CS6200 – Term Project – Building a Search Engine

Submitted by: Keith Mascarenhas – mascarenhas.k@husky.neu.edu
Date: April 20 2012.

Summary: In this project, the functionality of the Lemur index is replicated and used in conjunction with code created during Project 1 which implemented various retrieval functions, thus creating a full functional search engine.
I implemented the project in Python 2.7 using Eclipse as an IDE with the PyDev perspective plugin.

Building the functionality of the Lemur index.

Following are the steps involved in creating the functionality of the Lemur Index:

- Parsed the stop-word list from Project 1 to create a hash index of all the words present in that stop-list and pickle that hash map.
- Downloaded the Porter Stemmer for python and got it working.
- Downloaded the Glasgow version of the CACM collection and parsed the data so as to first stop and then stem every term encountered.
- An inverted-index text file was created for each of these and then all of these text files were combined into one large file that only contained document IDs and term-frequencies.
- Another file was used to access the offset for the words in these documents which also served as the term-id for each term.
- Several other files were created to map document-ids to their lengths and associated document information.
- A file was also created to map terms to their respective term-ids, their ctf/df counts.
- Also, files were created that contained total-corpus statistics like number of unique words, total number of words etc.

Parsing the stop-word list: The stop-word list was downloaded as a text file and then parsed so as to create a pickle of it. The pickle was then loaded during parsing and a check whenever a term was encountered, it was checked if the term existed as a key in the hash map, and if yes, the term was stopped. The hash-map facilitated looking up a key in O(1) time.

Using the Stemmer: The Porter Stemmer's Python version was downloaded and implemented. An object had to be created and the stem() function was used to stem the terms in the corpus.

Parsing the corpus: The Glasgow version was downloaded and parsed as below.

- On encountering the '.I' term, the docid to it's right was immediately recorded.
- Next, every term is recorded as a term besides the '.N', '.T', '.A' or '.B' terms.
- Each of these terms are first stopped by checking if they exist as a key in the stop-word pickle.
- If they are not in the stop list, the word is stemmed and a path to a file (with the file name same as the word) is created to the text file for that word. If the file already existed it is opened in append mode, else a new file is created by that name and is opened in append mode.
- The recorded docid is now appended to the currently opened file.
- On encountering the '.N' tem, the recording of terms is stopped until we encounter a '.I' again.
- This is done so as to ignore the bibliographic and crawl time

Note: One interesting point here is that no file can be created in Windows with a name as 'con'. Therefore, my IDE threw an error every time it attempted to create a file by the name 'con.txt'. Consequently, I decided to prefix a 'z' at the start of every filename so as to facilitate the creation of 'zcon.txt'.

Combining the files into one inverted index: After creating thousands of text files for each word, all of this data was combined into one single inverted index file that held docid:term-frequency pairs for each term. I went about doing the same in the following manner.

- A for loop iterated over each file in the folder in which all of these terms were kept. The term was recorded by taking a substring of the filename of the text file.
- This file was then opened and each term in it was processed.
- A temporary hash-map was created and the document-ids encountered in the file were recorded as keys in the hash-map. And for each occurrence of that document-id in the file, the corresponding key value in the hash was updated so as to reflect the number of occurrences of each of the document-ids in the file.
- Using the tell() function, the current offset in the inverted-index-file was now recorded to as to facilitate easy access at later times. This offset was also recorded as a term id since it would be unique for each term. Correspondingly, the termed-term hash-map was updated and also the term-term-offset hash-map, both of which were essentially the same.
- Next, I iterated over the keys in each of these hash maps and appended to the inverted-index-file document-ids and term-frequency pairs.
- On reaching the end I wrote the ' | ' as a delimiter to mark the end of one term and start of information for another term.
- During this process, the document frequency for each term and cumulative term frequencies are also calculated, recorded in hash-maps and pickled for easy use later.

Parsing the query files:

- I first manually edited the query file to remove terms that were not useful in retrieving relevant files. (This is explained in detail in a later section).
- The '.I' term is identified again the '.N' term marks the end of a query.
- We again stop and stem words in the query just as we do in the corpus-parsing.
- Finally we have a matrix 64 rows each which hold an array of terms held in the query.

Lemur API functionality:

After creating the inverted index, I built a set of functions that provided the functionality of Lemur.

