CMSC 676 : Information Retrieval(Spring 2020)

Student Name : Ambrose Tuscano
Student ID : RE90613

Homework 2

Background:

The approach followed earlier was time consuming, due to putting the whole documents, instead of tokens in a corpos, it took much more time to execute. The time taken to execute was inversely proportional to the number of files, with the 503 files it took up over 45 minutes to get results for sorting based on frequency and tokens. So, the solution though it gave output, would never be optimal in any case. And so, I scrapped my old algorithm altogether and built on a much efficient preprocessor from scrap.

Current Approach:

In this Project, the algorithm was coded right from basics. I started coding taking a step by step approach. Each stage is explained below:

1) Input and Output Directory :

I started by taking the input files and output files folder as command line arguments from the user. Then checked if the output directory was existing, if not the code created it.

2) Set a list of Unwanted characters and Stop Words for preprocessing needs.

The project wanted to remove stop words based on a specific stop word list so the use of NLTK package as done earlier was cancelled. I used the stopwords document provided to make a list of stop words and use them further in the algorithm. Also, I made a list of basic unwanted characters which was used too during further preprocessing plans. For these two I referred to stackoverflow and geeksforgeeks blogs respectively.

3) Initialize frequency dictionaries.

4) Preprocessing :

During Preprocessing, I used the classic HTML2Text Library to change the file from the website format and tokenize it. I referred to the program creek blog for the same process and implemented it, removing links,images and other such anchors during the process. So at the end I was left with a string of just text(words), here I used the bad characters defined in step 2 to remove the format specifiers and punctuation and get just the text. I replaced the same with space so it was easier later to split the text by the empty space and thus get tokens from that. I preprocessed the tokens and removed numbers and stop words before proceeding with calculating the frequencies

.

5) Implementation :

I referred to GeeksforGeeks and initialized two hashmaps for token frequency in file and one for frequency in the document with a particular token.

Then i calculated the Term Frequency(TF) using the formula **f(w)/|D|. i.e**

TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document).

Here, I first calculated the frequency of the word appearing in the particular document(*freqdict*), also maintained one of the number of documents which had the token(word) in them(*docfreq*).

Also, now according to the formula I divided the freqdict by the total tokens of the documents to get the term frequency.

In the same way, I calculated Inverse Data Frequency (idf) using the formula **idf = |c|/df(w) i.e**

IDF(t) = log_e(Total number of documents / Number of documents with term t in it).

Total number of documents is counted earlier while maintaining a loop of files for tf calculation. Also, Number of documents with term t in it are also counted earlier so it becomes simple to get a value.I referred to the FreeCodeCamp blog for the method.

Now, that I had the formula, i looped it across the whole set of tokens and got the tfidf values, where,

tfidf[tokens] = tf[file][tokens] * idf; i.e. tf*idf for each token, in each file.

6) Time Calculation and printing :

I calculated time in values for computing file tf and tf idf for files in range [10, 20, 40, 60, 80, 100, 200, 300, 400, 500] respectively. Each file was sorted as per its TF IDF value and is printed in decreasing order of the TF IDF in the format Token and TF IDF value.

Efficiency :

