Project Report III

Information Retrieval

Anuj Kulkarni (anuj@ccs.neu.edu)

12/10/2013

Contents

Problem

Indexing

In this project, you will replicate the functionality of the Lemur index used in Project 2, and in conjuction with the code you created implementing various retrieval functions for Project 2, you will have created a fully functioning search engine.
The Project 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.
Notes:
From the "Search Engines" book site, you should use the .tar.gz version. The .corpus version is in a special format meant for use by the (book's) Galago search engine.
You will note that the CACM "documents" are actually just abstracts of full articles, and, in many cases, not much more than titles. Before disk was cheap, many retrieval systems used no more than this.
Finally, you can ignore the columns of numbers. This is an encoding of bibliographic references. Just index the text.
When creating your index, you should first apply stop-wording using the stop-word list from Project 2.
When creating your index, you should then apply stemming. You may do so using any reasonable stemmer, such as the Porter stemmer or the KStem stemmer, each of which are freely available on the web: Porter stemmer and KStem stemmer.
Next, you should create an inverted index of the CACM collection documents, as described in class.
The index will typically consist of multiple files: (1) 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 (2) the inverted index file that maps term IDs to document IDs and associated term frequencies, and (3) a file that maps document IDs to document names and associated document information, such as document lengths.
The inverted index file constitutes the bulk of the index. For simplicity, you can build up to this file in stages:
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.
Finally, create code that, given a specified term and the index files above, replicates 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 fromProject 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 thequeries 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.
As with Project 2, experiment with various retrieval model parameters, such as the smoothing parameter, and compare and contrast your results here with those from Project 2: Do the same retrieval models work best? Do the optimal parameters change? And so on.

New: Extra Credit
For a maximum of 50 extra points, consider the following:

Many modern search engines end up indexing even stop words. Disk is cheap! But what are the tradeoffs? For extra credit, analyze the empirical time and space complexity of including stopword information in the index:
First, build an index without removing stopwords. How much larger is the index? How long did it take to create, compared to the smaller stopword-free index? If you chose to use compression in the index, what did you do?
Second, run the same queries against the same models as you did above *but still remove stopwords*. In other words, what effect does the bigger index have on runtime for the same set of queries? Since the collection is relatively small, I suggest running the queries 100 times to get accurate measurements.
Third, run the same queries *but without removing stopwords*. What happens to the query processing time and effectiveness?
Note that you should still stem the document and query terms. Points will be assigned for a clear description of the approach and presentation of the results.

Summary

In this project, the functionality of the Lemur index is replicated and used in conjunction with code created during Project 2 which implemented various retrieval models, thus creating a full functional search engine. I implemented the project in Python 3.3 using Liclipse as an IDE with the PyDev perspective plugin with test data as CACM documents.

Results obtained in Project 2:

No.	Retrieval Models	Mean Average Precision		Precision at 10 documents		Precision at 30 documents	
		NIST	IRClass	NIST	IRClass	NIST	IRClass
1.	Okapi-tf	0.2018	0.1471	0.2240	0.2800	0.1640	0.2213
2.	Okapi-tf-idf	0.2851	0.2416	0.2920	0.3720	0.2253	0.3160
3.	Laplace	0.2071	0.1405	0.2160	0.2720	0.1773	0.2507
4.	Jelinek	0.2113	0.1947	0.2560	0.3120	0.2027	0.2573
5.	BM25	0.2558	0.2322	0.2880	0.3480	0.2213	0.3013

Results obtained in Project 3:

No.	Retrieval Models	Mean Average Precision	Precision at 10 docs	Precision at 30 docs
1	Okapi-tf	0.2791	0.3269	0.1897
2	Okapi-tf-idf	0.3899	0.3827	0.2359
3	Laplace	0.2677	0.2923	0.1628
4	Jelinek	0.3003	0.3115	0.1872
5	BM25	0.3861	0.3596	0.2333

Analysis of the results

- Case sensitive search unlike while using Lemur index in project 2.
- Stemming increases relevant documents obtained with more precision.
- BM25 performs the best in comparison with other models.
- Okapi-tf-idf performs the best after BM25 in comparison with other models.
- Okapi-tf*idf performs the best when we deal with high volume processing. Giving the highest number of documents (1262) amongst all the models.
- The term frequency normalization used in BM25 model depends on a constant set to a term independent constant.
- Jelinek Mercer has higher recall in comparison with Laplace smoothing giving curves with lower slopes. Whereas laplace starts with a higher precision which drops as we retrieve more documents.
- Precision recall graph suggests that BM25 performs best in the beginning part of the retrieval process. Thus, we should BM 25 in case when we need to provide more relevant documents.

