Natural language processing

Published on

Work in progress version

Erasmus+ FITPED-AI

Future IT Professionals Education in Artificial Intelligence

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

Funded by the European Union

TABLE OF CONTENTS

Introduction

Chapter 1

1.1 Introduction to the course

1.1.1

Dear students,

you have just entered the introduction to the course entitled Natural Language Processing 1. In the course you will find out how computers understand text and how they process it. We will go through all the basic concepts, we will mention the importance of text preprocessing, we will talk more about vector models. We will also move to the programming part, where we will process certain texts in the Python language and learn how to work properly with the necessary libraries. We hope that the course will help you with obtaining essential information in this area.

1.2

1.2.1

Natural languages are different from those of programming languages. In normal human-to-human communication, we do not process complex mathematical operations. We can easily share information with each other, either by spoken word or by reading the text from this course. But over time, the need for human-computer communication also arose. As computing power grows every year, we were able to adapt computers to the speed of information processing and thus we can communicate with them in real time.

Natural language processing deals with human-computer communication and takes place using natural language, e.g. Slovak or English language. It is important to say that when solving this problem, knowledge is used not only from the field of informatics (artificial intelligence), but also from linguistics or psychology. After expert systems, this is the largest application of artificial intelligence.

DEFINITION

Natural language processing (NLP) is a field of computer science and artificial intelligence research that deals with the processing of natural languages such as English or Slovak. It deals with the analysis, generation of text or spoken word that requires some degree of understanding of natural language by a computer.

In the next chapters, we will simplify the term Natural Language Processing to **NLP**.

1.2.2

Nowadays, NLP brings a lot of use to people in computer technology and makes it easier for us to work with devices. We regularly encounter certain tasks, such as:

· Text classification – spam detection, word filter, ...

· Language modeling – chatbot, predicting next words, text generation, translation, spell check, ...

· System of questions and answers,

· Sentiment analysis of texts (tweets) or opinion formation,

· Text-to-speech (synthesis of speech from text).

1.2.3

From which important areas we use knowledge of NLP?

- Computer science, linguistics, psychology.
- Networking, journalism, psychiatry.
- Cyber security, culturology, religious studies.

Programming methods

Chapter 2

2.1 Suitable technology

📖 2.1.1

Even before we dive into solving some natural language processing problems, it is necessary to choose the right programming language and environment in which we will work.

In NLP, we have two options when choosing a suitable language, namely **Java** and **Python**.

📖 2.1.2

Even before we dive into solving some natural language processing problems, it is necessary to choose the right programming language and environment in which we will work.

In NLP, we have two options when choosing a suitable language, namely **Java** and **Python**.

📖 2.1.3

Java

We consider Java to be one of the most used programming languages today. However, when solving NLP problems, it does not provide us with the same flexibility as its competitor Python.

The *OpenNLP* library is used in Java for solving NLP problems. It is the most widely used in this area, together with the *CoreNLP* library, which also has integration into the Python language.

📖 2.1.4

Python

The aforementioned Python has long been the leading programming language for solving NLP problems. It offers both great support for integration with other tools and is popular for its simplicity and speed.

There are a huge number of regularly updated libraries and tools for the Python language, which we will go over in more detail in the next chapters. A special environment is also designed for Python, which is excellent for solving any problem - Jupyter.

📖 2.1.5

Jupyter notebook

A useful tool for interactive Python mode that makes data analysts' work easier is Jupyter Notebook, formerly also known as IPython Notebook. It is a web version of the Python console where we can enter commands and check the output. This flexible tool helps create readable analyzes by allowing you to preserve codes, images, comments, and formulas. Jupyter Notebook is also relatively extensible due to the fact that it can be easily implemented on a regular computer or on almost any server.

In Jupyter Notebook, commands and their results are saved. We can easily go back to them, edit them and add comments. Using the markup language Markdown, it is possible to insert text between commands, i.e. comments, and thus move smoothly from individual notes to the code. The entire document in Jupyter Notebook can be shared, and you can create e.g. a tutorial, slides in a presentation or even a scientific article.

2.2 NLP libraries

📖 2.2.1

NLP libraries provide developers with ready-made tools that simplify work with text. They allow them to focus on building quality models for machine learning. In the past, only real experts who had very high knowledge in mathematics, linguistics and machine learning could be part of projects dealing with natural language processing. Thanks to these libraries, even beginners in programming can make their mark today, as they offer us clear documentation and a developed community in the online space.

In this chapter, we will review the most used libraries for working with data. They contain functions that also help us in natural language processing, which means that we can use them outside of this area as well. They are specifically:

- nltk
- pandas
- numpy
- sleeping
- stanza
- scikit-learn

2.2.2

NLTK

Perhaps the most widely used library for natural language processing is certainly NLTK (Natural Language ToolKit). It provides us with implementations of almost all possible NLP problems. We can tokenize it, or use it for more complex problems such as **analyzing the structure** and **meaning** of sentences. In addition to complete algorithms, it also contains freely available corpora.

One of the disadvantages of the NLTK library is that it is relatively slow. However, it is fully sufficient for solving simple tasks.

The command to import the library looks like this:

```
import nltk
```

Many examples of its use can be found in the next chapters.

2.2.3

pandas

Pandas is a library for analyzing data that can be represented by a table. This is a common type of data representation also used in tables, databases or CSV files. In short, what can be done in Excel can be done in Pandas. The basic implementation of the Pandas library is implemented by default using the import command:

```
import pandas
```

You can also often find the import call with an **alias** assignment, e.g.: import **pandas** as **pd**, while the entire functionality of the library is called with the pd alias in the following code.

```
import pandas as pd
```

The library offers a basic DataFrame data type, i.e. a table. Individual records are organized in rows and columns.

```
data = {
  "kalorie": [420, 380, 390],
  "cas": [50, 40, 45]
}

# nacitame dataframe do objektu
df = pd.DataFrame(data)

print(df)
```

In addition to classic lists, we can define data using dictionaries or from a *csv* or *json* file. The last approach is the most common.

```
df1 = pd.read_csv('data.csv')

df2 = pd.read_json('data.json')
```

Complete documentation for the pandas library can be found here.

🕮 2.2.4

numpy

The *numpy* library is the successor to the Numeric library and is intended for working with multidimensional arrays. It implements arrays of types int, float, complex and others as classes, where it stores items in memory "one after the other". Operations with classic array structures are very slow, but numpy allows us to implement operations on such arrays much more efficiently.

On the next pages you will find ways to define an array using the numpy library.

2.2.5

First, we import the numpy library.

```
import numpy as np
```

We will define several fields using numpy. In the comments, you have explained how each field is defined.

```
a = np.array([1,2,3,4,5]) # integer vector from list
b = np.zeros(5) # zero field
c = np.ones(5) # one array
d = np.array([[1,2], [3,4]]) # integer array from list
e = np.zeros((3,4),int) # zero matrix 3x4, integer
f = np.zeros((3,4),float) # zero matrix 3x4, with real numbers

print(s)
print(b) # by default the field is in float, but it contains
dots
print(c) # by default the field is in float, but it contains
dots
print(d)
print(s)
print(f)
```

In the event that we have a field with too much data, it will appear in the output as follows:

```
g = np.ones((100,100))

print(g)
```

At the same time, numpy allows us to use the already mentioned operations very easily.

```
# majme zaefinované dve polia

a = np.array([1,2,3])
b = np.array([3,4,5])

# operácie riešime nasledovne
```

```
a + b
a - b
```

2.2.6

spacy

The spacy library is considered a strong competitor to the NLTK library. It supports 50 different languages, including Slovak. The book is characterized mainly by the fact that tools for preparing text for machine learning are integrated into it. It is machine learning that is an integral part of NLP, where after the basic editing of the text, it is necessary to train the processed text with the help of artificial intelligence.

Spacy provides us with a number of pre-trained models. For example, the "*en_core_web_sm*" model is an English model containing all core functions. It is trained on the basis of English texts from websites. We load it using the load() function. The following example shows the prediction of POS tags, i.e. words, using the *spacy* library.

```
import spacy

nlp = spacy.load("en_core_web_sm")

doc = nlp("My brother Jake likes apples.")

for token in doc:
    print(token.text, token.pos_)
```

2.2.7

stanza

Stanza is a natural language parsing package in Python. It contains tools for identifying morphological features of sentences, analysis of syntactic structure, lemmatization, etc. The toolkit is designed for more than 70 languages.

Stanza works on the basis of machine learning - a neural network. It was trained using available annotated texts.

To work with the library, it is necessary to import it and download the package for the Slovak language.

```
import stanza
stanza.download('sk')
```

📖 2.2.8

scikit-learn

A great library for beginning programmers in artificial intelligence. It provides tools not only for NLP, but also for data analysis and machine learning purposes. It works on lower-level libraries, namely numpy and scipy. Its use is mainly general machine learning, as it has limited functionality within deep learning. We use the TensorFlow and Keras libraries for this. We can solve problems such as regression, classification, clustering and many others with it.

📝 2.2.9

Which library do we use to load datasets, most often in a .csv file?

- scikit-learn
- lambdas
- numpy
- pandas

Text preprocessing

Chapter 3

3.1 Introduction to data preprocessing

📖 3.1.1

Now we know several libraries that we will use when working with NLP. We will imagine the use of some of them in examples of text preprocessing.

📖 3.1.2

In order for a computer to understand natural language, it is necessary to analyze the text from a linguistic point of view and then transform it into a useful representation.

One of the basic functions for working with data, which we can use to compare the size of the original and reference text, is to determine the number of words, characters and average word length in the text. For this, we will use the basic functions in the python language. For example, we can use the *len()* function to get the number of characters. The *split()* method is used for the number of words, which divides the string into a list of words, and its length can then be obtained with the mentioned function *len()*.

⌨ 3.1.3

How many words and characters does the file contain?

ASSIGNMENT

The input is a text file *data1.txt*. Find out how many words and how many characters (all) a text file contains.

The output should be:

the number of words is a **number**

the number of characters is a **number**

Example:

the number of words is 1

the number of characters is 5

3.2 Tokenization

3.2.1

Tokenization

Tokenization is the process of dividing the text into a set (list) of smaller parts, the so-called tokens. A token is actually a part of a whole, and can be understood in terms of sentences as well as words. A word is a part of a sentence and a sentence is a part of the entire text, or paragraph.

