CSCE 5533 Advanced Information Retrieval
Homework 1
Due: Thursday, September 6 at 11:59pm
By: Kyle Morley

Objectives

The objective of this homework was to build a program that could tokenize html documents. The goal was to build a program similar in functionality to an existing html parser such as Jsoup.

Approach

The language I used for the program was python. I used the python implementation of flex, Ply to build my tokenizer. The main component of the program is the class HTMLLexer which specifies the tokens that exist in the document. In the class you also define each token with a regular expression, and can write additional code to process the token when it has been matched to the regular expression.

The lexer is given as input one entire html document as a string. It tokenizes the text by the rules defined, and saves each token in a list. Once the end of the input is reached, the tokens are then output to a file. The existing list of tokens is then turned into a python dictionary with the token as the key and a value of 1. The dictionary then sums the values of all entries with matching keys to obtain the frequency of each token within the html document that was just tokenized. The HTMLlexer contains a dictionary that was used to keep track of the total frequency of each token. After the frequency dictionary for one document has been created, it is used to update the total frequency dictionary. At the end of the program when the frequency count needed to be saved to a file, I used the python “sorted” function to sort the dictionary once alphabetically by key and once by the values descending. The python sorted function uses a combination of merge and insertion sorts.

For the tokenization rules I had two rules for html tags: The first was a rule to recognize all occurences of <> with anything in between and would ignore these. The second was a special rule to recognize when an html tag is in the middle of a word for example such as: he<b>ll<\b>o.

The basic rule for capturing words and numbers was a rule that looks for all strings that contain only numbers and letters. All that needs to be done with these strings is to convert them to lowercase. The special cases I implemented were for words containing hyphens and abbreviations. Hyphenated words simply have the hyphen removed to form one token, abbreviations are any words that have periods with letters on both sides, the periods are stripped and it is treated as one token.

For numbers there are special rule for floats and numbers with commas. For floats everything to the right of the decimal place is ignored. Similarly numbers with commas just removes the commas.

Results

The results of tokenizing are

Most Frequent tokens (10)
Token, frequency

the 14281
and 9140
of 8892
to 6384
a 5584
in 4619
for 3932
is 3107
on 2570
1996 2221

Least Frequent tokens (10)
Token, frequency

whereabouts 1
illustrious 1
cs16 1
progam 1
nonthreatening 1
furmanspecific 1
furmanedu 1
irina 1
sundancecsouiucedu 1
zen 1

Alphabetically first tokens (10)
Token, frequency
0 953
00 34
0000 3
000000 29
000020 1
000033 1
000050 2
000080 1
000090 1
000099 4

Alphabetically last tokens (10)
Token, frequency

ztransforms 1
zucker 1

zuckerman 2
zurich 11
zvornik 1
zwaenepoel 1
zwiener 1
zwilling 2
zytkow 1
zyuganov 1

Discussion

One part of the program I did not like was my rule to tokenize abbreviations. The rule recognizes a period with text on both sides, which unfortunately also applies to URLs and treats strings such as “www.website.com” as an abbreviation.

Another problem with the program is that it completely ignores all content within html tags. So some tags such as meta tags which may have text that would be helpful for searching documents is simply ignored by my tokenizer.

Efficiency

Complexity Analysis

The complexity of this approach is O(some function of n) based on the number of documents? Unique tokens? Total tokens? Files? Characters?

Characters: For each additional character that is input the program must read the character and match it to some regular expression. My program has a fixed number of regular expression matching rules , in the worst case the program checks if each rule applies to each character and then runs the one matching rule. So for each character c we perform a constant number of comparisons and then run the rule which only strips out character or downcases, which only takes O(s) time where s is the length of the string matched. So parsing the documents grows linearly in time with the number of characters.

Number of files: This has minimal effect. If you have two sets of documents and both sets have a similar number of characters with differing number of files, the only additional overhead between the two will be the amount of time it takes python to open and close a file.

