Basic pretreatments and text similarity

Objective: the main goal of this session is to manipulate text pretreatments and embeddings.

Exercise 1

Comparing text relying on word frequency

As seen during lecture, all texts follow a zipf's law. Can we compare 2 texts only on counting words supposing that two similar documents are sharing similar word distributions?

For this exercise, go on google news, and copy/past two similar press articles and third one with a more distant subject (but ideally with some common points). For example, you can select two articles talking about Trump and Venezuela and one about trump and Groenland.

To simplify, just copy (by hand) the text content of an article into a python variable.

Now we need to identify words on a given text. It's done with **word tokenization**.

First of all, install NLTK, downloadable for free from at https://www.nltk.org/install.html. Follow the instructions there to download the version required for your platform. (You can use google Colab cloud to simplify (requires a gmail account).

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 francobritanniquedeLevallois-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 iskept 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 only is not enough. We can use **lemmatization** in order to reduce superficial dissimilarities.

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.

Also suppressing **uppercase** letter can help :

Conversion into lowercase and uppercase :

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

Withdraw **stopwords** helps also avoiding frequent uninformative 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']
```

Now we have a list of words, we can count the frequency of each one and finally compare our texts.

Exercise 2

Comparing text with static word embeddings (like word2vec)

What is the biggest limitation of the approach used on Exercise 1?

We cannot deal on different words that share a similar meaning. Word embeddings can help to easily manage similarity.

Based on the subset of words selected on exercise 1 we can go further using word embeddings.

The following code shows you how to load and use a word vector dictionary. It's a simple interface to the library Gensim that allows you to manage static word embeddings like word2vec, glove or fasttext:

https://colab.research.google.com/drive/18JSzTjCoHphVsGDuZDzEwMZn3r6Yc1-j?usp=sharing

In this context, comparing two words results on computing a cosinus similarity between two vectors. Comparing two text is then comparing two sets of vectors.

It's also possible to do clustering on a set of words/vectors and verify if a cluster representing a topic is covered by another document.