

CMSC 476/676 Information Retrieval

Homework 1 REPORT

JA52979

Steps to run the program:

In the IDE terminal, You can run the program described below:
`python3 tokens.py "input_direc_path" "output_direc_path"`.

Here, 'tokens.py' is the main python program. You must have the beautiful soup package installed and your current working directory containing python files in order to run the method described above.

Implementation:

I had implemented my code with my teammate Anushka Dhekne (VD19739). I used the 'BeautifulSoup', 'word_tokenize', and 'counter' libraries. For parsing the html documents, I used the 'BeautifulSoup' from the bs4 package, then after parsing each document, the parsed text is then cleaned using regular expressions. So, the non-alphabetic characters and additional symbols are handled using regular expressions. Then the cleaned text is then converted to tokens with the 'word_tokenize' library from 'nltk' package', later each word is converted to lowercase. In order to store the all tokens I maintained two lists i.e., A list of tokens for each individual file and a global list that contains all the tokens of all processed documents. In the code 'tokens' and 'all_tokens' are the two lists I used. Then to store the frequencies of each token, I made use of the 'Counter' library. So, the frequency of tokens across all documents is aggregated by the Counter object 'token_freq', which is created after processing all files.

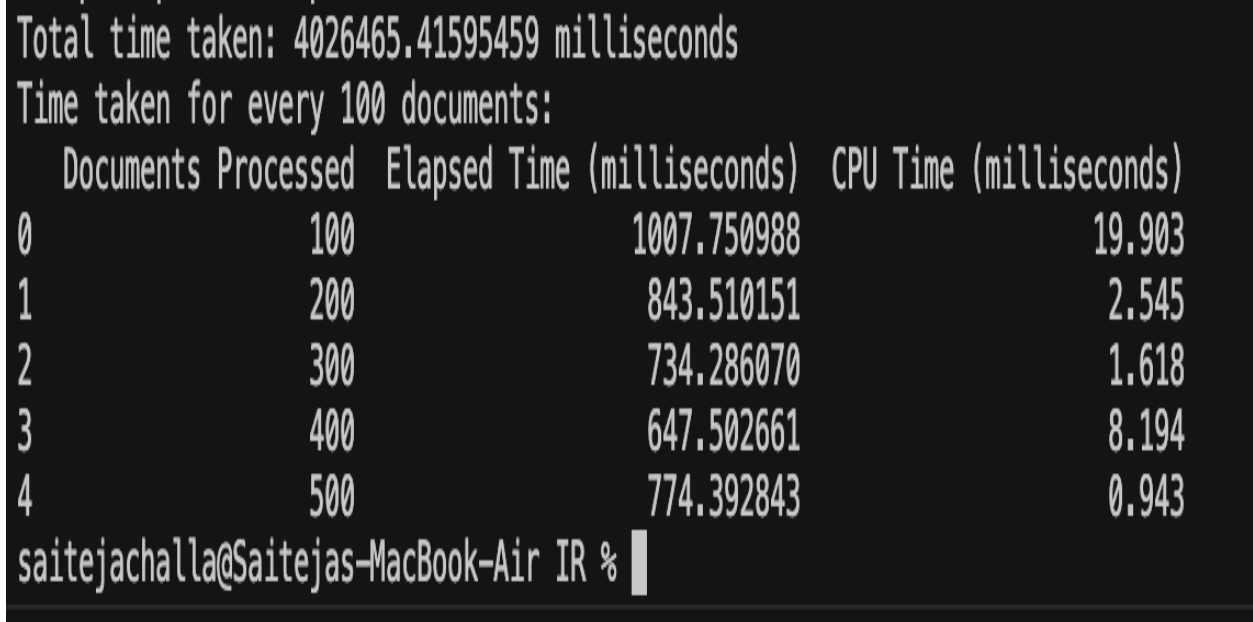
To compute overall frequencies, tokens from each document are gathered into a single list using the 'all_tokens.extend(tokens)' function. Before calculating all the tokens frequencies, I recorded the CPU process times and elapsed times for each 100 processed documents by using 'time' library. After calculating all token frequencies, the aggregated token frequencies are sorted alphabetically, and the sorted tokens and their frequencies are then written to the tokens_sorted_by_token.txt file. Similarly, the most_common() function of the Counter object is used to sort the tokens according to their frequency in descending order, creating the tokens_sorted_by_frequency.txt file. After storing the results in two separate files, for retrieving the first and last 50 lines, I used the 'colon(:)' operator. These retrieved lines are stored in two separate files. Finally, I used pandas dataframe for storing CPU and elapsed times, using the matplotlib library plotted the both time graphs. The results screenshots are displayed below.

Drawback:

One drawback I've observed that for some tokenizers is that it divides words with periods, like "U.S.A.", into distinct tokens, which in this instance are "u", "s", and "a". If the periods were removed completely, words from neighboring phrases that end up next to each other after being split on periods would be incorrectly joined. Periods break apart entities like abbreviations, but my tokenizer handled the words that contain periods. Here, I made use of the nltk 'punkt' model, where the Punkt knows that the periods in the tokens do not mark sentence boundaries. Hence, the periods are handled correctly. For example, in the document '281.html', I observed a word "J.R.Pole", which was correctly converted into "j r pole".

Performance:

The below screenshot demonstrates the times of every 100 documents processed. The cumulative sum of elapsed times and CPU times for every 100 documents are calculated.



