CS6200 Project 3: INDEXING

Name: Ajay Pandit /MS in Computer Science Fall '12 / CCIS Username: ajay / Email: pandit.aj@husky.neu.edu
Northeastern University
AJAY PANDIT

Inside the Report:

CS6200 Project 3: INDEXING

Name: Ajay Pandit /MS in Computer Science Fall '12 / CCIS Username: ajay / Email: pandit.aj@husky.neu.edu

----

[**Problem Statement**]: In this project, you will replicate the functionality of the Lemur index

Things to do:

- Download the CACM collection from the Search Engines: Information Retrieval in Practice test collection site.
- Create an index of the CACM collection, together with code replicating the functionality of the Lemur index used in Project 2
- Now, using the index you just built and your code from Project 2, perform retrieval experiments on all CACM queries using all five retrieval algorithms from Project 2. Record and report mean average precision and mean precision at cutoff 10 and 30 results, as you did for Project 2. You should, of course, use the queries and qrel file that come with the CACM collection. Note that you should use the "raw" queries, not the "processed" ones. This will allow you to stopword and stem your queries in exactly the same way you did the documents when indexing.

Files to produce:

- a file that maps term names to term IDs and associated term information, such as inverted index offset and length values (see below) and corpus frequency statistics
- the inverted index file that maps term IDs to document IDs and associated term frequencies, and
- a file that maps document IDs to document names and associated document information, such as document lengths.

Points to note when creating index:

- When creating your index, you should first apply stop-wording using the stop-word list
- When creating your index, you should then apply stemming

Hints Provided:

- As you process documents, maintain a separate file per unique term, adding document information to these files as you go
- Concatenate these files into one inverted index file and add the appropriate inverted index offset and length values to the term information file

Information Given:

- CACM collection
- Porter stemmer and KStem stemmer programs to help us with stemming functionality
- List of queries to be run on the corpus
- Qrel file

CS6200 Project 3: INDEXING

Name: Ajay Pandit /MS in Computer Science Fall '12 / CCIS Username: ajay / Email: pandit.aj@husky.neu.edu

1. Indexing Process:

We follow the below steps for creating an index:

1. **Convert all HTML files into a single TEXT file:** Read all html files and create a single text file which will only have all the text as it is displayed on the browser. We add a counter to separate each of the contents of the html files. By placing this counter before writing each of the html file into text file helps us in further steps while reading the text file
2. **Eliminate Column of Numbers from the TEXT file:** We provide the output obtained from step-1 as input to this step. Here we focus on removing the numbers that are present at the end of each of the files as it has been mentioned in the problem statement that we can ignore the column of numbers that are present at the end of each file
3. **Stemming and Stopping of text:** The output file obtained after step-2 is not given as input for this step where we focus on the process of stemming and stopping. We use Porter Stemmer technique which helps us in stemming the text file and we have been provided with a set of stop words which we have to use to run over our own set of text we obtained from the above two steps. So now the final output will contain the text which have been stopped and stemmed
4. **Create file for each word encountered in the TEXT file:** We pass the output file obtained from the step-3 as input at this step. Here we focus on to convert each word encountered in the TEX file into an additional file which will be stored separately under a folder named "files" and each of these files contains the document ID (docid), term frequency (tf), the word itself, a termID, and the document length (doclen). We keep on adding data to the word files till we have parsed all the words in the TEXT file. The count of the files will give us the number of unique terms.
5. **Generate required files to help Indexing**: Now that we have created a file for each word we then use these files to create 3 different helper files which will store different information about each of those words and this will actually help us in the process of indexing: The 3 files are as follows:
   a. File1 contains: [*Term - TermID - StartOffset - FinishOffset - CTF - DF]*
   b. File2 contains: [*TermID - Word - DocID - TermFreq]*
   c. File3 contains: [*DocID - DocName - Doclen]*

Description of File1: This file stores the term, term-id, starting-offset, finish-offset, the number of times the term occurs in the collection (ctf), the total number of documents that contain it (df) information for the entire corpus. Purpose of creating this helper file in the process of indexing is given below:

