Big Data Applications in Financial Markets

Text Mining

Part I: web-scraping

Prerequisite: we need first to import two fundamental packages for scientific computing with Python (to deal with multi-dimensional array and matrices).

```
1. import numpy as np
2. import pandas as pd
```

A. Web-scraping: a first example

We select a Wikipedia page (Yohan Blake). We want to import the data directly from the website.
We begin by loading the Requests module (that allows us to send HTTP/1.1 requests using Python).

```
1. import requests
```

Now, let's try to get a webpage. We define the relevant url and the response object called r.

```
1. url = "https://en.wikipedia.org/wiki/Yohan_Blake"
2. r = requests.get(url)
```

We confirm that the URL has been correctly encoded by printing the website link:

```
1. print(r.url)
```

https://en.wikipedia.org/wiki/Yohan_Blake

The idea now is to retrieve data that exists on the url and convert it into a format that is usable for analysis (much of the information in HTML is not interesting for us).

We need to load the Python package **Beautiful Soup** that is useful for parsing HTML and XML documents.

```
1. from bs4 import BeautifulSoup
```

We read the source code of the webpage and create a Beautiful Soup object: we use "r.content" to read the content of the server's response (in the relevant encoding) and we select one parser library included in the package ('html.parser'). Note that other external parsers libraries exist such as lxml's HTML parser ("lxml"), lxml's XML parser ("lxml-xml") or html5lib ("html5lib"). They might be faster but require the installation of CPython 3.4+ or PyPy.

```
1. soup = BeautifulSoup(r.content, 'html.parser')
```

The soup object contains all of the HTML in the original document. We want to get a visual representation of the parse tree created from the raw HTML content. We use: **soup.prettify()**

```
print(soup.prettify())
```

We want to extract only the text of the Wikipedia page. To do this, we can remark that the text is always included between the tags < p > and < /p >.

However, there is a problem: when we extract the text, we got useless annotations such as [1], [2]... and so on. So to get rid of them, we have to extract them from our text using the extract.() method. To do that, we erase all the text included between the tags < sup > and < /sup >, which corresponds to the annotations.

We first extract the text:

```
our_soup = soup.find_all('p')
```

Then, we get rid of the annotations:

```
[s.extract() for s in soup('sup')]
```

Finally, we display only pure text:

```
for n in soup.find_all('p'):
    print(n.text)
```

Now that we have successfully extracted the text, we can sum up all our steps into one single function after creating our own corpus.

B. Web-scraping: define our articles using lists of urls

We build a corpus of 9 Wikipedia pages split into three categories:

- Business people/entrepreneurs: we have chosen Lloyd Blankfein, Tim Cook and Richard Branson.
- Artists/musicians: we have chosen Snoop Dogg, Dr. Dre and Rihanna.
- Athletes: Shaun White, Yohan Blake and Tom Brady.

We merge all the names in a list call “Corpus_Names”.

```
URLs_business=[]
URLs_business.append('https://en.wikipedia.org/wiki/Lloyd_Blankfein')
URLs_business.append('https://en.wikipedia.org/wiki/Tim_Cook')
URLs_business.append('https://en.wikipedia.org/wiki/Richard_Branson')

URLs_artists=[]
URLs_artists.append('https://en.wikipedia.org/wiki/Snoop_Dogg')
URLs_artists.append('https://en.wikipedia.org/wiki/Dr._Dre')
URLs_artists.append('https://en.wikipedia.org/wiki/Rihanna')

URLs_athletes=[]
URLs_athletes.append('https://en.wikipedia.org/wiki/Shaun_White')
URLs_athletes.append('https://en.wikipedia.org/wiki/Yohan_Blake')
URLs_athletes.append('https://en.wikipedia.org/wiki/Tom_Brady')

URLs=URLs_business+URLs_artists+URLs_athletes
Corpus_Names=['Lloyd Blankfein','Tim Cook','Richard Branson','Snoop Dogg','Dr. Dre','Rihanna','Sha
un White','Yohan Blake','Tom Brady']
```

Part II: cleaning up and vectorization

A. Create our main function

First, we need to import **Natural Language Toolkit** (nltk): a platform for building Python programs to work with human language data.