How System works

- The system is divided into two major phases :
    - Parsing CACM collection to create indexes
    - Calculation of data based on models
- **CACM Parsing**
    - First, the CACM collection from the *Search Engines: Information Retrieval in Practice* test collection site was downloaded then, each document was processed to extract the required relevant information.
    - Then, the data was filtered by removing punctuation marks and by converting them to lower case.
    - After that stopping and stemming operations on the data was performed.
    - For performing stemming operations, I have used **Porter Stemmer algorithm**. The stem() method of this algorithm is available in the python implementation
    - Each word from the document was used to build an inverted index.
    - First file was created to store, each unique term along with their unique term id, ctf, df and the offset value of their mapping to the second file and the length to be read from the second file in order to reach the given term.
    - Second file was created to store the mapping between the terms in first file and documents in the third file. It stored the term id, document id's of those documents in which the term is occurring and the frequency of its occurrence in those respective documents.
    - Third file was created to store all the documents with their doc id's, doc name and the doc length.
    - About 64 raw queries from CACM collection are used for query processing. Punctuations are removed from the queries. Stemming and stopping algorithm is applied to these queries. All these queries are then run against the inverted index obtained from the system.
    - Five different models are used to perform retrieval experiments on all the CACM queries and following results are obtained as shown in this report.

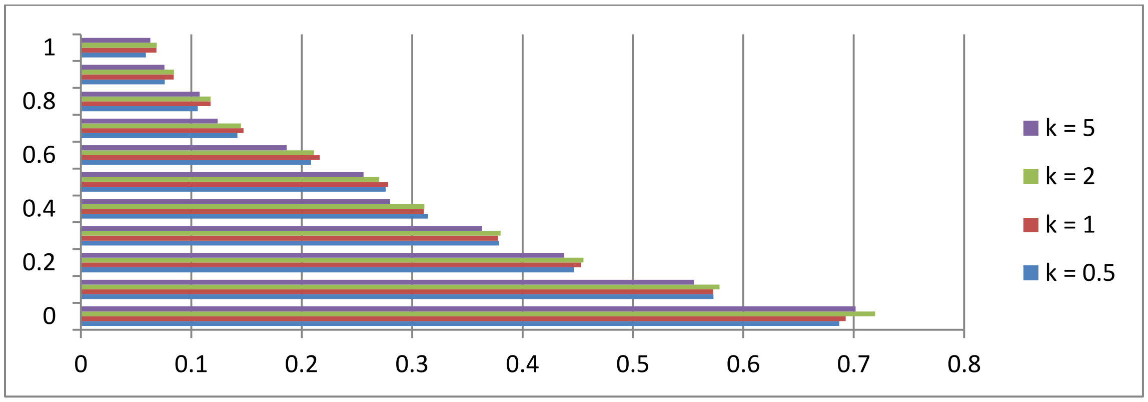
- **Model Computation**
    - The file 'cacm.query' is read and data is stored in a dictionary with the query number as the key.
    - An empty list is created and for each term in the 'cacm.query', we get the ID corresponding to that term and store its ctf and df. Then, using the offset value we store document ID, document length and the frequency of that term in that document (tf). The list is in the format: [ctf, df, documentID, document length, tf].
    - Then the scores for documents are computed using the five models:
        - Vector Space Model using Okapi tf,
        - Vector Space Model using Oakpi tf times IDF,
        - Language Model with Laplace Smoothing,
        - Language Model with Jelinek-Mercer Smoothing,
        - BM25 Model.
    - The top 1000 documents are computed based on the scores, stored in a file in the required format and given to the cacm.rel file for evaluation.