- A function called invlist(word) takes the word, find it's termid, it's corresponding term-offset and then pulls data from the inverted-index by using the seek() function.
- It pulls data from the text file until it reaches the ' | ' delimiter.
- We now have a string of terms, which we split() to get an array of terms which are docid-tf pairs.
- So we now iterate over this list and create a hash-map which holds these docid-tf pairs and finally return this hash-map as the inverted list for the word.
- Similary, I created functions to retrieve the termid, ctf, df, uniquewords, totalwords and idf so as to facilitate code refactoring and avoid redundancies.

Optimizations:

Additional stop words: After creating the inverted index, I analysed the top word occurences in the entire corpus and based on Zipf's law omitted some of the words that occurred most frequently.

Tem	Number of occurrences
march	3024
cacm	3024
1978	3016
jb	3001
pm	2220

These were five of the most frequent words that occurred in the corpus and adding them to the stoplist boosted the overall performance of all my systems by a significant score.

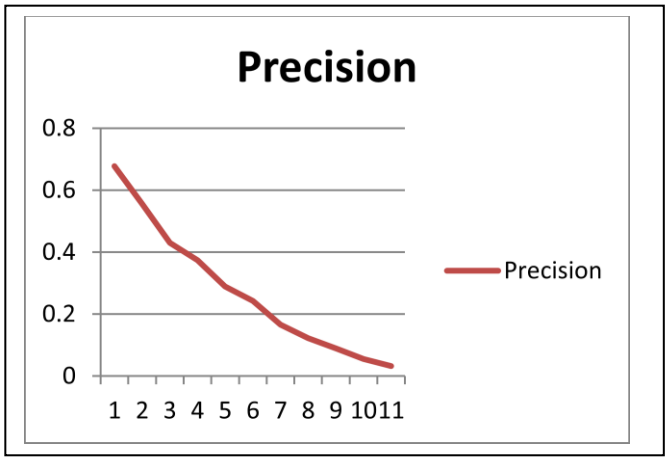
Manually edited queries: I manually edited the 64 queries so as to make the system more efficient. Although the queries were stop worded, several words had to removed from the queries to make them more effective at retrieving relevant documents. There were several conversational queries that had to be corrected. Below is a list of improvement types made to the queries.

- Remove conversational text like: 1: What articles deal with…?
  2,4,5,6,7..: I'm interested in articles written…
  14,15,16..: Find all discussions…
  30,31,..: Articles on…
- Very/specifically interested in: Several queries mention topics that they are specially interested in, however I omit the words 'specific, very interest, relevant' etc. as they add no knowledge to the query.
- Not related/irrelevant topics: Queries also mention topics that they are not interested in and they do not wish to be returned by the search engine. As a result I have completed omitted those words from the queries so as to avoid anything relevant to those terms being retrieved by the system.
- Bibliographic info: Since I am not parsing the bibliographic information for every query, I also do not parse any of the bibliographic information wherever mentioned for some of the queries. This helps improve the query score.
- Intelligently add query words based on what user is seeking:
  Some of the queries like 30, 64, 40 require you to read the query and manually add terms that are not present in the original query.

Uninterpolated mean average precision for all 5 systems.

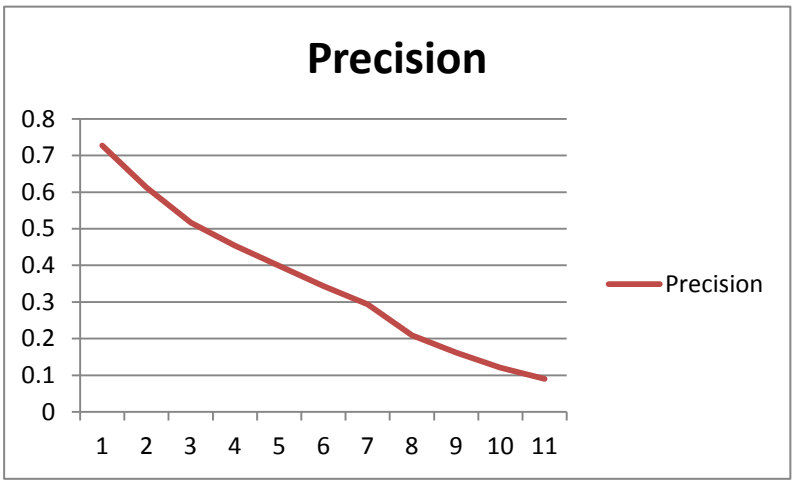
1. Vector space model without any weighting.