We need also **CountVectorizer** to convert a collection of text documents to a matrix of token counts. Word frequencies are important when working with text collection. Reducing a book to a list of word frequencies retains useful information. Treating texts as a list of word frequencies (a vector) also makes available a vast range of mathematic tools developed for studying and manipulating vectors.

In a typical corpus of words we may have many thousands of word. However, each document may only have a few hundred words. This means that the bag of words matrix will have many zeros. But some words do not convey any real significance. Words like 'i', 'me', 'my', 'his' and similar should be removed from documents before they are processed. These are called **Stop words**. We remove unimportant "stop" words from the corpus.
Eventually, we get the following code:

```
import requests
from bs4 import BeautifulSoup

import nltk
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()

vectorizer = CountVectorizer(stop_words='english')
stemmer = nltk.stem.SnowballStemmer('english')
```

The idea of stemming is a sort of normalizing method. Many variations of words carry the same meaning.

```
class StemmedCountVectorizer(CountVectorizer):
    def build_analyzer(self):
        analyzer = super(StemmedCountVectorizer,self).build_analyzer()
        return lambda doc: (stemmer.stem(w) for w in analyzer(doc))
```

We merge all the steps for web-scraping into one function:

```
def getData(url):
    r = requests.get(url)
    soup = BeautifulSoup(r.content, 'html.parser')
    main = soup.find_all("p")
    main = [x.text for x in main]
    main = " ".join(main)
    pattern = "\[[0-9]*\]"
    main = re.sub(pattern, "", main)
    vectorizer = StemmedCountVectorizer(stop_words='english')
    counts = vectorizer.fit_transform([main])
    counts = pd.Series(counts.toarray()[0],index=vectorizer.get_feature_names())
    return counts
```

B. Build our Corpus and Store it in a dataframe

```
1. import re
```

We import the Regular Expression Syntax. The function in this module are actually useful to check if a particular string matches a given regular expression.

We then append a list with the output of the function getData for each urls previously chosen. We transform the overall list into a dataframe and we print the first rows. We have now a matrix (called "df") of number that indicates words frequencies for all the article chosen:

```
1. data = []
2. for x in URLs:
3.     data.append(getData(x))
4. df = pd.concat(data,axis=1)
5. df.head()
```

For clarity, we implement the columns names (that refers the name of each Wikipedia page):

```
1. # Implement the column names
2. df.columns = Corpus_Names
3. df.fillna(0,inplace=True)
```

We then count the number of words present in each articles:

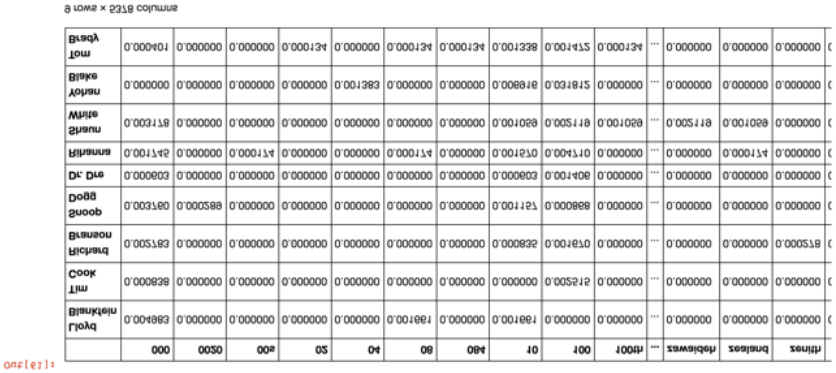
```
1. # Counting the number of words present in each articles
2. lengths = df.sum()
```

We divide each words by the length of the relevant article:

```
1. # Counting the ratio of words present in the corpus
2. df = df/lengths
3. # Tranposing the resutls
4. df = df.transpose()
```

We print the final matrix:

```
1. # Show the vectors with words representation in the DataFrame
2. df
```



Part III: Euclidian distance

Let's now compute the Euclidian distance between each pair of documents.
Euclidian Distance is the most standard metric in algebra. It is the "distance" between two points and can be measured in two dimensions with a ruler. It is a "true metric" as it satisfies four properties: non-negative value, equal to zero if the two object are identical, symmetric and it satisfy the triangle inequality. It is define as follow:

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

Where n is the number of dimensions.

