Safia Shah

CMSC 476/676: Information Retrieval

C. Pearce, Spring 2022

Phase 2: Extend Preprocessor and Calculate the term weights

1. **Abstract**

This report includes a summary and extensions made to the Phase 1 program which includes removing stopwords from the given stopwords file, removing words that only occur once in the entire corpus as well as all words of length one. The report will then detail the additions made to calculate the Term Weights and display a graph containing the timings for varying numbers of documents starting from the raw input to the end of the program. The program is a memory based implementation. The command to run the Phase 2 program is as follows:

**python3 calwts.py input-dir output-dir**

1. **Report**
   1. **Preprocessing**

To start the additional preprocessing I created a Node and linkedlist class for the postings lists that would be created for each bucket in the corpus dictionary. The nodes hold the following variables: document id, frequency of token [[1]](#footnote-0)in that document, term weight.

Within the parse() I used a temporary dictionary for each file with the token as the key and the word frequency as the value. In order for a token to be appended into that temporary dictionary it had to meet Phase 1’s token conditions as well as have a length greater than one, not be a stopword, and occur more than once. At the beginning of the program the provided stopwords file was read in, the words were stripped of non-alphanumeric characters in order for proper comparison against tokens, and stored in an array. Within the same conditional statement as the previous phases’s requirements, I checked if the token was a stopword by comparing the token to the items in the stop words array and checked the length to be more than 1. Once each token for a file was read in and filtered, the temporary dictionary was iterated and the terms were appended to the main corpus dictionary.

In order to remove terms that occur only once in the entire corpus all documents were run through and the corpus dictionary was first completed. Within the editDict() I iterated through the dictionary and for each bucket checked how many postings there were. If the current tokens' postings list had only one document and the frequency of the token in that document was one, I stored the token in a temporary array, term\_array[]. Term\_array[] was iterated through to match and pop all identical buckets off the corpus dictionary once the entire corpus was iterated. Because that then changes some of the document sizes, I subtract one from the TERM\_DICT value of that corresponding document id while the corpus iteration was taking place.

* 1. **Term Weighting**

The general structure for the term weighting was modifying Phase 1’s token dictionary. Phase 2’s dictionary had the token as the key and the value being a postings list. The postings list was created using a linked list structure with each node holding the term’s frequency in the document, weight, and the document id. After preprocessing was completed for each file, the temporary dictionary for the file was appended into the global token dictionary. If a token existed in the dictionary already, using one of the linked list functions, insertAtEnd(), the new posting was inserted at the end of the token’s postings list with the correct document number and frequency. If the token was new, the link list would first be initialized for the token bucket with a new linkedlist as the value and then the information would be appended using the aforementioned linked list function.

Two functions were made in calculating the term weights. One function, computeWts(), is a member of the linkedlist class. This member function is called on by the function computeWeights(). ComputeWeights() takes in the corpus dictionary, H\_DICT, a dictionary that holds the size of a document called TERM\_DICT, the number of documents in the corpus and the name of the output directory. For each key/bucket in H\_DICT countList(), linkedlist member function, is used to return the df value of that token, and then computeWts() is called on the document which has the df, corpus and document size passed in as parameters. ComputeWts() traverses through the postings list and using the tf\*idf equation computes the weight for every posting in the current bucket. The tf\*idf function used was the following from our previous lectures:

*tf\*idf = [freq of word w in doc i / total terms in doc i] \* [num docs in corpus / num docs that contain word w]*

This variant normalizes the term weights by the length of the documents in the simplest way according to the Phase 2 outline. This process repeats for every bucket. Once all weights are computed another linkedlist member function is called to write each posting for each bucket to the correct file. The function loops through the corpus dictionary for each posting list it, traverses through postings list, opens or creates the proper output file and appends the token and term weight and closes the file. I chose the append option ‘a’ with the write() because of the multiple open and close calls . The only annoyance with using append is that if the program is run multiple times, it will add to the existing files instead of overwriting the previous runs data. But, that is not an issue related to the requirements of the phase.

* 1. **Output Files**

The output files have the surviving token and the term weight written with a precision of 8 decimal places. Each output file name is of the following format: “(doc\_id).wts” .

