Lecture S2. Text Processing

Instructor: Luping Yu

May 30, 2023

I. Regular Expression (RE)

It is mainly used for searching and manipulating text strings. In simple words, you can easily search the pattern and replace them with the matching pattern with the help of regular expression.

This concept or tool is used in almost all the programming or scripting languages such as PHP, C, C++, Java, Perl, JavaScript, Python, Ruby, and many others.

RE Basics

In regular expressions, if a character is directly specified, it represents an exact match. `\d` can match a digit, and `\w` can match a letter or digit, `.` can match any character.

- `'00\d'` can match `'007'` but not `'00A'`
- `'\d\d\d'` can match `'010'`
- `'\w\w\d'` can match `'py3'`
- `'py.'` can match `'pyc'`, `'pyo'`, `'py!'`, and so on

To match complicated strings, we use certain symbols within a regular expression. `*` represents any number of characters (including zero), `+` represents at least one character, `?` represents zero or one character, `{n}` represents exactly n characters, and `{n,m}` represents a range of n to m characters.

Let's look at a complex example: `\d{3}\s+\d{3,8}`. Let's break it down from left to right:

- `\d{3}` matches three digits, for example, `'010'`
- `\s` can match a whitespace character (including tabs and other whitespace characters), so `\s+` represents at least one whitespace character, for example, matching `' '`, `'  '`, and so on
- `\d{3,8}` matches 3 to 8 digits, for example, `'1234567'`

Putting it all together, the above regular expression can match phone numbers with an area code separated by any number of spaces.

But what if we want to match a number like `'010-12345'`? Since `'-'` is a special character, we need to **escape** it with a backslash in the regular expression, so it becomes `'\d{3}\-\d{3,8}'`.

However, it still cannot match `'010 - 12345'` due to the space. Therefore, we need more complex matching methods.

RE Advanced

To perform more precise matching, we can use `[]` to represent a range. For example:

- `[0-9a-zA-Z]` can match a digit, letter, or underscore;
- `[0-9a-zA-Z_]+` can match a string consisting of at least one digit, letter, or underscore, such as 'a100', '0_Z', 'Py3000', and so on;
- `[a-zA-Z][0-9a-zA-Z_]*` can match a string that starts with a letter, followed by any number of digits, letters, or underscores. This represents a valid Python variable;
- `[a-zA-Z][0-9a-zA-Z_]{0,19}` further restricts the variable's length to be 1-20 characters (first character + up to 19 additional characters).
- `A|B` can match A or B, so `(P|p)ython` can match `'Python'` or `'python'`.
- `^` represents the beginning of a line, `^\d` means it must start with a digit.
- `$` represents the end of a line, `\d$` means it must end with a digit.

Regular Expression in Python

Python has a built-in package called `re`, which can be used to work with Regular Expressions.

In [1]:
```
import re
```

The `re` module functions fall into three categories: pattern **matching**, **substitution**, and **splitting**. Let's look at a simple example:

In [2]:
```
text = 'a b  c   d \t e \n d'

text.split(' ')
```

Out[2]:
```
['a', 'b', '', 'c', '', '', 'd', '\t', 'e', '\n', 'd']
```

Suppose we wanted to split a string with a variable number of whitespace characters (tabs, spaces, and newlines). The regex describing one or more whitespace characters is `\s+`:

In [3]:
```
re.split('\s+', text)
```

Out[3]:
```
['a', 'b', 'c', 'd', 'e', 'd']
```

When you call `re.split('\s+', text)`, the regular expression is first **compiled**, and then its split method is called on the passed text.

You can compile the regex yourself with `re.compile`, forming a reusable regex object:

In [4]:
```
regex = re.compile('\s+')

regex.split(text)
```

Out[4]:
```
['a', 'b', 'c', 'd', 'e', 'd']
```

If, instead, you wanted to get a list of all patterns matching the regex, you can use the `findall` method:

In [5]:
```
regex.findall(text)
```

Out[5]:
```
[' ', '  ', '   ', ' \t ', ' \n ']
```

IMPORTANT: To avoid unwanted escaping with `\</code> in a regular expression, use **raw string literals** like r'C:\x' instead of the equivalent 'C:\\x' .`

Let's consider a block of text and a regular expression capable of identifying most email addresses:

In [6]:
```
text = '''Dave dave@google.com XMU
Steve steve@gmail.com XMU SCHOOL OF MANAGEMENT
Rob rob@gmail.com THU
Ryan ryan@yahoo.com PKU
'''

text
```

