Durham University

Department of Computer Science

Master of Data Science

COMP42415
Text Mining and Language Analytics
Workshops

Dr Stamos Katsigiannis

2023-24

Preface

Welcome to the workshop manual for the *COMP42415 Text Mining and Language Analytics* course of Durham University. This manual has been meticulously crafted to serve as your comprehensive guide to the fascinating world of Natural Language Processing (NLP). As language plays a pivotal role in our daily lives, understanding and harnessing the power of natural language is increasingly crucial in various fields, ranging from artificial intelligence and data science to linguistics and computational linguistics.

This workshop is designed to provide you with a hands-on and in-depth exploration of NLP, equipping you with the knowledge and skills necessary to navigate the intricacies of natural language and to build practical applications. Whether you are a seasoned professional seeking to enhance your expertise or a beginner eager to delve into the realm of NLP, this manual is tailored to meet your needs.

Key Features of this Workshop Manual:

1. Comprehensive Coverage: This manual covers a wide array of topics, from the foundational concepts of NLP to advanced techniques and applications. Each section is structured to build upon the previous one, ensuring a logical progression of learning.
2. Hands-On Exercises: Learning by doing is at the core of this workshop. Throughout the manual, you will find hands-on exercises that allow you to apply the concepts you learn in real-world scenarios. These exercises are designed to reinforce your understanding and enhance your practical skills.
3. Real-World Applications: NLP has diverse applications, and this manual provides insights into how NLP techniques are used in real-world scenarios.
4. Do-It-Yourself: This manual is designed to support self-learning, providing detailed examples and explanations for the presented NLP techniques.

By the end of this workshop, you will have gained a solid understanding of NLP concepts, developed practical skills in implementing NLP solutions, and be well-prepared to tackle challenges in the ever-evolving landscape of NLP.

We hope you find this workshop manual informative, engaging, and instrumental in your quest to master Natural Language Processing.

Best wishes for a rewarding learning experience!

Dr Stamos Katsigiannis

Required Python packages

Required python version: This workshop manual has been tested on Python 3.11.

The following Python packages are required:

- re
- nltk
- numpy
- scipy
- math
- scikit-learn
- matplotlib
- seaborn
- pandas
- tensorflow
- pickle

Contents

Workshop 1: Text query with Regular Expressions

1.1 Regular Expressions Definition

A regular expression (shortened as regex or regexp) is a sequence of characters that define a search pattern. Usually such patterns are used by string-searching algorithms for “find” or “find and replace” operations on strings, or for input validation.

1.2 The “re” Python package

Python has a built-in package called “re”, which can be used to work with Regular Expressions. Let’s import this package and use a regular expression to check whether the sentence “*I started studying this year at Durham University*” ends with the word “University” or with the word “school”.

1.2.1 Example

```
import re # Import the re package

txt = "I started studying this year at Durham University"

x1 = re.search("University$", txt) # Returns a Match object if there is a match anywhere in the
    string with the regex
x2 = re.search("school$", txt)

print("x1:",x1)
print("x2:",x2)

if(x1):
    print("The text ends with 'University'")
else:
    print("The text does not end with 'University'")

if(x2):
    print("The text ends with 'school'")
else:
    print("The text does not end with 'school'")
```

The output will look like:

```
x1: <_sre.SRE_Match object; span=(39, 49), match='University'>
x2: None
The text ends with 'University'
The text does not end with 'school'
```

1.2.2 RegEx functions in “re”

The re module offers a set of functions that allows us to search a string for a match:

Function	Description
findall(*args*)	Returns a list containing all matches
search(*args*)	Returns a Match object if there is a match anywhere in the string
match(*args*)	Returns a Match object if there is a match starting at the beginning of the string
split(*args*)	Returns a list where the string has been split at each match
sub(*args*)	Replaces one or many matches with a string

1.2.3 Regular expression metacharacters

As you can see in our first example, we used the character "$" in order to indicate that a text matching the regular expression should end with the string preceding the character "$". For example, the regular expression "car$" indicates that the text should and with the string "car". In this case, "$" is considered as a metacharacter, i.e. a character with a special meaning. Below are the metacharacters supported by the "re" package:

Character	Description	Example
[ ]	A set of characters	"[a-f]"
\	Signals a special sequence (can also be used to escape special characters)	"\s"
.	Any character (except newline character)	"uni..rsity"
ˆ	Starts with	"ˆShe"
$	Ends with	"John$"
*	Zero or more occurrences	"o*"
+	One or more occurrences	"l+"
?	Matches 0 or 1 repetitions of the preceding regex	ab? will match either "a" or "ab"
{}	Exactly the specified number of occurrences	"o{2}"
|	Either or	"he|she"
()	Capture and group	

The first metacharacters we'll look at are [and]. They're used for specifying a character class, which is a set of characters that you wish to match. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a "-". For example, [abc] will match any of the characters a, b, or c; this is the same as [a-c], which uses a range to express the same set of characters. If you wanted to match only lowercase letters, your regex would be [a-z].

Metacharacters are not active inside classes. For example, [akm$] will match any of the characters "a", "k", "m", or "$". "$" is usually a metacharacter, but inside a character class it is stripped of its special nature.

You can match the characters not listed within the class by complementing the set. This is indicated by including a "ˆ" as the first character of the class. For example, [ˆ5] will match any character except "5". If the caret appears elsewhere in a character class, it does not have special meaning. For example: [5ˆ] will match either a "5" or a "ˆ".

Perhaps the most important metacharacter is the backslash, \. As in Python string literals, the backslash can be followed by various characters to signal various special sequences. It is also used to escape all the metacharacters so you can still match them in patterns. For example, if you need to match a [or \, you can precede them with a backslash to remove their special meaning: \[or \\.

Some of the special sequences beginning with "\" represent predefined sets of characters that are often useful, such as the set of digits, the set of letters, or the set of anything that isn't a white space.

1.2.4 Regular expression special sequences

Let's see some of the main regular expression special sequences. For a more detailed list, please refer to `https://docs.python.org/3/library/re.html#re-syntax`.

These sequences can be included inside a character class. For example, [\s,.] is a character class that will match any white space character, or "," or ".".

Sequence	Description
\d	Matches any decimal digit. Equivalent to the class [0-9].
\D	Matches any non-digit character. Equivalent to the class [ˆ0-9].
\s	Matches any white space character. Equivalent to the class [\t\n\r\f\v].
\S	Matches any non-white space character. Equivalent to the class [ˆ\t\n\r\f\v].
\w	Matches any alphanumeric character. Equivalent to the class [a-zA-Z0-9_].
\W	Matches any non-alphanumeric character. Equivalent to the class [ˆa-zA-Z0-9_].

1.2.5 Repeating regular expressions

Being able to match varying sets of characters is the first thing regular expressions can do that isn't already possible with the methods available on strings. However, if that was the only additional capability of regexes, they wouldn't be much of an advance. Another capability is that you can specify that portions of the regular expression must be repeated a certain number of times.

The first metacharacter for repeating things that we'll look at is "*". "*" does not match the literal character "*", but it specifies that the previous character can be matched zero or more times, instead of exactly once.

Example: "do*g" will match "dg" (zero "o" characters), "dog" (one "o" character), "doooog" (four "o" characters), and so on.

Repetitions such as "*","+", and "?" are greedy. When repeating a regular expression, the matching engine will try to repeat it as many times as possible. If later portions of the pattern don't match, the matching engine will then back up and try again with fewer repetitions. If this behaviour is undesirable, you can add "?" after the qualifier ("*?","+?", "??") to make it perform the match in non-greedy or minimal fashion, i.e. as few characters as possible will be matched.

Step-by-step example
Let's consider the expression a[bcd]*b. This matches the letter "a", zero or more letters from the class [bcd], and finally ends with a "b". Now imagine matching this regular expression against the string "abcbd".

Step	Matched	Explanation
1	a	The "a" in the regex matches.
2	abcbd	The engine matches "[bcd]*", going as far as it can, which is to the end of the string.
3	**FAILED**	The engine tries to match "b", but the current position is at the end of the string, so it fails.
4	abcb	Back up, so that "[bcd]*" matches one less character.
5	**FAILED**	Try "b" again, but the current position is at the last character, which is a "d".
6	abc	Back up again, so that "[bcd]*" is only matching "bc".
7	abcb	Try "b" again. This time the character at the current position is "b", so it succeeds.

The end of the regular expression has now been reached, and it has matched "abcb". This demonstrates how the matching engine goes as far as it can at first, and if no match is found it will then progressively back up and retry the rest of the regular expression again and again. It will back up until it has tried zero matches for "[bcd]*", and if that subsequently fails, the engine will conclude that the string does not match the regex at all.

Another repeating metacharacter is "+", which matches one or more times. Pay careful attention to the difference between "*" and "+". "*" matches zero or more times, so whatever's being repeated may not be present at all, while "+" requires at least one occurrence.

Example: "do+g" will match "dog" (one "o" character), "dooog" (three "o" characters), and so on, but will not match "dg" (zero "o" characters).

There are two more repeating qualifiers. The question mark character "?" matches either once or zero times. Think of it as marking something as being optional.

Example: "pre-?processing" matches either "preprocessing" or "pre-processing".

The most complicated repeated qualifier is "{m,n}", where m and n are decimal integers. This qualifier means there must be at least m repetitions, and at most n. For example, "a/{1,3}b" will match "a/b", "a//b", and "a///b", but it will not match "ab", which has no slashes, or "a////b", which has four slashes. You can omit either m or n. In this case, default values for m or n are used. Omitting m is interpreted as a lower limit of 0, while omitting n results in an upper bound of infinity.

Note: Some qualifiers are interchangeable. For example "{0,}" is the same as "*", "{1,}" is the same as "+", and "{0,1}" is the same as "?". "*", "+", and "?" make the regular expression easier to read, so try to use them if possible.

1.3 Using Regular Expressions to match strings

1.3.1 Matching example

Let's use the text "*I started studying this year at Durham University*" again and find out whether the string "at" or the string "in" exists in the text.

```
txt = "I started studying this year at Durham University"

x = re.search("at|in", txt)
print(x.string) # Returns the string passed into the function
print(x.span()) # Returns a tuple containing the start, and end positions of the match
print(x.group()) # Returns the part of the string where there was a match

print(txt[x.span()[0]:x.span()[1]]) # Print the content of the string at the positions of the match
```

The output will look like:

```
I started studying this year at Durham University
(15, 17)
in
in
```

As you can see, there was a match to our regex at the character with index 15 (counting starts from 0), ending at the character with index 17. Indeed, the string "in" was found within the word "studying".

However, if you read the input text, there should have been a second match for the word "at" but only the first match was returned. Note that if there is more than one match, only the first occurrence of the match will be returned by the search() function! We can use the findall() function to get a list of all matches in the order they are found.

```
x = re.findall("at|in", txt)
for match in x:
    print(match)
```

The output will look like:

```
in
at
```

As expected, the findall() function returned two matches, "in" and "at".

Consider the string "*stp stop stoop stoooooop stoooooooooooop*". Let's find matches where the character "o" appears one or more times and matches where the character "o" appears two times only.

```
x = re.findall("o+", "stp stop stoop stoooooop stoooooooooooop")
print("One or more occurrences of 'o':")
for match in x:
    print(match)

x = re.findall("o{2}", "stp stop stoop stoooooop stoooooooooooop")
print("\nTwo occurrences of 'o':")
for match in x:
    print(match)
```

The output will look like:

```
One or more occurrences of 'o':
o
oo
oooooo
oooooooooo

Two occurrences of 'o':
oo
oo
oo
oo
oo
oo
oo
oo
oo
```

Note that when looking for two occurrences of "o", the findall() function returned 9 matches that correspond to the 9 non-overlapping matches from the start (left) to the end (right) of the input text: "*stp stop st(oo)p st(oo)(oo)(oo)p st(oo)(oo)(oo)(oo)(oo)op*"

1.3.2 Matching to validate strings

Let's use regular expressions to check the validity of various strings. Consider an identification number that should consist of exactly 10 digits (from 0 to 9), and the strings: "0123456789", "12345", "0000a00005", "+000001111", "00000011115", "2030405060". How can we check whether these strings are valid identification numbers?

```
text = list()
text.append("0123456789")
text.append("12345")
text.append("0000a00005")
text.append("+000001111")
text.append("00000011115")
text.append("2030405060")

regex = "[0-9]{10}"

result = list()
for t in text:
    x = re.match(regex, t)
    if(x != None):
        print(t,"->",x.group())
    else:
        print(t,"-> No match")
    result.append(x)

for i in range(len(text)):
    if(result[i]!=None and result[i].group()==text[i]):
        print(text[i],"is a valid identification number")
    else:
        print(text[i],"is NOT a valid identification number")
```

The output will look like:

```
0123456789 -> 0123456789
12345 -> No match
0000a00005 -> No match
+000001111 -> No match
00000011115 -> 0000001111
2030405060 -> 2030405060
0123456789 is a valid identification number
12345 is NOT a valid identification number
0000a00005 is NOT a valid identification number
```

```
+000001111 is NOT a valid identification number
00000011115 is NOT a valid identification number
2030405060 is a valid identification number
```

Notice that string "00000011115" consists of 11 numerical digits, thus the regular expression matches the subset "0000001111". However, this is not a valid identification number according to the specification above. When validating input, remember to check whether the matched string is equal to the query string.

1.3.3 Credit card number validation

Let's try to validate whether the following strings are valid VISA or Mastercard credit card numbers: "1000000000000", "4000000000000", "5000000000000", "5000000000000000", "50000a0000000c000", "40123456789". VISA credit card numbers should start with a 4 and have 13 or 16 digits. Mastercard credit card numbers start with a 5 and have 16 digits.

```python
text = list()
text.append("1000000000000") # 13 digits - Not valid
text.append("4000000000000") # 13 digits - Valid VISA
text.append("5000000000000") # 13 digits - Not valid
text.append("5000000000000000") # 16 digits - Valid Mastercard
text.append("50000a0000000c000") # Not valid, contains letters
text.append("40123456789") # 11 digits - Not valid

regex = "(5[0-9]{15})|(4([0-9]{12}|[0-9]{15}))" # Number 5 followed by 15 digits OR number 4 followed
    by either 12 or 15 digits

result = list()
for t in text:
    x = re.match(regex, t)
    if(x != None):
        print(t,"->",x.group())
    else:
        print(t,"-> No match")
    result.append(x)

print("")
for i in range(len(text)):
    if(result[i]!=None and result[i].group()==text[i]):
        print(text[i],"is a valid VISA or Mastercard number")
    else:
        print(text[i],"is NOT a valid VISA or Mastercard number")
```

The output will look like:

```
1000000000000 -> No match
4000000000000 -> 4000000000000
5000000000000 -> No match
5000000000000000 -> 5000000000000000
50000a0000000c000 -> No match
40123456789 -> No match

1000000000000 is NOT a valid VISA or Mastercard number
4000000000000 is a valid VISA or Mastercard number
5000000000000 is NOT a valid VISA or Mastercard number
5000000000000000 is a valid VISA or Mastercard number
50000a0000000c000 is NOT a valid VISA or Mastercard number
40123456789 is NOT a valid VISA or Mastercard number
```

Let's analyse the regex that we used. Both VISA and Mastercard numbers start with a specific digit but Mastercard has exactly 16 digits in total, while VISA can have either 13 or 16 digits. Let's first create a regex for each case separately. For Mastercard, it should be "5[0-9]{15}", the digit 5 followed by exactly 15 digits (0-9), for a total of 16 digits. For VISA, it should be "4([0-9]{12}|[0-9]{15})", the digit 4 followed by either exactly 12 digits, for a total of 13 digits, or exactly 15 digits, for a total of 16 digits. Then, to include both the VISA and the Mastercard cases in our final regex, we can enclose each regex within parentheses and combine them with the OR ("|") operator.

Note that the expression [0-9] could be switched to [\d]:

```
regex = "(5[\d]{15})|(4([\d]{12}|[\d]{15}))" # Number 5 followed by 15 digits OR number 4 followed by
    either 12 or 15 digits

result = list()
for t in text:
    x = re.match(regex, t)
    if(x != None):
        print(t,"->",x.group())
    else:
        print(t,"-> No match")
    result.append(x)

print("")
for i in range(len(text)):
    if(result[i]!=None and result[i].group()==text[i]):
        print(text[i],"is a valid VISA or Mastercard number")
    else:
        print(text[i],"is NOT a valid VISA or Mastercard number")
```

The output will look like:

```
1000000000000 -> No match
4000000000000 -> 4000000000000
5000000000000 -> No match
5000000000000000 -> 5000000000000000
50000a0000000c000 -> No match
40123456789 -> No match

1000000000000 is NOT a valid VISA or Mastercard number
4000000000000 is a valid VISA or Mastercard number
5000000000000 is NOT a valid VISA or Mastercard number
5000000000000000 is a valid VISA or Mastercard number
50000a0000000c000 is NOT a valid VISA or Mastercard number
40123456789 is NOT a valid VISA or Mastercard number
```

1.3.4 Validation of United Kingdom's National Insurance numbers (NINO)

According to the rules for validating UK national insurance numbers[1], a NINO is made up of 2 letters, 6 numbers and a final letter, which is always A, B, C, or D. It looks something like this: QQ 12 34 56 A. The characters D, F, I, Q, U, and V are not used as either the first or second letter of a NINO prefix. The letter O is not used as the second letter of a prefix.

Let's create the required regex step-by-step and validate the following strings: "AA 123456 B", "AO 123456 B", "AQ123456 B", "QQ 123456 B", "AA 123456 X", "AA 12 34 56 B", "AA123456B", "AA12345B", "A 123456 B", "A 123456 B". We must also take white spaces into consideration. Let's consider the following two ways of writing a NINO: AA123456A and AA 12345 A.

1. The first letter should be any of A, B, C, E, G, H, J, K, L, M, N, O, P, R, S, T, W, X, Y: **(A|B|C|E|G|H|J|K|L|M|N|O|P|R|S|T|W|X|Y)**
2. The second letter should be any of A, B, C, E, G, H, J, K, L, M, N, P, R, S, T, W, X, Y: (A|B|C|E|G|H|J|K|L|M|N|O|P|R|S|T|W|X|Y)**(A|B|C|E|G|H|J|K|L|M|N|P|R|S|T|W|X|Y)**
3. The third letter can optionally be a white space character: (A|B|C|E|G|H|J|K|L|M|N|O|P|R|S|T|W|X|Y)(A|B|C|E|G|H|J|K|L|M|N|P|R|S|T|W|X|Y)**[\s]?**
4. Then, exactly 6 digits (0-9) are required: (A|B|C|E|G|H|J|K|L|M|N|O|P|R|S|T|W|X|Y)(A|B|C|E|G|H|J|K|L|M|N|P|R|S|T|W|X|Y)[\s]?**[0-9]{6}**
5. The next letter can optionally be a white space character: (A|B|C|E|G|H|J|K|L|M|N|O|P|R|S|T|W|X|Y)(A|B|C|E|G|H|J|K|L|M|N|P|R|S|T|W|X|Y)[\s]?[0-9]{6}**[\s]?**

[1] https://www.gov.uk/hmrc-internal-manuals/national-insurance-manual/nim39110

6. The final letter must be one of A, B, C, or D:
   (A|B|C|E|G|H|J|K|L|M|N|O|P|R|S|T|W|X|Y)(A|B|C|E|G|H|J|K|L|M|N|P|R|S|T|W|X|Y)[\s]?[0-9]{6}[\s]?**(A|B|C|D)**

```
text = list()
text.append("AA 123456 B") # Valid
text.append("AO 123456 B") # Not valid
text.append("AQ123456 B") # Not valid
text.append("QQ 123456 B") # Not valid
text.append("AA 123456 X") # Not valid
text.append("AA 12 34 56 B") # Not valid
text.append("AA123456B") # Valid
text.append("AA12345B") # Not valid
text.append("A 123456 B") # Not valid
text.append("A  123456 B") # Not valid

regex =
    "(A|B|C|E|G|H|J|K|L|M|N|O|P|R|S|T|W|X|Y)(A|B|C|E|G|H|J|K|L|M|N|P|R|S|T|W|X|Y)[\s]?[0-9]{6}[\s]?(A|B|C|D)"

result = list()
for t in text:
    x = re.match(regex, t)
    if(x != None):
        print(t,"->",x.group())
    else:
        print(t,"-> No match")
    result.append(x)

print("")
for i in range(len(text)):
    if(result[i]!=None and result[i].group()==text[i]):
        print("VALID\t",text[i])
    else:
        print("-----\t",text[i])
```

The output will look like:

```
AA 123456 B -> AA 123456 B
AO 123456 B -> No match
AQ123456 B -> No match
QQ 123456 B -> No match
AA 123456 X -> No match
AA 12 34 56 B -> No match
AA123456B -> AA123456B
AA12345B -> No match
A 123456 B -> No match
A  123456 B -> No match

VALID  AA 123456 B
-----  AO 123456 B
-----  AQ123456 B
-----  QQ 123456 B
-----  AA 123456 X
-----  AA 12 34 56 B
VALID  AA123456B
-----  AA12345B
-----  A 123456 B
-----  A  123456 B
```

1.3.5 Validation of hexadecimal numbers

Let's use regular expressions to check whether a string corresponds to a hexadecimal number. Consider the strings "xAF1400BD", "1299ab32", "xFF00FF5R", "0xaa00bb". How can we check if these strings are representations of hexadecimal numbers? Remember that valid hexadecimal digits are [0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F, a,b,c,d,e,f] and that in computers, hexadecimal numbers may be denoted with an "x" or "0x" (either lowercase or uppercase) in the beginning. For example, the hexadecimal number FFF, can also be written as fff, 0xfff, 0XFFF, xfff, XFFF and can also have mixed lowercase and uppercase letters.

```
text = list()
text.append("xAF1400BD") # Valid
text.append("1299ab32") # Valid
text.append("xFF00FF5R") # Not valid - Character R is not a valid headecimal digit
text.append("0xaa00bb") # Valid
text.append("xaa00bb4657AB000922334bce111A") # Valid
text.append("0xfff") # Valid
text.append("xFfF") # Valid
text.append("AAA") # Valid
text.append("ALA") # Not valid - Character L is not a valid headecimal digit

regex = "(0x|0X|x|X)?[0-9a-fA-F]+" # One optional occurence of 0x, 0X, x, or X, followed by at least
    one digit from 0 to 9, or lowercase letter from a to f, or uppercase letter from A to F

result = list()
for t in text:
    x = re.match(regex, t)
    if(x != None):
        print(t,"->",x.group())
    else:
        print(t,"-> No match")
    result.append(x)

print("")
for i in range(len(text)):
    if(result[i]!=None and result[i].group()==text[i]):
        print("VALID HEX\t",text[i])
    else:
        print("---------\t",text[i])
```

The output will look like:

```
xAF1400BD -> xAF1400BD
1299ab32 -> 1299ab32
xFF00FF5R -> xFF00FF5
0xaa00bb -> 0xaa00bb
xaa00bb4657AB000922334bce111A -> xaa00bb4657AB000922334bce111A
0xfff -> 0xfff
xFfF -> xFfF
AAA -> AAA
ALA -> A

VALID HEX   xAF1400BD
VALID HEX   1299ab32
---------   xFF00FF5R
VALID HEX   0xaa00bb
VALID HEX   xaa00bb4657AB000922334bce111A
VALID HEX   0xfff
VALID HEX   xFfF
VALID HEX   AAA
---------   ALA
```

Note that in the case of the "ALA" string, the regex found the match "A", which is a valid hexadecimal number, but the full string "ALA" is not a valid hexadecimal number. When validating input, remember to check that the matched string is equal to the query string.

1.4 Using Regular Expressions to search elements in files

1.4.1 Parsing XML files

Let's use regular expressions to parse an XML file. We will use the movies.xml file which contains the titles and release dates of 5 movies. We will use a regular expression to retrieve all the movie titles from the file. First, copy the file ”movies.xml” to your current working directory or retrieve the absolute path of the file. Then open the file, load its contents into a variable and close the file.

```
f = open("movies.xml", "r") # Opens the file for reading only ("r")
text = f.read() # Store the contents of the file in variable "text". read() returns all the contents
    of the file
f.close() # Close the file
print(text) # Print the contents of variable "text"
```

The output will look like:

```
<movies>
  <movie id="1">
    <title>And Now for Something Completely Different</title>
    <released>1971</released>
  </movie>
  <movie id="2">
    <title>Monty Python and the Holy Grail</title>
    <released>1974</released>
  </movie>
  <movie id="3">
    <title>Monty Python's Life of Brian</title>
    <released>1979</released>
  </movie>
  <movie id="4">
    <title>Monty Python Live at the Hollywood Bowl</title>
    <released>1982</released>
  </movie>
  <movie id="5">
    <title>Monty Python's The Meaning of Life</title>
    <released>1983</released>
  </movie>
</movies>
```

As you can see above, all movie titles in the movies.xml XML file are enclosed within the <title> and </title> tags. Let's use a regular expression to match all the strings that are enclosed within these tags.

```
regex = "<title>.*</title>" # <title>, followed by any number of characters except for the newline
    character, followed by </title>

x = re.findall(regex, text) # Find all matches of the regex

print(x,"\n")

print("Movie titles from movies.xml:")
for title in x:
    title = title.replace("<title>","") # Remove <title> by replacing it with empty string
    title = title.replace("</title>","") # Remove </title> by replacing it with empty string
    print(title)
```

The output will look like:

```
['<title>And Now for Something Completely Different</title>', '<title>Monty Python and the Holy
    Grail</title>', "<title>Monty Python's Life of Brian</title>", '<title>Monty Python Live at the
    Hollywood Bowl</title>', "<title>Monty Python's The Meaning of Life</title>"]

Movie titles from movies.xml:
And Now for Something Completely Different
Monty Python and the Holy Grail
Monty Python's Life of Brian
Monty Python Live at the Hollywood Bowl
Monty Python's The Meaning of Life
```

1.4.2 Parsing HTML files

Let's read the links.html HTML file and use regular expressions to find all links within the file. Remember that in HTML, links are denoted using the <a> and tags and the link url is provided using the "href" attribute within the <a> tag. For example a link for the main website of Durham University would be:

```
<a href="https://www.dur.ac.uk/">Durham University</a>
```

```
f = open("links.html", "r") # Opens the file for reading only ("r")
text = f.read() # Store the contents of the file in variable "text". read() returns all the contents
    of the file
f.close() # Close the file
print(text,"\n") # Print the contents of variable "text"

regex = '<a href=".*"'

x = re.findall(regex, text) # Find all matches of the regex

print(x,"\n")

print("Links from links.xml:")
for link in x:
   link = link.replace('<a href="',"") # Remove <a href=" by replacing it with empty string
   link = link.replace('"',"") # Remove " by replacing it with empty string
   print(link)
```

The output will look like:

```
<!doctype html>
<html lang="en-GB">

<head>
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <title>Test page for Text Mining and Language Analytics</title>
</head>

<body>
   <h1>Test page for Text Mining and Language Analytics</h1>
   <a href="https://www.durham.ac.uk/departments/academic/computer-science/">Link 1</a><br/>
   <a href="https://www.gov.uk/government/organisations/hm-revenue-customs">Link 2</a><br/>
   <a href="https://www.gov.uk/">Link 3</a><br/>
   <a href="https://www.dur.ac.uk/">Link 4</a><br/>
   <a href="https://www.nhs.uk/">Link 5</a><br/>
</body>

</html>

['<a href="https://www.durham.ac.uk/departments/academic/computer-science/"', '<a
    href="https://www.gov.uk/government/organisations/hm-revenue-customs"', '<a
    href="https://www.gov.uk/"', '<a href="https://www.dur.ac.uk/"', '<a href="https://www.nhs.uk/"']

Links from links.xml:
https://www.durham.ac.uk/departments/academic/computer-science/
https://www.gov.uk/government/organisations/hm-revenue-customs
https://www.gov.uk/
https://www.dur.ac.uk/
https://www.nhs.uk/
```

Note than we used single quotes to denote strings that included a double quote character.

As you can see above, we successfully retrieved the urls from links.html. Nevertheless, please note that using regular expressions is not the best approach for parsing HTML files due to the flexibility of HTML syntax. Solutions like XPath (`https://developer.mozilla.org/en-US/docs/Web/API/XPathExpression`) are more suitable for HTML parsing.

1.4.3 Parsing raw text

The file emails.txt contains a list of emails from various domains. Let's use regular expressions to find the emails from the durham.ac.uk domain.

```
f = open("emails.txt", "r") # Opens the file for reading only ("r")
text = f.read() # Store the contents of the file in variable "text". read() returns all the contents
    of the file
f.close() # Close the file
print(text,"\n") # Print the contents of variable "text"

regex = "[0-9a-zA-Z!#$%&'*+-/=?^_`{|}~.]+@durham.ac.uk"

x = re.findall(regex, text) # Find all matches of the regex

print(x,"\n")

print("Emails from durham.ac.uk:")
for email in x:
   print(email)
```

The output will look like:

```
john+acme.co@hotmail.com
bob@gmail.com
tom@durham.ac.uk
jerry@durham.ac.uk
scrooge@durham.ac.uk
donald@yahoo.co.uk
huey@yahoo.co.uk
dewey@gmail.com
louie.duck@durham.ac.uk
gyro.gearloose@yahoo.co.uk
bart@yahoo.co.uk
homer@gmail.com
stan@hotmail.com
kyle-broflovski@durham.ac.uk
eric@yahoo.co.uk
kenny@gmail.com
butters@durham.ac.uk
wendy@hotmail.com
randy_marsh@durham.ac.uk
chef@gmail.com

['tom@durham.ac.uk', 'jerry@durham.ac.uk', 'scrooge@durham.ac.uk', 'louie.duck@durham.ac.uk',
    'kyle-broflovski@durham.ac.uk', 'butters@durham.ac.uk', 'randy_marsh@durham.ac.uk']

Emails from durham.ac.uk:
tom@durham.ac.uk
jerry@durham.ac.uk
scrooge@durham.ac.uk
louie.duck@durham.ac.uk
kyle-broflovski@durham.ac.uk
butters@durham.ac.uk
randy_marsh@durham.ac.uk
```

As you can see above, we retrieved all emails from the durham.ac.uk email.