3.2.2

The first example will focus on using the **sent_tokenize()** function from the **nltk** library, which will create tokens in the form of sentences from continuous text.

```
from nltk.tokenize import sent_tokenize

text = "I sing sweet desires, desires for beauty, excited by
beauty, and in this echo of my soul, my world is completely
closed. She shines for me from the heights of the Tatras, she
flies to me from the fires of heaven, she moves my worlds. She
beckons to me from a hundred lives : Well, the center, the
element, the sky, the unity of my beauties, my Marina!"

print(sent_tokenize(text))
```

```
# vystup mozeme priradit do premennej

verse =  sent_tokenize(text)

print(verse[2])
```

The **sent_tokenize() f**unction 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.

3.2.3

The second and, in our opinion, more frequently used tokenization is the tokenization of sentences into individual words, or tokens. From the point of view of text processing, individual sentences consist of tokens and spaces. A token is therefore 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 problems:

- Dot at decimal places (eg number 2.06 or 2.06). The dot or comma character represents to the tokenizer the point at which the token can be separated. However, with numbers, it must be decided whether it will be appropriate to divide the number.

- The second problem with numbers is their (for better readability) "traditional" presentation with spaces (eg 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 problem of the so-called continuous forms (e.g.: within, on black, etc., Nitriansky region, Nitriansky self-governing region, etc.), where it is more appropriate to understand the continuous form as a whole and not to divide it into several tokens.

3.2.4

For the tokenization of sentences, it is advisable 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

text = "Ja sladké túžby, túžby po kráse spievam peknotou
nadšený, a v tomto duše mojej ohlase svet môj je celý zavrený.
Z výsosti Tatier ona mi svieti, ona mi z ohňov nebeských letí,
ona mi svety pohýna. Ona mi kýva zo sto životov: No centrom,
živlom, nebom, jednotou krás mojich moja Marína!"

print(word_tokenize(text))
```

In the next text, we will not discuss all tokenization options. The functions for tokenization with the possibility of defining your own regular expressions, or the possibility of training your own tokenizer to take into account the special characteristics of the analyzed text, are particularly interesting. More detailed information about these procedures can be found in the publication.

3.3 Lemmatization

3.3.1

Lematization is the process of changing a word to its basic form. It represents one of the possibilities of standardizing words in the language. In Slovak, there are certain conventions when creating a lemma, e.g. verbs are always in the indefinite form, nouns in the nominative singular, adjectives in the predicative position, etc.

An example of a lemmatizer using the *nltk* library:

```
import nltk
from nltk.stem import WordNetLemmatizer

wordnet_lemmatizer = WordNetLemmatizer()
text = "studies study student"

tokenization = nltk.word_tokenize(text)
for w in tokenization:
      print("{}, {}".format(w,
wordnet_lemmatizer.lemmatize(w)))
```

An example of a lemmatizer via the *spacy* library:

```
import spacy

nlp = spacy.load('en_core_web_sm')

sentence = ("Lemmatization is the process of converting a word
to its base form.")

doc = nlp(sentence)

for token in doc:
    print(token, token.lemma_)
```

3.4 Stemming

📖 3.4.1

We refer to the term **stemming** as the process of modifying a word to the basic root of the word by removing its prefixes and suffixes. This process is very similar to lemmatization mainly because they have the same goal - to reduce the total number of words in the document. The problem with stemming occurs with words whose root loses the meaning of the word, than when we have the word in its basic form.

Example of stemming for different forms of the word "close":

{closing, closes, closer, close} → **clos**

The most common problem with stemming occurs with words whose root does not make much sense as an independent word, and very strange word roots can often arise that do not always make sense, and thus their expressive value is not very great. Stemming also has its uses for certain NLP problems. One of the frequent uses of stemming is to correct words when writing.

📝 3.4.2

Most of the world's languages have their stemmer included in the nltk library, e.g. for the English language we know stemmers: **Porter Stammer** and **Lancaster Stammer**. Below we can see a sample of these two libraries.

```
from nltk.stem import PorterStemmer
from nltk.stem import LancasterStemmer

porter = PorterStemmer()
lancaster=LancasterStemmer()

print("Porter Stemmer")
print(porter.stem("closing"))
print(porter.stem("closes"))
print(porter.stem("closer"))
print(porter.stem("close"))
print("Lancaster Stemmer")
print(lancaster.stem("closing"))
print(lancaster.stem("closes"))
print(lancaster.stem("closer"))
print(lancaster.stem("close"))
```

3.5 Stop words (and why we should delete them...)

📖 3.5.1

Stop words

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

Typical stop words:

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 ...

📝 3.5.2

It is possible to use stop words for many languages within the NLTK library. After importing stop words, it is possible to display languages for which stop words exist in NLTK. Below we can observe a list of English stop words through the *nltk* library.

```
import nltk
from nltk.corpus import stopwords

stops = set(stopwords.words('english'))
print(stops)
```

📝 3.5.3

Subsequently, we can work with the word database. We come up with some text and use the for loop to remove the stop words:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
```

```
data = "This is just a sample sentence, showing off the stop
words filtration."

stopWords =set(stopwords.words('english'))
words = word_tokenize(data)
wordsFiltered = []

for w in words:
  if w not in stopWords:
    wordsFiltered.append(w)

print(wordsFiltered)
```

Frequently repeated words in the analyzed text sometimes unnecessarily suppress the meaning of other words. For example in the semantic analysis of the book "History of Castles and Chateaux of North America" it is not important how many times the word castle or castle occurs in individual chapters, because all chapters are actually about castles, and it is assumed that there will probably be the most of these words. Therefore, we often consider stop words to be words that appear frequently in most texts. For this reason, words are often searched using frequencies, i.e. the frequency of all words is determined and the most frequently used words are removed.

3.6 POS tagging

🕮 3.6.1

Marking of parts of speech (POS tagging)

Morphological annotation is the basic (and most common) linguistic information introduced especially in inflected languages. It contains word type and shape characteristics of words in context. It is usually preceded by lemmatization – assigning a basic (lexical) form to each word.

The result of the morphological analysis is the assignment of the so-called tags for individual words. For this reason, the so-called annotated ("tagged" in slang), i.e. assigning a short characteristic of a word within a sentence. A good aid in determining the morphological forms of words is the Slovak National Corpus project - https://korpus.sk/morpho.html. At this address you will find the full meaning of the individual parts of the tags.

3.6.2

Task: Determine the morphological markers of individual words in the sentence: *"My mother always said that miracles happen every day"*.

We will use the *stanza* library to solve the given task. We analyze the sentence with the already known nlp function.

```
import stanza

nlp = stanza.Pipeline(lang='en')
doc = nlp("My mother always said that miracles happen every
day.")
```

In addition to lemmatization, the command also performs morphological analysis. All you have to do is "read" its results. The word and lemma itself are found in the text and lemma properties. The new features we will use are:

- **upos** (or only pos) - the so-called universal part of speech tag,
- **xpos** – specific mark for the given language
- feats list of morphological properties for a specific language, i.e. "closer" properties such as gender, case, number, etc.

We will use the property **xpos** to determine the morphological markers.

```
for sentence in doc.sentences:
  for word in sentence.words:
    print(word.text + ' ( ' + word.xpos + ' ) ')
```

3.7 N-grams

3.7.1

N-grams

We consider an N-gram to be a group of elements, usually words, which is connected into a single unit. Its size is defined by the letter N. Until now, we worked with a unigram, i.e. with only one token. By the term bigram, we understand the division into elements into groups of two elements each. Trigrams have the size of three elements and we continue like this up to N-grams.

```
import nltk
```

```
text = "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."
tokens = nltk.word_tokenize(text)
```

```
bigram = list(nltk.bigrams(tokens))
trigram = list(nltk.trigrams(tokens))
ngram = list(nltk.ngrams(tokens, 4)) # we take 4
```

```
print(bigram)
# print(trigram)
# print(ngram)
```

We consider N-grams to be a very helpful mechanism for identifying entities. The frequency of occurrence of certain N-grams in documents is much higher for known multi-word entities. If we take bi-grams (New, York) and a completely random bi-gram such as (walked, him), it is obvious that the frequency of bi-gram (New, York) will be greater in the document. N-grams are also a useful tool for predicting other words in a sentence.

3.8 Data

📖 3.8.1

Data

All the mentioned data preprocessing options are mainly used for larger text sizes. What we need is a coherent set of related data, which we call a dataset. When choosing it correctly, we must take into account its relevance to our problem - EXAMPLE: if we want to analyze sentiment from the available data, a dataset with car brands will not be a suitable solution for this problem. In addition to the appropriate quantity of data, their quality is also important, mainly due to the better success of the algorithm used, at the same time we do not waste time with its cleaning.

In the topic of NLP, we can often encounter terms such as **dictionary** or **corpus**. The difference between them is that a dictionary is a list of words (in most cases sorted alphabetically) for which an explanation is given. Its disadvantage lies mainly in the omission of a large number of words and thus the context of words can be confused. The corpus is a structured, unified and very extensive source of language data. Thanks to corpora, we can e.g. observe language development. (Cermák, 2011).

There are a huge number of corpora, databases and data for different fields to choose from. It is not possible to list them all, so we will mention only some of them:

Dataset finders:

- Kaggle
- Dataset search from Google Inc.

3.8.2

WordNet

One of the very popular lexical databases is **WordNet**. It is a dictionary of words, where given words are arranged according to semantic relations. The nltk library offers us an interesting tool for discovering these relationships. The wordnet dictionary must be downloaded using the **download()** method before using it.

```
import nltk

nltk.download('wordnet')

from nltk.corpus import wordnet
```

Using the wordnet dictionary, it is possible to find out 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 try to find the word "joy" in wordnet.

```
syns = wordnet.synsets("joy")

print(syns)
```

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 still an "n" or "v" character between the word and the number. The character "n" indicates a noun, "v" a verb.