Recall	Precision
0	0.6771
0.1	0.5564
0.2	0.43
0.3	0.3735
0.4	0.2887
0.5	0.2427
0.6	0.1656
0.7	0.1222
0.8	0.0895
0.9	0.0543
1	0.032



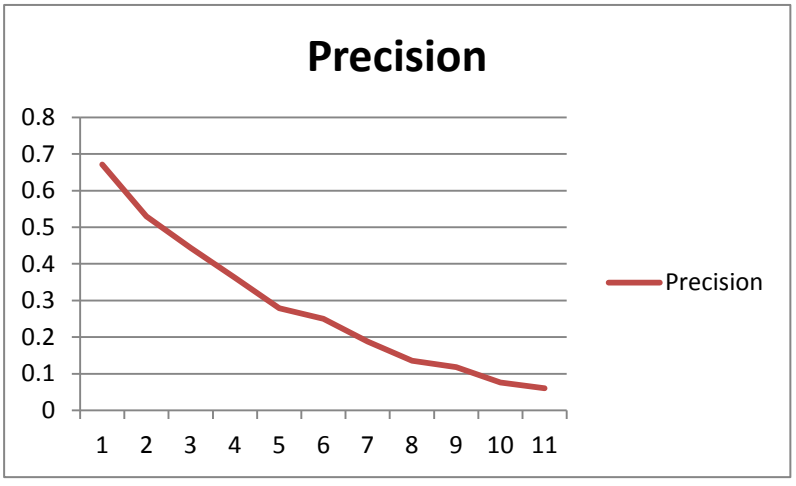
2. Vector space model, weighted by Okapi tf * IDF.

Recall	Precision
0	0.7269
0.1	0.6129
0.2	0.5164
0.3	0.4538
0.4	0.3984
0.5	0.3433
0.6	0.2934
0.7	0.2086
0.8	0.1616
0.9	0.1212
1	0.0899



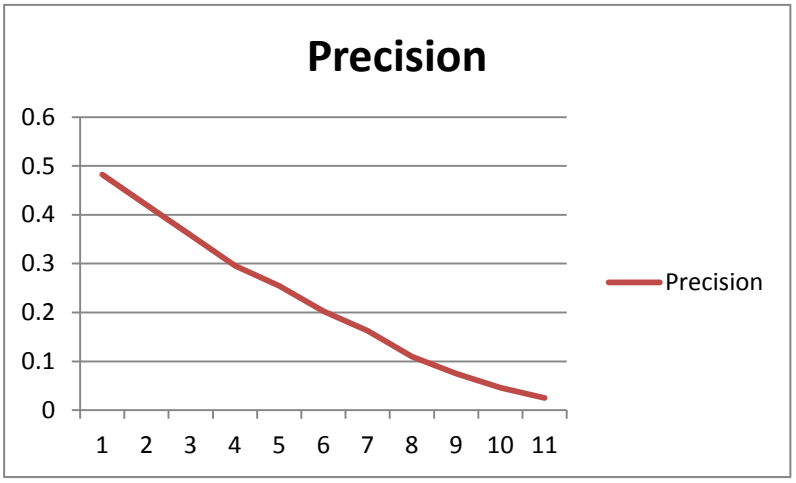
3. Language modelling, maximum likelihood estimated with Laplace smoothing only.