1.5 Using Regular Expressions to substitute strings

1.5.1 Substitution example

Let's now convert any string that matches the "at|in" regex in the text "I started studying this year at Durham University" with the string "FOO". To achieve this, we are going to use the sub() function.

```
txt = "I started studying this year at Durham University"
x = re.sub("at|in","FOO", txt)
print(x)
```

The output will look like:

```
I started studyFOOg this year FOO Durham University
```

As expected, two matches of the regex were converted to “FOO”. What if we wanted only the first match to be substituted with “FOO”? We can add an additional argument in the sub() function indicating the number of substitutions we would like to make.

```
x = re.sub("at|in","FOO", txt,1)
print(x)
```

The output will look like:

```
I started studyFOOg this year at Durham University
```

1.5.2 Email domain substitution

Let’s load again emails.txt and change the domain to “new.ac.uk” for all emails in the “durham.ac.uk”, “gmail.com”, and “yahoo.co.uk” domains.

```
f = open("emails.txt", "r") # Opens the file for reading only ("r")
text = f.read() # Store the contents of the file in variable "text". read() returns all the contents
    of the file
f.close() # Close the file
print("OLD EMAILS:")
print(text,"\n") # Print the contents of variable "text"

regex = "@durham.ac.uk|@gmail.com|@yahoo.co.uk"

print("NEW EMAILS:")
x = re.sub(regex,"@new.ac.uk", text)

print(x)
```

The output will look like:

```
OLD EMAILS:
john+acme.co@hotmail.com
bob@gmail.com
tom@durham.ac.uk
jerry@durham.ac.uk
scrooge@durham.ac.uk
donald@yahoo.co.uk
huey@yahoo.co.uk
dewey@gmail.com
louie.duck@durham.ac.uk
gyro.gearloose@yahoo.co.uk
bart@yahoo.co.uk
homer@gmail.com
stan@hotmail.com
kyle-broflovski@durham.ac.uk
eric@yahoo.co.uk
kenny@gmail.com
butters@durham.ac.uk
wendy@hotmail.com
randy_marsh@durham.ac.uk
chef@gmail.com

NEW EMAILS:
john+acme.co@hotmail.com
```

```
bob@new.ac.uk
tom@new.ac.uk
jerry@new.ac.uk
scrooge@new.ac.uk
donald@new.ac.uk
huey@new.ac.uk
dewey@new.ac.uk
louie.duck@new.ac.uk
gyro.gearloose@new.ac.uk
bart@new.ac.uk
homer@new.ac.uk
stan@hotmail.com
kyle-broflovski@new.ac.uk
eric@new.ac.uk
kenny@new.ac.uk
butters@new.ac.uk
wendy@hotmail.com
randy_marsh@new.ac.uk
chef@new.ac.uk
```

1.6 Exercises

Exercise 1.1 Validate the following identification numbers: "500011110000" (valid), "5000 1111 0000" (valid), "500001110000" (valid), "5000 0111 0000" (valid), "500021110000" (not valid), "5000 2111 0000" (not valid), "400001110000" (not valid), "4000 0111 0000" (not valid), "500011110009" (not valid), "5000 1111 0009" (not valid). A valid number should be 12 digits long, the first digit should always be 5, the 5th digit should be 0 or 1, and the last digit cannot be 8 or 9. The numbers can also be grouped in groups of four digits with a white space between groups.

Exercise 1.2 Redo the activity from Section 1.3.4 in order to also support the following writing of NINO strings: "AA 12 34 56 A"

Exercise 1.3 Use regular expressions to print a list of all the album titles, a list of all the artists, and the average price of all CDs in the cds.xml file.

Exercise 1.4 Create a regular expression that matches all the strings in the first column but none of those in the second column

affgfking	fgok
rafgkahe	a fgk
bafghk	affgm
baffgkit	afffhk
affgfking	fgok
rafgkahe	afg.K
bafghk	aff gm
baffgkit	afffhgk

Exercise 1.5 Use a regular expression to substitute the quantities of the bought items with "XX" in the following sentence: "*Yesterday we bought 120 packs of A4 paper, 5 bottles of ink, 10 boxes of paperclips, 200 notebooks, and 5.35 litres of fuel. The total cost for order #1290 was £1000.43.*". Note that the order number and cost must not be substituted. The resulting sentence must be: "*Yesterday we bought XX packs of A4 paper, XX bottles of ink, XX boxes of paperclips, XX notebooks, and XX litres of fuel. The total cost for order #1290 was £1000.43.*"

Workshop 2: Text pre-processing

2.1 NLTK corpora

The NLTK python package comes pre-loaded with a lot of corpora that can be used to experiment with text pre-processing.

2.1.1 Import NLTK

Let's load one of the available corpora in NLTK, called movie_reviews. The movie_reviews corpus contains documents consisting of reviews about movies and annotated in relation to their sentiment.

```
import nltk # Import the NLTK library
nltk.download('movie_reviews') # Download movie_reviews from NLTK
from nltk.corpus import movie_reviews # Import the movie_reviews corpus from NLTK
```

2.1.2 Corpus file IDs

Let's check the file IDs in the movie_reviews corpus:

```
movie_reviews.fileids() # List file-ids in the corpus
```

The output will look like:

```
['neg/cv000_29416.txt', 'neg/cv001_19502.txt', 'neg/cv002_17424.txt', 'neg/cv003_12683.txt',
    'neg/cv004_12641.txt', ...]
```

2.1.3 Corpus file categories

Let's examine what classes (categories) are included in the corpus:

```
movie_reviews.categories() # List categories in the corpus
```

The output will look like:

```
['neg', 'pos']
```

This means that the movie_reviews corpus contains documents that have been annotated as belonging to two distinct classes, more specifically the Negative class (neg) and the Positive class (pos), in relation to the sentiment of the respective movie review.

2.1.4 Corpus words

Let's see the list of words in the movie_review corpus and print their number:

```
print(movie_reviews.words())
length = len(movie_reviews.words())
print("Number of words in corpus: ", length)
```

The output will look like:

```
['plot', ':', 'two', 'teen', 'couples', 'go', 'to', ...]
Number of words in corpus: 1583820
```

2.1.5 Selecting files from specific category

Use the following in order to view the file IDs for files belonging to a specific category. Note that you can provide a list with categories, e.g ['neg','pos','other']

```
movie_reviews.fileids(['neg']) # List file ids with 'neg' category
movie_reviews.fileids(['pos']) # List file ids with 'pos' category
```

The output will look like:

```
['neg/cv000_29416.txt', 'neg/cv001_19502.txt', 'neg/cv002_17424.txt', 'neg/cv003_12683.txt',
'neg/cv004_12641.txt', 'neg/cv005_29357.txt', 'neg/cv006_17022.txt', 'neg/cv007_4992.txt', ...]
['pos/cv000_29590.txt', 'pos/cv001_18431.txt', 'pos/cv002_15918.txt', 'pos/cv003_11664.txt',
'pos/cv004_11636.txt', 'pos/cv005_29443.txt', 'pos/cv006_15448.txt', 'pos/cv007_4968.txt', ...]
```

2.1.6 Corpus sentences

To access the tokenised sentences included in the corpus:

```
nltk.download('punkt') # Download the Punkt sentence tokenizer from NLTK

movie_reviews.sents()
```

The output will look like:

```
[['plot', ':', 'two', 'teen', 'couples', 'go', 'to', 'a', 'church', 'party', ',', 'drink', 'and',
    'then', 'drive', '.'], ['they', 'get', 'into', 'an', 'accident', '.'], ...]
```

2.1.7 Accessing sentences from specific file

We can use the "fileids" argument in order to access the sentences from a specific file in the corpus:

```
movie_reviews.sents(fileids='pos/cv004_11636.txt')
```

The output will look like:

```
[['moviemaking', 'is', 'a', 'lot', 'like', 'being', 'the', 'general', 'manager', 'of', 'an', 'nfl',
    'team', 'in', 'the', 'post', '-', 'salary', 'cap', 'era', '--', 'you', "'", 've', 'got', 'to',
    'know', 'how', 'to', 'allocate', 'your', 'resources', '.'], ['every', 'dollar', 'spent', 'on',
    'a', 'free', '-', 'agent', 'defensive', 'tackle', 'is', 'one', 'less', 'dollar', 'than', 'you',
    'can', 'spend', 'on', 'linebackers', 'or', 'safeties', 'or', 'centers', '.'], ...]
```

2.1.8 Number of documents in each category

Let's see how many documents (files) does the movie_reviews corpus contain, and then how many documents of "neg" category and how many of "pos" category it contains:

```
documents = len(movie_reviews.fileids())
documents_neg = len(movie_reviews.fileids(['neg']))
documents_pos = len(movie_reviews.fileids(['pos']))
print("Number of documents: ", documents)
print("Number of documents in neg category: ", documents_neg)
print("Number of documents in pos category: ", documents_pos)
```

The output will look like:

```
Number of documents: 2000
```

```
Number of documents in neg category: 1000
Number of documents in pos category: 1000
```

As you can see here, the corpus is perfectly balanced between negative and positive sentiment reviews, having 1000 documents in each category.

2.1.9 Corpus raw text

To access the raw text of a specific file, use the ".raw()" function:

```
rawtext = movie_reviews.raw('neg/cv002_17424.txt').strip()[:500] # strip() removes blank spaces in
    the beginning and end of a string. [:500] is used to only retrieve the first 500 characters of
    the string
print(rawtext)
```

The output will look like:

```
it is movies like these that make a jaded movie viewer thankful for the invention of the timex
    indiglo watch .
based on the late 1960's television show by the same name , the mod squad tells the tale of three
    reformed criminals under the employ of the police to go undercover .
however , things go wrong as evidence gets stolen and they are immediately under suspicion .
of course , the ads make it seem like so much more .
quick cuts , cool music , claire dane's nice hair and cute outfits , car
```

2.2 Input text from text file

NLTK corpora are very useful for testing NLP and text processing algorithms. But what if we wanted to load text from a text file? Let's try to load the text from the "alice.txt" file. First, copy the file "alice.txt" to your current working directory or retrieve the absolute path of the file. Then open the file, load its contents into a variable and close the file.

```
f = open("alice.txt", "r") # Opens the file for reading only ("r")
text = f.read() # Store the contents of the file in variable "text". read() returns all the contents
    of the file
f.close() # Close the file
print(text) # Print the contents of variable "text"
```

The output will look like:

```
Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to
    do: once or twice she had peeped into the book her sister was reading, but it had no pictures or
    conversations in it, "and what is the use of a book," thought Alice "without pictures or
    conversations?"

So she was considering in her own mind (as well as she could, for the hot day made her feel very
    sleepy and stupid), whether the pleasure of making a daisy-chain would be worth the trouble of
    getting up and picking the daisies, when suddenly a White Rabbit with pink eyes ran close by her.

There was nothing so very remarkable in that; nor did Alice think it so very much out of the way to
    hear the Rabbit say to itself, "Oh dear! Oh dear! I shall be late!" (when she thought it over
    afterwards, it occurred to her that she ought to have wondered at this, but at the time it all
    seemed quite natural); but when the Rabbit actually took a watch out of its waistcoat-pocket, and
    looked at it, and then hurried on, Alice started to her feet, for it flashed across her mind that
    she had never before seen a rabbit with either a waistcoat-pocket, or a watch to take out of it,
    and burning with curiosity, she ran across the field after it, and fortunately was just in time
    to see it pop down a large rabbit-hole under the hedge.
```

2.3 Text pre-processing

In the text output above, you can see that the text contains various sentences, a mix of lowercase and uppercase letters, there are some parentheses, as well as some punctuation marks.

2.3.1 Sentence Tokenisation

Let's tokenise the text, i.e. chop the text into pieces. In this case, the text is in the English language and punctuation marks are used to separate sentences from each other and a blank space is used to separate words from each other. For splitting a string into sentences, NLTK has the sent_tokenize() default tokeniser function.

Let's divide the text from alice.txt into sentences:

```
from nltk import sent_tokenize # Import the sent_tokenize function from NLTK
sent_tokenize(text) # Tokenise "text" into sentences and print the output
```

The output will look like:

```
['Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing
    to do: once or twice she had peeped into the book her sister was reading, but it had no pictures
    or conversations in it, "and what is the use of a book," thought Alice "without pictures or
    conversations?"', 'So she was considering in her own mind (as well as she could, for the hot day
    made her feel very sleepy and stupid), whether the pleasure of making a daisy-chain would be
    worth the trouble of getting up and picking the daisies, when suddenly a White Rabbit with pink
    eyes ran close by her.', 'There was nothing so very remarkable in that; nor did Alice think it so
    very much out of the way to hear the Rabbit say to itself, "Oh dear!', 'Oh dear!', 'I shall be
    late!"', '(when she thought it over afterwards, it occurred to her that she ought to have
    wondered at this, but at the time it all seemed quite natural); but when the Rabbit actually took
    a watch out of its waistcoat-pocket, and looked at it, and then hurried on, Alice started to her
    feet, for it flashed across her mind that she had never before seen a rabbit with either a
    waistcoat-pocket, or a watch to take out of it, and burning with curiosity, she ran across the
    field after it, and fortunately was just in time to see it pop down a large rabbit-hole under the
    hedge.']
```

2.3.2 Word Tokenisation

Now let's divide the text from alice.txt into words using the word_tokenize() NLTK function:

```
from nltk import word_tokenize # Import the word_tokenize function from NLTK
word_tokenize(text) # Tokenise "text" into words and print the output
```

The output will look like:

```
['Alice', 'was', 'beginning', 'to', 'get', 'very', 'tired', 'of', 'sitting', 'by', 'her', 'sister',
    'on', 'the', 'bank', ',', 'and', 'of', 'having', 'nothing', 'to', 'do', ':', 'once', 'or',
    'twice', 'she', 'had', 'peeped', 'into', 'the', 'book', 'her', 'sister', 'was', 'reading', ',',
    'but', 'it', 'had', 'no', 'pictures', 'or', 'conversations', 'in', 'it', ',', '``', 'and',
    'what', 'is', 'the', 'use', 'of', 'a', 'book', ',', "''", 'thought', 'Alice', '``', 'without',
    'pictures', 'or', 'conversations', '?', "''", 'So', 'she', 'was', 'considering', 'in', 'her',
    'own', 'mind', '(', 'as', 'well', 'as', 'she', 'could', ',', 'for', 'the', 'hot', 'day', 'made',
    'her', 'feel', 'very', 'sleepy', 'and', 'stupid', ')', ',', 'whether', 'the', 'pleasure', 'of',
    'making', 'a', 'daisy-chain', 'would', 'be', 'worth', 'the', 'trouble', 'of', 'getting', 'up',
    'and', 'picking', 'the', 'daisies', ',', 'when', 'suddenly', 'a', 'White', 'Rabbit', 'with',
    'pink', 'eyes', 'ran', 'close', 'by', 'her', '.', 'There', 'was', 'nothing', 'so', 'very',
    'remarkable', 'in', 'that', ';', 'nor', 'did', 'Alice', 'think', 'it', 'so', 'very', 'much',
    'out', 'of', 'the', 'way', 'to', 'hear', 'the', 'Rabbit', 'say', 'to', 'itself', ',', '``', 'Oh',
    'dear', '!', 'Oh', 'dear', '!', 'I', 'shall', 'be', 'late', '!', "''", '(', 'when', 'she',
    'thought', 'it', 'over', 'afterwards', ',', 'it', 'occurred', 'to', 'her', 'that', 'she',
    'ought', 'to', 'have', 'wondered', 'at', 'this', ',', 'but', 'at', 'the', 'time', 'it', 'all',
    'seemed', 'quite', 'natural', ')', ';', 'but', 'when', 'the', 'Rabbit', 'actually', 'took', 'a',
    'watch', 'out', 'of', 'its', 'waistcoat-pocket', ',', 'and', 'looked', 'at', 'it', ',', 'and',
    'then', 'hurried', 'on', ',', 'Alice', 'started', 'to', 'her', 'feet', ',', 'for', 'it',
    'flashed', 'across', 'her', 'mind', 'that', 'she', 'had', 'never', 'before', 'seen', 'a',
    'rabbit', 'with', 'either', 'a', 'waistcoat-pocket', ',', 'or', 'a', 'watch', 'to', 'take',
```

```
'out', 'of', 'it', ',', 'and', 'burning', 'with', 'curiosity', ',', 'she', 'ran', 'across',
'the', 'field', 'after', 'it', ',', 'and', 'fortunately', 'was', 'just', 'in', 'time', 'to',
'see', 'it', 'pop', 'down', 'a', 'large', 'rabbit-hole', 'under', 'the', 'hedge', '.']
```

What if we wanted to tokenise each individual sentence of the text one by one?

```
for sentence in sent_tokenize(text): # Iterate the sentences in "text"
    print(word_tokenize(sentence)) # Print the word tokenisation results. Don't forget the four
    spaces indentation that Python expects!
```

The output will look like:

```
['Alice', 'was', 'beginning', 'to', 'get', 'very', 'tired', 'of', 'sitting', 'by', 'her', 'sister',
    'on', 'the', 'bank', ',', 'and', 'of', 'having', 'nothing', 'to', 'do', ':', 'once', 'or',
    'twice', 'she', 'had', 'peeped', 'into', 'the', 'book', 'her', 'sister', 'was', 'reading', ',',
    'but', 'it', 'had', 'no', 'pictures', 'or', 'conversations', 'in', 'it', ',', '``', 'and',
    'what', 'is', 'the', 'use', 'of', 'a', 'book', ',', "''", 'thought', 'Alice', '``', 'without',
    'pictures', 'or', 'conversations', '?', "''"]
['So', 'she', 'was', 'considering', 'in', 'her', 'own', 'mind', '(', 'as', 'well', 'as', 'she',
    'could', ',', 'for', 'the', 'hot', 'day', 'made', 'her', 'feel', 'very', 'sleepy', 'and',
    'stupid', ')', ',', 'whether', 'the', 'pleasure', 'of', 'making', 'a', 'daisy-chain', 'would',
    'be', 'worth', 'the', 'trouble', 'of', 'getting', 'up', 'and', 'picking', 'the', 'daisies', ',',
    'when', 'suddenly', 'a', 'White', 'Rabbit', 'with', 'pink', 'eyes', 'ran', 'close', 'by', 'her',
    '.']
['There', 'was', 'nothing', 'so', 'very', 'remarkable', 'in', 'that', ';', 'nor', 'did', 'Alice',
    'think', 'it', 'so', 'very', 'much', 'out', 'of', 'the', 'way', 'to', 'hear', 'the', 'Rabbit',
    'say', 'to', 'itself', ',', '``', 'Oh', 'dear', '!']
['Oh', 'dear', '!']
['I', 'shall', 'be', 'late', '!', "''"]
['(', 'when', 'she', 'thought', 'it', 'over', 'afterwards', ',', 'it', 'occurred', 'to', 'her',
    'that', 'she', 'ought', 'to', 'have', 'wondered', 'at', 'this', ',', 'but', 'at', 'the', 'time',
    'it', 'all', 'seemed', 'quite', 'natural', ')', ';', 'but', 'when', 'the', 'Rabbit', 'actually',
    'took', 'a', 'watch', 'out', 'of', 'its', 'waistcoat-pocket', ',', 'and', 'looked', 'at', 'it',
    ',', 'and', 'then', 'hurried', 'on', ',', 'Alice', 'started', 'to', 'her', 'feet', ',', 'for',
    'it', 'flashed', 'across', 'her', 'mind', 'that', 'she', 'had', 'never', 'before', 'seen', 'a',
    'rabbit', 'with', 'either', 'a', 'waistcoat-pocket', ',', 'or', 'a', 'watch', 'to', 'take',
    'out', 'of', 'it', ',', 'and', 'burning', 'with', 'curiosity', ',', 'she', 'ran', 'across',
    'the', 'field', 'after', 'it', ',', 'and', 'fortunately', 'was', 'just', 'in', 'time', 'to',
    'see', 'it', 'pop', 'down', 'a', 'large', 'rabbit-hole', 'under', 'the', 'hedge', '.']
```

2.3.3 Lowercasing

As you can see in the list of tokens above, some words have uppercase letters. Let's convert all characters in our tokens to lowercase. We can use the lower() function to convert all characters in a string to lowercase. For example:

```
test_string = "uNIverSIty"
test_string_lowercase = test_string.lower()
print(test_string_lowercase)
```

The output will look like:

```
university
```

Let's now convert all words in the text from alice.txt to lowercase, iterating through each sentence and word, and saving the lowercase words in a list:

```
words_lowercase = []
for sentence in sent_tokenize(text): # Iterate the sentences in "text"
    for word in word_tokenize(sentence): # Iterate the words in "sentence"
        words_lowercase.append(word.lower()) # Convert word to lowercase and add it to the
    "words_lowercase" list
print(words_lowercase) # Print the list of lowercase words
```

The output will look like:

```
['alice', 'was', 'beginning', 'to', 'get', 'very', 'tired', 'of', 'sitting', 'by', 'her', 'sister',
    'on', 'the', 'bank', ',', 'and', 'of', 'having', 'nothing', 'to', 'do', ':', 'once', 'or',
    'twice', 'she', 'had', 'peeped', 'into', 'the', 'book', 'her', 'sister', 'was', 'reading', ',',
    'but', 'it', 'had', 'no', 'pictures', 'or', 'conversations', 'in', 'it', ',', '``', 'and',
    'what', 'is', 'the', 'use', 'of', 'a', 'book', ',', "''", 'thought', 'alice', '``', 'without',
    'pictures', 'or', 'conversations', '?', "''", 'so', 'she', 'was', 'considering', 'in', 'her',
    'own', 'mind', '(', 'as', 'well', 'as', 'she', 'could', ',', 'for', 'the', 'hot', 'day', 'made',
    'her', 'feel', 'very', 'sleepy', 'and', 'stupid', ')', ',', 'whether', 'the', 'pleasure', 'of',
    'making', 'a', 'daisy-chain', 'would', 'be', 'worth', 'the', 'trouble', 'of', 'getting', 'up',
    'and', 'picking', 'the', 'daisies', ',', 'when', 'suddenly', 'a', 'white', 'rabbit', 'with',
    'pink', 'eyes', 'ran', 'close', 'by', 'her', '.', 'there', 'was', 'nothing', 'so', 'very',
    'remarkable', 'in', 'that', ';', 'nor', 'did', 'alice', 'think', 'it', 'so', 'very', 'much',
    'out', 'of', 'the', 'way', 'to', 'hear', 'the', 'rabbit', 'say', 'to', 'itself', ',', '``', 'oh',
    'dear', '!', 'oh', 'dear', '!', 'i', 'shall', 'be', 'late', '!', "''", '(', 'when', 'she',
    'thought', 'it', 'over', 'afterwards', ',', 'it', 'occurred', 'to', 'her', 'that', 'she',
    'ought', 'to', 'have', 'wondered', 'at', 'this', ',', 'but', 'at', 'the', 'time', 'it', 'all',
    'seemed', 'quite', 'natural', ')', ';', 'but', 'when', 'the', 'rabbit', 'actually', 'took', 'a',
    'watch', 'out', 'of', 'its', 'waistcoat-pocket', ',', 'and', 'looked', 'at', 'it', ',', 'and',
    'then', 'hurried', 'on', ',', 'alice', 'started', 'to', 'her', 'feet', ',', 'for', 'it',
    'flashed', 'across', 'her', 'mind', 'that', 'she', 'had', 'never', 'before', 'seen', 'a',
    'rabbit', 'with', 'either', 'a', 'waistcoat-pocket', ',', 'or', 'a', 'watch', 'to', 'take',
    'out', 'of', 'it', ',', 'and', 'burning', 'with', 'curiosity', ',', 'she', 'ran', 'across',
    'the', 'field', 'after', 'it', ',', 'and', 'fortunately', 'was', 'just', 'in', 'time', 'to',
    'see', 'it', 'pop', 'down', 'a', 'large', 'rabbit-hole', 'under', 'the', 'hedge', '.']
```

As you can see, there are no uppercase characters left in the words list.

2.3.4 Stop words removal

Stop words are common words in any language that don't hold semantic meaning of their own. In many NLP applications, it is useful to remove stop words from a text. To achieve this, we use stop words lists that have been compiled for each language. We can access the list of English stop words from the NLTK library using the following:

```
nltk.download('stopwords') # Download the stopwords lists from NLTK

from nltk.corpus import stopwords # Import the stop words lists from NLTK
stopwords_english = stopwords.words('english') # Load the stop words list for English in variable
    "stopwords_english"
print(stopwords_english) # Print the "stopwords_english" list
```

The output will look like:

```
['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"]
```

As you can see, the list of words from alice.txt contains a number of stop words. Let's use NLTK's English stop words list to remove them:

```
words_lowercase_nostopwords = [] # Create empty list for remaining words
words_removed = [] # Create empty list for removed words
for word in words_lowercase: # Iterate lowercase words' list
    if word not in stopwords_english: # Check if word is in the stop words list
        words_lowercase_nostopwords.append(word)
    else:
        words_removed.append(word)
print(words_lowercase_nostopwords) # Print list of remaining words
print(words_removed) # Print list of removed words
```

The output will look like:

```
['alice', 'beginning', 'get', 'tired', 'sitting', 'sister', 'bank', ',', 'nothing', ':', 'twice',
    'peeped', 'book', 'sister', 'reading', ',', 'pictures', 'conversations', ',', '``', 'use',
    'book', ',', "''", 'thought', 'alice', '``', 'without', 'pictures', 'conversations', '?', "''",
    'considering', 'mind', '(', 'well', 'could', ',', 'hot', 'day', 'made', 'feel', 'sleepy',
    'stupid', ')', ',', 'whether', 'pleasure', 'making', 'daisy-chain', 'would', 'worth', 'trouble',
    'getting', 'picking', 'daisies', ',', 'suddenly', 'white', 'rabbit', 'pink', 'eyes', 'ran',
    'close', '.', 'nothing', 'remarkable', ';', 'alice', 'think', 'much', 'way', 'hear', 'rabbit',
    'say', ',', '``', 'oh', 'dear', '!', 'oh', 'dear', '!', 'shall', 'late', '!', "''", '(',
    'thought', 'afterwards', ',', 'occurred', 'ought', 'wondered', ',', 'time', 'seemed', 'quite',
    'natural', ')', ';', 'rabbit', 'actually', 'took', 'watch', 'waistcoat-pocket', ',', 'looked',
    ',', 'hurried', ',', 'alice', 'started', 'feet', ',', 'flashed', 'across', 'mind', 'never',
    'seen', 'rabbit', 'either', 'waistcoat-pocket', ',', 'watch', 'take', ',', 'burning',
    'curiosity', ',', 'ran', 'across', 'field', ',', 'fortunately', 'time', 'see', 'pop', 'large',
    'rabbit-hole', 'hedge', '.']
['was', 'to', 'very', 'of', 'by', 'her', 'on', 'the', 'and', 'of', 'having', 'to', 'do', 'once',
    'or', 'she', 'had', 'into', 'the', 'her', 'was', 'but', 'it', 'had', 'no', 'or', 'in', 'it',
    'and', 'what', 'is', 'the', 'of', 'a', 'or', 'so', 'she', 'was', 'in', 'her', 'own', 'as', 'as',
    'she', 'for', 'the', 'her', 'very', 'and', 'the', 'of', 'a', 'be', 'the', 'of', 'up', 'and',
    'the', 'when', 'a', 'with', 'by', 'her', 'there', 'was', 'so', 'very', 'in', 'that', 'nor',
    'did', 'it', 'so', 'very', 'out', 'of', 'the', 'to', 'the', 'to', 'itself', 'i', 'be', 'when',
    'she', 'it', 'over', 'it', 'to', 'her', 'that', 'she', 'to', 'have', 'at', 'this', 'but', 'at',
    'the', 'it', 'all', 'but', 'when', 'the', 'a', 'out', 'of', 'its', 'and', 'at', 'it', 'and',
    'then', 'on', 'to', 'her', 'for', 'it', 'her', 'that', 'she', 'had', 'before', 'a', 'with', 'a',
    'or', 'a', 'to', 'out', 'of', 'it', 'and', 'with', 'she', 'the', 'after', 'it', 'and', 'was',
    'just', 'in', 'to', 'it', 'down', 'a', 'under', 'the']
```

As you can see in the list of the remaining words and the list of the removed words, we removed multiple words from the alice.txt text, including “was”, “to”, “she”, “it”, “for”, etc.