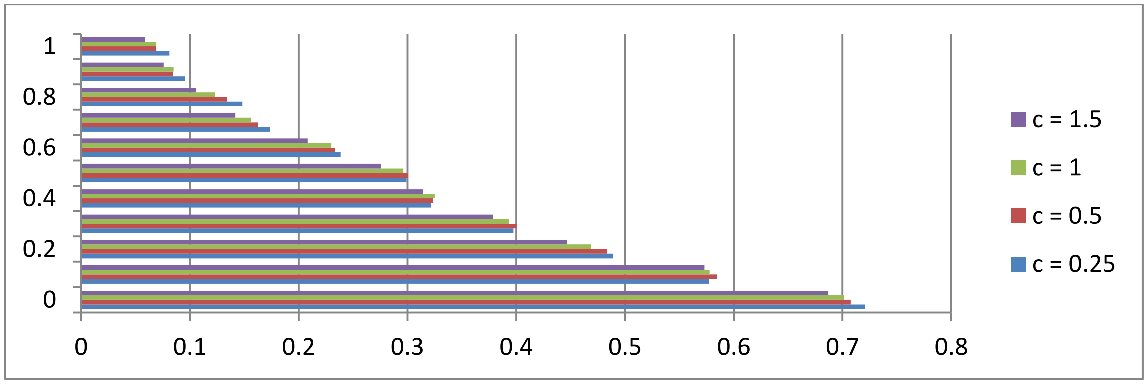
Results and Graphs

Okapi tf

Value of k at c = 1.5	Mean Average Precision
0.5	0.2791
1	0.2835
2	0.2849
5	0.2679

Value of c at k = 0.5	Mean Average Precision
0.25	0.3053
0.5	0.3019
0.1	0.2928
1.5	0.2791

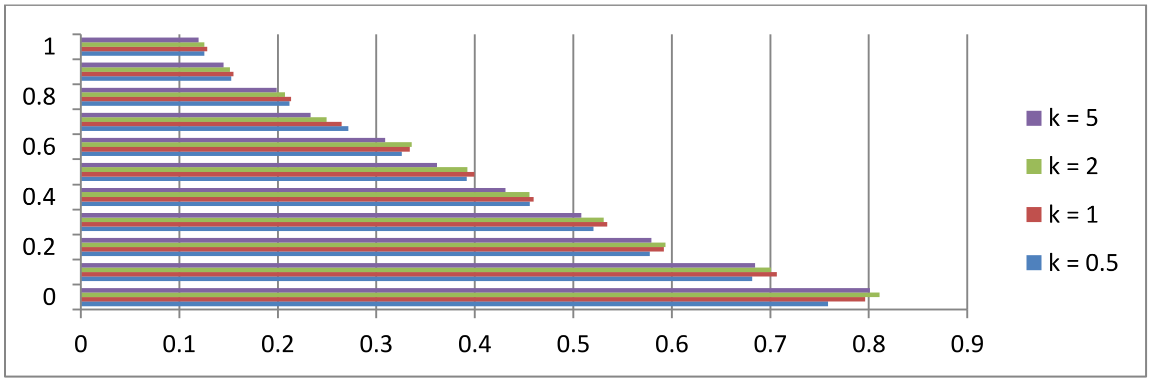


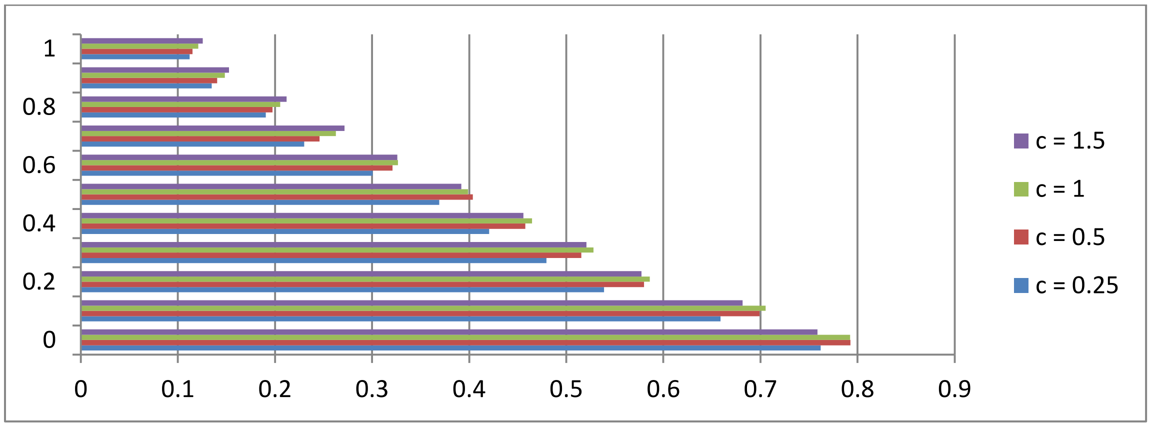


Okapi tf – idf

Value of k at c = 1.5	Mean Average Precision
0.5	0.3899
1	0.3965
2	0.3961
5	0.3792