* **The first example is of FILENAME: “001.htm” and OUTPUT: “001.wts”**

In Phase 1 the number of tokens present in the token file totalled to 1265. Due to the extra pre-processing the total number of tokens, document size, came to 623. A significant reduction in terms. This file has some tokens that have a weight of around 3 and one as high as 15. Every other term sticks in the range between 0 and 1. Even though the terms are normalized by document size there is still a drastic difference between the weight of some tokens. The higher weights due to the occurrence of a token in more documents as well as higher term frequencies. If I had used a different tf\*idf variant then the weights may have looked more consistent.

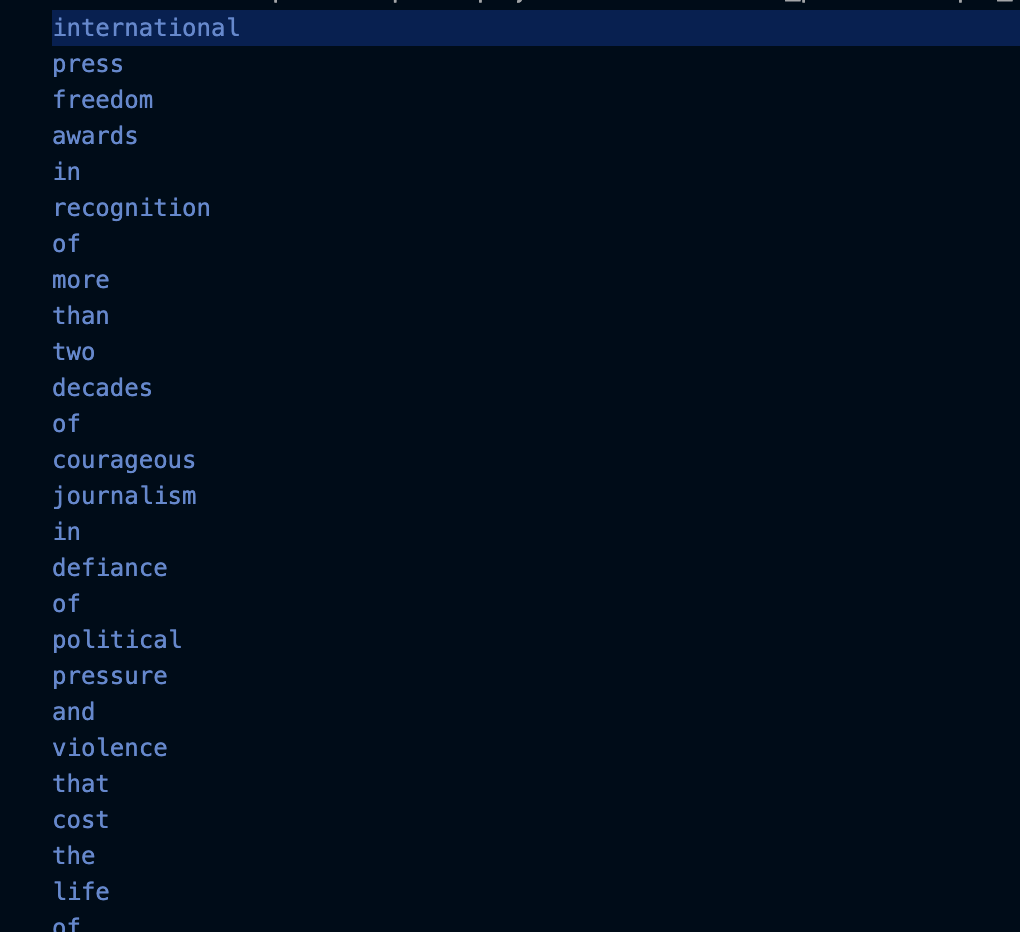
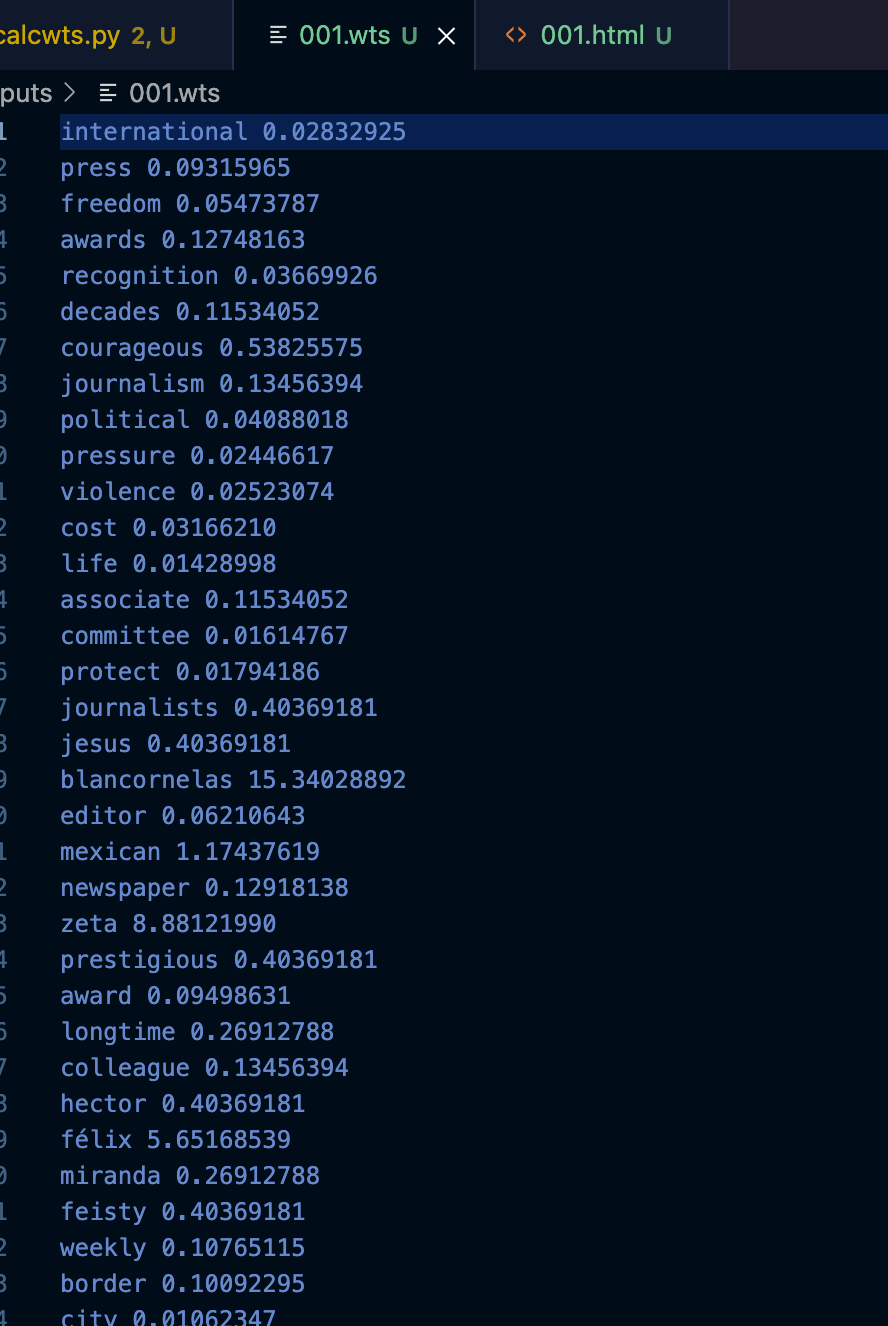