Since the articles are made of a large number of tokens, there is a heavy computation in high dimension and Euclidian Distance is not the ideal metric to be used. We will see later on other useful metrics.

A. Prepare our dataframe

We import the itertools module to use a set of functions for working with iterable data sets.

```
1. import itertools
```

We then create a list, named "combos", that gives 2 length subsequence of element from the initial input "df.index". Combinations are emitted in lexicographic sort order.

```
1. combos = list(itertools.combinations(df.index, 2))
2. #itertools.combinations takes an iterable and an argument for how big the groups will be and retur
   ns all combinations
3. print(combos)
```

Eventually, we transform that list into a dataframe:

```
1. dist = pd.DataFrame(combos,columns = ["Text 1","Text 2"])
2. print(dist)
3. #Let's show it as a dataframe
```

B. Euclidean Distance function between 2 documents

We now define a function that compute the Euclidian distance between two points:

```
1. def findEucDist(t1,t2):
2.     return sum((df.loc[t1]-df.loc[t2])**2)**.5
3. #And define a distance formula
```

C. Calculate the Euclidian distance between each pair of documents

We add a column to the dataframe "dist" in which we have the Euclidian distance (we use the function apply from the panda module):

```
1. dist["Euclidean_Distance"] = dist.apply(lambda row: findEucDist(row['Text 1'], row['Text 2']), axi
   s=1)
2. #If we apply this formula across the rows, we can get the distance between each string
3. print(dist)
```

Since these numbers are close, we could also scale between 0-1 to make it even more clear. We add a column:

```
1. #Since these numbers are close, we could also scale between 0-1 to make it even more clear
2. dist["Euclidean_Distance_Normalized"] = (dist["Euclidean_Distance"]-
   dist["Euclidean_Distance"].min())/(dist["Euclidean_Distance"].max()-
   dist["Euclidean_Distance"].min())
3. print(dist)
```

Let's now sort the words according to the normalized Euclidian Distance:

```
1. print(dist.sort_values("Euclidean_Distance_Normalized"))
```

D. Comparison between Expectations and Actual distances

We expect Euclidian distances between two documents of the same group (entrepreneurs, artists and athletes) to be lower than distances between two documents of different groups.
Obviously, it is not so simple. Indeed, even if the smallest Euclidian distance is given by 2 documents of the same group (athletes: Shaun White – Yohan Blake), the next distances are found between 2 documents of different groups (entrepreneurs and athletes).

Part IV: Search Retrieval

A. Define 5 search engine queries:

Let's denote 5 search engine queries:

```
q1="Pop and hip hop artist or born in Barbuda"
q2="Influencial banker who attended Harvard and live in New York"
q3="Professional snowboarder and skater who won gold medals"
q4="CEO of a Fortune 500 company that created the iPhone"
q5="Rapper and television personality arrested for illegal possession"
```

B. Calculate the Euclidian Distance with each of documents in the corpus

We merge all the engine queries into the same variable:

```
# Create a vector with the engine queries
queries=[q1,q2,q3,q4,q5]
queries
```

We then count the words frequencies (selecting only the words that convey real significance).
We define the relevant function:

```
def corpus_add(text):
    vectorizer = StemmedCountVectorizer(stop_words='english')
    counts = vectorizer.fit_transform([text])
    counts = pd.Series(counts.toarray()[0],index=vectorizer.get_feature_names())
    return counts
```

We test that function for the first engine query:

```
# Checked it worked properly with q1
corpus_add(q1)
```

We assign to a list, called "new_data" vectors of words frequencies for each engine query.
We then assign vectors of words frequencies for each article. Finally, we transform the list into a dataframe, called "df_query".

```
new_data = []
for query in queries:
    new_data.append(corpus_add(query))
for x in URLs:
    new_data.append(getData(x))
df_query = pd.concat(new_data,axis=1)
df_query
```

