[Tutorial] Text Processing

1. Introduction

1.1. What is Natural Language Processing?

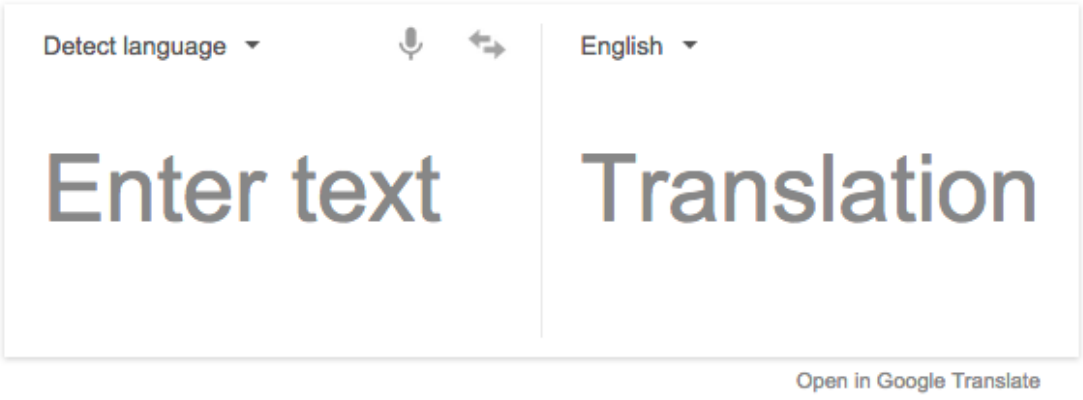
Natural Language Processing (often writen NLP) is a really wide field, covering (among many others) the following applications:

- Text translation
- Speech recognition
- Natural language understanding
- Sentiment analysis
- Topic modelling
- Summarization
- Grammatical correction

