Work Package 1 : starting with language datas

First of all, install NLTK 3.0, downloadable for free from at *https://www.nltk.org/install.html*. Follow the instructions there to download the version required for your platform.

Go on Google News and select 3 press articles (2 about the same topic and 1 really different). Copy/paste the text content of each article in 3 separate files.

The goal is to find the two nearest sentences (in a meaning/semantic way) in the articles on the same topic. The method used should also show a difference between the article on a different topic. The possible tools used to achieve this result are been presented the last session (tokenization, normalization, regular expressions, string distances).

Verify if your 3 articles respect the Zipf's law.

Here under, you will find some examples of this tools with NLTK.

Tokenization

To perform tokenization, we can import the sentence tokenization function. The argument of this function will be text that needs to be tokenized. The sent_tokenize function uses an instance of NLTK known as PunktSentenceTokenizer . This instance of NLTK has already been trained to perform tokenization on different European languages on the basis of letters or punctuation that mark the beginning and end of sentences.

Tokenization of text into sentences :

```
>>> import nltk
>>> tokenizer=nltk.data.load('tokenizers/punkt/english.pickle')
>>> text=" Hello everyone. Hope all are fine and doing well. Hope you find the book interesting"
>>> tokenizer.tokenize(text)
[' Hello everyone.', 'Hope all are fine and doing well.', 'Hope you find the book interesting']
```

Tokenization of text in other languages :

For performing tokenization in languages other than English, we can load the respective language pickle file found in tokenizers/punkt and then tokenize the text in another language, which is an argument of the tokenize() function. For the tokenization of French text, we will use the french.pickle file as follows:

```
>>> import nltk
>>> french_tokenizer=nltk.data.load('tokenizers/punkt/french.pickle')
>>> french_tokenizer.tokenize('Deux agressions en quelques jours, voilà ce qui a motivé hier matin le débrayage collège franco-britanniquedeLevallois-Perret. Deux agressions en quelques jours, voilà ce qui a motivé hier matin le débrayage Levallois. L'équipe pédagogique de ce collège de 750 élèves avait déjà été choquée par l'agression, janvier , d'un professeur d'histoire. L'équipe pédagogique de ce collège de 750 élèves avait déjà été choquée par l'agression, mercredi , d'un professeur d'histoire')
```

```
['Deux agressions en quelques jours, voilà ce qui a motivé hier matin le débrayage collège franco-britanniquedeLevallois-Perret.', 'Deux agressions en quelques jours, voilà ce qui a motivé hier matin le débrayage Levallois.', 'L'équipe pédagogique de ce collège de 750 élèves avait déjà été choquée par l'agression, janvier , d'un professeur d'histoire.', 'L'équipe pédagogique de ce collège de 750 élèves avait déjà été choquée par l'agression, mercredi , d'un professeur d'histoire']
```

Tokenization of sentences into words :

Now, we'll perform processing on individual sentences. Individual sentences are tokenized into words. Word tokenization is performed using a word_tokenize() function. The word_tokenize function uses an instance of NLTK known as TreebankWordTokenizer to perform word tokenization.

```
>>> import nltk
>>> from nltk.tokenize import TreebankWordTokenizer
>>> tokenizer = TreebankWordTokenizer()
>>> tokenizer.tokenize("Have a nice day. I hope you find the book interesting")
['Have', 'a', 'nice', 'day.', 'I', 'hope', 'you', 'find', 'the', 'book', 'interesting']
```

TreebankWordTokenizer uses conventions according to Penn Treebank Corpus. It works by separating contractions. This is shown here:

```
>>> import nltk
>>> text=nltk.word_tokenize(" Don't hesitate to ask questions")
>>> print(text)
['Do', "n't", 'hesitate', 'to', 'ask', 'questions']
```

Another word tokenizer is PunktWordTokenizer . It works by splitting punctuation; each word is kept instead of creating an entirely new token. Another word tokenizer is WordPunctTokenizer . It provides splitting by making punctuation an entirely new token. This type of splitting is usually desirable:

```
>>> from nltk.tokenize import WordPunctTokenizer
>>> tokenizer=WordPunctTokenizer()
>>> tokenizer.tokenize(" Don't hesitate to ask questions")
['Don', "'", 't', 'hesitate', 'to', 'ask', 'questions']
```

Tokenization using regular expressions(regex)

The tokenization of words can be performed by constructing regular expressions in these two ways:

- By matching with words
- By matching spaces or gaps

We can import RegexpTokenizer from NLTK. We can create a Regular Expression that can match the tokens present in the text:

```
>>> import nltk
>>> from nltk.tokenize import RegexpTokenizer
>>> tokenizer=RegexpTokenizer("[\w]+")
>>> tokenizer.tokenize("Don't hesitate to ask questions")
["Don't", 'hesitate', 'to', 'ask', 'questions']
```

Instead of instantiating class, an alternative way of tokenization would be to use this function:

```
>>> import nltk
>>> from nltk.tokenize import regexp_tokenize
>>> sent="Don't hesitate to ask questions"
>>> print(regexp_tokenize(sent, pattern='\w+|\$[\d\.]+|\S+'))
['Don', "'t", 'hesitate', 'to', 'ask', 'questions']
```

Instead of instantiating class, an alternative way of tokenization would be to use this function:

```
>>> import nltk
>>> from nltk.tokenize import regexp_tokenize
>>> sent="Don't hesitate to ask questions"
>>> print(regexp_tokenize(sent, pattern='\w+|\$[\d\.]+|\S+'))
['Don', "'t", 'hesitate', 'to', 'ask', 'questions']
```

For a list of regular expressions symbols please read :
https://www.rexegg.com/regex-quickstart.html

