Safia Shah

CMSC 476/676: Information Retrieval

C. Pearce, Spring 2022

Phase 1: Tokenization and Downcasing HTML text

# Abstract

This document will discuss two approaches to tokenizing and downcasing a collection of HTML documents. The approaches discussed include my use of beautiful soup in a custom tokenizer and Anshika Patel’s token implementation. In this report you will find a brief executive summary of each program which will include the handling of punctuation and numbers, calculation of word frequency, and any shortcomings. Along with three visual models, the efficiency of these programs will be discussed in order of magnitude and timings. Both programs will be analyzed and compared to determine which tokenizer seemingly produces the better output and why. The following command was used to run the program: **python3 tokenizer.py [input\_dir] [output\_dir]**

# Executive Summary

* 1. Program 1 - Safia

The first task of the tokenizer program is to capture and use command line inputs. For this python ‘argparse’ library was used. The command like when running the program captures the input and output directories needed for tokenization. The program has to be used in the same locations as both directories as it does not handle and has not been tested to handle changing directories or full path arguments. Once each directory name was captured and saved as a string, the input directory’s files were captured using the os.listdir() function and sorted to make sure the looping of input files to be done in order and match up with the corresponding output file when it is created.

For each file in the directory the tokenizer starts off by pulling html data from the files. There were two approaches, when creating the tokenizer, to pull and parse the information in which both used python’s Beautifulsoup library. The two parsers were the ‘html.parser’ and ‘html5lib’ parsers. Both were experimented and ‘html.parser’ was the one I eventually went with[[1]](#footnote-0). The html5lib parser was able to parse through all the documents, using the default beautifulsoup encoding of UTF-8, without any warnings. However, it was a bit lenient in what it pulled and parsed through as it included style scripts. Because of this the get.text() method that was used when parsing the returned string from beautiful soup also included the style script language as text. This was obviously not ideal. The ‘html.parser’ was better as it ignored style scripts within the html files and so when tokenizing the returned string it didn’t include tokens like ‘bold’, ‘size’, ‘format’ that were from the style scripts. However, using ‘html.parser’ gave a new problem which was the warning that read “some characters could not be decoded, and were replaced with REPLACEMENT CHARACTER” for the html files 377.html - 396.html. This problem was mitigated by specifying the encoding when using beautifulsoup to ‘iso-8859-1’ instead of using the default utf-8. The use of ‘iso-8859-1’ resulted in no warnings and, from the quick scans through the output files, the same resulting tokens. One thing this tokenizer does not handle are links embedded in the given html file. An example is the links in file 440.html in which most of the text extracted is the “textplain” from the description section. Any other html errors were not noticed by me[[2]](#footnote-1).

Once the files were read in and the beautifulsoup string was returned, the string was passed into my parse function along with the current page number and the output directory name. For each file’s resulting string, aka ‘soup’, I applied the get\_text() method on it, replaced all leading and trailing newline characters, and finally split them into tokens using .split(). The resulting array from .split was then iterated through and each index value was stripped of all non-alphanumeric characters and downcased using python regex method re.sub() and .lower() respectively. Hyphens between words were stripped and the words were joined. I chose not to strip the numeric characters from each string as for some files that would not make sense. If a username is ‘abc123’ it would not be the same as ‘abc’ which would then make the search for that username result in an incorrect answer. However, because there was no specification to keep anything other than just alphabetic words, all alphanumeric tokens were not added along with the only-number tokens[[3]](#footnote-2). Within this parse function, once the token was modified and met the alphabetic criteria it was then added to the dictionary, a built-in python hash table, used to hold the token and its corresponding frequency. Because the dictionary keeps track of all keys, the token was only added if it was not already found within the dictionary. If the token already existed, then instead the frequency/ value of that key/token was incremented by one. The token was then added to the output file along with a concatenated newline char to aid in the output file format. This was, again, done for each index in the resulting array from .split() (each token in the file). The current open file was then closed, finishing parse().