Figure 1a: “001.wts” Figure 1b: “001\_file.txt” - P1



Figure 2: “001.html”

The second example is of FILENAME: “444.html” and OUTPUT: “444.wts”

On the left was the resulting tokenized words from the Phase 1 parser I had implemented. On the right in figure 3b you can see interestingly the “textplain” token as a significantly higher tf\*idf score than every other word. Upon more research into why that is, the token occurred a total of 612 times (corroborated by the frequency given in Phase 1). It also occurred in 8 different documents many times. It seems like one of the few tokens that actually go that high above one and this is despite normalizing it by the document size and the document size being smaller than those in Phase 1 due to the extra pre-processing conditions.

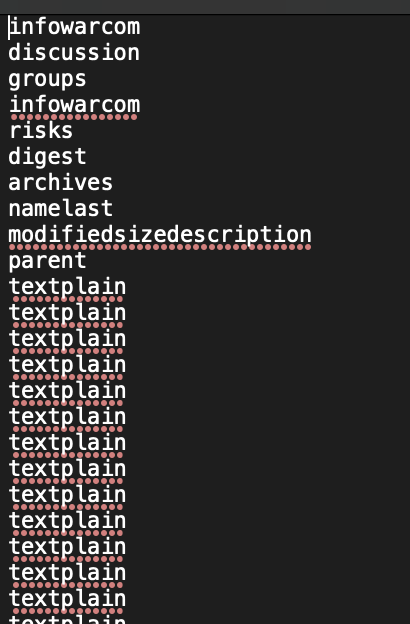


Figure 3a: 444\_file.txt Figure 3b: 444.wts

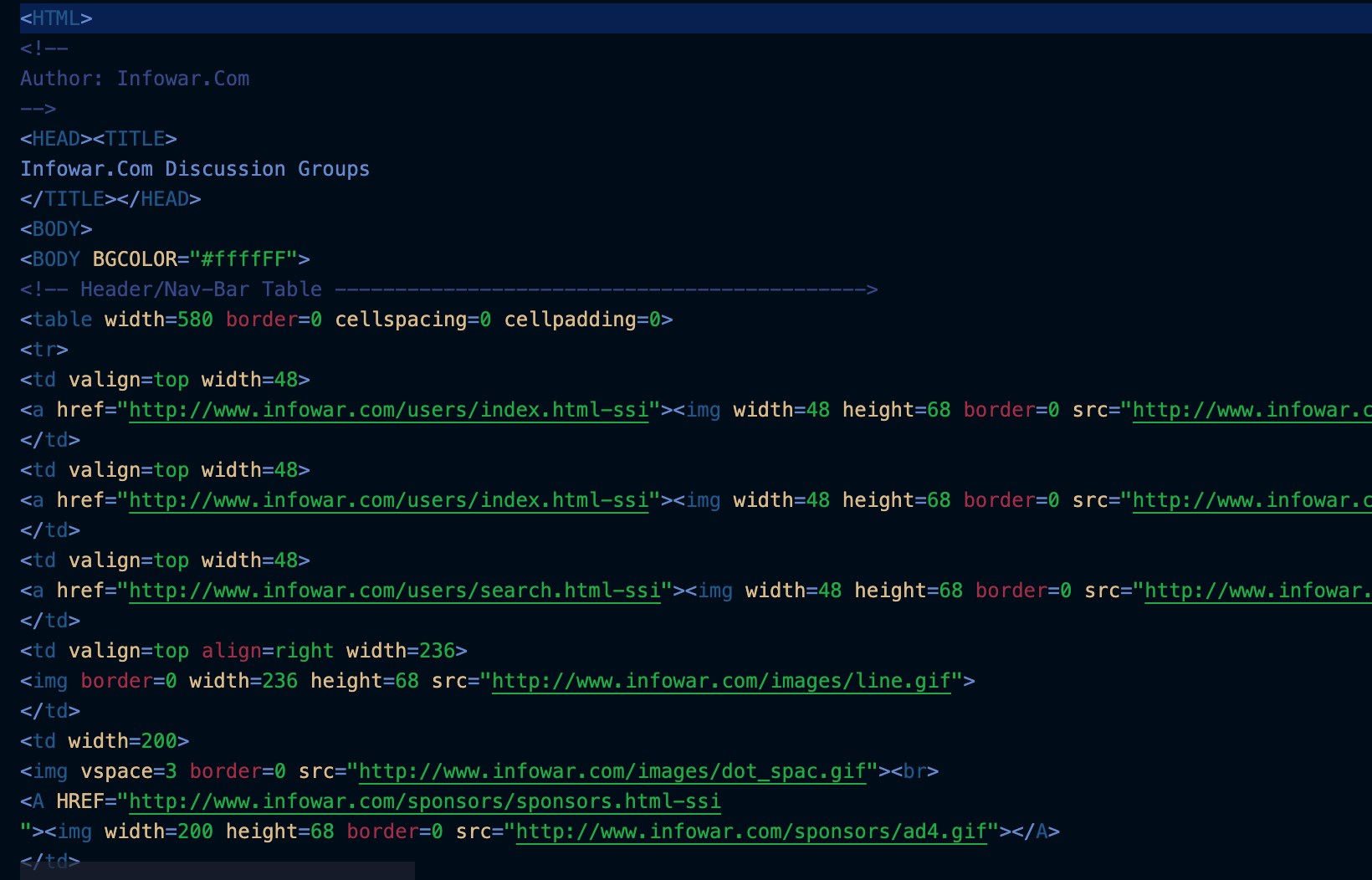
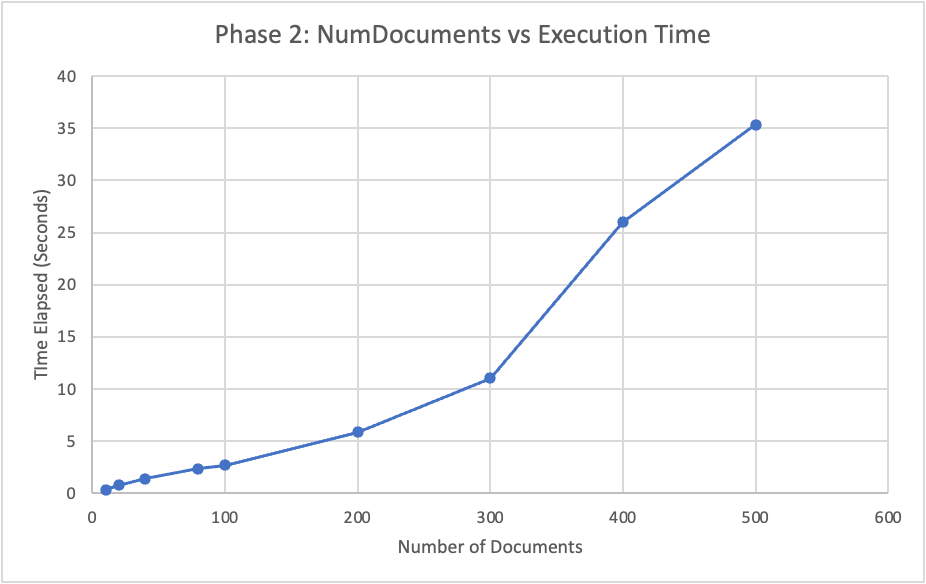


