Report of Phase Two

Steps to run the program:

You can run this program in two different ways:

1. You can run the python file using python simple.py <input_dir_path> <output_dir_path>
2. You can run the bash file using ./termweights <input_dir_path> <output_dir_path>

For both the above ways, you need to have the you need to have current working directory as the one which has python and shell files and you need to have beautifulsoup, matplotlib packages installed.

Improvements to tokenizer:

I have made a couple of adjustments to the tokenizer I have created in phase one. I have created a list of stop words and used it to remove all the stop words from my dictionaries of corpus of documents. I have also removed all the words whose length is one i.e., all the single letter words. All the words that occur only once in the entire corpus have been removed.

Having done that the resulting number of tokens for all the documents have been slightly decreased. This made my term weights more effective.

Term weighting variants used:

I have used two different tf-idf variants to calculate term weights.

a. I have computed TF-IDF as simple product of square root of term frequency and inverse document frequency.

TF-IDF = SQRT(TF) * IDF

For this method, I was required to calculate term frequency and inverse document frequency. I already have the term frequency of each token in a document from phase one. The reason for using square root of term frequency to reduce the effect that comes with under-representation of uncommon words. To calculate inverse document frequency, I need document frequency i.e., the number of documents that a particular token is appearing in. For this I maintained a two dimensioned dictionary where we have all the tokens and each token maintains own dictionary of all the documents it appears in and its frequency in each document it appears in. After doing that we have the document frequency of all the tokens. Using that we are able to calculate IDF of all the tokens using the below formula:

IDF[token] = log(length of corpus/number of documents the token appears in)

After calculating IDF of all the tokens, I calculated TF-IDF and normalized the TF-IDF by the length of documents. I have used proper sqrt of sum of squares vector normalization technique for doing the same and then I wrote the output to the generated output file.

b. I have also computed TF-IDF using BM25 and the formula we used is as follows.

BM25[token] = IDF[token] * (sqrt(TF) * (k + 1) / (sqrt(TF) + k * (1 – b + b * (length of document/average length of documents)))

We are using the square root of term frequency for the same reason as above i.e., to reduce the affect of over-representation of commonly occurring words. To calculate BM25 scores we need IDF and TF which we have already calculated in the above variant. We also need average length of documents which we calculated by summing up the length of all documents present in the corpus and dividing the value by the number of documents in the corpus. We use this particular (length of document/average length of documents) in order to normalize the impact of size of a document on a term weight. We have two tuning factors in BM25 and those are k and b. The tuning factor k is used to manage term frequency scaling which means the higher the k the more importance it gives to term frequency. Since we do not want that I have used the suggested value for the k which is 1.2. The tuning factor b is used to manage the scaling of document length which in turn means that more importance is given to shorter documents if we have a higher value of b. I have also used suggested value for the b which is 0.75.

The following is the first few lines of output I have got for the first document in the corpus.