- This files stores the start-offset which is the value indicating the line-number from which we need to start reading the File2 and the finish-offset which stores the value indicating the line-number till which we need to continue reading the File2
- This file stores the information that a particular Term occurs in how many documents and also it provides other relevant information about the document

Description of File2: This file stores the term-id, term, document-id and the term-frequency (tf) information for the entire corpus. Purpose of creating this helper file in the process of indexing is given below:

- It stores all the terms and the information related with respect to those terms
- It also provides information about in which all documents a term occurs and we can obtain the same using this file

Description of File3: This file stores the document-id, document-name and document-length information for the entire corpus. Purpose of creating this helper file in the process of indexing is given below:

- Creating Internal and External mapping between documents
- Obtaining document-length information for different models
- Obtaining document-name using the document-id which is shared with other files

Description of Main-Index: This file stores the document-id, term-frequency, term, term-id and document length. This file actually contains the actual inverted index for the words in the corpus.

Process of querying using our Index:

File1 and Main index are the 2 main files which will helps us in replicating the functionality of LEMUR index. Once we have a query we perform stemming and stopping and then divide each query into a sequence of words and provide each word to this Index.

For each word the File1 will help us with the offset information from which part of Index file we need to start reading and how many lines we need to seek. The main index file will then provide us with the list of all the documents that contain the term.

2. Building an IR System:

We have been provided with a set of 64 queries which are enclosed inside HTML tags. We need to parse all these queries and convert them into normal text files. We run these queries through a process where all the HTML tags are separated and we get queries in normal format which we can then provide input to out earlier built Information Retrieval System.

Processing of queries:

- Removing all the HTML tags
- Stemming and Stopping

Once we have the queries ready we then use the index files that we created in above procedure for getting the top ranked documents for a particular query or a term.
A sample function for calculating the scores for the document using various retrieval models are given below:

```
Function Score(query, num_docs, num_unique_words, avg_doclen)
    accumulator = 0
    for each word in query we do
        ctf = number of times the term occurs in the collection
        df = document frequency
        doclen = length of the document
        docid = internal document id
        tf = number of times the term occurs in the document

        calculate the score for each term for each document by using a formula depending upon the type of retrieval model that is being used

        now update each of the document score in the dictionary. If the document score already exists as it may have been due to earlier terms we add/multiply the new score to the existing score else we add a new entry for that document in dictionary

        accumulator = accumulator + new query score
    Endfor
    we repeat this process for each of the term in the query
    return accumulator
End function
```

Please note: In addition to the score that we calculate in the above process we even have to calculate the leftover probability that is added to the language models. Once all the scores are calculated we obtain an output file in a specific format that is then run through the REL files and this has been discussed in detail in the next section.

3. Analysis of Output file produced after running each of these models:

Once we have completed with the coding exercise for the models we are required generate a result text file in the following format:

query-number Q0 document-id rank score Exp

where:

- *query-number is the number of the query*
- *document-id is the external ID for the retrieved document*
- *rank is the position in the ranked list of this document*
- *score is the score that you system creates for that document against that query.*
- *"Q0" (Q zero) and "Exp" are constants that are used by some evaluation software and must be present, but won't make a difference for your score (Q0 takes an 0 and Exp takes an 1)*

Once we obtain 5 text files one for each retrieval model in the above mentioned format, we then have to use the TREC_EVAL file which has been provided and run these output files on REL file which has been provided

After running our 5 output files against the REL file we get 5 more files which provide us with the following information:

- Interpolated Recall Precision Averages
- Non-interpolated Precision Averages at count interval of documents
- Other information like the
    - query-number,
    - number of documents retrieved,
    - the number of relevant documents,
    - number of non-relevant documents

Using these files obtained from above process to calculate the following:

- Mean Average Precision
- Recall Precision
- Precision at 10 documents
- Precision at 30 documents
- Count of relevant documents retrieved

