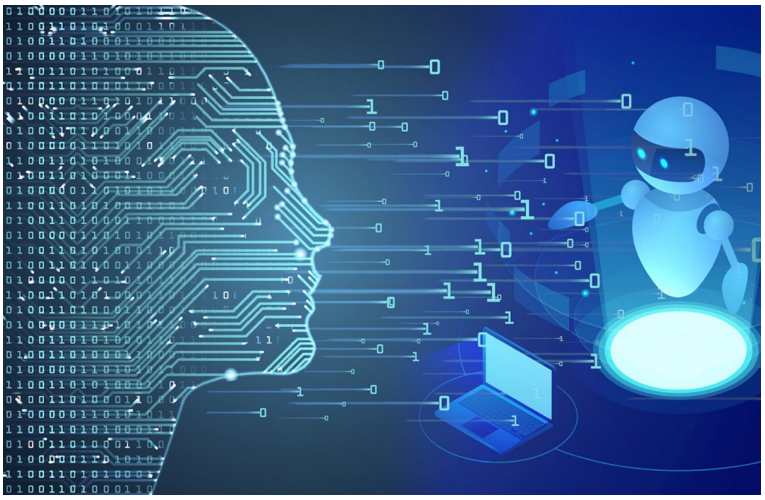
Durham University

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
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"
^	Starts with	"^She"
$	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 "^" as the first character of the class. For example, [^5] will match any character except "5". If the caret appears elsewhere in a character class, it does not have special meaning. For example: [5^] will match either a "5" or a "^".

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 [^0-9].
\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 [^\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 [^a-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 stooooooop stooooooooooop*". 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 stooooooop stooooooooooop")
print("One or more occurrences of 'o':")
for match in x:
    print(match)

x = re.findall("o{2}", "stp stop stoop stooooooop stooooooooooop")
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
ooooooooooo

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.

```
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.*"

Appendix: Test files

A.1 movies.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

```
<!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>
```

```
<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.