Once the input file’s tokens were written to its corresponding output file for all the files in the input directory, the final dictionary was then sorted and written to two different sort text files. Two functions, sortByKey() and sortByValue() were used to create and write the values of two temporary dictionaries. One dictionary was sorted by token in alphabetical order, and the other sorted by frequency. Both used the built in python sort function. This concludes the tokenizer.py. The time it took for each set of files (increments by 50) was rather efficient as the max number of files, 503, was processed in about 10 seconds using the html.parser method and 22 seconds using html5lib. When confirming file frequency was correct, I used command-f to get the resulting frequency of words of a chosen html output file and then only applied the parse function to that single file in order to see if the frequencies aligned and they did. This was useful to confirm that the total frequency after all files were parsed would be correct.

* 1. Program 2 - Anshika

Anshika’s tokenizer was divided into three stages. Stage one parsed the html files in the input directory to extract just the text from them and then wrote the token to text files in the output directory. She used the ‘html.parser’ parser type when using python's Beautifulsoup to get the html from the files. Because of decoding warning used with html.parser arising for some files, she used the encoding ‘iso-8859-1’ and then re-encoded the tokens when the file was parsed to ‘utf-8’ because of issues arising when saving to the text files. Then she went manually to strip html missed by the parser. The second stage included cleaning the text within the text files by conducting the following steps: downcase all tokens, join all hyphen separated words, and get rid of all non-alphabetic characters from each token. Then she included the use of python library NLTK’s tokenizer and saved it back to each respective file. In stage three Anshika sorted the dictionary used to hold all the tokens by token and then frequency using python’s sorted functions. Each sort type was written to their respective files.

# Differences Between Results

* 1. Chart 1a & 1b: Time of my tokenizer with Beautifulsoup. For consistency, encoding form is the default of UTF-8 as html5lib didn't give decoding warnings.

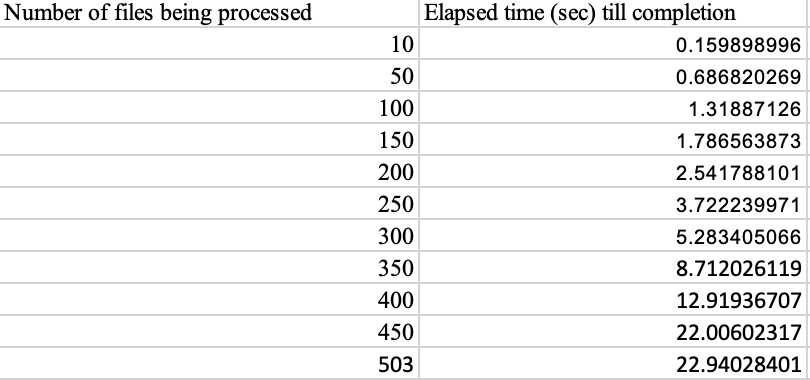


Figure 1a. Times using Html5lib type for Beautiful Soup

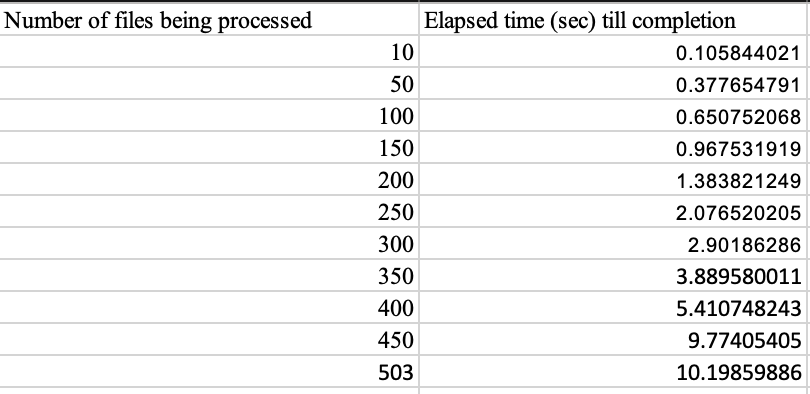
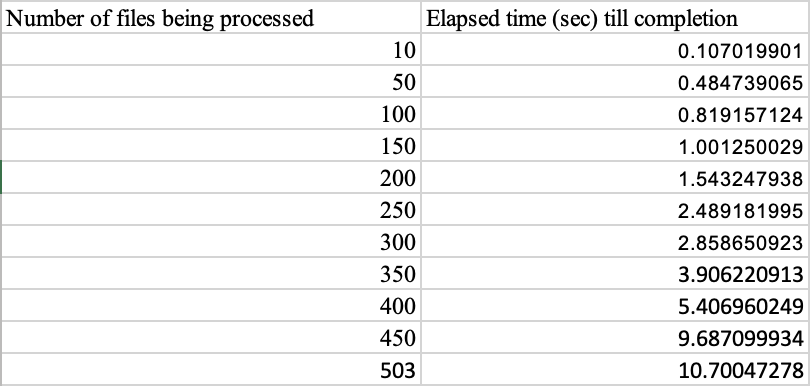
​​