4. Result Analysis:

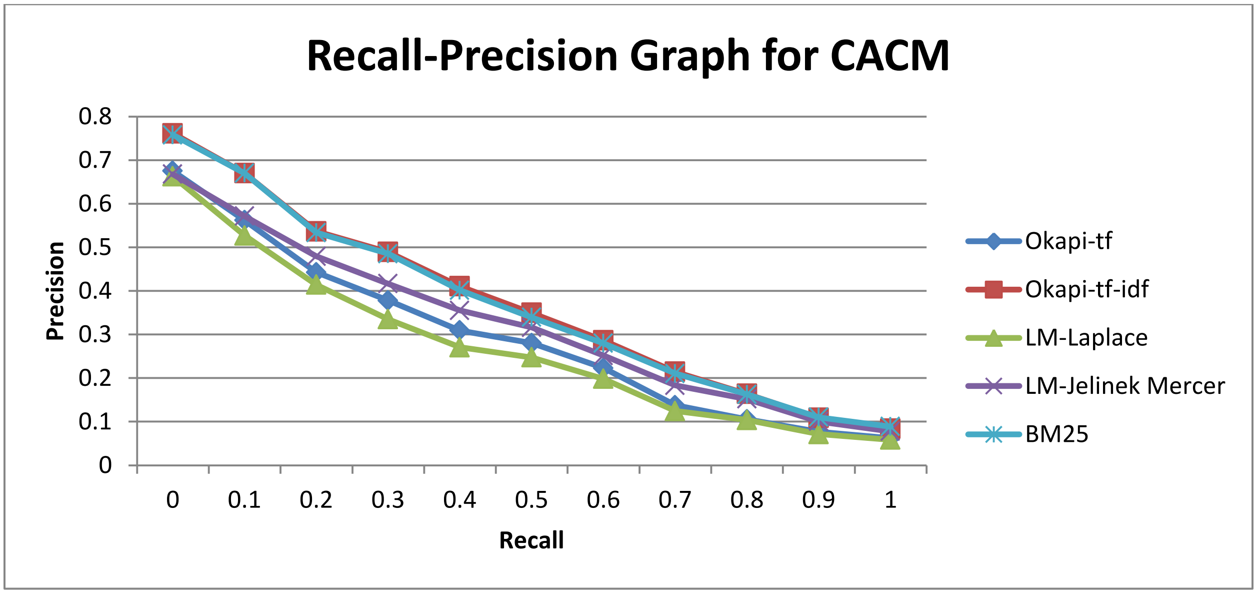
The database and the queries that were used were both stemmed and stopped.

Using CACM Database						
Calculating Values	Qrel File	Retrieval Models				
		Okapi-tf	Okapi-tf-idf	LM-Laplace	LM-Jelinek Mercer	BM25
Mean Average Precision	CACM.REL	0.278	0.3504	0.2565	0.3064	0.3465
Recall Precision	CACM.REL	0.2954	0.3496	0.278	0.3148	0.3481
Precision at 10 Documents	CACM.REL	0.2981	0.3635	0.2731	0.3077	0.3538
Precision at 30 Documents	CACM.REL	0.1833	0.216	0.1577	0.1955	0.2135

Analysis of the Table result:

- From the above table it can be seen that there are 2 models Vector Space Model - Okapi-tf-idf and the BM25 which are having the highest precision values as compared to the other 3 models. The order of the performance of all the models based on the precision values would be Okapi-tf-idf > BM25 > *Language Model-Jelinek-Mercer* > *Okapi-tf* > *Language Model-Laplace*
- Compared to the number of relevant documents retrieved we see that the Okapi-tf-idf model performs the best.