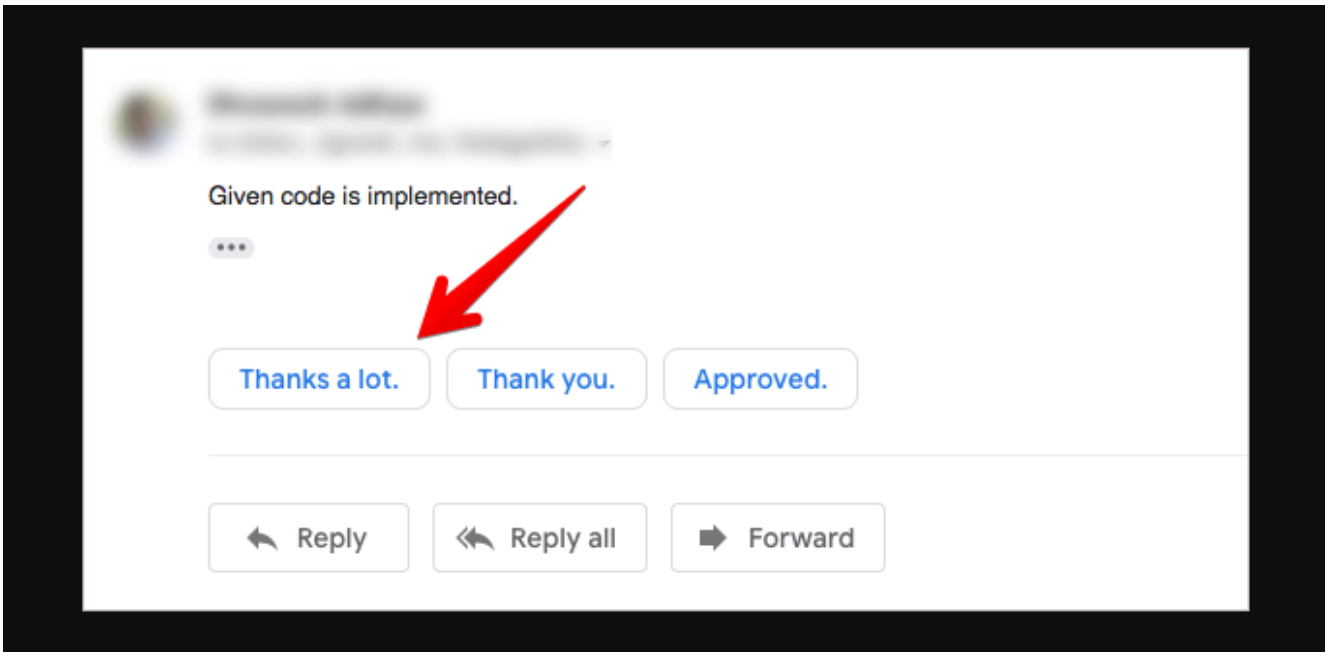
1.2. NLP everyday examples

We all use NLP based applications everyday without even noticing it!

👉 One of the most used NLP application is Google translate:

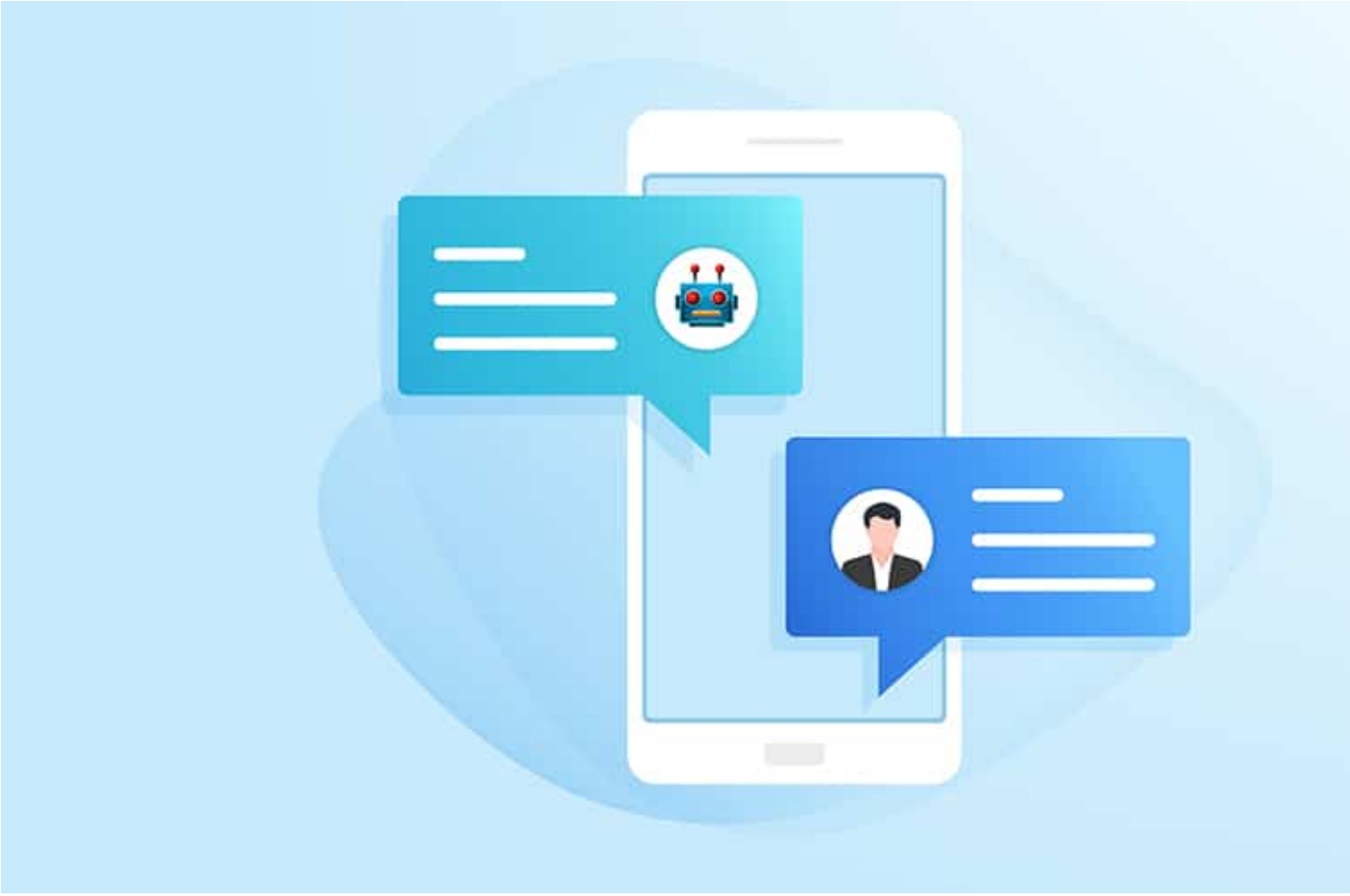


👉 You may also have noticed that Gmail now suggests smart replies:



👉 Another one is Siri on the iPhone. Using the same technology, Google home and Amazon Alexa are also good examples.

👉 Finally, many website are now using chatbots, based on NLP algorithms too!



> This week will just be an introduction, we won't be able to implement all these solutions at the end of the week. Indeed, many of them actually require deep learning models.

---

2. Text Preprocessing with NLTK

NLTK stands for **Natural Language Toolkit**. This is one of the most frequently used libraries for NLP, especially for doing **research**. Indeed, NLTK is **not particularly focusied on performance** but on providing a wide range of algorithms.

We will use it widely for **text preprocessing tasks** suck as tokenization, lemmatizaton and stemming.

> ⚙️ Install it with `pip install nltk`

2.1. Tokenization

Tokenization is usually one of the first steps of preprocessing when doing NLP. Tokenization is the process of **splitting a text into words** (hence, a word is being called a token). We already did some tokenization without knowing it, remember?

```
document = "Bryan is in the kitchen"
tokens = document.split(' ')
print(tokens)
```

```
['Bryan', 'is', 'in', 'the', 'kitchen']
```

That was an easy one, what happens when it's getting more complicated?

```
document = "Ain't no sunshine when she's gone. And she's always gone too long. Anytime she goes awa
y"
tokens = document.split(' ')
print(tokens)
```

```
["Ain't", 'no', 'sunshine', 'when', "she's", 'gone.', 'And', "she's", 'always', 'gone', 'too', 'lon
g.', 'Anytime', 'she', 'goes', 'away']
```

😱 Not that good, right? "Ain't" and "she's" are not actually one word, but two... And I'm not sure I want my punctuations like in "gone." to be in the words either...

➡️ Let's use NLTK to improve it!

```
# Import the tokenizer of NLTK
from nltk.tokenize import word_tokenize

# Tokenize our sentence
tokens = word_tokenize(document)
```

```
print(tokens)
```

```
['Ai', "n't", 'no', 'sunshine', 'when', 'she', "'s", 'gone', '.', 'And', 'she', "'s", 'always', 'gon
e', 'too', 'long', '.', 'Anytime', 'she', 'goes', 'away']
```

Okay, it doesn't look perfect, but at last, the number of words is correct. The punctuation is kept as a word, but it's easy to remove:

```
tokens = [t for t in tokens if t.isalpha()]
print(tokens)
```

```
['Ai', 'no', 'sunshine', 'when', 'she', 'gone', 'And', 'she', 'always', 'gone', 'too', 'long', 'Anyt
ime', 'she', 'goes', 'away']
```

Congratulations, we now have the right number of words without punctuation !

> 🖌️ Hint : Tokens can not only be words, but also tuple of words, sentences, or even characters depending on the context! However, the principles remain the same.

2.2. Removing Stop Words

Stop words are words that typically **add no value to the text**, but are only here for grammatical reasons. Let's see some examples of stop words with NLTK.