If we want to see words that have semantic, i.e. semantic relation to "joy" (translated as "joy"), we can list them using the **lemmas()** method.

```
for syn in syns[0].lemmas():
  print(syn.name())
```

```
for syn in syns[1].lemmas():
  print(syn.name())
```

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 list the words of all synsets where the word "joy" is found as follows.

```
for syn in syns:
  for lema in syn.lemmas():
    print(lema.name())
```

Additional information is also stored in synsets. For example, most synsets also contain a definition of the studied word.

```
print(syns[0].definition())
```

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 use of the word in the synset. This example, like the definition, may not be part of every synset.

```
print(syns[0].examples())
print(syns[1].examples())
```

3.8.3

Wordnet

In addition to 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, anatonyms are found only in pairs, i.e. word and its antonym. In the following example, we list all synonyms for the word "joy" together with the antonym of these synonyms (if any).

```
from nltk.corpus import wordnet

word = "joy"
synonyms = []
antonyms = []

for syn in wordnet.synsets(word):
  for lema in syn.lemmas():
    synonyms.append(lema.name())
    if lema.antonyms():
      antonyms.append(lema.antonyms()[0].name())

print("Synonyms:")
print(set(synonyms))
print("Antonyms:")
print(set(antonyms))
```

An interesting functionality of the nltk library is the detection of the so-called semantic similarity of words. For two words, we can find out their level of similarity, this metric is calculated by searching for the number of common synsets, synsets of synonyms, etc. The greatest degree of similarity is 1, words with different meanings have a similarity close to 0.

```
w1 = wordnet.synset('joy.n.01')
w2 = wordnet.synset('joyousness.n.01')

print(w1.wup_similarity(w2))
```

```
w1 = wordnet.synset('joy.n.01')
w2 = wordnet.synset('sorrow.n.01')

print(w1.wup_similarity(w2))
```

```
w1 = wordnet.synset('joy.n.01')
w2 = wordnet.synset('mouse.n.01')

print(w1.wup_similarity(w2))
```

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". For control, we present in the last example 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.

In conclusion, we present "proof" that "soccer brings more joy to a person than chess".

```
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))
```

3.8.4

WordNet is excellently processed for English within the nltk library. Slovak WordNet is currently processed in the form of a file containing: synset number, part of speech, words containing the given synset (separated by a semicolon) and a more detailed explanation of the synset. Slovak WordNet can be found at https://korpus.sk/WordNet.html

3.8.5

Wordnet - Question

What can we find out through Wordnet for the given word?

- antonym
- translation

- synonym
- lemma

Text representation as vector

Chapter 4

4.1 Introduction to Vector models

🕮 4.1.1

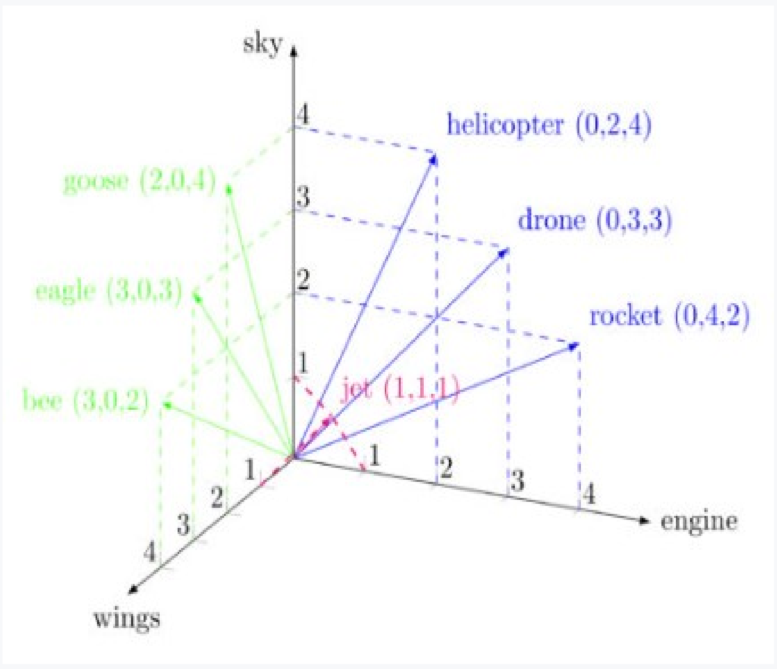
Introduction to vector models

In order for the computer to be able to understand the text, we have to convert the text into numbers. We call this process **word embedding**.

🕮 4.1.2

Word vector

In order for a computer to understand the meaning of words in natural language as best as possible, it is necessary to represent the words in a certain way in some form of water. The most common way of machine representation of a word is a multidimensional numerical **vector**. This is subsequently used as an input to more complex machine learning models - understanding the meaning of the text, voice recognition, machine translation and the like. An example of how such a word vector can be displayed:



The simplest way to represent the text by machine is to enter the number of occurrences of individual words in the text into a vector. We also call this method **Bag of words**.

4.1.3

Bag of Words

It is one of the simplest models out there. Its goal is to count how often a word occurs in the text, it does not solve the word order and collect sentences.

Bag-of-words is a simple model to understand, implement and offers us great flexibility in customizing specific textual data. Nevertheless, it has several drawbacks:

· **Vector size** – can be huge for large dictionaries

· **Semantics** - does not take into account the context and meaning of the words in the sentences. So a word that occurs frequently in the text will have the most weight, even though it is not a word that describes the entire document.

It is for this reason that we will continue to deal with models focused on the vector representation of words.

Below you can see how the BoW algorithm works. To improve the algorithm, we also used text preprocessing steps such as tokenization and removal of stop words.

```
import pandas as pd
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import CountVectorizer

docs = ["Machine learning uses historical data to predict
output values.",
             "It is seen as a part of artificial
intelligence.",
             "Machine learning programs can perform tasks
without being explicitly programmed to do so."]
```

```
def preprocess(document):
    # vsetky pismena nastavime na male
    document = document.lower()
    # tokenizuj do slov
    words = word_tokenize(document)
    # vymaz stop slova
    words = [word for word in words if word not in
stopwords.words("english")]
    # spoj slova, nech su naspat vety
    document = " ".join(words)

    return document

docs = [preprocess(doc) for doc in docs]
```

```
vectorizer = CountVectorizer()
bow_model = vectorizer.fit_transform(docs)
print(bow_model.toarray())
```

4.1.4

What is word embeddings?

- Converting words into numerical vectors
- Visualizing words in high-dimensional space
- Generating new words based on existing ones
- Compressing large text datasets

4.2 Representation of text documents

4.2.1

Models of representation of text documents

We use them when we want to process text effectively using text mining methods. Thanks to these models, we can formalize documents, especially with classification and clustering algorithms. For text documents, we know two types of access to data:

- **data with dependencies** - the approach is used in cases where it is necessary to know how words in a sentence follow one another, i.e. where exactly in the sentence they are located. Such an approach is used when examining the grammatical rules of the language, the style of the language, etc. However, these are mostly computationally and memory-intensive operations.
- **data without dependencies** - In most text mining problems, however, it is not necessary to know how words follow one another. It is enough if we know the information that the word is in the given sentence (and it is not important

in which part of the sentence). With this approach, we can significantly reduce the memory requirement of the solved problem.

The most widespread method of text analysis is probably document search. Due to the fact that these are often large files, comparing entire documents, paragraphs or sentences is not appropriate. In practice, therefore, the so-called document model that represents the representation of words in the document.

4.2.2

Boolean model

The simplest model is the so-called Boolean model. Its name already implies that it uses Boolean logic operators, which we know as AND and NOT, respectively. 1 and 0. For each document, it keeps information about the words found in it. The advantages of such a representation are mainly in a very simple and clear term search.

Let's say we have three vectors at our disposal:

· First (1,0,1,0,1)

· Second (1,1,1,0,0)

· Third (0,1,0,1,0)

If we work with Boolean vectors, we can multiply them, count them, etc. relatively easily. Our task is to find a vector that belongs to the first vector and, at the same time, does not belong to the third vector. The result would be as follows:

First AND NOT (Third)

(1,0,1,0,1) AND NOT (0,1,0,1,0) = 10101 * 10101 = 10101

Another advantage of the Boolean model is its use in searching for similarities. To calculate the similarity of vectors, we use the scalar product of vectors. This is an operation on two n-dimensional vectors. The result of this operation is a number (that is, a scalar, not a vector).

We calculate the scalar product as follows:

$$\vec{x} \cdot \vec{y} = (x_1 y_1 + x_2 y_2 + \cdots + x_n y_n)$$

4.2.3

We have vectors A (1,1,0,1), B (0,1,1,1), C (0,1,0,0), D (0,0,1,1), E (1,0,0,0), F(1,1,0,0).

Find which vectors have the highest similarity to vector A. Use the scalar product.

Write the answer in **LETTER and LETTER**.

4.2.4

Vector model

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 retrieval.

The vector model of the document is more applicable (the Boolean model also used vectors, the so-called Boolean vectors, whose elements take on the value 0 or 1). In the vector model, we also assume an invariant order of documents (in our example of fairy creatures) and also of words/terms (in our example of questioner properties). In contrast to the Boolean model, we can take into account the frequency of occurrence of words, 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. The example uses the **CountVectorizer()** method. It 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.

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()

doc = ["good, good good good bad good bad good good good good
good","good bad bad bad bad bad"]

vectorizer.fit(doc)

print("Slovník: ", vectorizer.vocabulary_)

vector = vectorizer.transform(doc)
```

```
print(vector.toarray())
```

4.2.5

Probabilistic model

He describes this model in more detail as a model operating on the principle of probabilistic evaluation. The system tries to estimate the probability of occurrence as best as possible and ranks the results in ascending order of relevance. This model is also based on the Boolean model. Its main principle is based on the calculation of the probability to what extent a document belongs to a set of documents that are relevant to the ideal set of documents in response to a query.

This relationship is expressed through the formula:

$$sim(d_j, q) = \frac{p(\frac{R}{q}, \bar{d}_j)}{p(\frac{\bar{R}}{q}, \bar{d}_j)}$$

The well-known models also include the model of distributed semantics, which is a generalization of the vector model.

4.2.6

TF-IDF

Relative frequencies are the basis of the TF-IDF statistical model. In the model, the frequency of the term TF (Term frequency) is defined, this represents the mentioned relative abundances. 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, w be any term in the document, then we 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 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), i.e. document frequency and express it as

$$df(t,D) = \sum (d \in D : t \in d),$$

, where D is the 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 the number of documents in the 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.

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 business cards. Most of the source code is practically the same as in the previous chapter, only the method for calculating the TF-IDF is replaced.

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer

doc = ["good, good good good bad good bad good good good good
good","good bad bad bad bad bad"]

vectorizer = TfidfVectorizer()

vectors = vectorizer.fit_transform(doc)

feature_names = vectorizer.get_feature_names_out()
dense = vectors.todense()
denselist = dense.tolist()

df = pd.DataFrame(denselist, columns=feature_names)

print(df)
```

4.2.7

What type of model is TF-IDF?