Recall	Precision
0	0.6713
0.1	0.5298
0.2	0.4436
0.3	0.362
0.4	0.2787
0.5	0.2499
0.6	0.1884
0.7	0.1357
0.8	0.1185
0.9	0.0763
1	0.0603



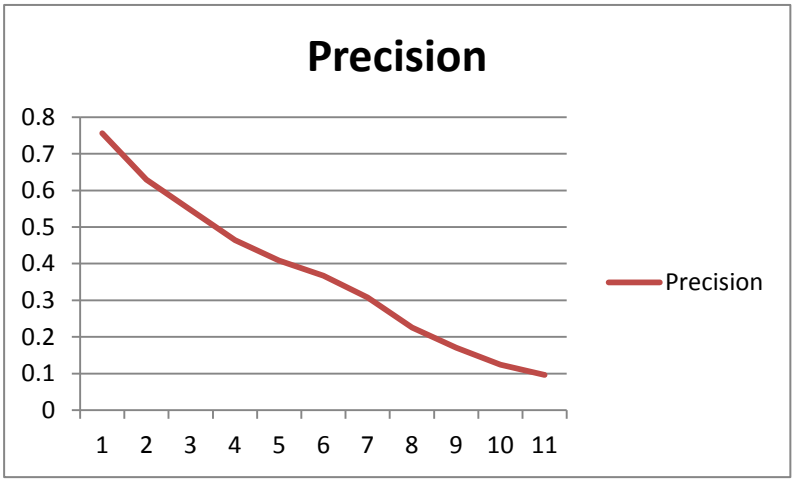
4. Language modelling, Jelinex-Mercer smoothing using 0.3 weight

Recall	Precision
0	0.4823
0.1	0.4202
0.2	0.3581
0.3	0.296
0.4	0.2544
0.5	0.2029
0.6	0.1624
0.7	0.1093
0.8	0.0751
0.9	0.046
1	0.0249



5. BM25 Model

Recall	Precision
0	0.7559
0.1	0.6295
0.2	0.5464
0.3	0.4644
0.4	0.4083
0.5	0.3671
0.6	0.3079
0.7	0.2259
0.8	0.1705
0.9	0.1239
1	0.0959



Observations:

I noticed that the results are very similar to the results from project 1.

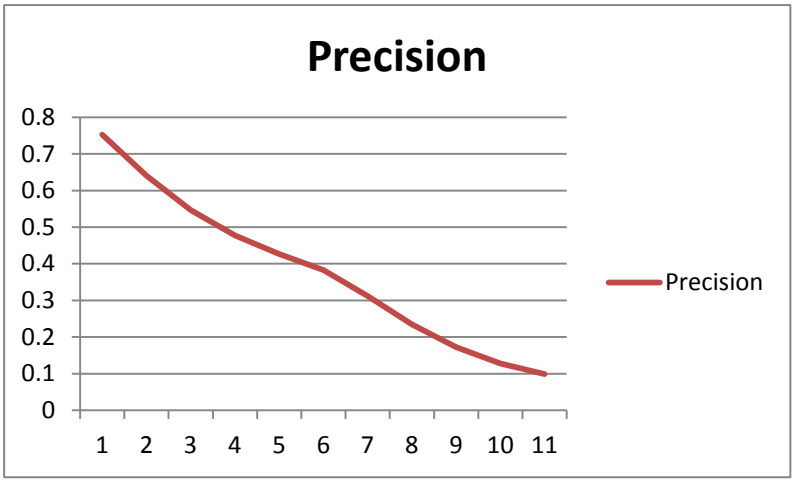
BM25 fares the best with Vector (weighted IDF) coming a close second.

Because, we stopword, stem and also manually edit a lot of the queries, the relative quality of precision on this CACM corpus is much much better as compared to the results from Lemur's TREC database.

EXTRA CREDIT MODELS:

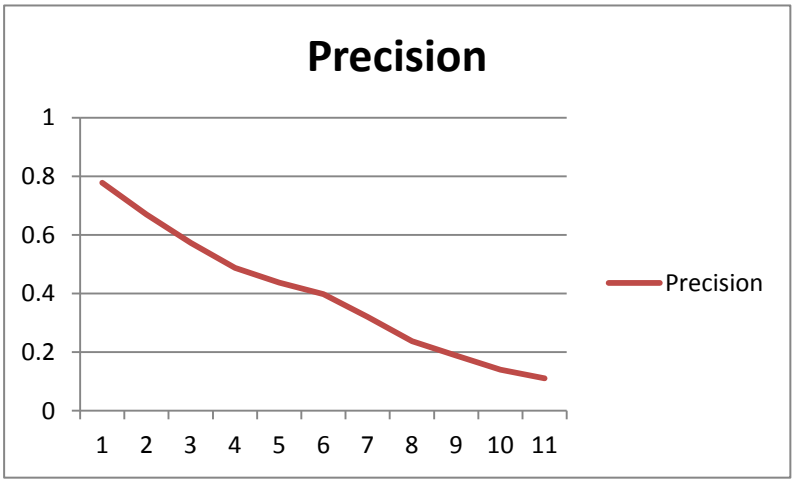
6. Metasearch of 2 best systems: BM25 + Vector space weighted by IDF