If Python raises an error while importing the stopwords, please use the following commands:

```
import nltk
nltk.download('stopwords')
```

```
from nltk.corpus import stopwords

stop_words = stopwords.words('english')
print(stop_words[:10])
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're"]
```

➡️ Let's remove those words from our tokens using NLTK:

```
tokens_no_stops = [t for t in tokens if t not in stop_words]
print(tokens_no_stops)
```

We can see that some of the tokens like: `And` were not removed.

That is because python is case sensitive:

```
    'And' == 'and' # False
```

So be careful to lower every tokens in your corpuses.

Now we can go to the final step of preprocessing: stemming and lemmatizing!

2.3. Stemming and Lemmatization

Stemming and lemmatization are basically the action of **keeping only the root of words**.

They don't work exactly the same:

- Stemming will basically keep **only the root** of the word (i.e. merely truncating the word)
- Lemmatization will **add some context to get the relevant root** of a word

```
import nltk
nltk.download('wordnet')
```

```
[nltk_data] Downloading package wordnet to /Users/gaelle/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

```
True
```

```
# Let's import the libraries for stemming and lemmatization
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer

# Then we have to create an instance
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()
```

Some examples

```
print(stemmer.stem('connection'), stemmer.stem('connected'), stemmer.stem('connective'))
```

```
connect connect connect
```

```
# but what if the words don't mean the same thing once truncated
print(stemmer.stem('meaning'), stemmer.stem('meanness'))
```

```
mean mean
```

Like we just saw with that last example, stemming has it's limits

```
raw_words = ['maximum', 'presumably', 'multiply', 'provision', 'owed', 'ear', 'saying', 'string', 'c
rying',
             'meant', 'cement', 'is', 'are', 'spectra', 'lives']

stemmed_words = [stemmer.stem(w) for w in raw_words]
lem_words = [lemmatizer.lemmatize(w) for w in raw_words]

print("stemmed words : ", stemmed_words)
print("lemmatized words : ", lem_words)
```

```
stemmed words :  ['maximum', 'presum', 'multipli', 'provis', 'owe', 'ear', 'say', 'string', 'cri',
'meant', 'cement', 'is', 'are', 'spectra', 'live']
lemmatized words :  ['maximum', 'presumably', 'multiply', 'provision', 'owed', 'ear', 'saying', 'str
ing', 'cry', 'meant', 'cement', 'is', 'are', 'spectrum', 'life']
```

As you can see, there are some differences... Actually, we did not use the lemmatization in a proper way. To do so, we need to **define what type of word** we are lemmatizing: 'n' for noun, 'v' for verb, 'a' for adjective and 'r' for adverbs. The default value is 'n' for noun.

```
print(lemmatizer.lemmatize('is', 'n'))
print(lemmatizer.lemmatize('is', 'v'))
```

```
is
be
```

Okay that works better on verbs now, but... do we have to give the type of every word by hands? No, don't worry!

Doing this classification automatically is called **POS-tagging** (POS stands for Part Of Speech). We can do it using NLTK and a small trick... We will see that later during the exercises !

> 🖌️ Just remember the different POS-tags:

Tag	Description
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun

Tag	Descr
PRP$	Posse
RB	Adver
RBR	Adver
RBS	Adver
RP	Partic
SYM	Symb
TO	to
UH	Interj
VB	Verb,
VBD	Verb,
VBG	Verb,
VBN	Verb,
VBP	Verb,
VBZ	Verb,
WDT	Whde
WP	Whpr
WP$	Posse
WRB	Whac

2.4. Ngrams

Up unitl now, we have only used unigrams of words. But sometimes, we may want to use also **bigrams**, or even **trigrams** of words. Or even unigrams, bigrams and trigrams of characters, why not?

👉 Let's see on a simple example what would be bigrams of the following sentence: « To be or not to be »

