Text Mining - Practice 1 Report

Student: David Medina Rosner

ID: F11115117

For the pre-processing part, I defined different methods for each pre-processing step:

- to_lower_case(string): takes a single string as input and returns it as lower cased.
- to_tokenize(string): takes a single string as input and returns a list of the string tokenized. I use the WordPunctTokenizer() class because it separates the punctuation from the words, and I thought that way, stopword removal, apostrophe, and punctuation removal would work better.
- to_remove_stopwords(string_list): takes a list of strings as input, and returns a list of strings with stopwords removed. For stopwords, I use the NLTK library of 179 stopwords, even though the number of stopwords is lesser than other libraries, it provided the better results for me. I tried increasing the number of stopwords by using the Scikit-learn library of 318 stopwords, but it performed worse than NLTK's.
- to_remove_apostrophe(string_list): takes a list of strings as input, and returns a list of strings with apostrophes removed. The method is fairly simple, it compares if the string is an apostrophe, if not, it stores the string in the list to return. Now you can see why I used the WordPunctTokenizer() class, as all apostrophes in the list are separated as single strings.
- to_stem(string_list): takes a list of strings as input, and returns a list of strings with its elements stemmed. For this I use the PorterStemmer() of the NLTK library because it provided the best performance, I tried implementing SnowballStemmer() and LancasterStemmer(), but they performed worse.
- to_remove_punctuation(string_list): takes a list of strings as input, and returns a list of strings with its punctuations removed. First, elements of the list that are only made of a single punctuation are removed from the list. I use the 'string.punctuation' method of the string class to define the strings I want to remove from the text. Then, punctuation is removed from within each string in the list. I use the str.maketrans() method to create a mapping table, and the translate() method to replace punctuation marks for empty strings.
- to_convert_num2words(string_list): takes a list of strings as input, and returns a list of strings with its numbers converted to words. I created this method but ended-up not using it. As when I applied the num2words step in pre-processing, my vocabulary performed worse, so I thought that maybe the numbers in the document are not that significant.
- to_preprocess(text): takes a string as input, and returns a pre-processed string. The method applies the steps of pre-processing to a text at a time. I applied the to_lower_case, to_tokenize, to_remove_stopwords, to_remove_apostrophe,

to_stem, and to_remove_punctuation methods in that order. I tried mixing different pre-processing steps, but this order is the one that performed best.

- to_preprocess_folder(folder_path): takes the direction of the folder with all the documents/queries as input, and returns two lists: one with all the pre-processed texts of the documents/queries (clean_text), and other for the name of the documents/queries files (text_names). The first part of the function is to sort the files before storing everything in list, latter I discovered that the order in which you provided query results doesn't matter. Then, I open every file in the folder by using the "with open()" method of the Path library, apply the pre-processing steps to the text, and append it to the clean_text list, and the name of the file to text_names list.

In the main() part of the pre-processing step, I called the function to_preprocess_folder() on the documents/queries folders to get the pre-processed texts.

For the TF*IDF part, I implemented the CountVectorizer()+TfidfTransformer() version. I instantiated the CountVectorizer(min_df=5) class, I specified the min_df to 5, meaning, when building the vocabulary terms that have a frequency lower than 5 are ignored. This change I made to min_df provided the best results, min_df=2 scored 0.279, and min_df=5 scored 0.28091. By doing this, I essentially reduced the vocabulary to more meaningful terms, without providing a value to min_df, my vocabulary was of 7160 terms, with min_df=2 I got 3886 terms, and with min_df=5 I got 2099 terms. So, I built the vocabulary, and create a document-term matrix by using the method fit_transform() on the clean_documents . Then I instantiate the TfidfTransformer(), that can transform the matrix of the vectorizer to a TF*IDF representation, and call fit_transform() to the document-term matrix of the vectorizer to get a TF*IDF matrix. Finally, I transform the queries to a vector matrix using the transform() method on vectorizer, and for the TF*IDF I also call transform() for the tfidf_transformer.

For the cosine similarity, I apply the cosine_similarity method to the queries and documents matrixes, after the vocabulary and the TF*IDF matrix have being created.

After that, I rank the top 10 most similar documents to the corresponding queries, and create a .csv by using the Pandas library for the results.