- Vector
- Probabilistic
- Statistical
- Conversion

More advanced word representation tools

Chapter 5

5.1 Libraries

📖 5.1.1

TensorFlow

The library, which is designed and developed by Google, is among the most popular libraries for developing applications using machine learning. It demonstrates the extreme adaptability of various options for complex neural network algorithm designs, along with high performance in numerical calculations. It goes without saying that Google also uses this library when working on its projects, so there is no need to doubt the quality of this library.

At the same time, TensorFlow is a low-level library, which we consider its disadvantage. The complexity of working with it is relatively high, and the complexity of finding errors in the code is also related to it. This negative is slightly offset by the use of the Keras library, which provides a simpler, higher-level interface to the TensorFlow library.

📖 5.1.2

Keras

Keras provides high computing power thanks to the backend running the TensorFlow library. It is TensorFlow that enables Keras to create sophisticated models of neural networks such as multilayer perceptron networks, convolutional neural networks or recurrent neural networks. The advantage of Keras over TensorFlow is the ease of working with it. Due to the fact that Keras works at a higher level, many details are moved to the abstract, and thus the code becomes much clearer and concise. Instead of solving how to correctly get the output of one function and put it as input to another, the developer can focus on the essence of his goal in solving an artificial intelligence problem, and the details of these processes are handled for us on the backend. Thanks to these advantages, Keras is one of the most popular variants when working with neural networks.

5.1.3

PyTorch

PyTorch is an open-source library based on the Torch library. It is the recommended tool for building machine learning and deep learning projects. It has its own basic data structure known as a tensor - a designation for a number, vector, matrix, but any n-dimensional array containing elements of the same data type.

We have already met the library that deals with multidimensional arrays - numpy. The difference between PyTorch and numpy is that tensors are able to automatically calculate the gradient and GPU support makes tensor operations much more efficient.

5.1.4

Gensim

GenSim can work with large text corpora more efficiently than other libraries. It specializes in identifying semantic similarity between two documents through vector space modeling and a set of topic modeling tools. It perceives the content of documents as sequences of vectors and clusters. Using the NumPy library, it can optimize memory usage and data processing speed.

The GenSim library has equally wide uses, such as:

- vectorization of text,
- summarizing the text,
- text generation,
- text analysis,
- semantic search,
- data exploration.

5.1.5

Which library is best for text vectorization, text generation, or semantic search?

5.2 Vector models

5.2.1

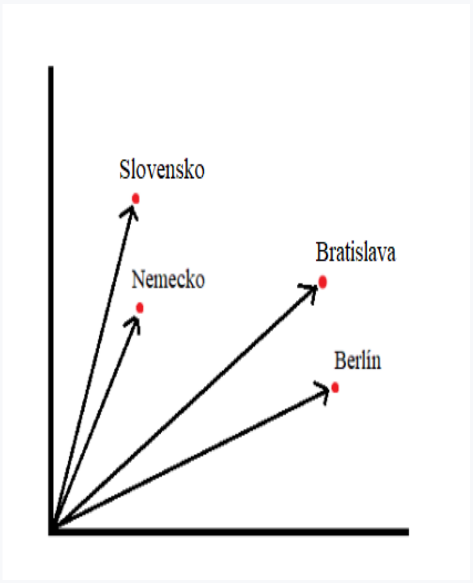
Word2Vec

This is one of the most widely used models for creating a vector representation of words, which is often referred to as word embedding. It uses a two-layer neural network and was created in 2013 from the need to capture the semantic similarity of words.

Word2Vec uses cosine similarity to determine the similarity between individual vectors:

$$cos(\theta) = \frac{A \cdot B}{||A|| \cdot ||B||} = \frac{\sum_{i=1}^{n} AiBi}{\sqrt{\sum_{i=1}^{n} Ai^2}\sqrt{\sum_{i=1}^{n} Bi^2}}$$

The smaller the angle, the higher the similarity. Example:

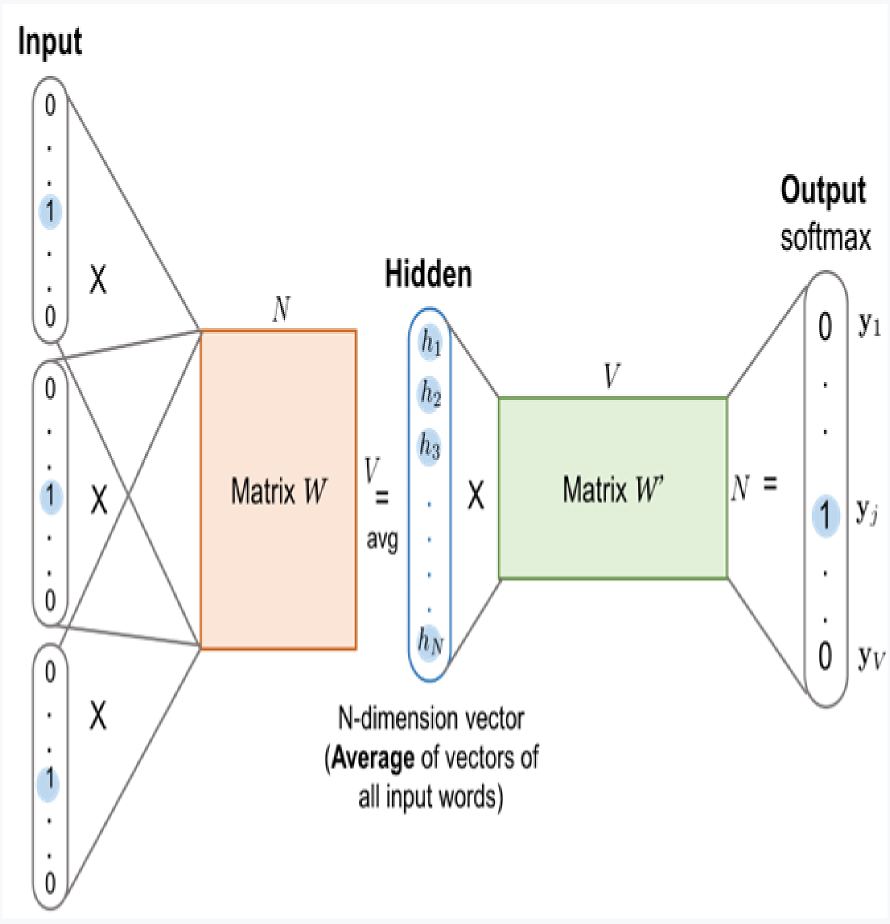


PRINCIPLE OF THE Word2Vec MODEL

The model consists of 3 layers:

- input,
- hidden
- output.

The input layer has as many neurons as there are words in the dictionary. The hidden layer usually contains fewer neurons than the input layer, most often between 100-300, which is actually the number of dimensions of the resulting vector. The output layer has a softmax activation function in it. Between these layers there are matrices of weights that are used for training.



Vectors for individual words are tried to be obtained by two algorithms:

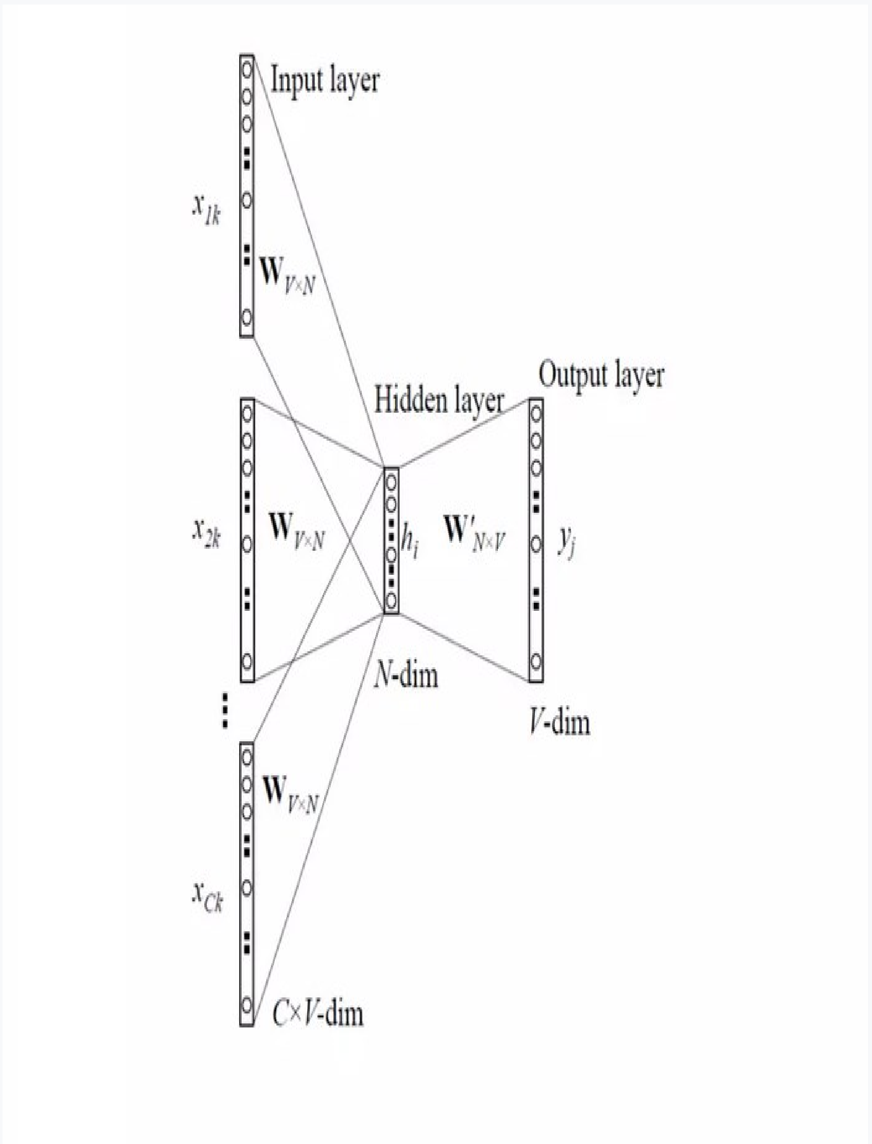
- Skip-gram - Works like the opposite of CBOW. It receives one word for input and searches for those words that are within the context in its immediate vicinity.