2.3.5 Punctuation removal

If you look at the list of the remaining words, you can see that some of its contents are not actual words, but they are punctuation marks instead. Let’s remove these punctuation marks from the words list:

```
from string import punctuation # Import the punctuation marks string
print(punctuation)
print("Variable type: ", type(punctuation))
```

The output will look like:

```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
Variable type: <class 'str'>
```

As you can see, the variable “punctuation” contains all the potential punctuation marks. However, the variable is a string. To help us iterate and compare with the list of words, we will first convert the string “punctuation” to a list with one character (punctuation mark) per element.

```
punctuation_list = list(punctuation) # Convert punctuation to a list
print(punctuation_list)

words_lowercase_nostopwords_no_punctuation = []
for word in words_lowercase_nostopwords:
```

```
    if word not in punctuation_list:
        words_lowercase_nostopwords_no_punctuation.append(word)
print(words_lowercase_nostopwords_no_punctuation)
```

The output will look like:

```
['!', '"', '#', '$', '%', '&', "'", '(', ')', '*', '+', ',', '-', '.', '/', ':', ';', '<', '=', '>',
    '?', '@', '[', '\\', ']', '^', '_', '`', '{', '|', '}', '~']
['alice', 'beginning', 'get', 'tired', 'sitting', 'sister', 'bank', 'nothing', 'twice', 'peeped',
    'book', 'sister', 'reading', 'pictures', 'conversations', '``', 'use', 'book', "''", 'thought',
    'alice', '``', 'without', 'pictures', 'conversations', "''", 'considering', 'mind', 'well',
    'could', 'hot', 'day', 'made', 'feel', 'sleepy', 'stupid', 'whether', 'pleasure', 'making',
    'daisy-chain', 'would', 'worth', 'trouble', 'getting', 'picking', 'daisies', 'suddenly', 'white',
    'rabbit', 'pink', 'eyes', 'ran', 'close', 'nothing', 'remarkable', 'alice', 'think', 'much',
    'way', 'hear', 'rabbit', 'say', '``', 'oh', 'dear', 'oh', 'dear', 'shall', 'late', "''",
    'thought', 'afterwards', 'occurred', 'ought', 'wondered', 'time', 'seemed', 'quite', 'natural',
    'rabbit', 'actually', 'took', 'watch', 'waistcoat-pocket', 'looked', 'hurried', 'alice',
    'started', 'feet', 'flashed', 'across', 'mind', 'never', 'seen', 'rabbit', 'either',
    'waistcoat-pocket', 'watch', 'take', 'burning', 'curiosity', 'ran', 'across', 'field',
    'fortunately', 'time', 'see', 'pop', 'large', 'rabbit-hole', 'hedge']
```

As you can see, the majority of the punctuation marks where removed. But what about the remaining “ ‘ ‘ ” and “ ’ ’ ”? The punctuation marks list that we used did not contain double quotes and as a consequence, “ ‘ ‘ ” and “ ’ ’ ” were treated as valid words. We can address this issue by adding the respective double quote characters in the list of punctuation marks.

Note that there are alternative ways for removing punctuation marks or special characters from text, like for example the use of regular expressions.

2.3.6 Stemming

Consider the words “walk”, “walks”, “walking”, and “walked”. It is evident that all these words are different forms of the word “walk”. We can use stemming to reduce each word to its respective stem , i.e. the core meaning-bearing unit of a word. NLTK comes with some common stemming algorithms. Let’s use the Porter Stemming algorithm to reduce these four words to their stems:

```
from nltk.stem import PorterStemmer # Import the Porter stemmer from NLTK

porter = PorterStemmer() # Create a Porter stemmer object
for word in ['walk','walks','walking','walked']:
    print(word,"->",porter.stem(word))
```

The output will look like:

```
walk -> walk
walks -> walk
walking -> walk
walked -> walk
```

Let’s now stem the words from the alice.txt text using the Porter stemmer:

```
words_stemmed = []
for word in words_lowercase_nostopwords_no_punctuation:
    words_stemmed.append(porter.stem(word))
print(words_stemmed)
```

The output will look like:

```
['alic', 'begin', 'get', 'tire', 'sit', 'sister', 'bank', 'noth', 'twice', 'peep', 'book', 'sister',
    'read', 'pictur', 'convers', 'use', 'book', 'thought', 'alic', 'without', 'pictur', 'convers',
    'consid', 'mind', 'well', 'could', 'hot', 'day', 'made', 'feel', 'sleepi', 'stupid', 'whether',
    'pleasur', 'make', 'daisy-chain', 'would', 'worth', 'troubl', 'get', 'pick', 'daisi', 'suddenli',
    'white', 'rabbit', 'pink', 'eye', 'ran', 'close', 'noth', 'remark', 'alic', 'think', 'much',
    'way', 'hear', 'rabbit', 'say', 'oh', 'dear', 'oh', 'dear', 'shall', 'late', 'thought',
```

```
    'afterward', 'occur', 'ought', 'wonder', 'time', 'seem', 'quit', 'natur', 'rabbit', 'actual',
    'took', 'watch', 'waistcoat-pocket', 'look', 'hurri', 'alic', 'start', 'feet', 'flash', 'across',
    'mind', 'never', 'seen', 'rabbit', 'either', 'waistcoat-pocket', 'watch', 'take', 'burn',
    'curios', 'ran', 'across', 'field', 'fortun', 'time', 'see', 'pop', 'larg', 'rabbit-hol', 'hedg']
```

As you can see, the Porter stemming algorithm reduced the words from alice.txt to their stems. However, it is evident that a lot of the stems do not correspond to real words from the English language.

2.3.7 Lemmatisation

Lemmatisation can address this issue by reducing each word to the respective dictionary headword. Let's lemmatise the words "walk", "walks", "walking", and "walked" using NLTK's WordNetLemmatizer:

```
nltk.download('wordnet') # Download the WordNetLemmatizer package
from nltk.stem import WordNetLemmatizer # Import the WordNetLemmatizer

wnl = WordNetLemmatizer() # Create a WordNetLemmatizer object
for word in ['walk','walks','walking','walked']:
    print(word,"->",wnl.lemmatize(word))
```

The output will look like:

```
walk -> walk
walks -> walk
walking -> walking
walked -> walked
```

As you can see, the words "walk" and "walks" were converted to the lemma "walk", but "walking" and "walked" were not changed. The reason for this is that the WordNetLemmatizer considers by default all inputs as nouns, thus "walks" is considered as the plural form of "walk" and is converted to "walk", but "walking" and "walked" are valid lemmas and remain unchanged. To lemmatise all words to their base verb form, we must indicate that we are inputting verb forms:

```
for word in ['walk','walks','walking','walked']:
    print(word,"->",wnl.lemmatize(word,pos='v'))
```

The output will look like:

```
walk -> walk
walks -> walk
walking -> walk
walked -> walk
```

Using the "pos" argument, the input words were handled as verb forms from the WordNetLemmatizer and returned the base form "walk" for all words. The options for "pos" are noun (n), verb (v), adverb (r), adjective (a). More information about the WordNetLemmatizer here: `https://www.nltk.org/api/nltk.stem.wordnet.html`

Let's lemmatise the text from alice.txt, treating the input as nouns and then as verbs:

```
lemmas_noun = []
lemmas_verb = []
for word in words_lowercase_nostopwords_no_punctuation:
    lemmas_noun.append(wnl.lemmatize(word,pos='n'))
    lemmas_verb.append(wnl.lemmatize(word,pos='v'))
print(lemmas_noun)
print(lemmas_verb)
```

The output will look like:

```
['alice', 'beginning', 'get', 'tired', 'sitting', 'sister', 'bank', 'nothing', 'twice', 'peeped',
    'book', 'sister', 'reading', 'picture', 'conversation', 'use', 'book', 'thought', 'alice',
    'without', 'picture', 'conversation', 'considering', 'mind', 'well', 'could', 'hot', 'day',
    'made', 'feel', 'sleepy', 'stupid', 'whether', 'pleasure', 'making', 'daisy-chain', 'would',
    'worth', 'trouble', 'getting', 'picking', 'daisy', 'suddenly', 'white', 'rabbit', 'pink', 'eye',
```

```
    'ran', 'close', 'nothing', 'remarkable', 'alice', 'think', 'much', 'way', 'hear', 'rabbit',
    'say', 'oh', 'dear', 'oh', 'dear', 'shall', 'late', 'thought', 'afterwards', 'occurred', 'ought',
    'wondered', 'time', 'seemed', 'quite', 'natural', 'rabbit', 'actually', 'took', 'watch',
    'waistcoat-pocket', 'looked', 'hurried', 'alice', 'started', 'foot', 'flashed', 'across', 'mind',
    'never', 'seen', 'rabbit', 'either', 'waistcoat-pocket', 'watch', 'take', 'burning', 'curiosity',
    'ran', 'across', 'field', 'fortunately', 'time', 'see', 'pop', 'large', 'rabbit-hole', 'hedge']
['alice', 'begin', 'get', 'tire', 'sit', 'sister', 'bank', 'nothing', 'twice', 'peep', 'book',
    'sister', 'read', 'picture', 'conversations', 'use', 'book', 'think', 'alice', 'without',
    'picture', 'conversations', 'consider', 'mind', 'well', 'could', 'hot', 'day', 'make', 'feel',
    'sleepy', 'stupid', 'whether', 'pleasure', 'make', 'daisy-chain', 'would', 'worth', 'trouble',
    'get', 'pick', 'daisies', 'suddenly', 'white', 'rabbit', 'pink', 'eye', 'run', 'close',
    'nothing', 'remarkable', 'alice', 'think', 'much', 'way', 'hear', 'rabbit', 'say', 'oh', 'dear',
    'oh', 'dear', 'shall', 'late', 'think', 'afterwards', 'occur', 'ought', 'wonder', 'time', 'seem',
    'quite', 'natural', 'rabbit', 'actually', 'take', 'watch', 'waistcoat-pocket', 'look', 'hurry',
    'alice', 'start', 'feet', 'flash', 'across', 'mind', 'never', 'see', 'rabbit', 'either',
    'waistcoat-pocket', 'watch', 'take', 'burn', 'curiosity', 'run', 'across', 'field',
    'fortunately', 'time', 'see', 'pop', 'large', 'rabbit-hole', 'hedge']
```

However, this approach is not practical. Ideally, we would like to know what part of speech each word is and use the lemmatiser accordingly.

2.3.8 Part of Speech (POS) tagging

Part of Speech (POS) tagging is used to detect which part of speech each word in a sentence refers to. Let's use NLTK's POS tagging algorithm to assign POS tags to each of the words in the sentence "I had been a student here for a long time.":

```
nltk.download('averaged_perceptron_tagger')

from nltk import pos_tag

pos_tagged_sentence = pos_tag(word_tokenize('I had been a student here for a long time'))
print(pos_tagged_sentence)
```

The output will look like:

```
[('I', 'PRP'), ('had', 'VBD'), ('been', 'VBN'), ('a', 'DT'), ('student', 'NN'), ('here', 'RB'),
    ('for', 'IN'), ('a', 'DT'), ('long', 'JJ'), ('time', 'NN')]
```

As you can see, each word in the sentence has been annotated with a POS tags. These POS tags refer to the Penn Treebank POS tags (`https://catalog.ldc.upenn.edu/docs/LDC95T7/cl93.html`). However, the WordNetLemmatizer expects different tag names. To address this issue, we have to first convert the Penn Treebank POS tags to the format expected by the WordNetLemmatizer.

```
'''Convert Penn Treebank POS tags to WordNet'''
def penn_to_wordnet(penn_pos_tag):
    tag_dictionary = {'NN':'n', 'JJ':'a','VB':'v', 'RB':'r'}
    try:
        # If the first two characters of the Penn Treebank POS tag are in the ``tag_dictionary''
        return tag_dictionary[penn_pos_tag[:2]]
    except:
        return 'n' # Default to Noun if no mapping avalable.

lemmas = []
for word, tag in pos_tagged_sentence:
    lemmas.append(wnl.lemmatize(word.lower(), pos=penn_to_wordnet(tag)))

print('I have been a student here for a long time')
print(lemmas)
```

The output will look like:

```
I had been a student here for a long time
['i', 'have', 'be', 'a', 'student', 'here', 'for', 'a', 'long', 'time']
```

As you can see, the sentence was properly lemmatised using POS tagging to inform the lemmatiser about the part of speech that each word refers to.
Note: Please note that the conversion from Penn Treebank POS tags to WordNet format used here is a simplification. Full conversion tables should be used for better results. Also, the code above uses exception handling via the "try" and "except" statements. You can read more about exception handling in Python from here: `https://docs.python.org/3/tutorial/errors.html`

Let's now lemmatise the alice.txt text:

```
lemmas_alice = []

for sent in sent_tokenize(text): # Tokenise text into sentences
    pos_tagged_sentence_alice = pos_tag(word_tokenize(sent)) # Get POS tags for each sentence
    for word, tag in pos_tagged_sentence_alice: # Iterate though POS tagged words
        if word.lower() not in punctuation_list: # Ignore words that are punctuation marks
            lemmas_alice.append(wnl.lemmatize(word.lower(), pos=penn_to_wordnet(tag))) # Lemmatise word
print(lemmas_alice)
```

The output will look like:

```
['alice', 'be', 'begin', 'to', 'get', 'very', 'tired', 'of', 'sit', 'by', 'her', 'sister', 'on',
    'the', 'bank', 'and', 'of', 'have', 'nothing', 'to', 'do', 'once', 'or', 'twice', 'she', 'have',
    'peep', 'into', 'the', 'book', 'her', 'sister', 'be', 'read', 'but', 'it', 'have', 'no',
    'picture', 'or', 'conversation', 'in', 'it', '``', 'and', 'what', 'be', 'the', 'use', 'of', 'a',
    'book', "''", 'think', 'alice', '``', 'without', 'picture', 'or', 'conversation', "''", 'so',
    'she', 'be', 'consider', 'in', 'her', 'own', 'mind', 'as', 'well', 'a', 'she', 'could', 'for',
    'the', 'hot', 'day', 'make', 'her', 'feel', 'very', 'sleepy', 'and', 'stupid', 'whether', 'the',
    'pleasure', 'of', 'make', 'a', 'daisy-chain', 'would', 'be', 'worth', 'the', 'trouble', 'of',
    'get', 'up', 'and', 'pick', 'the', 'daisy', 'when', 'suddenly', 'a', 'white', 'rabbit', 'with',
    'pink', 'eye', 'run', 'close', 'by', 'her', 'there', 'be', 'nothing', 'so', 'very', 'remarkable',
    'in', 'that', 'nor', 'do', 'alice', 'think', 'it', 'so', 'very', 'much', 'out', 'of', 'the',
    'way', 'to', 'hear', 'the', 'rabbit', 'say', 'to', 'itself', '``', 'oh', 'dear', 'oh', 'dear',
    'i', 'shall', 'be', 'late', "''", 'when', 'she', 'think', 'it', 'over', 'afterwards', 'it',
    'occur', 'to', 'her', 'that', 'she', 'ought', 'to', 'have', 'wonder', 'at', 'this', 'but', 'at',
    'the', 'time', 'it', 'all', 'seem', 'quite', 'natural', 'but', 'when', 'the', 'rabbit',
    'actually', 'take', 'a', 'watch', 'out', 'of', 'it', 'waistcoat-pocket', 'and', 'look', 'at',
    'it', 'and', 'then', 'hurry', 'on', 'alice', 'start', 'to', 'her', 'foot', 'for', 'it', 'flash',
    'across', 'her', 'mind', 'that', 'she', 'have', 'never', 'before', 'see', 'a', 'rabbit', 'with',
    'either', 'a', 'waistcoat-pocket', 'or', 'a', 'watch', 'to', 'take', 'out', 'of', 'it', 'and',
    'burn', 'with', 'curiosity', 'she', 'run', 'across', 'the', 'field', 'after', 'it', 'and',
    'fortunately', 'be', 'just', 'in', 'time', 'to', 'see', 'it', 'pop', 'down', 'a', 'large',
    'rabbit-hole', 'under', 'the', 'hedge']
```

2.4 Exercises

Exercise 2.1 Section 2.3.5: Create a new punctuation marks list to address the issue of the remaining " ‘ ‘ " and " ’ ’ ".

Exercise 2.2 Section 2.3.5: Remove stop words and punctuation marks without iterating twice through all words.

Exercise 2.3 Load the text from dune.txt. Compute the number of words in the text, not including punctuation marks.

Exercise 2.4 Lemmatise the text from dune.txt and print a list of all the lemmas. Remember to convert all words to lowercase and to remove punctuation marks.

Exercise 2.5 Create a list of the unique lemmas in dune.txt, count their number and print the list and the number of lemmas.

Exercise 2.6 Create a custom function to divide English text into sentences without using NLTK or other tokenisers. Consider that a sentence ends when one of the following characters occurs: ".", "?", "!". Also remember to take into consideration the line change character "\n". Test your function on the text from dune.txt.

Workshop 3: Text representation

In this lab we will work on different ways to represent text.

3.1 Load corpus

We will use the Gutenberg corpus from NLTK. Let's first load the Gutenberg corpus:

```
import nltk # Import the NLTK library
nltk.download('gutenberg') # Download the gutenberg corpus
from nltk.corpus import gutenberg # Import the gutenberg corpus from NLTK
gutenberg.fileids() # List file-ids in the corpus
```

The output will look like:

```
['austen-emma.txt',
 'austen-persuasion.txt',
 'austen-sense.txt',
 'bible-kjv.txt',
 'blake-poems.txt',
 'bryant-stories.txt',
 'burgess-busterbrown.txt',
 'carroll-alice.txt',
 'chesterton-ball.txt',
 'chesterton-brown.txt',
 'chesterton-thursday.txt',
 'edgeworth-parents.txt',
 'melville-moby_dick.txt',
 'milton-paradise.txt',
 'shakespeare-caesar.txt',
 'shakespeare-hamlet.txt',
 'shakespeare-macbeth.txt',
 'whitman-leaves.txt']
```

Let's compute some statistics for each document in the Gutenberg corpus, like the number of words, the number of sentences, and the number of characters:

```
print("Chars\tWords\tSents\tFile")
for fileid in gutenberg.fileids(): # Iterate through files in corpus
    num_chars = len(gutenberg.raw(fileid))
    num_words = len(gutenberg.words(fileid))
    num_sents = len(gutenberg.sents(fileid))
    print("%7.0f\t%7.0f\t%7.0f\t%s" % (num_chars,num_words,num_sents,fileid))
```

The output will look like:

```
Chars      Words   Sents   File
 887071   192427    7752  austen-emma.txt
 466292    98171    3747  austen-persuasion.txt
 673022   141576    4999  austen-sense.txt
4332554  1010654   30103  bible-kjv.txt
  38153     8354     438  blake-poems.txt
```

```
 249439   55563    2863 bryant-stories.txt
  84663   18963    1054 burgess-busterbrown.txt
 144395   34110    1703 carroll-alice.txt
 457450   96996    4779 chesterton-ball.txt
 406629   86063    3806 chesterton-brown.txt
 320525   69213    3742 chesterton-thursday.txt
 935158  210663   10230 edgeworth-parents.txt
1242990  260819   10059 melville-moby_dick.txt
 468220   96825    1851 milton-paradise.txt
 112310   25833    2163 shakespeare-caesar.txt
 162881   37360    3106 shakespeare-hamlet.txt
 100351   23140    1907 shakespeare-macbeth.txt
 711215  154883    4250 whitman-leaves.txt
```

3.2 Vocabulary

As we can see above, each document consists of a few thousand words. But these words are not unique. Natural language consists of words that convey meaning and are re-used and combined to form different sentences. The set of unique words that are used in each document constitutes its vocabulary.

3.2.1 Words in corpus

Consider the text "The new table is red. The blue table is broken." Let's compute its vocabulary. First we should tokenise the text into words, convert them all to lowercase and remove punctuation marks:

```
from nltk import word_tokenize # Import the word_tokenize function from NLTK
from string import punctuation

punctuation_list = list(punctuation) # Convert string with punctuation marks to list

text = "The new table is red. The blue table is broken."

text_tokens_processed = []
text_tokens = word_tokenize(text) # Tokenise the text into words
for token in text_tokens: # Iterate through the available tokens
    if token not in punctuation_list: # Omit tokens that are punctuation marks
        text_tokens_processed.append(token.lower()) # Add lowercase version of token to list

print("List of processed words:",text_tokens_processed)
```

The output will look like:

```
List of processed words: ['the', 'new', 'table', 'is', 'red', 'the', 'blue', 'table', 'is', 'broken']
```

3.2.2 Unique words in corpus

As you can see in the list of words, the words "the", "table" and "is" appear two times each in the text. Let's now compute the vocabulary of this text, i.e. the list of unique words used in this text. To achieve this, we will use Python's **set** type. A set is similar to a list but allows only unique elements.

```
vocabulary = set() # Create an empty set
for word in text_tokens_processed: # Iterate through available words
    vocabulary.add(word) # Add word to set

print("Vocabulary:",vocabulary)
print("Vocabulary size:",len(vocabulary))

vocabulary2 = set(text_tokens_processed)
print("\nVocabulary2:",vocabulary2)
print("Vocabulary2 size:",len(vocabulary2))
```

The output will look like:

```
Vocabulary: {'the', 'red', 'broken', 'blue', 'table', 'is', 'new'}
Vocabulary size: 7

Vocabulary2: {'the', 'red', 'broken', 'blue', 'table', 'is', 'new'}
Vocabulary2 size: 7
```

As you can see, the vocabulary used by the text "The new table is red. The blue table is broken." consists of the seven following words: is, table, blue, red, broken, new, the. Note that you don't have to add the contents of a list in a set one-by-one, as shown for variable "vocabulary2".

3.2.3 Vocabulary of multiple documents

Let's now compute the vocabulary for each document in the Gutenberg corpus:

```
for fileid in gutenberg.fileids(): # Iterate through documents in corpus
    vocabulary_of_document = set() # Create empty set
    for word in gutenberg.words(fileid): # Iterate through words in document
        if word not in punctuation_list: # Omit tokens that are punctuation marks
            vocabulary_of_document.add(word.lower())
    print("%6.0f\t%s" % (len(vocabulary_of_document),fileid))
```

The output will look like:

```
  7328  austen-emma.txt
  5820  austen-persuasion.txt
  6388  austen-sense.txt
 12755  bible-kjv.txt
  1521  blake-poems.txt
  3925  bryant-stories.txt
  1547  burgess-busterbrown.txt
  2622  carroll-alice.txt
  8313  chesterton-ball.txt
  7780  chesterton-brown.txt
  6335  chesterton-thursday.txt
  8432  edgeworth-parents.txt
 17215  melville-moby_dick.txt
  9007  milton-paradise.txt
  3019  shakespeare-caesar.txt
  4703  shakespeare-hamlet.txt
  3451  shakespeare-macbeth.txt
 12437  whitman-leaves.txt
```

As you can see, we computed the vocabulary for each document in the Gutenberg corpus and printed its size. However, vocabularies from different documents are expected to have similar words in them since all texts are in the same language. Let's compute the vocabulary that covers all documents in the dataset:

```
vocabulary_of_corpus = set() # Create empty set
for fileid in gutenberg.fileids(): # Iterate through documents in corpus
    for word in gutenberg.words(fileid): # Iterate through words in document
        if word not in punctuation_list: # Omit tokens that are punctuation marks
            vocabulary_of_corpus.add(word.lower())

print("Vocabulary of Gutenberg corpus:",len(vocabulary_of_corpus),"words")
```

The output will look like:

```
Vocabulary of Gutenberg corpus: 42314 words
```

As you can see, the vocabulary that covers all documents in the Gutenberg corpus is much larger than individual document vocabularies but significantly smaller that the sum of all the individual vocabularies.

3.3 One-hot encoding

3.3.1 One-hot encoding of words in vocabulary

Consider again the text “The new table is red. The blue table is broken.” We have already computed its vocabulary and would like to compute the One-Hot representation of each word in the vocabulary.

```
from numpy import array # Import array type from numpy
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder

vocabulary = ['is', 'table', 'blue', 'red', 'broken', 'new', 'the']
data = array(vocabulary) # Convert to array because it is required by the LabelEncoder() object
print(data,"\n")

# Integer encoding - Assigns a unique index to each unique word
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(data)
print(integer_encoded,"\n")

# One-Hot encoding - Assigns a One-Hot binary representation to each word
onehot_encoder = OneHotEncoder(sparse_output=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
onehot_encoded = onehot_encoder.fit_transform(integer_encoded)
print(onehot_encoded,"\n")

for i in range(len(data)):
    print(onehot_encoded[i],"->",data[i])
```

The output will look like:

```
['is' 'table' 'blue' 'red' 'broken' 'new' 'the']

[2 5 0 4 1 3 6]

[[0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1.]]

[0. 0. 1. 0. 0. 0. 0.] -> is
[0. 0. 0. 0. 0. 1. 0.] -> table
[1. 0. 0. 0. 0. 0. 0.] -> blue
[0. 0. 0. 0. 1. 0. 0.] -> red
[0. 1. 0. 0. 0. 0. 0.] -> broken
[0. 0. 0. 1. 0. 0. 0.] -> new
[0. 0. 0. 0. 0. 0. 1.] -> the
```

Let’s create a dictionary to easily encode our text and one-hot encode the word “red”:

```
dictionary = {}
for i in range(len(data)):
    dictionary[data[i]] = onehot_encoded[i]

print(dictionary)

print("\nred =",dictionary['red'])
```

The output will look like:

```
{'is': array([0., 0., 1., 0., 0., 0., 0.]), 'table': array([0., 0., 0., 0., 0., 1., 0.]), 'blue':
    array([1., 0., 0., 0., 0., 0., 0.]), 'red': array([0., 0., 0., 0., 1., 0., 0.]), 'broken':
    array([0., 1., 0., 0., 0., 0., 0.]), 'new': array([0., 0., 0., 1., 0., 0., 0.]), 'the':
```

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

red = [0. 0. 0. 0. 1. 0. 0.]
```

Consider the one-hot encoded word (0, 0, 0, 0, 1, 0, 0,). How can we convert it back to its respective real word? Let's create a function to do this:

```
def get_label_from_dictionary(dictionary,value):
    for word, one_hot in dictionary.items(): # Iterate all pairs (word, one-hot representation) in
     the dictionary
        if (one_hot == value).all(): # Compare equality between numpy arrays
            return word

print(get_label_from_dictionary(dictionary,[0,0,0,0,1,0,0]))
```

The output will look like:

```
red
```

3.3.2 One-hot encoding of text

Consider again the text "The new table is red. The blue table is broken." We have already computed its vocabulary and the one-hot representation of each word in the vocabulary. How can we one-hot encode the whole text? To do so, we have to perform a logical OR operation between the one-hot vectors of its constituent words.

```
from numpy import logical_or # Import the element-wise logical OR function from numpy
from numpy import zeros # Import the zeros function from numpy

text = "The new table is red. The blue table is broken."
print("List of processed words:",text_tokens_processed)

result = zeros(len(dictionary)) # Start from zero-valued vector - Convert to numpy array
for word in text_tokens_processed: # Iterate words in text
    print(result.astype(int), "OR", dictionary[word],"= ",end='')
    result = logical_or(result,dictionary[word]) # Compute the element-wise logical or between the
     partial result and the one-hot representation of the word
    print(result.astype(int))

print("\nOne-Hot encoded text:",result.astype(int))
```

The output will look like:

```
List of processed words: ['the', 'new', 'table', 'is', 'red', 'the', 'blue', 'table', 'is', 'broken']
[0 0 0 0 0 0 0] OR [0. 0. 0. 0. 0. 0. 1.] = [0 0 0 0 0 0 1]
[0 0 0 0 0 0 1] OR [0. 0. 0. 1. 0. 0. 0.] = [0 0 0 1 0 0 1]
[0 0 0 1 0 0 1] OR [0. 0. 0. 0. 0. 1. 0.] = [0 0 0 1 0 1 1]
[0 0 0 1 0 1 1] OR [0. 0. 1. 0. 0. 0. 0.] = [0 0 1 1 0 1 1]
[0 0 1 1 0 1 1] OR [0. 0. 0. 0. 1. 0. 0.] = [0 0 1 1 1 1 1]
[0 0 1 1 1 1 1] OR [0. 0. 0. 0. 0. 0. 1.] = [0 0 1 1 1 1 1]
[0 0 1 1 1 1 1] OR [1. 0. 0. 0. 0. 0. 0.] = [1 0 1 1 1 1 1]
[1 0 1 1 1 1 1] OR [0. 0. 0. 0. 0. 1. 0.] = [1 0 1 1 1 1 1]
[1 0 1 1 1 1 1] OR [0. 0. 1. 0. 0. 0. 0.] = [1 0 1 1 1 1 1]
[1 0 1 1 1 1 1] OR [0. 1. 0. 0. 0. 0. 0.] = [1 1 1 1 1 1 1]

One-Hot encoded text: [1 1 1 1 1 1 1]
```

Let's create a function for one-hot encoding text and do the same for the text "the broken table":

```
def get_text_one_hot_encoding(words_list, dictionary):
    result = zeros(len(dictionary)) # Start from zero-valued vector - Convert to numpy array
    for word in words_list: # Iterate words in text
        result = logical_or(result,dictionary[word]) # Compute the element-wise logical or between the
     partial result and the one-hot representation of the word
```

```
    return result.astype(int)

words_list = ['the','broken','table']

print("\nOne-Hot encoded text:",get_text_one_hot_encoding(words_list,dictionary))
```

The output will look like:

```
One-Hot encoded text: [0 1 0 0 0 1 1]
```

3.4 Term Frequency (TF) representation

3.4.1 Compute TF of words in text

Let's now compute the term frequency of each word in the text "The new table is red. The blue table is broken."

