Final Project Report of Information Storage and Retrieval

Amul Neupane

The final project of information retrieval and storage was aimed to give us practical knowledge on Tokenization, Normalization, Common words removal (and/or dropping of stop lists), Indexing: term document incidence matrix or inverted indexing, Relationship betIen the terms, Grouping (or clustering) based on the term similarity. That uses both hierarchical clustering and k-means clustering algorithms using python as programming language.

For this project we were given following input files.

Input Files:

1. all-topics-strings.lc.txt
2. big.txt
3. cat-descriptions_120396.txt
4. feldman-cia-worldfactbook-data.txt

At first, I accumulated all the text files from folder “texts” into a large corpus. The implemented code is:

```
file = glob.glob(os.path.join(os.getcwd(), "texts", "*.txt"))
datas = []
for text in file:
    with open(text) as f:
        txt = f.read()
        datas.append(txt)
```

I then tokenized the corpus. Where tokenization is Given a character sequence and a defined document unit, **tokenization** is the task of chopping it up into pieces, called tokens, perhaps at the same time throwing away certain characters, such as punctuation. Here is an example of tokenization:

Input: Friends, Romans, Countrymen, lend me your ears;

Output: Friends | Romans | Countrymen | lend | me | your | ears

The code implemented is:

```
def process_word(text, stem = True) :
    Table = str.maketrans(dict.fromkeys(string.punctuation))
    Text = text.translate(Table)
    tokens = word_tokenize(Text)
    tokens= [word for word in tokens if word.isalpha()]
    stopwords = nltk.corpus.stopwords.words('english')
    tokens = [w for w in tokens if not w in stopwords]
    if stem:
        stemmer = SnowballStemmer("english")
        tokens = [stemmer.stem(word) for word in tokens]
    return tokens
```

Then I went through the process of **normalization** where normalization is the process of canonicalizing tokens so that matches occur despite superficial differences in the character sequences of the tokens. The most standard way to normalize is to implicitly create equivalence classes , which are normally named after one member of the set. For instance, if the tokens anti-discriminatory and antidiscriminatory are both mapped onto the term antidiscriminatory, in both the document text and queries, then searches for one term will retrieve documents that contain either. This process included stemming and lemmatization as well. The code is given above.

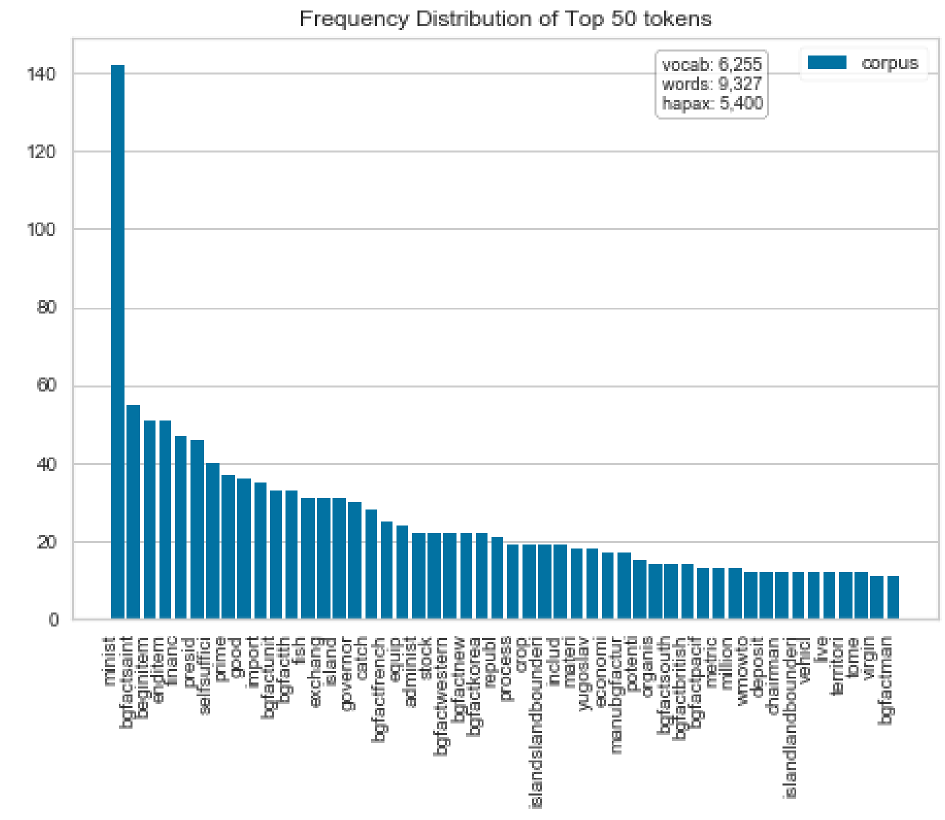
I then dropped or **eliminated stop words**. The code for that is:

```
stopwords = nltk.corpus.stopwords.words('english')
    tokens = [w for w in tokens if not w in stopwords]
    if stem:
        stemmer = SnowballStemmer("english")
        tokens = [stemmer.stem(word) for word in tokens]
    return tokens
```

