Program Documentation:

Jai-Kishan Timmapatruni
- EE36052

Summary:

To create weighted tokenized documents I have used Eclipse as my IDE, and created a maven project because I can utilize more Java libraries. The main aim of this project is to create weighted documents using the outputs generated in homework 1 which are the tokenized files as input.

The Approach for Creating Weighted Documents:

Firstly I have made a string array to store the following stoplist we have and then by using the args[] I have passed the command line input and output paths through it.
To use different libraries of Java I had to create a Maven project which helped utilize the Jsoup library to parse in the HTML documents and get the content from them. After getting the content we tried to check if the head and body content in the HTML files are not null then we can try to combine the following data into a string by making use of the functions we have like document. body().text() and document. head().text() (Note: Here .text() has helped us convert the following content to string.). If the body is null then we only try to take the content of the head tag in the HTML file. I have created the output files with the name of the respective input files so that we can understand that the following tokens belong to the following input html file.
After getting the content in the format of a string by making use of the StringTokenizer() we can go through each token present in the string and by making use of the regular expression and .replace() function I have eliminated any unnecessary symbols if present in the string.

As per the preprocessing conditions I have eliminated terms which are of length 1 and if the following term is present in the stop list by using a if condition. If it satisfies the condition I have imported the following token into a Tree map and calculated the respective frequency for each token in each of the files.

After getting the frequencies of all the tokens in the document I tried to iterate through each key inside the Treemap to eliminate the tokens which have a frequency of 1.

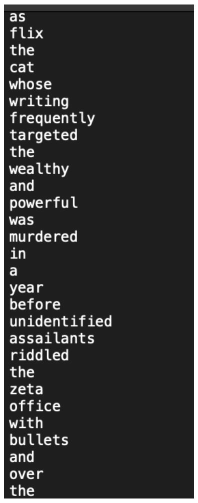
In the next step, I have called a cal_TF_IDF_Normalisation() which gets a preprocessed tree map as input and returns a tree map that contains the weights of each token in the document.

So initially we have calculated the sum of all the frequencies inside the treeMap which we have and a line has been created to get the documentfrequency which is used to store the document frequency of each term. Later for each valid entry in the stream, we increment the document frequency count for the corresponding term in the document frequency map. If the term is not present in the map, it initializes the count to 1. Here we have a keymapper function specifying that the key in the new map should be the same as in the original map. For each entry in the new map we extract the t_f value which is the term frequency. And then we calculate the normalized term frequency by dividing the term frequency by the sum of all the frequencies.
I have used the inversed document frequency method to find the i_df which is Math.log(1.0+weightedtermfrequency.size() / (double)weightedtermfrequency.size()).
Then in the end we return the product of normalized_length *i_df and compute the final TF-IDF like weight for the term in the document. I have made sure that for suppose if we get any duplicates which will not be happening but just in case I have made sure that (x1,x2)->x1, use the first value which is encountered and TreeMap::new helps in storing the results to make sure that the keys are sorted.

In the end, by using the bufferwritter I have written the following token weights of each token in their respective output files.

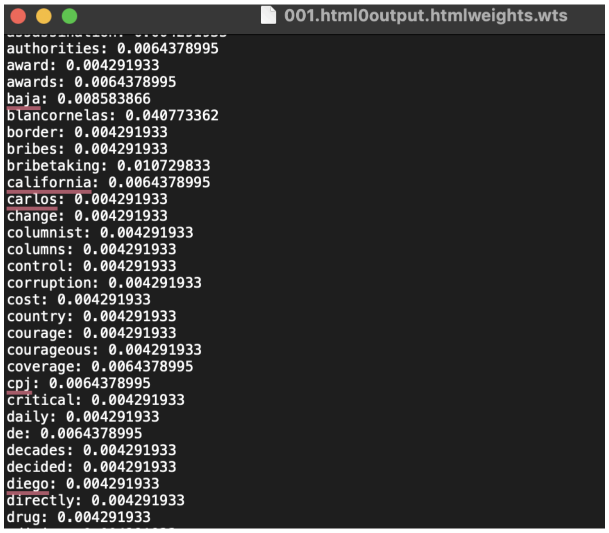
An example initially in the homework 001 file output we had words whose length is 1 and term frequency is 1 also some of the words are present in the stop list but now we have a clear set of terms with their weighted values.

Before After



```
as
flix
the
cat
whose
writing
frequently
targeted
the
wealthy
and
powerful
was
murdered
in
a
year
before
unidentified
assailants
riddled
the
zeta
office
with
bullets
and
over
the
```

——-->



001.html0output.htmlweights.wts

```
authorities: 0.0064378995
award: 0.004291933
awards: 0.0064378995
baja: 0.008583866
blancornelas: 0.040773362
border: 0.004291933
bribes: 0.004291933
bribetaking: 0.010729833
california: 0.0064378995
carlos: 0.004291933
change: 0.004291933
columnist: 0.004291933
columns: 0.004291933
control: 0.004291933
corruption: 0.004291933
cost: 0.004291933
country: 0.004291933
courage: 0.004291933
courageous: 0.004291933
coverage: 0.0064378995
cpj: 0.0064378995
critical: 0.004291933
daily: 0.004291933
de: 0.0064378995
decades: 0.004291933
decided: 0.004291933
diego: 0.004291933
directly: 0.004291933
drug: 0.004291933
```

And there is another example for the 045.html file we only had one word in it.