Conversion into lowercase and uppercase :

```
>>> text='HARdWork IS KEy to SUCCESS'
>>> print(text.lower())
hardwork is key to success
>>> print(text.upper())
HARDWORK IS KEY TO SUCCESS
```

Dealing with stop words :

NLTK has a list of stop words for many languages. We need to unzip datafile so
that the list of stop words can be accessed from nltk_data/corpora/stopwords/ :

```
>>> import nltk
>>> from nltk.corpus import stopwords
>>> stops=set(stopwords.words('english'))
>>> words=["Don't", 'hesitate','to','ask','questions']
>>> [word for word in words if word not in stops]
["Don't", 'hesitate', 'ask', 'questions']
```

The instance of nltk.corpus.reader.WordListCorpusReader is a stopwords corpus. It has the words() function, whose argument is fileid . Here, it is English; this refers to all the stop words present in the English file. If the words() function has no argument, then it will refer to all the stop words of all the languages. Other languages in which stop word removal can be done, or the number of languages whose file of stop words is present in NLTK can be found using the fileids() function:

```
>>> stopwords.fileids()
['arabic', 'azerbaijani', 'danish', 'dutch', 'english', 'finnish', 'french', 'german', 'greek', 'hungarian',
'indonesian', 'italian', 'kazakh', 'nepali', 'norwegian', 'portuguese', 'romanian', 'russian', 'spanish',
'swedish', 'turkish']
```

Example of the replacement of a text with another text

```
>>> import nltk
>>> from replacers import RegexpReplacer
>>> replacer= RegexpReplacer()
>>> replacer.replace("Don't hesitate to ask questions")
'Do not hesitate to ask questions'
>>> replacer.replace("She must've gone to the market but she didn't go")
'She must have gone to the market but she did not go'
```

The function of RegexpReplacer.replace() is substituting every instance of a replacement pattern with its corresponding substitution pattern. Here, must've is replaced by must have and didn't is replaced by did not , since the replacement pattern in replacers.py has already been defined by tuple pairs, that is, (r'(\w+)\'ve', '\g<1> have') and (r'(\w+)n\'t', '\g<1> not') .
We can not only perform the replacement of contractions; we can also substitute a token with any other token.

Performing substitution before tokenization :

```
>>> import nltk
>>> from nltk.tokenize import word_tokenize
>>> from replacers import RegexpReplacer
>>> replacer=RegexpReplacer()
>>> word_tokenize("Don't hesitate to ask questions")
['Do', "n't", 'hesitate', 'to', 'ask', 'questions']
>>> word_tokenize(replacer.replace("Don't hesitate to ask questions"))
['Do', 'not', 'hesitate', 'to', 'ask', 'questions']
```

Lemmatization

Lemmatization is the process in which we transform the word into a form with a different word category. The word formed after lemmatization is entirely different. The built-in morphy() function is used for lemmatization in WordNetLemmatizer. The inputted word is left unchanged if it is not found in WordNet. In the argument, pos refers to the part of speech category of the inputted word. Consider an example of lemmatization in NLTK:

```
>>> import nltk
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer_output=WordNetLemmatizer()
>>> lemmatizer_output.lemmatize('working')
'working'
>>> lemmatizer_output.lemmatize('working',pos='v')
'work'
>>> lemmatizer_output.lemmatize('works')
'work'
```

The WordNetLemmatizer library may be defined as a wrapper around the so-called WordNet corpus, and it makes use of the morphy() function present in WordNetCorpusReader to extract a lemma. If no lemma is extracted, then the word is only returned in its original form. For example, for works , the lemma returned is the singular form, work .
Let's consider the following code that illustrates the difference between stemming and lemmatization :

```
>>> import nltk
>>> from nltk.stem import PorterStemmer
>>> stemmer_output=PorterStemmer()
>>> stemmer_output.stem('happiness')
'happi'
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer_output=WordNetLemmatizer()
>>> lemmatizer_output.lemmatize('happiness')
'happiness'
```

In the preceding code, happiness is converted to happi by stemming.
Lemmatization doesn't find the root word for happiness , so it returns the word happiness.

Similarity measure

```
>>>import nltk
>>>from nltk.metrics import *
>>>edit_distance("relate","relation")
3
>>>edit_distance("suggestion","calculation")
7
```

Applying similarity measures using Jaccard's Coefficient.
Jaccard's coefficient, or Tanimoto coefficient, may be defined as a measure of the overlap of two sets, X and Y.
It may be defined as follows:

- Jaccard(X,Y)=|X∩Y|/|XUY|
- Jaccard(X,X)=1
- Jaccard(X,Y)=0 if X∩Y=0

Example :

```
>>> import nltk
>>> from nltk.metrics import *
>>> X=set([10,20,30,40])
>>> Y=set([20,30,60])
>>> print(jaccard_distance(X,Y))
0,6
```

Good to know :

For others tests, more than hundred corpus are available, provided by NLTK at :
http://www.nltk.org/nltk_data/

First steps on Deep Learning on your laptop.

Installation depends if you want to compute with a GPU or just on your CPU.
For CPU installation is more easy, but your code will run more slowly…

To proceed on CPU, please install Python, TensorFlow and Keras (in this order).

1) Install Python 3,6.

2) To install TensorFlow, start a terminal. Then issue the appropriate pip3 install command in that terminal. To install the CPU-only version of TensorFlow, enter the following command:

```
C:\> pip3 install --upgrade tensorflow
```

3) To install Keras, enter the following command:

```
C:\> pip install keras
```

For more information in order to use your GPU, please report to the following links :

https://www.tensorflow.org/install/ and https://keras.io/#installation