```
from nltk import FreqDist # Import the FreqDist function from NLTK

text = "The new table is red. The blue table is broken."
text_tokens_processed = ['the', 'new', 'table', 'is', 'red', 'the', 'blue', 'table', 'is', 'broken']
vocabulary = {'new', 'broken', 'the', 'blue', 'table', 'red', 'is'}

tf = FreqDist(text_tokens_processed) # Compute term frequency of words
print(tf,"\n")

vocabulary = sorted(vocabulary) # Sort alphabetiacally for better presentation
for word in vocabulary:
    print("%5.0f %s" % (tf[word],word))
```

The output will look like:

```
<FreqDist with 7 samples and 10 outcomes>

    1 blue
    1 broken
    2 is
    1 new
    1 red
    2 table
    2 the
```

3.4.2 TF representation of documents

Let's now compute the TF representation of the text "The new table is red. The blue table is broken." and the text "The new table is broken":

```
text_tf = []
for word in vocabulary:
    text_tf.append(tf[word])

print(text_tf,"->",text)

text2 = "The new table is broken"
text2_tokens_processed = ['the','new','table','is','broken']
tf2 = FreqDist(text2_tokens_processed) # Compute term frequency of words

text2_tf = []
for word in vocabulary:
    text2_tf.append(tf2[word])

print(text2_tf,"->",text2)
```

The output will look like:

```
[1, 1, 2, 1, 1, 2, 2] -> The new table is red. The blue table is broken.
[0, 1, 1, 1, 0, 1, 1] -> The new table is broken
```

3.5 Term Frequency - Inverse Document Frequency (TF-IDF)

Consider a corpus consisting of the three following documents:

1. "The new table is red. The blue table was broken."
2. "The new movie that we watched yesterday was terrible."
3. "We raised the red and blue flag yesterday."

3.5.1 Document Frequency (DF)

Let's compute the Document Frequency (DF) of each word in the above corpus. Remember that the DF of a word is equal to the number of documents in a corpus that the word appears in.

```
# Create a list of the lists of lowercase words without punctuation marks for each document
texts_words_processed = []
texts_words_processed.append(['the','new','table','is','red','the','blue','table','was','broken'])
texts_words_processed.append(['the','new','movie','that','we','watched','yesterday','was','terrible'])
texts_words_processed.append(['we','raised','the','red','and','blue','flag','yesterday'])

print(texts_words_processed)

# Create the vocabulary
vocabulary_texts = set()
for doc in texts_words_processed:
    for word in doc:
        vocabulary_texts.add(word)

vocabulary_texts = sorted(vocabulary_texts) # Sort vocabulary alphabetically for better presentation

print("\nVocabulary:",vocabulary_texts)

DF = dict() # Create an empty dictionary
for word in vocabulary_texts: # Iterate through words in vocabulary
    cnt = 0
    for doc in texts_words_processed: # Iterate through documents
        if word in doc:
            cnt += 1 # cnt += 1 is equal to cnt = cnt + 1
    DF[word] = cnt

print("\nDocument frequencies:",DF)
```

The output will look like:

```
[['the', 'new', 'table', 'is', 'red', 'the', 'blue', 'table', 'was', 'broken'], ['the', 'new',
    'movie', 'that', 'we', 'watched', 'yesterday', 'was', 'terrible'], ['we', 'raised', 'the', 'red',
    'and', 'blue', 'flag', 'yesterday']]

Vocabulary: ['and', 'blue', 'broken', 'flag', 'is', 'movie', 'new', 'raised', 'red', 'table',
    'terrible', 'that', 'the', 'was', 'watched', 'we', 'yesterday']

Document frequencies: {'and': 1, 'blue': 2, 'broken': 1, 'flag': 1, 'is': 1, 'movie': 1, 'new': 2,
    'raised': 1, 'red': 2, 'table': 1, 'terrible': 1, 'that': 1, 'the': 3, 'was': 2, 'watched': 1,
    'we': 2, 'yesterday': 2}
```

3.5.2 Inverse Document Frequency (IDF)

The Inverse Document Frequency (IDF) of a word is the logarithmically scaled inverse fraction of the documents that contain the word (obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient). IDF is defined as:

$$IDF(t, D) = \log\left(\frac{N}{DF(t, D)}\right) \tag{3.1}$$

where t is a word (term), D the corpus, and N the number of documents in the corpus.

Let's compute the IDF for the examined corpus:

```
import math # Import math library

N = 3 # The corpus contains 3 documents

IDF = dict() # Create an empty dictionary

for word in vocabulary_texts: # Iterate through words in vocabulary
    IDF[word] = math.log( N / DF[word] ) # Compute IDF of word

print("IDF:",IDF)
```

The output will look like:

```
IDF: {'and': 1.0986122886681098, 'blue': 0.4054651081081644, 'broken': 1.0986122886681098, 'flag':
    1.0986122886681098, 'is': 1.0986122886681098, 'movie': 1.0986122886681098, 'new':
    0.4054651081081644, 'raised': 1.0986122886681098, 'red': 0.4054651081081644, 'table':
    1.0986122886681098, 'terrible': 1.0986122886681098, 'that': 1.0986122886681098, 'the': 0.0,
    'was': 0.4054651081081644, 'watched': 1.0986122886681098, 'we': 0.4054651081081644, 'yesterday':
    0.4054651081081644}
```

As you can see above, the more documents that a word appeared in, the lower the IDF for the word. Note that IDF is 0 for the word "the" that appeared in all documents (as a consequence of $N = DF \Rightarrow IDF = log1 = 0$)

3.5.3 Term Frequency - Inverse Document Frequency (TF-IDF)

Term Frequency - Inverse Document Frequency (TF-IDF) is defined as the product of TF and IDF:

$$TFIDF(t, d, D) = TF(t, d) \cdot IDF(t, D) \tag{3.2}$$

where d is a document of corpus D.

Let's compute the TF-IDF for each word in each document of the examined corpus

```
from nltk import FreqDist # Import the FreqDist function from NLTK

TF = []
for doc in texts_words_processed: # Iterate though documents
    TF.append(FreqDist(doc)) # Compute word frequency

print(TF,"\n")

TFIDF = []
for tf_doc in TF:
    tfidf_doc = dict()
    for word in vocabulary_texts: # Iterate through words in vocabulary
        tfidf_doc[word] = tf_doc[word] * IDF[word] # Compute TF-IDF - tf_doc is of type FreqDist and
    will return 0 for words that don't exist
    TFIDF.append(tfidf_doc)

cnt = 0
for tfidf_doc in TFIDF:
    print("Text",cnt,"TF-IDF:",tfidf_doc,"\n")
    cnt += 1
```

The output will look like:

```
[FreqDist({'the': 2, 'table': 2, 'new': 1, 'is': 1, 'red': 1, 'blue': 1, 'was': 1, 'broken': 1}),
    FreqDist({'the': 1, 'new': 1, 'movie': 1, 'that': 1, 'we': 1, 'watched': 1, 'yesterday': 1,
    'was': 1, 'terrible': 1}), FreqDist({'we': 1, 'raised': 1, 'the': 1, 'red': 1, 'and': 1, 'blue':
    1, 'flag': 1, 'yesterday': 1})]

Text 0 TF-IDF: {'and': 0.0, 'blue': 0.4054651081081644, 'broken': 1.0986122886681098, 'flag': 0.0,
    'is': 1.0986122886681098, 'movie': 0.0, 'new': 0.4054651081081644, 'raised': 0.0, 'red':
    0.4054651081081644, 'table': 2.1972245773362196, 'terrible': 0.0, 'that': 0.0, 'the': 0.0, 'was':
    0.4054651081081644, 'watched': 0.0, 'we': 0.0, 'yesterday': 0.0}

Text 1 TF-IDF: {'and': 0.0, 'blue': 0.0, 'broken': 0.0, 'flag': 0.0, 'is': 0.0, 'movie':
    1.0986122886681098, 'new': 0.4054651081081644, 'raised': 0.0, 'red': 0.0, 'table': 0.0,
    'terrible': 1.0986122886681098, 'that': 1.0986122886681098, 'the': 0.0, 'was':
    0.4054651081081644, 'watched': 1.0986122886681098, 'we': 0.4054651081081644, 'yesterday':
    0.4054651081081644}

Text 2 TF-IDF: {'and': 1.0986122886681098, 'blue': 0.4054651081081644, 'broken': 0.0, 'flag':
    1.0986122886681098, 'is': 0.0, 'movie': 0.0, 'new': 0.0, 'raised': 1.0986122886681098, 'red':
    0.4054651081081644, 'table': 0.0, 'terrible': 0.0, 'that': 0.0, 'the': 0.0, 'was': 0.0,
    'watched': 0.0, 'we': 0.4054651081081644, 'yesterday': 0.4054651081081644}
```

3.6 Exercises

Exercise 3.1 Create a dictionary with the one-hot encoding of the vocabulary of the carroll-alice.txt file from the Gutenberg corpus.

Exercise 3.2 Based on the one-hot encoding of the vocabulary of the carroll-alice.txt from the Gutenberg corpus, one-hot encode the sentences “this is an old house”, “this is a new house”, and “he left his house” and compute their cosine distance.

Exercise 3.3 Using the vocabulary of the carroll-alice.txt document from the Gutenberg corpus, compute the TF-IDF representations and the respective cosine and euclidean distances of the documents in a corpus containing alice.txt and dune.txt.

Exercise 3.4 Use alice.txt to create a vocabulary and add to it the “unknown” word “<UNK>”. Use this vocabulary to create the one-hot, the TF, the log normalised TF, and the TF-IDF representations of the alice.txt and the dune.txt documents.

Workshop 4: N-Grams

4.1 N-grams

We know that the probability of a sequence of words can be computed as:

$$P(w_1, w_2, w_3, ..., w_n) = P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_1, w_2) \cdot ... \cdot P(w_n|w_1, w_2, w_3, ..., w_{n-1})$$

Unfortunately, we will never have enough data to compute the probability for any given word sequence. However, we can make some simplification assumptions and use N-grams to approximate the probabilities.

Let's compute various N-grams for the document "The new table is red. The blue table is broken.". Let's first load the document and compute the list of lowercase words, remove punctuation and compute the vocabulary.

```
text = "The new table is red. The blue table is broken."
words_processed = ['the', 'new', 'table', 'is', 'red', 'the', 'blue', 'table', 'is', 'broken']
vocabulary = set() # Create an empty set
for word in words_processed: # Iterate through available words
    vocabulary.add(word) # Add word to set

print("Document:",text)
print("Pre-processed words:",words_processed)
print("Document size:",len(words_processed))
print("Vocabulary:",vocabulary)
print("Vocabulary size:",len(vocabulary))
```

The output will look like:

```
Document: The new table is red. The blue table is broken.
Pre-processed words: ['the', 'new', 'table', 'is', 'red', 'the', 'blue', 'table', 'is', 'broken']
Document size: 10
Vocabulary: {'red', 'new', 'the', 'blue', 'broken', 'is', 'table'}
Vocabulary size: 7
```

As you can see, the pre-processed document consists of 10 words and uses a vocabulary of 7 words.

4.2 Unigrams (1-Grams)

4.2.1 Compute unigrams

Unigrams (1-Grams) make the assumption that the probability of a word in a sequence of words depends only on the word itself (0th order Markovian assumption). Let's now compute the unigrams for the document "The new table is red. The blue table is broken.". Each unique word in the vocabulary constitutes a unigram of the document. Let's compute the counts of each unigram in our document.

```
import nltk
from nltk import FreqDist # Import the FreqDist function from NLTK

tf = FreqDist(words_processed) # Compute term frequency of words
print(tf,"\n")

vocabulary = sorted(vocabulary) # Sort alphabetically for better presentation
unigrams = dict() # Create empty dictionary for unigrams
```

```
for word in vocabulary:
    unigrams[word] = tf[word]

print(unigrams)
```

The output will look like:

```
<FreqDist with 7 samples and 10 outcomes>

{'blue': 1, 'broken': 1, 'is': 2, 'new': 1, 'red': 1, 'table': 2, 'the': 2}
```

4.2.2 Unigram probability

The probability of a unigram for a word w_n is computed as:

$$P(w_n) = \frac{count(w_n)}{Total\,words} = \frac{count(w_n)}{\sum_{i=1}^{|V|} count(w_i)}$$

where w_n is a word, V the vocabulary, and $|V|$ the size of the vocabulary. Also, remember that when using unigrams, it is assumed that $P(w_n|w_{n-1}) \approx P(w_n)$

Let's now compute the probability of each word in the vocabulary:

```
total_words = len(words_processed) # Compute total words in corpus

unigram_probabilities = dict() # Create empty dictionary for unigram probabilities
for word in unigrams:
    unigram_probabilities[word] = unigrams[word] / total_words # Compute P(w_n)

print("Unigram probabilities:",unigram_probabilities)
```

The output will look like:

```
Unigram probabilities:
{'blue': 0.1, 'broken': 0.1, 'is': 0.2, 'new': 0.1, 'red': 0.1, 'table': 0.2, 'the': 0.2}
```

4.2.3 Sentence probability

Let's now compute the probability of the sentences "the new table is red" and "the black table" using the unigrams that we have computed.

$$P(\text{the new table is red}) \approx P(\text{the}) \cdot P(\text{new}) \cdot P(\text{table}) \cdot P(\text{is}) \cdot P(\text{red})$$

$$P(\text{the black table}) \approx P(\text{the}) \cdot P(\text{black}) \cdot P(\text{table})$$

Keep in mind that for words that don't exist in our corpus, the probability should be $P(\text{“unknown”}) = \frac{0}{Total\,words} = 0$

```
from collections import defaultdict

pw = defaultdict(lambda: 0, unigram_probabilities) # Create a dictionary that will return 0 for
    unknown words

print(pw,"\n")

p_text1 = pw["the"]*pw["new"]*pw["table"]*pw["is"]*pw["red"]
p_text2 = pw["the"]*pw["black"]*pw["table"]

print("P(the new table is red)= %f" % p_text1)
print("P(the black table)= %f" % p_text2)
```

The output will look like:

```
defaultdict(<function <lambda> at 0x0131CDB0>, {'blue': 0.1, 'broken': 0.1, 'is': 0.2, 'new': 0.1,
    'red': 0.1, 'table': 0.2, 'the': 0.2})

P(the new table is red)= 0.000080
P(the black table)= 0.000000
```

Note that we used the data structure “defaultdict” (`https://docs.python.org/3/library/collections.html#collections.defaultdict`) for storing the unigram probabilities. The reason for not using the default dictionary type of Python is that we need to set the probability to 0 for any unigram that is unknown and thus doesn't have a probability associated with it.

4.2.4 Smoothing

Notice that the probability for the sentence “the black table” is 0, as a result of the word “black” not existing in our corpus. However, this is a valid sentence in the English language. We will apply Add-λ smoothing in order to address the issue of zero-valued probabilities for unknown words.

$$P_{\text{Add-}\lambda}(w_n) = \frac{count(w_n) + \lambda}{\lambda|V| + \sum_{i=1}^{|V|} count(w_i)}$$

Let's now compute again the probability of the sentences “the new table is red” and “the black table” using the unigrams that we have computed and Add-λ smoothing for $\lambda = 0.001$. Remember that the probability of an unknown word when Add-λ smoothing is used will be:

$$P_{\text{Add-}\lambda}(\text{“unknown”}) = \frac{0 + \lambda}{\lambda|V| + \sum_{i=1}^{|V|} count(w_i)} = \frac{\lambda}{\lambda|V| + \sum_{i=1}^{|V|} count(w_i)}$$

```
V = len(vocabulary) # Compute words in vocabulary
total_words = len(words_processed) # Compute total words in corpus
l = 0.001 # Define lambda for Add-lambda smoothing

p_unknown = (0 + l) / ( (l*V) + total_words) # Compute the probability of unknown words using
    add-lambda smoothing
print("P(unknown)=%f\n" % p_unknown)

unigram_probabilities_addl = dict() # Create empty dictionary for unigram probabilities
for word in unigrams:
   unigram_probabilities_addl[word] = (unigrams[word] + l) / (total_words + (l*V)) # Compute P(w_n)

print("Unigram probabilities (Add-lambda smoothing):\n",unigram_probabilities_addl,"\n")

plw = defaultdict(lambda: p_unknown, unigram_probabilities_addl) # Create a dictionary that will
    return p_unknown for unknown words

pl_text1 = plw["the"]*plw["new"]*plw["table"]*plw["is"]*plw["red"]
pl_text2 = plw["the"]*plw["black"]*plw["table"]

print("P(the new table is red)= %f" % pl_text1)
print("P(the black table)= %f" % pl_text2)
```

The output will look like:

```
P(unknown)=0.000100

Unigram probabilities (Add-lambda smoothing):
 {'blue': 0.10002997901468971, 'broken': 0.10002997901468971, 'is': 0.1999600279804137, 'new':
    0.10002997901468971, 'red': 0.10002997901468971, 'table': 0.1999600279804137, 'the':
    0.1999600279804137}

P(the new table is red)= 0.000080
P(the black table)= 0.000004
```

As you can see above, we can now compute the probability of word sequences that contain words that were not included in the corpus we used for creating our unigrams.

4.3 Bigrams (2-Grams)

4.3.1 Compute bigrams

Bigrams (2-Grams) make a 1st order Markovian assumption that the probability of a word in a sequence of words depends on the word and the previous word.

Let's now compute the bigrams for the examined corpus:

```
from nltk.util import ngrams

text = "The new table is red. The blue table is broken."
words_processed = ['the', 'new', 'table', 'is', 'red', 'the', 'blue', 'table', 'is', 'broken']

bigrams = ngrams(words_processed,2) # Compute the bigrams in the text

bigrams_unique = set() # Create empty set for unique bigrams
for bigram in bigrams:
    print(bigram)
    bigrams_unique.add(bigram) # Add bigram to set

print("\nUnique bigrams:\n",bigrams_unique)
```

The output will look like:

```
('the', 'new')
('new', 'table')
('table', 'is')
('is', 'red')
('red', 'the')
('the', 'blue')
('blue', 'table')
('table', 'is')
('is', 'broken')

Unique bigrams:
 {('the', 'new'), ('is', 'broken'), ('red', 'the'), ('is', 'red'), ('blue', 'table'), ('the',
    'blue'), ('table', 'is'), ('new', 'table')}
```

4.3.2 Bigram probability

The probability of the bigram (w_{n-1}, w_n) is computed as:

$$P(w_n|w_{n-1}) = \frac{count(w_{n-1}, w_n)}{count(w_{n-1})}$$

Let's now compute the probability of each unique bigram in the examined corpus:

```
bigrams = ngrams(words_processed,2) # Compute the bigrams in the text

bigram_freq = FreqDist(bigrams).items() # Compute frequency distribution for all the bigrams in the
    text

print(bigram_freq)
```

The output will look like:

```
dict_items([(('the', 'new'), 1), (('new', 'table'), 1), (('table', 'is'), 2), (('is', 'red'), 1),
    (('red', 'the'), 1), (('the', 'blue'), 1), (('blue', 'table'), 1), (('is', 'broken'), 1)])
```

4.3.3 Sentence probability

Let's now compute again the probability of the sentences "the new table is red" and "the black table" using the bigrams and the unigrams that we have computed. We will use the symbols $< s >$ and $< /s >$ to indicate the start and the end of a sentence respectively

$$P(<\mathrm{s}> \text{the new table is red} </\mathrm{s}>) \approx$$
$$P(\text{the}|<\mathrm{s}>) \cdot P(\text{new}|\text{the}) \cdot P(\text{table}|\text{new}) \cdot P(\text{is}|\text{table}) \cdot P(\text{red}|\text{is}) \cdot P(</\mathrm{s}>|\text{red}) \approx$$
$$\frac{count(<\mathrm{s}>, \text{the})}{count(<\mathrm{s}>)} \cdot \frac{count(\text{the}, \text{new})}{count(\text{the})} \cdot \frac{count(\text{new}, \text{table})}{count(\text{new})} \cdot \frac{count(\text{table}, \text{is})}{count(\text{table})} \cdot \frac{count(\text{is}, \text{red})}{count(\text{is})} \cdot \frac{count(\text{red}, </\mathrm{s}>)}{count(\text{red})}$$

$$P(<\mathrm{s}> \text{the black table} </\mathrm{s}>) \approx P(\text{the}|<\mathrm{s}>) \cdot P(\text{black}|\text{the}) \cdot P(\text{table}|\text{black}) \cdot P(</\mathrm{s}>|\text{table})$$
$$\approx \frac{count(<\mathrm{s}>, \text{the})}{count(<\mathrm{s}>)} \cdot \frac{count(\text{the}, \text{black})}{count(\text{the})} \cdot \frac{count(\text{black}, \text{table})}{count(\text{black})} \cdot \frac{count(\text{table}, </\mathrm{s}>)}{count(\text{table})}$$

Keep in mind that for bigrams that don't exist in our corpus, the probability should be $P(\text{“unknown”}) = 0$. Please note that the use of the tokens $<\mathrm{s}>$ and $</\mathrm{s}>$ is optional!

```
text = "The new table is red. The blue table is broken."
# Add tokens indicating the start and end of a sentence in the respective position
text2 = "<s> The new table is red. </s> <s> The blue table is broken. </s>"
words_processed = ['<s>','the', 'new', 'table', 'is', 'red','</s>','<s>','the', 'blue', 'table',
    'is', 'broken','</s>']

vocabulary = set() # Create an empty set
for word in words_processed: # Iterate through available words
   vocabulary.add(word) # Add word to set

tf = FreqDist(words_processed) # Compute term frequency of words

vocabulary = sorted(vocabulary) # Sort alphabetically for better presentation
ugf = dict() # Create empty dictionary for unigram counts
for word in vocabulary:
   ugf[word] = tf[word]

ugf = defaultdict(lambda: 0, ugf) # Create a dictionary that will return 0 for unknown unigrams
print("Unigram counts:",ugf,"\n")

bigrams = ngrams(words_processed,2) # Compute the bigrams in the text
bigram_freq = FreqDist(bigrams).items() # Compute frequency distribution for all the bigrams in the
    text
print("Bigram counts:",bigram_freq,"\n")

bgf = defaultdict(lambda: 0, bigram_freq) # Create a dictionary that will return 0 for unknown bigrams

def p_big(bigram, bigram_frequencies, unigram_frequencies): # Create function to compute bigram
    probability
   if(bigram_frequencies[bigram]==0):
      return 0
   else:
      return bigram_frequencies[bigram] / unigram_frequencies[bigram[0]]

p_text1 = p_big(('<s>','the'),bgf,ugf)*p_big(('the','new'),bgf,ugf)*p_big(('new','table'),bgf,ugf)
*p_big(('table','is'),bgf,ugf)*p_big(('is','red'),bgf,ugf)*p_big(('red','</s>'),bgf,ugf)

p_text2 = p_big(('<s>','the'),bgf,ugf)*p_big(('the','black'),bgf,ugf)*
p_big(('black','table'),bgf,ugf)*p_big(('table','</s>'),bgf,ugf)

print("P(<s> the new table is red </s>)= %f" % p_text1)
print("P(<s> the black table </s>)= %f" % p_text2)
```

The output will look like:

```
Unigram counts: {'</s>': 2, '<s>': 2, 'blue': 1, 'broken': 1, 'is': 2, 'new': 1, 'red': 1, 'table':
    2, 'the': 2}

Bigram counts: dict_items([(('<s>', 'the'), 2), (('the', 'new'), 1), (('new', 'table'), 1),
    (('table', 'is'), 2), (('is', 'red'), 1), (('red', '</s>'), 1), (('</s>', '<s>'), 1), (('the',
    'blue'), 1), (('blue', 'table'), 1), (('is', 'broken'), 1), (('broken', '</s>'), 1)])
```

```
P(<s> the new table is red </s>)= 0.250000
P(<s> the black table </s>)= 0.000000
```

4.3.4 Smoothing

Notice that the probability for the sentence "$<$ s $>$ the black table $<$ /s $>$" is 0, as a result of the bigrams (black,the), (table,black), and (black,</s>) not existing in our corpus. However, this is a valid sentence in the English language. We will apply Add-λ smoothing in order to address the issue of zero-valued probabilities for unknown bigrams.

$$P_{\text{Add-}\lambda}(w_n|w_{n-1}) = \frac{count(w_{n-1}, w_n) + \lambda}{\lambda|V| + count(w_{n-1})}$$

Let's now compute again the probability of the sentences "$<$ s $>$ the new table is red $<$ /s $>$" and "$<$ s $>$ the black table $<$ /s $>$" using the bigrams that we have computed and Add-λ smoothing for $\lambda = 0.01$.

```
def pl_big(bigram, bigram_frequencies, unigram_frequencies,l): # Create function to compute bigram
     probability with add-lambda smoothing
    return (bigram_frequencies[bigram] + l) / ( (l*len(unigram_frequencies)) +
     unigram_frequencies[bigram[0]])

l = 0.01

pl_text1 = pl_big(('<s>','the'),bgf,ugf,l)*pl_big(('the','new'),bgf,ugf,l)*
pl_big(('new','table'),bgf,ugf,l)*pl_big(('table','is'),bgf,ugf,l)*pl_big(('is','red'),bgf,ugf,l)*
pl_big(('red','</s>'),bgf,ugf,l)

pl_text2 = pl_big(('<s>','the'),bgf,ugf,l)*pl_big(('the','black'),bgf,ugf,l)*
pl_big(('black','table'),bgf,ugf,l)*pl_big(('table','</s>'),bgf,ugf,l)

print("P(<s> the new table is red </s>)= %f" % pl_text1)
print("P(<s> the black table </s>)= %f" % pl_text2)
```

The output will look like:

```
P(<s> the new table is red </s>)= 0.185455
P(<s> the black table </s>)= 0.000002
```

4.4 Trigrams (3-Grams)

4.4.1 Compute trigrams

Trigrams (3-Grams) make a 2nd order Markovian assumption that the probability of a word in a sequence of words depends on the word and the previous two words.

Let's now compute the trigrams for the examined corpus, after adding the tokens "<s>" "<s>" and "</s>" "</s>" at the beginning and end of each sentence respectively.

```
text = "<s> <s> The new table is red. </s> </s> <s> <s> The blue table is broken. </s> </s>"
words_processed = ['<s>','<s>','the', 'new', 'table', 'is', 'red','</s>','</s>','<s>','<s>', 'the',
     'blue', 'table', 'is', 'broken','</s>','</s>']

trigrams = ngrams(words_processed,3) # Compute the trigrams in the text

trigrams_unique = set() # Create empty set for unique trigrams
for trigram in trigrams:
    print(trigram)
    trigrams_unique.add(trigram) # Add trigram to set

print("\nUnique trigrams:\n",trigrams_unique)
```

The output will look like:

```
('<s>', '<s>', 'the')
('<s>', 'the', 'new')
('the', 'new', 'table')
('new', 'table', 'is')
('table', 'is', 'red')
('is', 'red', '</s>')
('red', '</s>', '</s>')
('</s>', '</s>', '<s>')
('</s>', '<s>', '<s>')
('<s>', '<s>', 'the')
('<s>', 'the', 'blue')
('the', 'blue', 'table')
('blue', 'table', 'is')
('table', 'is', 'broken')
('is', 'broken', '</s>')
('broken', '</s>', '</s>')

Unique trigrams:
 {('<s>', 'the', 'blue'), ('is', 'red', '</s>'), ('</s>', '</s>', '<s>'), ('new', 'table', 'is'),
    ('blue', 'table', 'is'), ('is', 'broken', '</s>'), ('broken', '</s>', '</s>'), ('table', 'is',
    'broken'), ('table', 'is', 'red'), ('<s>', '<s>', 'the'), ('red', '</s>', '</s>'), ('the',
    'blue', 'table'), ('</s>', '<s>', '<s>'), ('<s>', 'the', 'new'), ('the', 'new', 'table')}
```

4.4.2 Trigram probability

The probability of the trigram (w_{n-2}, w_{n-1}, w_n) is computed as:

$$P(w_n|w_{n-2}, w_{n-1}) = \frac{count(w_{n-2}, w_{n-1}, w_n)}{count(w_{n-2}, w_{n-1})}$$

Let's now compute the probability of each unique trigram in the examined corpus:

```python
trigrams = ngrams(words_processed,3) # Compute the trigrams in the text

trigram_freq = FreqDist(trigrams).items() # Compute frequency distribution for all the trigrams in
    the text

print(trigram_freq)
```

The output will look like:

```
dict_items([(('<s>', '<s>', 'the'), 2), (('<s>', 'the', 'new'), 1), (('the', 'new', 'table'), 1),
    (('new', 'table', 'is'), 1), (('table', 'is', 'red'), 1), (('is', 'red', '</s>'), 1), (('red',
    '</s>', '</s>'), 1), (('</s>', '</s>', '<s>'), 1), (('</s>', '<s>', '<s>'), 1), (('<s>', 'the',
    'blue'), 1), (('the', 'blue', 'table'), 1), (('blue', 'table', 'is'), 1), (('table', 'is',
    'broken'), 1), (('is', 'broken', '</s>'), 1), (('broken', '</s>', '</s>'), 1)])
```

4.4.3 Sentence probability

Let's now compute again the probability of the sentences "$<$ s $>$ $<$ s $>$ the new table is red $<$ /s $>$ $<$ /s $>$" and "$<$ s $>$ $<$ s $>$the black table $<$ /s $>$ $<$ /s $>$" using the trigrams and the bigrams that we have computed.

$$\begin{aligned}
P(<\text{s}><\text{s}>\text{ the new table is red }<\text{/s}><\text{/s}>) \approx \\
P(\text{the}|<\text{s}>,<\text{s}>) \cdot P(\text{new}|,<\text{s}>,\text{the}) \cdot P(\text{table}|\text{the},\text{new}) \cdot P(\text{is}|\text{new},\text{table}) \cdot \\
P(\text{red}|\text{table},\text{is}) \cdot P(<\text{/s}>|\text{is},\text{red}) \cdot P(<\text{/s}>|\text{red},<\text{/s}>) \approx \\
\frac{count(<\text{s}>,<\text{s}>,\text{the})}{count(<\text{s}>,<\text{s}>)} \cdot \frac{count(<\text{s}>,\text{the},\text{new})}{count(<\text{s}>,\text{the})} \cdot \frac{count(\text{new},\text{table},\text{is})}{count(\text{new},\text{table})} \cdot \frac{count(\text{table},\text{is},\text{red})}{count(\text{table},\text{is})} \cdot \\
\frac{count(\text{is},\text{red},<\text{/s}>)}{count(\text{is},\text{red})} \cdot \frac{count(\text{red},<\text{/s}>,<\text{/s}>)}{count(\text{red},<\text{/s}>)}
\end{aligned}$$

$$P(<\text{s}><\text{s}>\text{the black table}</\text{s}></\text{s}>) \approx$$
$$P(\text{the}|<\text{s}>,<\text{s}>) \cdot P(\text{black}|<\text{s}>,\text{the}) \cdot P(\text{table}|\text{the},\text{black}) \cdot P(</\text{s}>|\text{black},\text{table}) \cdot$$
$$P(</\text{s}>|\text{table},</\text{s}>) \approx$$
$$\frac{count(<\text{s}>,<\text{s}>,\text{the})}{count(<\text{s}>,<\text{s}>)} \cdot \frac{count(<\text{s}>,\text{the},\text{black})}{count(<\text{s}>,\text{the})} \cdot \frac{count(\text{the},\text{black},\text{table})}{count(\text{the},\text{black})} \cdot$$
$$\frac{count(\text{black},\text{table},</\text{s}>)}{count(\text{black},\text{table})} \cdot \frac{count(\text{table},</\text{s}>,</\text{s}>)}{count(\text{table},</\text{s}>)}$$

Keep in mind that for trigrams that don't exist in our corpus, the probability should be $P(\text{“unknown”}) = 0$. Also, please note that the use of the $<\text{s}>$ and $</\text{s}>$ tokens is optional!