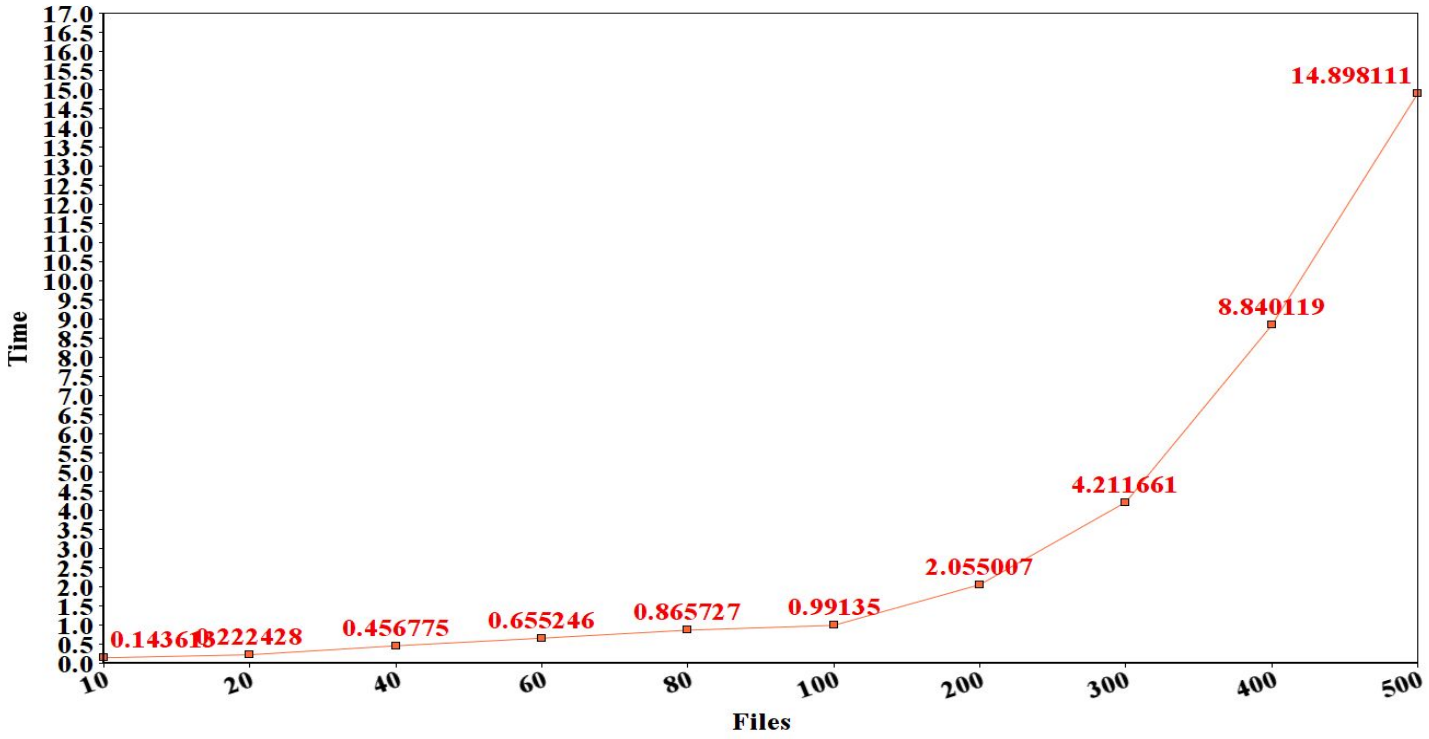
My algorithm was certainly faster than before. It took just around 14.91 second to tokenize and find the term frequencies of the 500 files. Finding the IDF and consecutively TF IDF took just 1.80 seconds. So altogether the algorithm runs in under 20 seconds. Find attached the preprocessing time.