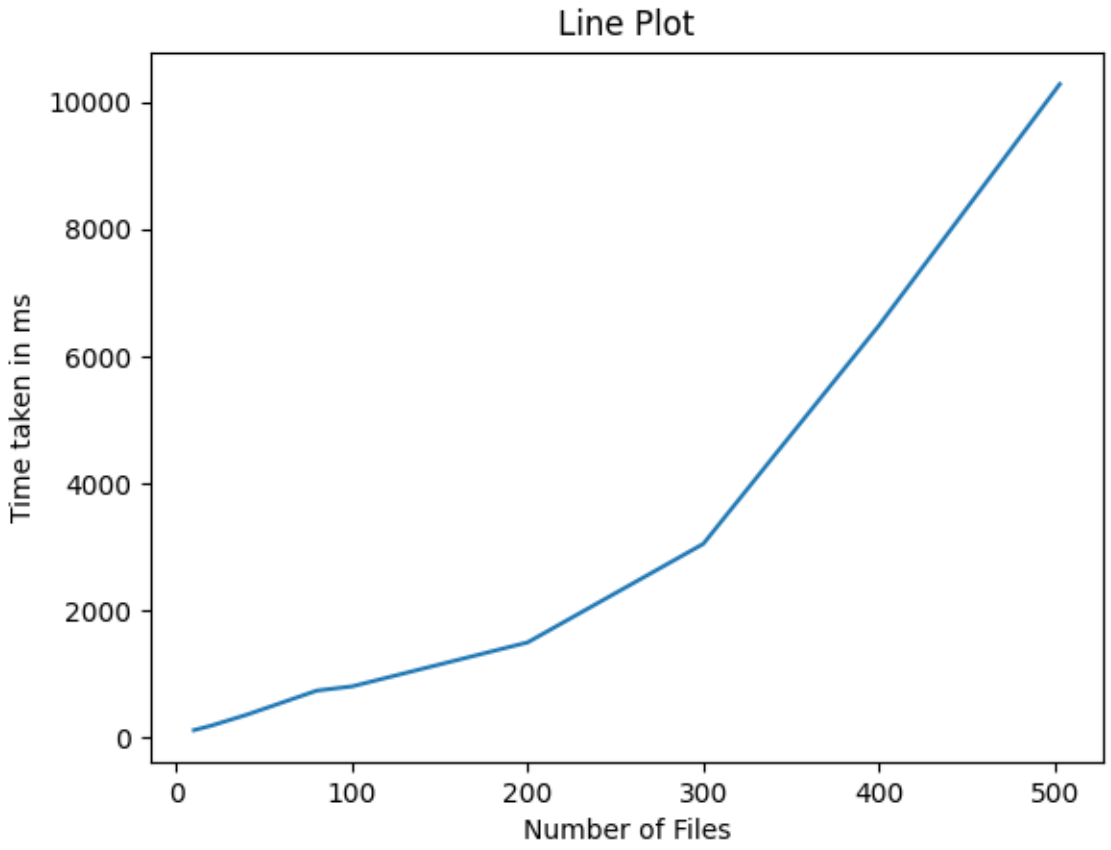
```
international :  TF-IDF  - 0.07993607669774339 |||  BM25  - 0.06294332029270962
press :  TF-IDF  - 0.11990411504661506 |||  BM25  - 0.07104226049131195
freedom :  TF-IDF  - 0.07993607669774339 |||  BM25  - 0.06294332029270962
awards :  TF-IDF  - 0.06922667309910706 |||  BM25  - 0.05978043410517448
recognition :  TF-IDF  - 0.03996803834887169 |||  BM25  - 0.04690244237131538
decades :  TF-IDF  - 0.05652334189442231 |||  BM25  - 0.055133003617036644
courageous :  TF-IDF  - 0.05652334189442231 |||  BM25  - 0.055133003617036644
journalism :  TF-IDF  - 0.05652334189442231 |||  BM25  - 0.055133003617036644
political :  TF-IDF  - 0.11304668378884462 |||  BM25  - 0.06995031279570363
pressure :  TF-IDF  - 0.03996803834887169 |||  BM25  - 0.04690244237131538
violence :  TF-IDF  - 0.03996803834887169 |||  BM25  - 0.04690244237131538
cost :  TF-IDF  - 0.05652334189442231 |||  BM25  - 0.055133003617036644
life :  TF-IDF  - 0.05652334189442231 |||  BM25  - 0.055133003617036644
associate :  TF-IDF  - 0.03996803834887169 |||  BM25  - 0.04690244237131538
committee :  TF-IDF  - 0.03996803834887169 |||  BM25  - 0.04690244237131538
protect :  TF-IDF  - 0.03996803834887169 |||  BM25  - 0.04690244237131538
journalists :  TF-IDF  - 0.14410681164656797 |||  BM25  - 0.07425142962473862
jesus :  TF-IDF  - 0.03996803834887169 |||  BM25  - 0.04690244237131538
blancornelas :  TF-IDF  - 0.17421664013428995 |||  BM25  - 0.07723888058447916
editor :  TF-IDF  - 0.03996803834887169 |||  BM25  - 0.04690244237131538
mexican :  TF-IDF  - 0.15987215339548677 |||  BM25  - 0.07592704062882331
newspaper :  TF-IDF  - 0.07993607669774339 |||  BM25  - 0.06294332029270962
zeta :  TF-IDF  - 0.13845334619821412 |||  BM25  - 0.07357905447405441
prestigious :  TF-IDF  - 0.03996803834887169 |||  BM25  - 0.04690244237131538
award :  TF-IDF  - 0.05652334189442231 |||  BM25  - 0.055133003617036644
```