- CBOW (Continuous bag of words) – the algorithm tries to predict the missing word depending on the other words in the document.

5.2.2

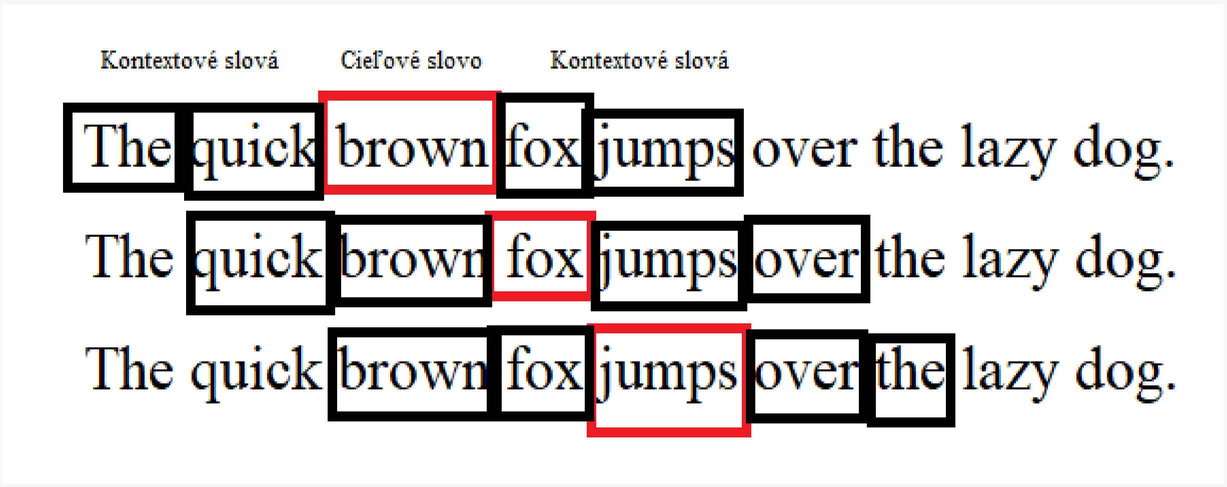
Word2Vec - CBOW

Continuous Bag of Words was created from the original BOW (Bag of Words) method by its extension. Its goal is to predict the target word based on the surrounding words in its vicinity.



CBoW version: predict center word from context

The image below is a visualization of the CBOW architecture popup of length 5 (kontextové slová = context words, cieľové slovo = target word).


Kontextové slová
Cieľové slovo
Kontextové slová
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.

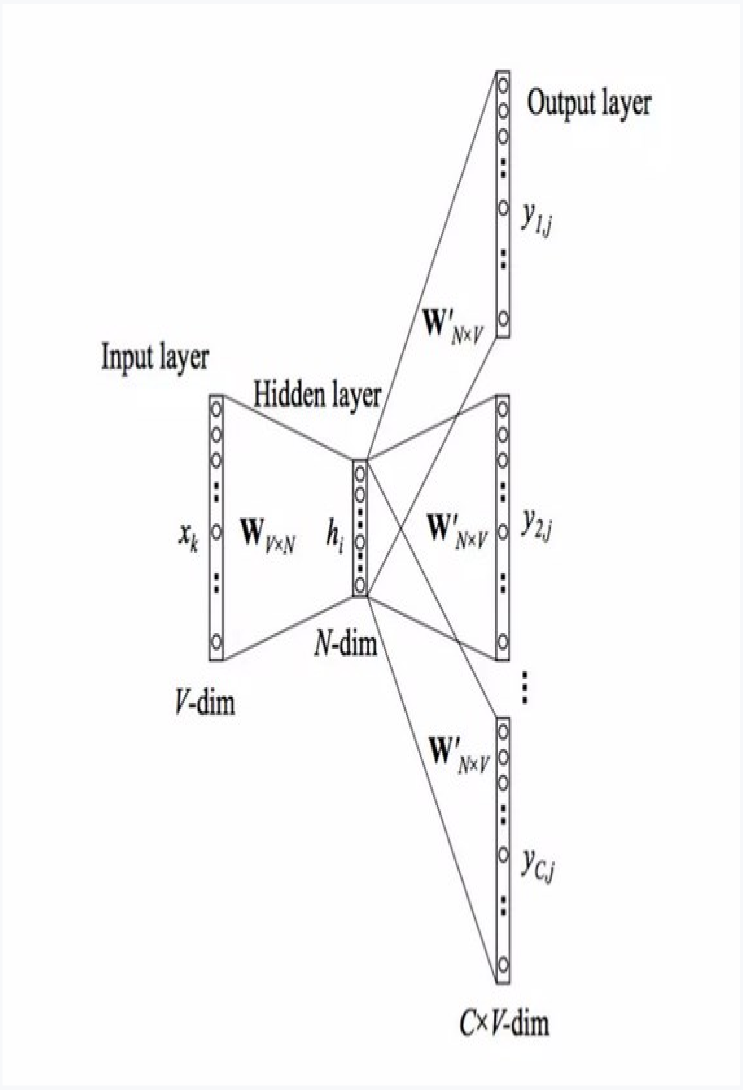
With a larger window length value, more connections between context words can be captured.

The advantage of this method is high speed and better representation of words that occur more often in the text.

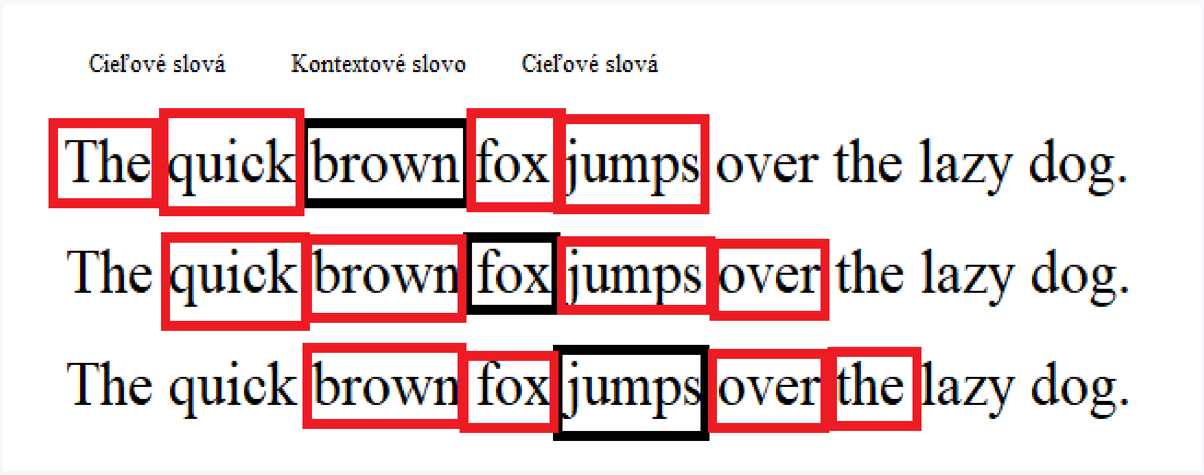
5.2.3

Word2Vec - Skip-gram

This model is the opposite of the CBOW model. Based on the target word, all surrounding words are predicted. The method shows significantly better results with smaller volumes of data and better represents rare words. On the other hand, it is a slower model than CBOW.



The figure below shows a visualization of the Skip-gram architecture popup of length 5 (cieľové slová = target words, kontextové slovo = context words).


Cieľové slová
Kontextové slovo
Cieľové slová
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.

We can definitely describe Word2Vec as one of the best implementations of vector word representation. It provides us with wide possibilities of use not only with text data (e.g. when creating music playlists).

Word2Vec is not the only technique that represents words using a unique vector. Word2Vec comes with a newer and faster version called FastText. another alternative is GloVe and, last but not least, Doc2Vec. We will describe all the mentioned techniques in the next subsections.

🕮 5.2.4

FastText

The FastText method was designed in 2016 by Meta.

The algorithm of the FastText method contains pre-trained models with English owl vectors learned from wikipedia and webcrawler. It uses the technique of dividing words into character N-grams, often between three and six characters.

<u>Example:</u>

word = where

n = 3

<wh, whe, her, ere, re>

Splitting words should guarantee a better vector representation of words that do not occur that often.

🕮 5.2.5

GloVe

One of the variants of Word2Vec, which also serves to train the vector representation of words, is GloVe (Global Vectors).

The learning of the Word2Vec model is based on the dependencies of the target words with their context. But the mistake is that the method does not respect the more frequent frequency of some words in the text. And it is GloVe that emphasizes the frequency of occurrence of words in the text. The combination of word vectors is therefore directly related to the probability that two words occur together in this combination.

ADVANTAGES

GloVe is a very fast method for training and is scalable even for large vocabularies. It provides good results even with a smaller dictionary and vectors with fewer dimensions.

DISADVANTAGES

It needs a lot of memory and is sensitive to parameter initialization.

📖 5.2.6

Doc2Vec

Doc2Vec is an extension of the Word2Vec method and is strongly based on its principle. Its goal is to calculate a vector representation of entire documents independent of their length.

The problem can be caused by the fact that the documents are not necessarily composed of a logical structure.

The main difference with Word2Vec is the extension of the so-called Paragraph ID, which is used to identify a specific document.