```
text = "The new table is red. The blue table is broken."
# Add tokens indicating the start and end of a sentence in the respective position
text3 = "<s> <s> The new table is red. </s> </s> <s> <s> The blue table is broken. </s> </s>"
words_processed = ['<s>','<s>','the', 'new', 'table', 'is', 'red','</s>','</s>','<s>','<s>','the',
    'blue', 'table', 'is', 'broken','</s>','</s>']

bigrams = ngrams(words_processed,2) # Compute the bigrams in the text
bigram_freq = FreqDist(bigrams).items() # Compute frequency distribution for all the bigrams in the
    text
print("Bigram counts:",bigram_freq,"\n")

trigrams = ngrams(words_processed,3) # Compute the trigrams in the text
trigram_freq = FreqDist(trigrams).items() # Compute frequency distribution for all the trigrams in
    the text
print("Trigram counts:",trigram_freq,"\n")

bgf = defaultdict(lambda: 0, bigram_freq) # Create a dictionary that will return 0 for unknown bigrams
tgf = defaultdict(lambda: 0, trigram_freq) # Create a dictionary that will return 0 for unknown
    trigrams

def p_trig(trigram, trigram_frequencies, bigram_frequencies): # Create function to compute trigram
    probability
    if(trigram_frequencies[trigram]==0):
        return 0
    else:
        return trigram_frequencies[trigram] / bigram_frequencies[(trigram[0],trigram[1])]

p_text1 = p_trig(('<s>','<s>','the'),tgf,bgf)*p_trig(('<s>','the','new'),tgf,bgf)*
p_trig(('the','new','table'),tgf,bgf)*p_trig(('new','table','is'),tgf,bgf)*
p_trig(('table','is','red'),tgf,bgf)*p_trig(('is','red','</s>'),tgf,bgf)*
p_trig(('red','</s>','</s>'),tgf,bgf)

p_text2 = p_trig(('<s>','<s>','the'),tgf,bgf)*p_trig(('<s>','the','black'),tgf,bgf)*
p_trig(('the','black','table'),tgf,bgf)*p_trig(('black','table','</s>'),tgf,bgf)*
p_trig(('table','</s>','</s>'),tgf,bgf)

print("P(<s> <s> the new table is red </s> </s>)= %f" % p_text1)
print("P(<s> <s> the black table </s> </s>)= %f" % p_text2)
```

The output will look like:

```
Bigram counts: dict_items([(('<s>', '<s>'), 2), (('<s>', 'the'), 2), (('the', 'new'), 1), (('new',
    'table'), 1), (('table', 'is'), 2), (('is', 'red'), 1), (('red', '</s>'), 1), (('</s>', '</s>'),
    2), (('</s>', '<s>'), 1), (('the', 'blue'), 1), (('blue', 'table'), 1), (('is', 'broken'), 1),
    (('broken', '</s>'), 1)])

Trigram counts: dict_items([(('<s>', '<s>', 'the'), 2), (('<s>', 'the', 'new'), 1), (('the', 'new',
    'table'), 1), (('new', 'table', 'is'), 1), (('table', 'is', 'red'), 1), (('is', 'red', '</s>'),
    1), (('red', '</s>', '</s>'), 1), (('</s>', '</s>', '<s>'), 1), (('</s>', '<s>', '<s>'), 1),
    (('<s>', 'the', 'blue'), 1), (('the', 'blue', 'table'), 1), (('blue', 'table', 'is'), 1),
    (('table', 'is', 'broken'), 1), (('is', 'broken', '</s>'), 1), (('broken', '</s>', '</s>'), 1)])

P(<s> <s> the new table is red </s> </s>)= 0.250000
```

```
P(<s> <s> the black table </s> </s>)= 0.000000
```

4.4.4 Smoothing

Notice that the probability for the sentence "$< s > < s >$ the black table $< /s > < /s >$" is 0, as a result of the trigrams ($< s >$,the,black), (the,black,table), (black,table,</s>), and (table,</s>,</s>) not existing in our corpus. However, this is a valid sentence in the English language. We will apply Add-λ smoothing in order to address the issue of zero-valued probabilities for unknown trigrams.

$$P_{\text{Add-}\lambda}(w_n|w_{n-2}, w_{n-1}) = \frac{count(w_{n-2}, w_{n-1}, w_n) + \lambda}{\lambda|V| + count(w_{n-2}, w_{n-1})}$$

Let's now compute again the probability of the sentences "$< s > < s >$ the new table is red $< /s > < /s >$" and "$< s > < s >$ the black table $< /s > < /s >$" using the trigrams that we have computed and Add-λ smoothing for $\lambda = 0.001$.

```
words_processed = ['<s>','<s>','the', 'new', 'table', 'is', 'red','</s>','</s>','<s>','<s>','the',
    'blue', 'table', 'is', 'broken','</s>','</s>']
vocabulary = set() # Create an empty set
for word in words_processed: # Iterate through available words
   vocabulary.add(word) # Add word to set

V = len(vocabulary) # Get size of vocabulary

def pl_trig(trigram, trigram_frequencies, bigram_frequencies,l,V): # Create function to compute
    trigram probability with add-lambda smoothing
   return (trigram_frequencies[trigram] + l) / ( (l*V) + bigram_frequencies[(trigram[0],trigram[1])])

l = 0.001

pl_text1 = pl_trig(('<s>','<s>','the'),tgf,bgf,l,V)*pl_trig(('<s>','the','new'),tgf,bgf,l,V)*
pl_trig(('the','new','table'),tgf,bgf,l,V)*pl_trig(('new','table','is'),tgf,bgf,l,V)*
pl_trig(('table','is','red'),tgf,bgf,l,V)*pl_trig(('is','red','</s>'),tgf,bgf,l,V)*
pl_trig(('red','</s>','</s>'),tgf,bgf,l,V)

pl_text2 = pl_trig(('<s>','<s>','the'),tgf,bgf,l,V)*pl_trig(('<s>','the','black'),tgf,bgf,l,V)*
pl_trig(('the','black','table'),tgf,bgf,l,V)*pl_trig(('black','table','</s>'),tgf,bgf,l,V)*
pl_trig(('table','</s>','</s>'),tgf,bgf,l,V)

print("P(<s> <s> the new table is red </s> </s>)= %f" % pl_text1)
print("P(<s> <s> the black table </s> </s>)= %f" % pl_text2)
```

The output will look like:

```
P(<s> <s> the new table is red </s> </s>)= 0.239523
P(<s> <s> the black table </s> </s>)= 0.000001
```

4.5 The number underflow issue

Let's use again the unigram model that we computed in Section 4.2.3 to compute the probability for the sentence "the new table is the broken blue table".

```
text = "the new table is the broken blue table"
words_list = ['the','new','table','is','the','broken','blue','table']

print(pw,"\n")

p = 1
for word in words_list:
   p = p * pw[word]
   print("P(%s)=%f" % (word,pw[word]))

print("\nP(%s)=%f" % (text,p))
```

The output will look like:

```
defaultdict(<function <lambda> at 0x017334B0>, {'blue': 0.1, 'broken': 0.1, 'is': 0.2, 'new': 0.1,
    'red': 0.1, 'table': 0.2, 'the': 0.2, 'black': 0})

P(the)=0.200000
P(new)=0.100000
P(table)=0.200000
P(is)=0.200000
P(the)=0.200000
P(broken)=0.100000
P(blue)=0.100000
P(table)=0.200000

P(the new table is the broken blue table)=0.000000
```

As you can see above, the probability of sentence was computed as 0. But this is not correct. All the words in the sentence have a probability higher than 0. If you use a calculator to compute the probability $P(\text{the new table is the broken blue table}) = P(\text{the}) \cdot P(\text{new}) \cdot P(\text{table}) \cdot P(\text{is}) \cdot P(\text{the}) \cdot P(\text{broken}) \cdot P(\text{blue}) \cdot P(\text{table})$, the result will be 0.00000032. However, storing this number requires more precision that the float number type in Python supports and as a result it causes the number to underflow and return the value of 0. To avoid this problem, we typically compute probabilities in log space. Remember that in log space:

$$log\Big(P(A) \cdot P(B) \cdot P(C) \cdot ... \cdot P(Z)\Big) = log(P(A)) + log(P(B)) + log(P(C)) + ... + log(P(Z))$$

```python
import math # Import math library

text = "the new table is the broken blue table"
words_list = ['the','new','table','is','the','broken','blue','table']

logp = 0
for word in words_list:
    logp = logp + math.log(pw[word])
    print("log(P(%s))=%f" % (word,math.log(pw[word])))

print("\nlog(P(%s))=%f" % (text,logp))
```

The output will look like:

```
log(P(the))=-1.609438
log(P(new))=-2.302585
log(P(table))=-1.609438
log(P(is))=-1.609438
log(P(the))=-1.609438
log(P(broken))=-2.302585
log(P(blue))=-2.302585
log(P(table))=-1.609438

log(P(the new table is the broken blue table))=-14.954945
```

As long as all the probabilities are computed in log space, a higher log probability will denote a higher probability, since $a > b \implies log(a) > log(b)$. For example, from above: $P(\text{the}) > P(\text{new})$ $(0.2 > 0.1)$ and $log(P(\text{the})) > log(P(\text{new}))$ $(-1.609438 > -2.302585)$.

4.6 Exercises

Exercise 4.1 Create the function get_sentence_probability_unigram(words_list,unigram_frequencies), which given a sentence in the form of a list of words (words_list) and a dictionary with the frequencies of each unigram from a corpus (unigram_frequencies) will return the probability of the sentence based on the unigram language model. Use the function to compute the probability of the sentence "The passage in the castle" based on a unigram model trained on the dune.txt text. **Note:** Remember to address the number underflow issue.

Exercise 4.2 Use the carroll-alice.txt document from the NLTK Gutenberg corpus to train a unigram, a bigram, and a trigram model. For simplicity, do not use any tokens for the start and end of a sentence. Use these models to predict the next word in the following sentences:

(a) we went for a ___________

(b) the food was ___________

(c) the weather was ___________

(d) yesterday she had ___________

(e) she was ___________

(f) since yesterday ___________

Exercise 4.3 Create the function get_sentence_probability_bigram(words_list, unigram_frequencies, bigram_frequencies), which given a sentence in the form of a list of words (words_list), a dictionary with the frequencies of each unigram from a corpus (unigram_frequencies), and a dictionary with the frequencies of each bigram from a corpus (bigram_frequencies) will return the probability of the sentence based on the bigram language model. Use the function to compute the probability of the sentence "It was a warm night" based on a bigram model trained on the dune.txt text. **Note:** Remember to address the number underflow issue.

Workshop 5: Word embeddings

In this workshop we are going to create word embeddings using word-word co-occurrence matrices based on the context of each word.

5.1 Word context

To compute the word-word co-occurrence matrix we have to count the occurrences of each word from the vocabulary, within the context of each word. The context of a word can be set as a specific number of words prior and after the word, or the whole sentence, or the whole text, or even the whole corpus. Let's first compute the context of the word "i" in the text "*I like playing tennis. I enjoy sports. Do I enjoy tennis?*", in the form of a word list for a context size equal to one word before and one after the word "i".

5.1.1 Load text

Fist, let's tokenise the text to create the respective words list.

```
from nltk import word_tokenize # Import the word_tokenize function from NLTK
from string import punctuation

punctuation_list = list(punctuation) # Convert punctuation to a list

text = "I like playing tennis. I enjoy sports. Do I enjoy tennis?"

tokens = word_tokenize(text.lower()) # Tokenise "text" into words

words_list = []
for word in tokens:
    if(word not in punctuation_list):
        words_list.append(word)

print(text,"->",words_list)
```

The output will look like:

```
I like playing tennis. I enjoy sports. Do I enjoy tennis? -> ['i', 'like', 'playing', 'tennis', 'i',
    'enjoy', 'sports', 'do', 'i', 'enjoy', 'tennis']
```

5.1.2 Compute context words

Then let's compute the words within the context of the word "i":

```
context_size = 1
query_word = "i"
context = []
for i in range(len(words_list)): # Iterate through word list
    if(words_list[i] == query_word): # Check if word is the query word
        print("Found '%s' at position %.0f. Context:" % (query_word,i))
        for j in range(i-context_size,i+context_size+1): # Iterate through the context
            if( (j != i) and (j>=0) and (j<len(words_list)) ): # Ignore query word and non-valid word
    indexes
                context.append(words_list[j]) # Add word to context list
```

```
            print("[%.0f][%.0f] %s" % (i,j,words_list[j]))

print("\nContext of '%s' -> %s" % (query_word,context))
```

The output will look like:

```
Found 'i' at position 0. Context:
[0][1] like
Found 'i' at position 4. Context:
[4][3] tennis
[4][5] enjoy
Found 'i' at position 8. Context:
[8][7] do
[8][9] enjoy

Context of 'i' -> ['like', 'tennis', 'enjoy', 'do', 'enjoy']
```

Indeed, if you manually inspect the text, you will see that these four words appear within the context of the word "i" when the context is defined as the one previous word and the one after.

5.1.3 Other contexts

Let's now compute the words within the context of the words "i" and "enjoy" for a context size equal to n words prior and after each word, for $n = 1, 2, 3$. To avoid writing the same code multiple times, we will define a function for computing the context words.

```
def get_context(word, words_list,context_size):
    context = []
    for i in range(len(words_list)): # Iterate through word list
        if(words_list[i] == word): # Check if word is the query word
            for j in range(i-context_size,i+context_size+1): # Iterate through the context
                if( (j != i) and (j>=0) and (j<len(words_list)) ): # Ignore query word and non-valid
 word indexes
                    context.append(words_list[j]) # Add word to context list
    return context

print("\nContext (size=%.0f) of '%s' -> %s\n" % (1,"i",get_context("i", words_list,1)))
print("\nContext (size=%.0f) of '%s' -> %s\n" % (2,"i",get_context("i", words_list,2)))
print("\nContext (size=%.0f) of '%s' -> %s\n" % (3,"i",get_context("i", words_list,3)))
print("\nContext (size=%.0f) of '%s' -> %s\n" % (1,"enjoy",get_context("enjoy", words_list,1)))
print("\nContext (size=%.0f) of '%s' -> %s\n" % (2,"enjoy",get_context("enjoy", words_list,2)))
print("\nContext (size=%.0f) of '%s' -> %s\n" % (3,"enjoy",get_context("enjoy", words_list,3)))
```

The output will look like:

```
Context (size=1) of 'i' -> ['like', 'tennis', 'enjoy', 'do', 'enjoy']


Context (size=2) of 'i' -> ['like', 'playing', 'playing', 'tennis', 'enjoy', 'sports', 'sports',
    'do', 'enjoy', 'tennis']


Context (size=3) of 'i' -> ['like', 'playing', 'tennis', 'like', 'playing', 'tennis', 'enjoy',
    'sports', 'do', 'enjoy', 'sports', 'do', 'enjoy', 'tennis']


Context (size=1) of 'enjoy' -> ['i', 'sports', 'i', 'tennis']


Context (size=2) of 'enjoy' -> ['tennis', 'i', 'sports', 'do', 'do', 'i', 'tennis']


Context (size=3) of 'enjoy' -> ['playing', 'tennis', 'i', 'sports', 'do', 'i', 'sports', 'do', 'i',
    'tennis']
```

5.2 Word-word co-occurrence matrix

5.2.1 Word-word co-occurrence matrix (Context size = 1)

Let's now compute the word-word co-occurrence matrix for the text, for a context equal to one word before and one after the word.

```
vocabulary = set(words_list) # Create vocabulary of unique words
vocabulary = sorted(list(vocabulary)) # Convert vocabulary to list to preserve ordering and sort it
    for better presentation
print("Vocabulary:",vocabulary,"\n")

context_size = 1

print("%7s" % "", end='')
for word in vocabulary:
   print("\t%7s" % word, end='')
print("\n")

for word in vocabulary:
   print("%7s" % word,end='')
   context = get_context(word, words_list,context_size)
   for context_word in vocabulary:
      print("\t%7.0f" % context.count(context_word),end='') # Prints the number of times that
    context_word appears in the context list
   print("\n")
```

The output will look like:

```
Vocabulary: ['do', 'enjoy', 'i', 'like', 'playing', 'sports', 'tennis']

             do    enjoy       i     like  playing  sports  tennis

     do       0        0       1        0        0       1       0

  enjoy       0        0       2        0        0       1       1

      i       1        2       0        1        0       0       1

   like       0        0       1        0        1       0       0

playing       0        0       0        1        0       0       1

 sports       1        1       0        0        0       0       0

 tennis       0        1       1        0        1       0       0
```

Note that we used the set type to create a vocabulary of unique words but we then converted it to a list. Sets in python do not support ordering of their contents, neither have they indexes assigned to each of their elements.

5.2.2 Word-word co-occurrence matrix (Context size = 2)

Let's now compute the word-word co-occurrence matrix for the text, for a context equal to two words before and two after the word.

```
context_size = 2

print("%7s" % "", end='')
for word in vocabulary:
   print("\t%7s" % word, end='')
print("\n")

for word in vocabulary:
   print("%7s" % word,end='')
   context = get_context(word, words_list,context_size)
```

```
    for context_word in vocabulary:
        print("\t%7.0f" % context.count(context_word),end='')
    print("\n")
```

The output will look like:

```
              do    enjoy       i     like  playing  sports  tennis

     do        0        2       1        0        0       1       0

  enjoy        2        0       2        0        0       1       2

      i        1        2       0        1        2       2       2

   like        0        0       1        0        1       0       1

playing        0        0       2        1        0       0       1

 sports        1        1       2        0        0       0       0

 tennis        0        2       2        1        1       0       0
```

5.2.3 Compute word-word co-occurrence matrix as numpy array

Let's create a function that given a vocabulary, a text in the form of a word list, and the context size, will return a numpy array with the word-word co-occurrence matrix.

```
import numpy as np

def compute_word_word_matrix(vocabulary,words_list,context_size):
    word_word_matrix = np.zeros(( len(vocabulary),len(vocabulary) ), dtype=int) # Create empty array
     of size VxV
    for i in range(len(vocabulary)):
        context = get_context(vocabulary[i], words_list,context_size)
        for j in range(len(vocabulary)):
            word_word_matrix[i,j] = context.count(vocabulary[j])
    return word_word_matrix


context_size = 2

word_word_matrix = compute_word_word_matrix(vocabulary,words_list,context_size)

print(word_word_matrix)
```

The output will look like:

```
[[0 2 1 0 0 1 0]
 [2 0 2 0 0 1 2]
 [1 2 0 1 2 2 2]
 [0 0 1 0 1 0 1]
 [0 0 2 1 0 0 1]
 [1 1 2 0 0 0 0]
 [0 2 2 1 1 0 0]]
```

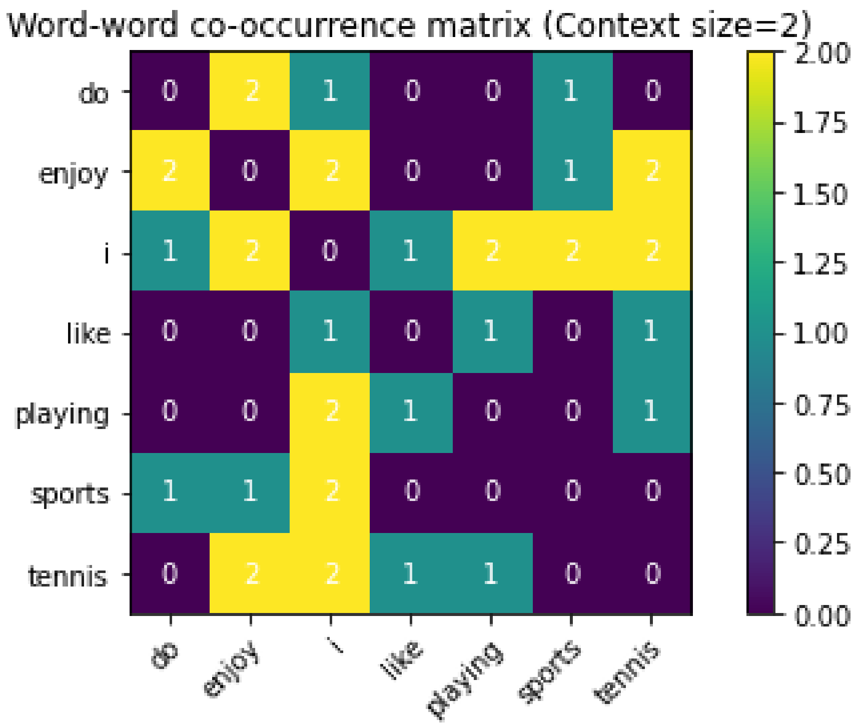
5.2.4 Word-word co-occurrence matrix visualisation

Let's visualise the word-word co-occurrence matrix as a heatmap.

```
import matplotlib
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
im = ax.imshow(word_word_matrix, cmap='viridis') # Create heatmap using the 'viridis' colour map
```

```
# Show all ticks
ax.set_xticks(np.arange(len(vocabulary)))
ax.set_yticks(np.arange(len(vocabulary)))
# Label ticks with the respective list entries
ax.set_xticklabels(vocabulary)
ax.set_yticklabels(vocabulary)

# Rotate the tick labels and set their alignment.
plt.setp(ax.get_xticklabels(), rotation=45, ha="right",rotation_mode="anchor")

# Loop over data dimensions and create text annotations.
for i in range(len(vocabulary)):
    for j in range(len(vocabulary)):
        text = ax.text(j, i, word_word_matrix[i, j], ha="center", va="center", color="w")

ax.set_title("Word-word co-occurrence matrix (Context size=%.0f)" % context_size)

plt.colorbar(im) # Add colour bar with colour range
plt.show() # Show plot
```

The output will look like:



5.3 Word embeddings

5.3.1 Word embeddings computation

We will use the word-word co-occurrence matrix that we have computed to create the word embeddings for the words in our text's vocabulary.

```
def get_word_embedding(word,word_word_matrix,vocabulary):
    word_index = vocabulary.index(word) # Gets word's index. Vocabulary must be of list type
    return word_word_matrix[word_index,:] # Return the word_index-th row of the word-word matrix

word_vectors = dict()
for word in vocabulary:
    word_vectors[word] = get_word_embedding(word,word_word_matrix,vocabulary)
```

```
    print(word,"->",get_word_embedding(word,word_word_matrix,vocabulary))

print("\n%s" % word_vectors)
```

The output will look like:

```
do -> [0 2 1 0 0 1 0]
enjoy -> [2 0 2 0 0 1 2]
i -> [1 2 0 1 2 2 2]
like -> [0 0 1 0 1 0 1]
playing -> [0 0 2 1 0 0 1]
sports -> [1 1 2 0 0 0 0]
tennis -> [0 2 2 1 1 0 0]

{'do': array([0, 2, 1, 0, 0, 1, 0]), 'enjoy': array([2, 0, 2, 0, 0, 1, 2]), 'i': array([1, 2, 0, 1,
    2, 2, 2]), 'like': array([0, 0, 1, 0, 1, 0, 1]), 'playing': array([0, 0, 2, 1, 0, 0, 1]),
    'sports': array([1, 1, 2, 0, 0, 0, 0]), 'tennis': array([0, 2, 2, 1, 1, 0, 0])}
```

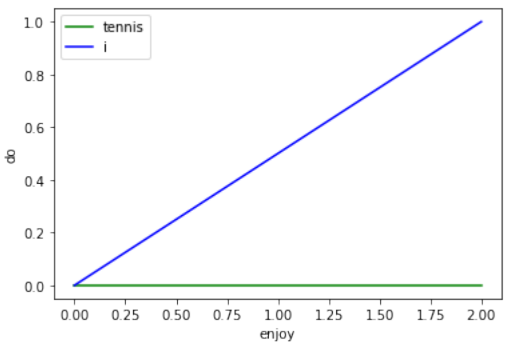
5.3.2 Word embeddings visualisation

We can visualise word embeddings as vectors in a V-dimensional space, where V is the size of the vocabulary. Let's visualise the vectors for the words "i" and "tennis" in the "enjoy" and "do" dimensions.

```
index_enjoy = vocabulary.index("enjoy") # Get index of word "enjoy" in vocabulary
index_do = vocabulary.index("do") # Get index of word "do" in vocabulary

# Create word embedding using only the values for the dimensions "enjoy" and "do"
embedding_i = word_vectors["i"][[index_enjoy,index_do]]
print("i ->",embedding_i)
embedding_tennis = word_vectors["tennis"][[index_enjoy,index_do]]
print("tennis ->",embedding_tennis,"\n")

fig = plt.subplots()
plt.plot([0,embedding_tennis[0]], [0,embedding_tennis[1]], 'g', label="tennis") # Plot line from
    (0,0) to the "tennis" coordinates
plt.plot([0,embedding_i[0]], [0,embedding_i[1]], 'b', label="i") # Plot line from (0,0) to the "i"
    coordinates
plt.xlabel('enjoy') # Set label for x axis
plt.ylabel('do') # Set label for y axis
plt.legend(loc="upper left") # Show plot legend at upper left location
plt.show() # Show plot
```

The output will look like:

```
i -> [2 1]
tennis -> [2 0]
```



5.3.3 Word embeddings distance

Let's now compute the pairwise cosine distance for all the words in the vocabulary using the word embeddings that we computed before.

```
from scipy.spatial import distance

print("Words cosine distance:")
for word1 in vocabulary:
    for word2 in vocabulary:
        print(word1,"->",word2,"=",distance.cosine(word_vectors[word1],word_vectors[word2]))
```

The output will look like:

```
Words cosine distance:
do -> do = 0.0
do -> enjoy = 0.6603168897566213
do -> i = 0.42264973081037427
do -> like = 0.7642977396044841
do -> playing = 0.6666666666666667
do -> sports = 0.33333333333333337
do -> tennis = 0.2254033307585167
enjoy -> do = 0.6603168897566213
enjoy -> enjoy = 0.0
enjoy -> i = 0.4770236396315093
enjoy -> like = 0.3594873847796515
enjoy -> playing = 0.32063377951324257
enjoy -> sports = 0.32063377951324257
enjoy -> tennis = 0.6491767922771884
i -> do = 0.42264973081037427
i -> enjoy = 0.4770236396315093
i -> i = 0.0
i -> like = 0.45566894604818275
i -> playing = 0.7113248654051871
i -> sports = 0.7113248654051871
i -> tennis = 0.47825080525004915
like -> do = 0.7642977396044841
like -> enjoy = 0.3594873847796515
like -> i = 0.45566894604818275
like -> like = 0.0
like -> playing = 0.2928932188134524
like -> sports = 0.5285954792089682
like -> tennis = 0.4522774424948339
playing -> do = 0.6666666666666667
playing -> enjoy = 0.32063377951324257
playing -> i = 0.7113248654051871
playing -> like = 0.2928932188134524
playing -> playing = 0.0
playing -> sports = 0.33333333333333337
playing -> tennis = 0.3545027756320972
sports -> do = 0.33333333333333337
sports -> enjoy = 0.32063377951324257
sports -> i = 0.7113248654051871
sports -> like = 0.5285954792089682
sports -> playing = 0.33333333333333337
sports -> sports = 0.0
sports -> tennis = 0.2254033307585167
tennis -> do = 0.2254033307585167
tennis -> enjoy = 0.6491767922771884
tennis -> i = 0.47825080525004915
tennis -> like = 0.4522774424948339
tennis -> playing = 0.3545027756320972
tennis -> sports = 0.2254033307585167
tennis -> tennis = 0.0
```

5.4 Exercises

Exercise 5.1 Create the word-word co-occurrence matrix for the dune.txt text for a context size equal to the 3 words prior and after a word. Visualise the word-word co-occurrence matrix as a heatmap.

Exercise 5.2 Use the word-word co-occurrence matrix that you computed in Exercise 5.1 in order to compute the respective word embeddings for all words in the vocabulary of dune.txt. Then compute the pairwise cosine distance between all words in the vocabulary and visualise them as a heatmap.

Exercise 5.3 Create the word-word co-occurrence matrix for the gatsby.txt document, compute the respective word embeddings, compute the pairwise cosine distance for all words in the vocabulary, and visualise these distances as a heatmap. Then, select the 5 dimensions (words) with the most counts and compute the word embeddings for the vocabulary using only these 5 dimensions. Consider the context as the 10 words prior and after a word.

Workshop 6: Document embeddings for machine learning

In this workshop, we are going to use word embeddings in order to create document embeddings that will be used for training machine learning models for the task of classification.

6.1 Loading data with pandas

Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation package, built on top of the Python programming language. You can find more information about the pandas Python package from here: `https://pandas.pydata.org/`

6.1.1 Load dataset file

Let's load the file testvectors.csv, which contains the word embeddings (of size 300) for a subset of the words from a 2017 dump of the English Wikipedia. The embeddings were created using the skipgram model and are stored in a comma-separated file, where each row contains the embedding of a word in 301 columns. The first column contains the word and the rest 300 columns the respective embedding.

We will use pandas to load the testvectors.csv file and store its content to a pandas dataframe object.