We have plotted Recall on the X-axis (range 0-1) and Precision on Y-axis (range 0-1)



Analysis for Graph Result:

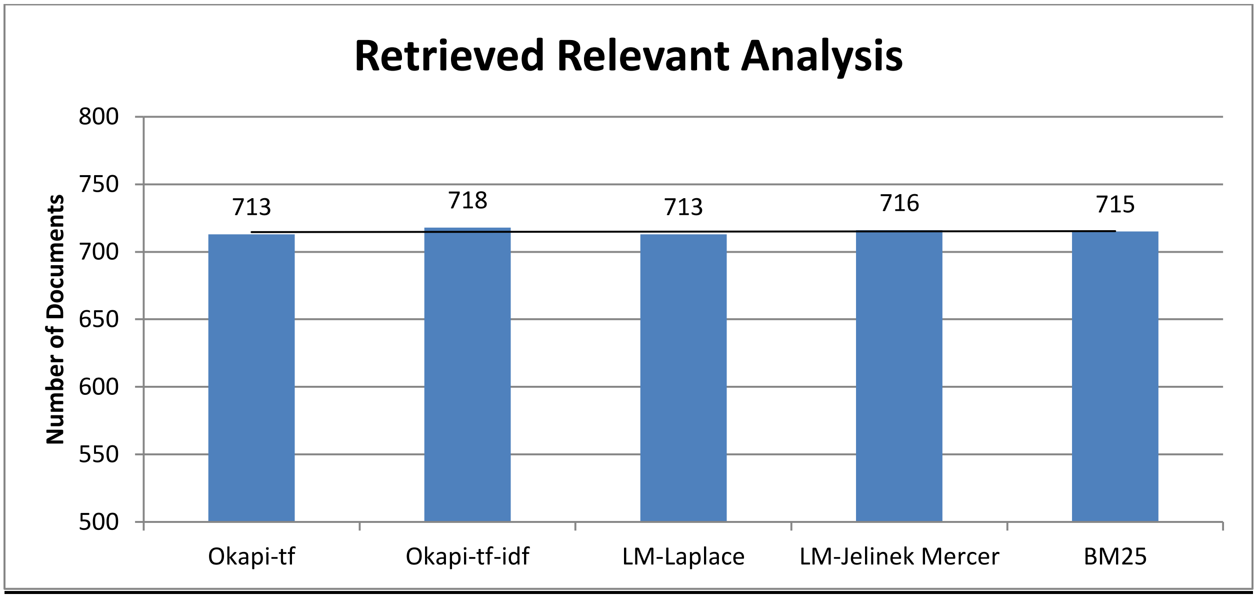
- It can be seen that BM25 model has a better precision value at the beginning but as we retrieve more and more documents then the Okapi-tf-idf model takes over from the BM25 and shows much better precision. The precision value of BM25 drops considerably once the recall reaches the value of 1

- It can be seen that LM-Laplace has the lowest precision values when compared to all other models

Based on the below graph we conclude that the number of relevant documents that are retrieved is the highest for Okapi-tf-idf model which retrieves around 718 documents. The ordering of the models based on the retrieval of relevant documents is Okapi-tf-idf > LM Jelinek Mercer > BM25 > Okapi-tf > LM-Laplace

From the above graph it can be concluded that although LM-Laplace and LM-Jelinek Mercer models are probabilistic models but we can see variation in performance of both the models. Clearly LM-Jelinek Mercer model performs better than the LM-Laplace model.