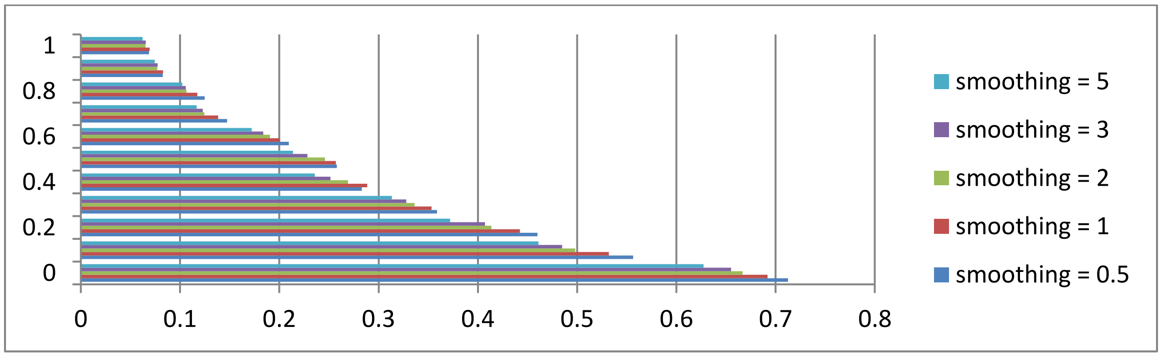
Value of c at k = 0.5	Mean Average Precision
0.25	0.3623
0.5	0.3869
0.1	0.3926
1.5	0.3899





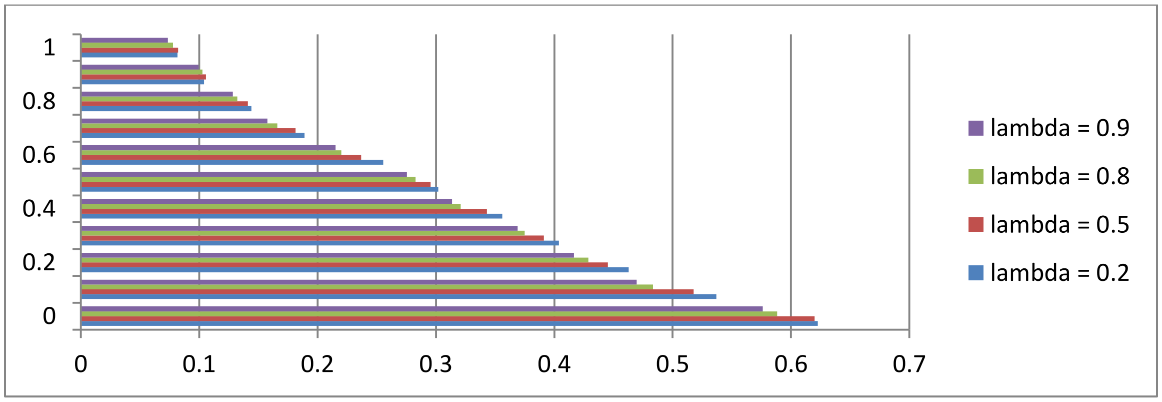
Laplace

Value of smoothing	Mean Average Precision
0.5	0.2759
1	0.2677
2	0.2536
3	0.2447
5	0.2313



Jelinek-Mercer Model

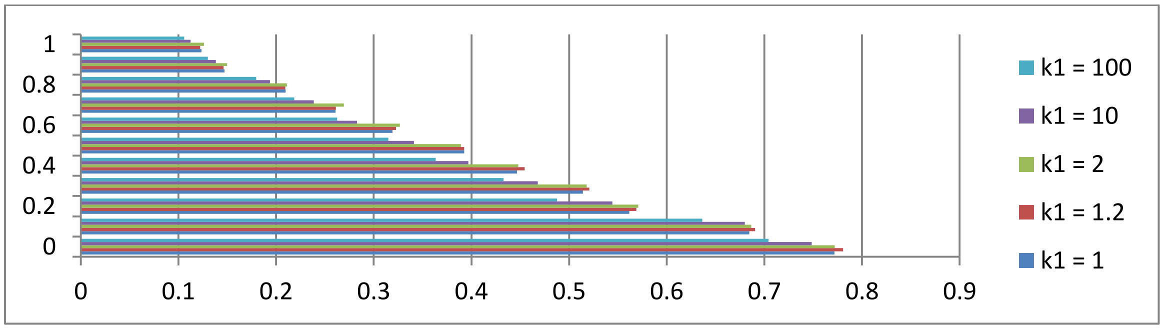
Lambda	Mean Average Precision
0.2	0.3003
0.5	0.2909
0.8	0.2744
0.9	0.2673