Recall	Precision
0	0.7524
0.1	0.6403
0.2	0.5467
0.3	0.4771
0.4	0.427
0.5	0.3826
0.6	0.312
0.7	0.234
0.8	0.1719
0.9	0.1281
1	0.0988



7. Metasearch of 4 systems: BM25 + Vector (weighted) + Jelinek + Laplace
   (Vector-unweighted is omitted because it is the same as Vector (weighted) but of lower quality scores).

Recall	Precision
0	0.7774
0.1	0.6702
0.2	0.5727
0.3	0.487
0.4	0.4376
0.5	0.3973
0.6	0.3203
0.7	0.2369
0.8	0.1876
0.9	0.1402
1	0.1109



Observations:

I combined the 2 best models, BM25 and Vector (weighted) using COMBSum to produce a new system that did much better than each of the 2 individually.

It is my understanding that since the 2 systems are very different in working from each other, they bring different types of knowledge to the table and thus combine to produce an optimal result.

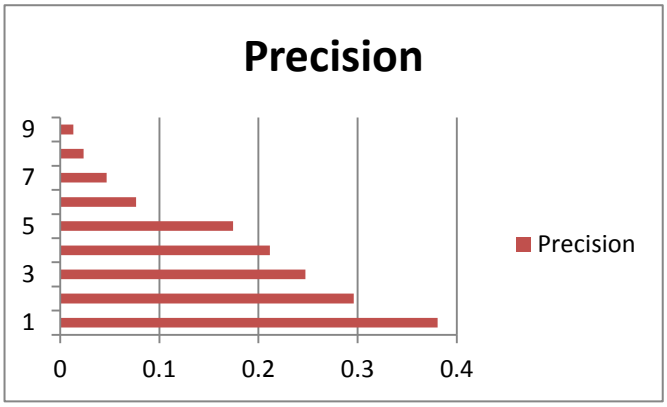
I also attempted to combine all 5 models together. However I left out Vector (unweighted) because it was only a poorer version of the Vector (weighted IDF) model and thus only brought down the score and it did not give any new knowledge.

All five systems together gave a very high score 0.375 and this proves that a combination of several search engines using meta search is very useful.

Precision at 10 and 30 documents for all models.

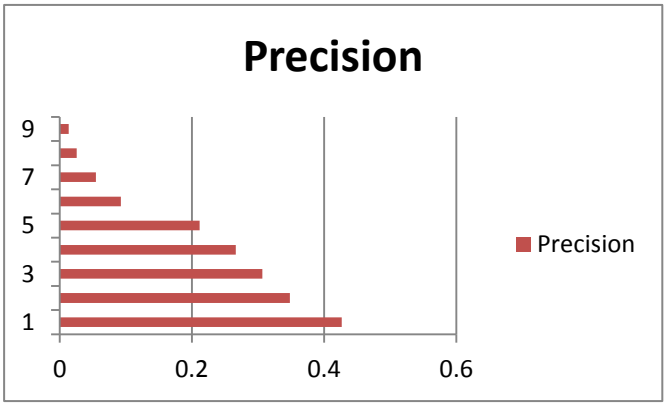
1. Vector Space Model without weighting.

At x Docs	Precision
5	0.3808
10	**0.2962**
15	0.2474
20	0.2115
30	**0.1744**
100	0.0767
200	0.047
500	0.0237
1000	0.0133



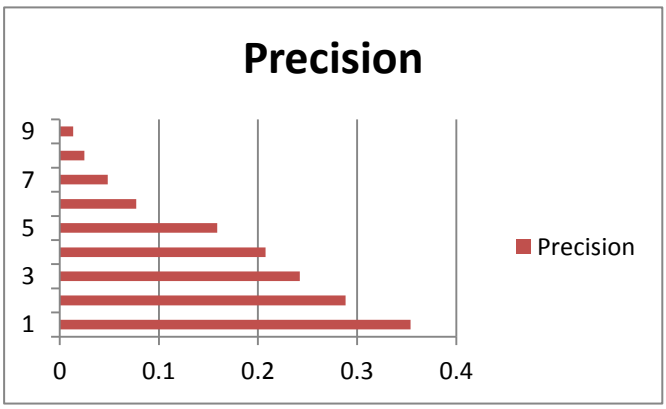
2. Vector Space Model with Okapi tf * IDF weighting