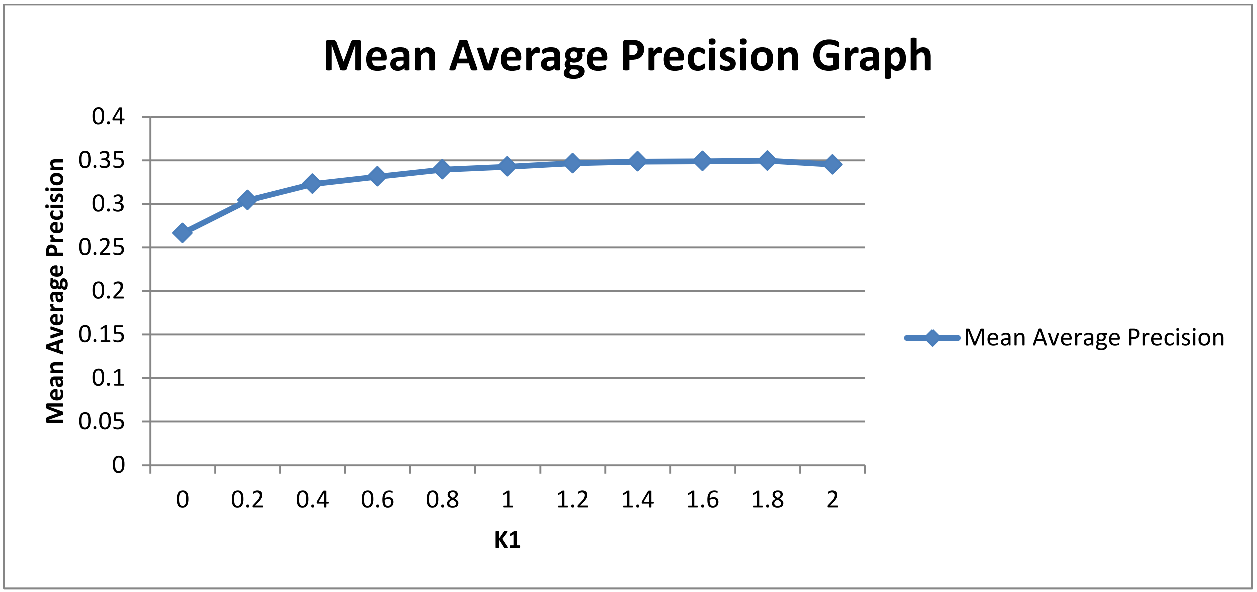
From the graph it can also be concluded that BM25 performs when the initial documents are received and is replaced by Okapi-tf-idf as the number of documents retrieved increases



5. Experiment with various retrieval formula parameters:

Analysis for BM25 using different values of k1 (range: 0.0 - 2.0):

We are varying the value of k1 which is the smoothing factor for the document term frequency from 0 to 2 and observing the effects it has on the mean average precision. We are using all 4 databases which have been provided to check the effects on different types of databases:



- From the above chart it can be seen that the average precision values of the database d3 and d1 are much higher than the average precision values of d2 and d0
- We can say that since both d3 and d1 database have been stemmed we have resulted in much higher precision value when compared to the d2 and d0 database which have not been stemmed
- After doing the analysis for different values of k1 we conclude the following:
    - Mean average precision increases with the increase in value of k1
    - Stemming affects the mean average precision values

Analysis for BM25 using different values of k2 (0-1000):

- We changed the value of k2 which is the query term smoothing factor from 100 - 1000 and found that there were no significant changes to the values of the mean average precision
- We tried to plot the mean average precision values against k2 graphically and found that the mean average values were constant at different values of k2. This is because query term frequencies are much lower and less variable than the document term frequencies

Comparison with retrieval experiments done using LEMUR interface:

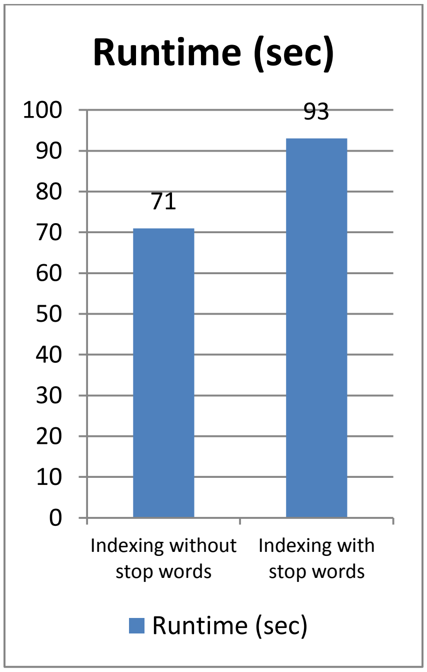
- Precision & Recall Values: The precision values that we obtained using our index are slightly higher that the values we obtained for the earlier project
- Formulas and Constants: The formula and the constants that we have used of the earlier project we have used the same for the current project and it has given us accurate results
- The order in which the relevant documents were retrieved have changed and it retrieved far more relevant documents as compared to earlier project