```
# Import NLTK
import nltk
from nltk.tokenize import word_tokenize

# Define the text
text = "To be or not to be"

# Tokenize the text
tokens = word_tokenize(text)

# Display the bigrams
list(nltk.bigrams(tokens))
```

```
[('To', 'be'), ('be', 'or'), ('or', 'not'), ('not', 'to'), ('to', 'be')]
```

As you can see, we now have bigrams, no unigrams. If you want to be more generic, you can use the module `ngrams` of NLTK:

```
# Import the module ngrams
from nltk.util import ngrams

# Print the bigrams and trigrams
print('bigrams:', list(ngrams(tokens, 2)))
print('trigrams:', list(ngrams(tokens, 3)))
```

```
bigrams: [('To', 'be'), ('be', 'or'), ('or', 'not'), ('not', 'to'), ('to', 'be')]
trigrams: [('To', 'be', 'or'), ('be', 'or', 'not'), ('or', 'not', 'to'), ('not', 'to', 'be')]
```

⚠️ Be careful : you need to recast as a `list` (or another container) the result of the ngrams function, since it returns a generator.

➡️ To summarize this chapter, here are the key points:

- text preprocessing can be done using NLTK
- tokenization transforms sentences or texts into words
- stopwords can be removed: they are words that do not add value
- lemmatization or stemming can be done to standardize all the words

---

3. Bag of Words

3.1 Principle

A bag of words (BOW) is just a **vector** keeping the information of **how many times each word has been encountered** in a text. It is a really simplistic approach to understand the content of a sentence or a text, and it works well for many applications.

> ⚠️ It **does not keep any information about the grammar or the order of the words** in a sentence.

Let's take an example with the following text : « Nicolas loves to watch Disney movies but everybody loves Disney movies. Pierre loves football, unlike Nicolas.»

The BOW would be:

```
BOW = {Nicolas: 2, loves: 3, to: 1, watch: 1, Disney: 2, movies: 2,
       but: 1, everybody: 1, Pierre: 1, football: 1, unlike: 1}
```

3.2 Application

Let's learn how to do our BOW using scikit-learn! To do that, we have to use the `CountVectorizer` module.

```
# Import the library
from sklearn.feature_extraction.text import CountVectorizer

# Create the input data: here we do not take care of preprocessing to ease the lecture
data = ['Nicolas loves to watch Disney movies, but everybody loves Disney movies',
        'Helene loves football, unlike Nicolas.']

# We create the output BOW, we can even reject directly the stop words and the punctuation, how magi
cal?
vectorizer = CountVectorizer(max_features=1000, stop_words='english')
BOW = vectorizer.fit_transform(data).toarray()

# Then we print the BOW
print(BOW)
```

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

What do those numbers mean ? We need to retrieve the associated vocabulary :

```
# Get the words associated to those numbers
tokens = vectorizer.get_feature_names()
print(tokens)
```

```
['disney', 'everybody', 'football', 'helene', 'loves', 'movies', 'nicolas', 'unlike', 'watch']
```

Now, we can build a dataframe keeping track of the word occurences in each sentence (a BOW is a **histogram** of word counts):

```
import pandas as pd

df = pd.DataFrame(data=BOW, columns=tokens)
df
```

	disney	everybody	football	helene	loves	movies	nicolas	unlike	watch
0	2	1	0	0	2	2	1	0	1
1	0	0	1	1	1	0	1	1	0

---

4. Regular expressions (regex)

Regular expressions (or more casually 'regex') are a powerful tool to **manipulate characters and strings**, which is often a pre-requisite for data science projects that involve text mining.

We have already used regex without even knowing it. For example, the wildcard * that allows you to find all patterns in a terminal (ex: all .txt text files) uses a regex.

```
ls *.txt
```

Regular expressions can do much (much) more than this. Someone knowing all the regex has a great power in his hands, especially when it comes to NLP where words are central. It is a whole world int itself, so this will be just a quick introduction.

In order to use regex in Python, you should import the module `re` (for regular expressions).

