CMSC676: Information Retrieval
Homework 3

John Seymour[*]
Department of Computer Science and Electrical Engineering
UMBC

March 26, 2014

The aim of this project is to create an inverted index for all words within a corpus of 503 HTML documents. The first assignment, the backbone of this project, compared alternative methods for downcasing, tokenizing, and calculating frequencies for all words within a corpus of 503 HTML documents. The second assignment calculated term weights for all words within the corpus. The first and second reports are summarized below.

The original project stripped out all html tags and then threw away all non-alphabetic input, including punctuation and numbers. This had two interesting effects: first, that several contractions may have been mistaken as possessives, and second, that all hyphenated words became single tokens, instead of splitting into a token for each word in the compound. It utilized the java.util.Scanner class to tokenize the input files, which has a method for creating custom delimiters, useDelimiter(). A custom regular expression was used to separate the words into tokens and to parse out html tags. However, the Scanner class first delimits based on whitespace, which had the effect of including a few multi-word html tags into the final tokenized output. HTML entities, such as &acute for the e in "Felix", were also not decoded, leading to interesting phenomena. Finally, the program sometimes had empty tokens, due to all characters in the token being removed because of the above rules. Such tokens were deleted.

The original, optimized approach utilized the hashmap data structure, with the key being the token and the value being the frequency. The program linearly scanned each input file for tokens based on the above rules

[*]`seymour1@umbc.edu`

and incremented counters corresponding to tokens in the file to calculate frequency. Empty tokens, stopwords and all words of length 1 (i.e. a, b, c, etc.) were removed after tokenizing the file. However, words that only occur once in the entire corpus could not be removed during the original tokenization of the file: whether a word only occurs in that particular document cannot be known until scanning all documents. It is for that reason that the removal of words that only appear once in the entire corpus happened at the calculation of the tf * idf stage.

I also needed to store the token/frequency pairs, from documents in the corpus, in some data structure in order to calculate the tf * idf after scanning all documents (whereas in assignment 1 it was sufficient to print the frequencies after tokenizing the file- only local operations were required). I used an ArrayList to be able to easily iterate over the documents for calculating tf * idf. Without such a data structure, it can be easily shown that two passes through the data are required, further increasing the I/O time required to weight the terms in the document. The equation used for weighting values is still the simple variant, given by the following equation:

$$weight = tf * idf / normalizationconstant \tag{1}$$

Where TF is the term frequency in the document, IDF is the log of the total number of documents divided by the number of documents containing the term, and the normalization constant is the sum of all frequencies in the document post-removal of terms.

I made several modifications to this project as compared to the last. The first, obvious modification was to write the dictionary and postings files. After the term weights are recorded, we can simply iterate over each term and find which documents contain the term along with how many, and their associated tf*idf value. This is all the information we need to create the dictionary and postings file. The final required field, the location of the first record for the term in the postings file, can easily be found by adding the block size of the previous term times the number of occurrences of the previous term, added to the location of the previous term in the postings file. This results in the dictionary records being in hash_order. I did not need to make any assumption based on the maximum length of a word for this corpus, as all words are shorter than the block size.

Instead of writing the term weights to a file, I used a memory-based algorithm and wrote them to a HashMap. Without storing the term weights in memory, they'll have to be retrieved from scanning the files, which results in an extremely large overhead. As a concrete timing example, the final project took around 25 seconds to index all files using the memory-based algorithm, but required just over 12 minutes to finish when scanning the files. What's more, the extra overhead of calculating the information for the postings and dictionary files is completely absorbed into the time saved from not having to write out the term weights to a file. Because they are not a required output this iteration, a lot of file I/O time was removed.

I used the linux time command to calculate the runtime for indexing various numbers of documents. Results can be found in Figure 1. The runtime seems to be on the order of n squared, where n is the number of documents. The reason for this can be seen in the program and from phase two: the runtime is still dominated by the nested for loop- the outer for loop looking over all of the tokens in all of the documents, and the inner for loop comparing searching for these tokens to all other documents. This could be made more efficient by storing which documents contained the search terms in the map of all tokens, using an array instead of an Integer for keeping track of frequencies in each document. However, for small numbers of documents, such an approach would not affect the runtime drastically.