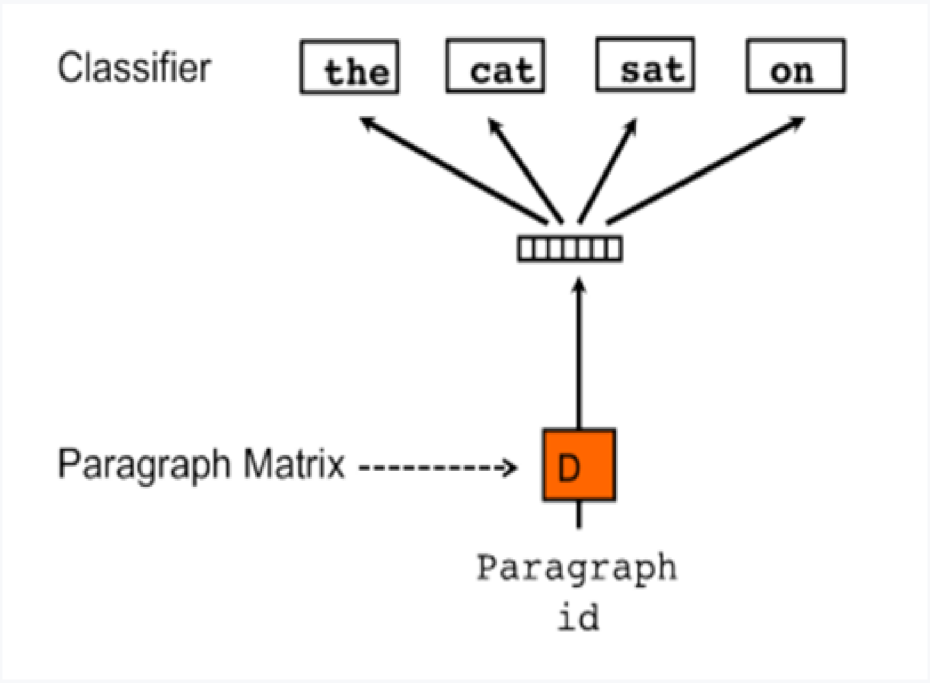
Like Word2Vec, Doc2Vec uses two techniques to train sentence vectors, namely:

- PV-DBOW (Distributed Bag of Words of Paragraph Vector), which neglects the context of words in the input,
- PV-DM (Distributed memory model of Paragraph Vectors) works on the CBOW principle.

PV-DBOW

It is an extension of the Skip-gram model. Instead of word prediction as in the original model, it uses the paragraph vector to classify the document. The advantage of this model is its speed, as it uses little memory because it does not need to store word vectors.

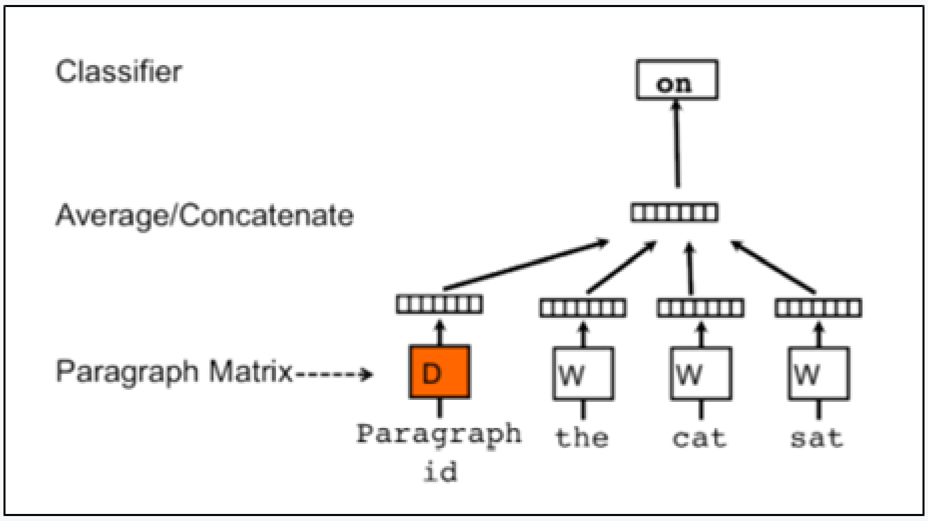
The image below is a graphical representation of the PV-DBOW model.



PV-DM

It is an extension of the original CBOW model and is added with a Paragraph ID value. This vector serves as a unique identifier of the document, part of the text and expresses the numerical representation of the document. The model itself serves as a memory that remembers what is missing in the text in the given context or expresses the theme of the document.

Its graphic representation in the image below.



Model progress:

1. Word vectors and paragraph vectors are initialized randomly,

2. Paragraph vectors are always assigned to only one document. Word vectors are in turn assigned to each document.

3. Word and paragraph vectors are subsequently averaged but concatenated and inserted into a stochastic gradient, and the gradient is subsequently obtained by backpropagation.

5.2.7

Word2Vec uses cosine similarity to determine vector similarity.

- True
- False

Sentiment analysis

Chapter 6

6.1 Sentiment analysis

📖 6.1.1

Introduction to sentiment analysis

For more than 20 years, they have been the creators of the available Internet as well as its content. In past years, this change was often referred to as a "buzzword" - Web 2.0. This term expressed the established designation for the stage of web development, in which the content of web pages was replaced by a space for sharing and joint creation of content. It covers the period from 2004 to the present. Discussion forums are one of the means of contributing to web content. These are also a source of knowledge about users' opinions, feelings and attitudes.

Most of us rather choose and look for well-known portals and opinions of people who have been to the destination, or have also stayed in the chosen hotel. If you do not find this information in the reviews/ratings, probably the choice of hotel or we will consider destinations. We may proceed similarly in the case of purchasing goods. Discussion forums and reviews are therefore often the source of the decision to purchase a product or service.

Discussion forums are often analyzed by political parties as well. They can find out what kind of "enthusiasm" citizens have met with their intended electoral program bodies, or whether citizens were very angry about their latest case, etc.

However, with the increase in the number of posts in the discussion forum, these opinions and attitudes become difficult for a person to read and process. Provide a basic overview and how much information the discussant assigns to goods, accommodation, etc. Information about the number of "likes", i.e. thumbs up and thumbs down, is similar. However, important information is found mainly in the texts of reviews and discussions. Their automatic processing is sentiment analysis (sentiment analysis). Often this discipline is also referred to as opinion classification, opinion mining (sentiment classification, sentiment analysis, opinion mining, opinion mining, etc.). Sentiment analysis focuses on the subjective aspects of the text, attitudes, opinions, feelings of the author, which refer to a certain object (goods, service, place, event, etc.), the authors of the posts always react to something.

📖 6.1.2

Sentiment analysis using a list of 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", "terrible" and so on. Basic sentiment analysis is therefore 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.

6.1.3

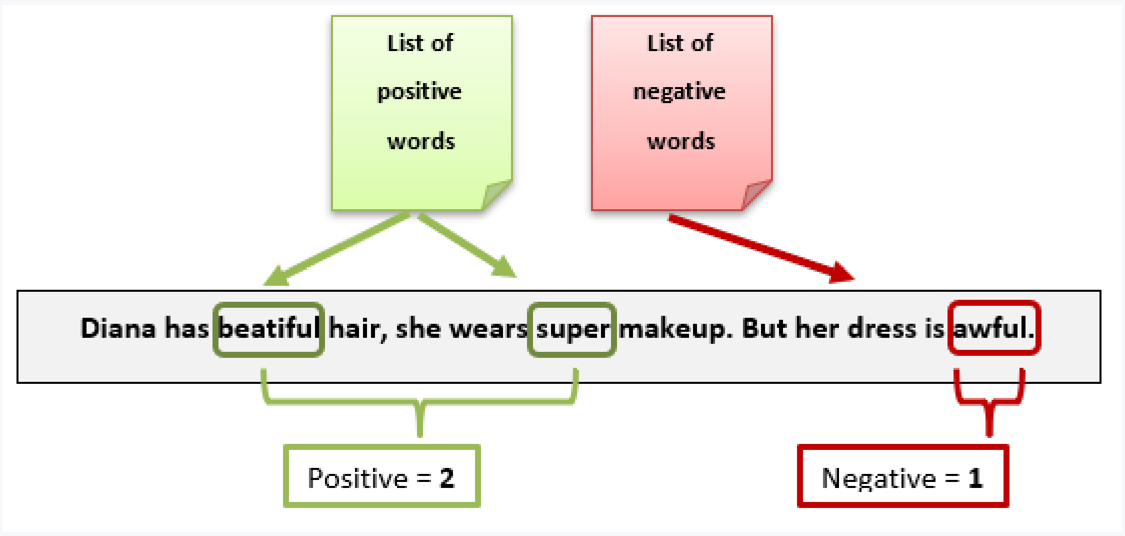
Sentiment analysis using a list of positive and negative words

Determining the sentiment of a post, in the case of a sentence, is a classification task. It is therefore necessary to say about each sentence whether it is positive, neutral or negative. Suppose there are lists of positive and negative words. Our lists of these words might look like this:

List of positive words	List of negative words
Good, great, super, beautiful, delicious, …	Bad, terrible, awful…

In the analysis itself, all you have to do is count the number of negative and positive words used in the post. As an example, we will classify the sentiment for the sentence:

„Diana has beatiful hair, she wears super makeup. But her dress is awful.


List of positive words
List of negative words
Diana has beatiful hair, she wears super makeup. But her dress is awful.
Positive = 2
Negative = 1

However, simple classification of sentences into these groups is seldom 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 created classifier, 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 like this:

```
from nltk.tokenize import word_tokenize

positive = ['good', 'beautiful', 'super', 'great']
negative = ['bad', 'terrible', 'awful']
```

We tokenize the sentence and insert the created tokens, in our case words, into the list.

```
sentence = "Diana has beautiful hair, she wears super makeup.
But her dress is awful."

sentence_token = word_tokenize(sentence)
sentence_token
```

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("Count of positive words: ", pos)
print("Count of negative words: ", neg)
```

It is clear from the result that the theorem is positive. We can even express a degree of positivity. For the sake of completeness, we also present this relatively simple source code.