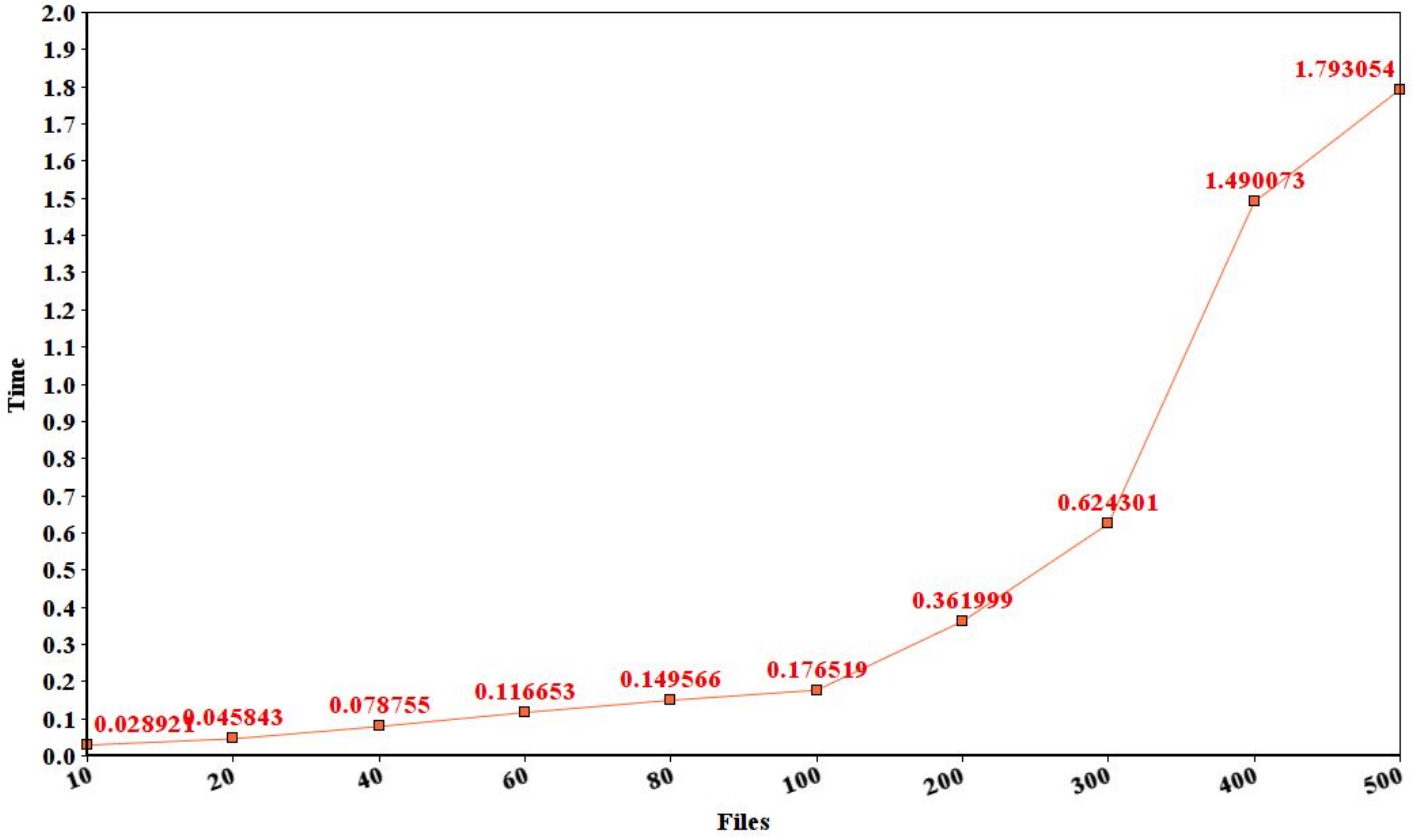
```
Time for Preprocessing:
10 0.14361262321472168
20 0.2224278450012207
40 0.4567747116088867
60 0.6552455425262451
80 0.8657269477844238
100 0.9913499355316162
200 2.055006742477417
300 4.211660861968994
400 8.840119123458862
500 14.89811086654663
```

```
Calculate weights:
10 0.028920650482177734
20 0.045842647552490234
40 0.07875466346740723
60 0.1166534423828125
80 0.1495659351348877
100 0.17651867866516113
200 0.3619992733001709
300 0.6243011951446533
400 1.4900729656219482
500 1.7930543422698975
```

Graphs:

1. **Preprocessing time and Calculating Term Frequency**





2. **IDF calculation and TF IDF Calculation**

Conclusion :

TFIDF gives us information on how relevant a token is to a particular document, which makes search engines handle query and find helpful data faster by maintaining a proper record. Here in document 25, Blancornelas has the highest TFIDF, so it can be said that chapter 25 has the most information on Blancornelas and so whenever it is searched, chap 25 can be highlighed. My approach though it is good, takes approximately 20 seconds to perform TFIDF on 500 files, so it a better approach to this is indeed needed and can be worked on by optimizing the code further.

```
blancornelas            0.15995803294542185
zeta            0.09770697190958688
tijuana         0.0888655738585677
mexican         0.08538705189283975
journalists             0.05501773888145335
felix           0.052698882944750156
newsprint               0.039481735639569956
bribe           0.03658555629594021
pri             0.03554622954342708
newspapers              0.028081295922285417
press           0.023964188700987613
monopoly                0.023663706448113733
murder          0.022036399672647068
assassinated            0.021951333777564128
baja            0.021843970841722106
in              0.020776132396153085
government              0.019003443455399793
ruffo           0.01777311477171354
bribes          0.01777311477171354
printed         0.0177477798360853
newspaper               0.01715265340132308
abc             0.01638297813129158
reporters               0.016010072229907454
columns         0.015792694255827983
cpj             0.015667032054163726
```

For TFIDF.c result for document 25

References :

https://programminghistorian.org/en/lessons/counting-frequencies
http://homepages.math.uic.edu/~jan/mcs507f11/freqdict.py
https://www.freecodecamp.org/news/how-to-process-textual-data-using-tf-idf-in-python-cd2bbc0a94a3/
https://github.com/gearmonkey/tfidf-python/blob/master/tfidf.py