases.

To overcome these challenges, KAI systems are typically designed to work within **specific knowledge domains**. For example, in a medical context, a KAI system can be designed to understand medical terminology and respond appropriately to questions about symptoms or treatments. Similarly, a customer service chatbot in the insurance field may be trained on the specifics of policies and claims. In addition, **agent ontology** has been developed to help KAI systems handle more complex tasks, allowing them to navigate complex information and provide more accurate, context-aware responses.

2.1.6

Which of the following is the main challenge in Knowledge acquisition and inferencing systems that makes them more complex than rule-based systems?

- Navigating complexities and nuances of human conversation
- Ability to generate responses without context
- Limited to "if-then-else" queries
- Lack of access to domain-specific information

2.1.7

Natural language generation (NLG)

NLG is a component responsible in making machines capable of interacting with humans through natural, human-readable text or speech. Once a machine understands a conversation through techniques like NLU and KAI, NLG is responsible for converting that understanding into responses that can be easily interpreted by humans.

- **Answer and response generation** - systems are designed to generate answers, responses, and feedback during human-machine dialogues. For example, a chatbot answering customer queries will use NLG to craft an appropriate response based on the question or statement posed by the user. The goal is to ensure that the generated responses sound natural and align with the context of the conversation.

- **Text-to-Speech synthesis** - NLG is often combined with **text-to-speech synthesis** to enable machines to "speak" their responses. This process converts the generated text into audible speech. For example, a virtual assistant like Siri or Alexa uses both NLG and text-to-speech to provide users with verbal responses to their questions or commands.
- **Machine Translation vs. NLG** - although NLG can be considered a form of **machine translation**, it differs from traditional translation methods in that it does not translate between different languages. Instead, NLG focuses on producing responses in the **same language** as the input. It's akin to translating the machine's understanding of a conversation into human-like text or speech in a seamless manner.

The goal of NLG is to produce output that feels natural and conversational, making the interaction between humans and machines more engaging and efficient.

2.1.8

Which of the following best describes the primary function of NLG?

- Converts machine understanding into human-readable text or speech
- Translates between different languages
- Analyzes the meaning of words and phrases
- Acquires knowledge from human conversations

2.1.9

Which of the following activities are covered by Natural language understanding?

- Identifying sentence structure and relationships between words
- Understanding the context and intent behind the language
- Converting human-readable text into machine code
- Generating human-like responses based on text inputs

2.1.10

Which activities are covered by Knowledge acquisition and inferencing systems?

- Acquiring relevant information to generate responses
- Drawing logical conclusions based on information
- Generating answers to human queries
- Converting machine-generated responses into speech

2.1.11

Which tasks are part of Natural language generation?

- Converting machine understanding into human-readable text

- Generating human-like responses in natural language
- Converting generated text into speech
- Understanding the meaning of sentences and words

2.2 NLTK library

2.2.1

The **Natural Language Toolkit (NLTK)** is one of the most widely used Python libraries for natural language processing (NLP). It provides a comprehensive suite of tools for working with human language data, making it an essential resource for researchers, developers, and students interested in NLP tasks. NLTK was designed to simplify text analysis and language processing, enabling users to quickly get started with various NLP techniques.

NLTK is built on a modular architecture, offering tools that can handle everything from basic text preprocessing to advanced linguistic analysis. The library is well-suited for a wide variety of NLP applications, including tokenization, stemming, part-of-speech tagging, named entity recognition, and more. NLTK also includes a vast collection of corpora and lexical resources, which can be used for training and experimenting with NLP models.

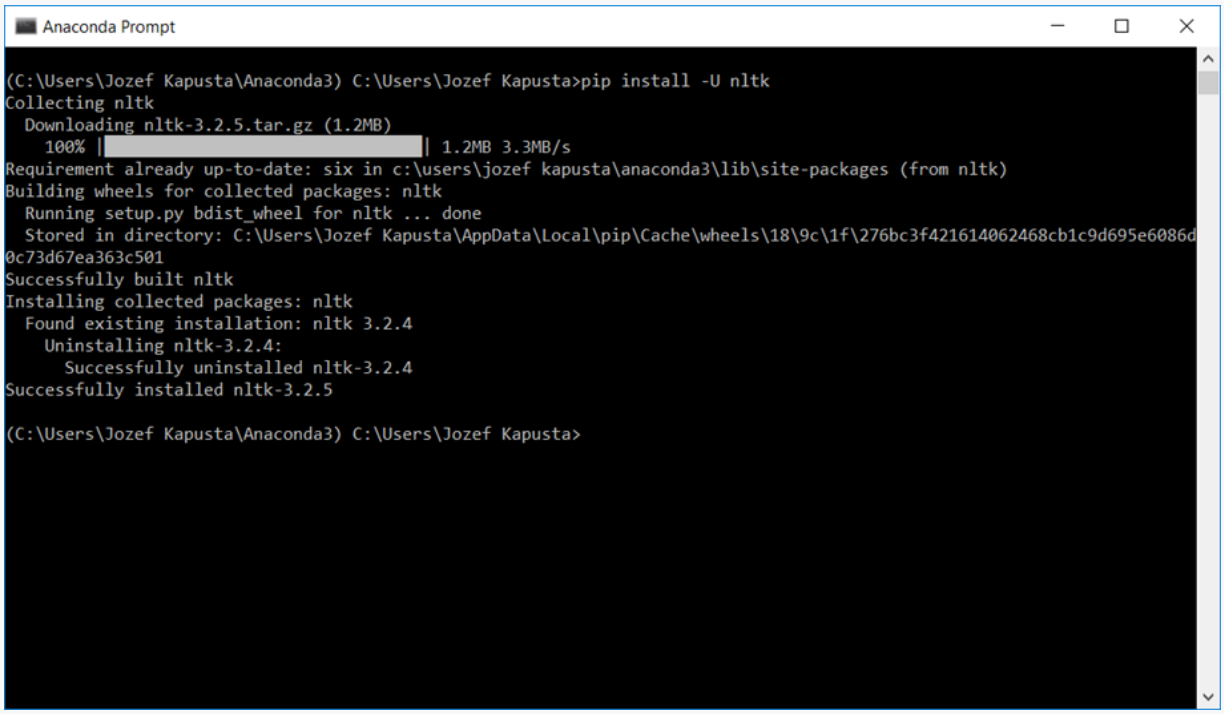
2.2.2

For natural language processing using Python, we will often use the NLTK (Natural Language Toolkit) library. Similarly, like many libraries, NLKT is also developed as open source. Originally, the library was designed for teaching, but later it was adapted to other areas due to its usefulness and robustness. The advantage of NLTK is that it supports most of natural language processing tasks and also provides access to many text corpora.

Several current environments have already the **nltk** library implemented inside. If we have an environment that does not have this library, the library can be installed directly in the Notebook without any problems using the **pip** command.

```
!pip install nltk
```

Similarly, using the **pip** command, the installation is also possible through the Anaconda Prompt (if we are using the Anaconda environment).



📖 2.2.3

Key features of NLTK

1. Text processing - provides simple tools for breaking down and processing text, such as tokenizing (splitting text into words or sentences), stemming (reducing words to their root forms), and lemmatization (mapping words to their dictionary forms).
2. Tagging and parsing - offers methods for tagging words with their parts of speech (POS) and analyzing the syntactic structure of sentences using grammars or parsers.
3. Named Entity Recognition (NER) - allows users to identify and classify named entities (e.g., people, organizations, locations) within text.
4. Corpora and lexicons - includes a large set of corpora (collections of texts) and lexical resources (such as WordNet, a lexical database of the English language).
5. Text classification - provides tools for creating and evaluating text classification models using machine learning algorithms.
6. Linguistic analysis - supports deep linguistic analysis, such as parsing syntactic structures, performing semantic analysis, and analyzing sentence structure in various ways.

Natural Language Understanding

Chapter 3

3.1 Speech recognition

📖 3.1.1

Natural language understanding

NLU refers to the machine's ability to interpret and understand human language in a meaningful way. NLU is a critical component of various applications, such as chatbots, virtual assistants, and language translation systems. The process of understanding spoken language can be broken down into four stages:

- **Speech recognition -** where the system converts spoken language into text. The system listens to human speech, processes the audio signals, and recognizes the words.
- **Syntactic (syntax) analysis** is used after speech is converted into text. It analyzes the structure of the sentence. Syntax analysis focuses on how the words in a sentence are ordered and how they relate to each other. It ensures that the machine understands sentence structure and grammar.
- **Semantic analysis** - at this stage, the machine understands the meaning of individual words and how they contribute to the overall meaning of the sentence. This involves identifying entities, actions, and relationships between elements in the sentence.
- **Pragmatic analysis** considers the context in which language is used, focusing on the intended meaning behind the words. It helps the system interpret things like tone, sarcasm, and implied meaning, ensuring the machine understands the real intent behind a sentence.

The typical applications of NLU are:

- Customer service chatbots where NLU helps chatbots understand user queries, process responses, and provide appropriate answers based on the user’s intent.
- Voice assistants (e.g., Siri, Alexa) rely on NLU to interpret spoken commands and perform tasks like setting reminders, playing music, or controlling smart devices.
- Language translation systems where NLU plays a key role in translating human language accurately by understanding both syntax and meaning.
- Text classification where NLU is used in sentiment analysis and spam detection to classify and filter text based on context and meaning.

📝 3.1.2

What are the stages involved in NLU?

- Speech recognition
- Semantic analysis
- Pragmatic analysis

- Data encryption
- Image processing

🕮 3.1.3

Speech recognition

Speech recognition is the **first stage** in NLU. It involves analyzing spoken language and converting it into text that can be processed by computers. The primary goal of this stage is to break down spoken words, or utterances, into distinct tokens that represent meaningful elements like words, sentences, and paragraphs.

Key components of speech recognition from linguistic point of view are:

- **Phonetic processing** - the process begins with phonetic analysis, where spoken language is broken down into individual sounds (phonemes). Helps in recognizing and distinguishing between different sounds, especially in languages with complex phonetic systems.This helps recognize words by identifying the smallest units of sound. Example: The word "uncanny" is analyzed into its phonetic components: "un" and "canny."
- **Phonological processing** deals with the sound patterns of language, including rules for combining sounds into syllables and words. It involves analyzing the sound structure of words to recognize patterns and understand how different sounds can affect word meaning. Example: Different accents or pronunciations of the same word can be understood by the system through phonological analysis.
- **Morphological processing** - morphology refers to the structure of words. It helps in recognizing word boundaries and identifying the meanings of words based on their components. The system breaks down words into their roots or stems and processes affixes (prefixes, suffixes, etc.) to determine meaning. Example: The word "uncanny" can be split into the root "canny" with the prefix "un-".
- **Spectrogram analysis** - spectrograms are visual representations of the frequency spectrum of sounds. Speech recognition models use spectrogram analysis to extract and identify the frequencies corresponding to words. It is used to analyze the acoustic properties of speech signals and extract relevant features for recognition. Example: The spectrogram for the word "uncanny" can identify distinct frequencies for "un" and "canny," helping the system distinguish between them.
- **Language-specific differences** - different languages have distinct phonetic and phonological rules, which affect how speech is recognized. It recognizes unique phonetic, phonological, and morphological characteristics that different languages have. Example: The same word spoken in English and Spanish may have different spectrograms due to the differences in pronunciation and accent.

Example: consider the word "uncanny". During the speech recognition process:

- The word is broken down into phonetic components: "un" and "canny".
- The system recognizes these components through spectrogram analysis, identifying the distinct frequencies that correspond to each part of the word.
- Morphological analysis identifies "un-" as a prefix and "canny" as the root word.

3.1.4

Which of the following is **NOT** a part of the speech recognition process in NLU?

- Data encryption
- Phonetic analysis
- Spectrogram analysis
- Morphological analysis

3.1.5

What does spectrogram analysis help identify in speech recognition?

- Distinct frequencies in speech
- Phonetic components of speech
- Word meanings
- Sentence structure

3.1.6

Project: Speech recognition

Identify words that are found in the following custom recordings using standard publicly available libraries.

Dataset

- original: open_speech, Audio-Samples-AMR-WB, sample-audio-files-for-speech-recognition
- local:
    1. harvard
    2. conference
    3. OSR_us

NLTK (Natural Language Toolkit) as previously introduced library does not include tools or modules specifically designed for speech recognition. Its primary purpose is the processing and analysis of text-based language after transcription, focusing on tasks such as tokenization, parsing, tagging, and sentiment analysis. For speech

recognition, we need to use a specialized library and then integrate it with NLTK for further processing.

SpeechRecognition provides easy access to Google Web Speech API, CMU Sphinx, and others. The functions convert spoken language into text. Example usage:

1. Import libraries

```
import speech_recognition as sr
import requests
```

2. Download the WAV file

```
url =
"https://priscilla.fitped.eu/data/nlp/speech/harvard.wav"
filename = "downloaded_harvard.wav"

# Download the file using requests library
response = requests.get(url)

# Check if download was successful
if response.status_code == 200:
  # Write the downloaded content to a file
  with open(filename, 'wb') as f:
    f.write(response.content)
else:
  print(f"Error downloading file. Status code:
{response.status_code}")
```

3. Analyze the downloaded file

```
recognizer = sr.Recognizer()
with sr.AudioFile(filename) as source:
  audio = recognizer.record(source)

  try:
    text = recognizer.recognize_google(audio)

    print("Text:", text)
  except sr.UnknownValueError:
    print("Sorry, could not understand audio")
  except sr.RequestError as e:
    print("Could not request results from Google Speech
Recognition service; {0}".format(e))
```

Program output:

```
Text: the stale smell of old beer lingers it takes heat to
bring out the odor a cold dip restores health and zest a salt
pickle taste fine with ham tacos al pastor are my favorite a
zestful food is the hot cross bun
```

3.1.7

Task: Speech recognition I.

Identify and copy the words identified by the recognizer

- File: https://priscilla.fitped.eu/data/nlp/speech/jackhammer.wav

```
# write your code
```

3.1.8

Task: Speech recognition II.

Identify and copy the words identified by the recognizer

- File: https://priscilla.fitped.eu/data/nlp/speech/OSR_us_000_0060_8k.wav

```
# write your code
```

3.2 Syntax analysis

3.2.1

Syntax analysis

Syntax analysis is the second stage of NLU after speech recognition. It is responsible for analyzing the structure of a sentence to understand its grammatical components. The primary goal of syntax analysis is to evaluate the grammatical structure of sentences, ensuring they conform to the rules of the language. The purposes are:

- **Syntactic correctness** - refers to the grammatical rules that govern the formation of sentences in a given language. The purpose of syntactic analysis in this context is to determine whether a sentence follows these rules. In order for a machine to correctly understand and process language, it must verify that the sentence is structured in a grammatically valid way. Grammatical errors or inconsistencies can cause confusion or ambiguity for both humans and machines. Syntactic analysis checks for common grammatical errors, such as subject-verb mismatches, misplaced modifiers,

incorrect punctuation, and sentence fragments. If a sentence contains a grammatical error, the parser flags or rejects it. Example. We have sentence: "She don't like pizza." The syntax parser will identify that the verb "don't" does not agree with the subject "she" (it should be "doesn't"). The sentence is syntactically incorrect and will be flagged.

- Syntactic structure refers to the arrangement of words in a sentence, where each word plays a specific role (such as subject, verb, object, etc.). **Syntactic analysis** breaks a sentence down into these components to understand how the words are related to each other. Understanding syntactic structure allows machines to interpret the meaning of a sentence by identifying how each word is related to the others. By analyzing the components of a sentence, machines can determine the relationships between the parts of the sentence (e.g., subject-verb-object). In syntactic analysis, a sentence is analyzed into its individual parts (often using a syntactic tree structure). The machine identifies the role of each word in the sentence, such as whether it is a subject, verb, object, or modifier. It also looks at how these parts interact in the sentence to convey meaning.

Example 1

Let's consider the sentence: **"Oranges to the boys"**

This sentence will be rejected by a syntax parser because it has a syntactic error. The sentence lacks a verb that ties the subject ("oranges") to the object ("boys"). Syntax analysis ensures that the structure of the sentence is valid according to the rules of the language. **Correct: "Oranges are to the boys."**

Here, the verb "are" makes the sentence grammatically correct, and syntax analysis can break it down into a subject ("oranges"), verb ("are"), and object ("the boys").

Example 2

Sentence: "The cat chased the mouse." - syntactic analysis would break this sentence down as follows:

- Subject: "The cat"
- Verb: "chased"
- Object: "the mouse"

By understanding this structure, the machine knows that the subject is performing an action (verb) on the object, which is critical for interpreting the meaning of the sentence.

Example 3

Sentence: "John gave Mary the book." - syntactic analysis breaks it down as:

- Subject: "John"

- Verb: "gave"
- Object: "the book"
- Indirect object: "Mary"

This structure helps the machine understand that John is the giver, Mary is the receiver, and the book is the item being given.

Example 4

Consider a more complex sentence: "While walking in the park, the dog saw a squirrel near the tree." - syntax analysis would break this sentence down into its core components:

- Subject: "the dog"
- Verb: "saw"
- Object: "a squirrel"
- Prepositional Phrase: "near the tree"
- Adverbial Clause: "While walking in the park"

The parser would also check the relationships between the parts of the sentence (e.g., the action "saw" is done by "the dog", and "a squirrel" is the object of the action).

📝 3.2.2

Which of the following sentences will be rejected by a syntax parser due to a grammatical error?

- They was going to the store.
- She enjoys reading books.
- The cat chased the mouse.
- John gave Mary the book.

📝 3.2.3

What does syntax analysis do in the process of NLU?

- Breaks down sentences into subject, verb, and object
- Checks if the sentence is grammatically correct
- Analyzes the meaning of individual words
- Generates human-like responses in dialogue systems

🕮 3.2.4

Applications of syntax analysis:

- **Grammatical validation** ensures that text or spoken language follows the grammatical rules of the language, reducing errors in further processing.

- **Language parsing he**lps break down complex sentences into manageable components for easier interpretation and further analysis, especially in tasks like machine translation or information extraction.
- **Question answering systems use synt**ax analysis as essential for interpreting the relationships between query terms and finding the correct answer in large datasets.
- **Speech-to-text systems** syntax analysis helps confirm whether the transcribed text is grammatically accurate.

3.2.5

Which part of the sentence "The boy quickly ran to the store" is the verb?

- Ran
- The boy
- Quickly
- To the store

3.3 Semantic analysis

3.3.1

Semantic analysis

Semantic analysis is the process of understanding the meaning of words, phrases, or sentences. It helps a machine move from simply knowing the structure of a sentence (syntax) to grasping its meaning (semantics). This analysis ensures that words and their combinations provide meaningful content, enabling effective communication.

While syntactic analysis focuses on the structure of a sentence, semantic analysis focuses on meaning. For a machine to truly understand human language, it must be able to interpret the intended message even when the words are correctly structured but do not make sense together.

For example, consider the phrase “hot snowflakes.” Although it has the correct syntactic structure (adjective + noun), it does not make logical sense because snowflakes cannot be hot. A semantic analyzer would reject this phrase as nonsense, despite its correct syntax.

3.3.2

Which of the following phrases will be rejected by a semantic analyzer as illogical or nonsensical?

- The cold snowflakes fell from the sky.
- The quick brown fox jumped over the lazy dog.

- The fish swam in the forest.
- He went to the bank to withdraw money.

📖 3.3.3

How semantic analysis works

- **Meaning of words** - each word in a sentence has a specific meaning in the dictionary. The semantic analyzer examines the meaning of each word and how it contributes to the overall meaning of the sentence. Example: In the sentence "The cat chased the mouse," the words "cat," "chased," and "mouse" have clear meanings. Semantic analysis confirms that the sentence describes a logical event (the cat chases the mouse).
- **Contextual and word relationships** - some words have multiple meanings depending on the context. For example, the word "bank" can refer to a financial institution or a riverbank. Example 1: "He went to the bank to cash a check." (In this case, "bank" refers to a financial institution.) Example 2: "The boat docked at the riverbank." ("bank" here means the edge of the river.) The semantic analyzer uses the context of the sentence to determine which meaning of the word is being used.
- **Checking for logical consistency** - the semantic analyzer also checks for logical consistency. If the words of a sentence, when combined, do not form a coherent idea or scenario, the analysis fails. Example: "Hot snowflakes were falling from the sky." The sentence is syntactically correct but semantically incorrect because snowflakes cannot be hot. A semantic analyzer would reject this as illogical or nonsensical.
- **Resolving ambiguities** - many words have multiple meanings (polysemy), or sentences can be interpreted in more than one way (semantic ambiguity). Semantic analysis aims to resolve these ambiguities by considering the context. Example: "She couldn't bear the weight of the box." "Bear" can mean to tolerate or carry something. The context (the weight of the box) helps the machine understand that in this case "bear" means to carry.

📝 3.3.4

Which word in the sentence "She went to the bank to fish" can have multiple meanings?

- Bank
- Went
- Fish
- To

📖 3.3.5

Applications of semantic analysis

- **Machine translation** – semantic analysis helps ensure that sentences are translated accurately by considering word meanings and context, rather than directly translating based on syntax.
- **Question answering systems** - in order for a machine to provide accurate answers, it must understand the intent of the question. For example, the sentence “Can you tell me the time?” is not just asking for the time – it is also a request for information, and the system must recognize this.
- **Information extraction** - in extracting meaningful data from large amounts of text, semantic analysis helps identify real-world entities contained in a document, such as names, dates, and places. It also helps understand the relationships between these entities.

📝 3.3.6

Which of the following applications relies on semantic analysis to ensure accurate interpretation of meaning in text?

- Machine Translation
- Information Extraction
- Syntax Checking
- Speech Recognition

📖 3.3.7

Examples of semantic analysis

1. Sentence: "The quick brown fox will jump over the lazy dog." - is syntactically and semantically correct because the words and phrases make sense and convey a clear image of the fox jumping over the dog.
2. Sentence: "Hot snowflakes were falling from the sky." - while the syntax is correct, the sentence is semantically incorrect because snowflakes cannot be hot. A semantic analyzer would call this a logical inconsistency.
3. Sentence: "He went to the bank to fish." - this sentence has semantic ambiguity. Does "bank" mean a financial institution or a riverbank? A semantic analyzer would use the context to understand that "bank" here probably refers to a riverbank.

📝 3.3.8

Which of the following sentences contains a semantic issue that would require further interpretation?

- She bought a bat at the store.

- I left my keys on the bank of the river.
- The company is looking for a new manager.
- He flew over the bank with his friends.

📖 3.3.9

Challenges in semantic analysis

- **Ambiguity** - many words have multiple meanings based on context. The challenge lies in correctly interpreting these meanings based on surrounding words and context.
- **Sarcasm and figurative language** - understanding sarcasm, metaphors, idioms, or cultural references is still a significant challenge for semantic analysis. For example, “break a leg” is an idiom that means “good luck,” but a machine may initially interpret it literally.
- **Contextual knowledge** - machines often need world knowledge to understand the meaning of certain words. For example, understanding the meaning of “snowflakes” requires knowledge of weather conditions, and understanding “bank” requires knowledge of different types of banks (financial or geographic).

📝 3.3.10

What does semantic analysis focus on in Natural language understanding?

- Determining if words have logical meanings and fit together sensibly
- Checking if a sentence follows grammar rules
- Identifying how words are syntactically structured
- Converting text into speech

3.4 Pragmatic analysis

📖 3.4.1

Pragmatic analysis is the fourth and most complex stage of Natural language understanding. It involves interpreting sentences or utterances based on real-world knowledge and context, beyond the literal meaning. While semantic analysis focuses on the dictionary meanings of words, pragmatic analysis requires understanding the underlying intent and context, which often involves common sense reasoning or expertise.

Pragmatic analysis goes beyond the individual meanings of words and considers how a sentence fits into a larger context. For example, the sentence "Will you crack open the door? I’m getting hot." uses the word "crack" in a figurative sense, meaning "to open," even though its literal meaning is "to break." A machine needs context to interpret this correctly.

Pragmatics often involves reasoning about the world and situations that require background knowledge. For example, if someone says, "Can you pass the salt?" in a dining context, pragmatic analysis understands that the speaker is likely requesting the salt to be passed, not asking about the physical ability to pass it.

Pragmatics also helps resolve ambiguities that can arise from multiple interpretations. For example, "I saw the man with the telescope" could mean that the speaker saw a man who had a telescope or that the speaker used a telescope to see the man. Pragmatic analysis helps choose the right interpretation based on context.

📝 3.4.2

Which of the following sentences requires pragmatic analysis to understand the intended meaning?

- Will you crack open the door? I’m getting hot.
- Can you open the window?
- Please hand me the book.
- She smiled at the gift.

🕮 3.4.3

Pragmatic analysis helps machines (or humans) interpret statements that go beyond the literal meanings of words. Context, real-world knowledge, and social norms play a major role in making sense of a sentence. Below are several examples of sentences that require pragmatic analysis to accurately interpret the intended meaning, beyond just the literal meanings of the words:

"Can you open the window?"

- Literal meaning: The speaker is asking if the listener has the ability to open the window.
- Pragmatic interpretation: The speaker is actually requesting that the listener open the window, not asking about the listener's ability to do so. Pragmatics interprets the sentence as a polite request, using "can" in a more functional, contextual way rather than literally.

"I’m going to the bank."

- Literal meaning: The speaker is going to a financial institution.
- Pragmatic interpretation: The word "bank" could also refer to the side of a river or an airfield. Pragmatic analysis would look at the context - such as the speaker’s location or the conversation topic (e.g., money or outdoor activities)- to decide whether the speaker means a financial institution or a natural bank.

"It’s getting cold in here."

- Literal meaning: The temperature in the room is dropping.
- Pragmatic interpretation: In context, this statement is likely a hint that the speaker wants someone to close the window or adjust the temperature. Pragmatics tells us that the sentence is a subtle way to request action, not just an observation.

"Can you pass me the salt?"

- Literal meaning: The speaker is asking whether the listener has the ability to pass the salt.
- Pragmatic interpretation: Here, "Can you" is used as a polite request, not a literal inquiry into the listener’s ability to pass the salt. Pragmatics helps interpret this as a request for action, given the context of the conversation (likely during a meal).

"I don’t have any cash, but I’ll pay you later."

- Literal meaning: The speaker does not currently have money, but intends to pay at a later time.
- Pragmatic interpretation: This sentence assumes an understanding that "pay you later" is a promise to fulfill the debt, despite the lack of cash at the moment. Pragmatic analysis considers the social context (trust, payment history) and interprets the sentence as a statement of intent.

"Could you lend me a hand?"

- Literal meaning: The speaker is asking if the listener has a hand they can give them.
- Pragmatic interpretation: "Lend me a hand" is an idiomatic expression that means "Could you help me?" Pragmatic analysis recognizes this idiomatic phrase and understands that the speaker is requesting assistance, not literally asking for a hand.

"She’s a real gem."

- Literal meaning: The speaker is comparing the person to a precious stone, implying that she is physically a gem.
- Pragmatic interpretation: The sentence is likely a compliment, implying that the person is valuable, kind, or exceptional. Pragmatics helps us understand that this is an expression of praise, not a literal statement.

"I love you."

- Literal meaning: The speaker is expressing strong affection or love for the listener.

- Pragmatic interpretation: Depending on the context, the meaning of "I love you" can vary. It could express romantic love, familial love, or even deep appreciation in a friendship. Pragmatics helps us determine the intended meaning based on factors such as the tone of voice, the relationship between the speaker and listener, and the context in which the statement is made.

"This task is a piece of cake."

- Literal meaning: The task is a dessert or cake that is easy to eat.
- Pragmatic interpretation: The phrase "a piece of cake" is an idiom meaning that the task is easy to accomplish. Pragmatics recognizes that the speaker is referring to the difficulty level of the task, not its literal nature.

"I'll call you when I get home."

- Literal meaning: The speaker is promising to make a phone call when they arrive home.
- Pragmatic interpretation: This sentence likely expresses a promise or intention rather than an immediate instruction. Pragmatics helps interpret this as a social agreement, understanding the common practice of keeping in touch once arriving home.

3.4.4

Which of the following statements requires pragmatic analysis to understand the speaker's intent?

- Can you open the window?
- I'll call you when I get home.
- I'm going to the bank.
- She's a real gem.

3.4.5

In the sentence "I'm getting cold," what does the speaker likely intend based on pragmatic analysis?

- The speaker is suggesting that someone close the window or turn up the heat.
- The speaker is merely stating that their body temperature is low.
- The speaker is asking someone for a blanket.
- The speaker is describing the cold weather outside.

3.4.6

Which of the following statements would need pragmatic analysis to determine whether the speaker is making a literal request or a polite suggestion?

- Could you lend me a hand?
- This task is a piece of cake.
- I love you.
- I'm going to the bank.

📖 3.4.7

While syntax and semantics deal with the structure and meaning of words, pragmatic analysis helps a machine understand the speaker's intent, the context of the conversation, and how the sentence fits into real-world situations.

Understanding context and intent

- Pragmatics focuses on the practical use of language in context. It requires knowledge of the world, common social practices, and the speaker's intentions. For example, the sentence "Could you open the window?" might sound like a question, but pragmatically, it's a polite request. A machine needs to understand that the speaker is asking for a favor, not just inquiring about someone's ability to open the window.
- Similarly, consider the sentence "I'm getting hot." On the surface, it may seem like a simple statement about temperature. However, pragmatically, it could imply that the speaker wants the temperature adjusted (e.g., by turning on the air conditioning or opening a window). A machine needs to recognize this social cue to interpret the speaker's intent correctly.

Resolving ambiguities

- Languages often have ambiguities that are resolved through pragmatic context. For instance, the phrase "Can you pass me the salt?" can be interpreted in two ways. In both cases, the sentence structure remains the same, but the intended meaning changes based on the context:
- As a literal question asking if the person is physically able to pass the salt.
- As a polite request asking the person to pass the salt.

Discourse understanding

- Pragmatics also helps machines follow the flow of a conversation. When people converse, they often rely on shared knowledge and references to earlier parts of the conversation. For example, in a conversation like:

1. Person A: "I'm starving."
2. Person B: "Do you want to go to a restaurant?"

- Without pragmatic analysis, a machine might interpret "I'm starving" as a mere statement about hunger, missing the underlying request to eat. Pragmatics helps understand the social cues and intent of the speaker.

Social context and politeness

- Understanding politeness forms another important application of pragmatic analysis. In many languages, speakers adjust their tone, word choice, and sentence structure based on social hierarchies and the level of formality. For example, saying "Could you please..." instead of a direct command ("Give me that") signals politeness. Pragmatic analysis interprets these signals, which is essential in applications like chatbots or customer service AI, where the system needs to respond appropriately based on the social context.

Speech acts theory

- One well-known theory in pragmatics is Speech act theory, which categorizes statements based on their function:

1. Assertives: Statements of fact, like "The sky is blue."
2. Directives: Requests or commands, like "Please close the door."
3. Commissives: Statements that commit to future actions, like "I will call you later."
4. Expressives: Statements of emotion, like "I'm sorry."

- Pragmatic analysis classifies sentences according to these speech acts to better understand what the speaker is trying to achieve, be it informing, requesting, or apologizing. This understanding is key for dialogue systems and virtual assistants, which must respond to different types of speech acts appropriately.

3.4.8

In the sentence, "It's getting cold in here," what is the likely pragmatic interpretation?

- The speaker is suggesting that someone close the window.
- The speaker is stating the temperature.
- The speaker is describing the outside weather.

3.4.9

Which of the following sentences requires pragmatic analysis to interpret its meaning?

- Can you help me with this?
- She is walking in the park.
- The sky is blue.

📖 3.4.10

Example applications in real-world systems

- **Chatbots and virtual assistants** like Siri, Alexa, or Google Assistant rely on pragmatic analysis to interpret ambiguous statements. For example, if a user says, "I'm thirsty," the system must interpret this as a request for a drink rather than simply noting the user's physical state. Similarly, if someone says, "Could you turn on the lights?", the system needs to understand the indirect request and respond appropriately.
- **Customer service AI** - in customer service applications, pragmatic analysis ensures that AI understands both the factual content of customer queries and the subtler, context-dependent meanings. If a customer says, "I'd like a refund for this faulty item," the AI needs to interpret the request in the context of the transaction and recognize it as a formal request rather than a casual complaint.
- **Machine translation** uses pragmatic analysis is vital part of machine translation systems. While syntax and semantics can handle the structure and meaning of words, pragmatic analysis ensures that idiomatic expressions or culturally specific phrases are translated appropriately. For instance, the English phrase "break a leg," meaning "good luck," would need to be understood in a contextual way when translating it into another language to avoid confusion.

📝 3.4.11

Why is pragmatic analysis important for customer service AI

- It helps the AI respond politely to customers.
- It helps the AI interpret ambiguous statements correctly.
- It allows the AI to translate different languages.

Knowledge Acquisition

Chapter 4

4.1 Knowledge acquisition

📖 4.1.1

NLP is a branch of AI that focuses on enabling machines to understand, interpret, and respond to human language. However, natural language is inherently complex because it relies on vast amounts of contextual, cultural, and semantic knowledge. For example, understanding a simple sentence like "It's raining cats and dogs" requires not just knowledge of vocabulary but also idiomatic expressions. To process language effectively, AI systems must access and utilize massive databases of knowledge. This requirement makes knowledge acquisition a critical foundation for the development of more advanced NLP systems.

Given the immense scale of knowledge needed, manually collecting and inputting this information is neither practical nor efficient. Automating the knowledge acquisition process is essential for advancing NLP capabilities. Techniques such as machine learning enable systems to extract knowledge from text, images, and structured data automatically. For instance, NLP models like ChatGPT or BERT use massive corpora of text to learn relationships, grammar, and semantics, allowing them to simulate human-like understanding and responses. Automation accelerates progress in NLP, bridging gaps in the system's ability to interpret and generate natural language.

While machine learning has achieved significant strides in improving performance, knowledge acquisition remains one of the biggest bottlenecks in building truly intelligent systems. Unlike incremental performance improvements - such as faster processing or slightly higher accuracy - knowledge acquisition addresses the fundamental challenge of equipping AI systems with the breadth and depth of information they need. Without sufficient knowledge, even the most advanced algorithms struggle to perform consistently across diverse tasks. By focusing research on improving automated knowledge acquisition, we can unlock the full potential of NLP and, by extension, intelligent systems as a whole.

📝 4.1.2

Which of the following best describes the relationship between knowledge acquisition and NLP?

- Knowledge acquisition is essential for NLP due to the complexity of natural language.
- NLP does not require knowledge acquisition to process language.
- Manual knowledge acquisition is sufficient for all NLP tasks.
- Machine learning has eliminated the need for knowledge acquisition.

4.1.3

Which of the following statements about knowledge acquisition and NLP are correct?

- Automating knowledge acquisition is necessary for advanced NLP.
- Knowledge acquisition is a bottleneck in developing intelligent systems.
- Manual knowledge collection is the primary method used in NLP.
- Incremental performance improvements are more important than knowledge acquisition in NLP.

4.1.4

Knowledge acquisition (KA) is a critical process in AI that involves obtaining and organizing information so that a computer system can use it to perform specific tasks. This process transforms raw data into structured information, often referred to as "knowledge" in AI. For instance, a system designed to diagnose medical conditions needs structured knowledge about symptoms, diseases, and treatments to make informed decisions. Knowledge acquisition ensures that the system has access to the right information in a usable format.

One key area of knowledge acquisition focuses on extracting structured information from natural language (NL). Natural language encompasses everyday communication, both spoken and written, such as conversations, textbooks, or emails. For example, an AI reading a science textbook might extract relationships like "photosynthesis produces oxygen." By processing these examples, the system converts unstructured text into a format it can use, such as a knowledge graph or database. This method is especially valuable because it enables AI to learn directly from human-created materials without manual intervention.

There are different approaches to acquiring knowledge for AI. Some involve interviewing human experts to gather insights, which are then translated into structured knowledge. Another method is asking experts to explicitly write rules, such as “if symptom X and symptom Y, then diagnose condition Z.” Alternatively, machine learning techniques allow systems to learn from examples of expert behavior, such as observing how a chess master plays the game. These approaches complement each other, and the choice often depends on the type of task and available data.

Unlike natural language methods, where the system extracts knowledge from unstructured sources like speech or text, direct knowledge expression involves formalizing information upfront. For example, an expert might manually encode knowledge into a rule-based system or train an AI with labeled datasets. While this can be precise, it requires significant effort and domain expertise. Natural language processing, on the other hand, reduces this burden by automating the extraction process, though it can be more challenging to ensure accuracy.

4.1.5

What is the main goal of knowledge acquisition in AI?

- To organize information so a computer system can perform tasks.
- To translate human knowledge into raw data.
- To write rules for all possible scenarios.
- To replace human experts completely.

4.1.6

Which of the following are methods of knowledge acquisition in AI?

- Interviewing human experts
- Writing formal rules
- Extracting information from textbooks
- Upgrading computer hardware

4.1.7

Entities

In knowledge extraction, entities are the essential building blocks. They represent real-world objects, concepts, or individuals, such as people, places, organizations, events, or equipment. For example, in a chemical accident database, entities might include specific chemicals, accident types, or the locations of incidents. Recognizing and categorizing entities is the first step toward creating structured knowledge that AI systems can utilize effectively.

Relationships

After identifying entities, the next step is to explore how they interact with one another. Relationships define these connections, capturing interactions like "located in," "caused by," or "part of." For instance, in academic research, relationships may link the background, objectives, and findings of a study. Extracting relationships helps AI systems make sense of how different pieces of information are interconnected, improving their ability to infer and analyze.

Semantic descriptions

Semantic descriptions enrich the extracted information by adding context and meaning. These can include attributes, properties, or definitions of entities. For example, in a chemical accident database, descriptions like "flammable" or "corrosive" provide essential details about substances. Similarly, academic papers can be enhanced by extracting semantic details such as the objectives and outcomes of the research. Semantic enrichment ensures a deeper understanding of the data.

4.1.8

What are entities in knowledge extraction?

- Real-world objects, concepts, or individuals
- Rules for analyzing texts
- Connections between databases
- Steps in the timeline of events

4.1.9

Which of the following are examples of relationships in knowledge extraction?

- Located in
- Part of
- Caused by
- Corrosive

4.1.10

Temporal information

Temporal information refers to details about time, such as dates, durations, and the sequence of events. This type of information is crucial for constructing timelines and understanding when events occurred in relation to one another. For example, in news analysis, temporal data helps establish whether one event happened "before," "after," or "during" another, creating a clearer picture of historical or ongoing events.

Sentiments

Sentiment analysis examines the emotional tone expressed in a text. This could involve identifying whether the sentiment is positive, negative, or neutral, as well as detecting specific emotions like happiness or anger. For instance, analyzing online articles to understand public opinions can reveal valuable insights into societal attitudes or community discussions.

Hidden themes

Latent topic structures focus on uncovering hidden themes or topics within a collection of texts. Topic modeling helps identify major subjects, detect trends, and group similar documents together. For example, in online communities of practice, topic modeling can reveal the recurring themes discussed by members, providing insights into shared concerns or interests.

4.1.11

What is the purpose of temporal information extraction in text analysis?

- To understand the timeline and order of events
- To detect emotions in text
- To summarize the main topics of a document
- To define the attributes of an entity

📝 4.1.12

Given the sentence:

"The Statue of Liberty, located in New York, was gifted by France in 1886."

Identify the entities as ordered in the sentence: _____, _____, _____

and their relationships as ordered in the sentence: _____, _____

- located in
- by
- Statue of Liberty
- New York
- The
- France
- was
- in 1886
- gifted by

4.2 Data sources

🕮 4.2.1

Data collection is a foundational step in building databases and enabling data-driven applications. It involves gathering information from various sources and organizing it for use in systems such as AI models, analytics tools, or decision-support applications. Effective data collection ensures the availability of accurate, relevant, and timely information, which is essential for making informed decisions.

The quality of data collected directly impacts the performance of systems relying on it. For instance, an AI model trained on incomplete or biased data may produce inaccurate predictions. On the other hand, a well-structured and comprehensive dataset can lead to insights that drive innovations, improve operational efficiency, and enhance decision-making processes.

Several approaches to collecting data exist, each suited to different scenarios. Some rely on manual input, while others harness automated tools like web scraping or APIs. Advanced techniques, such as subscribing to data feeds, allow organizations to access real-time information. Choosing the right method depends on factors like the scale of data required, the domain's complexity, and ethical considerations.

📖 4.2.2

Manual data entry

Manual data entry involves human operators directly inputting data into a database. This traditional approach is commonly used for smaller datasets or specialized domains where automated methods are impractical. For instance, researchers documenting rare historical artifacts may rely on manual entry to ensure detailed and accurate descriptions.

One advantage of manual data entry is its flexibility, as it allows customization and nuanced handling of unique information. However, it is time-consuming and prone to human errors, such as typos or inconsistencies. For large-scale datasets, these challenges can lead to inefficiencies and inaccuracies.

Despite its limitations, manual data entry remains valuable in scenarios requiring expert judgment or precise handling of highly specialized data. For example, cataloging medical case studies or archiving legal documents often necessitates the careful attention only a human can provide.

📝 4.2.3

What is a key limitation of manual data entry?

- Prone to human errors
- Requires extensive automation
- Inability to handle specialized data
- Lacks customization options

📖 4.2.4

Web scraping

Web scraping automates the extraction of data from websites. Using tools or scripts, web scraping software retrieves relevant information from online pages and converts it into structured formats suitable for databases. For example, an e-commerce platform might scrape competitor prices to adjust its own pricing strategies.

Web scraping is highly efficient for collecting large amounts of publicly available data, such as news articles or market trends. However, it requires careful attention to legal and ethical considerations, including copyright laws and terms of service. Improper use can lead to legal consequences or reputational harm.

This approach is widely used in industries like marketing, finance, and research. Applications include aggregating reviews, tracking stock prices, or analyzing consumer sentiment. Its automation and scalability make it ideal for handling dynamic and extensive datasets.

4.2.5

What is an ethical consideration when using web scraping?

- Copyright and terms of service compliance
- Automating data entry
- Ensuring manual accuracy
- Reducing scalability

4.2.6

APIs for data collection

Application Programming Interfaces (APIs) are interfaces provided by online services to enable programmatic access to data. Developers use APIs to retrieve information from specific sources in a structured and standardized manner. For instance, a weather app uses APIs to fetch real-time weather updates from a weather data provider.

APIs offer an efficient, reliable, and consistent way to collect data. They simplify integration between systems, reduce manual workload, and ensure up-to-date information. Additionally, APIs often come with documentation, making them easier for developers to implement and customize.

Common applications include financial platforms retrieving stock market data, social media apps accessing user metrics, or academic research tools gathering publication data. APIs streamline workflows, making them an indispensable tool in modern data collection.

4.2.7

What is a key advantage of using APIs for data collection?

- Real-time data retrieval
- Manual customization
- Legal ambiguity
- Lack of reliability

4.2.8

Data feeds

Data feeds provide a continuous stream of structured data from specific sources. Formats like RSS or Atom allow organizations to subscribe to relevant feeds and automatically update their databases with the latest information. This ensures access to timely and dynamic content without manual effort.

Data feeds are particularly valuable for domains requiring up-to-date information, such as news, sports, or financial markets. They allow seamless integration into databases, enabling organizations to focus on analyzing rather than collecting data.

Examples of data feed usage include aggregating news headlines, tracking sports scores, or monitoring changes in stock prices. Their automated nature ensures consistency and reduces the risk of human errors in data collection.

4.2.9

What is a primary benefit of using data feeds?

- Provides continuous, real-time data
- Requires manual updates
- Operates without standard formats
- Focuses on static content

4.2.10

Extracting information from Wikipedia

We can extract information programmatically from Wikipedia using the **wikipedia** Python library. This library provides a simple interface to query Wikipedia's MediaWiki API and retrieve information.

We will write a Python script to perform the following tasks:

1. Install and configure the **wikipedia** package.
2. Search for information about topic.
3. Retrieve and display key details, including their summary, full content, and categories.
4. Handle errors gracefully, such as missing pages or disambiguation.

By leveraging the wikipedia library, you can automate tasks like data extraction, research, and content generation, making your Python projects more efficient and informative.

- The wikipedia library can raise exceptions if the search fails or the page is not found. Use try-except blocks to handle these errors.
- Be mindful of Wikipedia's **rate limits** to avoid being blocked.
- While Wikipedia is a reliable source of information, **it's important to verify the accuracy of the retrieved data**, especially for critical applications.