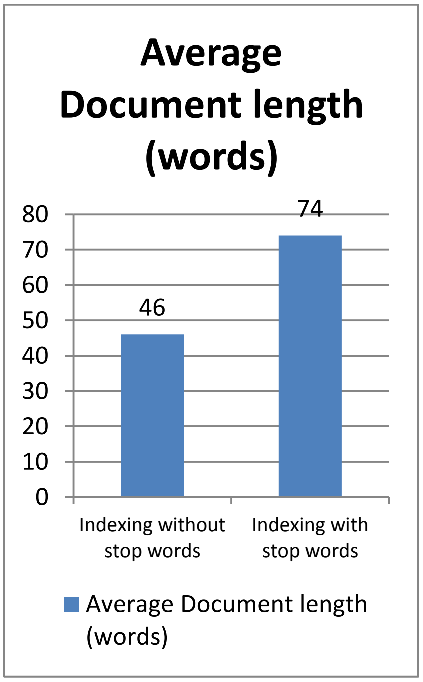
6. Extra Credit:

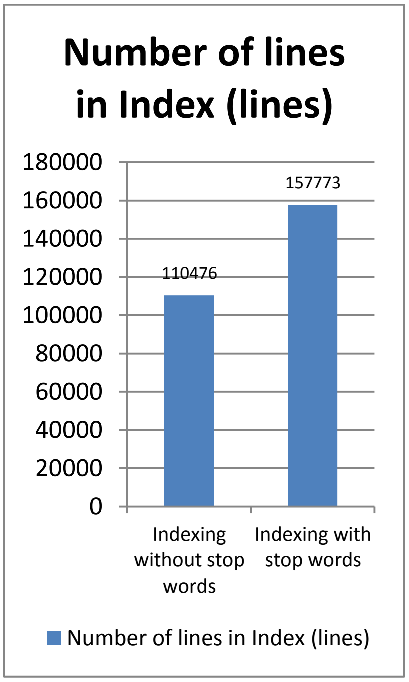
Part 1: Indexing process runtime comparison:

Description	Indexing without stop words	Indexing with stop words
Runtime (sec)	71 sec	93 sec
Average Document length (words)	46	74
Number of lines in Index (lines)	110476	157773

From the above table it can be seen that when we skip the stop words while indexing we have a considerably lower runtime and the number of lines that we need to write in the Index file is also considerably reduced which increases the efficiency and speed of the whole process. Whereas when stop words are considered while indexing then values of all the parameters increase considerably and which directly links to the stop words hence we conclude that skipping stop words increase the efficiency of the system and make them much faster.







Compression Used: Delta encoding technique on Inverted Index:

I had stored the document ID and the finish offset information in one of the helper index files since the amount of information was very large we had to use large numbers to represent that information. Hence we tried to apply Delta encoding technique where the document id and the offset information will then contain the difference information with respect to earlier document. Hence if we have a term which is present at 50 positions and at 55 positions then instead of storing 50 and 55 and we actually store 50 and 5 which will reduce the size of the information.

Example: Consider inverted list for words “who” and “entropy”. The word “who” is very common so we expect that most documents will contain it. When we use delta encoding on inverted list for “who” we see small gaps such as
1,3,4,5,4,5,2,3…….

By contrast word “hull” rarely appears in text so only a few documents will contain it. Therefore we expect to see larger gaps such as:
109, 3567, 654, 1867…….

However since “hull” is a rare word this list will not be very long, we will find that inverted lists for frequent terms compress very well whereas infrequent terms compress less well.