```
import pandas as pd # Import the pandas library

df = pd.read_csv('testvectors.csv', header=None) # Read csv file. Indicate that there is no row with
    column titles
print( df.head(5), "\n") # Print the first 5 rows of the dataframe

count_row = df.shape[0] # Gives number of rows
count_col = df.shape[1] # Gives number of columns

print("\nTotal words:",count_row)
print("Dimensions:",count_col-1) # Subtract 1 for the word column
```

The output will look like:

```
      0         1         2         3         4         5         6   \
0    one  0.073525 -0.031703  0.054010 -0.040015 -0.011894  0.002958
1   time  0.061892  0.066106  0.026482 -0.122901  0.016603  0.024152
2  would -0.005455 -0.064055  0.106359  0.000271 -0.005658  0.017313
3   made -0.033224 -0.046773 -0.022644 -0.115277 -0.037984  0.109500
4   well  0.032183  0.058166  0.102063 -0.054714 -0.037364  0.013909

        7         8         9    ...       291       292       293       294  \
0 -0.065406  0.079166  0.099932  ...  0.048875  0.014754 -0.038729  0.033155
1 -0.057542  0.119501  0.033247  ... -0.043285  0.045578 -0.139174  0.109938
2 -0.030339  0.014298  0.030399  ... -0.029693  0.083669 -0.089212  0.063672
3 -0.138488  0.038567  0.001758  ...  0.005144  0.002312 -0.043491  0.091255
4  0.025827  0.072022 -0.050159  ...  0.017727  0.056166 -0.106694  0.046184

        295       296       297       298       299       300
0 -0.120390 -0.065746 -0.023745  0.012824  0.005162 -0.130008
```

```
1 -0.057847 0.010336 0.114048 0.042011 0.032165 0.051062
2 -0.048096 0.028711 0.032499 0.104135 0.001100 0.024307
3 -0.065307 -0.060637 -0.029168 0.034441 0.024149 0.003539
4 -0.091665 -0.088526 0.016668 0.061213 -0.015307 -0.076759

[5 rows x 301 columns]


Total words: 155
Dimensions: 300
```

6.1.2 Available words in dataset

Let's create a list with the available words from the word embeddings dataset from testvectors.csv.

```
available_words = df[0].tolist() # Convert the first column of the dataframe to a list and store to a
    variable

print(available_words)
```

The output will look like:

```
['one', 'time', 'would', 'made', 'well', 'family', 'use', 'took', 'could', 'home', 'served', 'large',
    'like', 'day', 'final', 'near', 'much', 'book', 'came', 'late', 'side', 'started', 'way', 'take',
    'without', 'old', 'making', 'field', 'never', 'across', 'see', 'features', 'seen', 'mother',
    'either', 'get', 'close', 'reached', 'white', 'change', 'female', 'beginning', 'allowed',
    'night', 'week', 'natural', 'ran', 'thought', 'woman', 'room', 'nearly', 'sister', 'acquired',
    'whether', 'ancient', 'actually', 'feet', 'bank', 'floor', 'occurred', 'stone', 'twice', 'visit',
    'say', 'quite', 'castle', 'think', 'pop', 'shape', 'getting', 'reading', 'nothing', 'boy',
    'standing', 'mind', 'ahead', 'weather', 'let', 'door', 'feel', 'step', 'eyes', 'hot', 'hair',
    'moment', 'worth', 'afterwards', 'departure', 'shall', 'lay', 'passage', 'watch', 'looked',
    'seemed', 'bed', 'sitting', 'pictures', 'feeling', 'hear', 'generations', 'trouble', 'warm',
    'suddenly', 'considering', 'burning', 'remarkable', 'bore', 'pink', 'hanging', 'pleasure',
    'shadow', 'peer', 'darkness', 'picking', 'tired', 'lamp', 'witch', 'conversations', 'ought',
    'pile', 'rabbit', 'hedge', 'curiosity', 'stupid', 'jewels', 'dear', 'twenty-six', 'vaulted',
    'awakened', 'wondered', 'bulky', 'frenzy', 'hooded', 'hurried', 'oh', 'fortunately',
    'unbearable', 'sleepy', 'flashed', 'glittering', 'in', 'by', 'matted', 'the', 'dimmed', 'of',
    'and', 'daisies', 'a', 'to', 'scurrying', 'for', 'crone', 'on', 'is']
```

6.1.3 Access specific row in pandas dataframe

Let's now attempt to access a specific row from the pandas dataframe that we have stored our embeddings in. To access the first row from the dataframe we have to do the following:

```
vector = df.iloc[0] # Access the 0-th row of the dataframe
print(vector)
```

The output will look like:

```
0           one
1      0.073525
2     -0.031703
3       0.05401
4     -0.040015
         ...
296   -0.065746
297   -0.023745
298    0.012824
299    0.005162
300   -0.130008
Name: 0, Length: 301, dtype: object
```

6.1.4 Access specific columns in pandas dataframe

Let's now attempt to access only specific columns from a row in the pandas dataframe. The first column of each row (column 0) contains the word, while the respective embedding is stored in columns 1-301. To access from column 1 to column 5 from the first row in the dataframe and from column 1 to 301, we have to do the following:

```
vector = df.iloc[0,1:5] # Access columns 1 to 5 of the 0-th row of the dataframe
print(vector)

vector = df.iloc[0,1:301] # Access columns 1 to 301 of the 0-th row of the dataframe
print(vector)
```

The output will look like:

```
1    0.073525
2   -0.031703
3     0.05401
4   -0.040015
Name: 0, dtype: object
1      0.073525
2     -0.031703
3       0.05401
4     -0.040015
5     -0.011894
         ...
296   -0.065746
297   -0.023745
298    0.012824
299    0.005162
300   -0.130008
Name: 0, Length: 300, dtype: object
```

Remember that similar to lists and array types in Python, indexing in pandas dataframes starts from 0. For example, the first row has an index equal to 0, the second an index equal to 1, and the n-th and index equal to n-1.

We now have stored the embedding for the first word in the dataset to the variable "vector".

6.1.5 Convert dataframe to numpy array

The majority of analysis and processing functions require numerical input to be of numpy array type. Let's convert the word embedding that we stored above in the variable "vector" to a numpy array of similar length.

```
import numpy as np

word_embedding = vector.to_numpy(dtype=float)

print(word_embedding)
```

The output will look like:

```
[ 0.073525 -0.031703 0.05401 -0.040015 -0.011894 0.002958 -0.065406
  0.079166 0.099932 -0.134534 0.016396 0.056965 -0.057572 0.010251
  0.050909 0.072398 -0.061399 0.162734 0.082844 -0.052345 0.019466
  0.058028 0.010818 0.030207 0.000889 -0.021392 0.030583 0.038453
 -0.165726 0.112191 -0.058096 -0.111807 -0.033361 0.094921 0.013087
 -0.010414 0.073115 -0.058642 0.099049 0.07455 0.035721 -0.059482
  0.06792 -0.058917 0.004033 0.040625 -0.106188 0.088392 0.009051
  0.019659 -0.026113 -0.09077 0.043568 -0.062696 -0.030408 0.03884
  0.004035 0.013125 0.02305 -0.041966 -0.025127 -0.034154 0.016506
 -0.079488 -0.039066 0.013653 -0.063375 0.091463 0.05381 0.024699
  0.062898 0.12055 -0.048261 -0.017775 0.029978 0.022461 0.094113
 -0.098662 0.011137 -0.080405 -0.056159 -0.055153 -0.0306 0.108031
  0.082347 0.058076 -0.113915 0.003602 0.019997 -0.033987 0.027162
  0.007531 -0.019095 -0.048414 -0.024189 0.04996 0.032027 -0.009882
```

```
 0.045991 0.055946 -0.028849 -0.019265 -0.041771 -0.06572 -0.024059
-0.002533 0.02972  0.041398 0.123978 0.050968 0.072876 -0.091089
-0.011735 -0.014177 0.058902 -0.145468 -0.12125 0.084168 -0.155788
-0.009348 -0.015042 0.063526 0.039869 0.025571 0.044058 0.019486
 0.112362 0.031733 0.039299 -0.051543 -0.022537 -0.026686 -0.100046
 0.115309 0.008369 0.01551 -0.065277 0.031222 0.109851 -0.006308
-0.016031 -0.038418 0.034439 0.025142 0.142227 0.04277 -0.01852
-0.005247 -0.021285 -0.019829 0.131366 -0.002935 0.018499 0.040565
-0.03535 -0.075773 -0.017759 0.033599 0.023961 -0.106251 -0.040328
-0.012546 0.006421 -0.082573 -0.031654 -0.010218 0.053183 0.068255
-0.027139 -0.062169 0.043021 0.027036 -0.006469 -0.142859 0.022744
 0.000512 -0.065334 -0.052299 -0.017929 0.03619 0.030412 0.022339
 0.080582 -0.007923 -0.006414 -0.024119 -0.039354 -0.00177 0.01856
 0.079291 -0.037962 0.004094 0.057353 -0.126054 0.039407 -0.047057
 0.028695 -0.041185 -0.042427 0.063292 -0.015259 -0.012919 0.029772
 0.001388 -0.046082 0.112506 -0.004109 0.020585 0.018128 -0.025253
 0.016204 0.035294 -0.042431 -0.014868 -0.141065 -0.073506 -0.021315
 0.067625 0.073685 0.023866 -0.010576 -0.042903 -0.071802 -0.071728
 0.019136 -0.087325 -0.042621 -0.064981 -0.013045 0.039378 -0.029022
-0.054649 -0.008433 0.03112 -0.018196 -0.003567 0.021799 0.094146
-0.017547 0.036818 0.012625 0.053266 -0.078154 -0.069845 0.019453
 0.047343 -0.033813 -0.001188 0.04178 0.010779 0.005534 0.010311
 0.093365 -0.010763 0.040849 0.021107 0.047443 -0.017278 -0.068958
 0.033909 -0.011073 0.067898 -0.00054 -0.012711 -0.042463 -0.015018
 0.051556 -0.023581 -0.064241 0.026324 -0.039002 -0.013781 -0.060861
-0.048069 -0.02534 0.079009 0.102468 0.044573 -0.041997 0.081503
-0.010224 -0.028924 0.008187 -0.062565 0.076221 -0.039867 0.029696
 0.094235 -0.072826 0.038318 0.048875 0.014754 -0.038729 0.033155
-0.12039 -0.065746 -0.023745 0.012824 0.005162 -0.130008]
```

Note that when converting the pandas dataframe to a numpy array using the to_numpy() function, we indicated the type of numbers to be loaded. In this case, the numbers in the dataset are of float type. If for example the numbers were integers, we would have passed the "dtype=int" argument to the to_numpy() function.

6.2 Word embeddings

6.2.1 Load word embeddings

Let's now create a dictionary that will contain all the word embeddings available in the dataset in numpy array form. To test the dictionary, we will print the first 7 elements of the embeddings for the words "mother" and "boy".

```python
word_embeddings = dict()
for i in range(count_row): # Iterate through all rows in dataframe (words)
    word = df.iloc[i,0] # Get word
    embedding = df.iloc[i,1:count_col].to_numpy(dtype=float) # Get embedding and convert to float
     numpy array
    word_embeddings[word] = embedding

print("mother ->",word_embeddings["mother"][0:7]) # Print the first 7 elements of the embedding for
     word "mother"
print("boy   ->",word_embeddings["boy"][0:7]) # Print the first 7 elements of the embedding for word
     "boy"
```

The output will look like:

```
mother -> [ 0.007815 0.026617 -0.036383 -0.051246 0.000183 0.071259 0.017416]
boy    -> [ 0.040253 0.001048 0.023576 0.003103 -0.027837 0.035486 0.04829 ]
```

6.2.2 Distance of word embeddings

Let's now compute and print a matrix with the pairwise cosine distances of the embeddings of the words "mother", "boy", "sister", "family", "home", "rabbit", and "eyes".

```
from scipy.spatial import distance

test_words = ["mother","boy","sister","family","home","rabbit","eyes"]

print("%6s" % "", end="")
for word in test_words:
    print("\t%6s" % word, end="")
print("")
for word1 in test_words:
    print("%6s" % word1, end="")
    for word2 in test_words:
        print("\t%1.4f" % distance.cosine(word_embeddings[word1],word_embeddings[word2]), end="")
    print("")
```

The output will look like:

```
        mother     boy  sister  family    home  rabbit    eyes
mother  0.0000  0.4318  0.3882  0.4893  0.7175  0.7303  0.7059
   boy  0.4318  0.0000  0.6213  0.7388  0.8153  0.6080  0.6594
sister  0.3882  0.6213  0.0000  0.6224  0.7767  0.8759  0.8403
family  0.4893  0.7388  0.6224  0.0000  0.7678  0.8327  0.8262
  home  0.7175  0.8153  0.7767  0.7678  0.0000  0.8341  0.8851
rabbit  0.7303  0.6080  0.8759  0.8327  0.8341  0.0000  0.6599
  eyes  0.7059  0.6594  0.8403  0.8262  0.8851  0.6599  0.0000
```

Notice that the words “mother”, “sister”, “boy” and “family” have smaller distances among each other, compared to the other words. Considering that they have contextual relations, it is an indication that these word embeddings are able to encode contextual information about the words.

6.3 Document embeddings

Let’s now use the available word embeddings in order to create the document embeddings for the following documents:

1. My mother was sitting on the bed
2. The night looked remarkable at the beginning

There are various ways to create document embeddings. One of the most common approaches is to compute the embeddings for each word in a document and then compute the average embedding (applied element-wise) across the embeddings of its constituent words.

We will first tokenise the two documents:

```
from nltk import word_tokenize # Import the word_tokenize function from NLTK

text1 = "My mother was sitting on the bed"
text2 = "The night looked remarkable at the beginning"

tokens1 = word_tokenize(text1.lower()) # Tokenise "text1" into words
tokens2 = word_tokenize(text2.lower()) # Tokenise "text1" into words

words_list1 = []
for word in tokens1:
    words_list1.append(word)

print(text1,"->",words_list1)

words_list2 = []
for word in tokens2:
    words_list2.append(word)

print(text2,"->",words_list2)
```

The output will look like:

```
My mother was sitting on the bed -> ['my', 'mother', 'was', 'sitting', 'on', 'the', 'bed']
The night looked remarkable at the beginning -> ['the', 'night', 'looked', 'remarkable', 'at', 'the',
    'beginning']
```

We will now use the dictionary of the word embeddings that we created before in order to retrieve the embedding for each word in the two documents.

```
print("Text1 word embeddings:")
for word in words_list1:
    try:
        print(word,"->",word_embeddings[word][0:4]) # Print first 4 elements of embedding
    except:
        print(word,"-> n/a")

print("\nText2 word embeddings:")
for word in words_list2:
    try:
        print(word,"->",word_embeddings[word][0:4]) # Print first 4 elements of embedding
    except:
        print(word,"-> n/a")
```

The output will look like:

```
Text1 word embeddings:
my -> n/a
mother -> [ 0.007815 0.026617 -0.036383 -0.051246]
was -> n/a
sitting -> [9.7000e-05 4.1846e-02 8.3230e-02 5.5060e-03]
on -> [ 0.005252 -0.002234 -0.0648 -0.001852]
the -> [ 0.016258 -0.013271 -0.007168 -0.083179]
bed -> [ 0.0882  0.056767 -0.021443 0.014364]

Text2 word embeddings:
the -> [ 0.016258 -0.013271 -0.007168 -0.083179]
night -> [ 0.06248 -0.051441 -0.023803 0.038181]
looked -> [-0.011535 0.016638 0.063261 -0.033455]
remarkable -> [ 0.065266 -0.021301 0.060876 0.068074]
at -> n/a
the -> [ 0.016258 -0.013271 -0.007168 -0.083179]
beginning -> [-0.017209 0.131408 0.024707 -0.036714]
```

As you can see above, some of the words from the two documents are missing from the word embeddings dataset that we have. In this case, we will ignore the words with the missing embeddings and process each document as if these words do not exist.

6.3.1 Compute document embedding (Mean word embedding)

Let's now create a function that given a document in the form of a list of words, a dictionary of word embeddings and the size k of the requested document embedding, will return a document embedding of size k, computed as the first k elements of the mean word embedding of the document's constituent words. Then, we will print the embedding of the two documents for $k = 5$ and $k = 20$.

```
def get_document_embedding(word_list,word_embeddings,k):
    document_embedding = np.zeros(k,dtype=float) # Create embedding of k zero-valued elements
    valid_words = 0
    for word in word_list:
        try:
            document_embedding = document_embedding + word_embeddings[word][0:k] # Add word embedding
     to partial sum
            valid_words += 1
        except:
            pass # If word embedding is not available, then ignore the word
    document_embedding = document_embedding / valid_words # Divide all elements by number of valid
     words to get the mean
```

```
    return document_embedding

print("Text1 embedding (k=5) ->",get_document_embedding(words_list1,word_embeddings,5),"\n")
print("Text2 embedding (k=5) ->",get_document_embedding(words_list2,word_embeddings,5),"\n")

print("Text1 embedding (k=20) ->",get_document_embedding(words_list1,word_embeddings,20),"\n")
print("Text2 embedding (k=20) ->",get_document_embedding(words_list2,word_embeddings,20),"\n")
```

The output will look like:

```
Text1 embedding (k=5) -> [ 0.0235244 0.021945 -0.0093128 -0.0232814 0.0077766]

Text2 embedding (k=5) -> [ 0.02191967 0.008127 0.01845083 -0.021712 -0.008242 ]

Text1 embedding (k=20) -> [ 0.0235244 0.021945 -0.0093128 -0.0232814 0.0077766 0.0664142
 -0.0294452 0.0385194 0.0415822 0.0027638 0.038138 -0.0464678
  0.0299128 -0.0162724 0.0267094 -0.0193898 -0.0579416 0.0045326
  0.0272054 0.002882 ]

Text2 embedding (k=20) -> [ 0.02191967 0.008127 0.01845083 -0.021712 -0.008242 0.05405583
 -0.00840717 0.07847117 0.017932  -0.01513817 0.035567  -0.04270783
  0.05314033 -0.03148433 -0.0126355 -0.016384 -0.08776767 0.01809233
  0.06723833 -0.00525483]
```

6.3.2 Cosine distance of document embeddings

Let's now compute the cosine distance between the document embeddings of the two documents for $k = 5, 20, 15, 300$.

```
print("Cosine distance of text1 and text2 for k =
    5:",distance.cosine(get_document_embedding(words_list1,word_embeddings,5),
    get_document_embedding(words_list2,word_embeddings,5)))
print("Cosine distance of text1 and text2 for k =
    20:",distance.cosine(get_document_embedding(words_list1,word_embeddings,20),
    get_document_embedding(words_list2,word_embeddings,20)))
print("Cosine distance of text1 and text2 for k =
    150:",distance.cosine(get_document_embedding(words_list1,word_embeddings,150),
    get_document_embedding(words_list2,word_embeddings,150)))
print("Cosine distance of text1 and text2 for k =
    300:",distance.cosine(get_document_embedding(words_list1,word_embeddings,300),g
    et_document_embedding(words_list2,word_embeddings,300)))
```

The output will look like:

```
Cosine distance of text1 and text2 for k = 5: 0.3855626793684508
Cosine distance of text1 and text2 for k = 20: 0.16427167205198012
Cosine distance of text1 and text2 for k = 150: 0.28613327209110107
Cosine distance of text1 and text2 for k = 300: 0.3327962278223764
```

6.4 Classification using document embeddings

Let's now use document embeddings based on our word embeddings dataset to classify movie reviews as having positive or negative sentiment. We will first load the movie reviews dataset, which consists of 20 files, 10 files named pos_XX.txt that each contains one movie review with positive sentiment, and 10 files named neg_XX.txt that each contains one movie review with negative sentiment. XX is an identification number from 01 to 10.

```
text = []
label = []

for i in range(1,11):
    filename_pos = "pos_%02d.txt" % i # Create string with the filename for positive sentiment reviews
    filename_neg = "neg_%02d.txt" % i # Create string with the filename for negative sentiment reviews
```

```
    print(filename_pos)
    print(filename_neg)
    # Open positive sentiment file
    f = open(filename_pos, "r") # Opens the file for reading only ("r")
    text.append(f.read())
    f.close() # Close the file
    label.append("pos") # Add positive sentiment label to labels list
    # Open negative sentiment file
    f = open(filename_neg, "r") # Opens the file for reading only ("r")
    text.append(f.read())
    f.close() # Close the file
    label.append("neg") # Add negative sentiment label to labels list

print("No of texts:",len(text))
print("No of labels:",len(label))
```

The output will look like:

```
pos_01.txt
neg_01.txt
pos_02.txt
neg_02.txt
pos_03.txt
neg_03.txt
pos_04.txt
neg_04.txt
pos_05.txt
neg_05.txt
pos_06.txt
neg_06.txt
pos_07.txt
neg_07.txt
pos_08.txt
neg_08.txt
pos_09.txt
neg_09.txt
pos_10.txt
neg_10.txt
No of texts: 20
No of labels: 20
```

Now let's compute the document embedding for each document, using the function that we created before.

```
from nltk import word_tokenize
from string import punctuation

punctuation_list = list(punctuation)

text_embeddings = []
for i in range(len(text)): # Iterate through all texts
    tokens = word_tokenize(text[i].lower()) # Tokenise "text" into words
    words_list = []
    for word in tokens:
        if(word not in punctuation_list):
            words_list.append(word)
    text_embeddings.append(get_document_embedding(words_list,word_embeddings,300))

for i in range(len(text)): # Iterate through all texts
    print(i,text_embeddings[i][0:5],"->",label[i]) # Print the first 5 elements of each document
     embedding
```

The output will look like:

```
0 [-0.00144049 -0.01854695 0.01171497 -0.05752111 -0.03104189] -> pos
1 [-0.00351737 -0.02884196 0.00754981 -0.03615796 -0.028105 ] -> neg
```

```
2 [ 0.01211377 -0.01454308 0.01602908 -0.06364362 -0.02417946] -> pos
3 [ 0.0013629 -0.02655979 0.00252164 -0.05645312 -0.02949102] -> neg
4 [-0.025445  -0.03421304 0.012184  -0.04374732 -0.0253342 ] -> pos
5 [-0.0044761 -0.0279336 0.0044961 -0.04975507 -0.0246164 ] -> neg
6 [ 0.01035458 -0.02045229 0.00215542 -0.0624085 -0.04229192] -> pos
7 [-0.00919744 -0.02292037 0.01568922 -0.0474781 -0.02196531] -> neg
8 [-0.01577481 -0.02448513 0.00704729 -0.04966781 -0.02111129] -> pos
9 [ 0.00515638 -0.01661385 0.00617285 -0.04173085 -0.04571008] -> neg
10 [ 0.00058903 -0.01620987 0.0142112 -0.0417965 -0.0315724 ] -> pos
11 [ 0.0067564 -0.01518371 0.00774966 -0.04712666 -0.03590557] -> neg
12 [-0.00779024 -0.02070952 0.00332203 -0.05560338 -0.0289061 ] -> pos
13 [-0.00285736 -0.0158494 0.008223 -0.05303438 -0.02770218] -> neg
14 [-0.011991  -0.02553873 -0.01264482 -0.05226082 -0.02133141] -> pos
15 [-0.01091673 -0.02792297 0.00905532 -0.0435513 -0.03137696] -> neg
16 [-0.00434465 -0.01706052 -0.00136974 -0.04984022 -0.03162339] -> pos
17 [-0.01131559 -0.01907496 -0.00221819 -0.05785556 -0.01828148] -> neg
18 [ 0.00109655 -0.02176745 0.00632607 -0.0452914 -0.03907748] -> pos
19 [ 0.00515789 -0.01992803 0.02873918 -0.04443174 -0.01608311] -> neg
```

Note that the sentiment labels are of type string. Many machine learning packages and functions require the labels to be in numerical format. To achieve this, we will encode our sentiment labels to numbers.

```
from sklearn import preprocessing

le = preprocessing.LabelEncoder() # Create labelEncoder

labels_encoded=le.fit_transform(label) # Encode labels to numbers
print(label,"->",labels_encoded)
```

The output will look like:

```
['pos', 'neg', 'pos', 'neg', 'pos', 'neg', 'pos', 'neg', 'pos', 'neg', 'pos', 'neg', 'pos', 'neg',
    'pos', 'neg', 'pos', 'neg', 'pos', 'neg'] -> [1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0]
```

As you can see above, the negative ("neg") label is now denoted by 0 and the positive ("pos") label by 1.

6.4.1 Classification with the k Nearest Neighbour algorithm (kNN)

Let's use the k Nearest Neighbours algorithm for 3 nearest neighbours in order to classify the movie reviews in positive or negative. We will first divide our dataset into a training set consisting of 14 samples (7 negative and 7 positive) and a test set consisting of 6 samples (3 negative and 3 positive). We will then train the kNN model on the training set and compute the classification performance on the test set.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score

# Divide dataset to training (14 samples - 7 positive,7 negative) and test (6 samples - 3 positive, 3
    negative)
training_features = text_embeddings[0:14]
training_labels = labels_encoded[0:14]
test_features = text_embeddings[14:20]
test_labels = labels_encoded[14:20]


model = KNeighborsClassifier(n_neighbors=3)

# Train the model using the training set
model.fit(training_features,training_labels)

#Predict Output
predicted= model.predict(test_features)
print("Prediction :",predicted)
print("True labels:",test_labels)
```

```
cm = confusion_matrix(test_labels, predicted) # Create confusion matrix
accuracy = accuracy_score(test_labels, predicted) # Compute classification accuracy

print("Confusion matrix:\n%s" % cm)
print("Accuracy: %.2f%s" % (accuracy*100,"%"))
```

The output will look like:

```
Prediction : [1 0 1 1 1 1]
True labels: [1 0 1 0 1 0]
Confusion matrix:
[[1 2]
 [0 3]]
Accuracy: 66.67%
```

As you can see above, the kNN classifier for 3 nearest neighbours was able to classify correctly 4 out of the 6 samples in our test set, reaching a classification accuracy of 66.67%.

6.4.2 Classification with Linear Support Vector Machines (SVM)

Let's repeat our experiment using a Linear Support Vector Machine (SVM) model.

```
#Import svm model
from sklearn import svm

#Create a svm Classifier
clf = svm.SVC(kernel='linear') # Linear Kernel

#Train the model using the training set
clf.fit(training_features,training_labels)

#Predict the response for test dataset
predicted = clf.predict(test_features)
print("Prediction :",predicted)
print("True labels:",test_labels)

cm = confusion_matrix(test_labels, predicted)
accuracy = accuracy_score(test_labels, predicted)

print("Confusion matrix:\n%s" % cm)
print("Accuracy: %.2f%s" % (accuracy*100,"%"))
```

The output will look like:

```
Prediction : [1 0 1 1 0 0]
True labels: [1 0 1 0 1 0]
Confusion matrix:
[[2 1]
 [1 2]]
Accuracy: 66.67%
```

As you can see above, the SVM classifier achieved the same classification accuracy as the kNN classifier. However, note that it miss-classified one sample from each class, whereas the kNN classifier miss-classified two samples from the same class. Remember that metrics such as the classification accuracy are not sufficient to provide a complete overview of a machine learning model's performance.

6.4.3 Classification using Feed-Forward Neural Networks

Let's now use a Feed-Forward Neural Network for the same task. We will define a neural network that has an input layer of size 300 (300 being the size of the embedding), one hidden layer with 5 neurons and an output layer of 2 neurons (2 being the number of classes). A ReLU activation function will be used for the hidden layer, while a softmax activation function will be used for the output layer in order to convert the network's output to class probabilities.

```
import tensorflow as tf
```

```
EMBEDDING_SIZE = 300

model = tf.keras.models.Sequential() # Create a sequential neural network model
model.add(tf.keras.Input(shape=(EMBEDDING_SIZE,))) # The input size will be equal to the embedding
    size
model.add(tf.keras.layers.Dense(5, activation='relu')) # 1st hidden layer has a size of 5 neurons
model.add(tf.keras.layers.Dense(2, activation='softmax')) # Output layer has a size of 2 neurons

model.summary() # Print a summary of the model
```

The output will look like:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 5)                 1505

 dense_1 (Dense)             (None, 2)                 12

=================================================================
Total params: 1,517
Trainable params: 1,517
Non-trainable params: 0
_________________________________________________________________
```

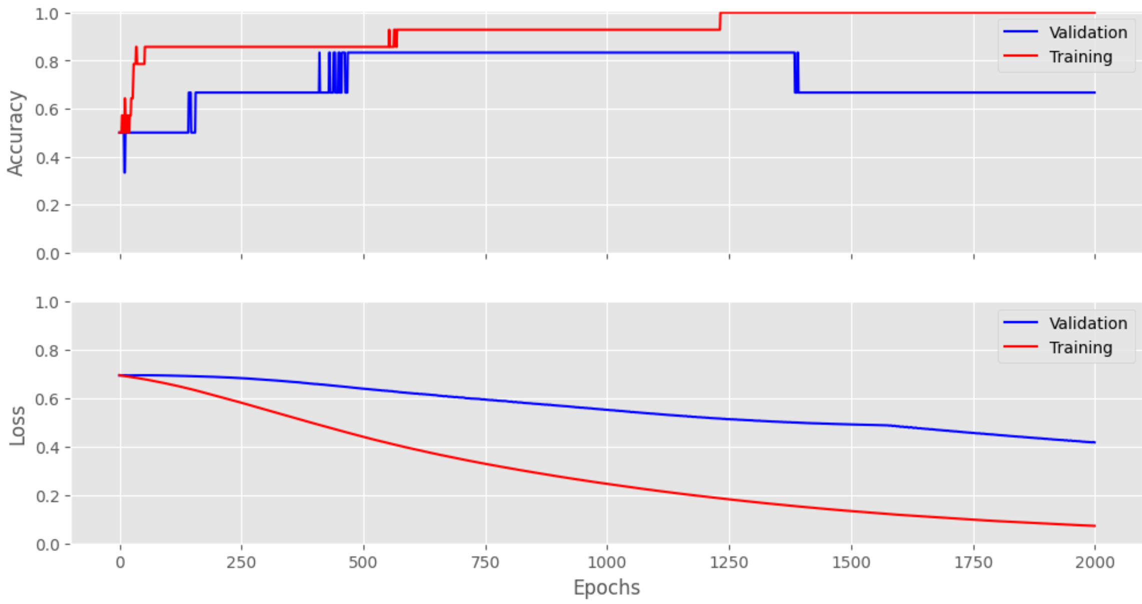
Now let's train the network for 2000 epochs, using the Adam optimiser, a learning rate of $\eta = 0.001$, and cross entropy loss as the loss function.