Install the required library (in our system is installed)

```
!pip install wikipedia
```

Basic usage:

```
import wikipedia

try:
   # Search for a page
   search_query = "Python (programming language)"
   page = wikipedia.page(search_query)

   # Access page information
   print(page.title)  # Output: Python (programming language)
   print(page.summary)  # Output: A concise summary of Python
   print(page.url)  # Output: The Wikipedia URL for the page
   print('-' * 10)
   print(page.links[:10],'...')  # Output: A list of links
from the page
except wikipedia.DisambiguationError as e:
    print("Disambiguation error:", e.options)
except wikipedia.PageError:
    print("Page not found!")
```

Program output:

```
Python (programming language)
Python is a high-level, general-purpose programming language.
Its design philosophy emphasizes code readability with the use
of significant indentation.
Python is dynamically typed and garbage-collected. It supports
multiple programming paradigms, including structured
(particularly procedural), object-oriented and functional
programming. It is often described as a "batteries included"
language due to its comprehensive standard library.
Guido van Rossum began working on Python in the late 1980s as
a successor to the ABC programming language and first released
it in 1991 as Python 0.9.0. Python 2.0 was released in 2000.
Python 3.0, released in 2008, was a major revision not
completely backward-compatible with earlier versions. Python
2.7.18, released in 2020, was the last release of Python 2.
Python consistently ranks as one of the most popular
programming languages, and has gained widespread use in the
machine learning community.


https://en.wikipedia.org/wiki/Python_(programming_language)
----------
```

```
['"Hello, World!" program', '3ds Max', '?:', 'ABC (programming
language)', 'ADMB', 'ALGOL', 'ALGOL 68', 'API', 'APL
(programming language)', 'ATmega'] ...
```

More advanced usage:

```
# Find pages related to a topic
pages = wikipedia.search("Artificial Intelligence")
print(pages)

# Get a random article
random_page = wikipedia.random()
print(random_page)

# Set language
wikipedia.set_lang("fr")  # Set language to French
# shows 2 sentences summary
summary = wikipedia.summary("Python (langage de
programmation)", sentences=2)
print(summary)
```

Program output:

```
['Artificial intelligence', 'Artificial general intelligence',
'A.I. Artificial Intelligence', 'Ethics of artificial
intelligence', 'Applications of artificial intelligence',
'Generative artificial intelligence', 'History of artificial
intelligence', 'Artificial Intelligence Act', 'Artificial
intelligence in healthcare', 'Hallucination (artificial
intelligence)']
Mstislav Mstislavich
Python (prononcé /pi.tɔ̃) est un langage de programmation
interprété, multiparadigme et multiplateformes. Il favorise la
programmation impérative structurée, fonctionnelle et orientée
objet.
```

4.2.11

Project: Extracting information from Wikipedia about people

The goal is to retrieve and analyze information about three notable individuals (e.g., Barack Obama, Albert Einstein, and Marie Curie). Show only 3 sentences about every person.

You will write a Python script to perform the following tasks:

1. Install and configure the **wikipedia** package.
2. Search for information about three people.
3. Retrieve and display key details, including their summary, full content, and categories.
4. Handle errors gracefully, such as missing pages or disambiguation.

1. Install the required library (in our system is installed)

```
!pip install wikipedia
```

```
import
# Search for information about all persons eg Barack Obama
```

Program output:

```
Barack Hussein Obama II (born August 4, 1961) is an American
lawyer and politician who served as the 44th president of the
United States from 2009 to 2017. A member of the Democratic
Party, he was the first African-American president in U.S.
history.
```

4.3 Data extraction

🕮 4.3.1

Named-entity recognition

Named-entity recognition (NER) is an NLP technique that identifies and classifies entities within a text, such as names of people, organizations, locations, and dates. It serves as a foundational step in extracting meaningful information from unstructured data. By tagging these entities, NER helps bridge the gap between raw text and structured databases.

Initially, NER systems relied on handcrafted features and rule-based algorithms. However, modern advancements have introduced sequence-to-sequence neural architectures, which leverage machine learning to improve accuracy. These systems are now capable of processing large text corpora with high precision, thanks to pre-trained models and contextual embeddings.

NER is widely used in fields like customer support, where it extracts names and issues from client communications, or in journalism to automatically identify key figures and locations in news articles. For example, a NER model might analyze a news report to identify "John Doe" as a person, "Google" as an organization, and "California" as a location.

Analyzing the sentence, "Elon Musk announced new Tesla factories in Texas," an NER system would tag "Elon Musk" as a person, "Tesla" as an organization, and "Texas" as a location.

4.3.2

Which of the following are tasks performed by Named-entity recognition?

- Identifying names of people
- Extracting dates from text
- Tagging sentence sentiment
- Classifying locations

4.3.3

Relationship extraction

Relationship extraction focuses on identifying and structuring connections between entities within text. By mapping relationships, it enriches raw data with contextual links, enabling the creation of knowledge graphs that represent how entities interact with each other.

Early methods used rule-based systems, but modern approaches leverage deep learning models for better representation. Distant supervision techniques, which generate training data using existing knowledge bases, are often employed to train these systems. For instance, relationships like "is located in" or "is employed by" are detected and categorized using neural network-based representation learning.

Relationship extraction plays a key role in building search engines, chatbots, and recommendation systems. For example, in analyzing a sentence like "Tesla, founded by Elon Musk, is headquartered in Texas," this technique identifies relationships such as "founded by" between "Tesla" and "Elon Musk," and "headquartered in" between "Tesla" and "Texas."

Given "Apple acquired Beats in 2014," a relationship extraction system would identify "acquired" as the relationship linking "Apple" and "Beats."

4.3.4

Which of the following are examples of relationships extracted from text?

- Located in
- Headquartered in
- Founded by
- Sentiment is positive

4.3.5

Text Classification

Text classification is the process of assigning predefined categories to text. It enables systems to group similar text snippets based on their semantic meaning,

enhancing information organization and retrieval. Sentences, paragraphs, or entire documents can be classified into themes, topics, or roles.

Advanced language models such as Word2Vec, ELMo, and BERT revolutionize text classification by quantifying semantic similarity. These models compare the meanings of sentences and accurately assign categories, such as mapping a sentence about renewable energy to a "Sustainability" theme.

Text classification is used in spam detection, content moderation, and sentiment analysis. For instance, analyzing user reviews can classify comments as "positive," "negative," or "neutral." A research paper abstract can also be classified into ontology elements like "objective" or "methodology."

For example, in the sentence, "The study evaluates the impact of AI on education," a text classification model could classify it under "Research Objective."

4.3.6

Which are common uses of text classification?

- Spam detection
- Sentiment analysis
- Categorizing research abstracts
- Identifying relationships between entities

4.3.7

Ontologies in information extraction

Ontologies provide structured frameworks that define the concepts and relationships within a domain. Ontology-based information extraction uses these frameworks to map text content to predefined categories, ensuring semantic consistency and accuracy.

By using a semantically rich framework, this method reduces ambiguity in text interpretation. For example, mapping "AI" to "Artificial Intelligence" ensures consistency across different sources mentioning the same concept.

Ontology-based extraction is widely used in medical research to map symptoms and treatments to diagnostic categories. It is also applied in education, where course materials are mapped to learning outcomes or academic standards.

A system analyzing the text "Diabetes is managed by insulin therapy" could map "Diabetes" to a disease ontology and "insulin therapy" to a treatment ontology.

4.3.8

Which of the following describe ontology-based information extraction?

- Mapping text to predefined concepts
- Using structured frameworks
- Ensuring semantic consistency
- Identifying sentiments in reviews

📖 4.3.9

Direct memory access parsing

Direct memory access parsing (DMAP) is a model for understanding text by referencing stored knowledge patterns. It incrementally matches textual references to concepts in a knowledge base, enabling structured interpretation of text content.

DMAP matches phrases or words in the text to stored patterns, generating higher-order conceptual references. For example, it can interpret "Paris is the capital of France" by linking "Paris" and "France" through the concept of "capital."

DMAP is used in AI systems like the Learning Reader to extract formal representations of text meaning. It is particularly effective in creating structured knowledge bases for tasks such as automated question answering or summarization.

For example: Analyzing "The Mona Lisa was painted by Leonardo da Vinci," DMAP links "Mona Lisa" to "artwork" and "Leonardo da Vinci" to "artist," generating the relationship "painted by."

📝 4.3.10

Which of the following are aspects of DMAP?

- Incremental matching of references
- Generating higher-order concepts
- Linking text to stored patterns
- Identifying emotional tone

📝 4.3.11

Project: Named entity and relation extraction

Write a Python script to extract named entities and their relationships from a user-provided sentence.

1. Import libraries

- we will use **spacy**

```
import spacy
from spacy import displacy
```

2. Load pre-trained model

A pre-trained model, like **en_core_web_sm**, is essential for accurate and efficient natural language processing tasks. It covers needs of:

- **Part-of-speech tagging** for identifying the grammatical role of each word (e.g., noun, verb, adjective).
- **Dependency parsing** for analyzing the grammatical structure of sentences to understand relationships between words.
- **Named entity recognition** for identifying and classifying named entities within the text.

```
# Load a pre-trained English NLP model
nlp = spacy.load("en_core_web_sm")  # Provides tools for
tokenization, parsing, entity recognition, etc.
```

3. Function to extract named entities and relationships

- Iterates over the doc.ents to extract named entities and their labels.
- Stores the entities as tuples of (text, label) in the entities list.
- Checks if the token's dependency relation is either "nsubj" (nominal subject) or "dobj" (direct object).
- If so, extracts the subject (head token), object (token itself), and relationship (dependency label).
- Appends the extracted relationship as a tuple to the relationships list.

```
# Define a function to extract named entities and
relationships
def extract_entities_and_relationships(text):
    # Process the input text with the NLP model
    doc = nlp(text)

    # Extract named entities
    entities = [(ent.text, ent.label_) for ent in doc.ents]  #
List of entity text and their labels (e.g., PERSON, DATE)

    # Extract relationships using dependency parsing
    relationships = []
    for token in doc:  # Iterate over tokens in the parsed
text
        # Focus on tokens marked as subject (nsubj) or object
(dobj) dependencies
        if token.dep_ == "nsubj" or token.dep_ == "dobj":
            subject = token.head.text  # Head word of the
token (e.g., a verb)
```

```
            object = token.text       # Text of the token
itself
            relationship = token.dep_ # Dependency label
("nsubj" for subject, "dobj" for object)
            relationships.append((subject, object,
relationship))  # Store the relationship tuple

    return entities, relationships
```

4. Use

- experiment with structure of input sentence

```
# Example text input for processing
text = "Barack Obama served as the 44th President of the
United States from 2009 to 2017 in the US."

# Extract named entities and relationships
entities, relationships =
extract_entities_and_relationships(text)

# Display extracted named entities
print("Named Entities:")
for entity in entities:
    print(f"- {entity[0]} ({entity[1]})")  # Output entity
text and its label

# Display extracted relationships
print("\nRelationships:")
for relationship in relationships:
    print(f"- {relationship[0]} {relationship[2]}
{relationship[1]}")  # Output relationship structure
```

Program output:

```
Named Entities:
- Barack Obama (PERSON)
- 44th (ORDINAL)
- the United States (GPE)
- 2009 (DATE)
- US (GPE)

Relationships:
- served nsubj Obama
```

4.3.12

Project: Named entity and relation extraction by NLTK

Alternatively we can also use **nltk** library to write a Python script to extract named entities and their relationships from a user-provided sentence.

- Tokenize the sentence into words and parts of speech.
- Use nltk's named entity recognition (NER) to identify entities.
- Extract relationships using simple pattern matching or regular expressions.

1. Import libraries

- we will use **nltk**

```
import nltk
from nltk import word_tokenize, pos_tag, ne_chunk
```

2. Load necessary files

Following downloads are typically required for preprocessing and analyzing text in NLP tasks, particularly for:

- **Tokenization (punkt)** - is used for breaking text into sentences and words (tokenization). This resource is essential for any task where you need to split text into smaller units, like words or sentences.
- **POS tagging (averaged_perceptron_tagger)** - tags each word in a sentence with its part of speech (e.g., noun, verb, adjective). Is used in tasks requiring grammatical analysis or understanding sentence structure, such as dependency parsing or sentiment analysis.
- **Named entity recognition (maxent_ne_chunker)** - identifies named entities (like people, locations, and organizations) in a sentence. Extracts structured information from unstructured text by identifying significant entities.
- **Word validation (words)** - provides a database of recognized words to assist with various NLP tasks, such as validating word existence or enhancing named-entity recognition. Supports tasks like spell-checking, entity recognition, or token validation.

```
# Download necessary NLTK data files
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('maxent_ne_chunker')
nltk.download('words')
```

Program output:

```
[nltk_data] Downloading package punkt to
/home/johny/nltk_data...
```

```
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /home/johny/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already
up-to-
[nltk_data]       date!
[nltk_data] Downloading package maxent_ne_chunker to
[nltk_data]     /home/johny/nltk_data...
[nltk_data]   Unzipping chunkers/maxent_ne_chunker.zip.
[nltk_data] Downloading package words to
/home/johny/nltk_data...
[nltk_data]   Unzipping corpora/words.zip.
```

4. Use

- experiment with structure of input sentence

```
# Example text input for processing
sentence = "Barack Obama served as the 44th President of the
United States from 2009 to 2017 in the US."

# Tokenize and tag parts of speech
tokens = word_tokenize(sentence)
tags = pos_tag(tokens)

# Extract named entities
entities = ne_chunk(tags)
print("Named Entities:")
print(entities)
```

Program output:

```
Named Entities:
(S
  (PERSON Barack/NNP)
  (PERSON Obama/NNP)
  served/VBD
  as/IN
  the/DT
  44th/CD
  President/NNP
  of/IN
  the/DT
  (GPE United/NNP States/NNPS)
  from/IN
  2009/CD
```

```
to/TO
2017/CD
in/IN
the/DT
(GSP US/NNP)
./.)
```

4.4 Data (knowledge) structure

📖 4.4.1

While data structures like arrays, linked lists, and graphs are excellent for storing information, they might not be the most suitable for storing knowledge, which often involves complex relationships, context, and understanding.

Knowledge graphs

Knowledge graphs are a graph-based structure designed to store knowledge by representing entities as nodes and their relationships as edges. These relationships often include contextual or semantic meaning, making knowledge graphs highly suitable for complex knowledge representation. They are particularly useful in applications like semantic search, recommendation systems, and question-answering systems, where understanding the connections between different entities is key.

For instance, a knowledge graph about a book might include nodes for the book's title, characters, themes, and plot points. Relationships like "Character A interacts with Character B" or "Theme X is central to Plot Point Y" would connect these nodes, providing a rich, interconnected understanding of the book.

A notable example is Google’s Knowledge Graph, which powers its search engine by linking information about people, places, and things. If you search for "Albert Einstein," the system retrieves data about him, his discoveries, and related topics, all stored as a web of interlinked nodes and edges.

Knowledge graphs are also essential for handling vast amounts of structured and semi-structured data.

📝 4.4.2

Which of the following are applications of knowledge graphs?

- Semantic search
- Recommendation systems
- Question answering
- Spreadsheet management

📖 4.4.3

Ontologies

Ontologies provide a formal framework for representing knowledge as a set of concepts and their relationships. Unlike knowledge graphs, ontologies emphasize defining and categorizing concepts with a high level of precision, often using a hierarchical structure. They are widely used in semantic web technologies, knowledge management systems, and artificial intelligence.

For example, an ontology in the medical field might define entities like "disease," "symptom," and "treatment." Relationships such as "symptom is associated with disease" or "treatment alleviates disease" are explicitly mapped. This structured representation facilitates interoperability between different systems, such as electronic health records and diagnostic tools.

A popular example is the Gene Ontology, which organizes knowledge about gene functions across different biological systems. Ontologies ensure that the information is both machine-readable and human-interpretable, making them valuable in fields requiring high accuracy and integration.

The development of an ontology involves defining the domain, listing concepts, and specifying relationships. Tools like Protégé allow users to create and manipulate ontologies for various applications, from healthcare to manufacturing.

📝 4.4.4

What is the primary goal of ontologies in knowledge representation?

- To define and categorize concepts
- To emphasize the accuracy and interoperability of concepts
- To connect entities with physical data storage
- To perform statistical analysis

📖 4.4.5

Semantic networks

Semantic networks are graphical structures that store knowledge by representing concepts as nodes and their semantic relationships as edges. They focus on capturing meaning and associations, making them an effective tool for natural language processing, information retrieval, and expert systems.

For example, a semantic network for a restaurant might include nodes for "food," "drink," "cuisine," and "price." Relationships like "is a type of," "is served with," or "has a price of" link these nodes. This network can help answer complex queries such as, "What cuisines include vegetarian options under $20?"

Semantic networks are also valuable in education and knowledge exploration, allowing users to visualize how concepts relate. For instance, in biology, a network might map relationships between animals, their habitats, and their diets, creating an intuitive learning tool.

These networks have applications in building expert systems that simulate human reasoning. By explicitly storing semantic relationships, they enable machines to draw inferences and provide contextualized responses.

4.4.6

Which relationships can be represented in a semantic network?

- Is a type of
- Is associated with
- Has a price of
- Stores binary data

4.4.7

Knowledge bases

Knowledge bases store large amounts of structured and unstructured information in a single repository. They are designed to support diverse applications like virtual assistants, customer service chatbots, and question-answering systems by providing quick and accurate access to information.

A typical knowledge base might store product information, customer queries, troubleshooting guides, and FAQs. For example, a customer service chatbot uses a knowledge base to retrieve solutions for common issues like "How to reset a password?" or "Where to find the user manual?"

Knowledge bases integrate data from various sources, such as documents, databases, and APIs. They use indexing and tagging to ensure efficient retrieval, often powered by AI algorithms for enhanced performance.

Modern knowledge bases like IBM Watson Assistant go beyond simple storage by incorporating natural language processing and machine learning to interpret queries and provide intelligent responses.

4.4.8

Which of the following applications utilize knowledge bases?

- Virtual assistants
- Customer service chatbots
- Knowledge graphs
- Question-answering systems

📖 4.4.9

Machine learning models

Machine learning models are a dynamic way to store and utilize knowledge by learning patterns from data. Unlike static storage methods, ML models adapt to new data and make predictions or decisions based on learned information.

For instance, a machine learning model trained on customer feedback can identify sentiment, predict user preferences, and suggest personalized recommendations. Applications include recommendation systems, natural language processing, and computer vision.

A concrete example is a spam email filter, which uses an ML model trained on labeled datasets of spam and non-spam emails. The model learns features like word frequency, sender details, and formatting patterns to classify incoming messages.

While ML models are powerful, they require large datasets, careful training, and validation. Choosing the right model architecture and parameters is critical for achieving accuracy and reliability.

📝 4.4.10

Which tasks are commonly performed using machine learning models?

- Learning patterns from data
- Making predictions
- Adapting to new data
- Static storage of unstructured data

📖 4.4.11

Large language models

Large language models (LLMs), such as OpenAI's GPT or Google's BERT, represent a revolutionary way of storing and utilizing knowledge. These models are trained on vast amounts of text data, learning patterns, context, and semantic relationships within the language. Unlike traditional databases or structured systems, LLMs store knowledge implicitly in their parameters, enabling them to understand and generate human-like responses.

An LLM stores knowledge in the form of weights and biases within its neural network. During training, the model processes billions of sentences, learning associations between words, phrases, and concepts. For example, it might learn that "Barack Obama" is associated with "44th President of the United States" without explicitly being programmed with this information. The knowledge is

distributed across the network, making it flexible and robust for various applications.

LLMs are particularly suited for dynamic and complex tasks, such as answering questions, summarizing documents, and generating creative content. For instance, if you ask an LLM, "Who discovered penicillin?" it can provide the answer "Alexander Fleming" based on the patterns it learned during training. This eliminates the need for a traditional, structured knowledge base.

However, LLMs have limitations. Their knowledge is static after training, meaning they may lack awareness of new events unless retrained. Additionally, they require massive computational resources for training and inference. Despite these challenges, their ability to store and retrieve knowledge in a nuanced and contextualized manner makes them invaluable for modern AI applications.

4.4.12

What makes LLMs a unique way of storing knowledge?

- Knowledge is stored in neural network parameters
- They can understand and generate language contextually
- They require explicit programming of facts
- They are always up-to-date without retraining

4.4.13

Choosing the right approach

Selecting the best approach to store knowledge depends on the complexity of the data, the relationships involved, and the intended use. Often, multiple methods are combined to achieve an optimal balance between efficiency and functionality.

For example, a digital library might use a knowledge graph to map relationships between books, authors, and genres while employing a knowledge base to store detailed information about each book. Machine learning models could provide recommendations based on user preferences, and semantic networks might be used to visualize connections between literary themes.

Real-world applications often require flexibility. In an e-commerce platform, a semantic network could represent product categories, while an ontology ensures precise definitions, and a machine learning model personalizes the user experience.

4.4.14

What factors influence the choice of a knowledge storage approach?

- Data complexity
- Relationships involved
- Intended use
- Memory hardware type

Knowledge Inferencing

Chapter 5

5.1 Inference

📖 5.1.1

Inference as knowledge expansion

Inference is a key process in knowledge systems, enabling the derivation of new information from existing data. It acts as a bridge between what is known and what can be discovered, expanding the utility of knowledge representation systems. For example, inference can deduce that if "A is larger than B" and "B is larger than C," then "A is larger than C," even if this information wasn't explicitly provided.

This process is particularly important in domains like artificial intelligence, where large-scale knowledge bases like knowledge graphs are used. Knowledge graph reasoning, for instance, allows systems to infer new facts, such as identifying potential connections between entities based on their attributes and relationships. Commonsense knowledge, a cornerstone of intelligent reasoning, is often inferred automatically from pre-existing knowledge bases, highlighting the ability of inference to enhance understanding without additional manual input.

Applications of inference span a wide range of fields. In healthcare, for instance, inference systems can analyze structured medical data to suggest potential diagnoses. Similarly, in NLP, inferencing aids in semantic search, enabling systems to understand user intent and provide accurate responses. These examples demonstrate inference's pivotal role in extending the boundaries of knowledge representation.

📝 5.1.2

What is a primary function of inference in knowledge systems?

- To derive new facts from existing knowledge
- To explicitly store new data
- To limit the scope of knowledge representation
- To replace manual reasoning processes

📖 5.1.3

Inference on Structured knowledge representations

For inference to operate effectively, it relies on structured representations of knowledge. These structured forms often include entities, relationships, and logical propositions extracted from raw data sources like text.NLU plays a crucial role here, transforming unstructured text into structured formats that are easier to analyze and reason about.

Semantically enriched databases are prime examples of this. By employing ontologies and NLP techniques, they structure information to allow efficient inferencing. For instance, in a database of industrial accidents, cause-and-effect relationships can be inferred by analyzing structured records, helping identify patterns that might improve safety protocols.

Such structured formats also make information retrieval faster and more effective. Structured representations ensure that inference mechanisms have a clear framework to work with, enhancing the reliability and accuracy of the reasoning process. This structured approach is vital for applications ranging from recommendation systems to automated reasoning engines.

5.1.4

Why are structured knowledge representations crucial for inference?

- They provide a framework for efficient reasoning
- They simplify raw data storage
- They replace inferencing mechanisms
- They ensure data remains unprocessed

5.1.5

Rule-based reasoning

Rule-based reasoning is one of the oldest and most widely used inferencing techniques. It relies on pre-defined or dynamically generated rules to derive new knowledge. For example, a rule might state, "If a vehicle is electric, it does not produce direct emissions." Given data about a vehicle, this rule allows the system to infer its environmental impact without explicit data.

This approach excels in domains where relationships and patterns are well understood. In legal systems, rule-based reasoning can automate decision-making processes by applying legal rules to specific cases. Similarly, in medical diagnosis, systems can use predefined symptom-to-disease rules to suggest possible ailments.

However, rule-based systems are not without limitations. They can become brittle when dealing with incomplete or ambiguous data and often struggle to scale to complex scenarios with overlapping rules. Despite this, rule-based reasoning remains a powerful tool when applied to well-structured problems with clear logic.

5.1.6

What is a major strength of rule-based reasoning?

- It captures logical relationships effectively
- It can scale easily to complex problems

- It eliminates the need for structured knowledge
- It learns patterns without rules

📖 5.1.7

Distributed representation-based reasoning

Distributed representation-based reasoning uses mathematical embeddings to represent entities and relationships in vector spaces. These embeddings allow systems to perform operations, like determining similarity or deducing relationships, by analyzing patterns in the data.

For instance, in a knowledge graph, entities like "Paris" and "France" might be embedded into a space where the relationship "capital of" can be inferred based on vector proximity. These representations leverage statistical patterns in large-scale data, making them particularly useful for applications involving complex and dynamic relationships.

This approach has been widely adopted in NLP tasks like semantic search, where embeddings help match queries with relevant content. For example, a search query like "largest city in Japan" can be matched to "Tokyo" using distributed reasoning. Despite its strengths, this technique often requires substantial computational resources and can lack interpretability compared to rule-based methods.

📝 5.1.8

What does distributed representation-based reasoning rely on?

- Statistical patterns in vector spaces
- Predefined logical rules
- Manual feature engineering
- Explicit data relationships

📖 5.1.9

Neural network-based reasoning

Neural network-based reasoning applies deep learning models to infer complex patterns and relationships within structured knowledge. These systems are trained on large datasets to learn how different entities and attributes interact, enabling them to perform reasoning tasks like prediction and classification.

For example, a neural network might analyze a dataset of customer reviews to infer sentiment trends, identifying relationships between product features and customer satisfaction. In knowledge graphs, neural networks are used to complete missing links, such as inferring an unknown relationship between two entities.

This approach is particularly powerful for large-scale, unstructured data. However, it has challenges, including high computational requirements and a lack of interpretability. Despite these challenges, neural network-based reasoning has become a cornerstone of modern AI, driving innovations in areas like image recognition, recommendation systems, and autonomous systems.

5.1.10

Which is a common application of neural network-based reasoning?

- Filling missing links in knowledge graphs
- Rule-based decision systems
- Manual data extraction
- Eliminating inference challenges

5.1.11

Analogical reasoning

Analogical reasoning involves drawing inferences based on similarities between entities or concepts. This human-inspired technique enables systems to identify new properties or relationships by comparing them to existing, similar cases. For example, if two diseases share similar symptoms, analogical reasoning might suggest they also share similar treatments.

This type of reasoning is widely used in education, research, and creative problem-solving. In AI, analogical reasoning is applied in recommendation systems, where products are suggested based on similar user preferences or item attributes.

However, analogical reasoning requires robust similarity metrics to ensure accurate inferences. These metrics often rely on structured representations of knowledge, making them an essential complement to other reasoning methods like neural network-based approaches.

5.1.12

What does analogical reasoning primarily rely on?

- Similarities between entities or concepts
- Predefined logical rules
- Neural embeddings
- Statistical probabilities

5.1.13

Project: Inference engine

Develop a basic inference engine that identifies relationships between entities in a given text using rule-based reasoning and NLP techniques.

Objectives and procedure:

- Extract named entities from text (e.g., people, places).
- Identify relationships using dependency analysis and predefined rules for inferring relationships between entities.
- Deduce additional information or conclusions based on the extracted relationships.

1. import necessary parts

```
import spacy

# Load a pre-trained English model
nlp = spacy.load("en_core_web_sm")
```

2. Entity extraction

We use spaCy to identify entities (such as people, organizations, and locations) in the text. The function **extract_entities_and_relationships** processes the text using SpaCy to extract named entities (e.g., "Alice," "TechCorp") and relationships (e.g., "works for," "located in"). Relationships are identified using token dependencies such as "prep" (prepositions) and relationships between subject and object.

```
# Extracts named entities and relationships from text using
SpaCy.
def extract_entities_and_relationships(text):

    doc = nlp(text)
    # Extract named entities
    entities = [(ent.text, ent.label_) for ent in doc.ents]

    # Extract relationships, including inferred ones
    relationships = []
    for token in doc:
        if token.dep_ == "nsubj" or token.dep_ == "dobj":
            # Extract relationship for subjects and direct
objects
            subject = token.head.text
            object_ = token.text
```

```
            relationship = token.dep_
            relationships.append((subject, object_,
relationship))
        if token.dep_ == "prep":
            # Extract relationships for prepositions (e.g.,
"works for", "located in")
            subject = token.head.text
            object_ = ' '.join([child.text for child in
token.children])
            relationship = f"related_to ({token.text})"
            relationships.append((subject, object_,
relationship))

    return entities, relationships
```

4. Inference

- In the **infer_relationships** function, rules are applied to infer new relationships from existing ones. For example, "works for" and "located in" are both valid prepositions that represent relationships.

1. If a relationship contains "works for," we infer that the object employs the subject.
2. If a relationship contains "located in," we infer that the object contains the subject.

```
# Infers additional relationships based on simple rules.  """
def infer_relationships(relationships):

    inferred_facts = []
    for subject, object_, relationship in relationships:
        # Check and apply inference rules based on the
relationship type
        if "works for" in relationship:
            inferred_facts.append(f"{object_} employs
{subject}")
        elif "located in" in relationship:
            inferred_facts.append(f"{object_} contains
{subject}")
        elif "part of" in relationship:
            inferred_facts.append(f"{object_} contains
{subject}")
        elif "owns" in relationship:
            inferred_facts.append(f"{subject} is owned by
{object_}")
```

```
        elif "related_to" in relationship:
            inferred_facts.append(f"{subject} is connected to
{object_}")

    return inferred_facts
```

- ... and main code:

```
# Input text
text = "Alice works for TechCorp. Bob is located in New York."
# "Barack Obama, the 44th President of the United States, was
born in Honolulu, Hawaii."

# Extract entities and relationships
entities, relationships =
extract_entities_and_relationships(text)

# Perform inferencing
inferred_facts = infer_relationships(relationships)

# Print results
print("Named Entities:")
for entity in entities:
    print(f"- {entity[0]} ({entity[1]})")

print("\nRelationships:")
for subject, object_, relationship in relationships:
    print(f"- {subject} {relationship} {object_}")

print("\nInferred Facts:")
if inferred_facts:
    for fact in inferred_facts:
        print(f"- {fact}")
else:
    print("No inferred facts.")
```

Program output:

```
Named Entities:
- Alice (PERSON)
- TechCorp (ORG)
- Bob (PERSON)
- New York (GPE)

Relationships:
- works nsubj Alice
```

```
- works related_to (for) TechCorp
- located related_to (in) York

Inferred Facts:
- works is connected to TechCorp
- located is connected to York
```

5.2 Applications of inference

🕮 5.2.1

Knowledge graph

Inference plays a pivotal role in enhancing the functionality and completeness of knowledge graphs. A knowledge graph is a type of database that represents knowledge as a set of entities and their interrelations, making it highly useful in various fields like search engines, AI, and natural language processing. Inference, in this context, is the process of deriving new relationships or facts from existing data that may not be explicitly stated but can be logically concluded.

One of the key applications of inference is in knowledge graph completion. Often, knowledge graphs contain incomplete information due to limitations in data collection or human input. Inference mechanisms can predict the missing links between entities by identifying patterns and logical relationships from the existing nodes. For example, in a knowledge graph about movies, if the graph knows that "Leonardo DiCaprio" acted in the movie "Inception," inference could help link "Leonardo DiCaprio" to the movie "Titanic" based on their shared relationships with other actors or directors, even if that connection was missing initially.

Inference techniques used for knowledge graph completion include rule-based reasoning and machine learning. Rule-based reasoning applies predefined rules to fill in missing information, such as "If X acted in a movie with Y, and Y acted in a movie with Z, then X and Z may have worked together." Machine learning models, such as neural networks, can also learn patterns from vast amounts of data and predict missing relationships with a high degree of accuracy. These inferred connections make the knowledge graph more complete and useful, providing a richer source of information for applications like search engines and virtual assistants.

Example: Consider a knowledge graph that represents geographical relationships. If we know that "Mount Everest" is located in "Nepal," inference mechanisms could potentially infer that "Nepal" is part of "Asia" based on pre-existing relationships about continents. These inferences help enhance the graph's value in applications that require comprehensive, structured data.

5.2.2

Which of the following is an example of an inference applied to knowledge graphs?

- Predicting missing relationships between entities based on existing data.
- Storing raw data in a structured format
- Providing a direct query result
- Extracting entities from text

5.2.3

Question answering systems

Inference is essential in building intelligent question answering (QA) systems, particularly those that rely on knowledge bases. A knowledge base is a structured repository of facts, often organized as a knowledge graph, where entities and their relationships are defined. Inference allows QA systems to answer complex questions by reasoning over the available information, even when the exact answer is not explicitly stored in the knowledge base.

When a user asks a question, a well-designed question answering system doesn't just search for a direct match in the database. Instead, it employs inference techniques to deduce the answer by considering related facts. For instance, if a user asks, "Who is the CEO of Microsoft?" and the system has the fact "Satya Nadella is the CEO of Microsoft" explicitly in the knowledge graph, it can directly return the answer. But in cases where the relationship isn't directly stated, inference mechanisms like semantic reasoning or pattern recognition help derive an answer from existing knowledge.

Inference in QA systems typically involves logical reasoning over the relationships stored in the knowledge base. For example, the system may employ forward chaining (deriving new facts from existing ones) or backward chaining (starting from the query and working backward to find the supporting facts). If a user asks a question about a topic that involves multiple entities (like "Who won the Nobel Prize in Literature in 1995?"), the system can use inference to deduce the relationships between the "winner" and "Nobel Prize" and provide the correct answer.

Example: In a healthcare application, if a user asks, "What are the symptoms of COVID-19?" and the system has stored facts like "COVID-19 is caused by a virus" and "fever and cough are symptoms of viral infections," it could infer the connection between the virus and the symptoms and answer the query appropriately, even though "COVID-19" might not have been explicitly linked to "fever" and "cough" in the data.

5.2.4

How does inference improve the performance of question answering systems?

- It helps the system infer answers even when exact matches do not exist.
- It allows the system to generate answers from predefined statements.
- It limits the system to answering only simple factual queries.
- It extracts raw data without considering relationships.

5.2.5

Recommender systems

Recommender systems have become a cornerstone of modern digital experiences, helping users discover products, services, or content tailored to their preferences. Inference plays a critical role in these systems, enabling them to go beyond basic matching and make intelligent predictions about what users might like.

For instance, traditional recommendation methods might suggest items based solely on direct similarities, such as recommending books in the same genre. However, inference allows recommender systems to utilize relationships in a knowledge graph. If a user frequently purchases books on leadership, the system can infer interest in related topics, like productivity or biographies of leaders. This reasoning is made possible by understanding not just the items themselves but the connections between them.

One practical example is in movie recommendation platforms. A system might infer that a user who enjoys movies directed by Christopher Nolan and featuring intricate plotlines would also appreciate films by other directors known for similar storytelling styles. This capability is powered by algorithms that analyze relationships in the knowledge graph, such as "directed by," "genre," or "similar audience appeal," making recommendations more personalized and dynamic.

Inference also enables systems to adapt to sparse data. If a new user with minimal browsing history joins a platform, the system can infer preferences based on broad relationships, like "users from this region often prefer these types of content," ensuring a seamless experience.

5.2.6

Which of the following demonstrates the use of inference in recommender systems?

- Recommending products based on relationships in a knowledge graph.
- Suggesting movies similar to those previously liked by the user.
- Displaying items in a random order with no consideration of user preferences.
- Recommending only the most popular items on the platform.

📖 5.2.7

Commonsense reasoning and everyday scenarios

Commonsense reasoning allows machines to mimic human intuition, drawing logical conclusions in everyday scenarios. Unlike explicit knowledge, commonsense knowledge encompasses unstated facts that humans take for granted. For example, if it is raining, one might infer that carrying an umbrella is sensible. Machines achieve this through inference mechanisms designed to work with knowledge graphs or commonsense databases.

A key application of commonsense reasoning is in virtual assistants. For instance, if a user says, "I'm running late to the airport," the assistant can infer that the user might need help with tasks such as finding flight details, traffic updates, or directions. This reasoning relies on understanding relationships like "running late" implies "urgency" and "airport" implies "flight travel."

Another example is autonomous systems, such as self-driving cars. A car might infer that a pedestrian near a crosswalk is likely to cross the street, even if they haven’t stepped onto the road yet. Such inferences are vital for safety and decision-making in real-time.

By enabling machines to "fill in the gaps" of knowledge, common sense reasoning opens the door to intelligent systems capable of making nuanced decisions. It bridges the gap between rigid rule-based systems and human-like understanding, enhancing user interactions and functionality in diverse applications.

📝 5.2.8

What is an example of commonsense reasoning in AI?

- Inferring urgency from a statement about running late.
- Predicting that a pedestrian near a crosswalk might cross the street.
- Stating the exact current temperature when asked about the weather.
- Providing directions to a destination without considering road closures.

📖 5.2.9

While inference is a powerful tool for expanding and applying knowledge, it faces several significant challenges that researchers are actively addressing. These challenges highlight the complexities of making inference methods more effective, efficient, and applicable across diverse domains.

One critical issue is **interpretability**. Many advanced inference techniques, such as those based on distributed representations or neural networks, are often seen as "black boxes." While they can produce highly accurate results, the reasoning behind these results is difficult to understand or explain. This lack of transparency can hinder trust and adoption, especially in sensitive applications like healthcare or

legal systems. Researchers are therefore exploring ways to make these methods more interpretable, such as by designing models that provide human-readable explanations for their inferences.

Another challenge is **scalability**, particularly when working with large-scale knowledge graphs. Performing inference on vast datasets with complex relationships requires significant computational resources. For instance, recommending personalized content for millions of users or reasoning over dynamic data streams demands efficient algorithms and optimized data structures. Addressing this requires innovations in algorithm design to balance accuracy and computational efficiency.

The incorporation of **multi-source information** is another pressing need. Current inference methods often rely on a single type of data, such as text. However, real-world scenarios often involve diverse sources, including images, sensor data, and unstructured text. For example, an autonomous vehicle must integrate information from cameras, radar, and maps to make safe driving decisions. Developing inference mechanisms capable of seamlessly combining these diverse inputs is a key area of research.

Lastly, the field is moving toward **dynamic knowledge reasoning**. In many applications, knowledge is not static but evolves over time. Social networks, for instance, constantly generate new relationships and interactions. To remain accurate and relevant, inference systems must be able to update and adapt as new information becomes available, rather than relying on static data snapshots.

By addressing these challenges, the future of inference research promises systems that are not only more powerful and efficient but also more transparent, adaptable, and capable of integrating knowledge from diverse, ever-changing sources.

5.2.10

Which of the following is a challenge faced by inference research?

- Making inference methods more interpretable and transparent.
- Limiting inference to text-based data only.
- Scaling inference for large, complex knowledge graphs.
- Avoiding the use of dynamic knowledge in reasoning.

Natural Language Generation

Chapter 6

6.1 NGL applications I.

📖 6.1.1

Natural Language Generation (NLG) is a field of artificial intelligence focused on enabling machines to produce human-like text from structured or unstructured data. It transforms raw information into coherent, meaningful language, making it easier for people to understand and interact with data. Applications of NLG range from creating summaries, stories, or translations to powering dialogue systems and personalized communication tools. By bridging the gap between complex data and human language, NLG plays a pivotal role in enhancing accessibility and user experience across various domains.

Data-to-text generation

Data-to-text generation is a transformative application of natural language generation that converts raw data into meaningful textual summaries. This technology is widely used to make data more accessible and actionable, particularly in fields where large datasets need to be communicated effectively. For example, financial analysts rely on NLG systems to generate quarterly performance reports that summarize revenue, expenses, and growth trends from numerical data. Similarly, meteorological agencies use this technology to produce weather forecasts, converting temperature readings, humidity levels, and wind speeds into clear, human-readable reports.

The process involves analyzing structured data and identifying patterns or key points to highlight. Advanced NLG systems often incorporate statistical models and templates to ensure that the output is accurate and contextually appropriate. For instance, a financial NLG tool might be programmed to detect unusual spikes in revenue and highlight them in the generated text as potential areas of interest for stakeholders.

One of the most significant advantages of data-to-text generation is its ability to handle repetitive or labor-intensive tasks. Instead of requiring human effort to manually draft reports, NLG systems can produce summaries in seconds, freeing professionals to focus on higher-level analysis. Furthermore, these systems can tailor content for specific audiences, generating detailed technical summaries for experts or simplified overviews for general readers.

📝 6.1.2

Which is an example of data-to-text generation?

- Generating a weather forecast summary from meteorological data.
- Translating a novel from English to French.
- Enabling a chatbot to answer customer service queries.
- Writing a fictional story from a creative prompt.

📖 6.1.3

Machine translation

Machine translation is an NLG application that focuses on converting text from one language to another. This technology has revolutionized global communication, breaking down language barriers and enabling people from diverse linguistic backgrounds to interact seamlessly. Tools like Google Translate and DeepL utilize machine translation to provide quick and often accurate translations for millions of users daily. For instance, a tourist in Japan can input a restaurant menu into a translation app to understand their meal options in their native language.

The underlying process involves complex algorithms, including neural machine translation models that analyze the grammatical structure, syntax, and semantics of sentences in the source language. These models then generate equivalent sentences in the target language while maintaining contextual meaning. Advances in machine learning have significantly improved the accuracy of translations, particularly for less commonly spoken languages.

Despite its advancements, machine translation faces challenges, such as accurately translating idiomatic expressions or phrases that have cultural nuances. For example, the French phrase *"c'est la vie"* is more than its literal translation ("that's life"); it conveys a philosophical acceptance of circumstances, which a machine might miss without proper context. Future developments aim to enhance cultural sensitivity and contextual understanding to address these limitations.

📝 6.1.4

What is a primary use of machine translation?

- Translating a speech or text from Spanish to English.
- Generating conversational responses for chatbots.
- Writing an original story from structured data.
- Summarizing financial performance reports.

📖 6.1.5

Dialogue systems

Dialogue systems enable computers to engage in human-like conversations, forming the backbone of virtual assistants like Siri, Alexa, and customer support chatbots. These systems combine natural language understanding (NLU) to interpret user input and NLG to generate meaningful and contextually relevant responses. For example, when a user asks, "What's the weather today?" a dialogue system retrieves weather data and generates an appropriate answer, such as, "It's sunny with a high of 75°F."

Developing effective dialogue systems requires training them to handle varied user intents and conversational flows. A well-designed system can manage complex queries, provide follow-up questions, and even exhibit personality traits to enhance user engagement. For instance, an AI-powered travel assistant might suggest, "Based on your preferences, would you like to explore budget-friendly hotels or premium accommodations?"

One challenge in dialogue systems is ensuring coherence and context awareness. The system must remember prior exchanges to avoid repetitive or irrelevant responses. Advanced models like ChatGPT leverage contextual embeddings to maintain conversational history, improving the flow of multi-turn conversations.

📝 6.1.6

Which is an example of a dialogue system application?

- Engaging in customer support conversations via a chatbot.
- Translating a scientific paper from German to English.
- Generating detailed weather summaries from meteorological data.
- Writing a creative story from a user-provided prompt.

📖 6.1.7

Story generation

Story generation is an exciting application of NLG that uses structured data or prompts to create engaging narratives. These systems are used in gaming, education, and creative industries to craft immersive stories tailored to user preferences. For instance, in a role-playing game, a story generator might develop unique side quests based on the player's decisions, enriching their experience.

The process involves a mix of creative algorithms and user inputs. Modern story generators leverage large language models trained on diverse datasets, enabling them to mimic different genres, tones, and styles. For example, a user could provide a prompt like "A knight ventures into a haunted forest," and the system could produce a suspenseful story with vivid descriptions and a coherent plotline.

A key challenge in story generation is maintaining narrative consistency and logical progression. Systems must ensure that characters act in alignment with their traits and that events follow a plausible sequence. As technology advances, story generators are becoming increasingly capable of creating compelling, human-like narratives, sparking new possibilities in entertainment and education.

📝 6.1.8

What is a primary use of story generation in NLG?

- Crafting unique quests for a video game based on user actions.

- Translating a user manual from English to Japanese.
- Generating a weather report based on real-time data.
- Engaging in customer support dialogues for retail services.

6.2 NLG applications II.

📖 6.2.1

Summarization

Summarization in NLG involves creating a concise and meaningful summary of a larger text. This is achieved by identifying and extracting the most important points while maintaining the original context. Summarization can be broadly categorized into **extractive summarization**, where sentences or phrases are directly taken from the original text, and **abstractive summarization**, which involves generating new phrases to convey the main idea in a natural, human-like manner.

Summarization is essential in various fields. For example, in journalism, it helps condense lengthy articles into headlines or short briefs. In research, it creates abstracts for scientific papers, enabling faster comprehension of core findings. Summarization also finds utility in customer service, where summarizing call logs can help agents quickly understand customer issues.

Imagine a legal professional working on a case. Instead of reading hundreds of pages of a deposition, a summarization tool could generate a concise summary, highlighting key facts and arguments, saving time and improving efficiency. Similarly, news platforms like Google News often provide summaries of trending articles, allowing readers to grasp the essence of stories at a glance.

📝 6.2.2

Which of the following are examples of summarization applications?

- Generating abstracts for scientific research papers.
- Producing concise versions of lengthy legal documents.
- Translating a document from English to Spanish.
- Generating automated programming code from natural language.

📖 6.2.3

Code generation

Code generation refers to using NLG techniques to translate natural language descriptions into programming code. This process leverages trained models like OpenAI Codex, enabling developers to focus on higher-level problem-solving while the tool automates repetitive or boilerplate coding tasks.

Code generation tools are widely used in software development. They assist in creating unit tests, database queries, API integration scripts, or even entire applications. For instance, a developer might describe a functionality in plain English, and the model generates Python or JavaScript code to implement it. This technology is valuable in both learning and professional environments, providing hands-on assistance and reducing errors.

Suppose a user needs to create a function in Python that calculates the factorial of a number. By typing "write a Python function to calculate the factorial of a number," a code generation tool could instantly provide a working implementation, complete with proper syntax and comments. Similarly, a data analyst could ask for SQL queries to extract insights from databases without writing them manually.

6.2.4

Which statements about code generation are true?

