RISHABH SACHDEVA (rishabs1@umbc.edu, UI73138)– INFORMATION RETRIEVAL – HOMEWORK 3- INDEX CONSTRUCTION: DICTIONARY AND POSTINGS

INSTALLATION AND PROGRAM EXECUTION:

Programming Language: Java 1.7

Prerequisite Installations:

Java 1.7, Apache Maven

Installation:

I used **Apache Maven** as a building tool, as a few external jars (like apache-commons, jsoup) were utilized for the execution. To install and create jar, following command can be executed from the location where pom.xml is located:

mvn clean install

assignment-rishabh-0.0.1-SNAPSHOT jar will be created in the target folder on successful building process.

Execution:

Program can be executed either via executable jar or manually creating class file and run.

Commands:

To compile: *javac informationRetrieval.assignment1.IndexConstruction.java*

To run: *java informationRetrieval.assignment1.IndexConstruction <input_folder_arg> <output_folder_arg>*

Output : dictionary.txt and postings.txt (format as mentioned in assignment)

APPROACH

Term Weights Used: I used term and term weights generated in Homework 2. These weights were calculated in my previous assignment using BM25 approach.

The calculated scores were normalized for each term in 0 to 1 range using following formula:

normalizedScore = (score_bm25 - min_wt)/(max_wt - min_wt);

where score_bm25 = actual bm25 score calculated for the term.

Min_wt = minimum bm25 score in whole corpus.

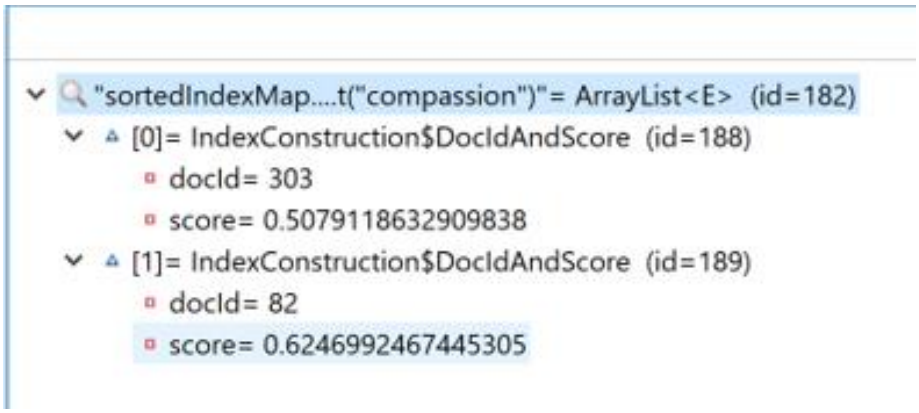
Max_wt = maximum bm25 score in whole corpus

RISHABH SACHDEVA (rishabs1@umbc.edu, UI73138)– INFORMATION RETRIEVAL – HOMEWORK 3- INDEX CONSTRUCTION: DICTIONARY AND POSTINGS

Logic: Program-wise, I created the output of the actual bm25 scores in the temporary folder, which were further used to fill a global hashmap created containing term VS posting list.

HashMap<String, List<DocIdAndScore>> invertedIndexMap= new HashMap<String, List<DocIdAndScore>>()

DocIdAndScore is an inner static class which is utilized to maintain <docid, normalized_score> tuple corresponding to each term. This map is then sorted based on key (alphabetically) so that resulting output files are readable and easier to understand. It contains each token as key and has corresponding list with doc_id and its bm25 score as value.



In above debug screenshot, the value of the map corresponding to key (token = “compassion”) is shown. So, this term is present in two documents with doc Ids 303 and 82 with normalized scores ~0.508 and ~0.624 respectively. Finally, this map is used to create posting and dictionary file because it contains everything we need.

The iteration over key set of the map is established. Further, second for loop is designed such that whole posting list is iterated to create entries in posting file. DocId, Term, normalized weight and occurrence in number of documents was directly fetched from the map. Location in posting file was maintained while writing the output files. Location was initialized by 1, and further incremented each time by size of list(posting) corresponding to the term.

location += posting.size();

Certain Issues:

Output files show a few special characters (Unicode entities), which got carried forward from initial submissions. I used self-created regex to fetch tokens, not the external libraries. I have fixed most of

them over the time, but there are still a few which appears. Other than that, the results are as I expected.

