Text Mining and Natural Language Processing

Fundamental Concepts in Text Processing and Information Retrieval

Apostolos N. Papadopoulos, Antoine Tixier, Michalis Vazirgiannis

In this lab, we are going to use the Python programming language to get some experience on basic text manipulation techniques that we have seen in class. In particular, we will use the Anaconda distribution and the **Spyder** Development Environment to illustrate some useful techniques that we can apply using Python and its related libraries for text processing.

1 Setting-up the environment

To work in this lab, you need to bring your own laptop. In addition, you must download and install the **Anaconda distribution** for **Python 2.7**. Anaconda comes with many useful libraries pre-installed. Moreover, we will need the **TextBlob library**. Use the following URL for relevant installation instructions:

http://textblob.readthedocs.org/en/latest/install.html

TextBlob requires the NLTK library (used for natural language processing), which is already contained in the Anaconda distribution.

2 Computing TF-IDF in document collections

In this part of the lab, we will learn how we can compute TF-IDF weights for words in a document collection. We will build TF-IDF weights "from scratch" for educational purposes. Later, we will see that these term weights can be generated far more easily using specific Python libraries. Recall, that the TF-IDF weight computation is based on the product of two separate factors, namely the Term Frequency (TF) and the Inverse Document Frequency (IDF). The intuition behind this measure, is that a term (word) is very important if it appears many times inside a document **AND** the number of documents that the this term is present, is relatively small.

More specifically, the importance (i.e., the weight) of a term t in document d, is quantified by:

$$\text{WEIGHT}(t, d) = \text{TF}(t, d) * \text{IDF}(t)$$

There are many different expressions that may be used for the computation of TF and IDF. Here, we are going to use the following:

TF(t, d) = (number of occurrences of term t in doc d) / (number of words of d)
IDF(t) = log (N/(1+Nt)), where N is the total number of docs and Nt the number of docs containing t

In order for the product TF(t, d) * IDF(t) to be large, both factors must be large. If either (or both) is small, the product becomes small as well.

First we are going to implement the tf-idf vectors without any help from libraries, in order to understand the

way they are computed. Later, we are going to use libraries to help us out. The relevant code is located in the source file:

tf-idf.py

Open the file from **Spyder** and take a look at the contents. We will focus on the functions that are defined in the beginning of the code:

```
def tf(word, blob):
    return blob.words.count(word) / float(len(blob.words))

def n_containing(word, bloblist):
    return sum(1 for blob in bloblist if word in blob)

def idf(word, bloblist):
    ret = math.log(len(bloblist) / (1.0 + n_containing(word, bloblist)))
    if (ret < 0.0):
        return 0.0
    return ret

def tfidf(word, blob, bloblist):
    return tf(word, blob) * idf(word, bloblist)
```

Note that we are using the concept of a text blob, which is a concept similar to a string. In fact, blobs support more operations and we are using them here. However, it is not mandatory to use the **TextBlob** library when you work with Python.

In the code you can see that we define four documents and then we organize these documents in a bloblist

```
bloblist = [doc1, doc2, doc3, doc4]
```

Now, we are going to find the most important words for each document! Note that, a word may be important for one document and not important for another one.

```
for i, blob in enumerate(bloblist):
    print("Top words in document {}".format(i + 1))
    scores = {word: tfidf(word, blob, bloblist) for word in blob.words}
    sorted_words = sorted(scores.items(), key=lambda x: x[1], reverse=True)
    for word, score in sorted_words[:10]:
        print("\tWord: {}, TF-IDF: {}".format(word, round(score, 5)))
    print
```

If you run the code, you will see that **it prints 10 words per document**, annotated with the tf-idf scores.

```
Top words in document 1
        Word: python, TF-IDF: 0.02841
        Word: A, TF-IDF: 0.02841
        Word: films, TF-IDF: 0.01704
        Word: The, TF-IDF: 0.01415
        Word: the, TF-IDF: 0.01415
        Word: California, TF-IDF: 0.01136
```

```
        Word: made-for-TV, TF-IDF: 0.01136
        Word: film, TF-IDF: 0.01136
        Word: Van, TF-IDF: 0.00568
        Word: both, TF-IDF: 0.00568

Top words in document 4
        Word: hello, TF-IDF: 0.69315
```

3 Cosine similarity using the Scikit-Learn library

Let's check now the concept of similarity between two documents in a collection. This concept is crucial in search engines as well as in text mining applications. In order to do something interesting with text we need a way to quantify the similarity between two documents D1 and D2. The relevant code is located in the source file

cosine.py

First, we are going to define our document collection. Assume the following 5 (very small) documents.

```
documents = ["Euler is the father of graph theory",
             "Graph theory studies the properties of graphs",
             "Bioinformatics studies the application of efficient algorithms in biological problems",
             "DNA sequences are very complex biological structures",
             "Genes are parts of a DNA sequence"]
```