- It automates the creation of software by generating code from natural language.
- It is useful for learning to program by providing examples and suggestions.
- It generates summaries of programming-related documents.
- It directly generates scientific research abstracts.

6.2.5

Personalized and adaptive communication

Personalized and adaptive communication uses NLG to tailor messages to an individual's preferences, behavior, or context. This can involve adjusting tone, content, or delivery based on factors like user demographics, previous interactions, or preferences. Such communication is vital in marketing, customer support, and user engagement.

One of the most common applications of this technology is in **chatbots and virtual assistants**. Bots like those used in e-commerce can provide product recommendations based on a customer's browsing history or answer specific questions about orders. Adaptive communication also enhances user experience in education, where chatbots adjust responses based on a student’s progress or understanding of a topic.

Consider a healthcare chatbot designed to assist patients with medication reminders. It could personalize interactions by using the patient’s name, reminding them of specific dosages, and adapting to their queries about side effects or alternative medications. Similarly, email marketing tools can create highly tailored promotional messages, dynamically adjusting content for different customer segments to improve engagement rates.

6.2.6

Which statements about personalized and adaptive communication are true?

- It involves tailoring messages to individual preferences or behavior.
- It is widely used in chatbots for customer service and e-commerce.
- It limits communication to fixed, pre-written responses.
- It has no relevance in educational applications.

6.3 NLG phases

6.3.1

Natural Language Generation (NLG) involves transforming raw data or structured information into human-readable text. While the end result may appear seamless, the process behind it is complex, requiring a series of well-defined phases to ensure the output is coherent, relevant, and grammatically correct. These phases, each addressing a critical aspect of language generation, work together to produce text that accurately reflects the intended meaning. By breaking down these stages, we can better understand how machines achieve such an intricate task. Below, we explore each phase in detail, complete with examples to illustrate their complexity and functionality.

6.3.2

Content determination

Content determination is the foundational phase of NLG, where the system decides what information to include in the generated text. This step is crucial for relevance, as including unnecessary or incomplete data could confuse the user or dilute the message. The system typically prioritizes data based on its importance and the user's needs, ensuring that only the most pertinent information is selected.

For instance, in a weather-reporting system, the system must decide whether to include data on temperature, humidity, wind speed, or severe weather alerts based on the context. If a storm is approaching, the system may prioritize details about its expected arrival and impact, omitting less critical information like daily humidity levels.

Content determination often relies on rules, machine learning models, or user preferences. For example, financial report generators might focus on key performance indicators like revenue and profit, ignoring granular details unless explicitly requested. This phase ensures the generated text serves its intended purpose effectively.

6.3.3

Which of the following are true about the content determination phase in NLG?

- Decides what information to include in the text.
- Prioritizes data based on importance and user needs.
- Ensures grammatical correctness of sentences.
- Focuses solely on choosing words and phrases.
- Organizes the text into a coherent structure.

6.3.4

Text planning

Once the content is determined, text planning organizes the information into a logical and coherent structure. This phase focuses on creating a blueprint for how the data will be presented, taking into account factors like discourse flow, emphasis, and rhetorical goals. A well-structured text ensures the reader can easily follow the narrative or argument.

For example, in a news article generator, text planning may involve placing the most critical news upfront (a headline or lead), followed by supporting details, and concluding with background context. This mirrors the "inverted pyramid" style common in journalism, where the most important information is delivered first.

Text planning also addresses the relationships between data points. For instance, in a medical report, the system might first present symptoms, followed by potential causes, and finally recommend treatments. This logical sequence enhances clarity and aligns with the reader's expectations.

6.3.5

Which of the following are true about the text planning phase in NLG?

- Organizes information into a coherent structure.
- Considers rhetorical and discourse goals.
- Focuses on grammatical accuracy of sentences.
- Prioritizes which data points to include.
- Generates the final text output.

6.3.6

Sentence planning

Sentence planning involves determining the specific words, phrases, and grammatical structures to use when expressing the information. This phase transforms abstract ideas into precise linguistic expressions, ensuring clarity and

appropriateness for the target audience. The choice of vocabulary and sentence style depends on factors like the text's purpose and the user's preferences.

For example, a customer service chatbot might generate the phrase, "Your order will arrive tomorrow," rather than a more formal alternative like, "The delivery of your order is scheduled for the subsequent day." Sentence planning ensures that the tone aligns with the context and user expectations.

This phase also includes referring expression generation, which decides how to refer to entities within the text. For instance, after introducing "Barack Obama" in a paragraph, subsequent references might use "Obama," "he," or "the former president," depending on the level of formality and clarity required.

6.3.7

Which of the following are true about the sentence planning phase in NLG?

- Chooses words and phrases to express information.
- Adjusts tone and style to suit the context.
- Decides what data to include in the text.
- Focuses on organizing data logically.
- Ensures the final text is grammatically correct.

6.3.8

Surface realisation

Surface realisation is the final phase of NLG, where the planned sentences are converted into fully formed text. This phase ensures grammatical accuracy, fluency, and naturalness in the generated text. It focuses on turning abstract representations into polished, human-readable sentences.

For example, after sentence planning determines the content and phrasing, surface realisation ensures proper syntax, punctuation, and formatting. The system might generate the text: "The weather tomorrow will be sunny with a high of 75°F," ensuring it adheres to grammatical rules and reads fluently.

Surface realisation often uses language models or templates to generate text that mimics natural language. For instance, chatbots rely on this phase to produce conversational and grammatically sound responses, such as, "I'm here to help. What do you need assistance with?"

6.3.9

Which of the following are true about the surface realisation phase in NLG?

- Ensures the text is grammatically correct.
- Converts planned sentences into polished text.

- Selects which information to include in the text.
- Focuses on organizing information logically.
- Prioritizes data points for inclusion in the text.

6.4 NLG projects

6.4.1

Project: Automated report generator for business data

Develop a system that generates detailed business reports from structured data (e.g., sales performance, customer feedback, financial metrics). The system should produce clear, concise, and grammatically correct text summarizing key insights, trends, and recommendations.

- Dataset: https://priscilla.fitped.eu/data/nlp/sales_data.csv

1. Libraries preparation

Use libraries like pandas to load and preprocess data from sources such as CSV files or databases.

```
import pandas as pd
import numpy as np
```

2. Data processing and analysis

Let's load the data and analyze the sales performance and customer satisfaction.

```
# Load the data
data =
pd.read_csv('https://priscilla.fitped.eu/data/nlp/sales_data.c
sv')

# Summarize sales performance
total_sales = data['Sales'].sum()
average_sales = data['Sales'].mean()
sales_trend = data.groupby('Product')['Sales'].sum()

# Summarize customer satisfaction
avg_satisfaction = data['Customer_Satisfaction'].mean()

# Detecting trends or anomalies (example: sales spikes or
drops)
sales_anomaly = data['Sales'].diff().max()  # Max difference
between consecutive sales
```

3. Text planning and sentence generation

Now, we'll use a simple sentence generation method to describe these insights. For the sake of simplicity, we will manually generate sentences. You can later integrate NLG libraries like SimpleNLG or GPT models for more dynamic sentence generation.

```
# Function to generate the report text
def generate_report(total_sales, avg_sales, sales_trend,
avg_satisfaction, sales_anomaly):
    report = []

    # General summary
    report.append(f"Total sales for the period:
${total_sales:.2f}")
    report.append(f"Average sales per day: ${avg_sales:.2f}")

    # Sales performance by product
    for product, sales in sales_trend.items():
        report.append(f"Total sales for {product}:
${sales:.2f}")

    # Customer satisfaction
    report.append(f"Average customer satisfaction:
{avg_satisfaction:.2f} out of 5")

    # Anomalies or trends
    if sales_anomaly > 1000:  # Arbitrary threshold for
detecting large anomalies
        report.append(f"A significant sales change was
detected: ${sales_anomaly:.2f} increase in one day.")

    return '\n'.join(report)
```

4. Report generation

```
# Generate the report text
report_text = generate_report(total_sales, average_sales,
sales_trend, avg_satisfaction, sales_anomaly)

# Output the report as plain text
print(report_text)

# Optional: Save the report to a text file
with open("business_report.txt", "w") as file:
```

```
    file.write(report_text)
```

Program output:

```
Total sales for the period: $900262.00
Average sales per day: $833.58
Total sales for Product A: $493752.00
Total sales for Product B: $406510.00
Average customer satisfaction: 3.51 out of 5
A significant sales change was detected: $1325.00 increase in
one day.
```

6.4.2

Project: Text generator

Another way to approach text generation, especially in Python, is to use transformer-based models from popular deep learning frameworks such as Hugging Face's Transformers. These models, like GPT-3, BERT, and T5, are highly effective at generating coherent and contextually appropriate text based on the input prompt.

Steps to implement Text generation with Hugging Face

1. Install required libraries

First, install the **Transformers** library along with **PyTorch** (or TensorFlow, depending on your preference). In our system is currently installed.

```
!pip install transformers torch
```

2. **Choose a pre-trained model**

- You can use any pre-trained model from Hugging Face's Model Hub, such as GPT-2, T5, or GPT-3 (via the OpenAI API). For this example, let's use GPT-2, where you do not need API key.

```
from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Load pre-trained model and tokenizer
model_name = "gpt2"
model = GPT2LMHeadModel.from_pretrained(model_name)
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
```

3. Write python code

- Here’s an example of using GPT-2 for text generation in Python, the description is directly in code.

```
# Set pad_token_id to eos_token_id to avoid the warning.
# This ensures the tokenizer uses the <|endoftext|> token
(eos_token) for padding,
# which is compatible with GPT-2 as it does not have a native
pad token.
tokenizer.pad_token = tokenizer.eos_token

# Encode the input prompt into token IDs.
# `input_text` is converted into a sequence of integers (token
IDs) that the model understands.
# `return_tensors="pt"` ensures the output is a PyTorch
tensor, required by the model.
input_text = "Say me how are you?"
input_ids = tokenizer.encode(input_text, return_tensors="pt")

# Generate text based on the input tokens.
# `input_ids`: The encoded input prompt used as a seed for
generation.
# `max_length=150`: The maximum length of the generated text
(including input tokens).
# `num_return_sequences=1`: Specifies generating only one
sequence.
# `no_repeat_ngram_size=2`: Prevents repeating n-grams (word
sequences of size 2).
# `temperature=0.7`: Controls the randomness of generation
(lower = more deterministic).
# `pad_token_id=tokenizer.pad_token_id`: Specifies the padding
token explicitly to avoid warnings.
output = model.generate(
    input_ids,
    max_length=150,
    num_return_sequences=1,
    no_repeat_ngram_size=2,
    temperature=0.7,
    pad_token_id=tokenizer.pad_token_id  # Set pad_token_id
explicitly
)

# Decode the generated token IDs into human-readable text.
# `tok` represents one generated sequence in `output`.
# `skip_special_tokens=True`: Removes tokens like
<|endoftext|> from the output.
for tok in output:
```

```
    generated_text = tokenizer.decode(tok,
skip_special_tokens=True)

    # Print the generated text to see the result.
    print(generated_text)
```

Program output:

```
Say me how are you? I'm not a doctor, I don't know what you're
talking about. I just want to know how you feel. You're not
going to tell me what to do.

"I'm sorry, but I can't do anything about it. It's not my
fault. If you want me to, you can. But I want you to be happy.
And I know you don' want that. So I'll just leave it at that."
...
```

For the given input "Say me how are you?", here's what happens at each step:

- Set padding token - the model uses **<|endoftext|>** for padding.
- Encode input "Say me how are you?" → [6473, 502, 783, 389, 345] (example token IDs).
- Generate output produces a sequence of token IDs based on the input prompt.
- Decode output converts the generated token IDs into readable text (e.g., "Say me how are you? I am doing well. How about you?").

6.4.3

Project: Financial data analysis and reporting using GPT-2

Use GPT-2 for generating a financial summary based on historical financial data. The goal is to understand how to integrate GPT-2 into a workflow to produce human-like financial text and to develop proficiency in working with transformers and preprocessing data.

1. Dataset Preparation

- **Download data** provide a CSV file containing sample data (e.g., quarterly revenue, profit, expenses, and growth rate, or linked file).
- **Understand the structure** you should explore the dataset using Python and libraries like **pandas** to understand its features.
- Dataset: https://priscilla.fitped.eu/data/nlp/sales_data.csv

```
# write your code
```

2. Generate structured summaries programmatically

- Clean and prepare the dataset for textual input.
- Extract meaningful patterns or insights (e.g., calculate percentage growth or compare revenues across quarters).

Example Task:

- Write a Python function to generate sentences like: **"In Q3, the company reported a revenue of $500M, a 10% increase compared to Q2 2022."**

```
# write your code
```

3. Integrate GPT-2 for enhancing the summaries.

- If fine-tuning is feasible, provide a set of financial summaries to fine-tune GPT-2 for generating domain-specific text.
- Otherwise, directly use GPT-2 for generation.

```
# write your code
```

4. Model integration

- Input Prompt Design: Craft prompts to guide GPT-2. Example: "Summarize the following financial data: Revenue: $500M, Growth: 10%, Profit: $50M."
- Generate Text: Use the GPT-2 model to generate summaries.

```
# write your code
```

5. Output evaluation

- Compare GPT-2's generated summary with a manually written summary.
- Evaluate based on clarity, accuracy, and fluency.

```
# write your code
```

Data Sources

Chapter 7

7.1 Structured and unstructured data

📖 7.1.1

Understanding the type of data we work with is crucial in NLP. Data can be broadly classified into **structured** and **unstructured** forms. **Structured data** is neatly organized and fits into rows and columns, such as spreadsheets or databases. For example, a table listing students' names, grades, and attendance is structured data because it follows a clear, predefined format. **Unstructured data**, on the other hand, doesn’t have a clear format. Examples include paragraphs of text, social media posts, images, and audio recordings.

Unstructured data is far more common in real-world scenarios. For instance, news articles, customer reviews, and chatbot conversations are all unstructured. While structured data can be directly analyzed using standard tools, unstructured data requires **specialized techniques** to make sense of it. NLP focuses on this challenge - turning raw, unstructured text into useful insights.

Why does unstructured data require special attention? Imagine receiving hundreds of emails daily. Without NLP tools, analyzing trends in the messages would be overwhelming. Techniques like tokenization, text cleaning, and linguistic parsing help convert unstructured data into a format that computers can process. This transformation is what makes advanced applications like chatbots and sentiment analysis possible.

📝 7.1.2

Which of the following is an example of structured data?

- A table of student grades in a database.
- A collection of tweets from social media.
- A list of photos on your phone.
- A conversation between two friends.

📝 7.1.3

Why is unstructured data challenging for NLP?

- It lacks a predefined format.
- It is stored in rows and columns.
- It is too small for NLP models.
- It can only be analyzed by humans.

📖 7.1.4

Before NLP techniques can be applied, we first need to **collect and store data** effectively. Sources of NLP data include **web scraping**, where information is

extracted from websites, and **APIs**, which allow programs to retrieve data from services like Twitter or Google. For example, a chatbot analyzing customer sentiment might rely on product reviews collected from e-commerce websites.

Once collected, the data needs to be stored in a way that allows efficient processing. Databases, cloud storage, and file systems are commonly used for structured data. For unstructured data, storage formats such as plain text files (.txt) or JSON files are more appropriate. For example, a dataset containing social media comments might be stored as a JSON file, with each comment as a separate entry.

It's essential to ensure the data is organized and labeled clearly. Consider a dataset for training an NLP model to classify news articles. If the articles are labeled as "sports," "technology," or "politics," the system can learn to recognize these categories. Without proper organization, the training process becomes chaotic and less effective.

7.1.5

Which source is commonly used to collect data for NLP?

- Web scraping
- APIs
- Collecting physical notebooks
- Random guesses files from the system

7.1.6

Why is it important to label data during storage?

- To ensure the model learns specific patterns.
- To make training more effective.
- To reduce the storage size.
- To randomly classify the data.

7.2 XPath

7.2.1

The Internet is currently the primary source of textual information, making it essential to preprocess text for any analysis. Text preprocessing involves **loading the text** from a source and applying **simple modifications** to prepare it for further processing.

One commonly used tool for this purpose is the **Requests library** for Python, widely praised for its simplicity and ease of use. The developers describe it as "HTTP for Humans™," emphasizing its user-friendly design.

The Requests library simplifies the process of sending **HTTP requests** to servers. It offers various functions that allow users to retrieve and interact with web content effectively. For example, it enables you to fetch a webpage's content and handle the server's responses. This functionality is particularly useful when working with data from the web.

In this case, the **get() function** from the Requests library is especially important. This function allows you to send an HTTP request to a webpage, effectively "downloading" its content and storing it in a variable. Once the data is retrieved, you can use the **.content** attribute to view the raw HTML code of the webpage. This is the first step in many text analysis workflows, as it allows us to access and process online textual data efficiently.

For example, you could use **get()** to fetch the HTML content of a news article, save it as a variable, and then preprocess it to remove unnecessary tags or extract the main text for analysis.

```
import requests

link = "https://www.google.sk"
page = requests.get(link)
```

```
print(page.content[:500])
```

Program output:

```
b'
```

7.2.2

When working with web pages, even simple ones (like the **Google homepage**), the raw HTML code retrieved is often quite large and difficult to navigate. To make sense of this complexity, the Requests library is typically paired with other tools designed to process and extract information from HTML. One such tool is the **LXML library**, a Python library built specifically for processing HTML and XML documents.

With LXML, we can execute **XPath queries** to locate and extract specific elements or data points within the HTML code. XPath is a **W3C standard technology** designed to navigate and manipulate the structure of XML or HTML documents. If you're familiar with XML processing, you might already know XPath, but here's a quick refresher.

What is XPath?

XPath provides a way to define a "path" to elements or attributes in XML or HTML documents. It's the foundational technology behind tools like XSLT and XQuery and is widely used for data selection and navigation. Think of XPath as similar to

navigating a file system: it uses **forward slashes** (/) to specify paths, just as you would move through folders in a directory structure. For example:

- /html/body/h1 selects the <h1> element inside the <body> tag.
- //div[@class='example'] selects all elements with a class attribute of "example."

This ability to specify precise paths makes XPath invaluable for extracting data from complex HTML documents. For instance, you can extract the main content of an article, specific table rows, or metadata like titles and descriptions.

By combining the **Requests library** to fetch HTML and the **LXML library** to process it with XPath queries, you can effectively extract structured data from unstructured web pages. For more details on XPath, you can refer to online tutorials like “XPath Tutorial.”

7.2.3

We will demonstrate the use of the **Request** and **LXML** libraries on a simple example, where we load the content of the element <title> from the www.ukf.sk page.

```
#import necessary libraries
import requests
from lxml import html

#load the homepage of UKF
link = "https://www.ukf.sk"
page = requests.get(link)

#load the page content as a DOM
tree = html.fromstring(page.content)

#create and execute an XPath expression
#XPath expression //title ensures that all
```

7.2.4

Task:

Using the **Request** and **LXML** libraries, load and display all menu items from the https://www.ukf.sk/en home page, if you know that each menu item is located in a element, this element is part of the <a> element, which is located in the element <li>.

```
#import necessary libraries
import requests
```

```
from lxml import html

#load the homepage of UKF
link = "https://www.ukf.sk/en"
page = requests.get(link)

#load the page content as a DOM
tree = html.fromstring(page.content)

#create and execute an XPath expression
result = tree.xpath("//li/a/span/text()")
print(result[:10])
```

Program output:

```
['University', 'History', 'Academic Insignia and Symbols',
'Former Leading Representatives ', 'Honorary Doctors',
'University Bodies', 'Academic Senate', 'Rector', 'Scientific
Council', 'Administrative Board']
```

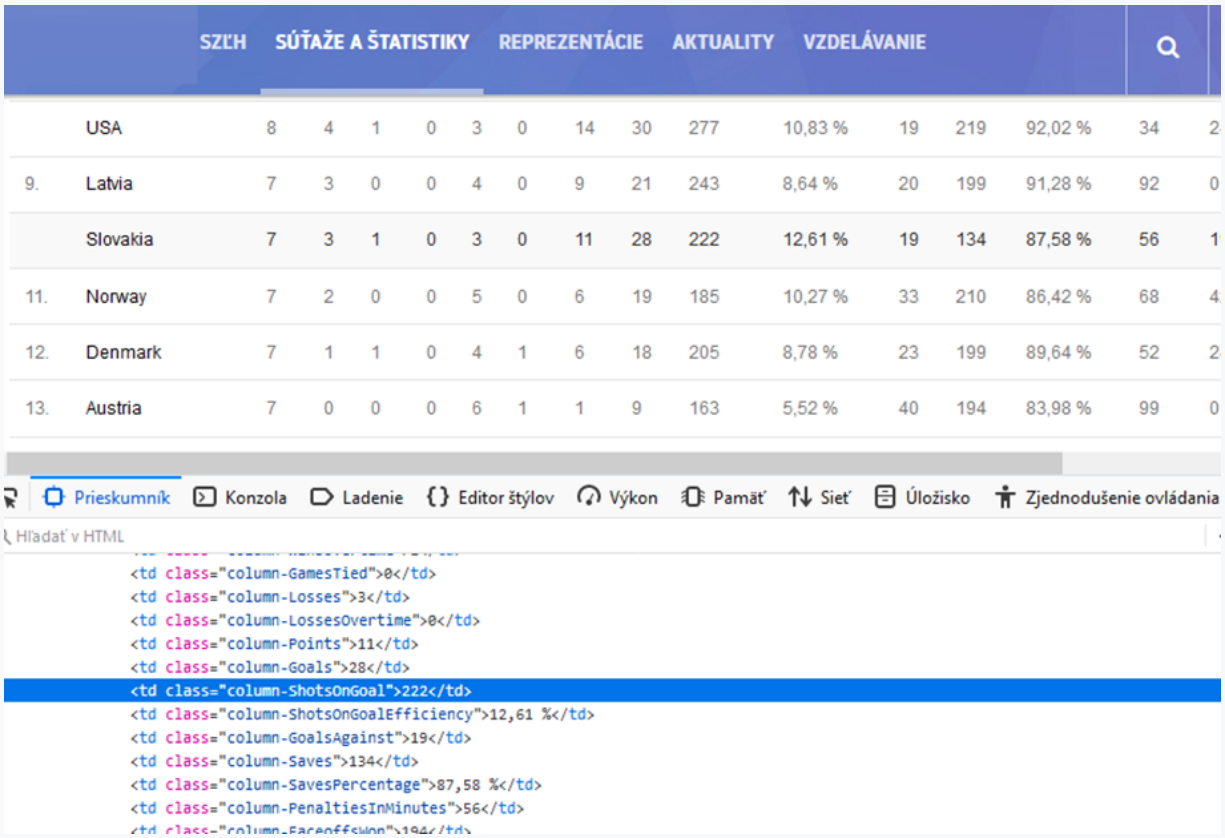
The combination of the Request and LXML libraries offers us a powerful tool for browsing the web, searching for information and their subsequent loading it in Python. Using these two tools, it is practically possible to load data from any page and further to use this data for own needs in the Jupyter Notebook environment. However, we must not forget that a properly defined XPath expression is a condition for the proper functioning of such scripts. We can create it by examining the HTML source code of the page being loaded.

7.2.5

Project: World Cup of Hockey

On the page https://www.hockeyslovakia.sk/sk/stats/competitors/697/ms-2019-slovensko, you will find complete team statistics from the 2019 World Cup of Hockey. Use the Request and LXML libraries to load and display the number of shots on goal, display the basic statistics (minimum, maximum, average).

If we visit the page https://www.hockeyslovakia.sk/sk/stats/competitors/697/ms-2019-slovensko, after viewing the source code, we will find that the data concerning the number of shots on goal is part of the HMTL code: **<td class="column-ShotsOnGoal">222</td>**



We create an XPath query to load the value of this element: **"//td[@class='column-ShotsOnGoal']/text()",** which finds all **<td>** elements with the class attribute (attribute name is preceded by the @ sign in the XPath) set to the value **'column-ShotsOnGoal'**. Using the **text()** function, we load the content from this element.

```
#import necessary libraries
import requests
from lxml import html

#load the homepage of UKF
link =
"https://www.hockeyslovakia.sk/sk/stats/competitors/697/ms-
2019-slovensko"
page = requests.get(link)

#load the page content as a DOM
tree = html.fromstring(page.content)

#create and execute an XPath expression
result = tree.xpath("//td[@class='column-
ShotsOnGoal']/text()")
print(result)
```

Program output:

```
['339', '374', '386', '337', '197', '289', '304', '277',
'243', '222', '185', '205', '163', '177', '120', '133']
```

The **result** variable currently contains a list of the number of shots on goal by individual teams. We can continue to work with this variable (list). For example, to find the required minimum, maximum, and average value. Currently, the result variable contains text values. Therefore, the first step will consist of their quick conversion to an integer.

```
shoots = [int(i) for i in result]
print(shoots)
```

Program output:

```
[339, 374, 386, 337, 197, 289, 304, 277, 243, 222, 185, 205,
163, 177, 120, 133]
```

Subsequently, we can load the list using the Pandas library, which has implemented basic functions for calculating the minimum, maximum, and average.

```
import pandas
df = pandas.DataFrame(shoots,columns = ['shots_on_the_goal'])
print(df.head())
```

Program output:

```
   shots_on_the_goal
0                339
1                374
2                386
3                337
4                197
```

```
print(df['shots_on_the_goal'].mean())
```

Program output:

```
246.9375
```

```
print(df['shots_on_the_goal'].max())
```

Program output:

```
386
```

```
print(df['shots_on_the_goal'].min())
```

Program output:

```
120
```

It is obvious that we can work with data from any page in a similar way. We know e.g. find out the minimum, maximum average price of a new phone or scooter (or any product) in the selected e-shop, provided that the correct XPath expression is formulated.

Preprocessing

Chapter 8

8.1 Preprocessing methods

📖 8.1.1

NLP is a branch of AI that focuses on enabling computers to understand, interpret, and respond to human language. NLP powers applications such as chatbots, virtual assistants, search engines, and sentiment analysis systems. To perform these tasks, NLP models analyze relationships within the text, such as between words, sentences, and paragraphs. However, raw text data is often messy and inconsistent, making it challenging for these models to process directly.

This is where **text preprocessing** becomes essential. Preprocessing transforms unstructured, noisy text into a clean and structured format that NLP models can interpret effectively. For instance, text on a webpage may include irrelevant content, such as HTML tags or excessive punctuation, which can confuse the model. Preprocessing ensures that only the meaningful components of the text are retained, improving model performance.

Some common preprocessing tasks include:

1. **Removing stop words** like "the," "is," and "and" are frequent but carry little meaning on their own. Removing them reduces noise.
2. **Tokenization** is splitting the text into smaller units, such as words or sentences, allows models to analyze language at granular levels.
3. **Stemming and lemmatization** reduce words to their base or root forms. For example, "running," "runs," and "ran" are reduced to "run," enabling the model to treat them as the same concept.

Preprocessing not only improves the efficiency of NLP systems but also standardizes text input, making it easier for models to draw meaningful insights. Without these steps, the models would struggle with inconsistencies, leading to lower accuracy.

📝 8.1.2

Which of the following are common tasks in text preprocessing?

- Removing stop words like "the" and "and."
- Tokenizing text into smaller units like words or sentences.
- Ignoring text formatting and punctuation entirely.
- Avoiding any modifications to the raw text data.

📖 8.1.3

Raw text data often contains inconsistencies, irrelevant details, and variations that can hinder the performance of NLP models. Preprocessing transforms this raw text into a clean and structured format that is easier for algorithms to analyze. The key

objectives of preprocessing include removing noise, standardizing the text, and preparing it for tokenization and feature extraction.

Steps in preprocessing typically include:

- **text cleaning - r**emove unwanted characters, numbers, or punctuation that do not contribute to understanding the text.
- **lowercasing** - convert all text to lowercase to ensure uniformity.

By cleaning and standardizing the text, we can eliminate unnecessary variations and make the data consistent for analysis.

📝 8.1.4

Why is text preprocessing important in NLP?

- Ensures text is consistent
- Reduces text size to save memory
- Improves the performance of models
- Removes all irrelevant words

🕮 8.1.5

Tokenization

Tokenization is the process of breaking text into smaller units, known as tokens. Tokens can be words, characters, or subwords, depending on the application's requirements. Tokenization helps convert unstructured text into a structured format that NLP models can process.

For example, the sentence *"NLP is fun!"* can be tokenized into individual words:

- ["NLP", "is", "fun"]

Advanced NLP models, like BERT, often use **subword tokenization** to handle unknown words. For instance, the word *"playing"* might be tokenized into ["play", "##ing"].

📝 8.1.6

What is the main purpose of tokenization in NLP?

- To split text into structured units
- To break text into sentences
- To divide text into tokens like words or subwords
- To identify grammatical errors

📖 8.1.7

Stopword removal

Stopwords are common words that do not carry significant meaning in the context of text analysis. Examples of stopwords include "the," "is," "and," and "in." Removing stopwords can reduce noise in the data and improve the focus on more meaningful words.

However, the decision to remove stopwords depends on the task. In sentiment analysis, for example, stopwords may carry important context.

Python libraries like **NLTK** or **spaCy** provide built-in lists of stopwords that can be filtered out during preprocessing.

📝 8.1.8

Why might removing stopwords improve NLP model performance?

- It reduces irrelevant data
- It removes all meaningful context
- It eliminates common words that carry little information
- It shortens text for tokenization

📖 8.1.9

Lemmatization and stemming

Lemmatization and stemming are methods used to reduce words to their base or root forms:

- **Stemming** cuts off prefixes or suffixes, often resulting in shortened, sometimes incomplete forms (e.g., "running" → "run").
- **Lemmatization** converts words to their dictionary base form, considering their context and part of speech (e.g., "running" → "run," "better" → "good").

Lemmatization is generally more accurate but computationally intensive, while stemming is faster but less precise.

📝 8.1.10

What is the difference between lemmatization and stemming?

- Stemming is less accurate but faster
- Lemmatization considers context and part of speech
- Stemming always produces dictionary forms
- Lemmatization is faster than stemming

8.2 Manual examples

8.2.1

Identify which words are part of the tokenized result.

"Artificial Intelligence is transforming industries around the world."

- Artificial
- Intelligence
- transforming
- AI
- transform
- the
- world
- industry

8.2.2

Tokenize following text into words ordered by alphabet.

"Artificial Intelligence is transforming industries around the world."

_____ _____ _____ _____ _____ _____ _____ _____

- industry
- AI
- Artificial
- transforming
- around
- is
- transforms
- industries
- change
- world
- the
- Intelligence

8.2.3

Identify which words remain after stopword removal (use original sentence order).

"The quick brown fox jumps over the lazy dog."

_____, _____, _____, _____, _____, _____

- the

- brown
- fox
- lazy
- The
- dog
- jumps
- quick
- over

8.2.4

Identify stopwords used by a standard English stopword list.

- The
- over
- the
- quick
- brown
- fox
- jumps

8.2.5

Apply lemmatization to reduce each word to its base form.

"Cats are running quickly to chase the mice."

Result:

```
"_____", "_____", "_____", "_____", "_____", "_____"
```

- running
- chase
- mouse
- be
- quickly
- is
- to
- cats
- run
- the
- cat
- are
- quick
- mice

8.2.6

Identify which words are part of the **stemmed** result:

"Working on stemming processes helps in text preprocessing."

- work
- on
- stem
- process
- help
- in
- text
- preprocess
- working
- helps
- the
- stemmed

8.3 Basic operations

8.3.1

Tokenization is the process of breaking down a stream of text into smaller units called tokens, such as words, phrases, or sentences. These tokens serve as the building blocks for many NLP tasks. Tokenization helps transform raw text into a structured format that is easier for models to analyze and process. For instance, a sentence like "I love NLP!" might be tokenized into ["I", "love", "NLP", "!"]. This step is crucial for enabling text analysis, such as sentiment detection, text classification, and translation.

One of the initial steps in data preparation is tokenization at the sentence level. The sent_tokenize() function from the Natural Language Toolkit (nltk) library in Python is used to split a given text into individual sentences. It is a part of the nltk.tokenize module and relies on pre-trained models to identify sentence boundaries effectively.

The function can handle **language-specific nuances** in sentence boundary detection, such as abbreviations (e.g., "Dr.", "etc.") or unusual punctuation. It uses periods, exclamation marks, and question marks to identify sentence boundaries, ensuring accurate splits.

- The input is a block of text containing one or more sentences.
- The function applies a pre-trained tokenizer model that identifies sentence boundaries based on punctuation, spacing, and linguistic rules.
- Output is a list of sentences, each as a string.

```
from nltk.tokenize import sent_tokenize

# Example text
text = "Natural Language Processing is exciting! It involves
various tasks, like tokenization. Isn't that amazing?"

# Tokenizing into sentences
sentences = sent_tokenize(text)

# Output the result
print(sentences)
```

Program output:

```
['Natural Language Processing is exciting!', 'It involves
various tasks, like tokenization.', "Isn't that amazing?"]
```

Why use **sent_tokenize()**?

- Many NLP models require input at the sentence level. Breaking down paragraphs into sentences is a common first step.
- Summarization and sentiment analysis often operate on sentences to identify key points or sentiments.
- Sentences are easier to analyze than large, unstructured blocks of text.

Limitations of **send_tokenize()** are:

- Dependency on language models - the accuracy of sent_tokenize() depends on the pre-trained model's quality and the specific language or dialect.
- Ambiguities in punctuation might struggle with complex cases, such as quotes or nested punctuation.

8.3.2

Project: Tokenization in different languages

- This example will focus on the use of the **sent_tokenize()** function, which creates tokens in the form of sentences from coherent text.

```
from nltk.tokenize import sent_tokenize

# assigning the text of the poem to the variable poem in one
string
poem = "He who carries a pure heart does not need much to be
happy. He easily takes off from the morning dew on lame wings
to heaven. Little children of God. Wandering stars. Lilies
```

```
follow them. And God knows it. And He will not forget. Neither
will we."

# tokenization into sentences
sentences = sent_tokenize(poem)
print(sentences)
```

Program output:

```
['He who carries a pure heart does not need much to be
happy.', 'He easily takes off from the morning dew on lame
wings to heaven.', 'Little children of God.', 'Wandering
stars.', 'Lilies follow them.', 'And God knows it.', 'And He
will not forget.', 'Neither will we.']
```

```
# the output of the function can also be assigned to a
variable
verse = sent_tokenize(poem)

# then it is possible to access individual sentences as
elements of a list
print(verse[0])
```

Program output:

```
He who carries a pure heart does not need much to be happy.
```

The **sent_tokenize()** function is an instance of **PunktSentenceTokenizer**. This instance is trained and works very well for most major world languages. We can improve its operation by setting the language in which we will do the tokenization. The next example shows how to improve tokenization by setting the language (English, Spanish) which is trained on.

```
import nltk.data
nltk.download('punkt')

# setting the tokenizer trained on English
en_tokenizer =
nltk.data.load('tokenizers/punkt/PY3/english.pickle')

en_bible_text = "In the beginning was the Word, and the Word
was with God, and the Word was God."

# tokenization of English text
en_bible = en_tokenizer.tokenize(en_bible_text)

print(en_bible)
```

Program output:

```
['In the beginning was the Word, and the Word was with God,
and the Word was God.']
[nltk_data] Downloading package punkt to
/home/johny/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

```
# setting the tokenizer trained on Spanish
sp_tokenizer =
nltk.data.load('tokenizers/punkt/PY3/spanish.pickle')

sp_bible_text = "En el principio era el Verbo, y el Verbo con
Dios, y el Verbo era Dios. Este era en el principio con Dios.
Todas las cosas por él fueron hechas, y sin él nada de lo que
ha sido hecho, fue hecho. En él estaba la vida, y la vida era
la luz de los hombres. La luz en las tinieblas resplandece, y
las tinieblas no prevalecieron contra ella."

# tokenization of Spanish text
sp_bible = sp_tokenizer.tokenize(sp_bible_text)

print(sp_bible)
```

Program output:

```
['En el principio era el Verbo, y el Verbo con Dios, y el
Verbo era Dios.', 'Este era en el principio con Dios.', 'Todas
las cosas por él fueron hechas, y sin él nada de lo que ha
sido hecho, fue hecho.', 'En él estaba la vida, y la vida era
la luz de los hombres.', 'La luz en las tinieblas resplandece,
y las tinieblas no prevalecieron contra ella.']
```

8.3.3

Sentence tokenization

The second and, in our opinion, more frequently used tokenization is the sentence tokenization into individual words, and/or tokens. From the point of view of text processing, individual sentences consist of tokens and spaces. Therefore, a token is any string of characters that usually must be between two spaces (whitespace). Sentence tokenization itself is not a simple process. During this process, it is also necessary to deal with the following issues:

- A decimal point (e.g. number 2.06). For the tokenizer, the point or comma character represents the point at which the token can be separated.

However, by numbers, it must decide whether it will be suitable to divide the number.
- The second issue with numbers is their (for better readability) "traditional" presentation with spaces (e.g. 3 113 416 123). In this case, the tokenizer must not treat spaces as token separators.
- Mastering various special characters, upper and lower case letters, subscript and superscript, colon, full stop, question mark, exclamation mark, quotation marks, asterisks, mathematical symbols and others.
- The issue of the so-called linking words (e.g.: within, on black, and so on, Nitriansky region, Nitriansky self-governing region, etc.), where it is more appropriate to understand the linking words as a whole and not to divide it into several tokens.

For sentence tokenization, it is suitable to use the **word_tokenize()** function, which, as we can see in the following example, also takes into account punctuation marks, which it also works with as tokens.

```
from nltk.tokenize import word_tokenize

first_sentence = "In the beginning was the Word, and the Word
was with God, and the Word was God."

print(word_tokenize(first_sentence))
```

Program output:

```
['In', 'the', 'beginning', 'was', 'the', 'Word', ',', 'and',
'the', 'Word', 'was', 'with', 'God', ',', 'and', 'the',
'Word', 'was', 'God', '.']
```

In the following text, we will not discuss all tokenization options. The functions for tokenization with the option to define own regular expressions, or to train own tokenizer which takes into account the special characteristics of the analysed text, are particularly interesting. More detailed information about these procedures can be found in the publication (Perkins, 2014).

8.3.4

Stopwords

Words in the analysed text that do not have much meaning from the point of view of semantics (especially prepositions, conjunctions and function words) are denoted as stopwords. Removing them will improve indexing, text analysis and reduce data size. Stopwords represent 20 to 30% of all words in a document. In the following text, we present a list of stopwords for the Slovak and English language:

- Slovak: a, aj, aby, ale, ako, áno, alebo, ani, asi, byť, bez, by, či, cez, do, čo, ešte, dnes, iba, ďalší, je, ho, i, jej, ja, jeho, každý, k, kam, ktorý, kde, mať, kto, môj, ku,

na, môcť, my, niet, nad, nie, nový, než, nič, po, o, od, on, prečo, pod, pred, podľa, práve, potom, preto, prvý, pri, s, so, sa, si, svoj, späť, tak, ten, takže, tuto, teda, tento, to, už, toto, z, tu, tvoj, ty, u, v, že váš, viac, však, všetko, vy, za, že ...

- English: [and, also, to, but, as, yes, or, even, probably, be, without, by, whether, through, to, what, still, today, only, next, is, him, her, me, his, every, to, where, which, where, have, who, my, to, on, can, we, no, above, no, new, than, nothing, after, about, from, he, why, under, before, according to, just, then, therefore, first, at, with, with, with, you, your, back, so, that, so, this, thus, this, that, already, this, from, here, your, you, u, in, that your, more, however, all, you, for, that ...]

It is possible to use stopwords for many languages within the NLTK library. After importing stopwords, it is possible to display languages for which stopwords exist in NLTK.

```
from nltk.corpus import stopwords

# first, let's check the languages for which we have stop
words in the library
print(stopwords.fileids())
```

Program output:

```
['arabic', 'azerbaijani', 'basque', 'bengali', 'catalan',
'chinese', 'danish', 'dutch', 'english', 'finnish', 'french',
'german', 'greek', 'hebrew', 'hinglish', 'hungarian',
'indonesian', 'italian', 'kazakh', 'nepali', 'norwegian',
'portuguese', 'romanian', 'russian', 'slovene', 'spanish',
'swedish', 'tajik', 'turkish']
```

In the case of stopwords, it is actually only a text list. For Slovak, we have prepared a list consisting of Slovak stopwords at
https://priscilla.fitped.eu/data/nlp/stop_words_slovak.txt

```
import pandas as pd

# URL of the file
url =
"https://priscilla.fitped.eu/data/nlp/stop_words_slovak.txt"

# Read the file as a text file using pandas
data = pd.read_csv(url, header=None)  # Read as a raw text
file with no headers

# Convert each row into a list of lines
lines = data[0].tolist()
```

```
# Print the list of lines
print(lines)
```

Program output:

```
['a', 'aby', 'aj', 'ak', 'ako', 'ale', 'alebo', 'and', 'ani',
'áno', 'asi', 'až', 'bez', 'bude', 'budem', 'budeš', 'budeme',
'budete', 'budú', 'by', 'bol', 'bola', 'boli', 'bolo', 'byť',
'cez', 'čo', 'či', 'ďalší', 'ďalšia', 'ďalšie', 'dnes', 'do',
'ho', 'ešte', 'for', 'i', 'ja', 'je', 'jeho', 'jej', 'ich',
'iba', 'iné', 'iný', 'som', 'si', 'sme', 'sú', 'k', 'kam',
'každý', 'každá', 'každé', 'každí', 'kde', 'keď', 'kto',
'ktorá', 'ktoré', 'ktorou', 'ktorý', 'ktorí', 'ku', 'lebo',
'len', 'ma', 'mať', 'má', 'máte', 'medzi', 'mi', 'mna', 'mne',
'mnou', 'musieť', 'môcť', 'môj', 'môže', 'my', 'na', 'nad',
'nám', 'náš', 'naši', 'nie', 'nech', 'než', 'nič', 'niektorý',
'nové', 'nový', 'nová', 'nové', 'noví', 'o', 'od', 'odo',
'of', 'on', 'ona', 'ono', 'oni', 'ony', 'po', 'pod', 'podľa',
'pokiaľ', 'potom', 'práve', 'pre', 'prečo', 'preto',
'pretože', 'prvý', 'prvá', 'prvé', 'prví', 'pred', 'predo',
'pri', 'pýta', 's', 'sa', 'so', 'si', 'svoje', 'svoj',
'svojich', 'svojím', 'svojími', 'ta', 'tak', 'takže', 'táto',
'teda', 'te', 'ten', 'tento', 'the', 'tieto', 'tým', 'týmto',
'tiež', 'to', 'toto', 'toho', 'tohoto', 'tom', 'tomto',
'tomuto', 'toto', 'tu', 'tú', 'túto', 'tvoj', 'ty', 'tvojími',
'už', 'v', 'vám', 'váš', 'vaše', 'vo', 'viac', 'však',
'všetok', 'vy', 'z', 'za', 'zo', 'že']
```

8.3.5

Frequency

The basic operations of discovering knowledge from a text include isolating words and finding the frequency of individual words. In the NLTK library we can find a useful function that can work with created tokens and determine their individual quantities (numbers). It is the **FreqDist()** function. The input to the function is a list (mostly words/tokens). This function is used to find the words frequency in a text. As a return value, it returns the data in the so-called dictionary. To read the return values, we need to know the associative indexes.

We will show the use of the function on the example of recording actions (interventions/actions) of action heroes.

```
hero_actions_in_order = ['Superman', 'Iron_man',
'Cpt_America', 'Superman', 'Iron_man', 'Superman']
```

```
from nltk.probability import FreqDist
frequency = FreqDist(hero_actions_in_order)

print(frequency)
```

Program output:

```
```

The output can also be read using the so-called associative indexes.

```
print(frequency['Iron_man'])
print(frequency['Superman'])
```

Program output:

```
2
3
```

It is also possible to display the first *n* most frequently occurring words in the list.

```
print(frequency.most_common(2))
```

Program output:

```
[('Superman', 3), ('Iron_man', 2)]
```

Applying the **FreqDist()** function will be the content of the next practical task. Through a practical example, we will show how this function works and also the use of previous knowledge about the **Requests** library and creation of **XPath** expressions. In our next example, we will attempt to analyse Greek fables. We believe that the reader is familiar with some of them.

8.4 Feature extraction

🕮 8.4.1

For text analysis, we need a lot of textual content (volume), and probably no need to remind that a text like "Lorem ipsum dolor sit amet..." would probably not be suitable. Just for the sake of interest, the mentioned text "Lorem ipsum..." has been used since the 16th century. The text resembles ordinary Latin, but in reality it is a meaningless mutilation.

The problem with selecting text for analysis is that a meaningful amount of text needs to be selected. For educational purposes, it is advisable for students to know this text at least in part (for easy verification of results and the meaningfulness of the examples). From a legal point of view, it is advisable to choose a text for which the author has permission to publish or is not subject to copyright.

From the point of view of these criteria, texts from the Bible are probably the most frequently analysed texts in books and publications about text mining. From a book whose at least (approximate) content is probably known to most readers, even in the Arab world, authors of publications can analyse the Old Testament. This book is not subject to copyright, and all the authors of this book would certainly not object to the dissemination of these texts even without their consent.

In our next example, we will try to analyse texts that also meet the above criteria, Greek fables.

8.4.2

Project: Aesop's fables analyse

- In terms of the basic characteristics of the text, analyse Aesop's fables: “The Lion and the Mouse”, “The Wolf and the Lamb” and “The Kingdom of the Lion”.

May be a good resource for Aesop's Fables is https://www.sacred-texts.com/cla/aesop/index.htm. We can copy the texts from this page.

Among the basic characteristics of the text that can be obtained will be:

- number of words in fables,
- number of sentences,
- number of unique words,
- average number of words in sentences,
- finding the longest sentence and others.

We will analyse the English texts of Aesop's fables. After the initial import of the necessary libraries, we create three variables with the copied text.