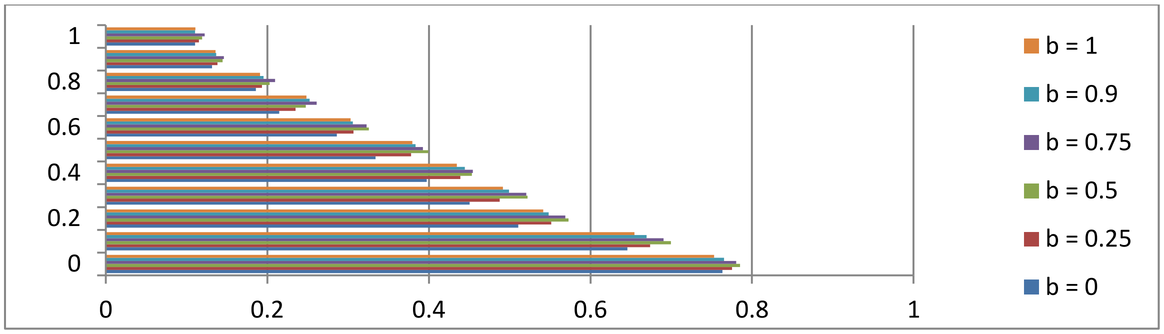


BM25 Model

Value of k1 for b = 0.75	Mean Average Precision
1	0.3838
1.2	0.3861
2	0.3895
10	0.3562
100	0.3269

Value of b for k1 = 1.2	Mean Average Precision
0.0	0.3464
0.25	0.3740
0.50	0.3883
0.75	0.3861
0.90	0.3723
1.0	0.3660





Smoothing

Smoothing helps generate common words in the query and at the same time avoids assigning zero probability to the unseen words in the document.

Dirichlet Smoothing

Formula:

N / (N + μ) * (Document) + μ / (N + μ) * (Collection)

Where, N: Document Length

μ: Constant

Reference:

http://ciir.cs.umass.edu/~metzler/indriretmodel.html

Witten Bell Smoothing:

Formula:

N / (N +V) * (Document) + V / (N + V) * (Collection)

Where N: Document length

V: Unique terms in the document

Reference:

http://nlp.stanford.edu/~wcmac/papers/20050421-smoothing-tutorial.pdf

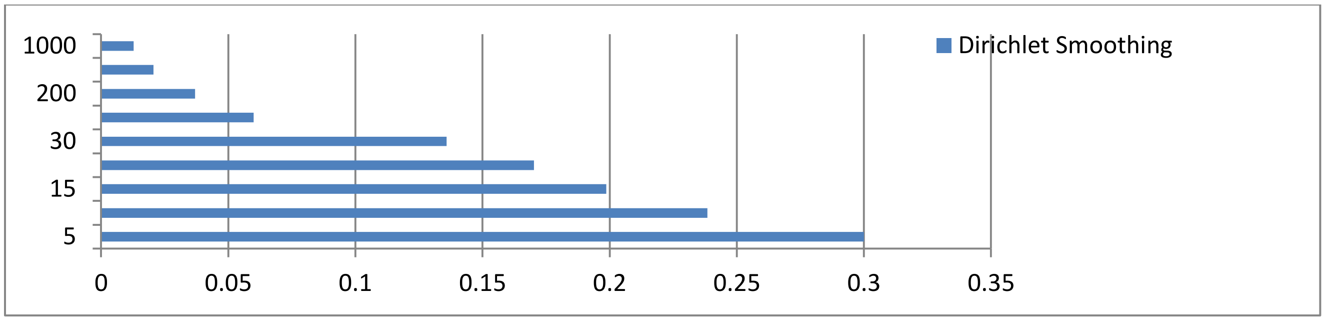
Implementation

Dirichlet Smoothing and Witten-Bell Smoothing

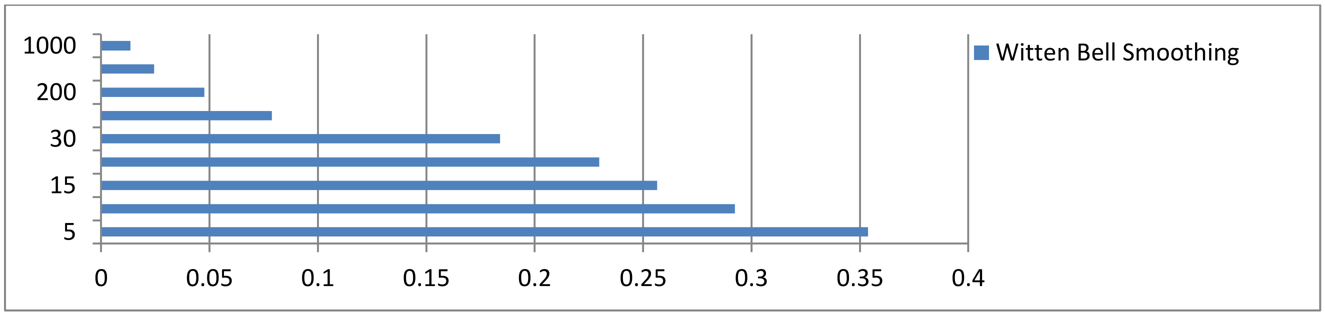
	Mean Average Precision	Precision at 10 docs	Precision at 30 docs
Witten Bell Smoothing (model6)	0.2858	0.2923	0.1840
Dirichlet Smoothing (model7)	0.2315	0.2385	0.1359

BM25 model and Vector Space Model 2 fare better compared with Dirichlet Smoothing and Witten Bell Smoothing.

Dirichlet Smoothing precision



Witten Bell Smoothing Precision



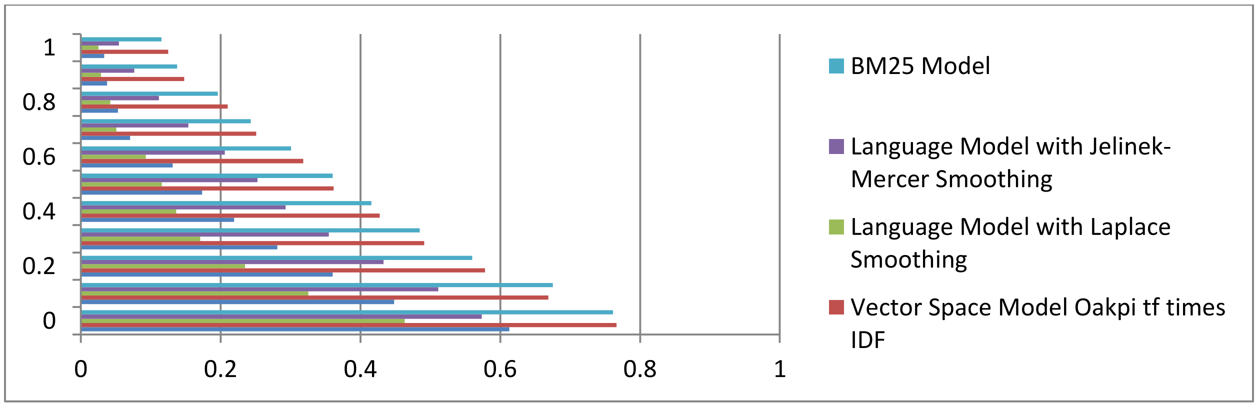
Extra Credit

As a part of extra credit I built the inverted index file by removing the stop words.

Removing stop words

Results on larger index without removing stop words from the query

No.	Retrieval Models	Mean Average Precision	Precision at 10 docs	Precision at 30 docs
1.	Okapi tf	0.2032	0.2635	0.1442
2.	Okapi tf - idf	0.3753	0.3788	0.2173
3.	Laplace Smoothing	0.1365	0.1692	0.1064
4.	Jelinek-Mercer Smoothing	0.2602	0.2885	0.1744
5.	BM25 Model	0.3676	0.3519	0.2205