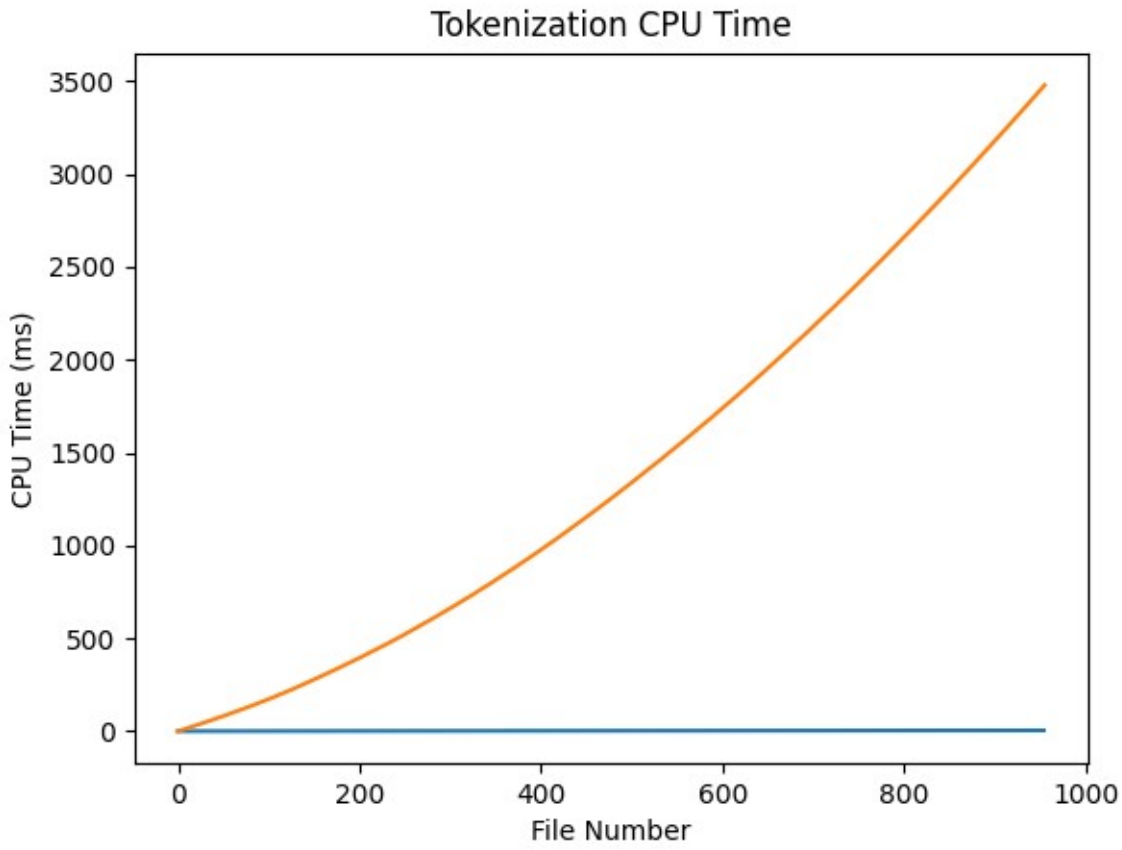
Unique tokens: I have to sort a dictionary that is the size of the number of unique tokens to output the total frequency counts. I sorted with Python’s inbuilt sorting method which is Timsort which runs in O(n log n) time.

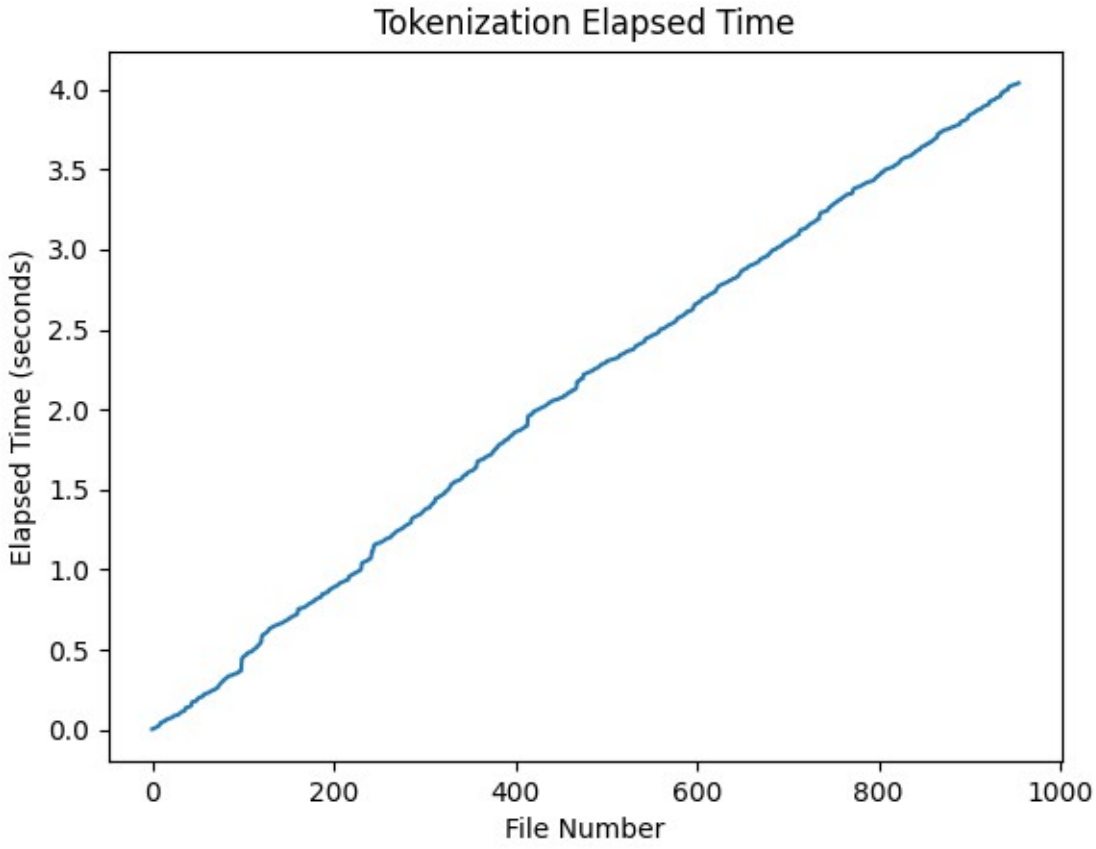
Total tokens: With each additional token, I have to perform one more addition of values in my dictionary, which requires a python dictionary search and an addition. A python dictionary is a hashmap so the search only takes O(n) time.