```
diff = pos - neg
if(diff>0):
  print("Sentence is positive")
  measure = pos / (pos+neg)
  print("Positivity measure: ", measure)
if(diff<0):
  print("Sentence is negative")
  measure = neg / (pos+neg)
  print("Negativity measure: ", measure)
if(diff==0):
  print("Sentence is neutral")
```

6.1.4

If we want to split each document into a collection of tokens, we have to use:

- import nltk
- from nltk.stem import WordNetLemmatizer
- from nltk.stem import PorterStemmer
- from nltk.tokenize import word_tokenize

6.1.5

The given classification of sentiment is quite simple. It has several problems:

1. There must be a list of positive and negative words. If it doesn't exist, a more sophisticated solution than just "just wondering what words to put in there" has to be created.
2. 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". How then to assign the weight of sentiment to individual words?
3. Sometimes just 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.
4. Arguably the biggest problem with sentiment analysis (not just our simple classifier) is irony. The traditional sentence of teenagers "well... it's really good" probably won't be positive despite the positive word used.

6.1.6

Problems 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 your own 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 our own 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 the centimeter. The classifier for these evaluations will teach you to correctly evaluate positive and negative reviews. Although we can imagine a very complicated process under the word "learned / trained", the whole "learning" will consist in our case of creating a list of positive and negative words with the right weight of sentiment.

Assessments are used to create the classifier, which also includes a graphic representation of the assessment, such as "like", "thumbs up", or rating expressed in stars. The picture shows an example of two approaches to evaluation (thumbs up/down, stars). If there is any additional sentiment information, we can create an input file for our classifier from such an evaluation.

However, in practice there are more sophisticated solutions. Their principle is very similar. A classifier is created, i.e. trained, on a sample of existing rated texts (i.e. texts to which information about their sentiment is assigned). We are offered a number of classification algorithms for training the classifier, we do not have to limit ourselves to calculating the sentiment weight of words. To prepare the text, TfidfVectorizer is used, 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.

6.1.7

In sentiment analysis:

- we create an answer based on a question
- we extract of key words from the document
- we assess the emotional content of a document
- we analyze of word structure according to their part of speech (POS tag)

6.2 Fake news analysis

📖 6.2.1

Fake news is currently the biggest threat to the developed world. In addition to the increase in the use of social networks, mainly for communication, there is also a high increase in the spread of fake news, hoaxes and other half-truths. People perceive social networks mainly as a space for expressing their opinions, but also as a space that brings together people with similar views.

If you release a false message on a social network, your close friends may still ask you where you came up with it. But the friends of your friends, who got the message by forwarding or liking one of their loved ones, will no longer contact you about the truth of the information. Anyone who forwards such a fake message adds a stamp of their own credibility to it. In addition, if the mentioned message (thanks to virality) reaches you from several loved ones at a similar time, it is not difficult to succumb to the delusion that it is serious information. In addition, by spreading the message virally (on social networks), you involve a much larger number of people in the communication. It's like people copying your fake newspaper for free on their photocopiers and voluntarily offering it to other people on the street. Most of us only give social networks a "quick glance". And so the time/space to confront the source of the information (compared to someone saying it to your face) is significantly less. You just register that this message passed you by.

The general awareness that the media has the ability to cross-check each other only exacerbates this problem. It is convenient to slip into the belief that when such a large number of people before me had the opportunity to verify the validity of the information, they certainly did.

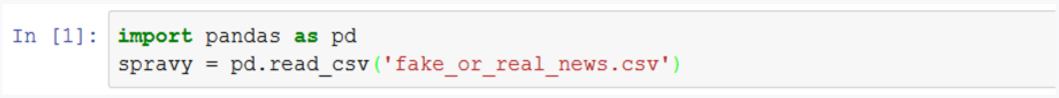
In the case of newspapers, both the spreader (publisher of the newspaper) and the recipient of the message had to pay for spreading a mass lie. In the case of social media, not only the costs of the end receiver, but also the costs of the broadcaster have disappeared. At the same time, this has "democratized" the approach to creating (fake) news and gives even relatively small groups of people a chance to trigger mass hysteria. Putin's or Trump's strategists understood that reporting fake news on social networks is a much more important weapon today than being on all televisions.

There are several ways in which informatics can help in the fight against fake news. One of them is their classification. If there is a set of "true" news and a set of fake news, we can train a classifier that can classify new news with some accuracy. In the next text, we will create a fake news classifier, while we will use the previous knowledge with text processing.

6.2.2

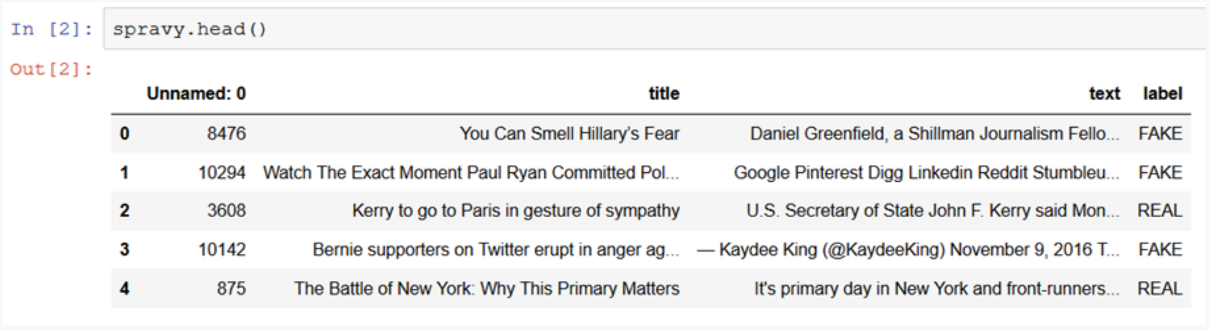
We have a dataset available at:
https://github.com/GeorgeMcIntire/fake_real_news_dataset.

We import the downloaded file into the table using the Pandas library.



```
In [1]: import pandas as pd
        spravy = pd.read_csv('fake_or_real_news.csv')
```

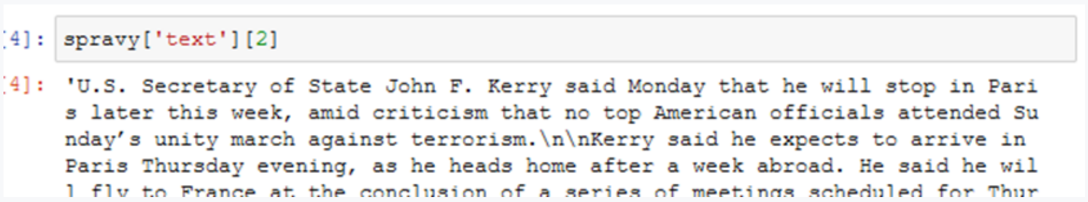
We can simply examine the data structure of the source file, currently stored in the management variable.



```
In [2]: spravy.head()
Out[2]:
```

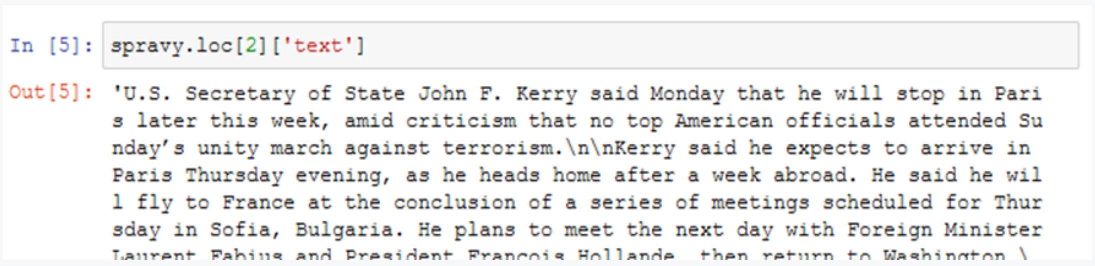
	Unnamed: 0	title	text	label
0	8476	You Can Smell Hillary's Fear	Daniel Greenfield, a Shillman Journalism Fello...	FAKE
1	10294	Watch The Exact Moment Paul Ryan Committed Pol...	Google Pinterest Digg Linkedin Reddit Stumbleu...	FAKE
2	3608	Kerry to go to Paris in gesture of sympathy	U.S. Secretary of State John F. Kerry said Mon...	REAL
3	10142	Bernie supporters on Twitter erupt in anger ag...	— Kaydee King (@KaydeeKing) November 9, 2016 T...	FAKE
4	875	The Battle of New York: Why This Primary Matters	It's primary day in New York and front-runners...	REAL

The parameter whether the message is true or false can be found in the label column. It is obvious that the content of individual messages is in the text column. To display the entire text of the message in the selected line, we can use the call using the so-called indexes. For example, for the message text in the second line, the message display would look like this.



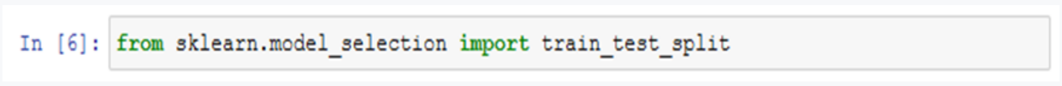
```
4]: spravy['text'][2]

4]: 'U.S. Secretary of State John F. Kerry said Monday that he will stop in Pari
    s later this week, amid criticism that no top American officials attended Su
    nday's unity march against terrorism.\n\nKerry said he expects to arrive in
    Paris Thursday evening, as he heads home after a week abroad. He said he wil
    l fly to France at the conclusion of a series of meetings scheduled for Thur
```

The second option for reading the entire message text, typical for programmers, is to use the loc indexer (or the iloc indexer that works similarly). Indexer loc selects primary rows, according to the index, i.e. table headers. In our example, e.g.



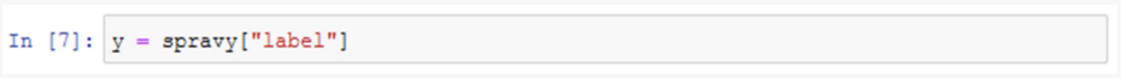
```
In [5]: spravy.loc[2]['text']