I then created tfidf matrix to calulate **term frequency and inverse document frequency**. Code for that is:

```
tfidf_vectorizer = TfidfVectorizer( tokenizer= process_word,
                                   max_df=0.4,
                                   min_df=0.1, stop_words='english',
                                   use_idf=True)
vectorizer = CountVectorizer(tokenizer=process_word,
                             max_df = 0.4,
                             min_df = 0.1,
                             stop_words = 'english')
tfidf_matrix = tfidf_vectorizer.fit_transform(datas)

tf_model = vectorizer.fit_transform(datas)
total_tf = [sum(x) for x in zip(*tf_model.toarray())]
terms = tfidf_vectorizer.get_feature_names()
term_list = []
popTerm = []
pt = []
freq = []
cloud_word = {}
for i,v in enumerate(total_tf):
    if v > 25:
        freq.append(v)
        popTerm.append(i)
        term_list.append(terms[i])
        cloud_word[terms[i]] = v
    pt.append(tf_model.transpose().toarray()[i])
print(freq)
print(term_list)
```

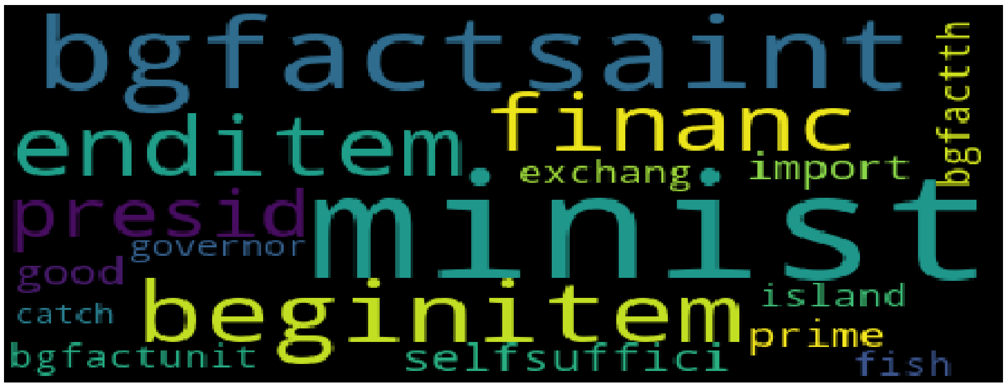
After that I **plotted term frequency histogram** of top 50 terms whose frequency is greater than 25 wsing frequency distribution visualizer. The code is above.



I then plotted **word cloud / frequencies** for which the code is given below.

```
visualizer = FreqDistVisualizer(features=terms, orient='v')
visualizer.fit(tf_model)
visualizer.show()

wordcloud = WordCloud(normalize_plurals= False).generate_from_frequencies(cloud_word)

plt.imshow(wordcloud)
plt.axis("off")
plt.show()
```

It gave us the output as:

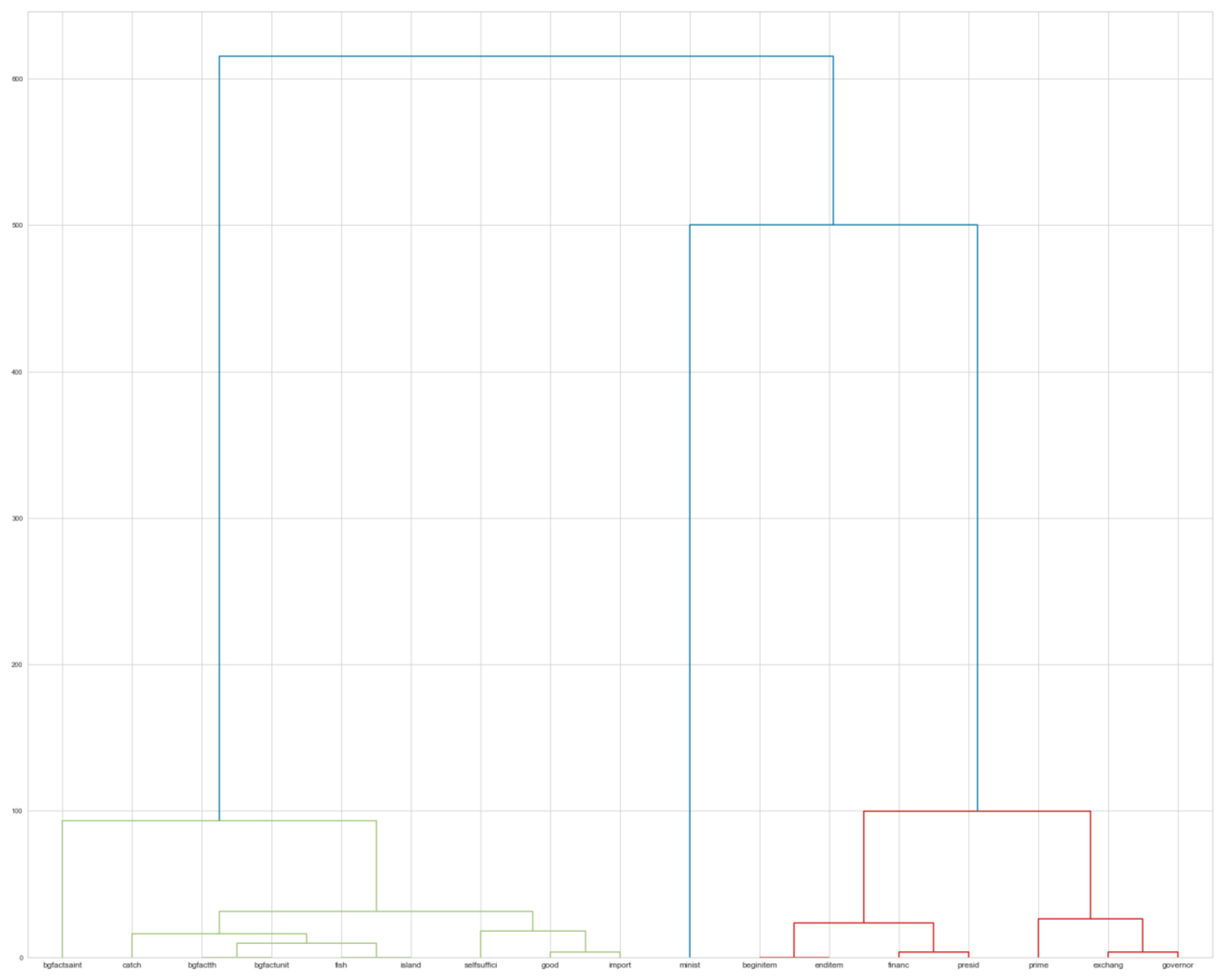


For **Grouping (or clustering) based on the term similarity**. I used both **hierarchical clustering** and **k-means clustering algorithms .**