```
import re
```

Then you should always use the following format to write a literal string using Python:

```
r"your_regex"
```

That way, your characters like `\` will not be escaped.

> 🔨 Hint : you can use tools like **regex101** (https://regex101.com/) to test your regex patterns. Don't forger to select the python interpreter.

4.1. Basic characters

```
# We create a pattern containing the character 'a'
pattern = r"a"
```

`match()` : check if some text matches the pattern

```
# We try to see if this pattern matches another string
match = re.match(pattern, "bcde")
print(match)
```

```
None
```

```
match = re.match(pattern, "abca")
print(match)
```

```
```

`group()` : retrieve the corresponding text after a `match()`

```
print(match.group())
```

```
a
```

4.2. Sets and Ranges

```
# Let's match 'b' with a set of characters 'abc'
print(re.match(r"[abc]", "b"))

# Let's match 'd' with a range of characters from 'a' to 'e'
print(re.match(r"[a-e]", "d"))
```

```
```

Range [a-e] is equivalent to set [abcde]. This works too with numbers: range [0-9] is equivalent to set [0123456789]. Finally, if you want both lower and upper letters, you can use a set of ranges: [a-zA-Z].

```
print(re.match(r"[a-z]", "V"))
print(re.match(r"[a-zA-z]", "V"))
```

```
None
```

4.3. Special characters

Element	Description	Equivalent
.	Match any character except \n	[^\n]
\d	Match any digit	[0-9]
\D	Match non-digit characters	[^0-9]
\s	Match whitespace character	[\t\n\r\f\v]
\S	Match non-whitespace character	[^\t\n\r\f\v]
\w	Match alphanumeric character	[a-zA-Z0-9]
\W	Match any non-alphanumeric character	[^a-zA-Z0-9]

4.4. Repetitions

Quantifier	Description	Example
+	one or more	\w+
{2}	exactly 2 times	\d{3}
{1,}	one or more times	\w{1,}
{2,4}	2, 3 or 4 times	\w{2,4}
*	0 or more times	A*B
?	zero or one character to its left	\d+?

4.5. Perform matches

As you can see, `match()` only looks for a regex in the first character of a given text. Using the function `search()`, you can find anywhere in a text. The function `findall()` will return all matches as a list.

```
text = "The sky is blue only 10 minutes a day, but the sun is in our hearts."

print(re.search(r"\d", text))
print(re.search(r"\d{2}", text))
print(re.findall(r"\d+", text))
```

```
['10']
```

```
print(re.findall(r"\W+", text))
print(re.findall(r"\w+", text))
print(re.findall(r"\w{2}", text))
```

```
[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ', ', ' ', ' ', ' ', ' ', ' ', ' ', '.']
['The', 'sky', 'is', 'blue', 'only', '10', 'minutes', 'a', 'day', 'but', 'the', 'sun', 'is', 'in',
'our', 'hearts']
['Th', 'sk', 'is', 'bl', 'ue', 'on', 'ly', '10', 'mi', 'nu', 'te', 'da', 'bu', 'th', 'su', 'is', 'i
n', 'ou', 'he', 'ar', 'ts']
```

4.6. Modify strings

Finally, one of the most useful functions in the `re` module is `sub()`. This function allows to replace (substitute) a pattern by another into a text. The syntax is the following:

```
sub(pattern, what_to_replace, text)
```

```
print(re.sub(r"\sany", " many", "I don't have any friends, but I have many cats."))
```

```
I don't have many friends, but I have many cats.
```

4.7. Real life example

Now let's say we have a dataset containing tweets. We want to preprocess the textual data, but when we open our file we can observe some tweets like this one:

```
my_tweet = "Yes @Alchemy998, you're right, Trump does not know what he is talking about."
```

Of course when we have something like that, we would want to clean it with everything learnt today. But what about the mention of another twitter user ? How can we remove it ?

Just by using the `sub` method we saw:

```
re.sub(r'@\w+', '', my_tweet)
```

```
"Yes , you're right, Trump does not know what he is talking about."
```