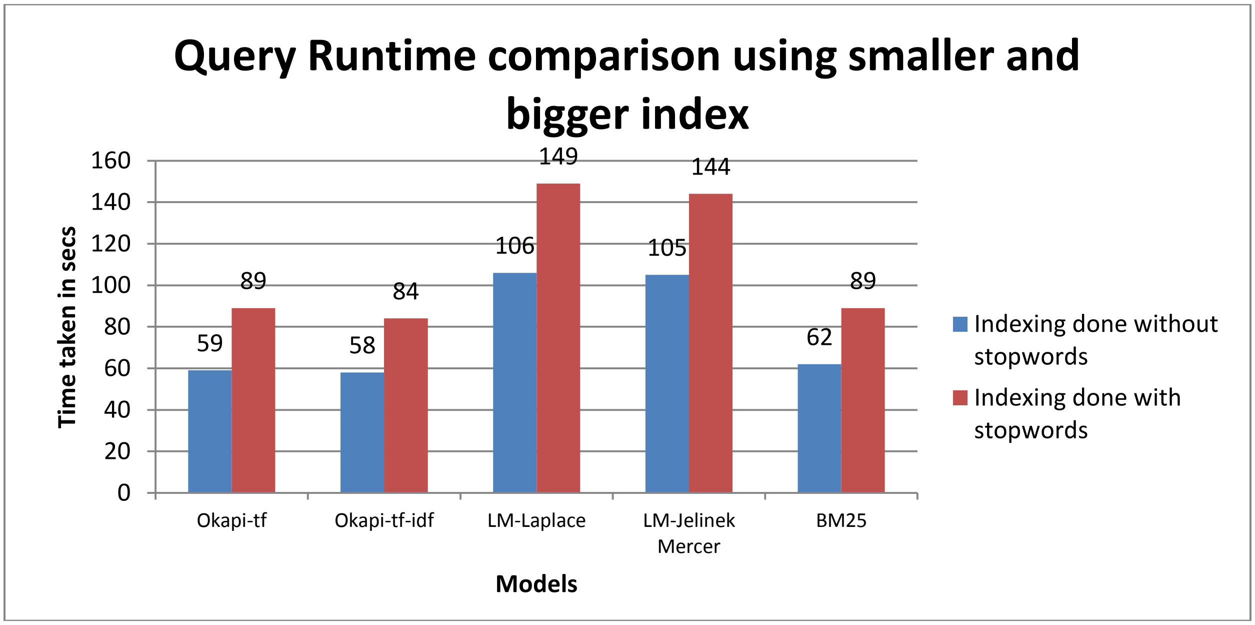
Hence we achieve good amount of compression by storing information about indexing in this format and reduce the size that is required on the disk plus achieve a better seek time on the data for individual terms.

Part 2: Query Runtime Comparison between smaller index and bigger index:

Retrieval Models	Processing time for queries without stop words when	
	Indexing done without stopwords	Indexing done with stopwords
Okapi-tf	59 sec	89 sec
Okapi-tf-idf	58 sec	84 sec
LM-Laplace	106 sec	149 sec
LM-Jelinek Mercer	105 sec	144 sec
BM25	62 sec	89 sec

We have run all the queries that were given to us from CACM on all the models 100 times and then we took the average of all the values we obtained prepared the table above and the graph below for all our findings:

- From the above table and below graph it can be seen that the runtime required for queries when index does not contains stop words is way lower when index contains stop words
- We can say approximately the runtime is 1.5 times more with stop words
- There were changes to the value of the precision due to the increase in the number of lines required to read and obtain the result increased considerably and hence this results in the runtime for all the models increasing