three frames

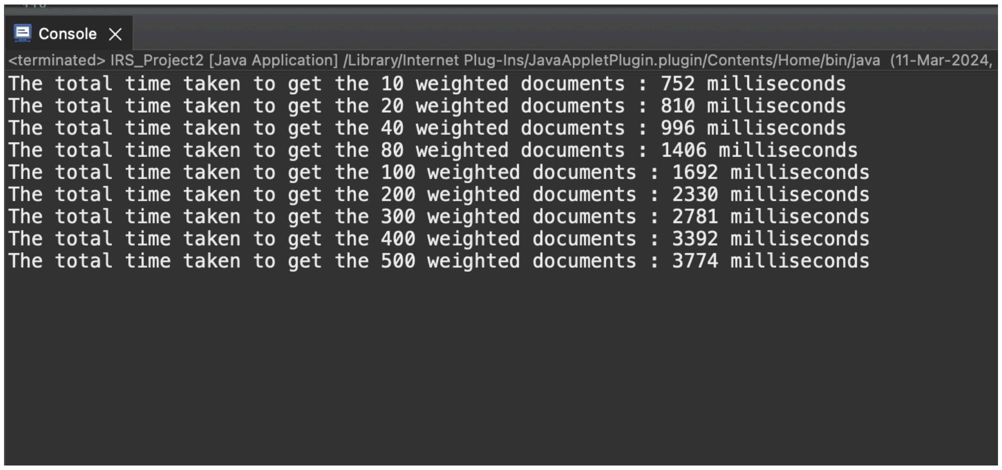
After giving the above file as input to find the tokenized weighted documents the output is an empty file because the word three frames has term frequency as 1 in the document is eliminated. And that is why we get the output of the 045 file as empty.

<u>The following files are in the Attached Folder:</u>

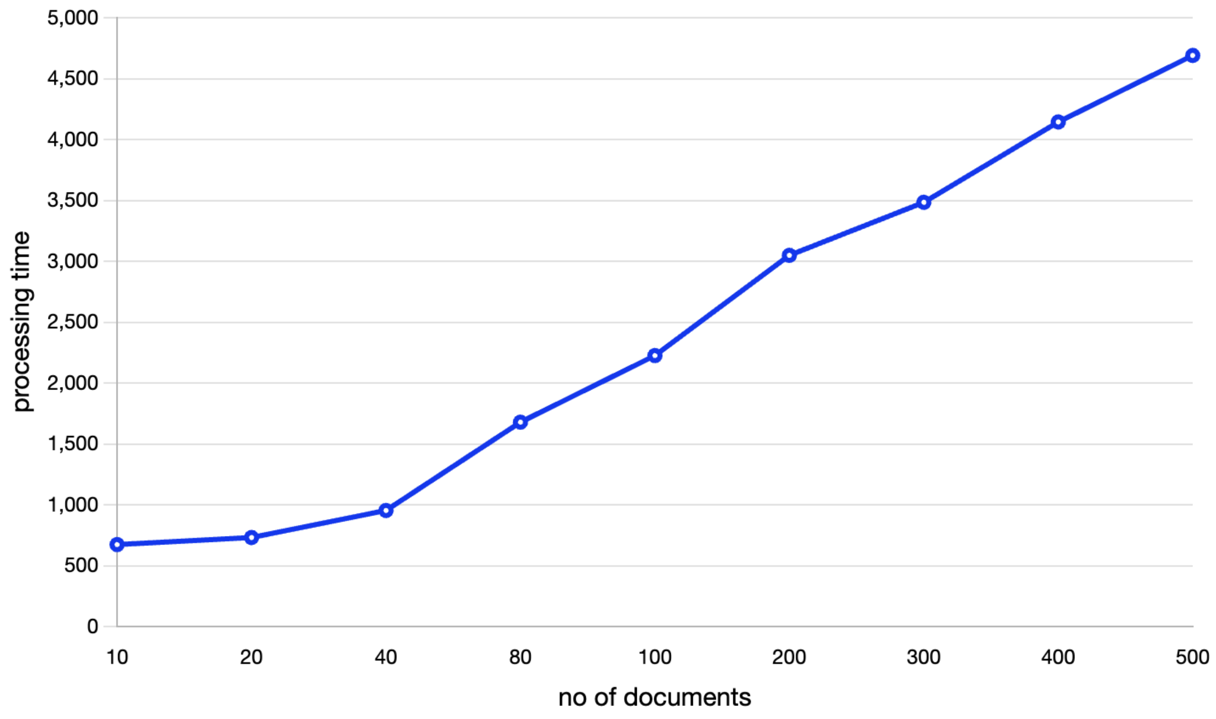
Input Directory: It contains a list of all the 503 html files.
Output Directory: Has 503 tokenized weighted documents for each input file

<u>**Execution:**</u>

By making use of the String args[] in Java I have given the directory path of the input and output files.
In Eclipse by editing the run configurations and giving the input through the command line we can run the following java file and here is the following output which is generated on the console.



```
<terminated> IRS_Project2 [Java Application] /Library/Internet Plug-Ins/JavaAppletPlugin.plugin/Contents/Home/bin/java  (11-Mar-2024,
The total time taken to get the 10 weighted documents : 752 milliseconds
The total time taken to get the 20 weighted documents : 810 milliseconds
The total time taken to get the 40 weighted documents : 996 milliseconds
The total time taken to get the 80 weighted documents : 1406 milliseconds
The total time taken to get the 100 weighted documents : 1692 milliseconds
The total time taken to get the 200 weighted documents : 2330 milliseconds
The total time taken to get the 300 weighted documents : 2781 milliseconds
The total time taken to get the 400 weighted documents : 3392 milliseconds
The total time taken to get the 500 weighted documents : 3774 milliseconds
```



Conclusion:

In the end, the code successfully does the preprocessing by removing all the tokens whose length is 1, if its respective term frequency is 1, and also if present in the stop list. Weights have been generated for each token present in the input file and the weighted document has been generated in .wts format by making use of tf-idf variants and normalization.