Now we assign the relevant name for each columns (that refer to the engine query and the articles). We replace all the NaN value by zero and we convert the values of the matrix in fractions (between 0 and 1). We eventually transpose the dataframe and we print the final output.

```
queries_name = ["Query " + str(i) for i in range(1,len(queries)+1)]
df_query.columns = queries_name + Corpus_Names
df_query.fillna(0,inplace=True)
lengths = df_query.sum()
df_query = df_query/lengths
# Tranposing the resutls
df_query = df_query.transpose()
df_query.head(14)
```

We then create a list, named “combos_final”, that gives a 2 length subsequence of elements from the initial input “df.index”. Combinations are emitted in lexicographic sort order.

```
combos_queries = list(itertools.combinations(queries_name, 2))
combos_all = list(itertools.combinations(df_query.index, 2))
combos_final = [x for x in combos_all if x not in combos]
combos_final = [x for x in combos_final if x not in combos_queries]
#itertools.combinations takes an iterable and an argument for how big the groups will be and retur
ns all combinations
print(combos_final)
```

We re-define the function to compute the Euclidian Distance:

```
def findEucDist_Query(t1,t2):
    return sum((df_query.loc[t1]-df_query.loc[t2])**2)**.5
```

And we apply the function to the previous list:

```
dist_queries = pd.DataFrame(combos_final,columns = ["Text 1","Text 2"])
dist_queries["Euclidean_Distance"] = dist_queries.apply(lambda row: findEucDist_Query(row['Text 1'
], row['Text 2']), axis=1)
print(dist_queries)
```

We have now computed the Euclidian Distance between the two articles in which each word appears the most.

C. Sort the pairs by distance (from shortest to longest)

We sort the words according to the normalized Euclidian Distance:

```
dist_queries["Euclidean_Distance_Normalized"] = (dist_queries["Euclidean_Distance"]-
dist_queries["Euclidean_Distance"].min())/(dist_queries["Euclidean_Distance"].max()-
dist_queries["Euclidean_Distance"].min())
print(dist_queries.sort_values("Euclidean_Distance_Normalized"))
```

For the sake of clarity, we chose to create an algorithm which only displays the best match for each query (i.e the celebrity who has the minimum Euclidian distance with the considered query)

```
#We create a duplicata
ordered_dist_queries = dist_queries.sort_values("Euclidean_Distance_Normalized")

#We reset the index
ordered_dist_queries = ordered_dist_queries.reset_index(drop=True)

list_of_best_matches = []
counter = 0

for i in range(1, 6):
    counter = 0
```

```
    for k in ordered_dist_queries['Text 1']:
        if k == ("Query " + str(i)):
            list_of_best_matches.append(ordered_dist_queries['Text 2'][counter])
            break
        counter += 1
```

D. Comments on results

The results of our previous algorithm are crystal clear:

Output:

```
['Pop and hip hop artist or born in Barbuda', 'Influencial banker who attended Harvard and live in New York', 'Professional snow
boarder and skater who won gold medals', 'CEO of a Fortune 500 company that created the iPhone', 'Rapper and television personal
ity arrested for illegal possession']

The celebrities corresponding to each query are respectively:

['Dr. Dre', 'Lloyd Blankfein', 'Shaun White', 'Tim Cook', 'Snoop Dogg']
```

Conclusion: For each query, the minimum distance is found for the right person (for instance, the minimum distance for the query "Professional snow boarder and skater who won gold medals" is well Shaun White). Then everything is well settled.

Part V: Alternative Distance Function – "Cosine Similarity"

Cosine Similarity is a distance measure typically used in textual analysis as an alternative to the standard Euclidean distance.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

In textual analysis, the Cosine Similarity is preferable over Euclidian Distance because it accounts for the length of the vectors. Let's take the following example:

We have a Wikipedia page A of 1,000 words and a Wikipedia page B of 10,000 words. Both pages are related to basketball. Due to their different sizes, the word "basketball" will appear more in page B than page A: then page B will be considered as more closely related to Basketball than page A, whereas it might not be the case at all. The strength of the Cosine Similarity is to take into account this weakness and to return more robust results by correcting for different lengths of vector (i.e Wikipedia pages).