The screenshot shows a terminal window with the following text:

```
Total time taken: 4026465.41595459 milliseconds
Time taken for every 100 documents:
  Documents Processed  Elapsed Time (milliseconds)  CPU Time (milliseconds)
0                   100             1007.750988             19.903
1                   200             843.510151             2.545
2                   300             734.286070             1.618
3                   400             647.502661             8.194
4                   500             774.392843             0.943
saitejachalla@Saitejas-MacBook-Air IR %
```

	Documents Processed	Elapsed Time (milliseconds)	CPU Time (milliseconds)
0	100	1007.750988	19.903
1	200	843.510151	2.545
2	300	734.286070	1.618
3	400	647.502661	8.194
4	500	774.392843	0.943

Graphs:

As mentioned above, the set of documents processed vs elapsed time and set of documents processed vs CPU time are plotted. So, to store all the times, I used a for loop and enumerate function in order to parse each file. Thus, after every 100 iterations, the two times are stored in two lists 'elapsed_times' and 'cpu_times' respectively. Therefore, using those lists and making use of the matplotlib library, the two different graphs are plotted, which are depicted below.

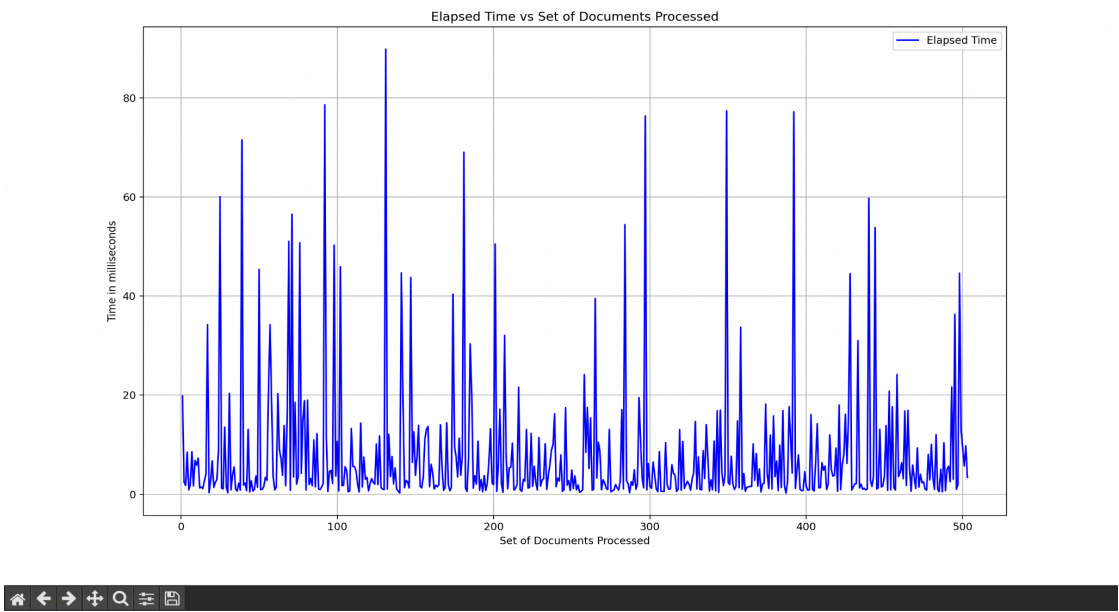


Fig-1: set of documents processed vs elapsed time

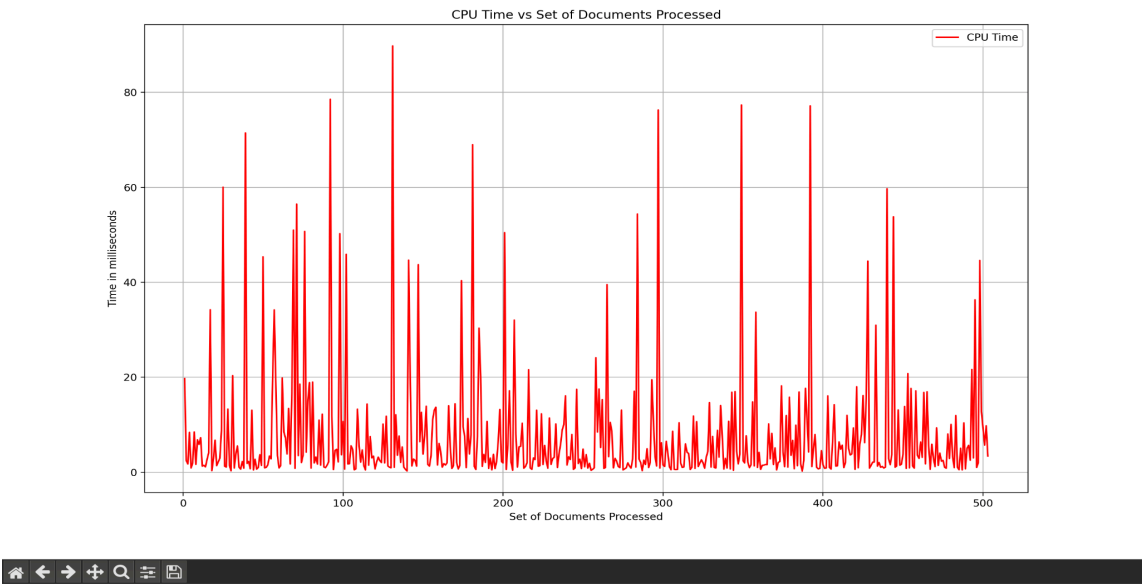


Fig-2: set of documents processed vs CPU time