```
import matplotlib.pyplot as plt

# Define Adam optimiser and set learning rate to 0.001
opt = tf.keras.optimizers.Adam(learning_rate=0.001)

# Compile model using sparse categorical cross entropy loss as loss function and the Adam optimiser
model.compile(optimizer=opt, loss='sparse_categorical_crossentropy', metrics='accuracy')

EPOCHS = 2000 # Train model for 2000 epochs

# Convert data to numpy arrays
X_train = np.array(training_features)
y_train = np.array(training_labels)
X_test = np.array(test_features)
y_test = np.array(test_labels)

# Train model and store the loss and the metrics for each epoch in the history variable
history = model.fit(x=X_train, y=y_train, validation_data=(X_test,y_test),epochs=EPOCHS,verbose=0)

# Plot training progress
plt.style.use('ggplot')
fig, (ax1, ax2) = plt.subplots(2, figsize=(12, 6), sharex=True)

ax1.set_ylim([0, 1.01])
ax1.plot(history.history['val_accuracy'],'b')
ax1.set_ylabel("Accuracy")
ax1.plot(history.history['accuracy'],'r')
ax1.legend(['Validation','Training'])

ax2.set_ylim([0, 1])
ax2.plot(history.history['val_loss'],'b')
ax2.plot(history.history['loss'],'r')
ax2.set_ylabel("Loss")
ax2.set_xlabel("Epochs")
ax2.legend(['Validation','Training'])
```

```
print("Max validation accuracy: %.2f%s" % (max(history.history['val_accuracy'])*100,"%"))

for i in range(EPOCHS):
    if(history.history['val_accuracy'][i] == max(history.history['val_accuracy'])):
        print("Max accuracy reached at epoch",i)
        break
```

The output will look like:

```
Max validation accuracy: 83.33%
Max accuracy reached at epoch 410
```



Note that the maximum accuracy and the epoch that it was reached may differ a bit every time your rerun the code due to the random initialisation of the neural network's weights.

6.5 Exercises

Exercise 6.1 Using the word embeddings from testvectors.csv, compute the word embeddings of size k for each word in the text "Once upon a time, my family was living in an ancient castle.". Then compute the pairwise cosine distance between the words in the text and visualise them as a heatmap. Repeat this for $k = 5, 50, 300$. Ignore words for which an embedding is not available.

Exercise 6.2 Repeat the classification task from Section 6.4.1 for an embedding of size $k = 50$ and an embedding of size $k = 200$.

Exercise 6.3 Improve the feed forward neural network's architecture from Section 6.4.3 in order to make it reach the maximum accuracy earlier (in less epochs). Note: Try to experiment with adding more layers, changing the size of the layers, using different activation functions for the hidden layers, training for more epochs, changing the learning rate, etc.

Workshop 7: Text Classification Using Traditional Classifiers

In this workshop, we are going to use traditional machine learning algorithms for the task of text classification and more specifically for the task of classifying emails as “spam” or “ham” (not spam).

7.1 Introduction to Python classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state. Compared with other programming languages, Python’s class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

More information about Python classes here: `https://docs.python.org/3/tutorial/classes.html`

7.1.1 A simple Python class

Let’s define a class that describes the number five (5). It will contain a numerical variable equal to 5 and a string variable equal to “five”.

```
class Five:
    value = 5
    name = "five"

no = Five()

print("Value:",no.value)
print("Name:",no.name)
```

The output will look like:

```
Value: 5
Name: five
```

As you can see above, we create a new object of type “Five” and stored it in variable “no”. We can now access the contents of the object using the variable name, followed by a dot and the name of the element we would like to access. For example, to access the object’s variable “value”, we have to type “no.value”.

7.1.2 Definition of class methods

Let’s now define a similar class for the number six (6) and add class methods for acquiring the double of the number 6 and the previous and next integer number of number 6.

```
class Six:
    value = 6
    name = "six"

    def get_double(self):
        return 2*self.value

    def get_previous_and_next_number(self):
        previous_no = self.value - 1
        next_no = self.value + 1
        return (previous_no,next_no)

no = Six()
print("no object:",no)
print("Double of no:",no.get_double())
print("Previous and next of no:",no.get_previous_and_next_number())
```

The output will look like:

```
no object: <__main__.Six object at 0x7fa64f8514c0>
Double of no: 12
Previous and next of no: (5, 7)
```

A dot followed by the name of the element is used to access the elements of a class object. Notice that when we tried to print the class object, the output stated that it is an object of type Six at a specific location in the memory. This happens because we have not defined what a string representation of the class should be.

7.1.3 Class initialisation and method overloading

Let's now create a class for representing numbers in general. The class should have a numerical value equal to the number we would like to depict and support addition, subtraction and multiplication between class objects. We will also define a string representation for the class, as well as a custom method for acquiring the represented number in the power of 2.

```
class MyNumber:
    value = 0

    # Define how a class object will be initialised
    def __init__(self,number):
        self.value = number

    # Define a string representation for the class object
    def __str__(self):
        return "%f" % (self.value)

    # Define the addition operation between two class obects
    def __add__(self,other):
        result = self.value + other.value
        return MyNumber(result)

    # Define the subtraction operation between two class obects
    def __sub__(self, other):
        return MyNumber(self.value - other.value)

    # Define the multiplication operation between two class obects
    def __mul__(self, other):
        return MyNumber(self.value * other.value)

    # Custom function that returns the power of 2 of the number
    def get_power_of_2(self):
        return self.value*self.value

ten = MyNumber(10)
two = MyNumber(2)
```

```
print("ten:",ten)
print("two:",two)

print("Addition:",(ten+two))
print("Subtraction:",(ten-two))
print("Multiplication:",(ten*two))

result = (ten * ten) + (ten * two) + (ten - two) # Compute (10*10)+(10*2)+(10-2)=128
print("(10*10)+(10*2)+(10-2) =",result)
print("[(10*10)+(10*2)+(10-2)]^2 =",result.get_power_of_2())
```

The output will look like:

```
ten: 10.000000
two: 2.000000
Addition: 12.000000
Subtraction: 8.000000
Multiplication: 20.000000
(10*10)+(10*2)+(10-2) = 128.000000
[(10*10)+(10*2)+(10-2)]^2 = 16384
```

As you can see above, we defined the class "MyNumber" that contains the variable "value" for storing its numerical value. By defining a class method named __init__, we can define what actions will happen when we create an object of class MyNumber. In this case, we indicated that the constructor of the class should take two arguments, the class object itself, as well as the number that the object will depict. As a result, the initialisation method will assign the number that we would like the object to depict to the class variable "value".

To define a string representation for the class MyNumber, we define the method __str__ that returns a string, which in this case is the string representation of the number that the class object depicts.

To define how objects of the class MyNumber can be added, subtracted or multiplied with each other, we defined the methods __add__, __sub__, and __mul__ respectively. These methods take as arguments the class object itself, as well as another object of class MyNumber and return the sum, difference and product of the variables "value" respectively.

Methods that start and end with __ are special methods in Python classes that are used for various operations. Overloading them, i.e. changing their standard definition with a custom one, alters the behaviour of the class for these operations, e.g. addition, multiplication, print, etc.

7.1.4 Definition of a custom class for text documents

Let's define a class for storing text documents that supports some operations that are helpful in Natural Language processing, such as tokenising the depicted text into words and preprocessing the text.

```
from nltk import word_tokenize # Import the word_tokenize function from NLTK
import re # Import the re package
from nltk.corpus import stopwords # Import the stop words lists from NLTK
from string import punctuation


class Document:
    text = "" # Variable to store raw text
    words = [] # Variable to store word tokenised text

    # Define how a class object will be initialised
    def __init__(self,textstring):
        self.text = textstring
        self.words = word_tokenize(self.text)

    # Define a string representation for the class object
    def __str__(self):
        return self.text

    # Returns list of lowercase words, omitting punctuation and english stopwords
    def get_words_preprocessed(self):
```

```
        punctuation_list = list(punctuation)
        stopwords_english = stopwords.words('english') # Load the stop words list for English
        result = []
        for word in self.words:
            if ((word not in punctuation_list) and (word not in stopwords_english)):
                result.append(word.lower())
        return result

    # Retuns list of words that match the regex. Results are lowercased
    def get_words_preprocessed_regex(self,regex):
        result = []
        for word in self.words:
            regex_check = re.match(regex, word)
            if(regex_check!=None):
                if(regex_check.group()==word):
                    result.append(word.lower())
        return result

d = Document("Yesterday we went to the cinema to watch a new movie. The movie was amazing but we paid
    20 pounds for each ticket!")

print("Text:",d)
print("\nTokens:",d.words)
print("\nTokens preprocessed:",d.get_words_preprocessed())
print("\nTokens preprocessed with regex '[a-zA-Z]+':",d.get_words_preprocessed_regex("[a-zA-Z]+"))
```

The output will look like:

```
Text: Yesterday we went to the cinema to watch a new movie. The movie was amazing but we paid 20
    pounds for each ticket!

Tokens: ['Yesterday', 'we', 'went', 'to', 'the', 'cinema', 'to', 'watch', 'a', 'new', 'movie', '.',
    'The', 'movie', 'was', 'amazing', 'but', 'we', 'paid', '20', 'pounds', 'for', 'each', 'ticket',
    '!']

Tokens preprocessed: ['yesterday', 'went', 'cinema', 'watch', 'new', 'movie', 'the', 'movie',
    'amazing', 'paid', '20', 'pounds', 'ticket']

Tokens preprocessed with regex '[a-zA-Z]+': ['yesterday', 'we', 'went', 'to', 'the', 'cinema', 'to',
    'watch', 'a', 'new', 'movie', 'the', 'movie', 'was', 'amazing', 'but', 'we', 'paid', 'pounds',
    'for', 'each', 'ticket']
```

7.2 Preparation of spam detection dataset

Let's use a large dataset with 6046 emails annotated as spam (1) or ham (0) in order to train traditional machine learning models for the task of classifying text between spam and ham.

7.2.1 Dataset loading

First, we will load the dataset from the completeSpamAssassin.csv file using the Pandas package and print the first 3 rows. The completeSpamAssassin.csv is a comma separated file that contains three columns. The first is unnamed and contains an index number for each email, the second is named "Body" and contains the email's text, and the third is called "Label" and contains the label of the email as 0 or 1, with 1 referring to spam and 0 to ham.

```
import pandas as pd

df = pd.read_csv("completeSpamAssassin.csv")

df.head(3)
```

The output will look like:

```
Unnamed: 0  Body  Label
0  0  \nSave up to 70% on Life Insurance.\nWhy Spend...  1
1  1  1) Fight The Risk of Cancer!\nhttp://www.adcli...  1
2  2  1) Fight The Risk of Cancer!\nhttp://www.adcli...  1
```

Then, let's count how many emails are contained in the dataset, create a list of the emails from the dataset, create a list of the labels in the dataset, and check if we loaded an equal number of emails and email labels.

```
count_row = df.shape[0] # Gives number of rows
print("Total emails in dataframe:",count_row,"\n")

emails = df["Body"].tolist() # Convert column "Body" of the dataframe to a list and store to a
    variable
labels = df["Label"].tolist() # Convert column "Label" of the dataframe to a list and store to a
    variable

print("Total emails:",len(emails))
print("Total labels:",len(labels))
```

The output will look like:

```
Total emails in dataframe: 6046

Total emails: 6046
Total labels: 6046
```

The emails are now stored in the list "emails", while the respective labels are stored in the list "labels" which is of equal size and the label at index i is associated with the i-th email in the "emails" list.

7.2.2 Word tokenisation

Let's now tokenise each email to its lower-cased constituent words and print the first email from the dataset to inspect the result. We will also remove any email that is empty or consists of only one word.

```
from nltk import word_tokenize # Import the word_tokenize function from NLTK

emails_tokenised = []
labels_final = []

print("Tokenising emails...",end="")
for i in range(len(emails)):
    try:
        tokens = word_tokenize(emails[i].lower()) # Tokenise email
        if(len(tokens)>1): # Discard single word emails
            emails_tokenised.append(tokens) # Add email tokens to list
            labels_final.append(labels[i]) # Add label for valid email to labels list
    except:
        pass
print("[DONE]\n")

print("Total emails:",len(emails_tokenised))
print("Total labels:",len(labels_final),"\n")

print(emails_tokenised[0]) # Print first email
```

The output will look like:

```
Tokenising emails...[DONE]

Total emails: 5507
Total labels: 5507

['save', 'up', 'to', '70', '%', 'on', 'life', 'insurance', '.', 'why', 'spend', 'more', 'than',
    'you', 'have', 'to', '?', 'life', 'quote', 'savings', 'ensuring', 'your', 'family', "'s",
    'financial', 'security', 'is', 'very', 'important', '.', 'life', 'quote', 'savings', 'makes',
```

```
    'buying', 'life', 'insurance', 'simple', 'and', 'affordable', '.', 'we', 'provide', 'free',
    'access', 'to', 'the', 'very', 'best', 'companies', 'and', 'the', 'lowest', 'rates.life',
    'quote', 'savings', 'is', 'fast', ',', 'easy', 'and', 'saves', 'you', 'money', '!', 'let', 'us',
    'help', 'you', 'get', 'started', 'with', 'the', 'best', 'values', 'in', 'the', 'country', 'on',
    'new', 'coverage', '.', 'you', 'can', 'save', 'hundreds', 'or', 'even', 'thousands', 'of',
    'dollars', 'by', 'requesting', 'a', 'free', 'quote', 'from', 'lifequote', 'savings', '.', 'our',
    'service', 'will', 'take', 'you', 'less', 'than', '5', 'minutes', 'to', 'complete', '.', 'shop',
    'and', 'compare', '.', 'save', 'up', 'to', '70', '%', 'on', 'all', 'types', 'of', 'life',
    'insurance', '!', 'click', 'here', 'for', 'your', 'free', 'quote', '!', 'protecting', 'your',
    'family', 'is', 'the', 'best', 'investment', 'you', "'ll", 'ever', 'make', '!', 'if', 'you',
    'are', 'in', 'receipt', 'of', 'this', 'email', 'in', 'error', 'and/or', 'wish', 'to', 'be',
    'removed', 'from', 'our', 'list', ',', 'please', 'click', 'here', 'and', 'type', 'remove', '.',
    'if', 'you', 'reside', 'in', 'any', 'state', 'which', 'prohibits', 'e-mail', 'solicitations',
    'for', 'insurance', ',', 'please', 'disregard', 'this', 'email', '.']
```

As you can see above, we successfully tokenised the emails. However, the number of valid emails in the dataset was reduced to 5507 after removing empty emails and emails consisting of only one word. Furthermore, as you can see above, the word list for the first email contains punctuation, numbers and some symbols.

7.2.3 Pre-processing

We will further pre-process the emails by removing any word that does not consist only of lowercase letters. Remember that we have already converted all text to lowercase. As a result, this step will eliminate any word consisting of symbols, numbers and punctuation. In addition, we will first remove hyphens and dots from words in order to avoid discarding words containing a hyphen (e.g. lower-case) or abbreviations (e.g. U.K.).

```python
from nltk.corpus import stopwords # Import the stop words lists from NLTK
import re # Import the re package

stopwords_english = stopwords.words('english') # Load the stop words list for English in variable

emails_preprocessed = emails_tokenised # Create copy of emails_tokenised

print("Preprocessing emails..",end="")
for i in range(len(emails_tokenised)):
    new_tokens = []
    for word in emails_tokenised[i]:
        word = word.replace("-","") # Remove hyphens from words, e.g. lower-case->lowercase
        word = word.replace(".","") # Remove dots from words to normalise abbreviations, e.g. U.K.->UK
        # Select only tokens that consist of letters from a to z
        regex_check = re.match("[a-z]+", word)
        if(regex_check!=None):
            if(regex_check.group()==word):
                new_tokens.append(word)
    emails_preprocessed[i] = new_tokens
print("[DONE]\n")

# Check if pre-processing led to any empty emails
for i in range(len(emails_preprocessed)):
    if(len(emails_preprocessed[i])==0):
        print("Email",i,"is empty!")

print(emails_preprocessed[0]) # Print first email
```

The output will look like:

```
Preprocessing emails..[DONE]

['save', 'up', 'to', 'on', 'life', 'insurance', 'why', 'spend', 'more', 'than', 'you', 'have', 'to',
    'life', 'quote', 'savings', 'ensuring', 'your', 'family', 'financial', 'security', 'is', 'very',
    'important', 'life', 'quote', 'savings', 'makes', 'buying', 'life', 'insurance', 'simple', 'and',
    'affordable', 'we', 'provide', 'free', 'access', 'to', 'the', 'very', 'best', 'companies', 'and',
    'the', 'lowest', 'rateslife', 'quote', 'savings', 'is', 'fast', 'easy', 'and', 'saves', 'you',
    'money', 'let', 'us', 'help', 'you', 'get', 'started', 'with', 'the', 'best', 'values', 'in',
    'the', 'country', 'on', 'new', 'coverage', 'you', 'can', 'save', 'hundreds', 'or', 'even',
```

```
    'thousands', 'of', 'dollars', 'by', 'requesting', 'a', 'free', 'quote', 'from', 'lifequote',
    'savings', 'our', 'service', 'will', 'take', 'you', 'less', 'than', 'minutes', 'to', 'complete',
    'shop', 'and', 'compare', 'save', 'up', 'to', 'on', 'all', 'types', 'of', 'life', 'insurance',
    'click', 'here', 'for', 'your', 'free', 'quote', 'protecting', 'your', 'family', 'is', 'the',
    'best', 'investment', 'you', 'ever', 'make', 'if', 'you', 'are', 'in', 'receipt', 'of', 'this',
    'email', 'in', 'error', 'wish', 'to', 'be', 'removed', 'from', 'our', 'list', 'please', 'click',
    'here', 'and', 'type', 'remove', 'if', 'you', 'reside', 'in', 'any', 'state', 'which',
    'prohibits', 'email', 'solicitations', 'for', 'insurance', 'please', 'disregard', 'this', 'email']
```

7.2.4 Join email words list

We will then join the list of words for each email into a single string, having a white-space character between the words. This step is not always mandatory but the method that we will use later for creating the TF-IDF vector for each email requires its input to be a single string.

```python
dataset = []
for i in range(len(emails_preprocessed)):
    text = " ".join(emails_preprocessed[i]) # Join words with an empty space between them
    dataset.append(text)

print(dataset[0])
```

The output will look like:

```
save up to on life insurance why spend more than you have to life quote savings ensuring your family
    financial security is very important life quote savings makes buying life insurance simple and
    affordable we provide free access to the very best companies and the lowest rateslife quote
    savings is fast easy and saves you money let us help you get started with the best values in the
    country on new coverage you can save hundreds or even thousands of dollars by requesting a free
    quote from lifequote savings our service will take you less than minutes to complete shop and
    compare save up to on all types of life insurance click here for your free quote protecting your
    family is the best investment you ever make if you are in receipt of this email in error wish to
    be removed from our list please click here and type remove if you reside in any state which
    prohibits email solicitations for insurance please disregard this email
```

7.3 Text classification for spam detection

7.3.1 Splitting of dataset into training and test sets

In order to avoid overfitting the machine learning models that we will create and in order to get a fair estimate of their performance, we will divide our dataset into a training set containing 80% of the dataset's samples and a test set containing the remaining 20% of the samples. Note that the 80-20 split is not mandatory. A 70-30, 60-40, 50-50 or any other split can be used, as long as the training and test sets are kept separate.

```python
from sklearn.model_selection import train_test_split

# Split the dataset to a test set with 20% of the emails and a traning set with the rest 80% of the
    emails
samples_train, samples_test, labels_train, labels_test = train_test_split(dataset, labels_final,
    test_size=0.2, random_state=42)

no_of_training_samples = len(samples_train)
no_of_test_samples = len(samples_test)
total_samples = no_of_training_samples+no_of_test_samples

print("Total samples:\t\t%4d" % total_samples)
print("Training samples:\t%4d (%2.2f%s)" %
    (no_of_training_samples,(no_of_training_samples/total_samples)*100,"%"))
print("Test samples:\t\t%4d (%2.2f%s)" %
    (no_of_test_samples,(no_of_test_samples/total_samples)*100,"%"))
```

The output will look like:

```
Total samples:  5507
Training samples: 4405 (79.99%)
Test samples:   1102 (20.01%)
```

Note that we set the value of the random state to a specific value in order to always acquire the same split. This is very useful when you are creating and testing your code but **it should NOT be used for real evaluations** of model performance.

7.3.2 Text classification using Naive Bayes

We will now use the training set to train a Naive Bayes classifier in order to predict whether the emails in the test set are spam or ham. To do this, we will define a model pipeline that first computes the TF-IDF vectors for the input text and then trains a Multinomial Naive Bayes model using the TF-IDF vectors of the input text and the respective class labels.

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline

# Build the Naive Bayes model by setting a pipeline where the input is first converted
# to TF-IDF vectors and then a Multinomial Naive Bayes is used
model = make_pipeline(TfidfVectorizer(), MultinomialNB())

model.fit(samples_train, labels_train) # Train the model on the training data
predicted_categories = model.predict(samples_test) # Predict the categories of the test data

print("Predicted:",predicted_categories.tolist()[0:10]) # Print the first 10 predictions
print("Ground truth:",labels_test[0:10]) # Print the first 10 ground truth values
```

The output will look like:

```
Predicted: [1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
Ground truth: [1, 1, 0, 0, 0, 0, 1, 1, 0, 0]
```

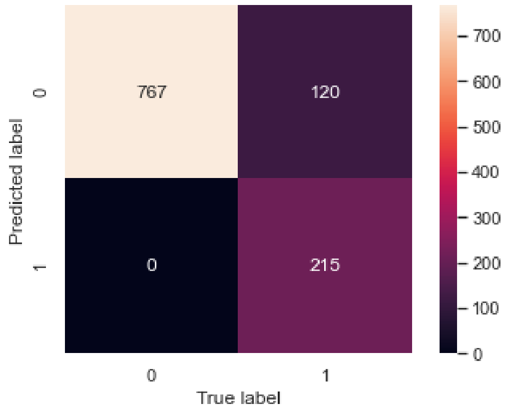
As you can see above, some of the predictions for the first 10 emails of the test set are not correct.

7.3.3 Computation and plotting of Naive Bayes's classification performance

Let's compute the confusion matrix, the accuracy, F1-score, precision, and recall for the Naive Bayes model that we just tested. We will also plot the confusion matrix to offer a visual description of the model's performance. Note that the F1-score, precision, and recall metrics are computed for each class. In order to compute a single value across our two classes, we have to indicate that we would like to compute the mean across the classes by setting the argument "average" equal to "macro" (for indicating a macro averaging).

```python
from sklearn.metrics import confusion_matrix, accuracy_score, f1_score,precision_score,recall_score,
    classification_report
import seaborn as sns
import matplotlib.pyplot as plt
sns.set() # use seaborn plotting style

# Plot the confusion matrix
mat = confusion_matrix(labels_test, predicted_categories)
sns.heatmap(mat.T, square = True, annot=True, fmt = "d")
plt.xlabel("True label")
plt.ylabel("Predicted label")
plt.show()

# Compute and print classification performance metrics
print("Accuracy:\t%f" % accuracy_score(labels_test, predicted_categories))
print("F1-score:\t%f" % f1_score(labels_test, predicted_categories, average='macro'))
print("Precision:\t%f" % precision_score(labels_test, predicted_categories, average='macro'))
print("Recall:\t\t%f" % recall_score(labels_test, predicted_categories, average='macro'))

print("\nClassification performance:\n%s" % classification_report(labels_test, predicted_categories))
```

The output will look like:




```
Accuracy:  0.891107
F1-score:  0.854633
Precision: 0.932356
Recall:    0.820896

Classification performance:
             precision    recall  f1-score   support

           0       0.86      1.00      0.93       767
           1       1.00      0.64      0.78       335

    accuracy                           0.89      1102
   macro avg       0.93      0.82      0.85      1102
weighted avg       0.91      0.89      0.88      1102
```

As you can see above, our model successfully classified 89.11% of our test emails and achieved an F1-score of 85.46%.

7.3.4 Text classification using kNN

Let's examine the performance of the k Nearest Neighbours (KNN) classifier for the same task, for $k = 3$.

```
from sklearn.neighbors import KNeighborsClassifier

# Build the kNN model by setting a pipeline where the input is first converted
# to TF-IDF vectors and then a kNN classifier for k=3 is used
model = make_pipeline(TfidfVectorizer(), KNeighborsClassifier(n_neighbors=3))

model.fit(samples_train, labels_train) # Train the model on the training data
predicted_categories = model.predict(samples_test) # Predict the categories of the test data

print("Predicted:",predicted_categories.tolist()[0:10]) # Print the first 10 predictions
print("Ground truth:",labels_test[0:10]) # Print the first 10 ground truth values
```

The output will look like:

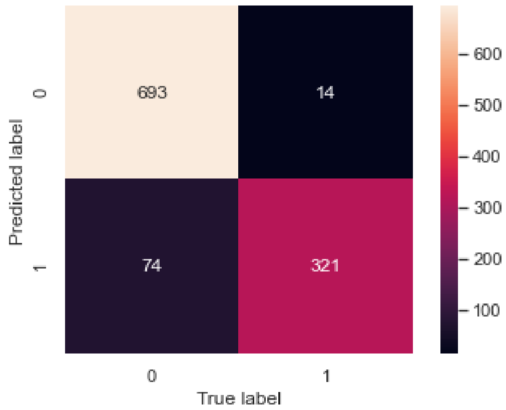
```
Predicted: [1, 1, 0, 0, 0, 0, 1, 1, 0, 0]
Ground truth: [1, 1, 0, 0, 0, 0, 1, 1, 0, 0]
```

7.3.5 Computation and plotting of kNN's classification performance

```
# Plot the confusion matrix
mat = confusion_matrix(labels_test, predicted_categories)
sns.heatmap(mat.T, square = True, annot=True, fmt = "d")
plt.xlabel("True label")
```

```
plt.ylabel("Predicted label")
plt.show()

# Compute and print classification performance metrics
print("Accuracy:\t%f" % accuracy_score(labels_test, predicted_categories))
print("F1-score:\t%f" % f1_score(labels_test, predicted_categories, average='macro'))
print("Precision:\t%f" % precision_score(labels_test, predicted_categories, average='macro'))
print("Recall:\t\t%f" % recall_score(labels_test, predicted_categories, average='macro'))

print("\nClassification performance:\n%s" % classification_report(labels_test, predicted_categories))
```

The output will look like:




```
Accuracy:  0.920145
F1-score:  0.909875
Precision: 0.896428
Recall:    0.930865

Classification performance:
              precision    recall  f1-score   support

           0       0.98      0.90      0.94       767
           1       0.81      0.96      0.88       335

    accuracy                           0.92      1102
   macro avg       0.90      0.93      0.91      1102
weighted avg       0.93      0.92      0.92      1102
```

As you can see above, kNN ($k = 3$) achieved a higher classification accuracy and F1-score compared to Naive Bayes.

7.4 Saving and loading a trained machine learning model

We saw how to train some machine learning models for specific classification tasks. However, it would be a waste of computational resources to retrain a model every time we would like to use it. The solution is to save the trained model into a file and load it every time it is needed.

7.4.1 Save trained model in a file for future use

Let's save the trained kNN model from the previous section in a file called "pickle_model_3NN_spamemails.pkl".

```
import pickle # Import pickle package for object serialisation

# Save to file in the current working directory
pkl_filename = "pickle_model_3NN_spamemails.pkl"
with open(pkl_filename, 'wb') as file: # Open file as binary file for writing (wb)
    pickle.dump(model, file)
```

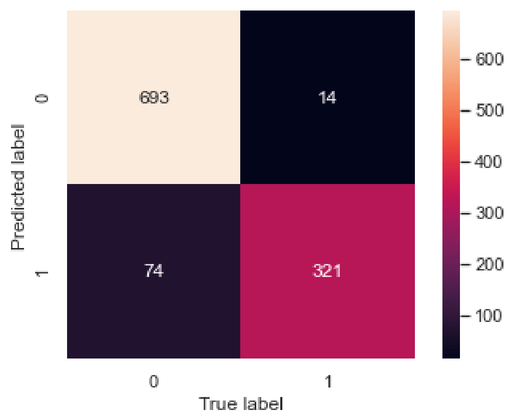
If you check your working directory, you should now have a file named "pickle_model_3NN_spamemails.pkl".