A. Cosine Similarity function between 2 documents

We now define a function that compute the Cosine similarity between two points:

```
1. def findCosSim(t1,t2):
2.     num = sum(df.loc[t1]*df.loc[t2])
3.     den1 = sum((df.loc[t1])**2)**.5
4.     den2 = sum((df.loc[t2])**2)**.5
5.     return num/(den1*den2)
6. #And define a distance formula
```

B. Calculate & comment the Cosine Similarity between each pair of documents

As in Part III Section C, we first add a column which contains the Cosine Similarity and then a new one which contains the same distance normalized. Finally, we sort (in descending order) the words according to the normalized Cosine Similarity.

```
1. dist["Cosine_Similarity"] = dist.apply(lambda row: findCosSim(row['Text 1'], row['Text 2']), axis=
   1)
2. #If we apply this formula across the rows, we can get the distance between each string
3. print(dist)
4. #Since these numbers are close, we could also scale between 0-1 to make it even more clear
5. dist["Cosine_Similarity_Normalized"] = (dist["Cosine_Similarity"]-
   dist["Cosine_Similarity"].min())/(dist["Cosine_Similarity"].max()-
   dist["Cosine_Similarity"].min())
6. print(dist.sort_values("Cosine_Similarity_Normalized",ascending=False))
```

We expect Cosine Similarities between two documents of the same group (entrepreneurs, artists and athletes) to be larger than Cosine Similarities between two documents of different groups, and we expect better results than those given by Euclidean Distance, as we know that in textual analysis, the Cosine Similarity is preferable over Euclidian Distance.

The largest Cosine Similarity is given by 2 documents of the same group (entrepreneurs: Lloyd Blankfein - Tim Cook), and the next Cosine Similarities are found between 2 documents the same group (artists), and this is what we expected. Moreover, even if those results are probably not perfect, they seem better than those given by the Euclidean Distance.

C. Search Retrieval – calculate & comment the Cosine Similarity with each of documents in the corpus

As in Part IV, we use our 5 search engine queries, we then calculate the Cosine Similarity with each of documents in the corpus and finally we sort the pairs by distance (from shortest to longest).

```
1.  def findCosSim_Query(t1,t2):
2.      num = sum(df_query.loc[t1]*df_query.loc[t2])
3.      den1 = sum((df_query.loc[t1])**2)**.5
4.      den2 = sum((df_query.loc[t2])**2)**.5
5.      return num/(den1*den2)
6.  #And define a distance formula
7.  dist_queries["Cosine_Similarity"] = dist_queries.apply(lambda row: findCosSim_Query(row['Text 1'],
     row['Text 2']), axis=1)
8.  print(dist_queries)
9.  dist_queries["Cosine_Similarity_Normalized"] = (dist_queries["Cosine_Similarity"]-
    dist_queries["Cosine_Similarity"].min())/(dist_queries["Cosine_Similarity"].max()-
    dist_queries["Cosine_Similarity"].min())
10. print(dist_queries.sort_values("Cosine_Similarity_Normalized",ascending=False))
```

As for Euclidean Distance, results are crystal clear and exactly what we should find: for each query, the maximum similarity is found for the right person (for instance, the maximum similarity for the query "Professional snow boarder and skater who won gold medals" is well Shaun White).

D. Comparison Between Euclidean Distance and Cosine Similarity

Theoretically, we knew that in textual analysis, the Cosine Similarity is preferable over Euclidian Distance because it accounts for the length of the vectors. We have confirmed this fact practically by calculating both Euclidean Distances and Cosine Similarities between pairs of documents and comparing them.

Even if for those 2 methods the results of our particular queries were perfect, we may think because of what we said previously that for harder queries, Cosine Similarity would give better results than Euclidean Distance.

Nevertheless, we should keep in mind that some other measures exist and may be preferable given the particular problem at hand.