```
import nltk
nltk.download('punkt')
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize

lion_and_mouse = "A lion was awakened from sleep by a Mouse
running over his face. Rising up angrily, he caught him and
was about to kill him, when the Mouse piteously entreated,
saying: 'If you would only spare my life, I would be sure to
repay your kindness.' The Lion laughed and let him go. It
happened shortly after this that the Lion was caught by some
hunters, who bound him by strong ropes to the ground. The
Mouse, recognizing his roar, came, gnawed the rope with his
teeth, and set him free, exclaiming, 'You ridiculed the idea
of my ever being able to help you, expecting to receive from
```

```
me any repayment of your favor; now you know that it is
possible for even a Mouse to confer benefits on a Lion.'"
wolf_and_lamb = "A Wolf, meeting with a Lamb astray from the
fold, resolved not to lay violent hands on him, but to find
some plea to justify to the Lamb the Wolf's right to eat him.
He thus addressed him: 'Sirrah, last year you grossly insulted
me.' 'Indeed,' bleated the Lamb in a mournful tone of voice,
'I was not then born.' Then said the Wolf, 'You feed in my
pasture.' 'No, good sir,' replied the Lamb, 'I have not yet
tasted grass.' Again said the Wolf, 'You drink of my well.'
'No,' exclaimed the Lamb, 'I never yet drank water, for as yet
my mother's milk is both food and drink to me.' Upon which the
Wolf seized him and ate him up, saying, 'Well! I won't remain
supperless, even though you refute every one of my
imputations. The tyrant will always find a pretext for his
tyranny.'"
lion_kingdom = "The beasts of the field and forest had a Lion
as their king. He was neither wrathful, cruel, nor tyrannical,
but just and gentle as a king could be. During his reign he
made a royal proclamation for a general assembly of all the
birds and beasts, and drew up conditions for a universal
league, in which the Wolf and the Lamb, the Panther and the
Kid, the Tiger and the Stag, the Dog and the Hare, should live
together in perfect peace and amity. The Hare said, 'Oh, how I
have longed to see this day, in which the weak shall take
their place with impunity by the side of the strong.' And
after the Hare said this, he ran for his life."
print(lion_kingdom)
```

Subsequently, according to already known commands, we will tokenize sentences and words in sentences.

```
sentences_fable1 = sent_tokenize(lion_and_mouse)
sentences_fable2 = sent_tokenize(wolf_and_lamb)
sentences_fable3 = sent_tokenize(lion_kingdom)

print(sentences_fable3)
```

Program output:

```
['The beasts of the field and forest had a Lion as their
king.', 'He was neither wrathful, cruel, nor tyrannical, but
just and gentle as a king could be.', 'During his reign he
made a royal proclamation for a general assembly of all the
birds and beasts, and drew up conditions for a universal
league, in which the Wolf and the Lamb, the Panther and the
```

```
Kid, the Tiger and the Stag, the Dog and the Hare, should live
together in perfect peace and amity.', "The Hare said, 'Oh,
how I have longed to see this day, in which the weak shall
take their place with impunity by the side of the strong.'",
'And after the Hare said this, he ran for his life.']
```

Functions **sent_tokenize()** and **word_tokenize()** give the result list. Whether it is a list of sentences or a list of words, it is always a list data type. This data type represents a collection (set of values) that is ordered and changeable. At the same time, the list allows duplication of its items (members). This means that a list can contain multiple identical items/members (equal values). In Python, lists are written in square brackets. Therefore, we create our own list simply by inserting the list elements (separated by commas) in square brackets.

8.4.3

List and Set

In our example, we will use two basic functions for working with the list:

- **Function len()** - the function counts the number of list items.

```
list_of_heroes = ['Hulk', 'Parker', 'Iron_man', 'Cpt_America',
'Superman']
print(len(list_of_heroes))

fibonacci_numbers = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
print(len(fibonacci_numbers))
```

Program output:

```
5
10
```

The function is interesting for its universality. The function calculates the number of elements in any object. For example if we insert a string into the function, i.e. object is a string, the **len()** function returns the number of characters (in string).

```
print(len('Hulk'))
```

Program output:

```
4
```

- **Function set()** - the function creates **a set** of objects. It is used to find the elements of a list without repetition. In the created set, the elements are no longer repeated, each one is represented exactly once.

```
hero_actions_in_order = ['Superman', 'Iron_man',
'Cpt_America', 'Superman', 'Iron_man']
heroes_in_actions = set(hero_actions_in_order)
print(heroes_in_actions)
```

Program output:

```
{'Superman', 'Cpt_America', 'Iron_man'}
```

8.4.4

We can conveniently use the mentioned functions to calculate the basic characteristics of the three Greek fables. The first option is to find how many sentences are in the fables.

```
print('The fable "The Lion and the Mouse" contains ' +
str(len(sentences_fable1)) + ' sentences.')
print('The fable "The Wolf and the Lamb" contains ' +
str(len(sentences_fable2)) + ' sentences.')
print('The fable "The Lion Kingdom" contains ' +
str(len(sentences_fable3)) + ' sentences.')
```

We directly wrote the result of the calculation into a text string with the **print()** function. For this reason, it was necessary to convert the result from a number (integer data type) to a text string (string data type). We performed the conversion with the **str()** function.

8.4.5

Project: Understanding content structure

Find out how many words (tokens) each fable contains and what is the average number of words per sentence.

To solve the problem, it is necessary to tokenize sentences in fables first. We perform tokenization with the well-known **word_tokenize()** function. We will use its result and the **len()** function to determine the number of words in fables. Please note that we perform tokenization directly for the whole fables´ text, not partially for individual sentences.

The result of **word_tokenize()** is a list of words/tokens in fables. For illustration, here is a list of the first few tokens for the fable The Lion and the Mouse.

```
import nltk
# nltk.download('punkt')
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize
```

```
lion_and_mouse = "A lion was awakened from sleep by a Mouse
running over his face. Rising up angrily, he caught him and
was about to kill him, when the Mouse piteously entreated,
saying: 'If you would only spare my life, I would be sure to
repay your kindness.' The Lion laughed and let him go. It
happened shortly after this that the Lion was caught by some
hunters, who bound him by strong ropes to the ground. The
Mouse, recognizing his roar, came, gnawed the rope with his
teeth, and set him free, exclaiming, 'You ridiculed the idea
of my ever being able to help you, expecting to receive from
me any repayment of your favor; now you know that it is
possible for even a Mouse to confer benefits on a Lion.'"
wolf_and_lamb = "A Wolf, meeting with a Lamb astray from the
fold, resolved not to lay violent hands on him, but to find
some plea to justify to the Lamb the Wolf's right to eat him.
He thus addressed him: 'Sirrah, last year you grossly insulted
me.' 'Indeed,' bleated the Lamb in a mournful tone of voice,
'I was not then born.' Then said the Wolf, 'You feed in my
pasture.' 'No, good sir,' replied the Lamb, 'I have not yet
tasted grass.' Again said the Wolf, 'You drink of my well.'
'No,' exclaimed the Lamb, 'I never yet drank water, for as yet
my mother's milk is both food and drink to me.' Upon which the
Wolf seized him and ate him up, saying, 'Well! I won't remain
supperless, even though you refute every one of my
imputations. The tyrant will always find a pretext for his
tyranny.'"
lion_kingdom = "The beasts of the field and forest had a Lion
as their king. He was neither wrathful, cruel, nor tyrannical,
but just and gentle as a king could be. During his reign he
made a royal proclamation for a general assembly of all the
birds and beasts, and drew up conditions for a universal
league, in which the Wolf and the Lamb, the Panther and the
Kid, the Tiger and the Stag, the Dog and the Hare, should live
together in perfect peace and amity. The Hare said, 'Oh, how I
have longed to see this day, in which the weak shall take
their place with impunity by the side of the strong.' And
after the Hare said this, he ran for his life."

words_fable1 = word_tokenize(lion_and_mouse)
words_fable2 = word_tokenize(wolf_and_lamb)
words_fable3 = word_tokenize(lion_kingdom)

print('The fable "The Lion and the Mouse" contains ' +
str(len(words_fable1)) + ' words.')
```

```
print('The fable "The Wolf and the Lamb" contains ' +
str(len(words_fable2)) + ' words.')
print('The fable "The Lion Kingdom" contains ' +
str(len(words_fable3)) + ' words.')

# print(words_fable1)
```

Program output:

```
The fable "The Lion and the Mouse" contains 152 words.
The fable "The Wolf and the Lamb" contains 191 words.
The fable "The Lion Kingdom" contains 143 words.
```

The solution to the second part of the task is to calculate the proportion of the number of words for each fable to the number of sentences. In this way, we will find out the average number of words per sentence for each fable.

```
sentences_fable1 = sent_tokenize(lion_and_mouse)
sentences_fable2 = sent_tokenize(wolf_and_lamb)
sentences_fable3 = sent_tokenize(lion_kingdom)

print('The average number of words (tokens) per sentence for
the fable "The Lion and the Mouse":')
print(len(words_fable1) / len(sentences_fable1))

print('The average number of words (tokens) per sentence for
the fable "The Wolf and the Lamb":')
print(len(words_fable2) / len(sentences_fable2))

print('The average number of words (tokens) per sentence for
the fable "The Lion Kingdom":')
print(len(words_fable3) / len(sentences_fable3))
```

Program output:

```
The average number of words (tokens) per sentence for the
fable "The Lion and the Mouse":
30.4
The average number of words (tokens) per sentence for the
fable "The Wolf and the Lamb":
19.1
The average number of words (tokens) per sentence for the
fable "The Lion Kingdom":
28.6
```

8.4.6

For the sake of completeness, we present the second method of calculation (only for “the Wolf and the Lamb” fable). We find the number of words/tokens for each sentence and calculate their averages. Naturally, the result is the same as in the first case.

```
import nltk
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize

wolf_and_lamb = "A Wolf, meeting with a Lamb astray from the
fold, resolved not to lay violent hands on him, but to find
some plea to justify to the Lamb the Wolf's right to eat him.
He thus addressed him: 'Sirrah, last year you grossly insulted
me.' 'Indeed,' bleated the Lamb in a mournful tone of voice,
'I was not then born.' Then said the Wolf, 'You feed in my
pasture.' 'No, good sir,' replied the Lamb, 'I have not yet
tasted grass.' Again said the Wolf, 'You drink of my well.'
'No,' exclaimed the Lamb, 'I never yet drank water, for as yet
my mother's milk is both food and drink to me.' Upon which the
Wolf seized him and ate him up, saying, 'Well! I won't remain
supperless, even though you refute every one of my
imputations. The tyrant will always find a pretext for his
tyranny.'"
sentences_fable2 = sent_tokenize(wolf_and_lamb)

# prepare a list to store the number of tokens in individual
sentences
token_counts = []

# for each sentence from the list of sentences in the fable
for sentence in sentences_fable2:
    # extract tokens from the sentence
    tokens = word_tokenize(sentence)
    # count the number of tokens
    token_count_in_sentence = len(tokens)
    # append the number of tokens (as a number) in the
sentence to the list token_counts
    token_counts.append(token_count_in_sentence)

# after counting the number of tokens for each sentence
# calculate the average
average = sum(token_counts) / len(token_counts)
print(average)
```

Program output:

```
19.1
```

8.4.7

Task: Find out the longest word used in a fable “The Lion and the Mouse”.

We will solve the task only using the well-known function **len()** which we can also use to find out the number of characters in a word.

```
import nltk
# nltk.download('punkt')
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize

lion_and_mouse = "A lion was awakened from sleep by a Mouse
running over his face. Rising up angrily, he caught him and
was about to kill him, when the Mouse piteously entreated,
saying: 'If you would only spare my life, I would be sure to
repay your kindness.' The Lion laughed and let him go. It
happened shortly after this that the Lion was caught by some
hunters, who bound him by strong ropes to the ground. The
Mouse, recognizing his roar, came, gnawed the rope with his
teeth, and set him free, exclaiming, 'You ridiculed the idea
of my ever being able to help you, expecting to receive from
me any repayment of your favor; now you know that it is
possible for even a Mouse to confer benefits on a Lion.'"

# Tokenization of the words in the fable "The Lion and the
Mouse"
words_fable1 = word_tokenize(lion_and_mouse)

# Initial variable settings
max_char_count = 0
# assuming there could be more words with the maximum length
# we create a list of words
max_words = []

# For all words
for word in words_fable1:
    # determine the length of the word
    length = len(word)
    # if it equals the maximum length
```

```
    if length == max_char_count:
        # add the word to the list
        max_words.append(word)
    # if it's greater than the maximum length
    if length > max_char_count:
        max_char_count = length
        # reset the list and start over
        max_words = []
        # add the word to the list
        max_words.append(word)
# Final output
print('The maximum number of characters in a word is ' +
str(max_char_count))
print('List of words with the maximum number of characters:')
print(max_words)
```

Program output:

```
The maximum number of characters in a word is 11
List of words with the maximum number of characters:
['recognizing']
```

8.5 Project

8.5.1

Project: Basic overview of Greek fables using methods of text analysis

Traveler Marek Polový, after several successful vacations on Senec and Duchonka Lakes, has decided to explore the Cyprus beaches this summer. In order to impress the locals with his knowledge of Greek culture, he decided to familiarize himself with Greek fables. However, he found that Greek fables (although they are mostly very short) are quite a lot. Therefore, we will attempt to make a basic overview of Greek fables using methods of text analysis.

For the analysis of Greek fables, we chose Aesop's fables. These are available, e.g. at https://www.sacred-texts.com/cla/aesop/index.htm. We will perform the analysis itself using the NLTK library and for the sake of simplicity (inflection and diacritics) we will analyse Aesop's fables in English.

In order to simplify the pre-processing of the input file, we will analyse the titles of Aesop's fables in the following text. We will be interested in which animal appears most often in Aesop's fables. Most of Aesop's fables have very apt titles, i.e. the title of the fable always includes its "main representatives", e.g. fable "Donkey, Fox, and Lion" or "Rabbit and Tortoise" etc.

To analyze all the titles of Aesop's Fables, in addition to the NLTK library, we will also need a library with functions for loading web content - Request. After loading the page content, i.e. the titles of Aesop's Fables, we tokenize the entire text, find the frequency of individual words, and display the first 20 most frequent tokens from the page. We start with the initial tokenization of the fable titles.

```
import requests
link = "https://www.sacred-texts.com/cla/aesop/index.htm"
f = requests.get(link)
fables = f.text

print(fables)
```

Based on the illustration, it is clear that we "downloaded" the entire text of the page using the request library. In this way, we can also analyse the text, but the results can be distorted by the HTML tags occurred in the text.

```
# Import necessary libraries
from nltk.tokenize import word_tokenize
from nltk.probability import FreqDist

# Tokenize the text into words
tokens_fables = word_tokenize(fables)  # Break down the text
into individual words

# Create a frequency distribution of words
frequency_fables = FreqDist(tokens_fables)  # Count the
occurrences of each word

# Print the 20 most common words and their frequencies
print(frequency_fables.most_common(20))  # Display the top 20
frequent words
```

Program output:

```
[('<', 983), ('>', 983), ("'", 646), ('A', 317), ('HREF=',
317), ('/A', 317), ('BR', 315), ('The', 307), ('and', 254),
('the', 244), (',', 58), ('Lion', 32), ('Fox', 31), ('Ass',
23), ('Wolf', 22), ('His', 21), ('Dog', 13), ('Man', 12),
('Two', 12), ('Eagle', 11)]
```

The result shows that in Aesop's fables **lion**, **fox**, **donkey**, **wolf**, and **dog** appear most often in the given order. It is also possible to determine the number of occurrences of individual animals. For example, seagull occurs only once in the title of Aesop's fables.

```
print(frequency_fables['Seagull'])
```

Program output:

```
1
```

If we take a closer look at the results, the most frequent tokens are '>' and '<'. Their frequency of occurrence is 978. Since we analysed a web page that is entirely in HTML code, the stated tokens are actually the beginning and end tags of the HTML elements of the page. Similarly, in the case of 'A', 'BR' or 'HREF=' tokens, which are actually names or attributes of HTML elements.

8.5.2

To perform the correct analysis, without the stated tokens (parts of the HTML code), we have several options. One of them is to supplement the so-called list of stopwords by the following tokens and then remove the tokens from the list of stopwords.

If we analyse a website, we can use technologies that operate on the website and are designated for content extraction. We can access to individual elements of HTML pages using the DOM - Document Object Model. The **XPath** query language is designated for DOM requests.

The **XPath** query language is intended for DOM requests, work with it was presented in the third chapter.

The most commonly used library for running XPath queries on web pages is the **LXML** library in Python. It is necessary to remember that the XPath language was first developed for queries on XML documents and it is of the greatest importance when processing XML. However, we can also apply its usefulness in the HTML language.

```
import requests
link = "https://www.sacred-texts.com/cla/aesop/index.htm"
f = requests.get(link)
fables = f.text
```

We already have all fables "stored" in the **fables** variable that we created in the previous section.

```
from lxml import html
tree = html.fromstring(fables)
```

All that remains is to call a query over the content of the page, loaded in the XPath fable variable. We will create this by examining the HTML code of the page "http://www.sacred-texts.com/cla/aesop/index.htm".

```
</CENTER>

<HR>
<A HREF="aes000.htm">Preface</A><BR>
<A HREF="aes001.htm">Life Of Aesop</A><BR>
<A HREF="aes002.htm">The Wolf and the Lamb</A><BR>
<A HREF="aes003.htm">The Bat and the Weasels</A><BR>
<A HREF="aes004.htm">The Ass and the Grasshopper</A><BR>
<A HREF="aes005.htm">The Lion and the Mouse</A><BR>
<A HREF="aes006.htm">The Charcoal-Burner and the Fuller</A><BR>
<A HREF="aes007.htm">The Father and His Sons</A><BR>
<A HREF="aes008.htm">The Boy Hunting Locusts</A><BR>
```

It should be noted that the source code of the page is not completely correctly formatted and the only thing we can use is that all fable titles are the content of the **a** element, thus a link. The XPath query will therefore have the form '**//a/text()**', which can be understood as loading the content of all elements **a** on the page.

After constructing the XPath query, we run it and we can display the first 10 results of the XPath query for checking.

```
# Extract all text content from anchor tags ('a') within the
entire XML tree
all_titles = tree.xpath('//a/text()')

print(all_titles[:10])
```

Program output:

```
['Sacred Texts', 'Classics', 'Preface', 'Life Of Aesop', 'The
Wolf and the Lamb', 'The Bat and the Weasels', 'The Ass and
the Grasshopper', 'The Lion and the Mouse', 'The Charcoal-
Burner and the Fuller', 'The Father and His Sons']
```

The first three results are not actually the titles of fables, but only the text of the other three links listed on the investigated page. It is a consequence of not exactly the most suitable page structure. However, there is no other way in this case (also due to the simplicity of the python code). It is clear, that the mentioned three "incorrect titles of the fables" will not affect our analysis of the most frequent words within the titles.

8.5.3

If we already have all the titles of Aesop's fables in the **all_titles** variable, we can repeat the tokenization and analysis of the most frequent words. Note that the **word_tokenize** function needs an input variable of string type. The **all_titles** variable is a list of strings. For this reason, we first merge the list of fable titles into one string variable. We name the variable **all_titles_together**. Merging can be done with several functions, e.g. **join()** and so on. However, to run them, we need to import the STR library. Therefore, for the sake of simplicity, we will do the unification of the

strings´ list into a single string using a cycle. In addition to merging the lists, we also convert all titles to lowercase using the **lower()** function. We can also call the control report of the newly created variable.

```
import requests
link = "https://www.sacred-texts.com/cla/aesop/index.htm"
f = requests.get(link)
fables = f.text
from lxml import html
tree = html.fromstring(fables)
all_titles = tree.xpath('//a/text()')

all_titles_combined = ''
for title in all_titles:
    all_titles_combined += ' ' + title.lower()

print(all_titles_combined)
```

We only have to repeat the tokenization and analysis of the most frequent words for the **all_titles_together** variable. Please note that we are still considering all words/tokens in fable titles. For this reason, the most frequent tokens are 'the' and 'and'.

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.probability import FreqDist

tokens_fables = word_tokenize(all_titles_combined)
frequency_fables = FreqDist(tokens_fables)
print(frequency_fables.most_common(20))
```

Program output:

```
[('the', 550), ('and', 254), (',', 52), ('lion', 32), ('fox',
31), ('ass', 23), ('wolf', 22), ('his', 21), ('dog', 13),
('man', 12), ('two', 12), ('eagle', 11), ('crow', 8), ('in',
8), ('sheep', 8), ('shepherd', 8), ('bull', 7), ('goat', 7),
('horse', 7), ('frogs', 7)]
```

To complete the task, we can use the list of stopwords and remove them from the tokens.

```
from nltk.corpus import stopwords

# Load the list of English stop words
sw = set(stopwords.words('english'))

# Add the token "," (comma) to the list of stop words
```

```
sw.update(',')

# Create new tokens that do not contain stop words
new_tokens = []
for word in tokens_fables:
    if word not in sw:
        new_tokens.append(word)
```

We will then use the new list of tokens as input to determine the frequency of words.

```
frequency_fables = FreqDist(new_tokens)
print(frequency_fables.most_common(10))
```

Program output:

```
[('lion', 32), ('fox', 31), ('ass', 23), ('wolf', 22), ('dog',
13), ('man', 12), ('two', 12), ('eagle', 11), ('crow', 8),
('sheep', 8)]
```

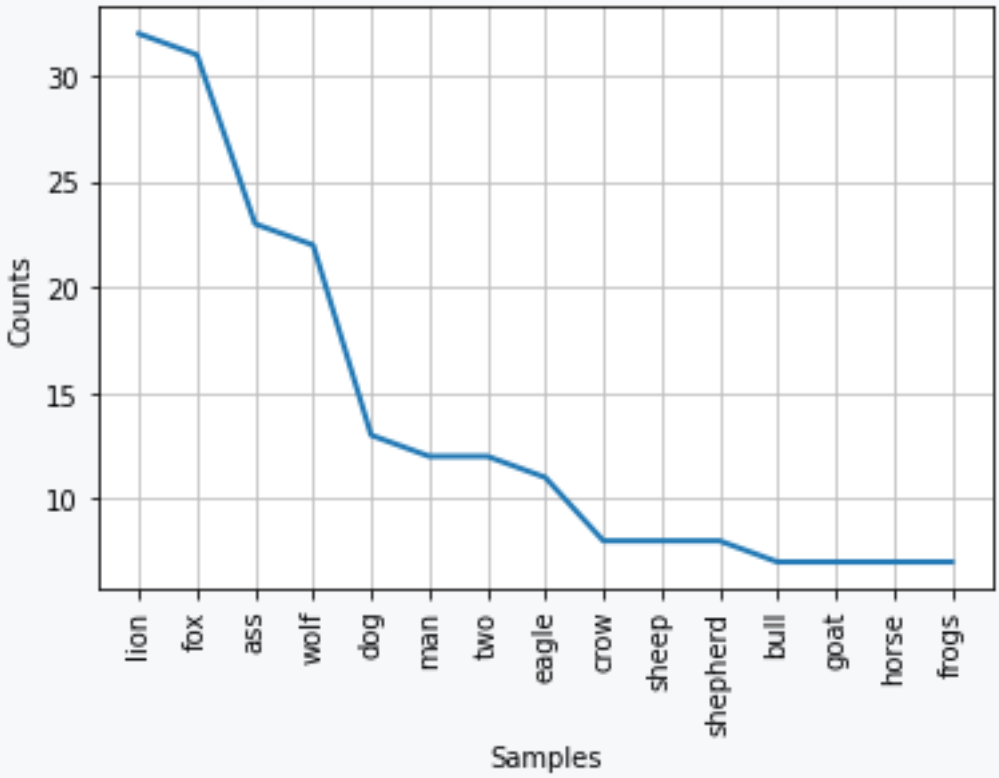
The result shows that lion, fox, ass, wolf, and dog appear most often in Aesop's fables. It is also possible to find out the number of their occurrences.

To conclude our practical example, we will use a function that graphically displays the most common words found in the titles of Aesop's fables.

```
# Command to display the graphic output directly in Jupyter
Notebook
%matplotlib inline

# Plotting the frequency graph for the first 15 words
frequency_fables.plot(15, cumulative=False)
```

Program output:

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


Counts
30
25
20
15
10
lion
fox
ass
wolf
dog
man
two
eagle
crow
sheep
shepherd
bull
goat
horse
frogs
Samples

WordNet

Chapter 9

9.1 WordNet

📖 9.1.1

WordNet is a large lexical database of the English language developed at Princeton University. It organizes English words into groups called **synsets**, which represent a set of synonyms that share the same meaning. Each synset includes definitions, examples of usage, and relationships to other words. This organization makes WordNet a powerful resource for understanding the structure and meaning of language.

One of WordNet's primary purposes is to serve as a bridge between language and computational systems. For example, it allows NLP tools to identify semantic relationships between words, enabling computers to process text more intelligently. It’s used in tasks like synonym replacement, word sense disambiguation, and semantic similarity calculations.

WordNet also provides hierarchical relationships between words. For instance, words like "vehicle" and "car" are connected in a hypernym (broader term) and hyponym (specific term) relationship. These features make it easier to analyze how words relate to each other in various contexts.

Lastly, WordNet integrates words' multiple meanings. For example, the word "bank" can refer to a financial institution or a riverbank. Each sense is associated with a different synset, helping distinguish meanings based on context.

📝 9.1.2

What is a synset in WordNet?

- A group of synonyms sharing the same meaning.
- A single word used in a sentence.
- A grammatical rule for sentence structure.

📖 9.1.3

Synonyms and antonyms

One of WordNet’s most useful features is its ability to identify **synonyms** and **antonyms**. Synonyms are words with similar meanings, and WordNet groups them into synsets. For example, the synset for "happy" includes words like "joyful," "content," and "cheerful." This allows systems to find alternative words while retaining the same meaning.

Antonyms, on the other hand, are words with opposite meanings. WordNet provides antonyms for many words, like "happy" and "sad" or "big" and "small." Identifying

antonyms is essential in tasks such as sentiment analysis, where opposite sentiments must be distinguished.

In computational linguistics, synonym and antonym identification is a common preprocessing step. For instance, when analyzing a document's tone, synonyms can help group similar expressions, while antonyms can highlight contrasting sentiments.

Applications like chatbots, search engines, and machine translation systems use these relationships to make text processing more effective and contextually accurate. For example, a chatbot might suggest alternative phrasing based on a user’s input, improving the conversational experience.

9.1.4

What can WordNet provide in terms of word relationships?

- Synonyms grouped into synsets.
- Antonyms for words.
- Hierarchical relationships like hypernyms and hyponyms.
- Sentence grammar rules.

9.1.5

Hypernyms and hyponyms

WordNet organizes words into a hierarchical structure using **hypernyms** (broader terms) and **hyponyms** (narrower terms). For example, "vehicle" is a hypernym of "car," and "car" is a hyponym of "vehicle." This hierarchy enables NLP systems to understand how concepts are related and classify text effectively.

These hierarchical relationships are useful in taxonomy creation and classification tasks. For instance, in e-commerce, understanding that "laptop" is a hyponym of "computer" helps group similar products together for recommendations.

Another application is in semantic search. A search engine can use hypernym relationships to expand queries. If a user searches for "sports cars," the system could include results for related hyponyms like "convertibles" or "coupe."

The hierarchy also supports generalization and specialization tasks. For example, summarization systems might generalize details (e.g., replacing "Dalmatian" with "dog") for brevity, while classification tasks benefit from recognizing specific terms.

9.1.6

What is the relationship between "dog" and "animal" in WordNet?

- "Dog" is a hyponym of "animal."

- "Dog" is a hypernym of "animal."
- "Dog" and "animal" are synonyms.

📖 9.1.7

Word sense disambiguation

Words often have multiple meanings, and identifying the correct one in context is called **Word Sense Disambiguation (WSD)**. WordNet is a valuable tool for this task because it organizes words into synsets for each meaning. For example, "bank" could refer to a financial institution or the side of a river, and WordNet provides separate synsets for these senses.

To perform WSD, an NLP system analyzes the context of the word. For example, in the sentence "I deposited money at the bank," the financial institution meaning is chosen based on surrounding words like "deposited" and "money." WordNet provides the definitions and example sentences needed to differentiate senses.

WSD has many practical applications. In machine translation, selecting the correct meaning ensures accurate translation. Similarly, in search engines, understanding the intended sense of a query word improves the relevance of results.

Using WordNet for WSD involves comparing the context of the target word with the definitions and examples in its synsets. Algorithms like Lesk and similarity-based approaches often rely on WordNet data to compute the correct sense.

📝 9.1.8

How does WordNet assist in word sense disambiguation?

- By organizing word meanings into synsets.
- By providing definitions and example sentences for each sense.
- By grouping words into sentences.
- By analyzing grammatical rules.

📖 9.1.9

Semantic similarity

Semantic similarity is a measure of how closely related two words or concepts are in meaning. WordNet provides a framework to calculate semantic similarity by leveraging its hierarchical structure and rich lexical relationships. This is particularly useful in applications like clustering, recommendation systems, and information retrieval, where understanding the context and relevance of terms is essential.

WordNet organizes words into synsets and links them hierarchically using hypernym (broader terms) and hyponym (specific terms) relationships. The

similarity between two words is determined by the proximity of their synsets in the hierarchy. For example, "car" and "truck" have a higher semantic similarity compared to "car" and "tree" because their synsets share a common parent ("vehicle") within a closer range.

In NLP tasks, semantic similarity is widely used:

- **Search engines** to rank search results based on conceptual relevance, not just keyword matches. For example, a query for "automobile" should also fetch results about "cars."
- **Recommendation systems** to identify similar items or concepts for personalized recommendations. For instance, recommending books about "poetry" when someone searches for "literature."
- **Text summarization** for replacing verbose terms with simpler synonyms without altering the meaning.

Imagine an educational chatbot helping students. If a student asks about "climate," the system can understand that "weather patterns" is semantically similar and provide relevant content. WordNet's similarity measures make this contextual matching possible.

9.1.10

Which of the following are examples of semantic similarity applications?

- Recommending books based on a user's reading history.
- Detecting semantic similarity between two words.
- Generating synonyms for words in essays.
- Predicting spelling errors in user input.

9.1.11

Cross-linguistic extensions

WordNet's utility goes beyond the English language, with cross-linguistic extensions enabling multilingual natural language processing. These extensions make WordNet an invaluable tool for machine translation, cross-lingual sentiment analysis, and global search applications.

Cross-linguistic extensions are adaptations of the WordNet database for other languages, such as French, Spanish, and Hindi. These versions maintain similar structures of synsets and relationships, allowing the same functionality as English WordNet. The extensions can be linked to the English version, providing a unified framework for multilingual analysis.

Applications of cross-linguistic WordNet

1. **Machine translation** uses mapping words and phrases in one language to their accurate counterparts in another. For instance, translating "apple" in English to "manzana" in Spanish, while maintaining contextual integrity.
2. **Multilingual sentiment analysis** identifies sentiment in text across languages by understanding synonyms and antonyms. For example, determining that "feliz" (Spanish) and "happy" (English) carry the same positive connotation.
3. **Cross-lingual information retrieval** enhances search engines to return relevant results for queries in multiple languages by linking related synsets.

A global e-commerce platform uses WordNet to improve search functionality. If a user searches for "ropa" (Spanish for clothing), the system identifies synonyms and retrieves results for "clothes," "apparel," and "outfits," regardless of language.

9.1.12

Which of the following describes a Cross-Linguistic WordNet feature?

- Linking synsets across multiple languages.
- Translating entire documents automatically.
- Extracting XML data from webpages.

9.1.13

Applications of WordNet

WordNet is widely used in various NLP applications to enhance semantic understanding. One common use case is **text summarization**, where WordNet helps generalize terms or find synonyms to condense information.

Another major application is in **sentiment analysis**, where WordNet's antonyms are used to identify contrasting sentiments. For instance, it can distinguish between positive words like "happy" and negative words like "sad."

Chatbots and **virtual assistants** also benefit from WordNet. By leveraging synonyms and hierarchical relationships, these systems can interpret a user's input in multiple ways and respond more effectively. For example, if a user says "I feel joyful," the chatbot can understand it as synonymous with "happy" and provide a relevant response.

Lastly, **semantic similarity** measures between words or sentences often use WordNet. For instance, applications like plagiarism detection or document clustering compute the similarity of terms based on their synsets, making WordNet an invaluable resource in advanced NLP tasks.

9.1.14

What are some NLP applications where WordNet is useful?

- Sentiment analysis
- Text summarization
- Syntax tree generation
- Audio signal processing

9.2 Practical use

9.2.1

Synonyms

In any text analysis, there is often a consideration, e.g. about synonyms or related words. The issue of "dependence" of words can be partially solved using the so-called WordNet. According to definition, it is a dictionary of words arranged according to semantic relationships.

The **nltk** library offers us an interesting tool for discovering these relationships. The **wordnet** dictionary must be downloaded using the **download()** method before applying.

```
import nltk

nltk.download('wordnet')

from nltk.corpus import wordnet
```

Using the WordNet dictionary, it is possible to find the meaning of words, synonyms and antonyms of words. When working with the dictionary, we will use the so-called synset. It is a simple option used in **nltk** to read relations. Synset instances are groups of synonymous words that express the same concept. Some words have only one synset and some have several. Let's attempt to find the word "joy" in wordnet.

```
from nltk.corpus import wordnet

syns = wordnet.synsets("joy")

print(syns)
```

Program output:

```
[Synset('joy.n.01'), Synset('joy.n.02'),
Synset('rejoice.v.01'), Synset('gladden.v.01')]
```

According to the search result, the word "joy" is found in four synsets. Each synset has a name that ends with the synset number. In our result, there is also an "n" or "v" character between the word and the number. The character "n" indicates a noun and "v" is a verb.

If we want to see words that have a semantic relation to the word "joy", we can report them using the **lemmas()** method.

```
for syn in syns[0].lemmas():
    print(syn.name())
```

Program output:

```
joy
joyousness
joyfulness
```

```
for syn in syns[1].lemmas():
    print(syn.name())
```

Program output:

```
joy
delight
pleasure
```

Note that the first synset contains three words, the second has the same amount, but some words are different. It often happens that a synset contains only one word. We can report the words of all synsets where the word "joy" is found as follows.

```
for syn in syns:
    for lemma in syn.lemmas():
        print(lemma.name())
```

Program output:

```
joy
joyousness
joyfulness
joy
delight
pleasure
rejoice
joy
gladden
joy
```

Additional information is also stored in synsets. For example, most synsets also contain a definition of the examined word.

```
print(syns[0].definition())
print(syns[1].definition())
```

Program output:

```
the emotion of great happiness
something or someone that provides a source of happiness
```

Interestingly, the word "joy" has a different definition in the first synset and a different one in the second. It is common because, e.g. the word "table" can be in a synset with words denoting furniture, but also in a synset of words denoting a data structure - a table.

Another option is to display examples of the word usage in the synset. This example, like the definition, may not be part of every synset.

```
print(syns[0].examples())
print(syns[1].examples())
```

Program output:

```
[]
['a joy to behold', 'the pleasure of his company', 'the new
car is a delight']
```

9.2.2

Antonyms

Besides similar words, it is possible to find in the synset the so-called antonyms. These indicate contrasting concepts (opposites) to the given word, i.e. words with opposite meanings. It is interesting that if we can sometimes find a whole list of synonyms for the word under investigation, antonyms are found only in pairs, i.e. word and its antonym. In the following example, we report all synonyms for the word "joy" together with the antonym of these synonyms (if exist).

```
import nltk
# nltk.download('wordnet')
from nltk.corpus import wordnet

word = "joy"
synonyms = []
antonyms = []

for syn in wordnet.synsets(word):
    for lemma in syn.lemmas():
```

```
        synonyms.append(lemma.name())
        if lemma.antonyms():
            antonyms.append(lemma.antonyms()[0].name())

print('Synonyms:')
print(set(synonyms))
print('Antonyms:')
print(set(antonyms))
```

Program output:

```
Synonyms:
{'joyousness', 'joyfulness', 'joy', 'gladden', 'delight',
'rejoice', 'pleasure'}
Antonyms:
{'sadden', 'sorrow'}
```

9.2.3

Semantic similarity

In the example, we present the calculation of the semantic similarity of the words "joy" and "joyousness" (it is not exactly the similarity of the words, but rather the similarity of the synsets in which the words are found). Considering that these are synonyms, their similarity is equal to 1. The word "sorrow" also has a fairly close meaning. It is actually an antonym of the word "joy". To make sure, in the last example we present the similarity of the words "joy" and "mouse", which are not at all close in meaning. The value of their semantic proximity is very small.

```
import nltk
# nltk.download('wordnet')
from nltk.corpus import wordnet

w1 = wordnet.synset('joy.n.01')
w2 = wordnet.synset('joyousness.n.01')
print(w1.wup_similarity(w2))
```

Program output:

```
1.0
```

```
w1 = wordnet.synset('joy.n.01')
w2 = wordnet.synset('sorrow.n.01')
print(w1.wup_similarity(w2))
```

Program output:

```
0.7142857142857143
```

```
w1 = wordnet.synset('joy.n.01')
w2 = wordnet.synset('mouse.n.01')
print(w1.wup_similarity(w2))
```

Program output:

```
0.1
```

In conclusion, we present "proof" that "soccer brings more joy to a person than chess" :o)

```
w1 = wordnet.synset('joy.n.01')
w2 = wordnet.synset('football.n.01')
print(w1.wup_similarity(w2))

w1 = wordnet.synset('joy.n.01')
w2 = wordnet.synset('chess.n.01')
print(w1.wup_similarity(w2))
```

Program output:

```
0.23529411764705882
0.1
```

9.2.4

Project: Synonym recommendation system

Build a system that recommends synonyms for a given word based on semantic similarity.

You should use the WordNet database to identify synonyms for words. The goal is to develop a system that, given an input word, returns a list of synonyms sorted by their semantic similarity.

**You can implement different methods of measuring similarity, such as path similarity, Wu-Palmer similarity, or Leacock-Chodorow similarity.

1. Load the WordNet database using the NLTK library.

```
# write your code
```

2. Define a function to fetch synonyms for a given word.

```
# write your code
```

3. Implement different similarity metrics to rank the synonyms.

```
# write your code
```

4. Evaluate the performance of the system with example inputs.

```
# write your code
```

9.2.5

Project: Semantic text similarity for document matching

Use WordNet to calculate semantic similarity between two texts (documents) to match similarity.

Apply WordNet to measure the semantic similarity between two documents. The project will involve tokenizing the documents, extracting key words, and calculating the overall semantic similarity based on the WordNet synonyms and word meanings. This approach can be applied in real-world scenarios like document retrieval, plagiarism detection, or recommendation systems.

1. Load the WordNet database using the NLTK library.

```
# write your code
```

2. Tokenize the text documents into words and remove stop words.

```
# write your code
```

3. For each word in the document, retrieve its synonyms and meanings using WordNet.

```
# write your code
```

4. Calculate pairwise semantic similarity between the words using WordNet's similarity functions.

```
# write your code
```

5. Combine individual word similarities to compute the overall document similarity score.

```
# write your code
```

6. Apply the system to a set of documents/texts and use it for document matching or ranking.

```
# write your code
```

Document Models

Chapter 10

10.1 Introduction

📖 10.1.1

Document models are a foundational concept in text analysis and mining, enabling efficient representation and processing of textual data. When working with textual information, the choice of data representation directly impacts the success of the analysis. There are two primary approaches to representing text documents: **data with dependencies** and **data without dependencies**.

The first approach, **data with dependencies**, considers the sequence and positional relationships of words within a sentence. This approach is vital for applications where grammatical structure or stylistic elements are analyzed, such as in natural language generation or linguistic studies. For example, understanding the difference between "The cat chased the dog" and "The dog chased the cat" requires this type of representation. However, this method can be computationally expensive due to the need to maintain word-order information.

The second approach, **data without dependencies**, focuses solely on the presence of words, disregarding their order in a sentence. This representation is widely used in applications such as text classification and topic modeling, where the position of words is less important than their occurrence. For instance, in document classification, knowing that a document contains the words "finance," "budget," and "investment" might be sufficient to classify it as financial content.

This distinction between dependency-aware and dependency-agnostic models helps in choosing the most appropriate method for specific problems.

📝 10.1.2

Which of the following are features of data without dependencies?

- Words are represented based on their occurrence without considering their position.
- Suitable for text classification and topic modeling.
- Requires high computational resources to maintain word order.
- Focuses on grammatical structure and style analysis.

📖 10.1.3

When analyzing textual data, especially in large datasets, the use of **document models** simplifies processing. A document model transforms the text into a structured format, enabling faster and more accurate analysis.

The **Bag of Words (BoW)** model is one of the simplest representations. In BoW, documents are represented as a collection of unique words (vocabulary), where each word's frequency in the document is recorded. For example, the sentence *"The*

cat chases the mouse" becomes a vector: {"the": 2, "cat": 1, "chases": 1, "mouse": 1}. This model disregards word order, focusing solely on occurrence and frequency.

Another common model is the **TF-IDF (Term Frequency-Inverse Document Frequency)**. This approach builds on BoW but also considers the importance of words. Words frequently appearing in a document but rarely in the dataset are given higher weights, reflecting their uniqueness. For instance, in a collection of news articles, the word "economy" might have a high weight in a financial news piece but low weight across the entire corpus.

Both BoW and TF-IDF are computationally efficient, making them ideal for tasks such as document search, classification, and clustering.

10.1.4

What is the key feature of the Bag of Words model?

- Represents documents as a collection of unique words and their frequencies.
- Considers the order of words in a sentence.
- Applies weights based on the rarity of words in the corpus.

10.1.5

Beyond BoW and TF-IDF, **vector models** offer a more advanced representation of textual data. These models convert words or documents into numerical vectors, enabling mathematical operations for similarity and classification tasks.

One such model is the **Word2Vec**, which uses neural networks to map words into continuous vector spaces. Unlike BoW, Word2Vec captures semantic relationships between words. For example, in a trained Word2Vec model, the vector for "king" minus "man" plus "woman" often results in a vector close to "queen." This semantic understanding is crucial for applications like recommendation systems and sentiment analysis.

Another approach is **doc2vec**, which extends Word2Vec to represent entire documents instead of individual words. Doc2Vec captures the context of a document, making it suitable for paragraph or document-level tasks.

These models are widely used in applications requiring an understanding of word semantics and contextual relationships.

10.1.6

Which of the following are characteristics of **Word2Vec**?

- Represents words as continuous vectors.
- Captures semantic relationships between words.

- Focuses on word frequency alone.
- Represents entire documents without context.

📖 10.1.7

The choice of document models often depends on the specific task and the available computational resources.

Bag of Words and **TF-IDF** are simple yet effective models for tasks that do not require semantic understanding. They are computationally efficient but fail to capture relationships between words. For example, these models cannot distinguish between synonyms like "car" and "automobile."

In contrast, **vector-based models** like Word2Vec and doc2vec require more computational power but offer richer representations. These models are better suited for tasks requiring semantic understanding, such as summarization or translation. However, their training can be time-consuming, and they often require large datasets.

The challenge lies in balancing simplicity and effectiveness, considering factors such as the size of the dataset, the complexity of the task, and available resources.

📝 10.1.8

Which factor influences the choice of a document model?

- Availability of computational resources.
- Predefined relationships in a knowledge graph.
- Lack of text data for analysis.

📖 10.1.9

Document models are at the heart of many NLP applications. In **document retrieval**, models like TF-IDF are used to rank documents based on their relevance to a search query. For example, search engines use document models to retrieve and rank web pages.

In **topic modeling**, models like LDA (Latent Dirichlet Allocation) analyze large text corpora to identify common themes or topics. For instance, LDA might identify topics such as "sports," "politics," or "entertainment" in a collection of news articles.

Sentiment analysis also leverages document models to determine the sentiment of text, such as whether a product review is positive, negative, or neutral. Using BoW or vector-based models, sentiment analysis systems can accurately classify the sentiment by analyzing the words used in the text.

10.1.10

Which of the following are applications of document models?

- Document retrieval in search engines.
- Topic modeling in large text corpora.
- Speech recognition systems.
- Image classification tasks.

10.2 Boolean document model

10.2.1

The Boolean document model is one of the simplest approaches to representing text documents in information retrieval systems. It provides a straightforward method for determining the presence or absence of specific words in a document. This binary representation records whether a word exists in the document (1) or not (0).