First, let's use the sickit-learn (sklearn) library to compute the tf-idf vectors of the documents:

```
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(documents)
```

Let's print the matrix:

```
print tfidf_matrix
```

The output looks like this:

```
 (0, 24)        0.358389345779
 (0, 11)        0.358389345779
 (0, 15)        0.250262619875
 (0, 9)         0.444214356446
 (0, 23)        0.297495531631
 (0, 14)        0.444214356446
 (0, 8)         0.444214356446
 (1, 12)        0.460352570237
 (1, 18)        0.460352570237
 (1, 22)        0.371409555052
 (1, 24)        0.371409555052
 (1, 11)        0.371409555052
 (1, 15)        0.259354608022
 (1, 23)        0.308303481491
…
…
```

If we need to take the tabular form of the matrix, we have to convert the matrix to an array format.

```
print tfidf_matrix.toarray()
```

[Change the code in order to get the tabular form of the tf-idf matrix]

You should see something like the following:

```
[[ 0.          0.          0.          0.          0.          0.          0.
   0.          0.44421436  0.44421436  0.          0.35838935  0.          0.
   0.44421436  0.25026262  0.          0.          0.          0.          0.
   0.          0.          0.29749553  0.35838935  0.        ]
 [ 0.          0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.37140956  0.46035257
   0.          0.          0.25935461  0.          0.          0.46035257
   0.          0.          0.          0.37140956  0.30830348  0.37140956
   0.        ]
 [ 0.35206585  0.35206585  0.          0.35206585  0.28404451  0.          0.
   0.35206585  0.          0.          0.          0.          0.
   0.35206585  0.          0.19834776  0.          0.35206585  0.          0.
   0.          0.          0.28404451  0.2357826   0.          0.        ]
 [ 0.          0.          0.33067681  0.          0.33067681  0.40986539
   0.33067681  0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.          0.
   0.40986539  0.40986539  0.          0.          0.          0.40986539]
 [ 0.          0.          0.3753856   0.          0.          0.
   0.3753856   0.          0.          0.          0.46528078  0.          0.
   0.          0.          0.26213107  0.46528078  0.          0.
   0.46528078  0.          0.          0.          0.          0.          0.        ]]
```

Let's define a matrix that can be used to compare the documents. This is called the document-document similarity matrix:

```
# define the doc-doc similarity matrix based on the cosine distance
print "This is the doc-doc similarity matrix :"
ddsim_matrix = cosine_similarity(tfidf_matrix[:], tfidf_matrix)
print ddsim_matrix
print
```

The output looks like this:

```
[[ 1.          0.42284413  0.1197833   0.          0.06560161]
 [ 0.42284413  1.          0.22963185  0.          0.0679849 ]
 [ 0.1197833   0.22963185  1.          0.09392693  0.05199311]
 [ 0.          0.          0.09392693  1.          0.24826263]
 [ 0.06560161  0.0679849   0.05199311  0.24826263  1.        ]]
```

[Check the values in the main diagonal. Can you explain the values? What else can you say about the matrix?]

4 Executing queries over a document collection (the naïve way)

Search engines can execute a user's query against a huge document collection. We are going to demonstrate this behavior using a simple example. The relevant code is contained in the source file:

queries.py

First, let's see our document collection which is an array of sentences (strings):

```
documents = ["Euler is the father of graph theory",
             "Graph theory studies the properties of graphs",
             "Bioinformatics studies the application of efficient algorithms in biological problems",
             "DNA sequences are very complex structures",
             "Genes are parts of a DNA sequence",
             "Run to the hills, run for your lives",
             "The loneliness of the long distance runner",
             "Heaven can wait til another day",
             "Road runner and coyote is my favorite cartoon",
             "Heaven can wait"] # the last document is our query
```

We are going to use the last document in the collection as our query "Heaven can wait".

The similarity measure we are going to use is the cosine similarity. Although there exist implementations in sklearn, we are going to construct a function for this purpose, to better understand the way it works. The cosine similarity function is as follows:

```
def my_cosine_similarity(vector1, vector2):
    dot_product = sum(p*q for p,q in zip(vector1, vector2))
    magnitude = math.sqrt(sum([val**2 for val in vector1])) *
    math.sqrt(sum([val**2 for val in vector2]))
    if not magnitude:
        return 0
    return dot_product/magnitude
```

Let's compute now the tf-idf vectors of the documents, as previously.

```
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(documents)
tfidf_matrix = tfidf_matrix.toarray()
```

We set our query to be the last document in the collection.

```
query_vector = tfidf_matrix[9,:]
```

We compute now the cosine similarity between the query and each doc in the collection.

```
print "Similarity among the query and the documents: "
for x in range(0,9):
    print my_cosine_similarity(query_vector, tfidf_matrix[x,:])
```

[Run the code and check the results. Try to explain the form of the query vector. Also, try to explain why there is only one non-zero similarity in the final results.]