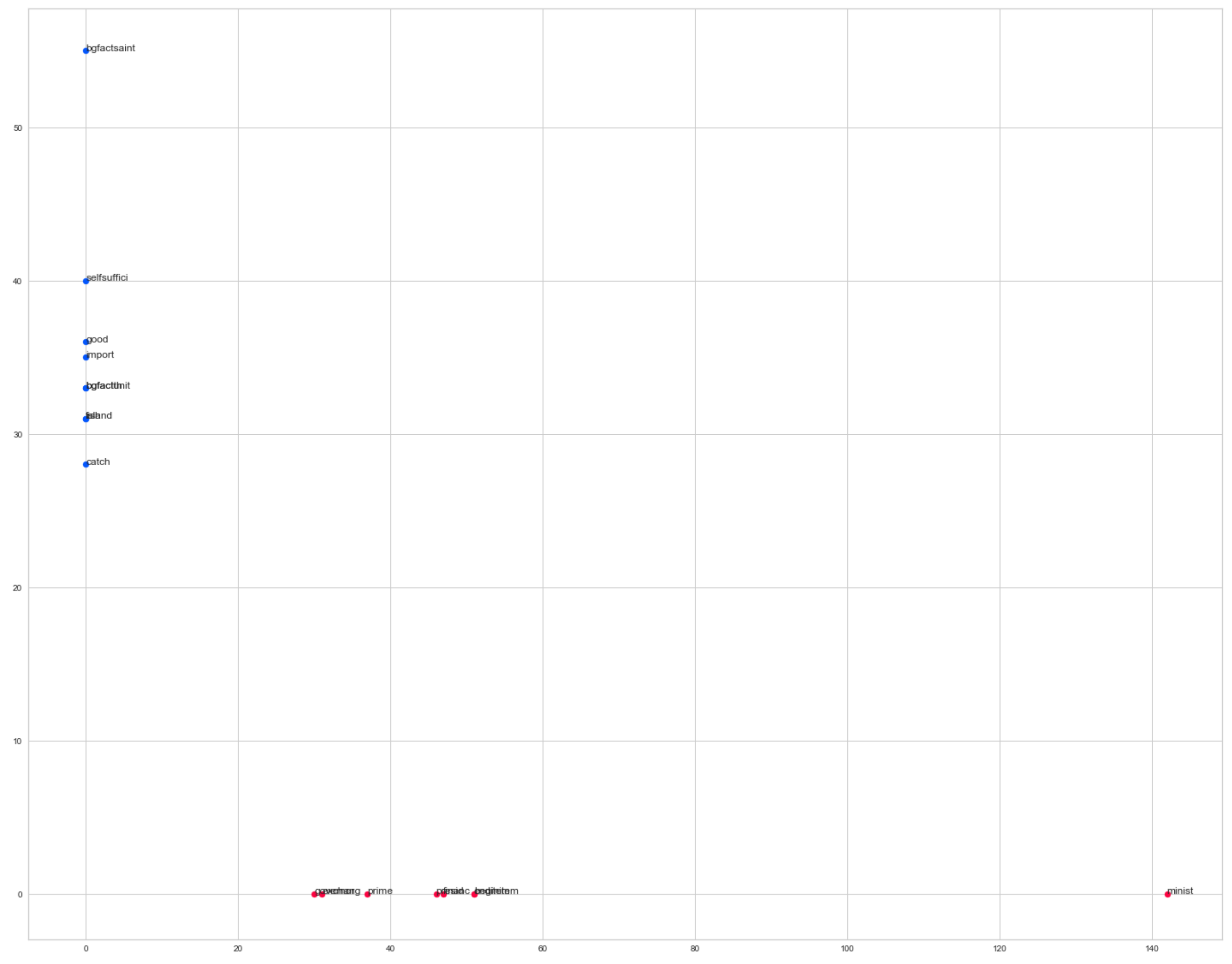
The code used to plot that is :

```
km = KMeans(n_clusters= 2)
km = km.fit(tf_model.transpose())
print(tf_model.shape)
clusters = km.labels_.tolist()

color = [ '#0356fc' if x == 0 else '#fc0341' for x in clusters]
figure, a = plt.subplots(figsize=(25,20))


for i in popTerm:
   a.scatter(tf_model.toarray()[2][i], tf_model.toarray()[3][i], c =color[i])
   a.annotate(terms[i], (tf_model.toarray()[2][i], tf_model.toarray()[3][i]))
   print(terms[i])
plt.show()

distance_matrix = euclidean_distances(pt)
link = ward(distance_matrix)
figure, a = plt.subplots(figsize=(25,20))
a = dendrogram(link, labels= term_list, orientation='top')
plt.tick_params(\
   which='both',
   bottom='off',
   top='off',
   axis= 'x',
   labelbottom='off')
plt.tight_layout()
plt.show()
```

The output that we obtained are:



Hierarchical Clustering



K-means Clustering

Output Plots:

1. FrequencyHistogram.png
2. WordFrequencies.png
3. HierarchalClusteringPlot.png
4. KmeansClusteringPlot.png