For example, if we analyze the sentence *"The cat chases the mouse,"* the Boolean representation could look like this for a vocabulary of {"the," "cat," "dog"}:

- "the": 1
- "cat": 1
- "dog": 0

This simplicity makes the Boolean model highly efficient for basic search tasks, such as finding documents containing specific keywords. It operates on logical queries involving AND, OR, and NOT operations. For instance, a search query "cat AND mouse" will retrieve only those documents containing both words.

The Boolean document model's advantage lies in its simplicity and ease of implementation. However, it does not consider word frequency or the context of the words, which limits its effectiveness in more nuanced applications.

10.2.2

What does the Boolean document model record about words in a document?

- Whether a word is present in the document or not.
- The exact position of words in a document.
- The frequency of words in the document.

10.2.3

The Boolean document model relies on logical operations to retrieve relevant documents based on user queries. These operations include:

1. AND - retrieves documents containing all specified keywords. For example, "cat AND mouse" will return documents where both "cat" and "mouse" are present.
2. OR - retrieves documents containing at least one of the specified keywords. For example, "cat OR mouse" will return documents with either "cat," "mouse," or both.
3. NOT - Excludes documents containing the specified keyword. For example, "cat NOT mouse" will return documents with "cat" but not "mouse."

Consider a simple database of three documents:

- Document 1: "The cat is sleeping."
- Document 2: "The dog is barking."
- Document 3: "The cat and the dog are playing."

For the query "cat AND dog," only Document 3 is returned since it contains both words. For "cat OR dog," Documents 1, 2, and 3 are returned. For "cat NOT dog," only Document 1 is returned.

These operations make the Boolean model efficient for straightforward searches, especially in systems where users are expected to know the exact keywords they are looking for.

10.2.4

Which of the following are Boolean operations used in the document model?

- AND
- OR
- NOT
- XOR

10.2.5

The Boolean document model offers significant advantages due to its simplicity and clarity. It is computationally efficient, as it involves only binary operations, making it suitable for systems with limited resources. The model is particularly effective for exact-match queries, such as legal or medical document retrieval, where precision is crucial.

However, the Boolean model has notable limitations. It does not consider word frequency or context, which are often important for understanding document relevance. For example, in a document about cats, the Boolean model treats the word "cat" the same whether it appears once or a hundred times. Additionally, the model does not rank documents by relevance, meaning all retrieved documents are considered equally suitable for the query.

To address these shortcomings, more advanced models like the vector space model or TF-IDF are often used in conjunction with or as alternatives to the Boolean model.

10.2.6

What is a limitation of the Boolean document model?

- It does not rank documents by relevance.
- It is computationally inefficient.
- It cannot handle exact-match queries.

10.2.7

Task: Princess Fiona is looking for a groom. She would like to choose one who is beautiful, good and does not envy. As a good computer scientist, she recorded the basic characteristics of the suitors in a clear table.

					
goodness	1	0	1	0	1
beauty	1	1	1	0	0
envy	0	1	0	1	0
wealth	1	1	0	1	0

It is clear from the table that Prince Charming is beautiful, envious and lacks goodness. Shrek is good and, according to Fiona, beautiful. Similarly, the characteristics of the other two suitors can be read from the table. In this way, Fiona actually created a Boolean model of the properties of her suitors.

From it, we can easily make vectors of the suitor's properties (columns of the table). For example, the vector for Prince Charming is (0,1,1,1), Shrek is (1,1,0,0), etc. We assume that the order of the examined properties will not change, and also that no new properties will be added. Similarly, (if the order of suitors is fixed) the vector for **goodness** (1,0,1,0,1), **beauty** (1,1,1,0,0), and **envy** (0,1,0,1, 0).

If we work with Boolean vectors, we can relatively easily multiply them, count them, etc. Princess Fiona is looking for a groom who is good and not jealous. To find him, we just need to multiply the **good** vector with the negation of the **envy** vector.

```
Goodness AND NOT(Envy)
(1,0,1,0,1) AND NOT(0,1,0,1,0) = 10101 * 10101 = 10101
```

The resulting vector tells us that the first fairy creature in the table - Princess Fiona, the third - Shrek and the last (fifth) fairy-tale creature - Papa Smurf meet the search result. They are beings who are good and not jealous. So with this approach, Fiona would choose Papa Smurf or Shrek as her groom.

What if Fiona wants someone, who is "handsome and not jealous?".

```
Beauty AND NOT(Envy)
(1,1,1,0,0) AND NOT(0,1,0,1,0) = 11100 * 10101 = 10100
```

Besides Fiona, a third fairy creature fulfils the query result - Shrek. It is obvious that with several fairy creatures and a whole range of their examined properties, the calculation would be equally simple and, above all, fast.

10.2.8

Task: Princess Fiona wants to find which suitor is most like her.

The second advantage of the Boolean model is its use in searching for similarities. From the previous text, we know that individual suitors are expressed using a vector. For each suitor, we need to determine a number that expresses the degree of similarity to Fiona. In simplicity, this number will represent the number of common properties. The easiest choice is to count how many properties they have in common.

```
Fiona ·Charming = (1,1,0,1) · (0,1,1,1) = 1.0 + 1.1 + 0.1 +
1.1 = 2
Fiona ·Shrek = (1,1,0,1) · (1,1,0,0) = 1.1 + 1.1 + 0.0 + 1.0 =
2
Fiona ·Scrooge McDuck = (1,1,0,1) · (0,0,1,1) = 1.0 + 1.0 +
0.1 + 1.1 = 1
Fiona ·Papa Smurf = (1,1,0,1) · (1,0,0,0) = 1.1 + 1.0 + 0.0 +
1.0 = 1
```

The results show that Princess Fiona is the most similar to Prince Charming and Shrek.

For the similarity of the vectors of fairy creatures, we actually used vector dot product. The dot product is an operation over two n-dimensional vectors. The result of this operation is a number (a scalar, not a vector).

Let $\vec{x} = (x_1, x_2, \ldots, x_n)$ and $\vec{y} = (y_1, y_2, \ldots, y_n)$ be vectors, then we can easily calculate the vector dot product as

$$\vec{x} \cdot \vec{y} = (x_1 y_1 + x_2 y_2 + \cdots + x_n y_n).$$

10.3 Vector model of the document

📖 10.3.1

The Boolean document model provides a simple binary representation of text data but has significant limitations in capturing the complexity of language. For example, it cannot account for the frequency or importance of words in a document, which are often critical in real-world text analysis. To address these issues, the **vector document model** was developed.

In the vector model, documents and words are represented numerically in a high-dimensional space. Unlike the Boolean model, which uses binary values (0 or 1) to indicate the presence or absence of a word, the vector model allows for more detailed representation by including features like word frequency and importance. Each document is represented as a vector of numbers, with each dimension corresponding to a unique term in the vocabulary.

For instance, consider three sentences:

1. "The cat sleeps."
2. "The dog sleeps."
3. "The cat chases the dog."

A vector representation would encode the occurrence of each unique word (e.g., "cat," "dog," "sleeps") across all documents. This richer representation enables more advanced text analysis, such as determining document similarity or identifying the most important terms in a corpus.

📝 10.3.2

Which of the following are features of the vector document model?

- It accounts for word frequency.
- It represents documents in a numerical vector space.
- It uses binary values for words.
- It ignores the importance of words.

📖 10.3.3

The vector document model introduces significant improvements over the Boolean model. One key advantage is its ability to incorporate **word frequency** into the representation. Words that appear more frequently in a document contribute more to its vector, reflecting their importance within that specific context. This is useful in applications such as document classification and clustering.

Additionally, the vector model supports the use of **weighting schemes**, such as TF-IDF (Term Frequency-Inverse Document Frequency), to emphasize terms that are

highly relevant to a particular document but less common across the entire corpus. For example, in a news article about space exploration, the word "space" would receive a higher weight compared to common words like "the" or "and."

The flexibility of the vector model also enables advanced similarity measures, such as cosine similarity, to compare documents. This makes it well-suited for search engines and recommendation systems, where ranking documents by relevance is essential.

📝 10.3.4

What is a benefit of the vector document model over the Boolean model?

- It incorporates word frequency and importance.
- It uses binary values for all features.
- It cannot compare document similarity.

🕮 10.3.5

In the vector model, documents are often represented using a **word frequency table**. Each document is treated as a vector, where each dimension corresponds to a term in the vocabulary, and the values indicate the term's frequency in that document.

For example, using Python's **CountVectorizer** from the **scikit-learn** library, we can construct a word frequency vector for the sentences:

1. "The cat sleeps."
2. "The dog sleeps."
3. "The cat chases the dog."

After applying **CountVectorizer**, we might get the following matrix:

```
cat | dog | sleeps | chases
 1  |  0  |    1   |   0     # Document 1
 0  |  1  |    1   |   0     # Document 2
 1  |  1  |    0   |   1     # Document 3
```

Each row represents a document, and each column represents a word in the vocabulary. This table makes it possible to compare documents quantitatively, such as calculating their similarity or clustering them into related groups.

📝 10.3.6

What does a word frequency table represent in the vector model?

- The count of each word in a document.

- The numerical representation of documents in vector space.
- The relative position of words in a sentence.
- The binary presence or absence of words.

10.3.7

Task: Princess Fiona discovers that the suitors are not equally rich. Although Charming and Scrooge McDuck, in the position of the vector element expressing wealth, have the value 1, but the amount of their wealth differs significantly. How do we express "size", and/or the importance of the examined properties?

The Boolean model presented in the previous subsection is very simple for document representation, but mainly for the document search itself. However, it cannot capture the quantity or quality of stored features/words in a document. For this reason, it is practically only used for studying an introduction to the field of document search.

The vector model of the document is more applicable (the Boolean model also used vectors, the so-called Boolean vectors, whose elements take the value 0 or 1). In the vector model, we also assume an invariant order of documents (in our example - fairy creatures) and also of words/terms (in our example – suitors properties).

In contrast to the Boolean model, we can take into account the frequency of words occurrence, the importance of words, etc. in the vector model.

In the following sample, we can see a typical result of the word frequency table in a sentence. **CountVectorizer()** method is used to create a word frequency vector directly from the texts. The input to the method is a list of texts. The method is part of the **scikit-learn** library.

```
shrek = "heroism heroism heroism heroism heroism heroism
heroism heroism heroism heroism envy envy"
charming = "heroism envy envy envy envy envy"
```

After loading the simplified texts, we import the libraries. We will use the **pandas** library for an informative model report.

```
import pandas
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
```

We will create the document vectors themselves with the **fit_transform()** function. For checking, we convert the output corpus into a data frame and print out.

```
corpus = vectorizer.fit_transform([shrek, charming])
```

```
print(pandas.DataFrame(corpus.A,
columns=vectorizer.get_feature_names_out(), index=["shrek",
"charming"]))
```

Program output:

```
           envy  heroism
shrek         2       10
charming      5        1
```

It is clear that we have created two frequency vectors

- shrek (2,10)
- charming (1,5).

10.3.8

In the chapter about the Boolean model of documents, we investigated how many properties the fairy creatures have in common. So we determined for the *fiona* vector its proximity to the *shrek* and *charming* vector. This was done by a simple vector dot product. We can choose the same procedure in the case of the vector model. Let's imagine that the fairy creatures from the Kingdom beyond the seven mountains make their badges on an Internet dating site. For the sake of simplicity, we will focus only on two traits: "heroism" and "envy". Badges could look like this:

The trait "heroism" will belong to the first number of the vector, and the trait "envy" to the second number. If we consider word frequency vectors, they will look like this.

$$\overrightarrow{fiona} = (4\,;2), \qquad shrek = (10\,;2), \qquad \overrightarrow{\text{charming}} = (1;5)$$

Similar to the Boolean document model, a vector dot product can also be used in this case.

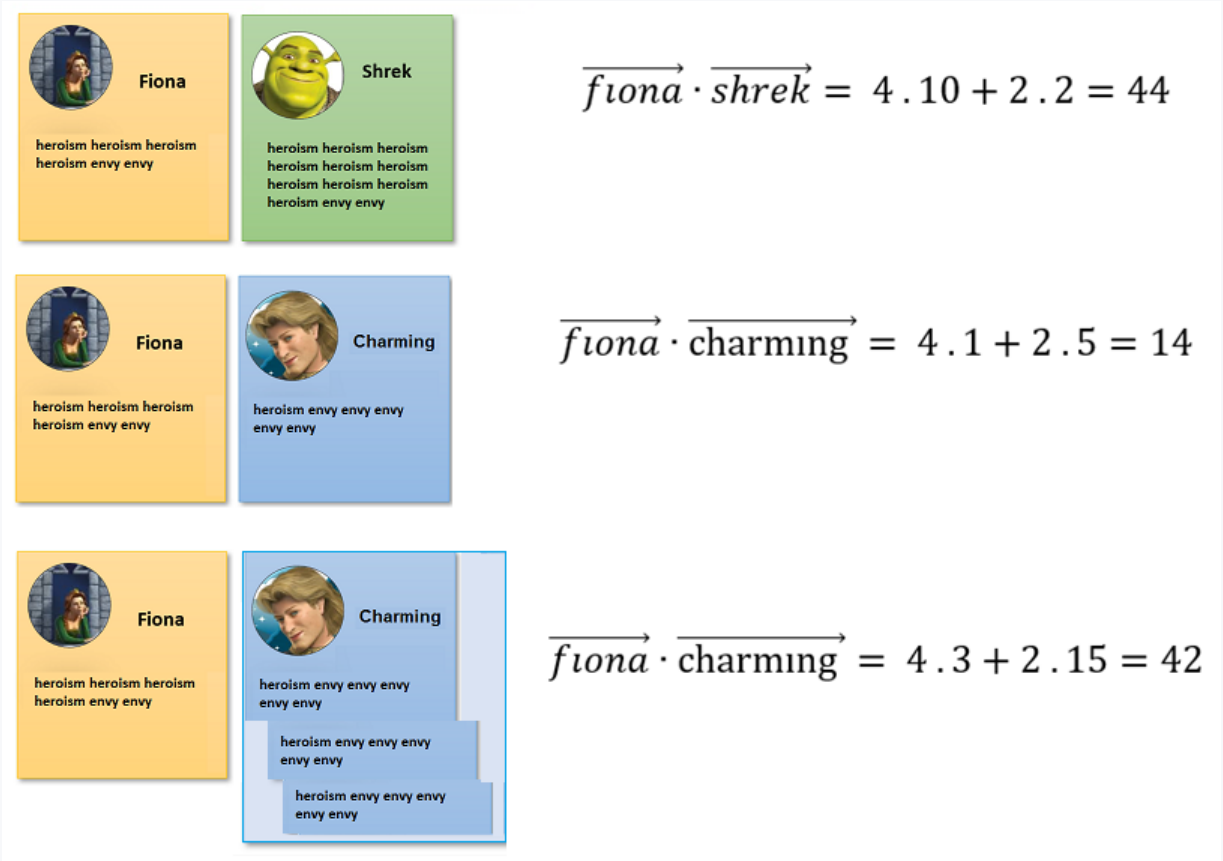
$$\overrightarrow{fiona} \cdot \overrightarrow{shrek} = 4.10 + 2.2 = 44$$
$$\overrightarrow{fiona} \cdot \overrightarrow{charming} = 4.1 + 2.5 = 14$$

From the result, it can be seen that Fiona has more features in common with Shrek, even taking into account the frequency of their occurrence. We are in a fairy tale????.

10.3.9

However, the vector model has its weak point. If fairy creatures wrote their own badges, Shrek probably would not say much about himself. On the other hand, Charming would definitely "expand" and boast about himself. Charming's badge would probably be longer, i.e. with more text than Shrek's budge. For the simplicity of the example, let's assume that Charming only copies the text of his budge three times in order to achieve the largest budge. The word "heroism" will appear 3 times and envy 15 times in such badge. The vector representing Charming's badge is three times his previous one, i.e.

Then the feature similarity calculation would look like this:

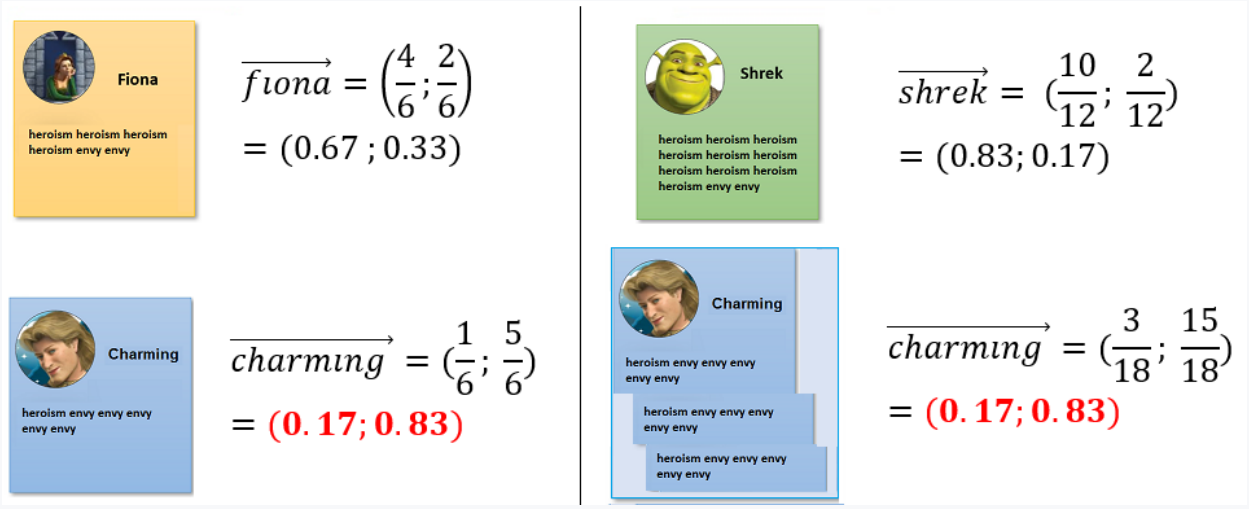


📖 10.3.10

Despite the fact that Shrek's resemblance to Fiona is the highest, let's note that in the long Charming's badge, his resemblance is almost similar. It is obvious that if Charming had copied his text, e.g. 10 times, he would be the best candidate (the candidate with the highest similarity value) for all the dating “hungry” women on the dating site.

From the example, it is clear that the vector model with frequencies of words/terms greatly disqualifies short texts and disproportionately helps documents with long texts.

A simple solution is not to create a vector of (so-called absolute) word frequencies, but to work with relative frequencies. The terms absolute and relative frequencies in statistics do not mean anything other than the actual number and the number converted to the proportional (percentage) representation of words in the document. In the example with badges, it is necessary **to divide the number of words by the number of all words** on the badge of the fairy creature. By relative frequencies, the creature vector would look as follows:



It can be seen from the example that the vector is the same for the relative frequencies even for Charming's original badge and also for copying its properties three times. For completeness, we present the final calculation of similarities using the vector dot product.

$$\overrightarrow{fiona} \cdot \overrightarrow{shrek} = 0.67 * 0.83 + 0.33 * 0.17 = 0.5561 + 0.0561 = 0.6122$$

$$\overrightarrow{fiona} \cdot \overrightarrow{charming} = 0.67 * 0.17 + 0.33 * 0.83 = 0.1139 + 0.2739 = 0.3878$$

$$\overrightarrow{fiona} \cdot \overrightarrow{3\,x\,charming} = 0.67 * 0.17 + 0.33 * 0.83 = 0.1139 + 0.2739 = 0.3878$$

10.4 Bag of words

📖 10.4.1

The **Bag of Words (BoW)** model is a fundamental method for representing text data in machine learning and NLP. It is called "Bag of Words" because it treats each document as a collection (or bag) of words, disregarding the order of the words and focusing solely on their presence or frequency.

In this model, a vocabulary of unique words across all documents in a corpus is created. Each document is then represented as a vector, where each element corresponds to a word in the vocabulary. The value of each element is typically the count of the word's occurrence in the document.

For example, in a corpus with two sentences:

1. "The cat sat on the mat."
2. "The dog sat on the mat."

The vocabulary is: ["The," "cat," "sat," "on," "the," "mat," "dog"]. Each document can be represented as a vector of word frequencies:

- Sentence 1: [1, 1, 1, 1, 0, 1, 0]
- Sentence 2: [1, 0, 1, 1, 0, 1, 1]

The BoW vector can be either boolean or numerical, depending on how it represents the presence or frequency of words:

- Boolean vector - where each element of the vector is either 0 or 1 (0 indicates the word is not present in the document, 1 indicates the word is present in the document).
- Numerical vector where each element of the vector is a number that represents the frequency of the word in the document.

In practice, **numerical vectors** are more frequently used, as they provide richer information about the document by considering word frequency, which is important for many machine learning applications.

📝 10.4.2

What are the characteristics of the Bag of Words model?

- It creates a vector of word counts.
- It builds a vocabulary of unique words.
- It focuses on word order.
- It uses grammatical relationships between words.

📖 10.4.3

The BoW model involves the following steps:

1. Tokenization split the text into individual words or tokens.
2. Vocabulary creation create a list of all unique words in the corpus.
3. Vectorization represent each document as a vector based on the frequency of words in the vocabulary.

For example, consider the corpus:

- Document 1: "I love programming."
- Document 2: "Programming is fun."

The vocabulary is: ["I," "love," "programming," "is," "fun"].

The vectors are:

- Document 1: [1, 1, 1, 0, 0]
- Document 2: [0, 0, 1, 1, 1]

This process converts text into numerical data that can be used for machine learning models, but it does not consider the order or context of the words.

📝 10.4.4

What is the first step in creating a Bag of Words representation?

- Splitting the text into individual tokens.
- Counting the frequency of words in the document.
- Sorting the vocabulary alphabetically.

📖 10.4.5

Advantages and limitations

The Bag of Words model is simple, interpretable, and easy to implement. It is especially effective for tasks like document classification and spam detection.

However, it has several limitations:

- Lack of context - the model disregards the order of words, making it unable to understand context or meaning.
- High dimensionality - for large corpora, the vocabulary can become very large, leading to sparse and high-dimensional vectors.
- Inability to handle synonyms - words like "happy" and "joyful" are treated as completely unrelated.

Despite these limitations, BoW remains a useful foundational model for text analysis, and its simplicity makes it a good starting point for NLP tasks.

📝 10.4.6

Which of the following are limitations of the Bag of Words model?

- It creates high-dimensional vectors for large vocabularies.
- It cannot handle synonyms effectively.
- It captures word order in sentences.
- It is difficult to implement.

🕮 10.4.7

Bag of Words applications

The Bag of Words model is widely used in various text analysis tasks:

- Spam detection - represent emails as vectors of word counts to identify spam words like "free," "offer," or "win."
- Document classification - classify news articles into categories like sports, politics, or entertainment based on their word usage.
- Sentiment analysis - determine whether a review is positive or negative by analyzing the frequency of words like "great" or "terrible."
- Information retrieval - search engines use BoW to match user queries with relevant documents.

These applications demonstrate the practical utility of BoW despite its simplicity. However, for more nuanced tasks requiring context, other models may be more effective.

📝 10.4.8

Which of the following is an application of the Bag of Words model?

- Classifying emails as spam or not spam.
- Predicting the next word in a sentence.
- Translating text from one language to another.

🕮 10.4.9

Enhancements to Bag of Words

To address the limitations of the Bag of Words model, various enhancements have been developed:

- TF-IDF (Term Frequency-Inverse Document Frequency) adjusts the word frequency by its importance in the corpus, giving less weight to common words.
- N-grams - instead of individual words, N-grams consider sequences of N words, capturing some context and word order.
- Dimensionality reduction - techniques like Principal Component Analysis (PCA) can reduce the high dimensionality of BoW vectors.

For instance, applying TF-IDF to a Bag of Words model for spam detection would downweight common words like "the" and emphasize more distinctive terms like "win" or "offer."

10.4.10

Project: Bag of word implementation

(by https://builtin.com/machine-learning/bag-of-words)

Implement the bag-of-words model.

1. Method implementation

```
# Assumes that 'doc' is a list of strings and 'vocab' is some
iterable of vocab
# words (e.g., a list or set)
def get_bag_of_words(doc, vocab):
   # Create initial dictionary which maps each vocabulary word
to a count of 0
   word_count_dict = dict.fromkeys(vocab, 0)
   # For each word in the doc, increment its count
   for word in doc:
       word_count_dict[word] += 1
   # Now, initialize the vector to a list of zeros
   bag = [0] * len(vocab)
   # For every vocab word, set its index equal to its count
   for i, word in enumerate(vocab):
       bag[i] = word_count_dict[word]
   return bag
```

2. Method use

- This code assumes that we have already managed to represent our document as a list of separated strings.

```
import re
```

```
# Define the vocabulary
vocab = ['a', 'am', 'and', 'anywhere', 'are', 'be', 'boat',
'box', 'car',\
        'could', 'dark', 'do', 'eat', 'eggs', 'fox', 'goat',
'good', 'green',\
        'ham', 'here', 'house', 'i', 'if', 'in', 'let',
'like', 'may', 'me',\
        'mouse', 'not', 'on', 'or', 'rain', 'sam', 'say',
'see', 'so', 'thank',\
        'that', 'the', 'them', 'there', 'they', 'train',
'tree', 'try', 'will',\
        'with', 'would', 'you']
# Define the document
doc = ("I would not like them here or there.\n"
      "I would not like them anywhere.\n"
      "I do not like green eggs and ham.\n"
      "I do not like them")
# Convert to lowercase
doc = doc.lower()
# Split on all non-alphanumeric characters (i.e., whitespace
and punctuation)
doc = re.split("\W", doc)
# Drop empty strings that arise from splitting
doc = [s for s in doc if len(s) > 0]
bag_of_words = get_bag_of_words(doc, vocab)
print(bag_of_words)
```

Program output:

```
[0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 0, 0, 0, 1, 1, 1,
0, 4, 0, 0, 0, 4, 0, 0, 0, 4, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 3,
1, 0, 0, 0, 0, 0, 0, 2, 0]
```

We used a regular expression library to split the document into words only; the pattern "\W" represents any non-word character.

10.4.11

Project: Visualisation of Bag of Words representation for a document

Create a Python project that converts a single document into its Bag of Words (BoW) representation and visualizes the word frequencies using various visual methods, such as bar charts, word clouds, and heatmaps.

1. Import libraries

```
import matplotlib.pyplot as plt
from wordcloud import WordCloud
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd
```

Program output:

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

2. Input Document:

- Accept a document as input (either as plain text or from a file).

```
# Step 1: Load Document
document = "He who carries a pure heart does not need much to
be happy. He easily takes off from the morning dew on lame
wings to heaven. Little children of God. Wandering stars.
Lilies follow them. And God knows it. And He will not forget.
Neither will we."
```

3. Preprocessing:

- Tokenize the document.
- Convert text to lowercase.
- Remove stop words, punctuation, and special characters.
- Optional: Perform stemming or lemmatization.

```
# Step 2: Preprocess Text
def preprocess(text):
    import re
    from nltk.corpus import stopwords
    stop_words = set(stopwords.words('english'))
    text = re.sub(r'[^\w\s]', '', text.lower())
    tokens = text.split()
    tokens = [word for word in tokens if word not in
stop_words]
    return ' '.join(tokens)
```

```
processed_text = preprocess(document)
```

4. Bag of Words representation:

- Create a BoW model using a library like CountVectorizer from scikit-learn.
- Alternatively, manually count word frequencies.

```
# Step 3: Create BoW Representation
vectorizer = CountVectorizer()
bow_matrix = vectorizer.fit_transform([processed_text])
bow_df = pd.DataFrame(bow_matrix.toarray(),
columns=vectorizer.get_feature_names_out())
```

5. Data preparation for visualization:

- Extract unique words and their frequencies from the BoW model.
- Sort words by frequency for better visualization.

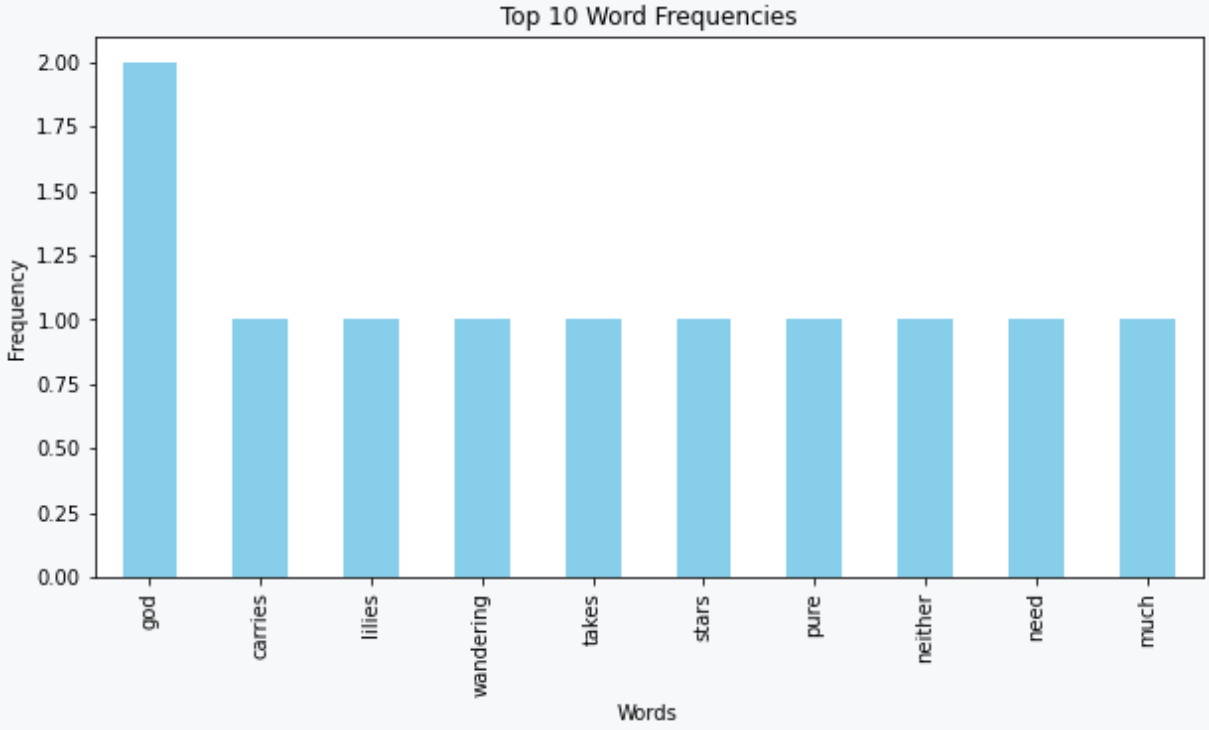
```
word_counts = bow_df.sum().sort_values(ascending=False)
```

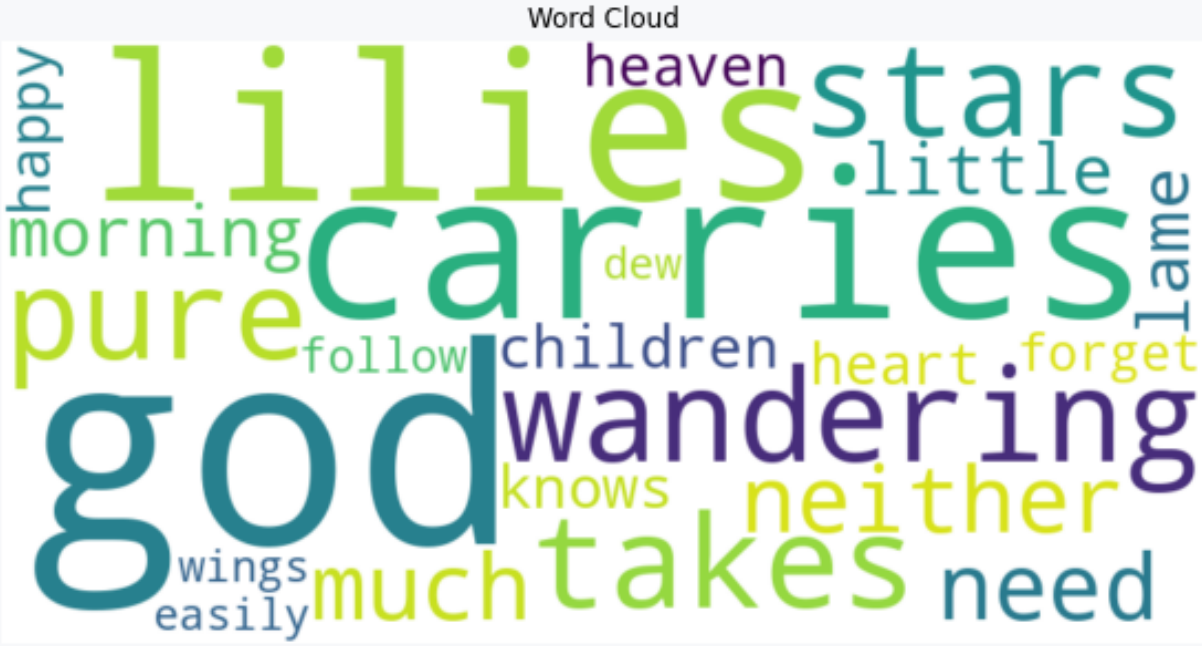
6a. Visualization:

- **Bar chart** displays the top N most frequent words and their counts.
- **Word cloud** shows word frequencies in a cloud, where the size of each word is proportional to its frequency.

```
# Step 4: Visualization
# Bar Chart
plt.figure(figsize=(10, 5))
word_counts.head(10).plot(kind='bar', color='skyblue')
plt.title("Top 10 Word Frequencies")
plt.xlabel("Words")
plt.ylabel("Frequency")
plt.show()

# Word Cloud
wordcloud = WordCloud(width=800, height=400,
background_color='white').generate_from_frequencies(word_count
s)
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("Word Cloud")
plt.show()
```

Program output:





5b. Visualization:

- **Heatmap** creates a heatmap of word counts for sentences or sections of the document.

```
# finish code
```

10.5 TF-IDF model

📖 10.5.1

In text analysis and NLP, not all words are equally important. While some words, like "the" or "and," appear frequently across most documents, they often carry little significance for distinguishing between documents. To address this, the concept of **Inverse Document Frequency (IDF)** is introduced.

IDF is a statistical measure that evaluates how important a word is to a document relative to a collection of documents (also called a corpus). The key idea is that words occurring in fewer documents are more informative and receive a higher IDF value, while words that appear in many documents have lower IDF values.

For instance, consider a corpus of three documents:

1. "The cat sleeps on the mat."
2. "The dog sleeps in the house."
3. "The bird sleeps in the tree."

Here, the word "sleeps" appears in all three documents, making it less unique. On the other hand, words like "cat," "dog," and "bird" appear in only one document each, giving them higher IDF values.

📝 10.5.2

What is the purpose of IDF in text analysis?

- To determine how important a word is relative to a corpus.
- To emphasize rare words across documents.
- To measure the frequency of a word in a single document.
- To increase the weight of commonly occurring words.

📖 10.5.3

The IDF value for a term is calculated using the following formula:

$$IDF(t) = \log\left(\frac{N}{n_t}\right)$$

Where:

- **t** is the term (word) being evaluated.
- **N** is the total number of documents in the corpus.
- $\mathbf{n_t}$ is the number of documents containing the term **t**.

This formula ensures that terms appearing in fewer documents get higher IDF values, while terms present in many documents get lower values. For example, if a corpus has 100 documents, and the term "cat" appears in 5 of them, the IDF for "cat" would be:

$$IDF(\text{"cat"}) = \log\left(\frac{100}{5}\right) = \log(20) \approx 1.3$$

In contrast, if the term "the" appears in all 100 documents, its IDF would be:

$$IDF(\text{"the"}) = \log\left(\frac{100}{100}\right) = \log(1) = 0$$

10.5.4

What does n_t represent in the IDF formula?

- The number of documents containing the term.
- The frequency of the term in a single document.
- The total number of terms in the corpus.

10.5.5

IDF is a cornerstone of text analysis because it allows algorithms to focus on meaningful words while ignoring less informative ones. When combined with Term Frequency (TF), IDF becomes part of the **TF-IDF weighting scheme**, which is widely used in search engines, document classification, and recommendation systems.

Consider a search engine indexing millions of webpages. If a user searches for "machine learning," the term "machine" might appear in many unrelated documents, but "learning" might be more unique to relevant content. By applying IDF, the search engine can prioritize documents where "learning" is significant, improving the relevance of the search results.

For example, in a dataset of articles:

- "AI revolutionizes machine learning."
- "Learning new recipes is fun."
- The term "learning" would have a higher IDF if it appears in fewer unrelated contexts.

IDF also plays a role in filtering out stop words. Words like "and," "the," and "is" have near-zero IDF values, ensuring they don't dominate the analysis, allowing for a cleaner focus on more distinctive terms.

10.5.6

How does IDF improve text analysis?

- It increases the weight of distinctive terms.
- It helps filter out stop words.
- It prioritizes common words across documents.
- It measures the total frequency of terms in a document.

10.5.7

Relative frequencies are the basis of the TF-IDF model. In the model, the frequency of the term TF (Term frequency) is defined, it represents the mentioned relative frequencies. Thus, TF is the number of occurrences of a term/word in a document normalized by dividing it by the total number of terms/words in the document. It is calculated as the ratio of term frequency in the document to the total number of terms.

Let *t* be a term/word, *d* be a document, and *w* be any term in the document, then we can calculate the frequency of the term/word *t* in document *d* as

$$tf(t,d) = \frac{f(t,d)}{f(w,d)},$$

where f(t,d) is the number of terms/words in the document d and f(w,d) is the number of all terms in the document.

When calculating the TF-IDF, the number of all documents in which the term/word occurs is also taken into account. We denote this number by df(t,D) - document frequency and express it as

$$df(t,D) = \sum (d \in D : t \in d),$$

where D is a corpus of all documents we work with.

Based on the document frequency, it is possible to calculate the Inverse Document Frequency, which expresses how common a term/word is in the corpus of documents. The more common the term, the lower its value. We calculate it as

$$idf(t,D) = \log \frac{N}{df(t,D)+1}$$

where N is a number of document in corpus and df(t,D) is the already mentioned document frequency. The value of +1 in the formula is due to the division by zero treatment, since the document frequency can also be 0.

The mentioned formulas represent partial calculations for the calculation of TF-IDF (Term frequency – inverse document frequency), which determines how important the selected word is for a given document in the corpus of documents. We calculate the TD-IDF as

$$tfidf(t,d,D) = tf(t,d) \times idf(t,D)$$

that is, exactly according to its name as the term/word frequency multiplied by the inverse document frequency.

10.5.8

TF-IDF vectors are calculated using the **TfidfVectorizer** method of the **scikit-learn** library. Unlike the example for **CountVectorizer()** from the previous chapter, we need to consider the complete corpus of documents, i.e. all badges. For this reason, we also added the text of Princess Fiona's visiting. Most of the source code is practically the same as in the previous chapter, only the method for calculating the TF-IDF is replaced.

```
fiona = "heroism heroism heroism heroism envy envy"
shrek = "heroism heroism heroism heroism heroism heroism
heroism heroism heroism heroism envy envy"
charming = "heroism envy envy envy envy envy"

import pandas
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()

corpus = vectorizer.fit_transform([fiona, shrek, charming])

print(pandas.DataFrame(corpus.A,
columns=vectorizer.get_feature_names_out(), index=["fiona",
"shrek", "charming"]))
```

Program output:

```
              envy   heroism
fiona     0.447214  0.894427
shrek     0.196116  0.980581
charming  0.980581  0.196116
```

From the results, it is possible to easily find the TF-IDF vector, e.g. for Princess Fiona

$$\overrightarrow{fiona} = (0.894427\,;0.447214)$$

as well as other beings. Finally, please note that the TF-IDF calculation has several modifications and enhancements. In this chapter we presented the basic approach for TF-IDF calculation, while the **TfidfVectorizer** method contains an already enhanced approach. If we were to calculate the TF-IDF "manually" according to the given formulas, the results would be slightly different. In our example, e.g. the occurrence of two terms in all three documents (practically impossible in a reality) leads to a negative result of the logarithm. However, by TF-IDF enhancing, we would complicate the text comprehensibility and clarity. Our main goal was to present a simple and comprehendible approach to TF-IDF.

10.5.9

Project: Count TF-IDF

(by https://medium.com/nlplanet/two-minutes-nlp-learn-tf-idf-with-easy-examples-7c15957b4cb3)

TF-IDF was first designed to search for documents and retrieve information by running a query to find the most relevant documents. Suppose the query is the text "the error". The system would assign each document a higher score proportional to the frequency of the query words found in the document, weighing rarer words like "error" against more common words like "the".

Suppose we are searching for documents using the query Q and our database consists of documents D1, D2, and D3.

- Q: The cat.
- D1: The cat is on the mat.
- D2: My dog and cat are the best.
- D3: The locals are playing.

There are several ways to calculate TF, the simplest being the raw number of times a word appears in a document. We calculate the TF score using the ratio of the number of instances to the length of the document.

```
TF(word, document) = "number of occurrences of the word in the
document" / "number of words in the document"
    TF("the", D1) = 2/6 = 0.33
    TF("the", D2) = 1/7 = 0.14
    TF("the", D3) = 1/4 = 0.25
    TF("cat", D1) = 1/6 = 0.17
```

```
    TF("cat", D2) = 1/7 = 0.14
    TF("cat", D3) = 0/4 = 0
```

The IDF can be calculated by taking the total number of documents, dividing it by the number of documents that contain the word, and taking the logarithm. If the word is very common and occurs in many documents, this number will be close to 0. Otherwise, it will be close to 1.

```
IDF(word) = log("number of documents" / "number of documents
that contain the word")
    IDF("the") = log(3/3) = log(1) = 0
    IDF("cat") = log(3/2) = 0.18
```

Multiplying TF and IDF gives you the TF-IDF score of a word in a document. The higher the score, the more relevant the word is in that document.

```
TF-IDF(word, document) = TF(word, document) * IDF(word)
    TF-IDF("the", D1) = 0.33 * 0 = 0
    TF-IDF("the", D2) = 0.14 * 0 = 0
    TF-IDF("the", D3) = 0.25 * 0 = 0
    TF-IDF("cat", D1) = 0.17 * 0.18 = 0.0306
    TF-IDF("cat", D2) = 0.14 * 0.18 = 0.0252
    TF-IDF("cat", D3) = 0 * 0 = 0
```

The next step is to use the ranking function to rank the documents according to the TF-IDF scores of their words. We can use the average TF-IDF scores of the words in each document to obtain the rankings D1, D2, and D3 with respect to the query Q.

```
    Average TF-IDF of D1 = (0 + 0.0306) / 2 = 0.0153
    Average TF-IDF of D2 = (0 + 0.0252) / 2 = 0.0126
    Average TF-IDF of D3 = (0 + 0) / 2 = 0
```

Finally, when executing the query "The cat" over the document collection D1, D2, and D3, the results will be sorted:

1. D1: The cat is on the mat.
2. D2: My dog and cat are the best.
3. D3: The locals are playing.

The word "the" does not contribute to the TF-IDF score of each document. This is because "the" occurs in all documents and is therefore not considered a relevant word.

Document Similarity

Chapter 11

11.1 Introduction

📖 11.1.1

Document similarity refers to the measure of how alike two pieces of text are. It is widely used in various applications, such as detecting plagiarism, clustering similar documents, and recommending content. Document similarity is not only about matching exact words but also capturing semantic or contextual similarity.

For example, the sentences “The cat is on the mat” and “A feline sits on a rug” have different words but similar meanings. Measuring similarity between such sentences requires advanced techniques that go beyond simple word matching.

Common approaches to measuring document similarity include:

- Lexical similarity - measures the overlap of words between documents.
- Semantic similarity - considers the meaning of words using techniques like WordNet or embeddings.

Imagine searching for research papers about machine learning. A system that uses document similarity can recommend articles that discuss neural networks, even if the term “machine learning” is not explicitly mentioned in them.

📝 11.1.2

Which of the following statements about document similarity are correct?

- It measures how alike two pieces of text are.
- It can involve both lexical and semantic approaches.
- It is only useful for matching exact words.
- It is irrelevant for document clustering.

📖 11.1.3

Lexical similarity

Lexical similarity compares documents based on the overlap of their words. This method is straightforward and works well for applications where exact word matching is sufficient. For example, comparing two product descriptions to check if they mention the same features.

Key techniques:

1. **Jaccard similarity** - measures the ratio of the intersection of words to the union of words.

- Example: For sets **A = {apple, orange}** and B = **{apple, banana}**, Jaccard similarity is:

```
J(A,B)=Intersection / Union = 1 / 3
```

- **Cosine similarity** calculates the cosine of the angle between two vectors (e.g., Bag of Words). Will be explained later.

For two sentences:

- "I like apples and bananas."
- "I enjoy apples."

Lexical similarity methods will focus only on matching the exact words like apples.

11.1.4

Which techniques are examples of lexical similarity?

- Cosine similarity
- Jaccard similarity
- Neural embeddings
- WordNet-based similarity

11.1.5

Semantic similarity

Semantic similarity goes beyond word matching and considers the meaning of words and their context. It is essential for cases where two texts share similar ideas but use different words.

Key methods

- WordNet-based similarity uses a lexical database to compute the closeness of word meanings. Example: The words car and automobile are semantically similar in WordNet.
- Word embeddings represent words as vectors in a high-dimensional space, capturing their meanings. Example: In word embeddings, the words king and queen are close in the vector space.