5 Executing queries over a document collection (a <u>better</u> way)

For large document collections, matching the query terms against all documents is not the most efficient way to process the query. Most of the Information Retrieval systems utilize an *Inverted Index* to organize the corpus. The index is composed of two parts: 1) the vocabulary, which contains all unique terms in collection and 2) the posting lists, containing the occurrences of the terms in the documents. In many cases we also maintain the position of each word in the document, in order to be able to answer *phrase queries*. An example inverted index is shown next:

```
...
well        ----> {'f0006.txt': [158], 'f0000.txt': [89]}
went        ----> {'f0009.txt': [277], 'f0003.txt': [26], 'f0006.txt': [36]}
west        ----> {'f0000.txt': [3, 7]}
wherever    ----> {'f0006.txt': [35]}
whether     ----> {'f0004.txt': [353]}
white       ----> {'f0009.txt': [167], 'f0004.txt': [93]}
...
```

Executing a multi-term query using the index involves the examination of documents that may be part of the answer. Usually, we are interested in the Top-k documents that best match the query terms. Ranking is achieved by using a ranking function such as Cosine Similarity, BM25, etc. The index provides significant help in checking only candidate documents avoiding scanning the whole document collection.

Please check the code in **`inverted_index.py`** and follow the instructions in class. The document collection is composed of 10 txt files corresponding to book abstracts. First, we apply a text preprocessing and remove *stopwords* from the corpus. Then, we build the index gradually. Finally, we execute some queries: single-term, multi-term and phrase queries.

6 Latent Semantic Analysis (LSA) and Dimensionality Reduction

In this last warm-up exercise, we will use a famous technique in Information Retrieval that reveals the concepts that are hidden in the document collection. This technique is Latent Semantic Analysis (LSA) and is used frequently for dimensionality reduction in Text Mining and Information Retrieval applications. The relevant source file is:

`lsa.py`

The document collection we will use is composed of six documents:

```
documents = ["Euler is the father of graph theory",
             "Graph theory studies the properties of graphs",
             "Graph theory is cool!",
             "DNA sequences are very complex biological structures",
```

```
            "Genes are biological structures that are parts of a DNA sequence",
            "Genes are very important biological structures"]
```

Note that, three of the documents are about Graph Theory, and the rest are about Bioinformatics. First, let's create the tf-idf matrix:

```
tfidf_vectorizer = TfidfVectorizer()
A = tfidf_vectorizer.fit_transform(documents)
A = A.toarray()
```

LSA is based on Singular Vale Decomposition. So, let's apply SVD on the tf-idf matrix:

```
U, S, V = np.linalg.svd(A)
print U.shape
print S.shape
print V.shape
```

Our matrix A associates documents to terms. Matrix U, associates documents to concepts, matrix S contains the strengths of the concepts and finally matrix V associates concepts to terms. If we multiply from the right-hand side the original matrix A with the transpose of V we can achieve dimensionality reduction, converting our data to the **concept space** instead of the **term space**.

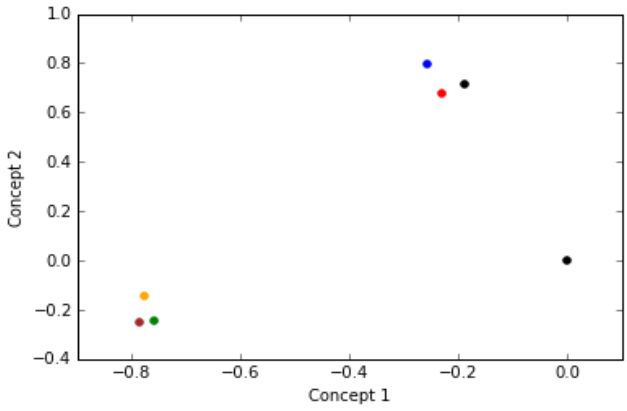
First, let's keep the first 2 concepts with the largest strength. To do this, we simply keep the first 2 lines of V.

```
V2 = V[:2,:]
```

Now, let's create the matrix M

```
M = np.dot(A,V2.transpose())
```

Each document has been converted to a point in the 2-dimensional space, since we kept only two concepts (dimensions). Next, let's create a scatter plot of the 2-dimensional data:



[Do you have any comments?]