Out[6]:
```
'Dave dave@google.com XMU\nSteve steve@gmail.com XMU SCHOOL OF MANAGEMEN
T\nRob rob@gmail.com THU\nRyan ryan@yahoo.com PKU\n'
```

In [7]:
```
pattern = r'[a-z0-9]+@[a-z0-9]+.[a-z]{3}'
regex = re.compile(pattern)

regex.findall(text)
```

Out[7]:
```
['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.co
m']
```

Relatedly, `sub` will return a new string with occurrences of the pattern replaced by the a new string:

In [8]:
```
print(regex.sub('REDACTED', text))
```

```
Dave REDACTED XMU
Steve REDACTED XMU SCHOOL OF MANAGEMENT
Rob REDACTED THU
Ryan REDACTED PKU
```

Suppose you wanted to find email addresses and simultaneously segment each address into its three components: username, domain name, and domain suffix. To do this, put `( )` around the parts of the pattern to segment:

In [9]:
```
pattern = r'([a-z0-9]+)@([a-z0-9]+).([a-z]{3})'

regex = re.compile(pattern)

regex.findall(text)
```

Out[9]:
```
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

Regular Expression Characters

There are following different type of characters of a regular expression:

- Metacharacters
- Quantifier
- Groups and Ranges
- Escape Characters or character classes

Metacharacters

Metacharacters	Description	Example
^	This character is used to match an expression to its right at the start of a string.	^a is an expression match to the string which starts with 'a' such as "aab", "a9c", "apr", "aaaaab", etc.
$	The $sign is used to match an expression to its left at the end of a string.	r$ is an expression match to a string which ends with r such as "aaabr", "ar", "r", "aannn9r", etc.
.	This character is used to match any single character in a string except the line terminator, i.e. /n.	b.x is an expression that match strings such as "bax", "b9x", "bar".
|	It is used to match a particular character or a group of characters on either side. If the character on the left side is matched, then the right side's character is ignored.	A|b is an expression which gives various strings, but each string contains either a or b.
A	It is used to match the character 'A' in the string.	This expression matches those strings in which at least one-time A is present. Such strings are "Amcx", "mnAr", "mnopAx4".

Metacharacters	Description	Example
Ab	It is used to match the substring 'ab' in the string.	This expression matches those strings in which 'Ab' is present at least one time. Such strings are "Abcx", "mnAb", "mnopAbx4".

Quantifiers

The quantifiers are used in the regular expression for specifying the number of occurrences of a character.

Characters	Description	Example
+	This character specifies an expression to its left for one or more times.	s+ is an expression which gives "s", "ss", "sss", and so on.
?	This character specifies an expression to its left for 0 (Zero) or 1 (one)times.	as? is an expression which gives either "a" or "as", but not "ass".
*	This character specifies an expression to its left for 0 or more times	Br* is an expression which gives "B", "Br", "Brr", "Brrr", and so on...
{x}	It specifies an expression to its left for only x times.	Mab{5} is an expression which gives the following string which contains 5 b's: "Mabbbbb"
{x, }	It specifies an expression to its left for x or more times.	Xb{3, } is an expression which gives various strings containing at least 3 b's. Such strings are "Xbbb", "Xbbbb", and so on.
{x,y}	It specifies an expression to its left, at least x times but less than y times.	Pr{3,6}a is an expression which provides two strings. Both strings are as follows:"Prrrr" and "Prrrrr"

Groups and Ranges

The groups and ranges in the regular expression define the collection of characters enclosed in the brackets.

Characters	Description	Example
( )	It is used to match everything which is in the simple bracket.	A(xy) is an expression which matches with the following string: "Axy"
{ }	It is used to match a particular number of occurrences defined in the curly bracket for its left string.	xz{4,6} is an expression which matches with the following string: "xzzzzz"
[ ]	It is used to match any character from a range of characters defined in the square bracket.	xz[atp]r is an expression which matches with the following strings: "xzar", "xztr", and "xzpr"
[pqr]	It matches p, q, or r individually.	Following strings are matched with this expression: "p", "q", and "r".
[pqr][xy]	It matches p, q, or r, followed by either x or y.	Following strings are matched with this expression: "px", "qx", and "rx", "py", "qy", and "ry".

Characters	Description	Example
[a-z]	It matches letters of a small case from a to z.	This expression matches the strings such as: "a", "python", "good".
[A-Z]	It matches letters of an upper case from A to Z.	This expression matches the strings such as: "EXCELLENT", "NATURE".
[0-9]	It matches a digit from 0 to 9.	This expression matches the strings such as: "9845", "54455"
^[a-zA-Z]	It is used to match the string, which is either starts with a small case or upper-case letter.	This expression matches the strings such as: "A854xb", "pv4fv", "cdux".
ab[^4-9]	It matches those digits or characters which are not defined in the square bracket.	This expression matches those strings which do not contain 5, 6, 7, and 8.

Escape Characters or Character Classes

Characters	Description
\s	It is used to match a one white space character.
\0	It is used to match a NULL character.
\n	It helps a user to match a new line.
\d	It is used to match one decimal digit, which means from 0 to 9.
\D	It is used to match any non-decimal digit.
\w	It is used to match the alphanumeric [0-9a-zA-Z] characters.
\W	It is used to match one non-word character

Interactive Exercises

https://regexone.com/

II. Fuzzy Match

String matching can be useful for a variety of situations, for example, joining two tables by an firm's name when it is spelled or punctuated differently in both tables. `FuzzyWuzzy` uses a some similarity ratio between two sequences and returns the **similarity percentage**.

Edits and edit distance

The fuzzy string matching algorithm seeks to determine the degree of closeness between two different strings. This is discovered using a distance metric known as the "edit distance". The edit distance determines how close two strings are by finding the minimum number of "edits" required to transform one string to another.

There are four main types of edits:

- Insert (add a letter)
- Delete (remove a letter)
- Switch (swap two adjacent letters)
- Replace (change one letter to another)

`FuzzyWuzzy` uses edit distance (aka. **Levenshtein** distance) to calculate the degree of closeness between two strings.

Simple Fuzzy String Matching

The library can be installed by using pip:

In [4]:
```
pip install fuzzywuzzy
pip install python-Levenshtein
```

In [11]:
```
from fuzzywuzzy import fuzz
```

Simple Ratio

When you have a very simple set of strings which look almost similar with their words, you can use the simple ratio from the `FuzzyWuzzy` package.

In [12]:
```
str1 = 'FuzzyWuzzy is a lifesaver!'
str2 = 'fuzzy wuzzy is a LIFE SAVER.'