The sentences “The dog chased the cat” and “A canine pursued a feline” are semantically similar, even though they do not share exact words.

11.1.6

Which of the following are examples of semantic similarity methods?

- Cosine similarity of word embeddings
- WordNet-based similarity
- Jaccard similarity
- Bag of Words

📖 11.1.7

Cosine similarity

Cosine similarity is a commonly used method to measure the similarity between two documents. It calculates the cosine of the angle between two vectors representing the documents.

$$\text{Cosine Similarity} = \frac{A \cdot B}{\|A\|\|B\|}$$

Where **A** and **B** are term frequency vectors of the two documents.

Example: Consider two documents with Bag of Words vectors:

- Document 1: [2, 0, 1, 1]
- Document 2: [1, 1, 0, 1]

The cosine similarity is:

$$\frac{(2 \times 1 + 0 \times 1 + 1 \times 0 + 1 \times 1)}{\sqrt{(2^2 + 0^2 + 1^2 + 1^2)} \times \sqrt{(1^2 + 1^2 + 0^2 + 1^2)}}$$

Applications of similarity:

- Recommending similar products.
- Grouping similar articles in search engines.

📝 11.1.8

Which of the following statements about cosine similarity are correct?

- It is used to measure the similarity of document vectors.
- It is based on the angle between vectors.
- It calculates the intersection of words between documents.
- It cannot be used with Bag of Words.

📖 11.1.9

Applications of document similarity

Document similarity has numerous applications across industries and fields. Some common applications include:

- Plagiarism detection systems identify cases of copied content by comparing text similarity. Example: A university system that checks students' essays for similarity with existing papers.
- Recommendation systems suggest content (articles, products) based on textual similarity. Example: Recommending movies by analyzing their descriptions.
- Text clustering groups similar documents together for topic analysis. Example: Clustering news articles by topic (e.g., sports, politics).
- Information retrieval ranks search results by similarity to the query.

11.1.10

Which of the following are applications of document similarity?

- Plagiarism detection
- Clustering similar articles
- Tokenization
- Removing stop words

11.2 Application

11.2.1

Project: Find similar suitor

If the princess finds which of her suitors is similar to her, she can narrow down the selection of suitors (if she believes that "opposites" attract mutually, she can also narrow down the selection, only she will focus on the least similar suitors, from the point of view of the calculation, it is the same a task with a different result ranking according to similarity). Apart from the world of fairy tales, finding similarities between documents is, if not "every second", then certainly a daily task of many systems. A typical example is Internet search engines, which constantly search for documents most similar to the user's search query. However, this area is not the only one. Systems, e.g. check students´ works for plagiarism, naturally based on the similarity of the documents. Email servers verify and check spam based on the similarity between a message and messages labelled as spam in the past, hotel reservation services are constantly searching for offers similar to customer requests, and constantly "pushing" ads in the browser based on the similarity of the ad to your previous website visits is also a daily occurrence.

The basis for determining similarity is to design an appropriate object representation. Not only for documents, the most suitable representation is the form of vectors, which we described in the previous text.

Task: Princess Fiona wants to choose a suitor who is similar to her. She narrowed her choice to Shrek and Prince Charming. For the sake of simplicity, she only

wants to track the traits "envy" and "heroism". So she is looking for a suitor who has the same (similar) amount of envy and is a similar hero as she is.

The simplified measures of envy and heroism are shown in the following table, compiled by Princess Fiona. It was necessary for her to evaluate herself as well (she will be looking for someone most similar to her). She self-critically admitted that she is also "a little" envious

			
heroism	4	1	10
envy	2	5	2

We can insert the fairy creatures into the table using the Pandas library.

```
import pandas as pd
import numpy as np

beings = pd.DataFrame([
    {'being': 'Fiona', 'heroism': 4, 'envy': 2},
    {'being': 'Shrek', 'heroism': 10, 'envy': 2},
    {'being': 'Charming', 'heroism': 1, 'envy': 5},
])

print(beings)
```

Program output:

```
      being  heroism  envy
0     Fiona        4     2
1     Shrek       10     2
2  Charming        1     5
```

We will use the NumPy library to work and determine the vector similarity. NumPy is a basic library for computing. It is ideal for working with matrices and vectors. For this reason, we will do the following basic calculation of the similarity of fairy creatures, which will be represented by a vector. From the data we currently have loaded using Pandas, we can create three vectors representing fairy creatures.

```
fiona = np.array(beings.iloc[0, [1, 2]])
shrek = np.array(beings.iloc[1, [1, 2]])
charming = np.array(beings.iloc[2, [1, 2]])
```

```
print(fiona)
print(charming)
```

Program output:

```
[4 2]
[1 5]
```

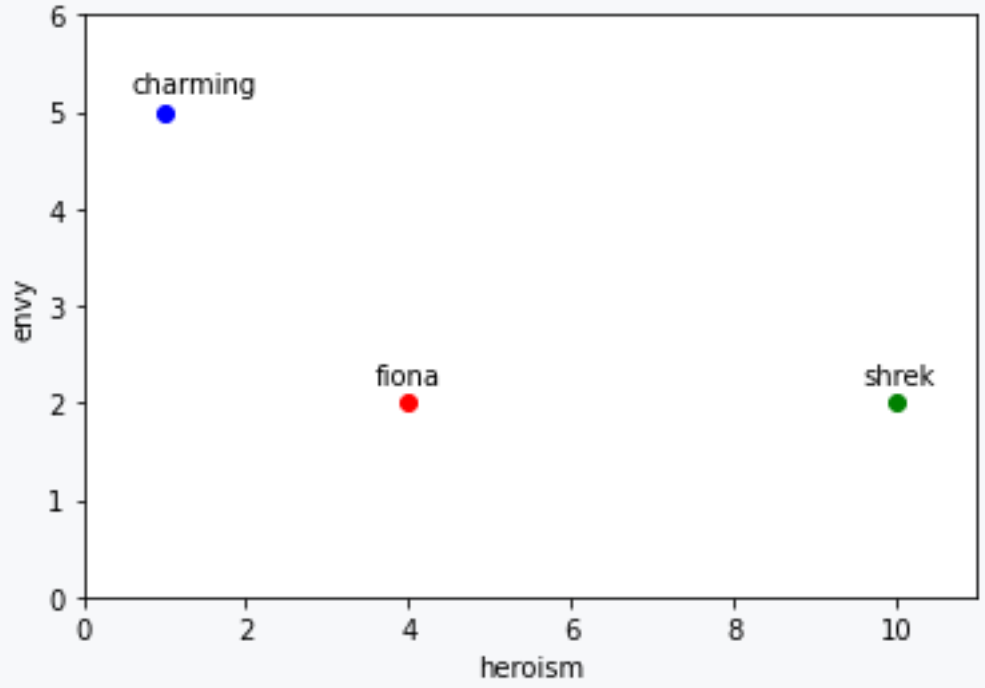
We assume that most real-world examples will have data in Pandas. If we do not use Pandas, we can also directly load the vectors of the three fairy creatures in NumPy.

```
fiona = np.array([4, 2])
shrek = np.array([10, 2])
charming = np.array([1, 5])
```

Whether we do the given operation directly by typing it into **NumPy** or by loading **NumPy** from **Pandas**, at the end of both procedures we will have three variables *fiona*, *shrek*, and *charming*, in which a vector representing the properties of fairy creatures will be stored. In our case, these are two-dimensional vectors, for better visualization we can plot them on a graph using the PyPlot library.

```
import warnings
warnings.filterwarnings("ignore")
import matplotlib.pyplot as plt

# Set axis labels for X and Y
plt.xlabel('heroism')
plt.ylabel('envy')

# Plotting the points on the graph
plt.plot(fiona[0], fiona[1], 'ro', color='r')
plt.plot(shrek[0], shrek[1], 'ro', color='g')
plt.plot(charming[0], charming[1], 'ro', color='b')

# Setting annotations for the points
plt.annotate('fiona', (fiona[0]-0.4, fiona[1]+0.2))
plt.annotate('shrek', (shrek[0]-0.4, shrek[1]+0.2))
plt.annotate('charming', (charming[0]-0.4, charming[1]+0.2))

# Setting the range of the displayed axes X and Y
plt.axis([0, 11, 0, 6])

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

Program output:



Even at the first look at the graph, you can see that Princess Fiona is closer to Charming than to Shrek. In our task, according to the assignment, we choose a suitor for Fiona who is most similar to her. "Similarity" is practically the basic building block for various data mining methods, such as recommendation system, object clustering, anomaly classification and detection, etc. The similarity measure is a metric indicating how similar or how far two objects are (distance). If the distance is small, it will be a high degree of similarity. A large distance means a low degree of similarity. Similarity is subjective and depends on the domain area of the issue being addressed.

The aim of our endeavour will be to find a number representing the relationship between two fairy creatures. We represented beings using vectors:

```
Fiona = (4,2)
Shrek = (10,2)
Charming = (1,5)
```

The proven procedure of mathematicians, how to express the "vector relationship" with a number, is to calculate the vector dot product. In this way, we have already calculated the similarity for the Boolean and vector models, the calculation was carried out according to the formula. The vector dot product is defined as the product of the absolute values of the magnitudes of two vectors and the cosine of the angle between them. The result is always a scalar.

We calculate the dot product of vectors $\vec{x}$ and $\vec{y}$ as:

$$\vec{x} \cdot \vec{y} = |\vec{x}||\vec{y}| \cos(\angle\alpha) = x_1y_1 + x_2y_2 + \cdots + x_ny_n, \quad (1)$$

where $|\vec{x}|$ and $|\vec{y}|$ is the mentioned size of the vector $\vec{x}$ and $\vec{y}$. For our three fairy creatures represented by vectors, the calculation is simple.

$$\overrightarrow{fiona} \cdot \overrightarrow{shrek} = fiona_1 shrek_1 + fiona_2 shrek_2 = 4 \cdot 10 + 2 \cdot 2 = 44$$

$$\overrightarrow{fiona} \cdot \overrightarrow{krason} = 4 \cdot 1 + 2 \cdot 5 = 14$$

It is similarly simple in Python, where we use the ***dot()*** function from the NumPy library to calculate the vector dot product.

```
fiona = np.array([4, 2])
shrek = np.array([10, 2])
charming = np.array([1, 5])

print("Similarity of fairy tale characters: \n",
      "Fiona and Shrek:", np.dot(fiona, shrek), "\n",
      "Fiona and Charming:", np.dot(fiona, charming))
```

Program output:

```
Similarity of fairy tale characters:
 Fiona and Shrek: 44
 Fiona and Charming: 14
```

11.2.2

Euclidean distance

The basic metric for determining similarity is the Euclidean distance. Euclidean distance is also known as simple distance. The Euclidean distance between two points is the length of the path connecting them.

We calculate the Euclidean distance m_e of vectors $\vec{x}$ and $\vec{y}$ as:

$$m_e(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2},$$

where $\vec{x}$ and $\vec{y}$ are vectors with the same number of elements.

In our example, we can easily make a function to calculate the Euclidean distance.

```
def euclidean_distance(x, y):
    return np.sqrt(np.sum((x - y) ** 2))
```

For the definition of the function, we used the NumPy library with the following functions:

	Description	Example
sqrt	The function is used to calculate the square root.	In [16]: `np.sqrt(81)` Out[16]: 9.0
sum	Function for the sum of individual elements of vectors	In [15]: `v = np.array([1,4,2])` `np.sum(v)` Out[15]: 7
**	Power operator of the given number	In [17]: `4 ** 3` Out[17]: 64 In [18]: `4 ** 2` Out[18]: 16

By calling the designed function **euclidean_distance()**, we can find the similarity (distance) between the fairy creatures. We remind that these beings are represented by a vector.

```
import numpy as np
fiona = np.array([4, 2])
shrek = np.array([10, 2])
charming = np.array([1, 5])

print("Euclidean distance: \n",
      "Fiona and Shrek:", euclidean_distance(fiona, shrek),
"\n",
      "Fiona and Charming:", euclidean_distance(fiona,
charming))
```

Program output:

```
Euclidean distance:
 Fiona and Shrek: 6.0
 Fiona and Charming: 4.242640687119285
```

From the results, it can be seen that Fiona is more similar to Charming, as the distance between the vector representing Charming and Fiona is smaller than between Fiona and Shrek.

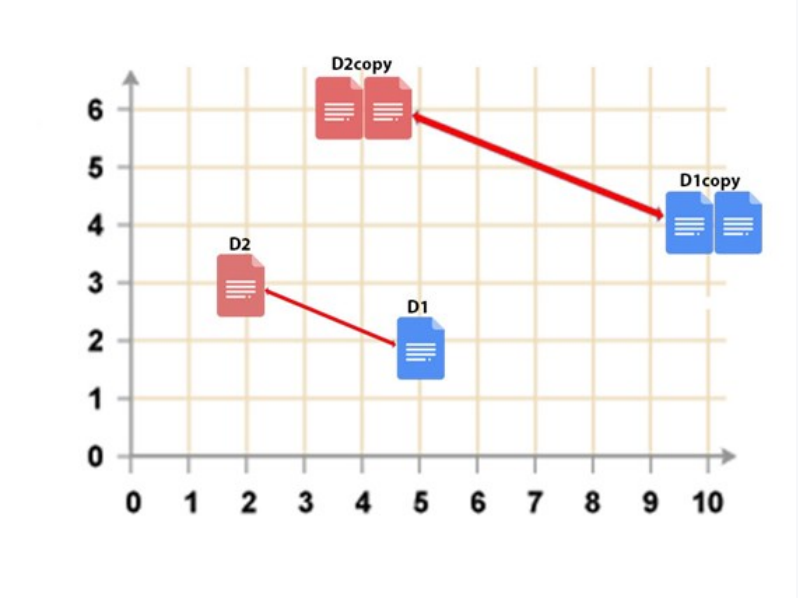
Although we "romantic souls" do not like this fact very much, it should be noted that any similarity measure will correctly determine the distance/similarity of objects. However, there are similarity measures that are more suitable for the studied domain. For the area of similarity of text documents (as well as vectors of fairy-tale creatures), there are more suitable similarity measures than the aforementioned Euclidean similarity.

11.2.3

Cosine similarity

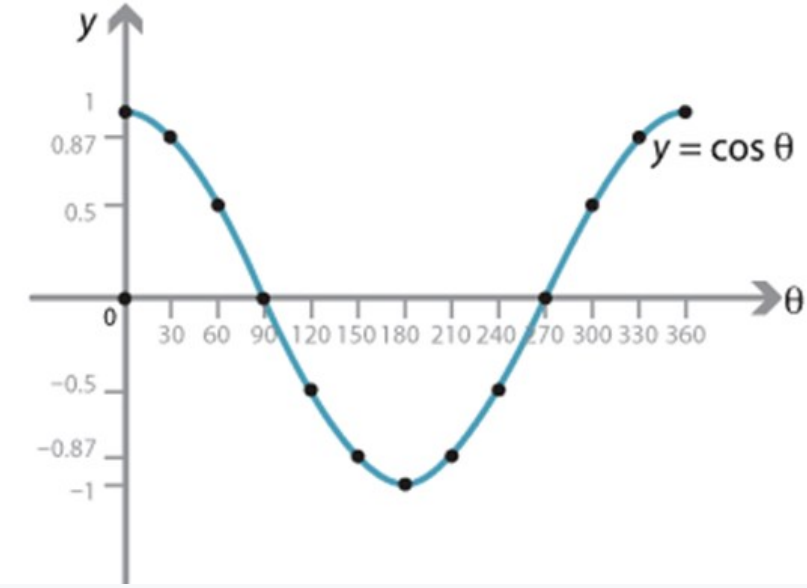
A commonly used vector text representation approach is based on the number of words in documents. However, this approach has one flaw. As the document size increases, the number of common words increases.

If we consider two documents D_1 and D_2. We would create a D_1copy document by only duplicating the content of the document (i.e. the D_1 document would be included twice). Similarly, we would also create a D_2copy document. In the vector representation of documents, the individual elements of the vector would be doubled in the D_1copy and D_2copy documents. With metrics like Euclidean distance, we would then find that D_1 and D_2 are closer to each other (more similar) than D_1copy and D_2copy. But that is not true.



Cosine similarity together with the TF-IDF model helps to overcome this fundamental error in the "word count" approach or Euclidean distance. Cosine similarity is a metric used to measure the similarity of documents regardless of their size. It is actually an expression of the size of the angle formed by two vectors representing two documents.

The cosine of 0° is 1, and is less than 1 for any other angle. So it is a view on the orientation and not the size of the vectors. Two vectors with the same orientation have a cosine similarity of 1, two vectors at 90° have a similarity of 0, and two diametrically opposite vectors have a similarity of -1, regardless of their sizes.



For two documents (vectors) $\vec{a}$ and $\vec{b}$ we calculate the cosine similarity as follows:

$$Cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \ \|\vec{b}\|} = \frac{\sum_1^n a_i b_i}{\sqrt{\sum_1^n a_i^2} \ \sqrt{\sum_1^n b_i^2}}$$

where: $\vec{a} \cdot \vec{b} = \sum_1^n a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$ is vector multiplication.

Then, in Python, we can calculate the cosine similarity as follows.

```
def cosine_similarity(x, y):
    return np.dot(x, y) / (np.sqrt(np.dot(x, x)) *
np.sqrt(np.dot(y, y)))
```

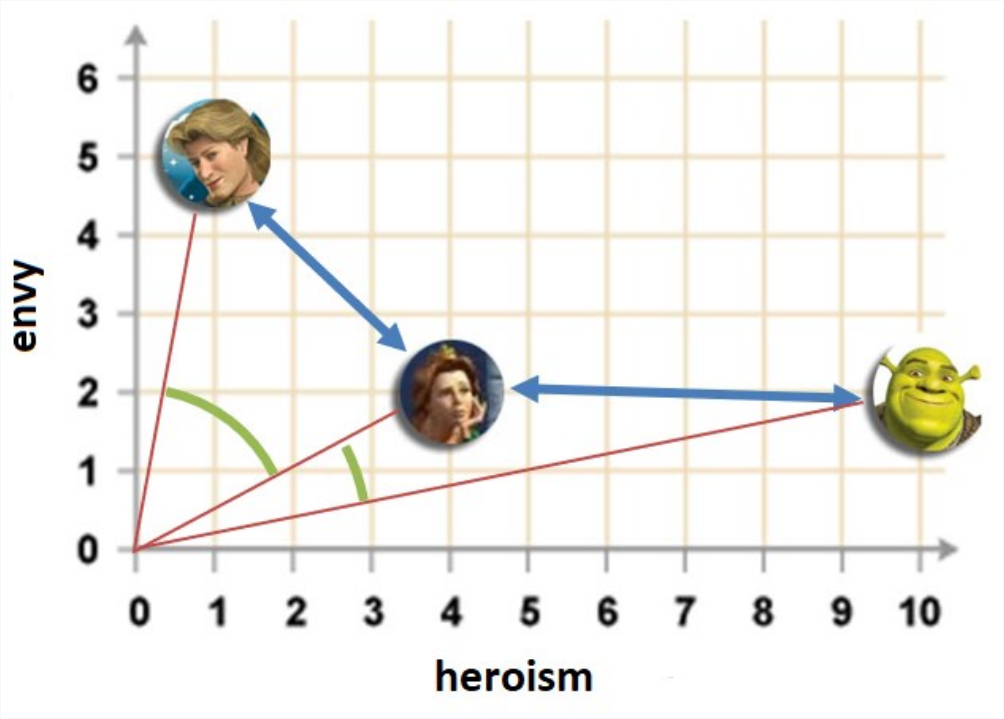
Finally, all that remains is to state the happy ending of the fairy tale of Shrek and Fiona.

```
import numpy as np
fiona = np.array([4, 2])
shrek = np.array([10, 2])
charming = np.array([1, 5])

print("Cosine similarity: \n",
      "Fiona and Shrek:", cosine_similarity(fiona, shrek),
"\n",
      "Fiona and Charming:", cosine_similarity(fiona,
charming))
```

Program output:

```
Cosine similarity:
 Fiona and Shrek: 0.9647638212377322
 Fiona and Charming: 0.6139406135149205
```

According to the cosine similarity, these two are certainly the closest to each other (a value of 1 represents a complete match), which is also illustrated in the following graph.



At the end of the chapter, it should be noted that the most common approach is the use of the TF-IDF document model in combination with cosine similarity. In practice, however, the calculation of the TF-IDF document model instead of the frequency vector model is only a matter of changing the method in a few lines of source code. For this reason, in the chapter we worked with a vector model of the word frequency in a document. We present the combination of TF-IDF and cosine similarity in the Application Example at the end of this publication.

Sentiment Analysis

Chapter 12

12.1 Introduction

📖 12.1.1

The internet has evolved significantly in the past 15 years, transforming users from mere consumers to active creators of content. This transformation, often termed "Web 2.0," marked a new era of web development where static content was replaced by dynamic platforms for sharing and collaboration. From 2004 onward, discussion forums, reviews, and user-generated content have become integral to online experiences, offering a wealth of information about users' opinions, attitudes, and feelings.

For instance, before planning a vacation, many people rely on reviews of hotels and destinations shared on popular travel websites. Negative feedback in these reviews can influence decisions, causing users to reconsider their choices. Similarly, reviews and ratings play a crucial role in purchasing decisions for goods and services. This growing repository of online opinions has become a rich source of insights into public sentiment.

Political parties, too, analyze these forums to gauge public reactions. By studying discussions, they can understand the public's reception of proposed policies or assess reactions to recent controversies. Such analysis provides invaluable feedback, but as the volume of user-generated content grows, manually analyzing this data becomes impractical.

📝 12.1.2

Which of the following are examples of the evolution of Web 2.0?

- Dynamic sharing and collaboration platforms.
- Online spaces for user-generated content.
- Static websites with fixed content.
- Single-purpose, read-only web pages.

📖 12.1.3

Sentiment analysis is a subfield of NLP that focuses on extracting subjective information from text. It aims to determine the sentiment, opinion, or emotion expressed in the text, often classifying it as positive, negative, or neutral. Sentiment analysis is widely applied in areas like product reviews, social media analysis, and customer feedback.

The method allows businesses to understand public perception of their products and services. For instance, analyzing customer reviews can help companies identify what customers like or dislike about their offerings. Similarly, social media sentiment analysis can provide insights into public reactions to current events or brand campaigns.

Sentiment analysis can be broadly categorized into two approaches: **lexicon-based methods** and **machine learning-based methods**. Lexicon-based methods rely on predefined lists of positive and negative words, while machine learning methods involve training models on labeled datasets to learn patterns in sentiment.

This field has gained prominence due to the rise of Web 2.0, where users actively create content on platforms like discussion forums, blogs, and social media. Analyzing this data enables better decision-making in marketing, politics, and other areas.

12.1.4

Which of the following are common applications of sentiment analysis?

- Analyzing customer reviews for opinions about products.
- Tracking public reaction to social media campaigns.
- Predicting numerical ratings of a product.
- Measuring the frequency of words in a text.

12.1.5

Lexicon-based sentiment analysis

Lexicon-based sentiment analysis is one of the simplest methods used to determine sentiment in text. This method involves using predefined lists of positive and negative words, often referred to as **sentiment lexicons**. For example, words like great, excellent, and amazing are positive, while words like poor, horrible, and terrible are negative.

A sentiment score is calculated by matching the words in a text with the lexicon and summing the positive and negative scores. For instance, the review "The movie was great, but the ending was horrible" would yield a mixed sentiment score based on the number of positive and negative matches.

This method is computationally efficient and easy to implement. However, it has limitations. For instance, lexicon-based methods may struggle with sarcasm or context. Consider the sentence: "Well, that was just fantastic, wasn't it?" The word fantastic is positive, but the tone conveys a negative sentiment.

Despite its simplicity, this approach is a great starting point for sentiment analysis tasks, particularly when datasets are small or when quick insights are needed.

12.1.6

What are the characteristics of lexicon-based sentiment analysis?

- It uses predefined lists of positive and negative words.
- It struggles with sarcasm and context.

- It requires a large labeled dataset for training.
- It is computationally intensive.

📖 12.1.7

Machine learning-based sentiment analysis

Machine learning-based sentiment analysis uses algorithms to learn patterns in text data. These methods rely on labeled datasets, where each text sample is annotated with its sentiment class (e.g., positive, negative, neutral). Models such as **logistic regression**, **support vector machines**, or **neural networks** are commonly used.

The first step is feature extraction. Text is converted into numerical formats, such as bag-of-words, TF-IDF, or word embeddings, to serve as input for machine learning models. Once trained, these models can generalize to predict sentiment in unseen text.

Machine learning approaches outperform lexicon-based methods in many scenarios, particularly when dealing with nuanced sentiment, such as mixed opinions or sarcasm. For example, models trained on movie reviews can accurately classify sentences like "The plot was predictable, but the characters were likable."

One challenge is the need for large, labeled datasets to train the models. However, pre-trained language models like BERT and GPT-3 have made it easier to achieve high accuracy without extensive labeled data.

📝 12.1.8

What are key steps in machine learning-based sentiment analysis?

- Extracting features from text data.
- Training models on labeled datasets.
- Annotating the dataset with numerical scores.
- Relying solely on predefined word lists.

📖 12.1.9

Challenges

While sentiment analysis has proven useful, it comes with challenges. One significant issue is understanding **context**. Words that are positive in one scenario can be neutral or even negative in another. For example, the word *cheap* is positive when describing prices but negative when describing quality.

Another challenge is dealing with **sarcasm**. Humans can easily detect sarcasm through tone and context, but models often misinterpret sentences like *"Oh great, another traffic jam."* Similarly, sentiment analysis may struggle with **ambiguous language**, where the author's intent isn't clear.

Domain-specific language also poses difficulties. Words and phrases that have specific meanings in certain industries, like *bullish* in finance, need specialized models or lexicons to be accurately analyzed.

Despite these challenges, advancements in deep learning and the use of pre-trained models are helping address these issues, improving accuracy in complex scenarios.

12.1.10

What are common challenges in sentiment analysis?

- Interpreting sarcasm in text.
- Handling ambiguous language.
- Converting text to numerical formats.
- Processing structured data formats.

12.1.11

Applications of sentiment analysis

Sentiment analysis has numerous real-world applications. In **customer feedback analysis**, businesses use sentiment analysis to understand customer satisfaction and improve products or services. Similarly, it helps monitor **brand reputation** by analyzing social media sentiment.

In **politics**, sentiment analysis is used to gauge public opinion about policies, speeches, or election campaigns. For instance, analyzing tweets during a political debate can provide insights into voters' reactions in real-time.

In **healthcare**, patient reviews of medications or hospitals are analyzed to assess quality and satisfaction. Additionally, sentiment analysis is applied in **financial markets**, where public sentiment about a company can influence stock prices.

These applications demonstrate how sentiment analysis is a powerful tool for decision-making across various domains, turning subjective data into actionable insights.

12.1.12

Which fields use sentiment analysis extensively?

- Customer feedback analysis.
- Brand reputation monitoring.
- Real-time stock price prediction.
- Diagnosing diseases directly from patient reviews.

12.2 Types of sentiment analysis

📖 12.2.1

Emotion detection

Emotion detection is a type of sentiment analysis focused on identifying the emotions behind text data. It aims to classify text into categories like frustration, happiness, or sadness.

This type of analysis uses predefined **emotion lexicons**—lists of words associated with specific emotions. For example, the word *angry* may indicate frustration, while *joyful* indicates happiness. Advanced methods may also use machine learning models to detect emotions based on patterns in the text.

Emotion detection is particularly useful in areas such as:

- **Customer support** - identifying frustrated customers for prompt assistance.
- **Social media monitoring** - understanding public sentiment on key topics.
- **Mental health** - analyzing discussions to identify signs of emotional distress.

Despite its usefulness, emotion detection can be challenging due to **ambiguous language** and **sarcasm**, where the true emotion may be masked.

📝 12.2.2

What is the primary focus of emotion detection in sentiment analysis?

- Identifying emotions like frustration or happiness.
- Classifying text into numerical scores.
- Detecting sarcasm directly.

📖 12.2.3

Intent detection

Intent detection focuses on uncovering the purpose or message behind a text. Every piece of communication has an underlying intent, and this type of sentiment analysis extracts that intent.

For example:

- A customer says, *"I need help with your product."* This indicates a request for technical support.
- Another says, *"I'm canceling my subscription."* This signals the intent to stop using the service.

Businesses often use intent detection to improve **customer interactions**. By recognizing customer intent in emails, chats, or social media messages, companies can tailor their responses accordingly.

Advanced techniques for intent detection involve NLU and machine learning models trained on datasets labeled with different intents.

📝 12.2.4

What is a common use of intent detection in sentiment analysis?

- Identifying customer complaints or requests.
- Measuring the frequency of keywords in text.
- Converting text into numerical representations.

📖 12.2.5

Fine-grained sentiment analysis

Fine-grained sentiment analysis goes beyond simple positive, negative, or neutral classification. It categorizes text on a detailed scale, often from 1 (extremely negative) to 5 (extremely positive).

This method provides a nuanced understanding of text sentiment. For instance:

- "The product is terrible" might score 1.
- "The product is good, but shipping was slow" might score 3.

Fine-grained sentiment analysis is widely used in:

- **Product reviews** - providing detailed feedback for improvement.
- **Social media** - gauging public reaction to events or campaigns.

This level of detail helps businesses identify not just sentiment but also its intensity, enabling targeted responses.

📝 12.2.6

What is the main advantage of fine-grained sentiment analysis?

- It provides detailed sentiment ratings like extremely negative or very positive.
- It focuses only on specific aspects of a text.
- It simplifies text into binary categories.

📖 12.2.7

Multi-lingual sentiment analysis

Multi-lingual sentiment analysis is designed to analyze text data written in multiple languages. It extracts emotional tones across different dialects, which is particularly useful for global brands or platforms with diverse users.

Challenges in multi-lingual analysis include:

- **Preprocessing** - tokenization and stop word removal often need to be adapted for each language.
- **Linguistic diversity** - grammar and syntax vary widely between languages, making it difficult to apply uniform techniques.

Despite these challenges, advanced NLP models like multilingual BERT (mBERT) and XLM-Roberta enable effective multi-lingual sentiment analysis.

📝 12.2.8

What is a challenge unique to multi-lingual sentiment analysis?

- Adapting preprocessing techniques to different languages.
- Converting text to numerical formats.
- Detecting sarcasm across languages.

📖 12.2.9

Aspect-based sentiment analysis

Aspect-based sentiment analysis identifies and analyzes sentiments tied to specific aspects of text rather than treating it as a whole.

For instance, in the review *"The food was delicious, but the service was slow,"*:

- The sentiment for *food* is positive.
- The sentiment for *service* is negative.

Applications of aspect-based sentiment analysis include:

- **Customer reviews** - understanding opinions about individual product features.
- **Social media analysis** - monitoring public sentiment on specific aspects of an event.

This method provides more actionable insights, as it allows businesses to pinpoint what customers like or dislike about specific aspects of their offerings.

12.2.10

What is the focus of aspect-based sentiment analysis?

- Analyzing sentiments about specific aspects in text.
- Categorizing sentiments on a detailed numerical scale.
- Detecting emotions like frustration or happiness.

12.3 Sentiment analysis approaches

12.3.1

Lexical-based sentiment analysis

Lexical-based sentiment analysis identifies text sentiment by categorizing each word into positive, neutral, or negative groups based on its polarity. The overall impression of the text is determined by the group with the highest number of words. For example, text with multiple negative words is labeled as negative.

Key features of the lexical-based approach:

- Tokenization converts words into tokens (sentiment lexicons) predefined with polarity.
- No training data required - it is an unsupervised method suitable for feature or sentence analysis.
- Challenges are in domain-dependence - words can have different polarities in different domains. For example, low weight is positive for laptop but negative for winter jacket.

The approach includes two methods:

- Corpus-based methods use domain-specific lexicons.
- Statistical methods rely on statistical analysis of text data.

12.3.2

Which of the following is a key characteristic of the lexicon-based approach to sentiment analysis?

- It categorizes words based on predefined polarity lists.
- It requires labeled training data for analysis.
- It uses machine learning algorithms to analyze text.
- It processes data without any predefined rules.

12.3.3

What is a key challenge of the lexicon-based approach?

- Polarity of words may vary by domain.
- It requires labeled training data.
- It cannot analyze at the sentence level.

12.3.4

How does the lexicon-based approach determine the sentiment of a text?

- By counting the number of positive and negative words in the text.
- By analyzing the grammatical structure of the text.
- By using machine learning to classify the sentiment.
- By translating the text into a sentiment-specific language.

12.3.5

Machine learning approach

The machine learning approach is one of the most effective and widely used methods for sentiment analysis. It employs supervised or unsupervised learning to train models that can predict the polarity of text (positive, negative, or neutral) based on patterns learned from the training data. This approach leverages algorithms to analyze linguistic features and patterns, enabling the model to determine sentiment with high accuracy.

Supervised learning requires labeled datasets where each text sample is annotated with its corresponding sentiment. During training, the model learns to associate specific patterns, words, or combinations of words with their respective sentiment labels. For example, words like "amazing" or "excellent" might be associated with positive sentiment, while words like "terrible" or "poor" would correspond to negative sentiment. In contrast, unsupervised learning does not rely on labeled data but uses clustering or other techniques to infer sentiment patterns.

Several machine learning algorithms are commonly employed in sentiment analysis:

- **Support vector machine** is highly effective for text classification tasks. It works by finding the optimal boundary that separates different sentiment classes.
- **K-nearest neighbors** classifies text based on the similarity to nearby examples in the dataset, making predictions based on the sentiment of its closest neighbors.
- **Logistic regression** predicts the probability of a text belonging to a specific sentiment class, often used due to its simplicity and interpretability.

- **Decision tree** splits the data into branches based on features, creating a tree-like structure to classify sentiment.
- **Naive Bayes** as a probabilistic classifier that assumes independence between features, Naive Bayes is lightweight and effective for many sentiment analysis tasks.

The machine learning approach has several advantages. It is capable of handling large datasets efficiently and can learn complex patterns and relationships in the data. This makes it a versatile tool for analyzing large-scale sentiment data, such as social media posts or customer reviews. Additionally, machine learning models can adapt to new patterns when retrained with updated data, improving their performance over time.

However, there are some limitations to this approach. In supervised learning, the model's success depends heavily on the availability and quality of labeled data. Creating such datasets can be time-consuming and expensive. Additionally, machine learning models may struggle with domain-specific language or nuanced expressions unless they are trained on sufficient examples from that specific domain.

In summary, the machine learning approach is a powerful technique for sentiment analysis, offering flexibility and accuracy across diverse datasets. By selecting the right algorithm and ensuring high-quality training data, organizations can harness the full potential of this approach for various applications.

12.3.6

What is a benefit of the machine learning approach for sentiment analysis?

- It can handle large datasets and learn patterns.
- It eliminates the need for labeled training data.
- It automatically generates lexicons.

12.3.7

Neural network approach

The neural network approach leverages artificial neural networks, which are computational models inspired by the human brain, to process and classify text data. This method excels in tasks requiring complex pattern recognition, such as sentiment analysis. Neural networks can understand the sequential nature of text and capture dependencies between words, making them highly effective for analyzing sentiment nuances.

Popular algorithms in this category include **Recurrent Neural Networks (RNNs)**, which are adept at handling sequential data, and **Convolutional Neural Networks (CNNs)**, which are often used for feature extraction in text. These algorithms process text by encoding it into numerical representations, which are then analyzed

for patterns indicating sentiment. For instance, an RNN might analyze the phrase "not great but not terrible" by considering the sequence and context of words, something simpler models may overlook.

Neural networks require extensive labeled datasets and significant computational power for training. However, once trained, they can handle diverse data sources and even identify subtle shifts in sentiment, such as sarcasm or irony. This makes them particularly valuable in analyzing social media data, customer reviews, or any content with rich emotional subtext.

The neural network approach represents a significant advancement in sentiment analysis, combining high accuracy with the flexibility to adapt to various applications. However, its complexity and computational demands make it less accessible than simpler methods like lexicon-based analysis.

12.3.8

What are characteristics of the neural network approach to sentiment analysis?

- It uses artificial neural networks to process and analyze text data.
- Models like RNNs and CNNs are commonly employed for this approach.
- It relies primarily on predefined sentiment lexicons for analysis.
- It avoids using machine learning techniques altogether.

12.4 Practical examples

12.4.1

Project: Positive and negative words

A probably simpler method of sentiment analysis is based on the assumption that in the case of a positive review we also use the so-called positive words. If we like the selected product or service, we will probably write words like "good", "excellent", "perfect" etc. in our review. With a negative attitude, we will probably use words like "bad", "terrible", "horrible" and so on. Therefore, a basic sentiment analysis is based on the assumption of the existence of lists of such positive and negative words. For the English language, it is not difficult to find lists of positive and negative words on the Internet.

Task: Princess Fiona wants to find how she, as a person and ruler, is perceived by her subjects. Due to the potential threat to the kingdom's sovereignty, Facebook, Instagram, Twitter, VKontakte, and TikTok are banned and blocked in the kingdom. The only way how subjects can write a review or a contribution to the discussion is through the discussion forum on the official website of the Kingdom beyond the Seven Mountains, which is monitored and censored.

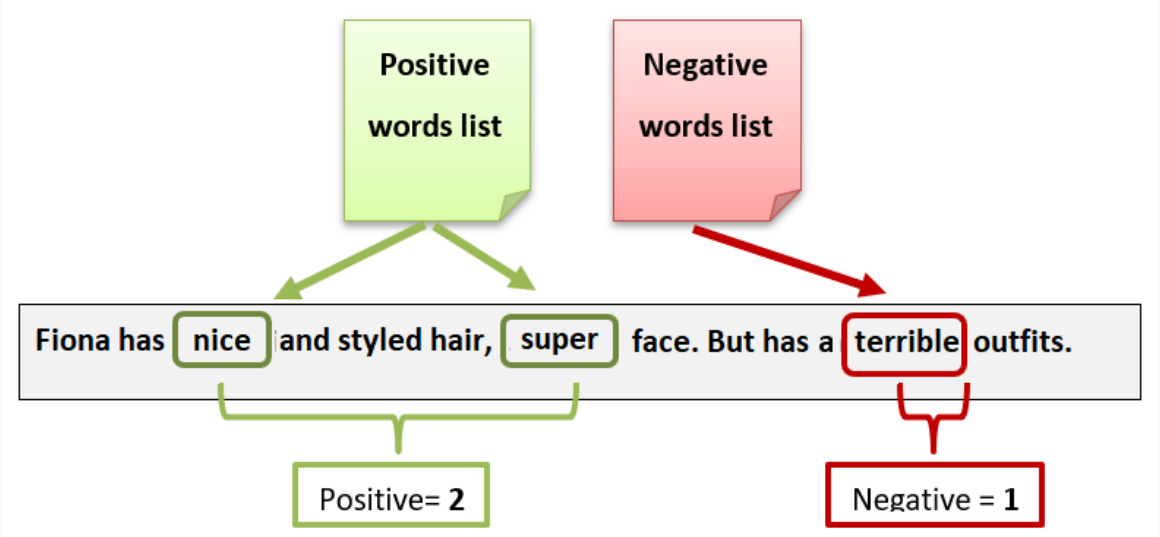
From all the ratings and reviews of her subjects, posted on the official website of the Kingdom beyond the Seven Mountains, Fiona wants to evaluate and analyse the sentiment. We all already suspect that it will be positive.

The first step of the task will be to prepare and pre-process all reviews/posts posted on the Kingdom beyond Seven Mountains site. If these will be in the form of texts, it is necessary to select only posts evaluating Princess Fiona. This step will also be easy, enough if the word "Fiona" occurs in the post. When analysing sentiment, we need to evaluate the sentiment of respective posts. The resulting sentiment will represent their average, i.e. average of sentiments of individual discussion/review posts.

Determining the sentiment of a post or a sentence represents a classification task. Therefore, it is necessary to note whether the sentence is positive, neutral or negative. We suppose the existence of positive and negative words lists. Our lists of these words might look as follows:

Positive words list	Negative words list
good, nice, super, excellent, wonderful, ...	bad, terrible, horrible, ...

In the analysis itself, it is enough to count the number of negative and positive words occurred in the post. As an example, we will classify sentiment for the sentence: "Fiona has nice and styled hair, super makeup. However, she wears terrible outfits."



However, simple classification of sentences into these groups is rarely used. Rather, the classifier is expected to determine the magnitude of positivity or negativity. Then we talk about determining the so-called sentence orientation or polarity.

In the classifier we are creating for Princess Fiona, for the sake of simplicity, we will create our own simple list of positive and negative words. Along with importing the **nltk** library it might look as follows:

```
from nltk.tokenize import word_tokenize

positive = ['good', 'nice', 'super', 'excellent']
negative = ['bad', 'terrible', 'horrible']
```

We tokenize the sentence and insert the created tokens, in our case words, into the list.

```
sentence = 'Fiona has nice and styled hair, super makeup.
However, she wears terrible outfits.'
sentence_token = word_tokenize(sentence)
print(sentence_token)
```

Program output:

```
['Fiona', 'has', 'nice', 'and', 'styled', 'hair', ',',
'super', 'makeup', '.', 'However', ',', 'she', 'wears',
'terrible', 'outfits', '.']
```

We ensure the number of positive and negative words from our lists by a simple cycle in which we check whether the word is in these lists.

```
pos = 0
neg = 0
for word in sentence_token:
    if word in positive:
        pos = pos + 1
    if word in negative:
        neg = neg + 1
print('Number of positive words: ', pos)
print('Number of negative words: ', neg)
```

Program output:

```
Number of positive words:  2
Number of negative words:  1
```

It is clear from the result that the sentence is positive. We can even express a degree of positivity. For the sake of completeness, we also present this relatively simple source code.

```
difference = pos - neg
if(difference > 0):
    print('The sentence is positive')
```

```
    positivity_rate = pos / (pos + neg)
    print('Positivity rate: ', positivity_rate)
if(difference < 0):
    print('The sentence is negative')
    negativity_rate = neg / (pos + neg)
    print('Negativity rate: ', negativity_rate)
if(difference == 0):
    print('Neutral sentence')
```

Program output:

```
The sentence is positive
Positivity rate:  0.6666666666666666
```

The given classification of sentiment is quite simple. It has several issues:

- There must be a list of positive and negative words. If no existence, a more sophisticated solution than just "simple thinking and reasoning which words should be included in" has to be designed.
- Words have varying degrees of sentiment. It would certainly be good to rate more, e.g. "she has **the best** outfits in the kingdom" than "she has **good** outfits". Similarly, in the case of negative sentiment, there is a difference in the degree of sentiment in the sentences "Her outfits are terrible" and "Her outfits are not good". Then, how to assign the weight of sentiment to respective words?
- Sometimes only words are not enough. For example, the word "good" can be used in a positive sentence such as "She has **good** outfits.", but also in the sentence "Her outfits are not **good**." The second sentence is obviously no longer positive, despite the fact that it contains one positive word.
- Arguably the biggest issue with sentiment analysis (not just our simple classifier) is irony. The traditional sentence of teenagers "well... it's really **good**" probably will not be positive despite the positive word used.

12.4.2

Project: Sentiment analysis on Amazon reviews

(by https://www.kaggle.com/code/anantpandey29/sentiment-analysis-on-amazon-reviews)

Show how sentiment analysis can be performed using python.

Dataset:

- original: https://www.kaggle.com/code/anantpandey29/sentiment-analysis-on-amazon-reviews?select=amazon.xlsx
- reduced: https://priscilla.fitped.eu/data/nlp/sentiment/amazon_reviews.csv

```
!pip install textblob
```

TextBlob is a popular NLP library in Python, designed for simplicity and ease of use. It provides tools to work with text, such as processing, analysis, and sentiment analysis. Components does:

TextBlob

The TextBlob class is used to represent text and provides a wide range of functionality for text processing. We can create a TextBlob object to work with text data and perform tasks like:

- Tokenization (splitting text into words or sentences)
- Sentiment analysis (analyzing text for positive, negative, or neutral sentiment)
- Part-of-speech tagging (identifying nouns, verbs, etc.)
- Text translation and spelling correction

Word

The Word class represents individual words and provides word-specific functionalities, such as:

- Lemmatization (getting the base form of a word)
- Spell checking and correction
- Pluralization and singularization

1. Import libraries and data

```
import numpy as np
import pandas as pd

from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_val_score,
GridSearchCV, cross_validate
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import MultinomialNB

import nltk
from nltk.corpus import stopwords
from nltk.sentiment import SentimentIntensityAnalyzer
from textblob import Word, TextBlob
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
```

```
import matplotlib.pyplot as plt
import plotly.graph_objs as go
import plotly.io as pio
from wordcloud import WordCloud

df =
pd.read_csv("https://priscilla.fitped.eu/data/nlp/sentiment/am
azon_reviews.csv")
print(df.head())
```

Program output:

```
/home/johny/.local/lib/python3.9/site-
packages/matplotlib/projections/__init__.py:63: UserWarning:
Unable to import Axes3D. This may be due to multiple versions
of Matplotlib being installed (e.g. as a system package and as
a pip package). As a result, the 3D projection is not
available.
  warnings.warn("Unable to import Axes3D. This may be due to
multiple versions of "
   Star  HelpFul
Title  \
0     5        0                                      looks
great
1     5        0  Pattern did not align between the two
panels.
2     5        0               Imagery is stretched. Still
fun.
3     5        0                 Que se ven elegantes muy
finas
4     5        0                            Wow great
purchase


                                          Review
0                                    Happy with it
1  Good quality material however the panels are m...
2  Product was fun for bedroom windows.