Figure 1b. Times using Html.parser type for Beautiful Soup



// Bonus Chart: using html.parser with encoding = iso-8859-1

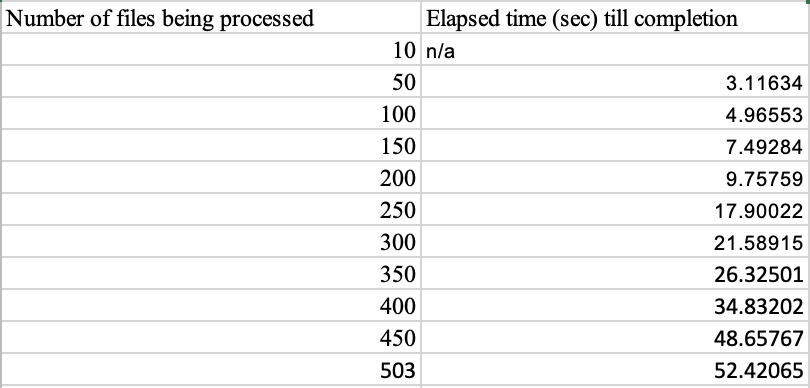


Figure 2: Anshika Patel’s implementation of Beautiful Soup and NLTK

# Determining Which Is Better

Both Anshika and I used the beautifulsoup parser for our implementation in different ways. Anshika’s addition of using NLTK to add an extra layer of tokenization is something that I did not use. From the executive summary, I definitely did not follow as many steps tokenizing the html text as anshika did. I modified my tokens, appended to the dictionary and wrote to the output file as I looped through each input file's resulting text. Anshika got the html text, wrote to the file and changed its encoding, and then for each file tokenized and used NLTK and then created her sorted files. When comparing output files for frequency, she does have more tokenized words and sometimes the same tokens have higher frequencies (e.g stops words like ‘a’, or others ‘aaa’, ‘and’ etc). I believe this is because she stripped some text file alphanumeric tokens of their numeric characters and included that while I had excluded the alphanumeric characters all together. It's also entirely possible that her program was able to detect and tokenize text better than her program was due to her use of NLTK’s tokenizer. The frequencies and tokens are not too different sans the difference in approach and tokenization.

However, when it comes to run time my program is significantly better. Given the max input of files my current run time is about 50 seconds or so faster as well as much faster in all other file increments. I would personally go for my approach as it is simpler in processing structure, time efficient, and produces similar results to Anshika’s program. Given modifications to include alphanumeric tokens, stripping numeric characters from tokens, different encodings, and different parser types are all included as comments for variation in the tokenizer there is more experimentation allowed with files that may not be included in the input file directory given. Different encoding for file types, formats differences, and languages can be catered to.

1. The html5lib parser type is commented into the code where it would have been had I used it. I think it's important to mention as it played a role in early development of this tokenizer. [↑](#footnote-ref-0)
2. I say this because I am new to parsing and using html so there is the possibility of human error when going through the files and seeing what is different or wrong. It is also very time consuming and not efficient. [↑](#footnote-ref-1)
3. Included in the tokenizer.py file is a commented out line of code that allows the inclusion of alphanumeric characters and not numbers if that is better. [↑](#footnote-ref-2)