Figure 4: 444.html

1. **Graph**



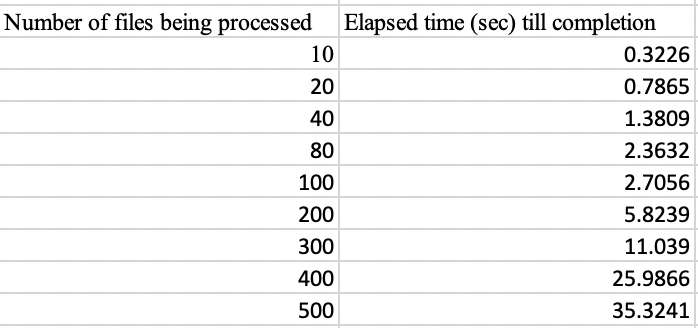


Figure 5: Efficiency Graph Figure 6: Numerical Chart version of Fig 4

Given the depiction on the number of Documents vs time graph, the data does not show much anomaly. The data shows natural progression in the time it takes to execute the program from the starting raw documents to finishing with no document interval going extremely over half a minute. What is interesting is that for each interval the time almost doubles in size for some intervals like 10 to 20 documents and 200 to 300 documents as well as 300 to 400 documents. Those increases in time occur when the document number is also almost doubled. It may also be that some documents take longer to process than others because later html files may include different languages or that some files are larger. Overall, I believe that the efficiency is pretty decent as there are no major increases in time that do not correlate to the increase in documents being processed.

1. Token and term are used interchangeably as well as linked list and postings list [↑](#footnote-ref-0)