Imag...
3   Lo unico que me gustaria es que sean un poco ...
4  Great bang for the buck I can't believe the qu...
```

2. Data preprocessing

- Checking for nulls in the dataset

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

Program output:

```
Star         0
HelpFul      0
Title       52
Review      18
dtype: int64
```

- Since 18 reviews are null, we will drop them from the dataset. The other columns will remain unchanged, as our primary focus in this notebook is on Sentiment Analysis.

```
df.dropna(subset=['Review'], inplace=True)
print(df.isnull().sum())
```

Program output:

```
Star         0
HelpFul      0
Title       43
Review       0
dtype: int64
```

- Applying stop words from NLTK

```
# nltk.download('stopwords')
sw = stopwords.words('english')
print(sw)
```

Program output:

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves',
'you', "you're", "you've", "you'll", "you'd", 'your', 'yours',
'yourself', 'yourselves', 'he', 'him', 'his', 'himself',
'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its',
'itself', 'they', 'them', 'their', 'theirs', 'themselves',
'what', 'which', 'who', 'whom', 'this', 'that', "that'll",
'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be',
'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does',
'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or',
'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for',
'with', 'about', 'against', 'between', 'into', 'through',
'during', 'before', 'after', 'above', 'below', 'to', 'from',
'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under',
'again', 'further', 'then', 'once', 'here', 'there', 'when',
'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few',
'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not',
```

```
'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't',
'can', 'will', 'just', 'don', "don't", 'should', "should've",
'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren',
"aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn',
"doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven',
"haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't",
'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't",
'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "weren't",
'won', "won't", 'wouldn', "wouldn't"]
```

The following preprocessing has been applied to the "Review" column.

- Lowercasing
- Non-word character removal
- Digits removal
- Stop word removal

```
df['Review'] = df['Review'].str.lower()
df['Review'] = df['Review'].str.replace('[^\w\s]', '')
df['Review'] = df['Review'].str.replace('\d', '')
df['Review'] = df['Review'].apply(lambda x: " ".join(x for x
in str(x).split() if x not in sw))
```

We will find rare words in your dataset and store them separately in a temporary DataFrame,

- The goal of the following code is to remove rare words (those that occur only once or very infrequently) from the 'Review' column in the df DataFrame.
- By doing this, we reduce noise in the text data, which can be particularly useful for our task, where rare words might not contribute meaningfully to the analysis and could even distort the model's understanding of the text.

```
temp_df = pd.Series('
'.join(df['Review']).split()).value_counts()
print(temp_df)
```

Program output:

```
love                1271
curtains            1251
like                1017
look                 818
great                721
                    ...
inserts                1
months.
```

```
stood       1
bleak                   1
requested               1
studio/living           1
Name: count, Length: 10606, dtype: int64
```

```
drops = temp_df[temp_df <= 1]
df['Review'] = df['Review'].apply(lambda x: " ".join(x for x
in x.split() if x not in drops))
```

Lemmatization

- This code performs lemmatization on the text in the 'Review' column of the DataFrame df. It first downloads the necessary NLTK resources for lemmatization (wordnet and omw-1.4). Then, for each review, it splits the text into words, applies lemmatization (reducing words to their base form using Word(word).lemmatize()), and joins the words back into a string, ensuring that missing or NaN values are replaced with an empty string.

```
nltk.download('wordnet')
nltk.download('omw-1.4')
df['Review'] = df['Review'].apply(lambda x: "
".join([Word(word).lemmatize() for word in str(x).split()]) if
pd.notna(x) else '')
print(df.head())
```

Program output:

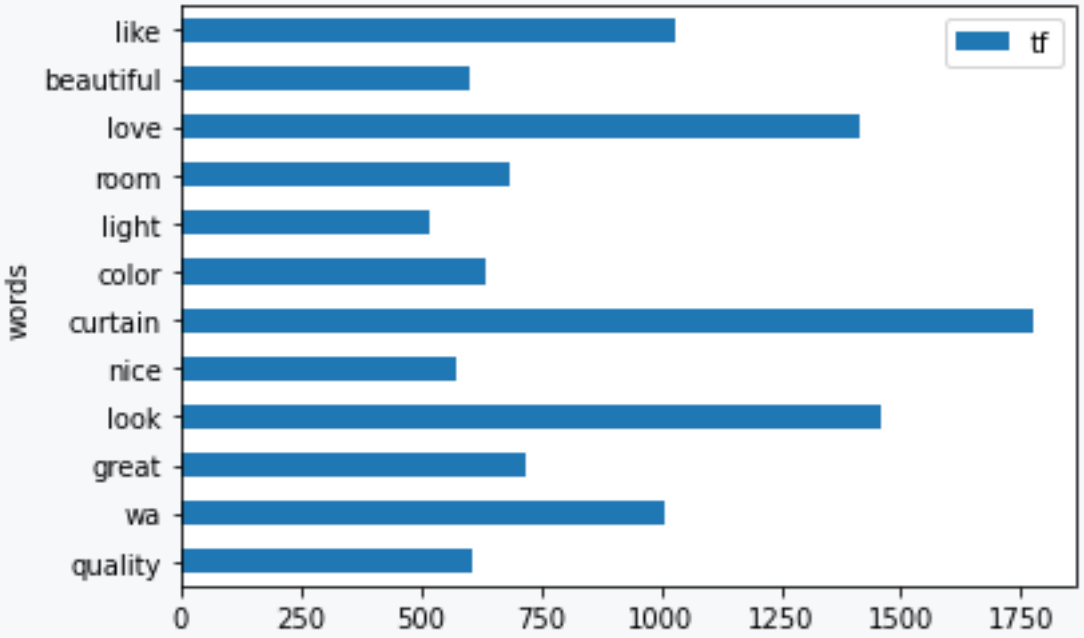
```
[nltk_data] Downloading package wordnet to
/home/johny/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to
/home/johny/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
   Star  HelpFul
Title  \
0     5        0                                    looks
great
1     5        0  Pattern did not align between the two
panels.
2     5        0               Imagery is stretched. Still
fun.
3     5        0                 Que se ven elegantes muy
finas
4     5        0                            Wow great
purchase
```

```
                                                Review
0                                                happy
1                  good quality material however panel
2  product fun bedroom windows.

<="" pre="">
3. Exploratory data analysis
Exploratory Data Analysis (EDA) in NLP involves analyzing text
data to discover patterns, trends, and anomalies. It helps
data scientists and NLP practitioners understand the dataset
better before moving on to more complex tasks like text
classification, sentiment analysis, or machine translation.
Through both visual and statistical methods, EDA provides
valuable insights into the text data.
Extracting term fFrequencies
Term frequency (TF) is a measure used in NLP to count how
often a word or phrase appears in a document or a collection
of documents. It's a basic but important concept in text
mining and information retrieval. TF is usually calculated by
dividing the number of times a word appears by the total
number of words in the document.
TF(t,d) = Number of times term t appears in document d /
Total number of terms in document d

tf = df["Review"].apply(lambda x: pd.Series(x.split("
")).value_counts()).sum(axis=0).reset_index()
print(tf.head())
```

Program output:

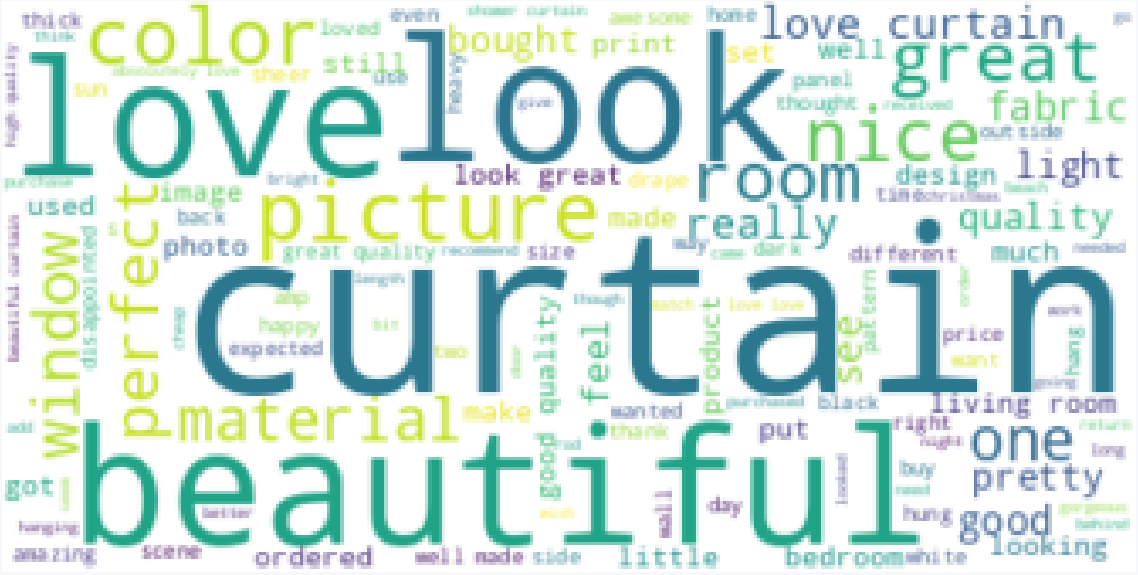
```
      index      0
0     happy  171.0
1      good  453.0
2   quality  609.0
3  material  398.0
4   however   36.0

sort data by frequences

tf.columns = ["words", "tf"]
tf.sort_values("tf", ascending=False)
Drawing term frequencies

tf[tf["tf"] > 500].plot.barh(x="words", y="tf")
plt.show()
```

Program output:




```
Prepare data to wordcloud visualising:

text = " ".join(i for i in df.Review)

wordcloud = WordCloud(background_color="white").generate(text)

# Display the word cloud
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.show()
```

Program output:



```
4. Sentiment analysis

print(df["Review"].head())
```

Program output:

```
0                                                   happy
1                  good quality material however panel
2    product fun bedroom windows.

<="" pre="">
Initializing VADER SIA
VADER (Valence Aware Dictionary and sEntiment Reasoner) is a
lexicon and rule-based sentiment analysis tool specifically
designed for analyzing the sentiment of text in the context of
social media and informal online text. It was developed by
researchers at the Georgia Institute of Technology.
The output will include scores for positivity, neutrality, and
negativity, as well as a compound score that summarizes the
overall sentiment of the text.
nltk.download('vader_lexicon')
```

Program output:

```
[nltk_data] Downloading package vader_lexicon to
[nltk_data]     /home/johny/nltk_data...
[nltk_data]   Package vader_lexicon is already up-to-date!
and to try
sia = SentimentIntensityAnalyzer()
print(sia.polarity_scores("The film was awesome"))
```

Program output:

```
{'neg': 0.0, 'neu': 0.423, 'pos': 0.577, 'compound': 0.6249}

print(sia.polarity_scores("I liked this music but it is not
good as the other one"))
```

Program output:

```
{'neg': 0.207, 'neu': 0.666, 'pos': 0.127, 'compound': -0.298}

print(df["Review"][0:10].apply(lambda x:
sia.polarity_scores(x)))
```

Program output:

```
0    {'neg': 0.0, 'neu': 0.0, 'pos': 1.0, 'compound...
1    {'neg': 0.0, 'neu': 0.58, 'pos': 0.42, 'compou...
2    {'neg': 0.0, 'neu': 0.571, 'pos': 0.429, 'comp...
```

```
3    {'neg': 0.0, 'neu': 1.0, 'pos': 0.0, 'compound...
4    {'neg': 0.0, 'neu': 0.495, 'pos': 0.505, 'comp...
5    {'neg': 0.0, 'neu': 1.0, 'pos': 0.0, 'compound...
6    {'neg': 0.0, 'neu': 0.631, 'pos': 0.369, 'comp...
7    {'neg': 0.0, 'neu': 0.541, 'pos': 0.459, 'comp...
8    {'neg': 0.0, 'neu': 1.0, 'pos': 0.0, 'compound...
9    {'neg': 0.0, 'neu': 0.753, 'pos': 0.247, 'comp...
Name: Review, dtype: object

df["polarity_score"] = df["Review"].apply(lambda x:
sia.polarity_scores(x)["compound"])

5. Feature engineering
Adding Sentiments from the generated scores.
df["Review"][0:10].apply(lambda x: "pos" if
sia.polarity_scores(x)["compound"] > 0 else "neg")

df["sentiment_label"] = df["Review"].apply(lambda x: "pos" if
sia.polarity_scores(x)["compound"] > 0 else "neg")
print(df["sentiment_label"].value_counts())
```

Program output:

```
sentiment_label
pos    4787
neg     806
Name: count, dtype: int64

print(df.head())
```

Program output:

```
   Star  HelpFul                                        Title  \
0  5     0                                        ooks great
1  5     0  Pattern did not align between the two panels.
2  5     0               Imagery is stretched. Still fun.
3  5     0                 Que se ven elegantes muy finas
4  5     0                             Wow great purchase

                                  Review  polarity_score  \
0                                  happy          0.5719
1     good quality material however panel          0.4404
2  product fun bedroom windows.

<="" pre="">
df = df.sample(frac = 1)
print(df.head())
```

Program output:

```
      Star  HelpFul                      Title  \
4034     5        0               Very pretty
3315     3        0            Cute but runny
1078     5        0  Brings the beach to you.
1538     5        0         Great decoration!
5511     5        0                      best

                                    Review  polarity_score  \
4034                love color came two window          0.6369
3315  cute got wet color window left open storm cat ...  0.4588
1078                 beautiful, remind islands.          0.5994
1538  hanging nail room salon ambiance. hung using h...  0.6588
5511               excellent product . advise everyone.  0.5719

     sentiment_label
4034             pos
3315             pos
1078             pos
1538             pos
5511             pos
Label encoding:
LabelEncoder().fit_transform(df["sentiment_label"]) is used to
convert the textual sentiment labels ("pos" and "neg") into
numeric labels (e.g., 1 for positive and 0 for negative). This
transformation is necessary because machine learning models
typically require numerical data as input.
Updating the DataFrame:
The encoded labels are then assigned back to the
"sentiment_label" column replacing the original text-based
labels with numeric values.
Feature and target separation:
y = df["sentiment_label"] assigns the numeric sentiment labels
(the target variable) to the variable y.
X = df["Review"] assigns the text data (the feature variable)
to the variable X. These two variables, X (features) and y
(target labels), will be used in training a machine learning
model for sentiment classification.
df["sentiment_label"] =
LabelEncoder().fit_transform(df["sentiment_label"])
print(df.head())

y = df["sentiment_label"]
X = df["Review"]
```

Program output:

```
      Star  HelpFul                      Title  \
4034     5        0               Very pretty
3315     3        0            Cute but runny
1078     5        0  Brings the beach to you.
1538     5        0         Great decoration!
5511     5        0                      best

                                                 Review
polarity_score  \
4034                         love color came two window
0.6369
3315  cute got wet color window left open storm cat ...
0.4588
1078                         beautiful, remind islands.
0.5994
1538  hanging nail room salon ambiance. hung using h...
0.6588
5511               excellent product . advise everyone.
0.5719

      sentiment_label
4034                1
3315                1
1078                1
1538                1
5511                1
```

Application of CountVectorizer()

This object cover a text vectorization method.
It converts a collection of text documents (in this case, the X **variable, which contains the text reviews) into a matrix of token counts.**
The goal is to represent text as numerical data, which is required for machine learning models to process it.

```
vectorizer = CountVectorizer()
X_count = vectorizer.fit_transform(X)
```

Application of TF-IDF and N-Gram

One of the most common ways to do this is through vectorization techniques like TF-IDF (Term Frequency-Inverse Document Frequency) **and** N-grams. **These techniques help capture the importance of words in a text document, which is essential for tasks like sentiment analysis or text classification.**
The TF-IDF method represents text data by evaluating the importance of words in a document relative to the entire

corpus. It assigns a weight to each word based on how frequently it appears in a document (term frequency, TF) and how common or rare it is across all documents (inverse document frequency, IDF). This helps highlight words that are important for distinguishing between documents.
N-grams can be used to extract and analyze phrases or idioms that are indicative of sentiment in text data.

```
tf_idf_word_vectorizer = TfidfVectorizer()
X_tf_idf_word = tf_idf_word_vectorizer.fit_transform(X)

tf_idf_ngram_vectorizer = TfidfVectorizer(ngram_range=(2, 3))
X_tf_idf_ngram = tf_idf_ngram_vectorizer.fit_transform(X)
```

In sentiment analysis, the goal is to classify text data (e.g., reviews) into different sentiment categories, typically positive or negative. One effective way to achieve this is to use a Multinomial Naive Bayes classifier, which is specifically suited for text classification tasks where features are represented by word count or frequency. The process typically involves the following steps:
The first step is to initialize the Multinomial Naive Bayes model, which is a probabilistic model based on Bayes' theorem. In this context, it assumes that the presence of a particular word in a document is independent of other words, given the class (positive or negative sentiment). This model is trained to estimate the probability that a document belongs to a certain class, based on the frequency of words (or features) in the document.
After initializing the model, the next step is to fit the model to the training data. This involves training a Naive Bayes classifier using features extracted from the text that have been transformed using techniques such as TF-IDF or N-gram vectorization. These vectorization methods convert text data into numerical representations that the model can work with. During the fitting process, the model learns the conditional probability of each word (or sequence of words) given the sentiment class (positive or negative).
After the model has been fitted, cross-validation is used to assess the performance of the model. Cross-validation splits the data into multiple subsets (in this case, 5 folds) and trains the model on different combinations of training and validation data. This process helps evaluate how well the model performs on different subsets of the data and ensures that the model is not over-fitted to a particular part of the

```
data. Cross-validation provides a more reliable estimate of
the model's performance.
Once a model is trained and validated, the model's
generalization is tested. This refers to the model's ability
to perform well on new, unseen data. The goal is for the model
to generalize from the patterns learned during training and
use them to predict sentiment on new data, not just the
training set. Generalization ensures that the model has
learned the underlying structure and is not simply memorizing
specific examples (overfitting). In practice, this is tested
by evaluating the model on a separate test set that it has not
seen before.
nb_model = MultinomialNB().fit(X_tf_idf_word, y)
nb_score = cross_val_score(nb_model,
                X_tf_idf_word,
                y,
                scoring="accuracy",
                cv=5).mean()
print(nb_score)
```

Program output:

```
0.8693009507274734
Testing generalizability
# introducing a positive review
new_review = pd.Series("this product is amazing")

# vectorize the our review
new_review = TfidfVectorizer().fit(X).transform(new_review)

# predict the sentiment of our review
print(nb_model.predict(new_review))
```

Program output:

```
[1]
# introducing a negative review
new_review = pd.Series("my experience was horrible")

# vectorize the our review
new_review = TfidfVectorizer().fit(X).transform(new_review)

# predict the sentiment of our review
print(nb_model.predict(new_review))
```

Program output:

```
[0]
```

12.4.3

Project: Custom classifier for Slovak language

The issues with the existence of a list of positive and negative words that are also assigned a sentiment weight (i.e. the problems mentioned in points a. and b.) can be solved by creating custom list of words together with determining the weight of the word. The existence of such a list of words will also mean for us the possibility of creating custom classifier.

To create a custom word list (as all successful classifiers do) training the classifier is required. It means that we will bring a sufficient amount of data (ratings, reviews) to the classifier, which will already have the correct weight assigned to sentiment. Based on these evaluations, the classifier will learn/train to correctly evaluate positive and negative reviews. Although we can imagine a very complicated process under the word "learn/train", the whole "learning" will consist in our case of creating a list of positive and negative words with the correct weight of sentiment.

On the official website of the Kingdom beyond the Seven Mountains, only a text review can be added. It is not possible to add a "like", "thumbs up" or other graphic rating.

However, the dissidents who emigrated from the Kingdom behind the Seven Mountains created the so-called "dark web" own rating system. It is free, cannot be canceled by censorship, and besides a rating, it allows to add a "thumbs up" or "thumbs down" to a post. This system can be used to train a classifier.

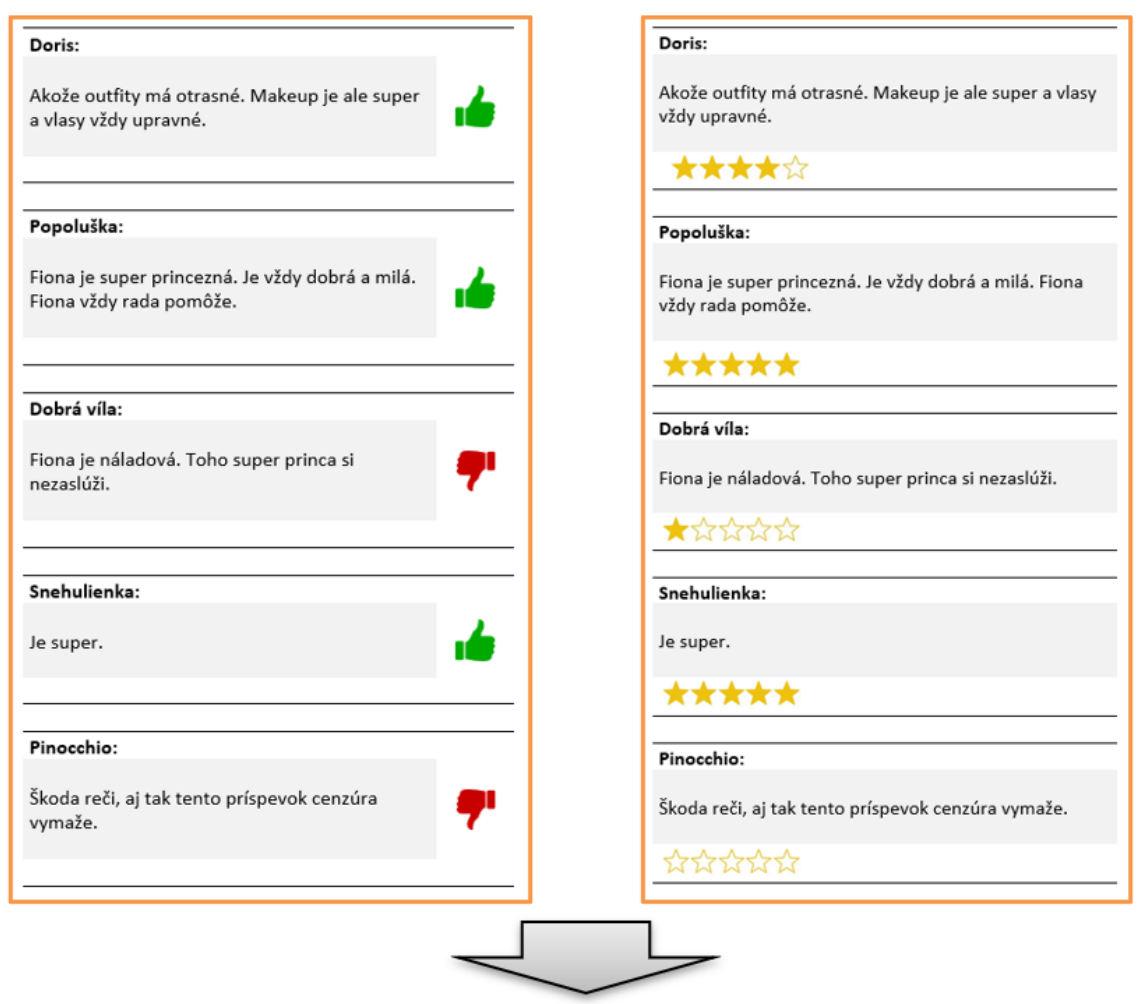
Assessments are used to create the classifier, which also includes a graphic representation of the assessment, such as "like", "thumbs up", or rating expressed by stars. The picture shows an example of two approaches to evaluation (thumbs up/down and stars). If there is any additional sentiment information, we can create an input file for custom classifier from such an evaluation.

The sample intentionally uses two different types of graphic sentiment information (thumbs up/down and stars). Both are easy to change in the input dataset to sentiment values of the message: positive/negative. For information about sentiment using stars, it can be done, for example, by a simple rule: if the number of stars is 3 or more, then the rating is positive, otherwise negative.

Assessments are used to create the classifier, which also includes a graphic representation of the assessment, such as "like", "thumbs up", or rating expressed by stars. The picture shows an example of two approaches to evaluation (thumbs up/down and stars). If there is any additional sentiment information, we can create an input file for custom classifier from such an evaluation.

The sample intentionally uses two different types of graphic sentiment information (thumbs up/down and stars). Both are easy to change in the input dataset to sentiment values of the message: positive/negative. For information about

sentiment using stars, it can be done, for example, by a simple rule: if the number of stars is 3 or more, then the rating is positive, otherwise negative.



Evaluator	Text	Sentiment
Doris	Akože outfity má otrasné. Makeup je ale super a vlasy vždy upravené.	pozitivna
Popoluška	Fiona je super princezná. Je vždy dobrá a milá. Fiona vždy rada pomôže.	pozitivna
Dobrá víla	Fiona je náladová. Toho super princa si nezaslúži.	negativna
Snehulienka	Je super.	pozitivna
Pinocchio	Škoda reči, aj tak tento príspevok cenzúra vymaže.	negativna
Mabel	To čo má za muža. Takúto hroznú ozrutu doniesla do kráľovstva.	negativna
Kyklop	Outfity čo nosí sú otrasné.	negativna
Merlin	Naša dobrá princezná je zároveň aj pekná.	pozitivna
Artuš	Je dobrá, môže byť.	pozitivna
Jack	Naposledy mala otrasný účes. Tá farba vlasov je tiež hrozná.	negativna

In our example, we will use a dataset of ratings with assigned sentiment information to train the classifier. We will use it to classify the text, i.e. other ratings. This may seem unusual at first glance. However, it is necessary to realize that sentiment analysis is often used for the analysis of discussion forums, or customer reactions, and the opinion of voters. There, graphic information in the form of thumbs up/down or stars is not used. Therefore, a classifier is often

trained, e.g. on the rating of products by customers from other portals, where there is also graphic sentiment information, which latter will be used for discussion forums.

To start creating custom classifier, we load the necessary libraries. The created dataset compiled from the ratings in the image, which will be in the file "sentiment_fiona.csv", will be loaded using the Pandas library.

```
!pip install stanza
```

```
import pandas
import stanza
stanza.download('sk')

nlp = stanza.Pipeline(lang='sk')
```

```
reviews =
pandas.read_csv('https://priscilla.fitped.eu/data/nlp/sentimen
t/sentiment_fiona.csv', sep=';', index_col=None)
print(reviews.head())
```

Program output:

```
                                                  Text
Sentiment
0  Fiona má pekné a upravené vlasy, super makeup....
pozitivna
1  Fiona je super princezná. Je vždy dobrá a milá...
pozitivna
2          Naša dobrá princezná je zároveň aj pekná.
pozitivna
3                                        Je super.
pozitivna
4                                Je dobrá, môže byť.
pozitivna
```

Especially in the case of inflected types of language, such as Slovak, it is necessary to perform text lemmatization. With this operation, we ensure the unification of words in terms of their "flexibility", i.e. from the point of view of word forms (e.g. the words "of students", "to students", "students" are rewritten to the uniform form "student").

We have loaded the dataset using Dataframe Pandas. Therefore, lemmatization will consist of loading a row in Pandas, applying the function for lemmatization, and inserting the resulting lemmas into a separate column. For this purpose, we will create the **get_lemmas()** function. The function rewrites individual words in the input text to their lemmas, i.e. the output are also sentences, but with words in

lemmatized form. Within Pandas, for each row/message, we need to display the message text, convert the words in the text to lemmas, and write it into a new column for the corresponding row.

There are several options for going through all rows in Pandas. We chose a simple loop through all the rows using the **iterrows()** method. Although this method is the slowest (you need to have a lot of patience with large datasets), we chose it for its simplicity and ease of understanding.

```
def get_lemmas(input_text):
    output = ''
    # on the input text, we apply NLP (using the Stanza
library) for morphological analysis
    document = nlp(input_text)

    # for all sentences in the result
    for sentence in document.sentences:
        # for all words
        for word in sentence.words:
            # we add the lemma of the given word to the output
            output = output + word.lemma + ' '
    return output
```

```
# In the loop, we assume that there is an existing column
"Lemma".
# For this reason, we create it first and assign it a
temporary empty value.
reviews['Lemma'] = ''
# for each row in the Pandas DataFrame
for index, row in reviews.iterrows():
    # Get lemmatized form of words from the 'Text' column
    lemmas = get_lemmas(row['Text'])
    # Update the 'Lemma' column with the lemmatized words
    reviews.at[index, 'Lemma'] = lemmas
```

We simply check the correctness in the DataFrame.

```
print(reviews.head())
```

Program output:

```
                                                Text
Sentiment  \
0  Fiona má pekné a upravené vlasy, super makeup....
pozitivna
1  Fiona je super princezná. Je vždy dobrá a milá...
pozitivna
```

```
2          Naša dobrá princezná je zároveň aj pekná.
pozitivna
3                                          Je super.
pozitivna
4                                Je dobrá, môže byť.
pozitivna

                                                 Lemma
0  fiona mať pekný a upravený vlas , super makeup...
1  fiona byť super princezná . byť vždy dobrý a m...
2        náš dobrý princezná byť zároveň aj pekný .
3                                       byť super .
4                            byť dobrý , môcť byť .
```

In the dataset prepared in this way, we have several options for further proceeding. The first and typical option is to use lemmatized sentences as input to custom classifier. In our task, we want to find individual positive and negative words.

We will use the CountVectorizer() method, which can create a vector of words for a document by specifying the number of individual words. The CountVectorizer() method takes continuous text as input. For this reason, we merge all positive rows and all negative rows into one text.

```
positive_messages = reviews[reviews['Sentiment'] ==
'pozitivna']['Lemma']
negative_messages = reviews[reviews['Sentiment'] ==
'negativna']['Lemma']

print(positive_messages)
print(negative_messages)
positive_text = ''
for sentence in positive_messages:
    positive_text += sentence
    print(sentence)

negative_text = ''
for sentence in negative_messages:
    negative_text += sentence
```

In our case, there are only 5 ratings. **Positive** and **negative** variable values can be displayed.

```
print(positive_text)
print(negative_text)
```

Program output:

```
fiona mať pekný a upravený vlas , super makeup . nosiť však
otrasný outfita . fiona byť super princezná . byť vždy dobrý a
milý . fiona vždy rád pomôcť . náš dobrý princezná byť zároveň
aj pekný . byť super . byť dobrý , môcť byť .
fiona byť náladový . to super princ si saslúžiť . outfita čo
nosiť byť otrasný . škoda reč , aj tak tento príspevok cenzúra
vymacť . to čo mať za muž . takýto hrozný ozruta doniesť do
kráľovstvo . naposledy mať otrasný účes . tá farba vlas byť
tiež hrozný .
```

The variables prepared in this way will represent the input to the **fit_transform()** method for **CountVectorizer()**, which will create a vector from the frequencies of individual words/lemmas in the ratings.

```
from sklearn.feature_extraction.text import CountVectorizer
sent_vectorizer = CountVectorizer()

corpus = sent_vectorizer.fit_transform([positive_text,
negative_text])
```

The created **corpus** variable represents a 2x49 matrix. It means that all lemmas in positive and negative messages were 49. The first row of the matrix represents the frequency of lemmas in positive and the second row in negative ratings.

For the sake of interest, we also present the possibility how the values of the **corpus** variable can be displayed using **Pandas**.

```
# Creating a DataFrame named "evaluated_words" from the matrix
(korpus.A)
# The columns are assigned feature names from the vectorizer
(word features)
evaluated_words = pandas.DataFrame(corpus.A,
columns=sent_vectorizer.get_feature_names())

# Displaying the DataFrame
print(evaluated_words)
```

Program output:

```
   aj  byť  cenzúra  do  dobrý  doniesť  farba  fiona  hrozný
kráľovstvo  \
0   1    6        0   0      3        0      0      3       0
0
1   1    3        1   1      0        1      1      1       2
1
```

```
   ...  upravený  vlas  vymacť  však  vždy  za  zároveň  účes
čo  škoda
0  ...         1     1       0     1     2   0        1     0
0      0
1  ...         0     1       1     0     0   1        0     1
2      1

[2 rows x 48 columns]
/home/johny/.local/lib/python3.9/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning:
Function get_feature_names is deprecated; get_feature_names is
deprecated in 1.0 and will be removed in 1.2. Please use
get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

It is obvious that the first row represents the frequency of positive and the second row represents the frequency of negative words. At the beginning of this chapter, we wanted to solve the problem with the missing list of positive and negative words. At the same time, we wanted to create a list with an assigned weight to the sentiment of individual words. If we multiply the number of negative words by -1 in the **DataFrame evaluated_words** and then we average the values, we can create such a list.

```
# Multiplying the second row by -1
evaluated_words.iloc[1] = (evaluated_words.iloc[1] * (-1))

# Adding the last row - the average of the values
evaluated_words.loc['average'] = (evaluated_words.iloc[0] +
evaluated_words.iloc[1]) / 2

# Displaying the DataFrame
print(evaluated_words)
```

Program output:

```
          aj  byť  cenzúra   do  dobrý  doniesť  farba  fiona
hrozný  \
0        1.0  6.0      0.0  0.0    3.0      0.0    0.0    3.0
0.0
1       -1.0 -3.0     -1.0 -1.0    0.0     -1.0   -1.0   -1.0
-2.0
average  0.0  1.5     -0.5 -0.5    1.5     -0.5   -0.5    1.0
-1.0
```

```
         kráľovstvo  ...  upravený  vlas  vymacť  však  vždy
za  zároveň  \
0               0.0  ...       1.0   1.0     0.0   1.0   2.0
0.0      1.0
1              -1.0  ...       0.0  -1.0    -1.0   0.0   0.0 -
1.0      0.0
average        -0.5  ...       0.5   0.0    -0.5   0.5   1.0 -
0.5      0.5

         účes   čo  škoda
0         0.0  0.0    0.0
1        -1.0 -2.0   -1.0
average  -0.5 -1.0   -0.5

[3 rows x 48 columns]
```

The list is currently created in a DataFrame. For a simple check, we can find the values of the words "good", "terrible", "horrible".

```
print(evaluated_words['dobrý']['average'])
print(evaluated_words['hrozný']['average'])
print(evaluated_words['škoda']['average'])
```

Program output:

```
1.5
-1.0
-0.5
```

In the preparation of text ratings, we have not used the removal of the so-called stopwords. Over a closer examination, it can be seen that words like "also", "but", "do", etc. have sentiment values from -0.5 to 0.5. It means that they occur equally in positive and negative texts. It is obvious that for a larger number of texts, their sentiment weight (average of positive and negative occurrences*-1) will be close to 0.

For completeness, here is an example of how such a list can be used to evaluate the sentiment of a new sentence. We will evaluate the sentence: "**I think Fiona is great, but the censorship is terrible in the kingdom!**" We will use the **get_lemmas()** function, which we defined earlier in the text of this chapter. The function returns the result in the form of text separated by spaces. The result of the function, i.e. all lemmas, which are identified in the evaluated sentence, we put them in the list.

```
# Original sentence
sentence = "Fionu pokladám za super, ale cenzúru v kráľovstve
máme hroznú!"
```

```
# Tokenizing the sentence into lemmas and splitting it by
spaces
sentence_tokens = get_lemmas(sentence).split(' ')

# Displaying the tokens
print(sentence_tokens)
```

Program output:

```
['fion', 'pokladať', 'za', 'super', ',', 'ale', 'cenzúra',
'v', 'kráľovstvo', 'mať', 'hrozný', '!', '']
```

From our list of rated words, we assign all the words that are rated to the **word_list** variable. To check, we will excerpt the beginning (first 9 words) of this variable.

```
word_list = sent_vectorizer.get_feature_names()
print(word_list[:9])
```

Program output:

```
['aj', 'byť', 'cenzúra', 'do', 'dobrý', 'doniesť', 'farba',
'fiona', 'hrozný']
/home/johny/.local/lib/python3.9/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning:
Function get_feature_names is deprecated; get_feature_names is
deprecated in 1.0 and will be removed in 1.2. Please use
get_feature_names_out instead.
  warnings.warn(msg, category=FutureWarning)
```

Finally, we calculate the average of the sentiment values of the words from the analysed sentence and write the result. It is obvious that a negative number in the result means a negative sentiment, a positive one means a positive one.

```

# Initializing counters
sentiment_score = 0
word_count = 0

# For each tokenized word in the sentence
for word in sentence_tokens:

    # If the word is in the list of evaluated words
    if word in evaluated_words:

        # Retrieve the sentiment weight from the pandas
DataFrame
        sentiment_score += evaluated_words[word]['average']
        word_count += 1
```

```
# Calculate the final sentiment
final_sentiment = sentiment_score / word_count
print('Sentence sentiment:', str(final_sentiment))
```

Program output:

```
Sentence sentiment: -0.3333333333333333
```

12.4.4

Project: Sentiment classifier from the existing free available reviews

Creating custom classifier with the sentiment weight of individual words is a good example for understanding sentiment analysis. However, in practice there are more sophisticated solutions. Their principle is very similar. A classifier is created, and/or trained, on a sample of existing rated texts (i.e. texts to which information about their sentiment is assigned). For training the classifier a number of classification algorithms is offered, we do not have to limit to calculating the sentiment weight of words.

Task: Create a sentiment classifier from the existing free avaible reviews of The Kingdom beyond the Seven Mountains.

We will create a sentiment classifier using the so-called Bayesian classifier. It belongs to the simplest classification algorithms. It is based on Bayesian probability calculations. Due to its simplicity, it is referred to as the **Naive** Bayes classifier.

We will work with the review dataset created in the previous subsection. From the previous chapter the **get_lemmas()** function will also be used.

```
reviews
```

```
def get_lemmas(input_text):
    output = ''
    # Apply morphological analysis to the input text
    # using the nlp function from the stanza library
    document = nlp(input_text)

    # For all sentences in the result
    for sentence in document.sentences:
        # For all words
        for word in sentence.words:
            # Append the lemma of the word to the output
            output = output + word.lemma + ' '
    return output
```

To prepare the text, we will use the **TfidfVectorizer**, which transforms the review reports into a TF-IDF document model. The basis of classification methods is training. It consists of feeding known examples to the input of the classifier. In our case, the TF-IDF vector of each review will be given to the input of the classifier, together with the information whether the review was positive or negative. Using classification methods, the classifier "learns" to distinguish between positive and negative reviews.

Traditionally, all known examples (reviews) are not all included in the input of the classifier. The whole set of examples is divided into so-called training and test set. The test set (which is not used in training) serves to verify the success of training the classifier. It is also often used to estimate the corret ratio between model success and robustness.

```
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer

X_train, X_test, y_train, y_test =
train_test_split(reviews['Lemma'], reviews['Sentiment'],
random_state = 0, test_size=0.2)
```

For completeness, we present the entire notation for dividing the dataset into training and test sets:

X_train, X_test, y_train, y_test = train_test_split(reviews['Lema'], reviews['Sentiment'], random_state = 0, test_size=0.2)

The train_test_split() function outputs four subsets. **X_train** is data for training (in our case, reviews), **X_test** is assigned information, whether a particular row is a positive or negative review. It should be noted that the order is important for both variables. The same applies to the variables **y_train** and **y_test**. These subsets will only be used to verify the accuracy of the classifier.

```
X_train, y_train, X_test, y_test
```

In the **train_test_split()** function, the first parameter is the column with reviews in **Pandas** (in our case – the processing of Slovak – these are identified lemmas), the second is the sentiment assigned to the reviews. The last parameter **test_size** determines what part of the examples from the dataset is kept as a test set. In our case, we set it to 0.2, which represents 20% of all data from the dataset. 80% of all data, i.e. in our case, 8 reviews will be used for training and 20%, i.e. two cases for testing. The 80:20 ratio is quite common. For a distribution of the dataset, the ratio is not exactly given. Besides the one set by us, a ratio of 70:30 or 90:10 is also used.

In the next step, we convert the reviews from the training set into TF-IDF vectors. Note that in the case of calling the variable **X_train_tfidf**, which is a "list of vectors", i.e. a matrix, Python will only show us the size of this resulting matrix. In our case it

is 8x44. It means there are 8 review vectors because we divided 10 reviews in ratio 80:20. The second dimension of 44 means that 44 unique lemmas (words) were identified across all reviews.

```
tfidf_vect = TfidfVectorizer()
X_train_tfidf = tfidf_vect.fit_transform(X_train)

X_train_tfidf
```

If we would like to list the variable **X_train_tfidf** (but it is not necessary), we can use the call to convert it to **Pandas**.

```
pandas.DataFrame(X_train_tfidf.A, columns =
tfidf_vect.get_feature_names()).head(10)
```

The most important part of our example consists in creating the Naive Bayes classifier. To creation, we will use the **MultinomialNB** method, which implements a naive Bayesian algorithm for multinomial data. It is one of the two classic naive Bayes variants used in text classification (where the data are usually represented as vectors of word counts or TF-IDF vectors).

```
from sklearn.naive_bayes import MultinomialNB
clf = MultinomialNB().fit(X_train_tfidf, y_train)
```

We will create the classifier within the **clf** variable using the typical **fit()** function for training any classifier. The input to the function will be the TF-IDF vectors of the reviews along with the associated sentiment.

Part of the classifier training is the evaluation of its success. To evaluate the classifier created by us, we will use the test sets from the variable **X_test**. Note that we also have to convert this set into TF-IDF vectors first. We will find the successfulness using the **accuracy_score()** method. We present this method only because of the completeness of solving the example. It is obvious that we could calculate the successfulness of the classification even without it.

```
y_pred = clf.predict(tfidf_vect.transform(X_test))
from sklearn import metrics
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

Our classifier achieved a success rate of 50%. A classifier that outperforms a random generator is usually considered successful. We classified into two classes, the expected success rate of random generation is also 50%. So our classifier tied with the random generator ????. It is necessary to realize that the success of the classifier is influenced by the number of examples from the training set. In our example, we worked with 10 reviews, which we split 80:20 into training and test sets. So the training was done with only 8 examples, which is of course very few.

The remaining two examples were used for testing. Thus, an accuracy of 50% means that one was classified correctly and the other incorrectly.

After training the classifier, our last step is to demonstrate its use. The **predict()** function is used to classify the sentence. The output of this function is the name of the class to which the input sentence was assigned after classification. We have to convert the sentence that we want to classify into lemmas and from them to create a TF-IDF vector. In the following samples, we present one positively and one negatively classified sentence.

```
sentence = "V kráľovstve sa máme všetci dobre, super si tam žijeme"
sentence_lemmas = get_lemmas(sentence)
print(clf.predict(tfidf_vect.transform([sentence_lemmas])))
```

```
sentence = "Ten muž je zelená ozruta."
sentence_lemmas = get_lemmas(sentence)
print(clf.predict(tfidf_vect.transform([sentence_lemmas])))
```

12.5 Available libraries

12.5.1

Sentiment is often just a kind of additional information alongside other textual analyses. In such a case, training custom classifier would take a lot of time. Also, we do not always have a suitable dataset for training. Fortunately, there are several libraries that are trained to determine sentiment.

Sentiment analysis using the Polyglot library

The library is presented by its authors as a solution that supports multilingual natural language processing. According to the project site https://polyglot.readthedocs.io, the library supports sentiment analysis for 136 languages.

We will present the work with the library on a simple sample. Due to the "sensitivity" of the library to the version of Python, the relatively large scope of the library itself as well as the necessary supporting libraries and corpora, we demonstrated the sample in the Google Colab environment (https://colab.research.google.com).

The library for its functionality needs additional libraries such as **pyicu**, **pycld2**, and **morfessor**. These along with the library need to be installed.

```
!pip install polyglot
!pip install pyicu
!pip install pycld2
!pip install morfessor
```

After importing the library, you still need to download the sentiment analysis package in Slovak.

```
from polyglot.text import Text

!polyglot download sentiment2.sk
```

Program output:

```
[polyglot_data] Downloading package sentiment2.sk to
[polyglot_data]     /home/johny/polyglot_data...
```

We will examine the sentiment of the sentence from the reviews of the Kingdom beyond the Seven Mountains from the previous chapter.

```
text = Text("Fionu pokladám za super, ale cenzúru v kráľovstve
máme hroznú!")
```

Polyglot includes functionality of automatic language detection. For checking, we can write the identified language of the analysed sentence.

```
print("Language Detected: Code={},
Name={}\n".format(text.language.code, text.language.name))
```

Program output:

```
Language Detected: Code=sk, Name=Slovak
```

We can find the sentiment of the sentence with a simple call.

```
print(text.polarity)
```

Program output:

```
1.0
```

Despite the fact that we identified the sentence as positive, in our opinion, the sentiment for Slovak in the Polyglot library still has great reserves. It will probably be disappointing a finding how it calculated this value, i.e. how Polyglot evaluated the individual words of the analysed sentence.

```
for w in text.words:
    print("{:<16}{:>2}".format(w, w.polarity))
```

Program output:

```
Fionu            0
pokladám         0
za               0
super            1
,                0
```

```
ale                0
cenzúru            0
v                  0
kráľovstve         0
máme               0
hroznú             0
!                  0
```

It is clear from the sample that it found the value 1 only by identifying only one positive word "super" to which it assigned the value 1☹.