So the total complexity is O(c + n log n) where c is the number of characters in the documents and n is the number of unique tokens in the documents.

Timings

Graph of time taken **on turing** as a function of the number of documents. Please provide 2 charts (or one chart with two lines) showing the "real" time (elapsed time from start to end) and "user" time (CPU time used by the program).





Testing

The main method I used to test the tokenization was the test method within my HTMLLexer class. The idea to test the lexer this way and the code for the test method are from the examples in the Ply documentation. The test method takes as input a string and outputs the tokenization to the console. When adding a new rule I would add an example string as input to the test method inside of my main function to test the new rule. It would also allow me to see the output of previous rules and make sure the new rule did not interfere with the old rules. Testing in this manner allowed me to quickly add and debug new rules.

Past that I would just run the program on the files simple.html, medium.html, and hard.html and visually check the tokenized output to inspect if the tokenization was performing adequately.

Sample Run
Typescript or screenshot of program running **on turing**.

For this assignment I wasn't able to get ply to properly run on Turing. For Ply it is possible to simply copy the source code of lex.py into your project, but I made the mistake of developing the whole program on my computer with the version of Ply installed with pip, which was a slightly different version that was incompatible with the source code.

Resulting Tokens: simple.html

this is a test file this file has some text in it the contents are all simple english it has some rare content my last name gauch that is all nothing very interesting or much content

Resulting Tokens: medium.html

another test file this file has some text in it sentences designed to test for tricky punctuation sgauch uarkedu 1234567890 http wwwcsceuarkedu sgauch jgauch gmailcom elephants are big if x lt 20 then cout small number penny nickel dime quarter in 2007 i joined the csce department there were 20 grams of gold in the ring 1500472 people live in cincinnati maybe h20 is a chemical compound

Resulting Tokens: hard.html

susan gauch phd this file has some text in it the text is designed to see how you handle html tags information retrieval home page

Resulting Tokens: 119.html

server netscapecommerce 1 date tuesday 26nov96 0007 15 gmt lastmodified thursday 15jun95 0038 37 gmt contentlength 1947 contenttype text html programmming systems research group david k gifford professor electrical engineering and computer science the programming systems research group is exploring methodologies to facilitate the discovery and use of information our current projects include an intelligent query access system that implements resource discovery on the internet nii digital video storage composition and presentation mobile access to internet resources caching architectures for information and personal information discovery agents these projects are presently based on extensions to semantic file systems mosaic www servers wais servers and object repositories each of our projects is designed to test a new concept examples of concepts we are currently examining include a href http wwwpsrglcsmitedu projects crs introhtml content routing a href http wwwpsrglcsmitedu projects avs introhtml algebraic video and the creative use highly asymmetic bandwidth our approach to research is to use formal tools implementation and experimentation to help us understand the strenghts and limitations of a given system the end goals of a project are to publish new algorithms and

experimental findings to create a complete system that can be released for use by people outside of mit and to have fun of course