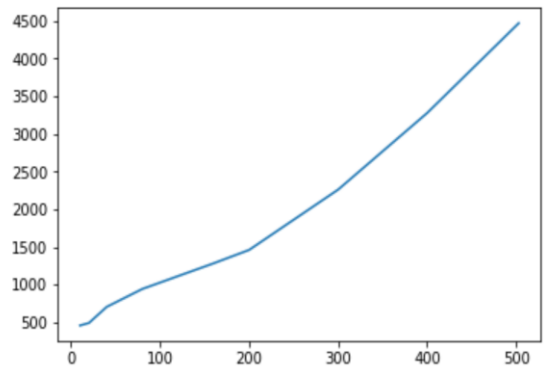
MAXIMUM LENGTH CONSTRAINT:

The maximum length of the term is kept as 45. Any term with more that 45 terms is ignored. The longest word in English language is Pneumonoultramicroscopicsilicovolcanoconiosis with length 45, and its rare that any authentic word will be encountered in the corpus above this length. I made this constraint because a couple of words with huge length (like from Content-Transfer-Encoding body) were encountered which doesn't make much sense to be included.

RUNTIME AND MEMORY ANALYSIS:

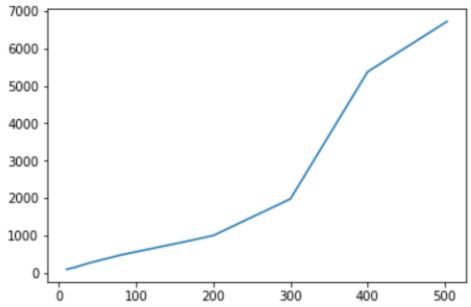
Number_files	Execution_Time(ms)	Postings_size(kB)	Dictionary_size(kB)	Dictionary +Posting Size
10	456	98	65	163
20	489	145	85	230
40	703	271	130	401
80	941	480	194	674
160	1281	820	260	1080
200	1459	1000	297	1297
300	2260	1979	411	2384
400	3277	5375	1526	6901
503	4471	6720	1794	8514

Num_Files VS Execution Time:

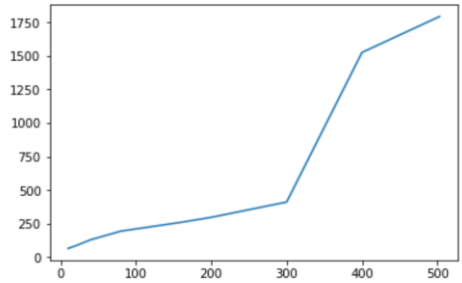


RISHABH SACHDEVA (rishabs1@umbc.edu, UI73138)– INFORMATION RETRIEVAL – HOMEWORK 3- INDEX CONSTRUCTION: DICTIONARY AND POSTINGS

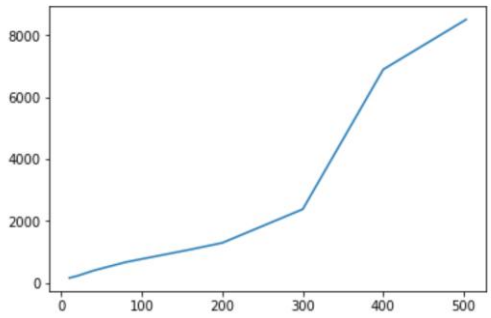
Num_Files VS Postings_Size:



Num_Files VS Dictionary_Size:



Num_Files VS Total_Output_Size (Posting and Dictionary):



Hence, the running time and output size increases input corpus size increase. Also, a sharp increment in output size is observed between corpus size from 300 and 400. It appears that these documents have more terms (stronger memory wise) compared to others.

And, it turns out, their average per file size (33.6kB) is far above the average per file size of corpus (23.85 kB). It is approximately 1.4 times the avg per file size of corpus, therefore such an increment is observed.