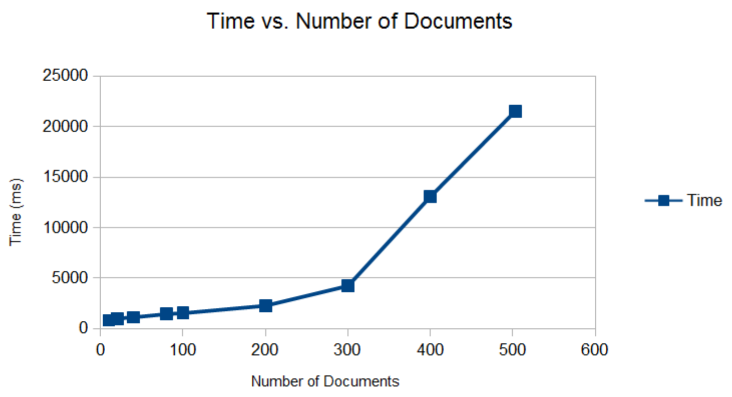

Figure 1: The amount of time required to tokenize and weight documents for various sample sizes. All times are averaged over 10 trials on the corpus.

I also determined the size of the output files for various numbers of documents. Results can be found in Figures 2 and 3. The total size of the files seem to be on the order of n squared, but it's hard to determine because the number of terms and their sizes can influence the sizes of the dictionary and postings file.

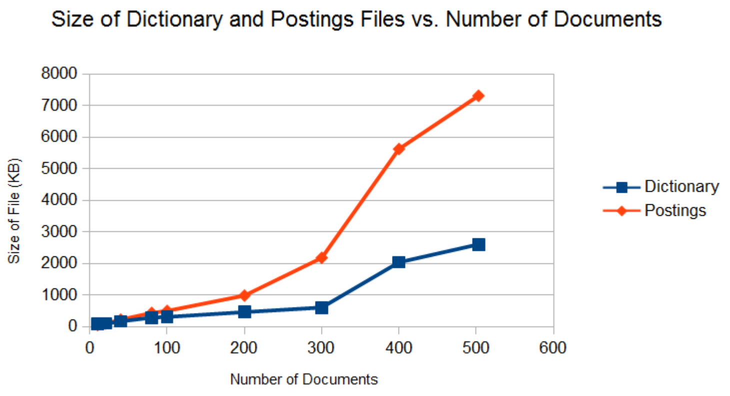

Figure 2: The size for the dictionary and postings files for various sample sizes. All sizes are averaged over 10 trials on the corpus.

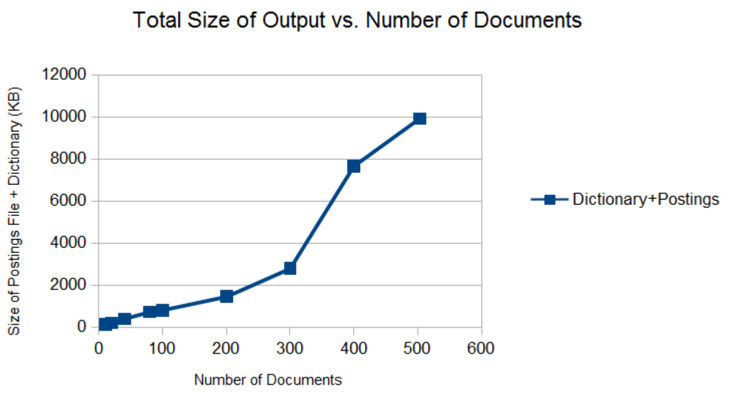

Figure 3: The total size of the output files for various sample sizes. All sizes are averaged over 10 trials on the corpus.