Despite the fact that, in the case of Slovak, the library developers (or corpus designers) still have a lot of work, for English the library is relatively successful. If the reviews of Kingdom Beyond the Seven Mountains were written in English, their sentiment analysis would look as follows.

```
!polyglot download sentiment2.en

text = Text("I consider Fiona super, but we have terrible
censorship in the kingdom!")

print("Language Detected: Code={},
Name={}\n".format(text.language.code, text.language.name))
```

Program output:

```
[polyglot_data] Downloading package sentiment2.en to
[polyglot_data]     /home/johny/polyglot_data...
Language Detected: Code=en, Name=English

print(text.polarity)
```

Program output:

```
0.0
```

```
for w in text.words:
    print("{:<16}{:>2}".format(w, w.polarity))
```

Program output:

```
I                  0
consider           0
Fiona              0
super              1
,                  0
but                0
we                 0
```

```
have              0
terrible         -1
censorship        0
in                0
the               0
kingdom           0
!                 0
```

In our opinion, the polarity was determined correctly as neutral. Also, two words were correctly identified; one positive and one negative.

12.5.2

Sentiment analysis using the NLTK library

In the well-known **NLTK** library, there is an integrated tool Vader Sentiment Analyzer, which uses a dictionary of positive and negative words to evaluate the sentiment of the text. VADER (Valence Aware Dictionary and sEntiment Reasoner) is a dictionary tool based on sentiment analysis rules that is specially trained on moods (sentiments) expressed in social media.

Before we determine the sentiment of the sentence, it is necessary to import the **Vader** tool and download the corresponding corpus.

```
import nltk
nltk.download('vader_lexicon')
from nltk.sentiment.vader import SentimentIntensityAnalyzer
```

Program output:

```
[nltk_data] Downloading package vader_lexicon to
[nltk_data]     /home/johny/nltk_data...
[nltk_data]   Package vader_lexicon is already up-to-date!
```

We can find the sentiment by simply calling the **polarity_scores()** method.

```
sentence = 'I consider Fiona super, but we have terrible
censorship in the kingdom!'
senti_analyzer = SentimentIntensityAnalyzer()
sentiment_result = senti_analyzer.polarity_scores(sentence)
```

As a result, we can determine the value of the positivity and negativity of the sentence. Besides these data, we can also find the calculated neutrality of the sentence and, above all, the overall polarity in the **compound** property.

```
print(sentiment_result)
```

Program output:

```
{'neg': 0.28, 'neu': 0.566, 'pos': 0.154, 'compound': -0.4574}
```

If we take into account a small deviation in the case of neutral results, the final result of the sentiment can be written.

```
if sentiment_result['compound'] >= 0.05:
    print("Positive")
elif sentiment_result['compound'] <= -0.05:
    print("Negative")
else:
    print("Neutral")
```

Program output:

```
Negative
```

It is obvious that the **Vader** tool is currently not trained to determine the sentiment of sentences in the Slovak language.

Spam Identification

Chapter 13

13.1 Spam

📖 13.1.1

Spam refers to unsolicited or irrelevant messages often sent in bulk, typically for advertising, phishing, or malicious purposes. Spam is commonly associated with emails but can also appear in text messages, social media, or other communication platforms. Identifying and mitigating spam is crucial to protect users from scams, malware, and other security risks.

In the context of NLP, spam detection involves analyzing textual data to determine whether a given message is spam (irrelevant or harmful) or ham (legitimate). This process uses linguistic features, patterns, and statistical techniques to differentiate between the two.

Spam has unique characteristics such as repetitive keywords (e.g., "free," "win"), suspicious links, or requests for personal information. However, spammers continuously adapt, making spam detection a challenging and evolving field.

Key points to note include the need for automated spam detection systems that analyze large volumes of data efficiently and the ethical considerations in ensuring that legitimate messages are not misclassified.

📝 13.1.2

What is the primary purpose of spam detection in NLP?

- To differentiate spam from legitimate messages.
- To analyze user emotions.
- To enhance message encryption.
- To generate automatic responses.

📖 13.1.3

Spam messages often share a set of defining features that make them detectable through NLP techniques. These features include:

- Use of specific keywords - common spam words like "free," "urgent," and "click here" are used to attract attention.
- Suspicious URLs - spammers frequently include links that redirect users to malicious websites.
- Irregular formatting - spam messages may use unusual capitalization or excessive punctuation to evade detection.
- Phishing content - spam often seeks sensitive information by impersonating legitimate entities.

Understanding the structure of spam messages is essential for developing effective detection algorithms. While these characteristics are helpful in identification, spammers continually evolve their techniques, creating the need for robust and adaptable detection systems.

13.1.4

Which of the following is a common feature of spam messages?

- Suspicious URLs
- Detailed user feedback
- Encrypted attachments
- Formal writing style

13.1.5

Spam identification involves analyzing messages using various computational methods. There are three main approaches:

- Keyword-based detection - involves scanning messages for predefined spam keywords. Although simple, it may lead to high false positives.
- Pattern recognition uses algorithms to identify suspicious patterns in message structure, such as repeated phrases or unusual links.
- Machine learning models - modern spam detection heavily relies on machine learning, where algorithms are trained on labeled datasets to classify messages accurately.

These methods work together to improve the accuracy of spam detection, with machine learning providing adaptability to evolving spam tactics.

13.1.6

Which methods are commonly used for spam detection?

- Pattern recognition
- Keyword-based detection
- Formal grammar analysis
- Biometric identification

13.1.7

Machine learning has revolutionized spam detection by introducing models capable of learning from data. Supervised learning, the most common approach, involves training models on labeled datasets where messages are marked as spam or ham. Algorithms like Naive Bayes, Support Vector Machines (SVM), and Logistic Regression are widely used.

These models extract features from text data, such as word frequencies, n-grams, and sentiment scores, to make predictions. For example, if a model identifies frequent use of suspicious keywords and links in a message, it is likely to classify it as spam.

The adaptability of machine learning models is a significant advantage, allowing them to detect emerging spam patterns effectively. However, the quality and size of the training dataset significantly impact model performance.

13.1.8

What is a key advantage of using machine learning for spam detection?

- Can adapt to new spam patterns.
- Requires no labeled data.
- Eliminates the need for feature extraction.
- Works only on predefined rules.

13.1.9

Challenges in spam detection

Despite advancements, spam detection faces several challenges:

- Evolving spam techniques - spammers constantly update their strategies to evade detection, such as using obfuscated text or images.
- Imbalanced datasets - spam datasets often have fewer spam messages compared to legitimate ones, which can affect model training.
- False positives and negatives - misclassification of legitimate messages as spam (false positives) and spam as legitimate (false negatives) can lead to user dissatisfaction.
- Multilingual spam - detecting spam in multiple languages requires sophisticated NLP models capable of handling diverse linguistic structures.

These challenges underscore the need for ongoing research and innovation to enhance spam detection systems.

13.1.10

Which challenges are commonly associated with spam detection?

- Multilingual spam
- Evolving spam techniques
- Perfect classification accuracy
- Consistent datasets

13.2 Spam project

13.2.1

Project: Spam Classifier with TF-IDF and Naive Bayes

(by https://hussnain-akbar.medium.com/understanding-and-implementing-na%C3%AFve-bayes-algorithm-for-email-spam-detection-85a14b330fc6). Project is a copy of project in course Artificial intelligence in cyber security. We include it here for the purpose of smooth navigation through the content.

Create a spam classifier using a Naive Bayes algorithm in combination with TF-IDF (Term Frequency-Inverse Document Frequency) for feature extraction.

The Naïve Bayes classifier is a supervised machine learning model that predicts the probability of an event by analyzing related features. Here, "Naïve" means that the model assumes that all features are independent, meaning that each feature contributes to the prediction independently. In simpler terms, the model considers each feature separately, without assuming any relationships between them.

For now, we will start with a simple version of the model to make it easier to understand. To do this, we will create a small, sample dataset.

```
#Essential libraries required for this model
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score,
classification_report
```

This code will create a data frame with random emails and their corresponding labels (spam or not spam). Each email will consist of a random selection of words from the **word_list**. However, the above code will have the following output.

```
# Create a random dataset
np.random.seed(42)  # For reproducibility

# Generate random words for features (words in emails)
word_list = ['discount', 'offer', 'sale', 'free', 'click',
'buy', 'win', 'money', 'gift', 'limited']

# Generate random emails
num_emails = 1000
emails = []
```

```
labels = []
for _ in range(num_emails):
    email = ' '.join(np.random.choice(word_list,
size=np.random.randint(5, 15)))
    emails.append(email)
    # Assign labels (spam or not spam)
    labels.append(np.random.choice(['spam', 'not spam'],
p=[0.3, 0.7]))

# Create a DataFrame
data = pd.DataFrame({'email': emails, 'label': labels})

# Display the first few rows of the dataset
print(data.head())
```

Program output:

```
                                               email     label
0  free money click win limited sale win money cl...  not spam
1  click offer money buy offer click discount lim...      spam
2  sale win free gift sale click sale win click g...  not spam
3              limited gift limited click offer free      spam
4  money sale discount free offer money free offe...  not spam
```

Let's walk through the steps to build and train a Naïve Bayes classifier using the dataset we created. Here is a breakdown of the four main steps:

1. Data preprocessing

In this step, we will convert the text data to numeric characters. We will use the TF-IDF (Term Frequency-Inverse Document Frequency) technique, which transforms the text into a format understood by the Naïve Bayes classifier. The TF-IDF approach helps highlight important words in a dataset while reducing the impact of common words that may not provide significant meaning.

Steps:

- Tokenization: Splitting text into individual words or tokens.
- Lowercase: Convert all text to lowercase for consistency.
- Eliminating Stop Words: Eliminate common words (such as "the", "is", "and") that do not add much to the meaning.
- TF-IDF Calculation: Calculate the TF-IDF score for each word in each document.

In following code we apply only conversion of text data into numerical features using techniques like TF-IDF.

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score,
classification_report

# Preprocessing: Convert text data to numerical features
tfidf_vectorizer = TfidfVectorizer(max_features=1000)  # Limit
features to 1000 for simplicity
X = tfidf_vectorizer.fit_transform(data['email'])
y = data['label']
```

2. Splitting the data

Next, we need to split the dataset into two parts: one for training the model and another for testing its performance. A typical split might allocate 70-80% of the data for training and the remaining 20-30% for testing.

We will use a library **sklearn** to split the dataset into training and testing sets, ensuring that both sets contain a representative distribution of classes (e.g., spam and not spam).

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

3. Training the Naïve Bayes Model

Now we can train the Naïve Bayes classifier using the training data. The model will learn from the features extracted in the preprocessing step.

Steps:

- Create an instance of the Naïve Bayes classifier.
- Fit the model on the training data, allowing it to learn the relationship between the features and the labels (spam or not spam).

```
# Initialize and train the Naive Bayes classifier
naive_bayes = MultinomialNB()
naive_bayes.fit(X_train, y_train)
```

4. Evaluating the Model

After training the model, we'll evaluate its performance on the testing data to see how well it predicts new, unseen data.

Steps:

- Use the trained model to make predictions on the testing set.
- Compare the predicted labels to the actual labels to calculate performance metrics such as:
- Accuracy: The proportion of correctly classified instances.
- Precision: The proportion of true positive predictions to the total positive predictions.
- Recall (Sensitivity): The proportion of true positive predictions to the total actual positives.
- F1 Score: The harmonic mean of precision and recall, providing a balance between the two.

```
y_pred = naive_bayes.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred,
zero_division=0)

print(f'Accuracy: {accuracy}')
print('Classification Report:\n', report)
```

Program output:

```
Accuracy: 0.66
Classification Report:
              precision    recall  f1-score   support

    not spam       0.66      1.00      0.80       132
        spam       0.00      0.00      0.00        68

    accuracy                           0.66       200
   macro avg       0.33      0.50      0.40       200
weighted avg       0.44      0.66      0.52       200
```

The accuracy of our Naive Bayes classifier on the test data is 66%. This means that the model correctly identified about two-thirds of the emails in our test set. However, when we look closer at the classification report, we notice that the precision, recall, and F1 score for the “spam” class are quite low.

Low precision means that when the model predicts an email is spam, it often turns out to be wrong. Low recall indicates that the model is missing many actual spam emails, failing to identify them correctly. Essentially, this suggests that our model struggles to accurately recognize spam emails, which is a significant concern for applications that rely on effective spam detection.

The final step is to use our trained Naive Bayes model to predict whether new emails are spam or not. To do this, we run the following code, which takes the new email data and applies the model we’ve trained. After running the prediction, we can analyze the output to see how well the model identifies spam in this new data.

```
# Example of a new email to be predicted
new_email = "Limited time offer! Click here to win a free
gift."

# Preprocess the new email using the TF-IDF vectorizer from
the training
new_email_features = tfidf_vectorizer.transform([new_email])

# Make prediction using the trained Naive Bayes classifier
predicted_label = naive_bayes.predict(new_email_features)

# Print the predicted label
print(f"Predicted Label: {predicted_label[0]}")
```

Program output:

```
Predicted Label: not spam
```

13.2.2

Project: SMS spam classifier

(by https://www.milindsoorya.co.uk/blog/build-a-spam-classifier-in-python). Project is a copy of project in course Artificial intelligence in cyber security. We include it here for the purpose of smooth navigation through the content.

In today's instant messaging world, SMS a IM spam is becoming a growing problem. As unwanted advertising messages, scams and phishing attempts are on the rise, it is essential to have effective tools to identify and filter these spam messages. In this project, we will develop a machine learning model to classify SMS/IM messages as spam or ham.

Our goal is to create a model that can analyze the content of an message and accurately predict whether it is spam. Machine learning models can learn patterns in the text itself, making them more adaptive and robust.

Used Spam Collection is a set of SMS tagged messages that have been collected for SMS Spam research. It contains one set of SMS messages in English of 5,574 messages, tagged according to being ham (legitimate) or spam. The data was obtained from UCI's Machine Learning Repository,

The local version is available at
https://priscilla.fitped.eu/data/cybersecurity/spam/sms_spam_894.txt

The steps in the project will be focused on

Data processing

- Import packages
- Loading data
- Data set preprocessing and exploration
- Creating a word cloud to see which message is spam and which is not.
- Removing stop words and punctuation
- Convert text data to vectors

Creating a spam classification model for SMS

- Splitting data into train and test files
- Use built-in Sklearn classifiers to build models
- Training data on the model
- Making predictions based on new data

Import the required packages

```
import matplotlib.pyplot as plt
import csv
import sklearn
import pickle
from wordcloud import WordCloud
import pandas as pd
import numpy as np
import nltk
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import CountVectorizer,
TfidfTransformer
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import
GridSearchCV,train_test_split,StratifiedKFold,cross_val_score,
learning_curve
```

Loading the Dataset

```
data =
pd.read_csv('https://priscilla.fitped.eu/data/cybersecurity/sp
am/sms_spam_894.txt', encoding='latin-1', delimiter='\t',
header=None)
print(data.head())
```

Program output:

```
      0                                                   1
0   ham  Go until jurong point, crazy.. Available only ...
1   ham                      Ok lar... Joking wif u oni...
```

```
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...
3   ham  U dun say so early hor... U c already then say...
4   ham  Nah I don't think he goes to usf, he lives aro...
```

Name the columns for better processing.

```
data.rename(columns={0: 'label', 1: 'text'}, inplace=True)
print(data.head())
```

Program output:

```
  label                                                text
0   ham  Go until jurong point, crazy.. Available only ...
1   ham                      Ok lar... Joking wif u oni...
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...
3   ham  U dun say so early hor... U c already then say...
4   ham  Nah I don't think he goes to usf, he lives aro...
print(data['label'].value_counts())
```

Program output:

```
label
ham     4825
spam     747
Name: count, dtype: int64
```

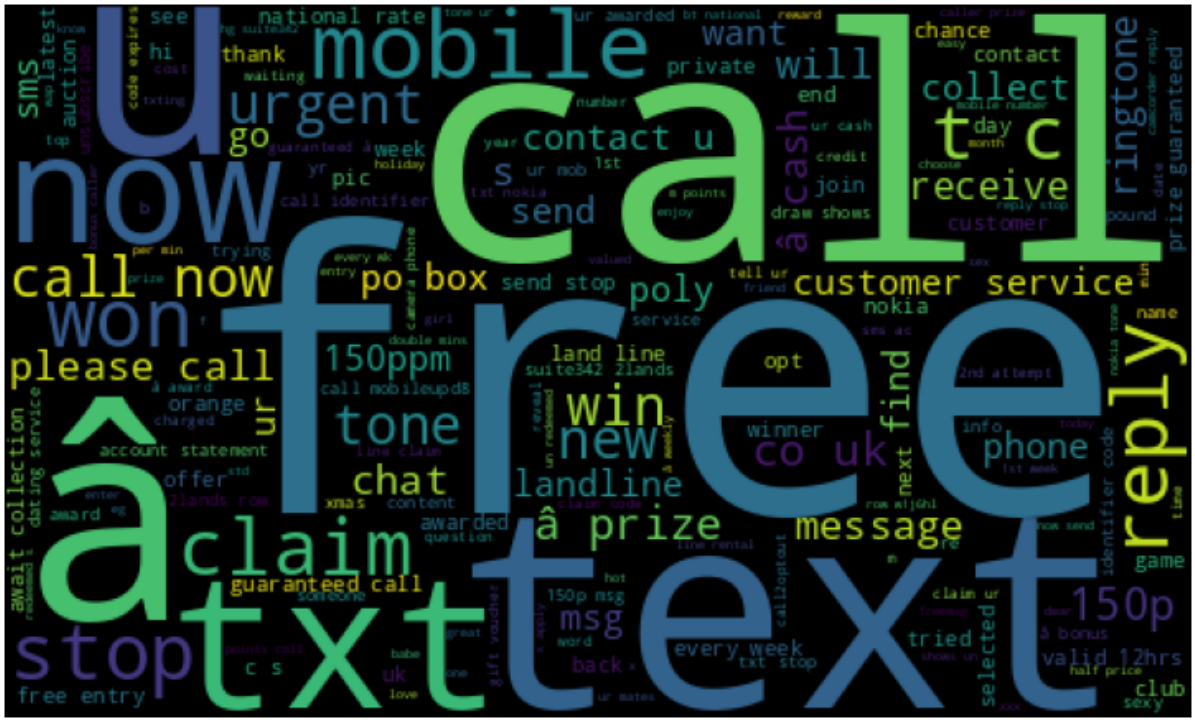
Preprocessing and Exploring the Dataset

Build word cloud to see which message is spam and which is not

```
ham_words = ''
spam_words = ''
# Creating a corpus of spam messages
for val in data[data['label'] == 'spam'].text:
    text = val.lower()
    tokens = nltk.word_tokenize(text)
    for words in tokens:
        spam_words = spam_words + words + ' '

# Creating a corpus of ham messages
for val in data[data['label'] == 'ham'].text:
    text = text.lower()
    tokens = nltk.word_tokenize(text)
    for words in tokens:
        ham_words = ham_words + words + ' '
# Create Spam word cloud and ham word cloud.
```

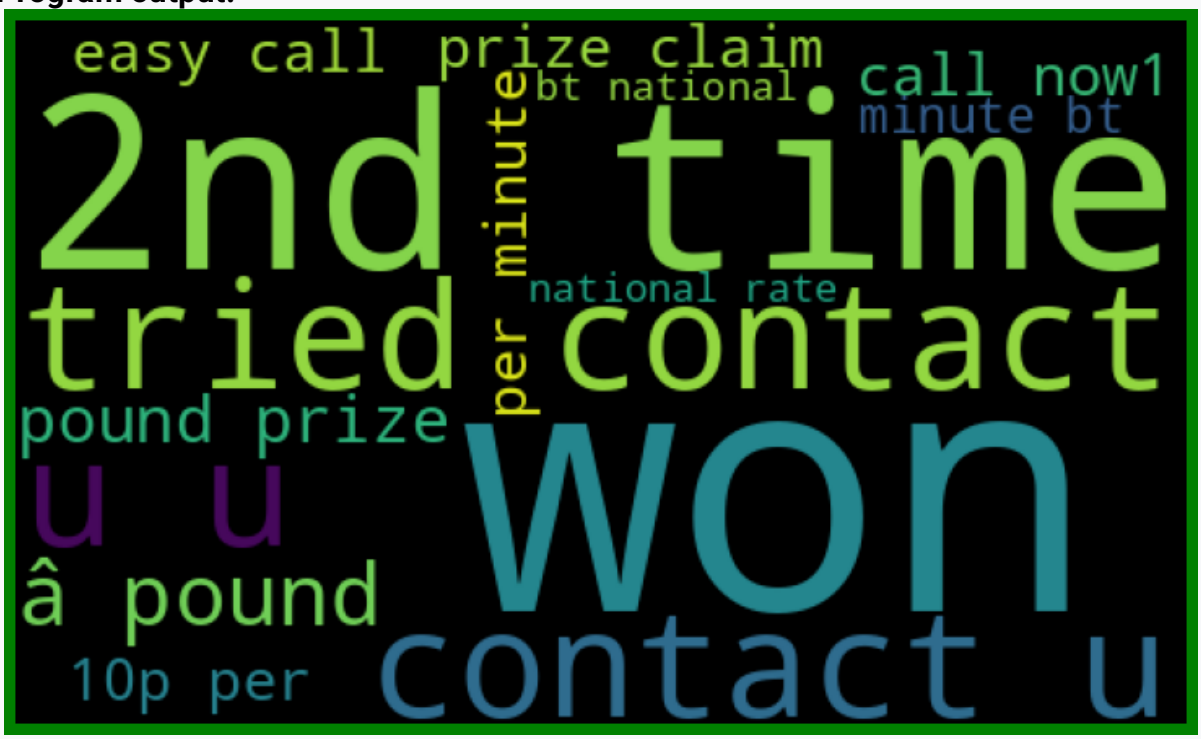
```
spam_wordcloud = WordCloud(width=500,
height=300).generate(spam_words)
ham_wordcloud = WordCloud(width=500,
height=300).generate(ham_words)

#Spam Word cloud
plt.figure( figsize=(10,8), facecolor='w')
plt.imshow(spam_wordcloud)
plt.axis("off")
plt.tight_layout(pad=0)
plt.show()
```

Program output:




```
#Creating Ham wordcloud
plt.figure( figsize=(10,8), facecolor='g')
plt.imshow(ham_wordcloud)
plt.axis("off")
plt.tight_layout(pad=0)
plt.show()
```

Program output:



from the spam word cloud, we can see that "free" is most often used in spam.

Now, we can convert the spam and ham into 0 and 1 respectively so that the machine can understand.

```
data = data.replace(['ham','spam'],[0, 1])
print(data.head(10))
```

Program output:

```
   label                                                text
0      0  Go until jurong point, crazy.. Available only ...
1      0                      Ok lar... Joking wif u oni...
2      1  Free entry in 2 a wkly comp to win FA Cup fina...
3      0  U dun say so early hor... U c already then say...
4      0  Nah I don't think he goes to usf, he lives aro...
5      1  FreeMsg Hey there darling it's been 3 week's n...
6      0  Even my brother is not like to speak with me. ...
7      0  As per your request 'Melle Melle (Oru Minnamin...
8      1  WINNER!! As a valued network customer you have...
9      1  Had your mobile 11 months or more? U R entitle...
:1: FutureWarning: Downcasting behavior in `replace` is
deprecated and will be removed in a future version. To retain
the old behavior, explicitly call
`result.infer_objects(copy=False)`. To opt-in to the future
```

```
behavior, set `pd.set_option('future.no_silent_downcasting',
True)`
  data = data.replace(['ham','spam'],[0, 1])
```

Removing punctuation and stopwords from the messages

- Punctuation and stop words do not contribute anything to our model, so we have to remove them. Using NLTK library we can easily do it.

```
#remove the punctuations and stopwords
import string
def text_process(text):

    text = text.translate(str.maketrans('', '',
string.punctuation))
    text = [word for word in text.split() if word.lower() not
in stopwords.words('english')]

    return " ".join(text)

data['text'] = data['text'].apply(text_process)
print(data.head(10))
```

Program output:

```
   label                                               text
0      0  Go jurong point crazy Available bugis n great ...
1      0                            Ok lar Joking wif u oni
2      1  Free entry 2 wkly comp win FA Cup final tkts 2...
3      0                U dun say early hor U c already say
4      0       Nah dont think goes usf lives around though
5      1  FreeMsg Hey darling 3 weeks word back Id like ...
6      0     Even brother like speak treat like aids patent
7      0  per request Melle Melle Oru Minnaminunginte Nu...
8      1  WINNER valued network customer selected receiv...
9      1  mobile 11 months U R entitled Update latest co...
```

Now, create a data frame from the processed data before moving to the next step.

```
text = pd.DataFrame(data['text'])
label = pd.DataFrame(data['label'])
```

Converting words to vectors

We can convert words to vectors using either Count Vectorizer or by using TF-IDF Vectorizer.

TF-IDF is better than Count Vectorizers because it not only focuses on the frequency of words present in the corpus but also provides the importance of the words. We can then remove the words that are less important for analysis, hence making the model building less complex by reducing the input dimensions.

I have included both methods for your reference.

Converting words to vectors using Count Vectorizer

```
## Counting how many times a word appears in the dataset
from collections import Counter

total_counts = Counter()
for i in range(len(text)):
    for word in text.values[i][0].split(" "):
        total_counts[word] += 1

print("Total words in data set: ", len(total_counts))
```

Program output:

```
Total words in data set:  11426
```

```
# Sorting in decreasing order (Word with highest frequency
appears first)
vocab = sorted(total_counts, key=total_counts.get,
reverse=True)
print(vocab[:60])
```

Program output:

```
['u', '2', 'call', 'U', 'get', 'Im', 'ur', '4', 'ltgt',
'know', 'go', 'like', 'dont', 'come', 'got', 'time', 'day',
'want', 'Ill', 'lor', 'Call', 'home', 'send', 'one', 'going',
'need', 'Ok', 'good', 'love', 'back', 'n', 'still', 'text',
'im', 'later', 'see', 'da', 'ok', 'think', 'Ã¼', 'free',
'FREE', 'r', 'today', 'Sorry', 'week', 'phone', 'mobile',
'cant', 'tell', 'take', 'much', 'night', 'way', 'Hey',
'reply', 'work', 'give', 'make', 'new']
```

```
# Mapping from words to index
vocab_size = len(vocab)
word2idx = {}
#print vocab_size
for i, word in enumerate(vocab):
    word2idx[word] = 1
```

```
# Text to Vector
def text_to_vector(text):
    word_vector = np.zeros(vocab_size)
    for word in text.split(" "):
        if word2idx.get(word) is None:
            continue
        else:
            word_vector[word2idx.get(word)] += 1
    return np.array(word_vector)

# Convert all titles to vectors
word_vectors = np.zeros((len(text), len(vocab)),
dtype=np.int_)
for i, (_, text_) in enumerate(text.iterrows()):
    word_vectors[i] = text_to_vector(text_[0])

print(word_vectors.shape)
```

Program output:

```
:21: FutureWarning: Series.__getitem__ treating keys as
positions is deprecated. In a future version, integer keys
will always be treated as labels (consistent with DataFrame
behavior). To access a value by position, use `ser.iloc[pos]`
  word_vectors[i] = text_to_vector(text_[0])
(5572, 11426)
```

Converting words to vectors using TF-IDF Vectorizer

```
#convert the text data into vectors
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()
vectors = vectorizer.fit_transform(data['text'])
print(vectors.shape)
```

Program output:

```
(5572, 9459)
```

```
# You can choose type of converted data
#features = word_vectors
features = vectors
```

Splitting into training and test set

```
#split the dataset into train and test set
X_train, X_test, y_train, y_test = train_test_split(features,
data['label'], test_size=0.15, random_state=111)
```

Classifying using sklearn's pre-built classifiers

- In this step we will use some of the most popular classifiers out there and compare their results.

```
#import sklearn packages for building classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.naive_bayes import MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

#initialize multiple classification models
svc = SVC(kernel='sigmoid', gamma=1.0)
knc = KNeighborsClassifier(n_neighbors=49)
mnb = MultinomialNB(alpha=0.2)
dtc = DecisionTreeClassifier(min_samples_split=7,
random_state=111)
lrc = LogisticRegression(solver='liblinear', penalty='l1')
rfc = RandomForestClassifier(n_estimators=31,
random_state=111)

#create a dictionary of variables and models
clfs = {'SVC' : svc,'KN' : knc, 'NB': mnb, 'DT': dtc, 'LR':
lrc, 'RF': rfc}

#fit the data onto the models
def train(clf, features, targets):
    clf.fit(features, targets)

def predict(clf, features):
    return (clf.predict(features))

pred_scores_word_vectors = []
for k,v in clfs.items():
    train(v, X_train, y_train)
    pred = predict(v, X_test)
```

```
    pred_scores_word_vectors.append((k, [accuracy_score(y_test
, pred)]))
```

Predictions using TFIDF Vectorizer algorithm

```
print(pred_scores_word_vectors)
```

Model predictions

```
#write functions to detect if the message is spam or not
def find(x):
    if x == 1:
        print ("Message is SPAM")
    else:
        print ("Message is NOT Spam")

newtext = ["Free entry"]
integers = vectorizer.transform(newtext)

x = mnb.predict(integers)
find(x)
xx = knc.predict(integers)
find(xx)
```

Program output:

```
Message is SPAM
Message is SPAM
```

Large Language Models

Chapter 14

14.1 Large language models

📖 14.1.1

Large Language Models (LLMs) are a breakthrough in artificial intelligence, designed to process and generate human-like text. These models, such as GPT-3, are built using deep learning techniques and are trained on massive datasets, often comprising billions of words. LLMs are capable of performing a wide range of NLP tasks, from answering questions and translating text to summarizing content and generating creative writing.

The foundation of LLMs lies in their ability to predict the next word or phrase in a sequence, based on patterns learned during training. This predictive power makes LLMs incredibly versatile, capable of handling everything from basic tasks like text classification to more complex functions like conversational agents and creative text generation. LLMs work by capturing context and syntactic rules from vast amounts of data, allowing them to generate responses that are contextually appropriate and grammatically accurate.

The significance of LLMs in the AI landscape is immense, as they can be applied to multiple industries, including healthcare, finance, marketing, and customer service. For instance, in customer service, LLMs can be integrated into chatbots to provide real-time support, while in healthcare, they can assist in generating medical reports or providing health information.

📝 14.1.2

What is a major benefit of using LLMs in industries such as customer service and healthcare?

- They provide accurate and personalized responses at scale.
- They eliminate the need for human employees entirely.
- They reduce all costs associated with service delivery.
- They completely remove bias from decision-making processes.

📖 14.1.3

The Transformer model, introduced in 2017 by Vaswani et al., is the core architecture that powers modern LLMs. Unlike earlier models like Recurrent Neural Networks (RNNs), the Transformer relies on a mechanism called "self-attention" to process all words in a sentence simultaneously, rather than sequentially. This allows the model to capture long-range dependencies in text more effectively, leading to a better understanding of context and meaning across longer sequences.

Self-attention works by assigning different weights to the words in a sentence, indicating their relevance to each other. This enables the model to focus on important parts of the sentence while processing the entire input in parallel. This

parallelization makes Transformers much faster and more efficient than previous architectures, especially when dealing with large datasets. This is why Transformers are the backbone of popular models like GPT-3, BERT, and T5, which are used in a variety of NLP applications.

In addition to self-attention, the Transformer model uses a position encoding mechanism, which helps the model understand the order of words in a sentence. The combination of self-attention and position encoding allows Transformers to understand both the content and structure of text more effectively, making them a game-changer in the field of natural language processing.

📝 14.1.4

What is the primary advantage of the Transformer model compared to earlier models like RNNs?

- It processes sequences in parallel, improving efficiency.
- It uses fewer parameters, making it faster to train.
- It relies solely on convolutional layers for computation.
- It requires labeled data for training.

📖 14.1.5

Training large language models (LLMs) is a complex and resource-intensive process that requires massive computational power. The training process typically involves two stages: pretraining and fine-tuning. During pretraining, the model is exposed to a large and diverse corpus of text, learning patterns in language, grammar, and contextual relationships. This stage is crucial because it enables the model to learn a broad understanding of language.

Fine-tuning is the second phase, where the model is adapted to specific tasks or domains. In this stage, the model is trained on a smaller, more specialized dataset that is tailored to a particular application, such as medical text or legal documents. Fine-tuning helps the model become more accurate and effective at performing specific tasks, such as answering legal questions or generating medical reports.

The sheer size of the datasets and the number of parameters (often in the billions) makes training LLMs extremely expensive and time-consuming. In addition, the process requires advanced hardware, such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs), to speed up the computations. Despite these challenges, the results are highly rewarding, as LLMs can perform a wide array of tasks with impressive accuracy.

📝 14.1.6

Why is fine-tuning important when training large language models?

- It adapts the model to specific tasks or domains.

- It reduces the size of the model for faster processing.
- It eliminates the need for pre-training on large datasets.
- It ensures the model uses only supervised learning methods.

📖 14.1.7

Large language models have a wide range of practical applications that extend across multiple industries. One of the most common uses is in **text generation**, where LLMs can create coherent and contextually relevant content from a given prompt. This has revolutionized fields such as content creation, marketing, and even creative writing, enabling automated generation of articles, advertisements, and social media posts.

Another significant application is **text summarization**, where LLMs can condense long documents or articles into shorter, more digestible summaries. This is useful for professionals who need to quickly grasp the main points of lengthy reports or research papers. Similarly, LLMs are used for **language translation**, where they can accurately translate text between different languages, improving global communication.

LLMs are also making their mark in **question answering systems**. These systems, powered by LLMs, can answer queries posed in natural language, providing users with detailed and contextually relevant responses. This has led to the development of AI-powered assistants and chatbots that can provide real-time, accurate information on a wide range of topics.

📝 14.1.8

What is one of the most common applications of large language models in industries like content creation and marketing?

- Generating creative and persuasive written content.
- Predicting stock market trends.
- Detecting fraudulent transactions.
- Simulating scientific experiments.

📖 14.1.9

Ethical considerations and challenges

While LLMs are powerful tools, they also come with ethical considerations and challenges:

- **Bias** - LLMs can inherit biases present in the data they are trained on. This can lead to biased or discriminatory outputs, which can be harmful in sensitive applications.
- **Misinformation** - LLMs can inadvertently generate false or misleading information, as they may not have a true understanding of the content.

- **Privacy concerns** - if trained on private or sensitive data, LLMs could inadvertently reveal personal information in their outputs.
- **Energy consumption** - training large models requires significant computational resources, which can contribute to environmental concerns.

14.1.10

What is one ethical challenge associated with large language models that can result in biased outputs?

- Learning biases present in training data.
- Lack of computational efficiency.
- Insufficient training data.
- Limited availability of pre-trained models.

14.2 LLM in practice

14.2.1

Fine-tuning is a critical step in adapting large language models (LLMs) for specific applications. After a model like GPT-3 is pretrained on massive amounts of general language data, it can be fine-tuned on smaller, task-specific datasets. This process allows the model to focus on particular domains, such as legal, medical, or technical language, improving its relevance and accuracy for those areas. For example, a fine-tuned model trained on medical data can provide accurate summaries of patient reports or answer complex medical queries.

The fine-tuning process leverages the foundational knowledge gained during pretraining while incorporating domain-specific nuances. By training on a smaller dataset tailored to the target task, the model avoids the need to start from scratch, saving time and computational resources. This also ensures that the model retains its general language capabilities while excelling in its specialized area.

Fine-tuning can address the unique requirements of various industries by adapting the model to understand specialized terminology, sentence structures, or contextual nuances. For instance, in the legal domain, fine-tuning on court judgments or contracts enables the model to perform tasks like contract review or legal research efficiently.

14.2.2

Which statements are true about fine-tuning large language models?

- It uses domain-specific datasets to adapt the model.
- Fine-tuned models retain general language knowledge.

- It requires starting from scratch for every task.
- It eliminates the need for pretraining.

📖 14.2.3

LLMs showcase remarkable capabilities, but they are not without limitations. One fundamental issue is their lack of genuine understanding. While LLMs generate text based on patterns learned during training, they do not grasp the meaning of the words. This can result in plausible-sounding outputs that lack factual accuracy or logical consistency. For example, an LLM might confidently provide a detailed but incorrect explanation about a topic.

Another limitation is the dependence on training data quality. If the training data contains biases, inaccuracies, or outdated information, the model's outputs will likely reflect these shortcomings. This makes careful data selection and preprocessing critical for minimizing errors and bias in the model's responses.

LLMs also face constraints in handling long texts. Although they are designed to process lengthy sequences, there is a limit to how much context they can consider at once. This limitation can affect their coherence when generating or summarizing extensive texts, particularly when the input exceeds their context window.

📝 14.2.4

What are limitations of large language models?

- They lack true understanding of the text.
- Their performance depends on training data quality.
- They always produce factually accurate outputs.
- They have unlimited context processing capabilities.

📖 14.2.5

LLMs are continuously evolving, with researchers exploring ways to improve their capabilities and address existing challenges. One key focus is on enhancing their efficiency. New architectures, such as sparse models, aim to reduce computational demands without compromising performance. This innovation is crucial as LLMs become larger and require more resources to train and deploy.

Ethical considerations are another priority. Researchers are working to minimize biases in LLMs by diversifying training data and refining algorithms to promote fairness and inclusivity. These efforts are essential to ensure that the models generate outputs that are less biased and more representative of diverse perspectives.

The future also includes broader integration of LLMs across industries. From healthcare to finance, LLMs are expected to play significant roles, such as assisting in patient data analysis or providing market predictions. As they become more

sophisticated, their potential applications will expand, transforming how people interact with technology.

14.2.6

What are advancements being pursued for LLMs?

- Developing sparse models for efficiency.
- Reducing biases for ethical outputs.
- Eliminating all resource requirements.
- Expanding applications across industries.

14.2.7

Transfer learning is a foundational technique that contributes significantly to the efficiency and versatility of large language models (LLMs). Rather than training models from scratch for every specific task, transfer learning allows knowledge gained during general training to be applied to new, task-specific scenarios. This is particularly useful when dealing with specialized domains, such as healthcare or legal services, where data availability may be limited.

For example, an LLM pretrained on general language data can be fine-tuned using a smaller dataset for tasks like sentiment analysis or medical diagnosis. This reuse of foundational knowledge significantly reduces computational costs and accelerates the training process. Additionally, transfer learning enables models to maintain the broader language understanding gained during pretraining while adapting to specific contexts.

The adaptability offered by transfer learning makes LLMs highly versatile for various industries. Whether predicting customer preferences in e-commerce or analyzing market trends in finance, transfer learning ensures that models can perform these tasks effectively without requiring extensive retraining.

14.2.8

What are the benefits of transfer learning in large language models?

- Reduces computational cost for task-specific training.
- Applies knowledge from pretraining to new tasks.
- Requires training models from scratch for every task.
- Eliminates the need for general language pretraining.

14.2.9

Practical implementation

Implementing large language models (LLMs) in real-world applications requires careful planning and a structured approach. The first step is **model selection**, where the choice of an LLM depends on the specific task. For instance, GPT models are commonly used for text generation, while BERT excels in tasks like question answering and sentiment analysis. Selecting the right model ensures optimal performance for the intended application.

Data preprocessing is the next critical step. Raw data needs to be cleaned, tokenized, and formatted to align with the model's input requirements. This step minimizes noise in the data and improves the model's ability to generate accurate predictions or responses. For instance, in sentiment analysis, preprocessing might involve removing stop words, lemmatization, or handling missing values in the dataset.

After preprocessing, **fine-tuning** comes into play. This involves training the model on domain-specific data to tailor it to the desired task. Fine-tuning improves the model’s accuracy and relevance for tasks like generating chatbot responses or summarizing technical documents. Finally, the **deployment phase** integrates the model into the application, such as a customer service chatbot or a recommendation engine, using frameworks like TensorFlow or PyTorch for seamless implementation.

14.2.10

What are important steps in implementing LLMs in real-world applications?

- Selecting an appropriate model for the task.
- Preprocessing data to align with model requirements.
- Training the model from scratch for every task.
- Using tools like TensorFlow for deployment.

Bibliography and Sources

Chapter 15

15.1 Bibliography and sources

📖 15.1.1

Bibliography and sources:

1. Arinze, B. 1989. A natural language front-end for knowledge acquisition. SIGART Bull., 108 (April 1989), 106–114. https://doi.org/10.1145/63266.63280
2. https://arxiv.org/abs/1909.01066
3. https://builtin.com/machine-learning/bag-of-words
4. https://letsdatascience.com/tf-idf/
5. https://medium.com/@datathon/introduction-to-sentiment-analysis-c8cd6228313f
6. https://medium.com/@fhuqtheta/sentiment-analysis-of-news-articles-a-lexicon-based-approach-4672246a12a2
7. https://medium.com/@robdelacruz/sentiment-analysis-using-natural-language-processing-nlp-3c12b77a73ec
8. https://medium.com/analytics-vidhya/nlp-getting-started-with-sentiment-analysis-126fcd61cc4a
9. https://medium.com/nlplanet/two-minutes-nlp-learn-tf-idf-with-easy-examples-7c15957b4cb3
10. https://www.coursera.org/articles/natural-language-generation
11. https://www.deeplearning.ai/resources/natural-language-processing/
12. https://www.kaggle.com/code/benroshan/sentiment-analysis-amazon-reviews
13. https://www.kaggle.com/code/jonathanoheix/sentiment-analysis-with-hotel-reviews
14. https://www.kaggle.com/datasets/pavanelisetty/sample-audio-files-for-speech-recognition/code
15. https://www.learndatasci.com/glossary/tf-idf-term-frequency-inverse-document-frequency/
16. Chang Hoyeon, Park Jinho, Ye Seonghyeon, Yang Sohee, Seo Youngkyung, Chang Du-Seong, Seo Minjoon. How Do Large Language Models Acquire Factual Knowledge During Pretraining?, https://doi.org/10.48550/arXiv.2406.11813
17. Chen Hainan, Luo Xiaowei, An automatic literature knowledge graph and reasoning network modeling framework based on ontology and natural language processing, Advanced Engineering Informatics, Volume 42, 2019, 100959, ISSN 1474-0346, https://doi.org/10.1016/j.aei.2019.100959
18. Chen Xiaojun, Jia Shengbin, Xiang Yang, A review: Knowledge reasoning over knowledge graph,Expert Systems with Applications, Volume 141, 2020, 112948, ISSN 0957-4174, https://doi.org/10.1016/j.eswa.2019.112948
19. Chowdhary, K., & Chowdhary, K. R. (2020). Natural language processing. Fundamentals of artificial intelligence, 603-649.

20. Ji, S., Pan, S., Cambria, E., Marttinen, P., & Philip, S. Y. (2021). A survey on knowledge graphs: Representation, acquisition, and applications. IEEE transactions on neural networks and learning systems, 33(2), 494-514.
21. Petroni Fabio, Rocktäschel Tim, Lewis Patrick, Bakhtin Anton, Wu Yuxiang, Miller Alexander H., Riedel Sebastian. Language Models as Knowledge Bases? https://doi.org/10.48550/arXiv.1909.01066
22. Potter, Stephen. A Survey of Knowledge Acquisition from Natural Language (2001). https://www.researchgate.net/publication/240700260_A_Survey_of_Knowledge_Acquisition_from_Natural_Language
23. Singh, S. (2018). Natural language processing for information extraction. arXiv preprint arXiv:1807.02383.
24. Single Johannes I., Schmidt Jürgen, Denecke Jens, Knowledge acquisition from chemical accident databases using an ontology-based method and natural language processing, Safety Science, Volume 129, 2020, 104747, ISSN 0925-7535, https://doi.org/10.1016/j.ssci.2020.104747
25. Wilensky, R. (2011). Knowledge Acquisition and Natural Language Processing. In: Chipman, S., Meyrowitz, A.L. (eds) Foundations of Knowledge Acquisition: Cognitive Models of Complex Learning. The Springer International Series in Engineering and Computer Science, vol 194. Springer, Boston, MA. https://doi.org/10.1007/978-1-4615-3172-2_9
26. Zang LJ, Cao C, Cao YN et al. A survey of commonsense knowledge acquisition. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 28(4): 689-719 July 2013. https://doi.org/10.1007/s11390-013-1369-6
27. Zhang, Z., Webster, P., Uren, V. S., Varga, A., & Ciravegna, F. (2012, May). Automatically Extracting Procedural Knowledge from Instructional Texts using Natural Language Processing. In LREC (Vol. 2012, No. 520-527, pp. 520-527)

📖 15.1.2

Statement regarding the use of Artificial Intelligence in content creation

This content has been developed with the assistance of artificial intelligence tools, specifically ChatGPT, Gemini, and Notebook LM. These AI technologies were utilized to enhance the text by providing suggestions for rephrasing, improving clarity, and ensuring coherence throughout the material. The integration of these AI tools has enabled a more efficient content creation process while maintaining high standards of quality and accuracy.

The use of AI in this context adheres to all relevant guidelines and ethical considerations associated with the deployment of such technologies. We acknowledge the importance of transparency in the content creation process and aim to provide a clear understanding of how artificial intelligence has contributed to the final product

PRISCILLA

priscilla.fitped.eu