Out[5]: 'U.S. Secretary of State John F. Kerry said Monday that he will stop in Pari
        s later this week, amid criticism that no top American officials attended Su
        nday's unity march against terrorism.\n\nKerry said he expects to arrive in
        Paris Thursday evening, as he heads home after a week abroad. He said he wil
        l fly to France at the conclusion of a series of meetings scheduled for Thur
        sday in Sofia, Bulgaria. He plans to meet the next day with Foreign Minister
        Laurent Fabius and President Francois Hollande, then return to Washington.\
```

After familiarizing ourselves with the data, we can create and train the classifier. One of several options is to use the scikit-learn library. For now, we will need parts of it to create document vectors and to split the set into training and testing subsets.



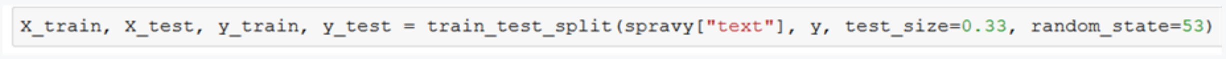
```
In [6]: from sklearn.model_selection import train_test_split
```

We will select the dependent variable that I will classify



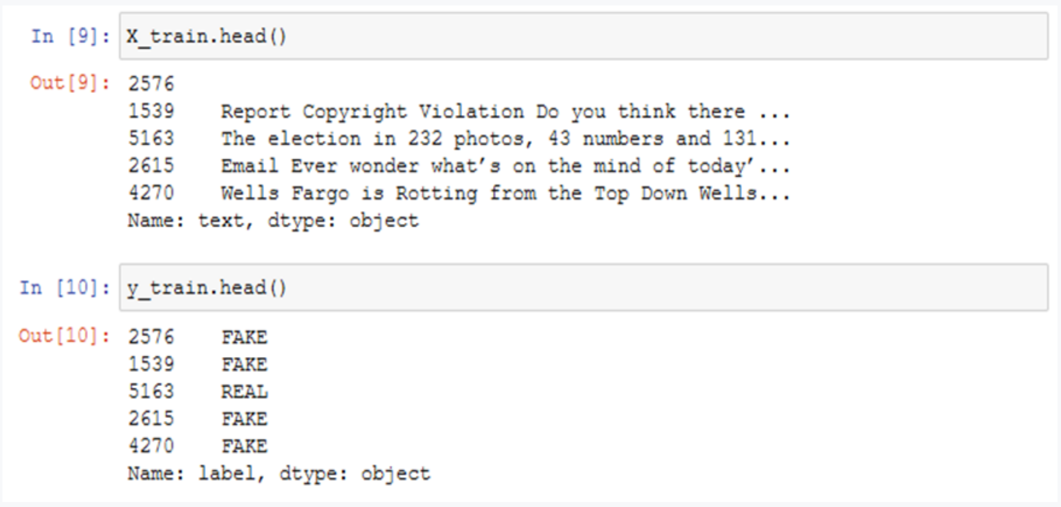
```
In [7]: y = spravy["label"]
```

and we divide the set into test and training data.

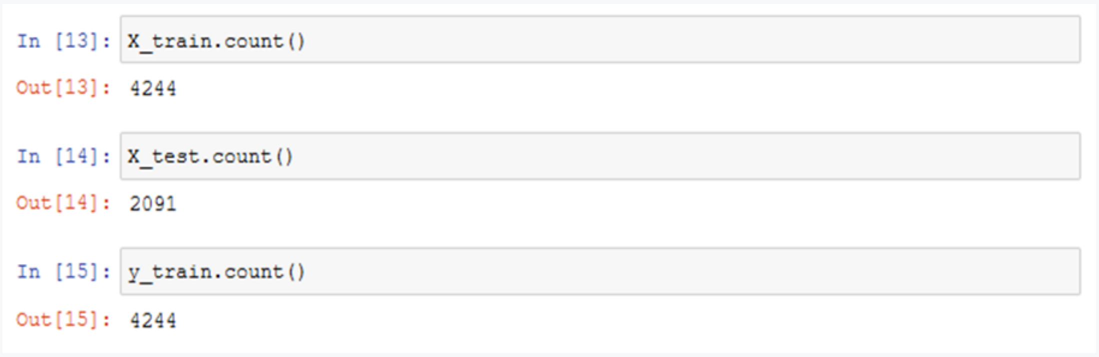


```
X_train, X_test, y_train, y_test = train_test_split(spravy["text"], y, test_size=0.33, random_state=53)
```

The first two parameters are fields with the independent and dependent variable (both must have the same size), followed by the specification of other parameters. In our case, using the test_size parameter, we divided the set in the proportion of 33% for test data and the remaining 67% for training data (the value is given from the interval <0.1>). The remaining random_state parameter specifies the initial setting of the random number generator. The function has four outputs that we will use for classification training. We can list some of the created subsets, similarly we can also list the number of messages in the created subsets, e.g.

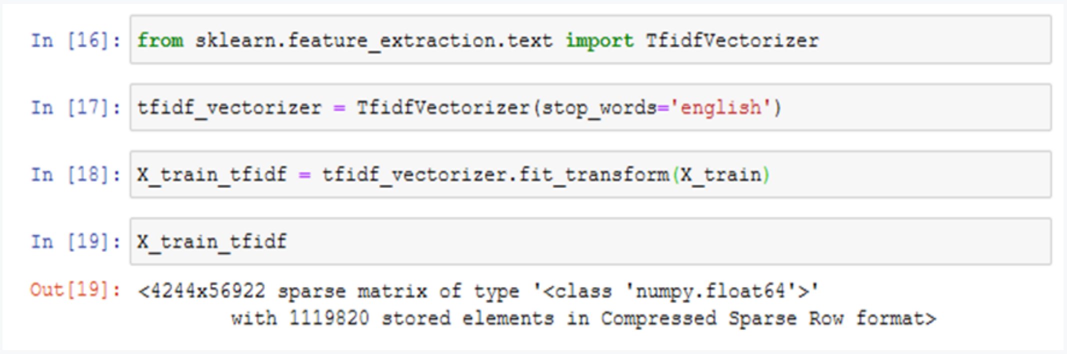


```
In [9]: X_train.head()

Out[9]: 2576
        1539    Report Copyright Violation Do you think there ...
        5163    The election in 232 photos, 43 numbers and 131...
        2615    Email Ever wonder what's on the mind of today'...
        4270    Wells Fargo is Rotting from the Top Down Wells...
        Name: text, dtype: object

In [10]: y_train.head()

Out[10]: 2576    FAKE
         1539    FAKE
         5163    REAL
         2615    FAKE
         4270    FAKE
         Name: label, dtype: object
```



```
In [13]: X_train.count()
Out[13]: 4244

In [14]: X_test.count()
Out[14]: 2091

In [15]: y_train.count()
Out[15]: 4244
```

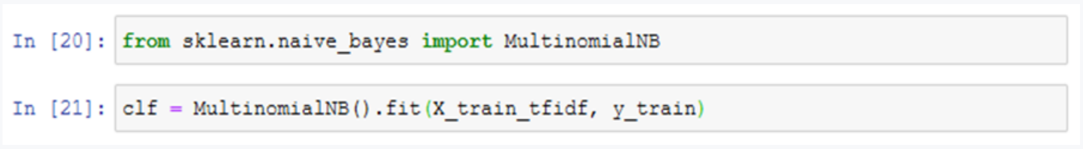
If we have messages divided into training and test samples, it is necessary to convert them into TF-IDF vectors. We will use the well-known TfidfVectorizer method, with which, in addition to creating a vector, we can remove stop words from input texts.



```
In [16]: from sklearn.feature_extraction.text import TfidfVectorizer

In [17]: tfidf_vectorizer = TfidfVectorizer(stop_words='english')

In [18]: X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)

In [19]: X_train_tfidf
Out[19]: <4244x56922 sparse matrix of type '<class 'numpy.float64'>'
         	with 1119820 stored elements in Compressed Sparse Row format>
```

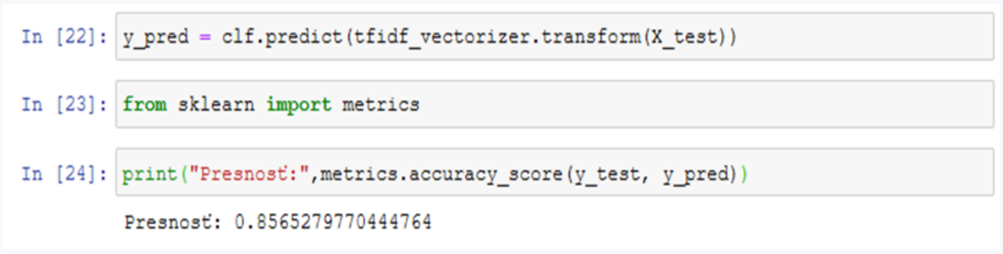
Please note that the fit_transform method creates document vectors from the training set. We will use a similar transform method, which will convert the new texts into TF-IDF vectors. The difference is in the need to create a fixed order of terms/words whose values will be represented by vectors. The first fit_transfom method, in addition to creating vectors, also creates the aforementioned fixed order of words, i.e. the vector index is set. The second transform method converts text to vectors, where the vector index already exists.

If we have generated TF-IDF vectors, we can use them as input for training the classifier. We chose probably the simplest Naive Bayes classifier from the available ones. We will train him with familiar commands.



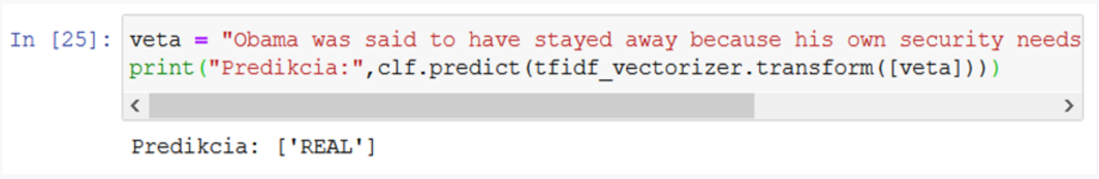
```
In [20]: from sklearn.naive_bayes import MultinomialNB

In [21]: clf = MultinomialNB().fit(X_train_tfidf, y_train)
```

The last step is to determine the success of the created classifier. We will find this out with the help of a test set. This set was not included in training. Texts from this set must also be converted to TF-IDF vectors, fed to the input of the classifier, find out the prediction by the classifier and compare it with the actual value.



```
In [22]: y_pred = clf.predict(tfidf_vectorizer.transform(X_test))

In [23]: from sklearn import metrics

In [24]: print("Presnosť:",metrics.accuracy_score(y_test, y_pred))

         Presnosť: 0.8565279770444764
```

At the end of the chapter, we present the procedure for classifying an unknown message. Please note that, just like the training and test messages had to be transformed into vectors, the unknown message must first be converted into a TF-IDF vector before being classified.



```
In [25]: veta = "Obama was said to have stayed away because his own security needs
         print("Predikcia:",clf.predict(tfidf_vectorizer.transform([veta])))

         Predikcia: ['REAL']
```

PRISCILLA

priscilla.fitped.eu