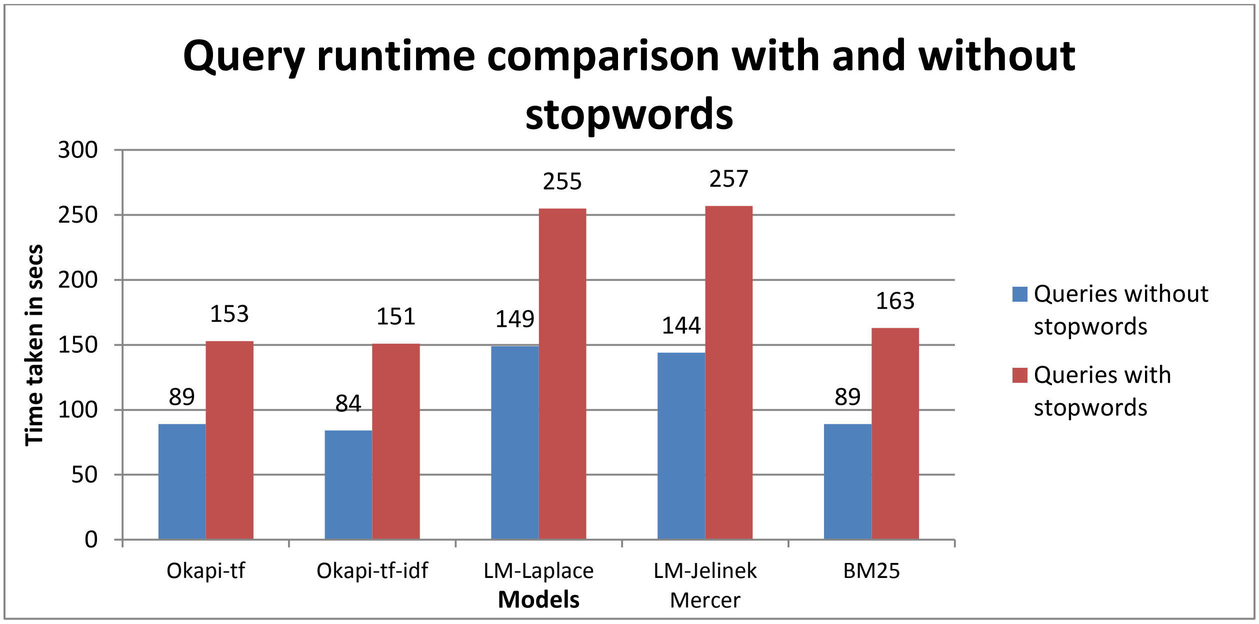
Hence it can be concluded that in order to obtain efficient and fast runtime for the queries we should exclude the stop words from Indexing.

Part 3: Query runtime comparison with and without stopwords:

Retrieval Models	Processing time when indexing was done with stopwords	
	Queries without stopwords	**Queries with stopwords**
Okapi-tf	89 sec	153 sec
Okapi-tf-idf	84 sec	151 sec
LM-Laplace	149 sec	255 sec
LM-Jelinek Mercer	144 sec	257 sec
BM25	89 sec	163 sec

We divided the queries into 2 parts, once containing the stop words and second not containing stop words and using the index that was created using the stop words we run these query files for each of these models and obtain the following average results which has been shown in the table and graph:

- From the above table and below graph it can be seen that the runtime required for queries with stop words was considerably higher when compared to queries without stop words
- We can say approximately the runtime was 2 times for the queries with stop words
- The precision value decreased for all the models when we included the stop words in the queries and hence the quality of the result would have reduced and documents with more of these stop words would have got higher rank as compared to the documents with relevance
- The number of relevant retrieved documents did not change for all the models
- The sudden increase in the runtime can be directly related to the increase in the length of each query because of inclusion of stop words. When stop words were skipped the average was 14 and when we added the stop words the average query length increased to 22 and hence there were more number of words that required to be processed



Hence it can be concluded that in order to obtain efficient and fast runtime for the queries we should exclude the stop words from Indexing and also from the queries.

Tools/Technologies used for implementing the project:

System Configuration:

- Operating System used: Windows 7/Linux
- Processor: Intel i5 processor
- RAM: 6 GB
- System type: 64-bit operating system

Programming Language:

- Language used: Python (v2.7.3), Perl
- IDE: Eclipse IDE, Python's Integrated Development Environment, Active Perl

Other Software's used:

- Notepad++
- Microsoft Office 2007
- Command Prompt

References:

- http://www.ccs.neu.edu/course/cs6200f12/proj3.html
- http://www.search-engines-book.com/collections/
- http://stackoverflow.com/questions/8369219/how-do-i-read-a-text-file-into-a-string-variable-in-python
- http://www.niso.org/publications/tr/tr02.pdf
- http://www-connex.lip6.fr/~amini/RelatedWorks/Sak01.pdf
- http://www.l3s.de/~papapetrou/publications/edbt08-fulltextindexingandir.pdf