fuzz.ratio(str1, str2)
```

Out[12]: 52

As seen in the above code, the first string matches to the second one with 52%. This ratio uses a simple technique which involves calculating the edit distance between two strings.

For an instance, it recognizes missing punctuations, case-sensitive words, misspelled words etc.

In [13]:
```
fuzz.ratio(str1.lower(), str2.lower())
```

Out[13]: 93

The similarity ratio percentage here is 93%. We can say the first string has a similarity of 93% to the second string **when both are lowercase**.

Partial Ratio

FuzzyWuzzy also has more powerful functions to help with matching strings in more complex situations. `partial_ratio()` function allows us to perform **substring** matching. This works by taking the shortest string and matching it with all substrings that are of the same length.

In [14]:
```
fuzz.ratio('Xiamen, Fujian', 'Xiamen')
```

Out[14]: 60

In [15]:
```
fuzz.partial_ratio('Xiamen, Fujian', 'Xiamen')
```

Out[15]: 100

Using the `partial_ratio()` function above, we get a similarity ratio of 100.

When dealing with substrings, i.e., one short string being a part of some other long string, we use `partial_ratio()` function.

The mechanism of this ratio deals with something known as **optimal partial logic**. For an instance, let the shorter string length be 'm' and the longer string length be 'n'. Then, partial ratio finds a best-matching sub-string of length 'm'.

Token Sort Ratio

If we want to ignore the **ordering** of the words in the strings but still determine how similar they are - token sort helps you do exactly that.

In [16]:
```
fuzz.ratio("fuzzy wuzzy was a bear", "wuzzy fuzzy was a bear")
```

Out[16]: 91

In [17]:
```
fuzz.token_sort_ratio("fuzzy wuzzy was a bear", "wuzzy fuzzy was a bear")
```

Out[17]: 100

Token Set Ratio

When you don't care about the number of times a word in the string is **repeated**, then it is better to use the Token Set Ratio from the package.

In [18]:
```
fuzz.token_sort_ratio("Xiamen University is the best", "Xiamen Xiamen Uni
```

Out[18]: 89

In [19]:
```
fuzz.token_set_ratio("Xiamen University is the best", "Xiamen Xiamen Univ
```

Out[19]: 100