7.4.2 Loading and use of saved trained model

Let's now load the model from "pickle_model_3NN_spamemails.pkl" and use it to classify the emails from our test set. The results should be identical as the ones shown in Section 7.3.5.

```
# Load model from pickle file
with open("pickle_model_3NN_spamemails.pkl", 'rb') as file: # Open file as binary file for reading
    (rb)
    pickle_model = pickle.load(file)

# Use loaded model
predicted_categories = pickle_model.predict(samples_test) # Predict the categories of the test data

# Plot the confusion matrix
mat = confusion_matrix(labels_test, predicted_categories)
sns.heatmap(mat.T, square = True, annot=True, fmt = "d")
plt.xlabel("True label")
plt.ylabel("Predicted label")
plt.show()

# Compute and print classification performance metrics
print("Accuracy:\t%f" % accuracy_score(labels_test, predicted_categories))
print("F1-score:\t%f" % f1_score(labels_test, predicted_categories, average='macro'))
print("Precision:\t%f" % precision_score(labels_test, predicted_categories, average='macro'))
print("Recall:\t\t%f" % recall_score(labels_test, predicted_categories, average='macro'))

print("\nClassification performance:\n%s" % classification_report(labels_test, predicted_categories))
```

The output will look like:




```
Accuracy:  0.920145
F1-score:  0.909875
Precision: 0.896428
Recall:    0.930865

Classification performance:
             precision    recall  f1-score   support

          0       0.98      0.90      0.94       767
          1       0.81      0.96      0.88       335

    accuracy                          0.92      1102
   macro avg       0.90      0.93      0.91      1102
weighted avg       0.93      0.92      0.92      1102
```

As you can see, the results are identical with the ones from Section 7.3.5, as expected.

7.5 Exercises

Exercise 7.1 Run the kNN experiment from Section 7.3.4 for all k from 1 to 20 and plot the achieved classification F1-score vs. the k of the kNN. Note: Create a function to compute the F1-score for each k.

Exercise 7.2 Preprocess the spam email dataset again in a similar way as in Section 7.2.3, but also remove the English stopwords. Use Naive Bayes to classify the test set and compare the achieved performance with the performance from Section 7.3.3. Remember to use the same random state (42) as in Section 7.3.1 in order to get the same split for the training and test sets.

Exercise 7.3 Repeat Exercise 7.2 but do not apply any preprocessing to the input text apart from converting all characters to lowercase. Remember to use the same random state as in Exercise 7.2 in order to get the same split for the training and test sets.

Workshop 8: Text classification using Recurrent Neural Networks (RNNs)

In this workshop, we are going to use a type of Recurrent Neural Network (RNN) called Long Short-term Memory (LSTM) for the task of text classification and more specifically for the task of classifying documents as referring to fake news or real news.

8.1 Dataset preparation

8.1.1 Load fake news dataset

Let's first load the fake news dataset from the news.csv file. The dataset contains four columns. The first one is unnamed and denotes an identification number for each text. The second is the title of each text and has the title "title". The third contains the main body of each text and has the title "text" and the fourth contains the label of each text (FAKE, REAL) and has the title "label". We are going to use Pandas to create a dataframe with the dataset's data.

```
import pandas as pd

df = pd.read_csv("news.csv")

df.head(5)
```

The output will look like:

```
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
```

8.1.2 Dataset pre-processing

We will then concatenate the title and the main body of each text and create a new column in the dataset containing the concatenated text. Then, we will convert the label "FAKE" to 1 and the label "REAL" to 0. Finally, we will remove any column from the dataset apart from the new column and the label column.

```
df['label'] = (df['label'] == 'FAKE').astype('int') # Set value of label to 1 if FAKE, else to 0 for
    REAL
df['alltext'] = df['title'] + ". " + df['text'] # Concatenate title and text into column alltext
df = df.reindex(columns=['alltext','label']) # Transform the dataset to contain only the label and
    alltext columns

df.head(5) # Show first 5 rows in dataset
```

The output will look like:

```
                                             alltext  label
0  You Can Smell Hillary's Fear. Daniel Greenfiel...  1
1  Watch The Exact Moment Paul Ryan Committed Pol...  1
2  Kerry to go to Paris in gesture of sympathy. U...  0
3  Bernie supporters on Twitter erupt in anger ag...  1
4  The Battle of New York: Why This Primary Matte...  0
```

Then, we will remove any text that is less than 50 characters long and convert all characters to lowercase. Furthermore, at this stage we could apply any other pre-processing step, such as removing punctuation, removing stop words, etc.

```
# Remove texts that are less than 50 characters long
df.drop(df[df.alltext.str.len() < 50].index, inplace=True)
df = df.reset_index(drop=True)

def convert_to_lowercase(text): # Convert all characters to lowercase
    return text.lower()

# Convert text to lowercase
df['alltext'] = df['alltext'].apply(convert_to_lowercase)

print("Samples:",len(df['alltext']))
print("Labels:",len(df['label']),"\n")

print(df['alltext'].iloc[0][0:1000]) # Print the first 1000 characters of the first text as an example

df.to_csv('fakenews_processed.csv',index=False) # Save processed dataset to csv file
```

The output will look like:

```
Samples: 6327
Labels: 6327

you can smell hillary's fear. daniel greenfield, a shillman journalism fellow at the freedom center,
    is a new york writer focusing on radical islam. in the final stretch of the election, hillary
    rodham clinton has gone to war with the fbi. the word "unprecedented" has been thrown around so
    often this election that it ought to be retired. but it's still unprecedented for the nominee of
    a major political party to go war with the fbi. but that's exactly what hillary and her people
    have done. coma patients just waking up now and watching an hour of cnn from their hospital beds
    would assume that fbi director james comey is hillary's opponent in this election. the fbi is
    under attack by everyone from obama to cnn. hillary's people have circulated a letter attacking
    comey. there are currently more media hit pieces lambasting him than targeting trump. it wouldn't
    be too surprising if the clintons or their allies were to start running attack ads against the
    fbi. the fbi's leadership is bei
```

8.1.3 Divide dataset into training and test

Then, we will divide the dataset into a training set containing 70% of the dataset's samples and a test set that contains 30% of the dataset's samples.

```
from sklearn.model_selection import train_test_split

# Split dataset into 30% for test and 70% for training
train, test = train_test_split(df, test_size = 0.30, random_state = 42)

X_train = train["alltext"].values # Get the documents for training
Y_train = train["label"].values # Get the labels for training
X_test = test["alltext"].values # Get the documents for testing
Y_test = test["label"].values  # Get the labels for testing
```

The training samples are now stored in the X_train variable, the test samples in the X_test variable, whereas the training labels are stored in the Y_train variable and the test labels in the Y_test variable.

8.2 Create LSTM architecture

8.2.1 Create a preprocessing layer that prepares the vocabulary

We will first use a command to avoid getting some harmless warnings.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3' # Reduce the number of shown warnings
```

Attention: Only disable warnings in your code if you are sure that they are harmless.

We will then create a preprocessing layer which maps text features to integer sequences.

```
import tensorflow as tf
from tensorflow.keras.layers import TextVectorization

MAX_VOCABULARY_WORDS = 5000 # The maximum number of words to be used (most frequent)
MAX_SEQUENCE_LENGTH = 200 # Number of words in each text. Sequence length to pad the outputs to.
EMBEDDING_DIM = 10 # Size of the word embedding to be used

# Create a preprocessing layer which maps text features to integer sequences
vectorize_layer = TextVectorization(
    max_tokens=MAX_VOCABULARY_WORDS, # Maximum size of the vocabulary for this layer
    output_mode='int', # Represent each word in the vocabulary with an integer
    output_sequence_length=MAX_SEQUENCE_LENGTH) # Pad the sequence length to size MAX_SEQUENCE_LENGTH

vectorize_layer.adapt(X_train) # Computes a vocabulary of string terms from tokens in a dataset.
vocabulary = vectorize_layer.get_vocabulary() # Get the vocabulary

print("Vocabulary size: " + str(len(vocabulary)) + " words")
```

The output will look like:

```
Vocabulary size: 5000 words
```

Note that we defined that our vocabulary will have a maximum size of 5,000 words, each text will have a maximum size of 200 words, and we will use a word embedding with 10 dimensions. All these are hyperparameters that we can adjust through experimentation. The smaller the vocabulary, the text size, and the embedding dimensions, the faster our models will be trained. However, small values may affect the performance negatively.

8.2.2 Define network architecture

Let's define a neural network architecture for the task of text classification using the following layers:

- The text vectorisation preprocessing layer that we created above.
- An embedding layer that converts a text, in the form of a list of numbers corresponding to tokens in the vocabulary, to a list of embeddings of a required size.
- Two stacked bidirectional LSTM layers with a dropout layer in the output of the last LSTM layer.
- One dense layer with one neuron that uses the sigmoid activation function and will be the output of the network.

Note that although we have two output classes, we only used one neuron in the output layer. This works in binary classification, where the output is either 0 or 1, as the sigmoid activation function will provide an output between 0 and 1. By rounding the output to the nearest integer (0 or 1) we can compute the predicted class. In case that we had more classes to predict, the size of the output layer should be equal to the number of the classes and a SoftMax activation function should be used.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Input, Embedding, LSTM, Dense, Bidirectional

model = Sequential(name="MyLSTM")

model.add(Input(shape=(1,), dtype=tf.string))
```

```
model.add(vectorize_layer)
model.add(Embedding(input_dim=MAX_VOCABULARY_WORDS, # Size of the vocabulary
                  output_dim=EMBEDDING_DIM, # Size of the word embedding
                  input_length=MAX_SEQUENCE_LENGTH)) # Length of input sequences
model.add(Bidirectional(LSTM(16,return_sequences=True))) # Set return_sequences=True if you want
    additional LSTM layer
model.add(Bidirectional(LSTM(16,go_backwards=True,dropout=0.2)))
model.add(Dense(1, activation='sigmoid'))

model.summary() # Print model summary
```

The output will look like:

```
Model: "MyLSTM"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 text_vectorization (TextVec (None, 200)               0
 torization)

 embedding (Embedding)       (None, 200, 10)           50000

 bidirectional (Bidirectiona (None, 200, 32)           3456
 l)

 bidirectional_1 (Bidirectio (None, 32)                6272
 nal)

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 59,761
Trainable params: 59,761
Non-trainable params: 0
```

As you can see above, the selected network architecture with the selected hyperparameters contains 59,761 trainable parameters. Increasing the size of the used embeddings and the size and the number of the layers will lead to more trainable parameters and consequently to higher computational requirements for training the model.

8.2.3 Define hyperparameters and initialise model

Let's define the hyperparameters of our network, including the optimiser, the loss function and the performance metric that will be used for training the model, and initialise a model using the network that we defined.

```
from tensorflow.keras.optimizers import Adam

EPOCHS = 50
BATCH_SIZE = 64
LEARNING_RATE = 0.01

opt = Adam(learning_rate=LEARNING_RATE) # Initialise Adam optimiser with a leanring rate of 0.01

model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy']) # Initialise model
```

8.3 Train and test an LSTM model

8.3.1 Train the LSTM model

Let's now train the model. We will define a callback function that will stop the training after 4 epochs without any improvement to the validation accuracy. "Early stopping" is a common strategy used when training neural networks in order to avoid wasting computational resources and training time if a model is not improving. In addition, we will use 10% of the training data as validation data in order to evaluate the trained model at each epoch.

```
from tensorflow.keras.callbacks import EarlyStopping

early = EarlyStopping(monitor="val_accuracy",patience=4,restore_best_weights=True,mode="auto")

history = model.fit(X_train, Y_train,
                  epochs=EPOCHS, batch_size=BATCH_SIZE,
                  validation_split=0.1, # Use 10% of training data for validation
                  callbacks=[early]
                 )
```

The output will look like:

```
Epoch 1/50
63/63 [==============================] - 18s 145ms/step - loss: 0.4135 - accuracy: 0.8095 - val_loss:
    0.3251 - val_accuracy: 0.8623
Epoch 2/50
63/63 [==============================] - 6s 101ms/step - loss: 0.1593 - accuracy: 0.9458 - val_loss:
    0.2813 - val_accuracy: 0.8826
Epoch 3/50
63/63 [==============================] - 6s 99ms/step - loss: 0.0685 - accuracy: 0.9792 - val_loss:
    0.3142 - val_accuracy: 0.8758
Epoch 4/50
63/63 [==============================] - 6s 103ms/step - loss: 0.0529 - accuracy: 0.9804 - val_loss:
    0.4749 - val_accuracy: 0.8646
Epoch 5/50
63/63 [==============================] - 6s 97ms/step - loss: 0.0500 - accuracy: 0.9834 - val_loss:
    0.3479 - val_accuracy: 0.8804
Epoch 6/50
63/63 [==============================] - 6s 98ms/step - loss: 0.0267 - accuracy: 0.9935 - val_loss:
    0.4661 - val_accuracy: 0.8646
```

All the metrics computing at each training epoch are stored in the history variable. Note that the output may differ a bit when you run the following code due to the random initialisation of the model's weights.

8.3.2 Plot the loss and accuracy per epoch

We will now plot the training and validation loss computed at each training epoch.

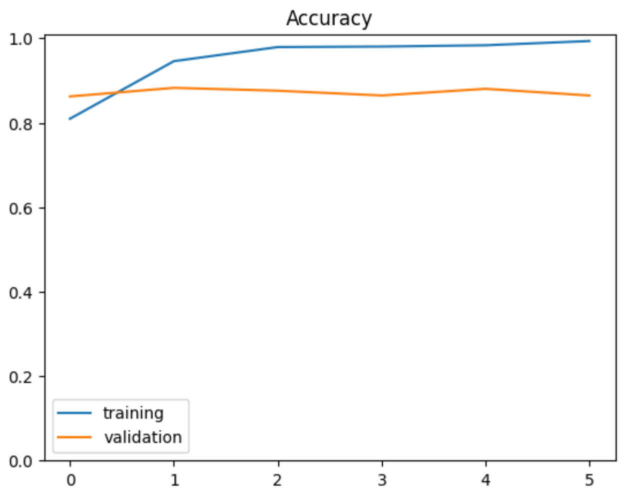
```
import matplotlib.pyplot as plt

plt.title('Loss')
plt.plot(history.history['loss'], label='training')
plt.plot(history.history['val_loss'], label='validation')
plt.ylim(bottom=0.0)
plt.legend()
plt.show();
```

The output will look like:

We will then plot the training and validation accuracy computed at each training epoch.

```
plt.title('Accuracy')
plt.plot(history.history['accuracy'], label='training')
plt.plot(history.history['val_accuracy'], label='validation')
plt.ylim(bottom=0.0,top=1.01)
plt.legend()
plt.show();
```

The output will look like:



8.3.3 Evaluate trained LSTM model on the test set

We will now evaluate the trained LSTM model on our test set.

```
# Use model to predict the class of the documents in the test set
pred = model.predict(X_test)

# Print numpy array containing the predicted class labels
print("Predicted values:\n" + str(pred))

# Use the numpy.round() function to round the predictions to the nearest integer
pred = pred.round()

pred = pred.astype(int) # Convert predicted values to integers
print("\nRounded values:\n" + str(pred))
```

The output will look like:

```
60/60 [==============================] - 3s 19ms/step
Predicted values:
[[0.9803157 ]
 [0.96182203]
 [0.9780567 ]
 ...
 [0.53544426]
 [0.9282485 ]
 [0.25072488]]

Rounded values:
[[1]
 [1]
 [1]
 ...
 [1]
 [1]
 [0]]
```

As you can see above, the model predicts values between 0 and 1 due to the use of a sigmoid activation function in its output. We then round the predicted values to the nearest integer in order to convert them to class labels (0 or 1).

We will the compare the predicted labels with the test labels from or dataset to evaluate the performance of our model.

```
from sklearn.metrics import classification_report

print(classification_report(Y_test,pred))
```

The output will look like:

```
              precision    recall  f1-score   support

           0       0.92      0.89      0.90       949
           1       0.89      0.92      0.91       950

    accuracy                           0.91      1899
   macro avg       0.91      0.91      0.91      1899
weighted avg       0.91      0.91      0.91      1899
```

As you can see above, the model reached an 90% classification accuracy for the test set. Note that we opted to stop the training of the model after 4 epochs without an improvement in the validation accuracy. In real applications, models should be trained for more epochs, ideally until they stop improving.

8.4 Classify text using trained model

We will now use the trained model to predict whether the sentence "the earth is not flat" is fake or real news. **Note:** When using trained models to predict the class of a text, make sure that you pre-process the input text in a similar manner as the text that was used to train the model.

```
labels = {1: "FAKE", 0: "REAL"}

text = "the earth is not flat"

pred = model.predict([text])
pred = pred[0][0].round().astype(int)

print("\n" + text + ": " + labels[pred])
```

The output will look like:

```
1/1 [==============================] - 0s 38ms/step
```

```
the earth is not flat: REAL
```

Please note that since the model was trained on bigger passages of text compared to the one used in the example above, it is not expected to perform very well with inputs that are very different than the texts used to train it. Always remember that a trained machine learning model will always reflect the data that it was trained on.

8.5 Store and load a model from a file

8.5.1 Save the trained model using pickle

We will now save the trained model to a file using the pickle library.

```
import pickle

pickle.dump(model, open('mylstm.pkl', 'wb'))
```

The model is now stored in a file named 'mylstm.pkl'. Note that tensorflow and keras also have internal functions for storing a trained model using a standardised format. However, the TextVectorisation layer that we used in this workshop is not yet supported by the internal tensorflow function.

8.5.2 Load model using pickle

We will then load the model from the 'mylstm.pkl' file and use it to predict the class of the texts in our test set. The result should be the same as before.

```
loaded_model = pickle.load(open('mylstm.pkl', 'rb'))

new_pred = loaded_model.predict(X_test)

# Print numpy array containing the predicted class labels
print("Predicted values:\n" + str(new_pred))

# Use the numpy.round() function to round teh predictions to the nearest integer
new_pred = new_pred.round()

new_pred = new_pred.astype(int) # Convert predicted values to integers
print("\nRounded values:\n" + str(new_pred))

print(classification_report(Y_test,new_pred))
```

The output will look like:

```
60/60 [==============================] - 3s 18ms/step
Predicted values:
[[0.9803157 ]
 [0.96182203]
 [0.9780567 ]
 ...
 [0.53544426]
 [0.9282485 ]
 [0.25072488]]

Rounded values:
[[1]
 [1]
 [1]
 ...
 [1]
 [1]
 [0]]
              precision    recall  f1-score   support

           0       0.92      0.89      0.90       949
```

```
           1       0.89      0.92      0.91       950

    accuracy                           0.91      1899
   macro avg       0.91      0.91      0.91      1899
weighted avg       0.91      0.91      0.91      1899
```

As expected, the results are similar to the ones achieved in Section 8.3.3.

8.6 Exercises

Exercise 8.1 Create a non-bidirectional LSTM model with a single LSTM layer with 32 units. Train it by keeping all hyperparameters and training settings the same as for the model above. Plot the loss and accuracy and compare them with the model above.

Exercise 8.2 Pre-process the dataset again in order to remove punctuation and stop words. Retrain the LSTM model from 8.2.2 using the new pre-processed dataset and compare its performance with when punctuation and stopwords are not removed.

Exercise 8.3 Train the model from Exercise 8.1 using the dataset from exercise 8.2 and the same hyperparameters and settings. Compare the performance to the one achieved in Exercise 8.1.

Appendix: Test files

A.1 movies.xml

```xml
<movies>
  <movie id="1">
    <title>And Now for Something Completely Different</title>
    <released>1971</released>
  </movie>
  <movie id="2">
    <title>Monty Python and the Holy Grail</title>
    <released>1974</released>
  </movie>
  <movie id="3">
    <title>Monty Python's Life of Brian</title>
    <released>1979</released>
  </movie>
  <movie id="4">
    <title>Monty Python Live at the Hollywood Bowl</title>
    <released>1982</released>
  </movie>
  <movie id="5">
    <title>Monty Python's The Meaning of Life</title>
    <released>1983</released>
  </movie>
</movies>
```

A.2 links.html

```html
<!doctype html>
<html lang="en-GB">

<head>
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <title>Test page for Text Mining and Language Analytics</title>
</head>

<body>
   <h1>Test page for Text Mining and Language Analytics</h1>
   <a href="https://www.durham.ac.uk/departments/academic/computer-science/">Link 1</a><br/>
   <a href="https://www.gov.uk/government/organisations/hm-revenue-customs">Link 2</a><br/>
   <a href="https://www.gov.uk/">Link 3</a><br/>
   <a href="https://www.dur.ac.uk/">Link 4</a><br/>
   <a href="https://www.nhs.uk/">Link 5</a><br/>
</body>

</html>
```

A.3 emails.txt

john+acme.co@hotmail.com
bob@gmail.com
tom@durham.ac.uk
jerry@durham.ac.uk
scrooge@durham.ac.uk
donald@yahoo.co.uk
huey@yahoo.co.uk
dewey@gmail.com
louie.duck@durham.ac.uk
gyro.gearloose@yahoo.co.uk
bart@yahoo.co.uk
homer@gmail.com
stan@hotmail.com
kyle-broflovski@durham.ac.uk
eric@yahoo.co.uk
kenny@gmail.com
butters@durham.ac.uk
wendy@hotmail.com
randy_marsh@durham.ac.uk
chef@gmail.com

A.4 cds.xml

```
<?xml version="1.0" encoding="ISO8859-1" ?>
<CATALOG>
<CD>
<TITLE>Empire Burlesque</TITLE>
<ARTIST>Bob Dylan</ARTIST>
<COUNTRY>USA</COUNTRY>
<COMPANY>Columbia</COMPANY>
<PRICE>10.90</PRICE>
<YEAR>1985</YEAR>
</CD>
<CD>
<TITLE>Hide your heart</TITLE>
<ARTIST>Bonnie Tylor</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>CBS Records</COMPANY>
<PRICE>9.90</PRICE>
<YEAR>1988</YEAR>
</CD>
<CD>
<TITLE>Greatest Hits</TITLE>
<ARTIST>Dolly Parton</ARTIST>
<COUNTRY>USA</COUNTRY>
<COMPANY>RCA</COMPANY>
<PRICE>9.90</PRICE>
<YEAR>1982</YEAR>
</CD>
<CD>
<TITLE>Still got the blues</TITLE>
<ARTIST>Gary More</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>Virgin redords</COMPANY>
<PRICE>10.20</PRICE>
<YEAR>1990</YEAR>
</CD>
<CD>
<TITLE>Eros</TITLE>
<ARTIST>Eros Ramazzotti</ARTIST>
```

```xml
<COUNTRY>EU</COUNTRY>
<COMPANY>BMG</COMPANY>
<PRICE>9.90</PRICE>
<YEAR>1997</YEAR>
</CD>
<CD>
<TITLE>One night only</TITLE>
<ARTIST>Bee Gees</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>Polydor</COMPANY>
<PRICE>10.90</PRICE>
<YEAR>1998</YEAR>
</CD>
<CD>
<TITLE>Sylvias Mother</TITLE>
<ARTIST>Dr.Hook</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>CBS</COMPANY>
<PRICE>8.10</PRICE>
<YEAR>1973</YEAR>
</CD>
<CD>
<TITLE>Maggie May</TITLE>
<ARTIST>Rod Stewart</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>Pickwick</COMPANY>
<PRICE>8.50</PRICE>
<YEAR>1990</YEAR>
</CD>
<CD>
<TITLE>Romanza</TITLE>
<ARTIST>Andrea Bocelli</ARTIST>
<COUNTRY>EU</COUNTRY>
<COMPANY>Polydor</COMPANY>
<PRICE>10.80</PRICE>
<YEAR>1996</YEAR>
</CD>
<CD>
<TITLE>When a man loves a woman</TITLE>
<ARTIST>Percy Sledge</ARTIST>
<COUNTRY>USA</COUNTRY>
<COMPANY>Atlantic</COMPANY>
<PRICE>8.70</PRICE>
<YEAR>1987</YEAR>
</CD>
<CD>
<TITLE>Black angel</TITLE>
<ARTIST>Savage Rose</ARTIST>
<COUNTRY>EU</COUNTRY>
<COMPANY>Mega</COMPANY>
<PRICE>10.90</PRICE>
<YEAR>1995</YEAR>
</CD>
<CD>
<TITLE>1999 Grammy Nominees</TITLE>
<ARTIST>Many</ARTIST>
<COUNTRY>USA</COUNTRY>
<COMPANY>Grammy</COMPANY>
<PRICE>10.20</PRICE>
<YEAR>1999</YEAR>
</CD>
<CD>
<TITLE>For the good times</TITLE>
<ARTIST>Kenny Rogers</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>Mucik Master</COMPANY>
```

```
<PRICE>8.70</PRICE>
<YEAR>1995</YEAR>
</CD>
<CD>
<TITLE>Big Willie style</TITLE>
<ARTIST>Will Smith</ARTIST>
<COUNTRY>USA</COUNTRY>
<COMPANY>Columbia</COMPANY>
<PRICE>9.90</PRICE>
<YEAR>1997</YEAR>
</CD>
<CD>
<TITLE>Tupelo Honey</TITLE>
<ARTIST>Van Morrison</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>Polydor</COMPANY>
<PRICE>8.20</PRICE>
<YEAR>1971</YEAR>
</CD>
<CD>
<TITLE>Soulsville</TITLE>
<ARTIST>Jorn Hoel</ARTIST>
<COUNTRY>Norway</COUNTRY>
<COMPANY>WEA</COMPANY>
<PRICE>7.90</PRICE>
<YEAR>1996</YEAR>
</CD>
<CD>
<TITLE>The very best of</TITLE>
<ARTIST>Cat Stevens</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>Island</COMPANY>
<PRICE>8.90</PRICE>
<YEAR>1990</YEAR>
</CD>
<CD>
<TITLE>Stop</TITLE>
<ARTIST>Sam Brown</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>A and M</COMPANY>
<PRICE>8.90</PRICE>
<YEAR>1988</YEAR>
</CD>
<CD>
<TITLE>Bridge of Spies</TITLE>
<ARTIST>T`Pau</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>Siren</COMPANY>
<PRICE>7.90</PRICE>
<YEAR>1987</YEAR>
</CD>
<CD>
<TITLE>Private Dancer</TITLE>
<ARTIST>Tina Turner</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>Capitol</COMPANY>
<PRICE>8.90</PRICE>
<YEAR>1983</YEAR>
</CD>
<CD>
<TITLE>Midt om natten</TITLE>
<ARTIST>Kim Larsen</ARTIST>
<COUNTRY>EU</COUNTRY>
<COMPANY>Medley</COMPANY>
<PRICE>7.80</PRICE>
<YEAR>1983</YEAR>
```

```
</CD>
<CD>
<TITLE>Pavarotti Gala Concert</TITLE>
<ARTIST>Luciano Pavarotti</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>DECCA</COMPANY>
<PRICE>9.90</PRICE>
<YEAR>1991</YEAR>
</CD>
<CD>
<TITLE>The dock of the bay</TITLE>
<ARTIST>Otis Redding</ARTIST>
<COUNTRY>USA</COUNTRY>
<COMPANY>Atlantic</COMPANY>
<PRICE>7.90</PRICE>
<YEAR>1987</YEAR>
</CD>
<CD>
<TITLE>Picture book</TITLE>
<ARTIST>Simply Red</ARTIST>
<COUNTRY>EU</COUNTRY>
<COMPANY>Elektra</COMPANY>
<PRICE>7.20</PRICE>
<YEAR>1985</YEAR>
</CD>
<CD>
<TITLE>Red</TITLE>
<ARTIST>The Communards</ARTIST>
<COUNTRY>UK</COUNTRY>
<COMPANY>London</COMPANY>
<PRICE>7.80</PRICE>
<YEAR>1987</YEAR>
</CD>
<CD>
<TITLE>Unchain my heart</TITLE>
<ARTIST>Joe Cocker</ARTIST>
<COUNTRY>USA</COUNTRY>
<COMPANY>EMI</COMPANY>
<PRICE>8.20</PRICE>
<YEAR>1987</YEAR>
</CD>
</CATALOG>
```


A.5 alice.txt

Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, ”and what is the use of a book,” thought Alice ”without pictures or conversations?”

So she was considering in her own mind (as well as she could, for the hot day made her feel very sleepy and stupid), whether the pleasure of making a daisy-chain would be worth the trouble of getting up and picking the daisies, when suddenly a White Rabbit with pink eyes ran close by her.

There was nothing so very remarkable in that; nor did Alice think it so very much out of the way to hear the Rabbit say to itself, ”Oh dear! Oh dear! I shall be late!” (when she thought it over afterwards, it occurred to her that she ought to have wondered at this, but at the time it all seemed quite natural); but when the Rabbit actually took a watch out of its waistcoat-pocket, and looked at it, and then hurried on, Alice started to her feet, for it flashed across her mind that she had never before seen a rabbit with either a waistcoat-pocket, or a watch to take out of it, and burning with curiosity, she ran across the field after it, and fortunately was just in time to see it pop down a large rabbit-hole under the hedge.

A.6 dune.txt

In the week before their departure to Arrakis, when all the final scurrying about had reached a nearly unbearable frenzy, an old crone came to visit the mother of the boy, Paul.
It was a warm night at Castle Caladan, and the ancient pile of stone that had served the Atreides family as home for twenty-six generations bore that cooled-sweat feeling it acquired before a change in the weather.
The old woman was let in by the side door down the vaulted passage by Paul's room and she was allowed a moment to peer in at him where he lay in his bed.
By the half-light of a suspensor lamp, dimmed and hanging near the floor, the awakened boy could see a bulky female shape at his door, standing one step ahead of his mother. The old woman was a witch shadow - hair like matted spiderwebs, hooded 'round darkness of features, eyes like glittering jewels.