Observations

Not removing stop words

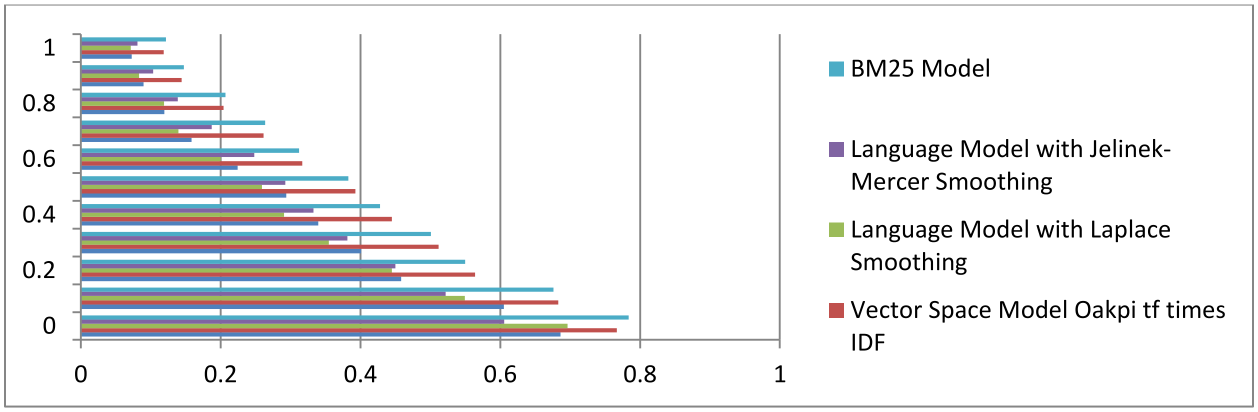
- Time required to create index ~ 40 seconds
- Number of unique terms: 8698
- Total number of terms: 212468
- Total number of documents: 3204
- Average document length: 66

Analysis

- For smaller index, Vector Space Model 2 and BM25 model give the best precision values while, Laplace Smoothing gives the worst precision values.
- Overall precision values are less compared to the ones obtained on the smaller index (due to not removing stop words from the query primarily).
- Inverted index file size increases by almost twice in case of not removing the stop words.

Not Removing stop words

No.	Retrieval Models	Mean Average Precision	Precision at 10 docs	Precision at 30 docs
1.	Okapi tf	0.2909	0.3212	0.1872
2.	Okapi tf – idf	0.3825	0.3750	0.2327
3.	Laplace Smoothing	0.2700	0.2923	0.1635
4.	Jelinek-Mercer Smoothing	0.2904	0.2923	0.1840
5.	BM25 Model	0.3772	0.3596	0.2282



Observations

Removing stop words

- Time taken to create index ~ 10 sec
- No of unique terms = 8480
- Total number of terms = 136225
- Total number of docs = 3204
- Average document length = 42

Analysis

- BM25 and Vector Space Model 2 give the best precision values while Laplace Smoothing gives the worst precision value.
- Building an index by removing or not removing the stop words does not affect the precision, but removing the stop words from the query has a lot of effect.
- Removing stop words gives you better precision values.

Additional Experiments

- Query processing time on smaller index is ~ 7 seconds when stop words are removed or nnot removed from the query.
- Query processing time on larger index is ~ 8 seconds when stop words are removed from the query.
- Query processing time on larger index is ~ 16 seconds when stop words are not removed from the query.

Using Compression

I have used compression using a B-tree data structure

- A B-tree contains multiple keys stored in normalized form which allows keys to be quickly compared and used in compressing.
- So in my search engine model we first compress the term file, then calculate offset and length using the values are stored in index file. During the query term search, I retrieve the offset and length value from the first file and send them to second file to fetch the data.
- Now the data fetched is in compressed format. Then I decompression can be applied on fetched data to retrieve the original data.
  Referred from ("Prefix B-Tree" ACM Transactions on Database Systems,Bayer,R,et.all.)

Additional analysis

Query Expansion

- In this method, additional synonymous terms to the query terms are added to the query which retrieves relevant documents.
- After query expansion on query 12 gives: Portable Compatible Transportable Operating System.
- It causes a change in the page rank order of documents indicating that adding a synonym of the term in the query increases weightage of a document that has both of this term in it as well as document id gets higher score.

Zipf's Law

- Ziphian model can be used generate the stopping list for a collection.
- All the documents in the collection are parsed to calculate the tf.
- Using the tf and applying ziphian parameters, stopping list can be developed.
- Words with very high frequency and words with very low term frequency can be removed.