Note: The above output is for displaying TF-IDF and BM25 scores at the same time. If you wish to see only one score you use the following commands to run the program.

i. TF-IDF - python simple.py <input_dir_path> <output_dir_path> “tfidf”
ii. BM25 - python simple.py <input_dir_path> <output_dir_path> “bm25”

Performance:

Number of Documents	Time in Milliseconds
10	122
20	194
40	364
80	744
100	810
200	1502
300	3051
400	6484
502	10282



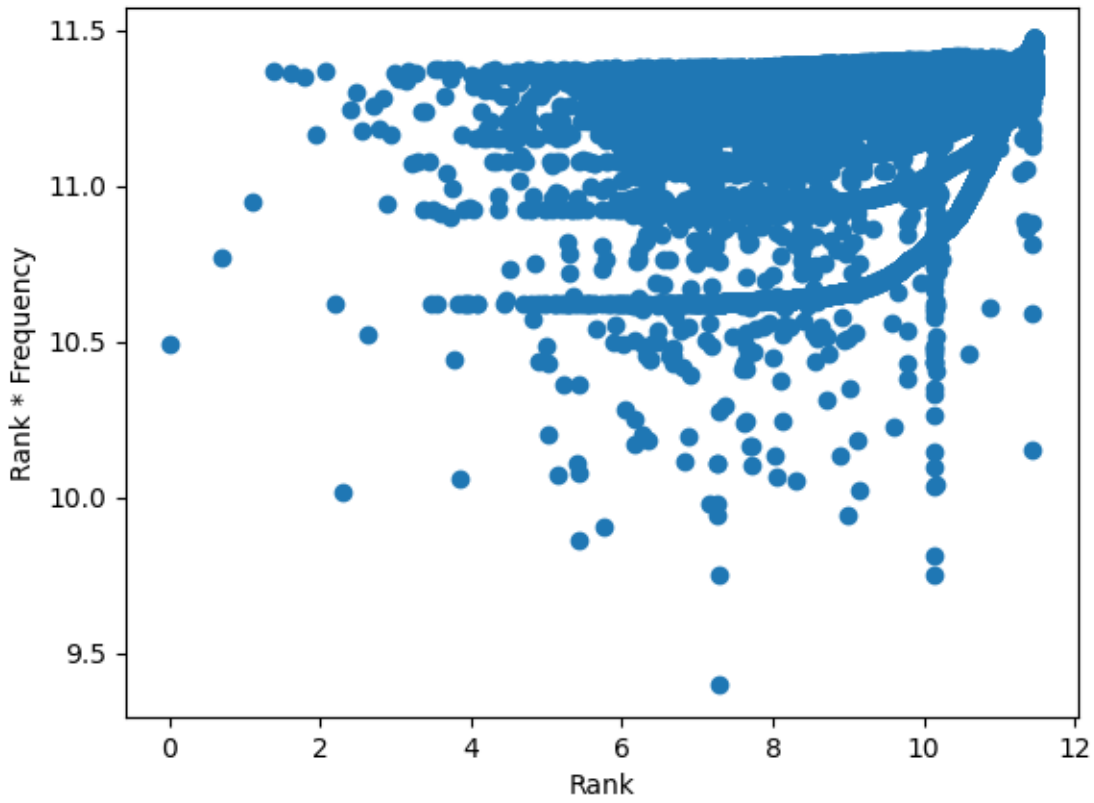
Analysis: The amount of time it takes to run the program has seen a significant raise and the reason for that is that the complexity has increased significantly. We are doing multiple reads and writes and also using a lot of temporary lists and dictionaries for our program to work effectively.

Calculating the rank times the frequency:

In the second part of the program, we calculate the rank times frequency and plot a log-log graph between rank and rank times frequency.

For this we have created a dictionary which maintains all the tokens and their ranks. We get all the ranks based on the global dictionary which we are maintaining with all the tokens in the corpus and their frequency across all the documents of the corpus.

Post that we maintain another dictionary which contains all the tokens and their rank times frequency values. Then we calculate the logs of these values and plot the graph.



From the above graph we can see that our corpus in a way follows zipf's law. We can see that the product of rank and frequency are ending up in the area of between 10.0 and 11.5 for most of the tokens which signifies that the product between a rank and the number of times the token occurs in the document is a constant.