At x Docs	Precision
5	0.4269
10	**0.3481**
15	0.3064
20	0.2663
30	**0.2115**
100	0.0927
200	0.0548
500	0.0257
1000	0.0136



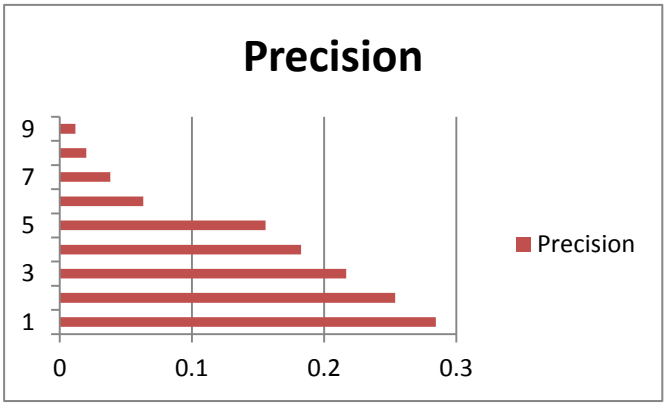
3. Language modelling, Maximum likelihood estimates with Laplace smoothing

At x Docs	Precision
5	0.3538
10	**0.2885**
15	0.2423
20	0.2077
30	**0.159**
100	0.0771
200	0.0484
500	0.025
1000	0.0136



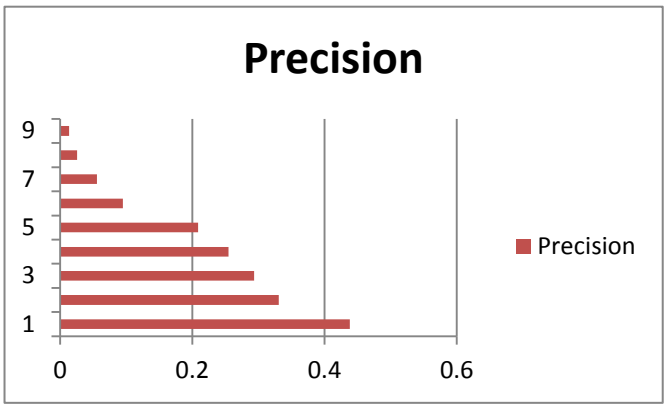
4. Language Modelling, Jelinek-Mercer smoothing using 0.3 weighting.

At x Docs	Precision
5	0.2846
10	**0.2538**
15	0.2167
20	0.1827
30	**0.1558**
100	0.0633
200	0.0383
500	0.0202
1000	0.0119



5. BM25 Model

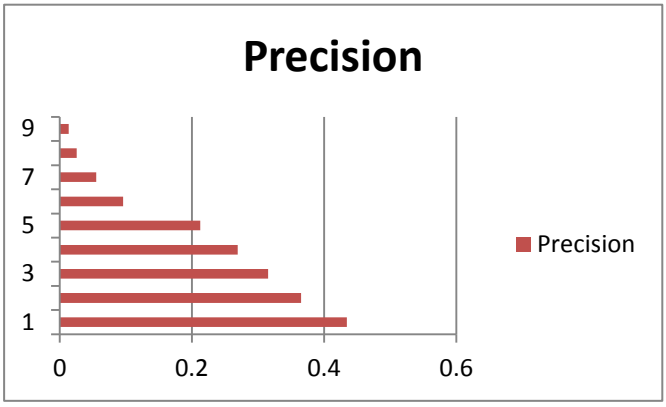
At x Docs	Precision
5	0.4385
10	**0.3308**
15	0.2936
20	0.2548
30	**0.209**
100	0.0948
200	0.0558
500	0.0258
1000	0.0137



EXTRA CREDIT MODELS: Metasearch

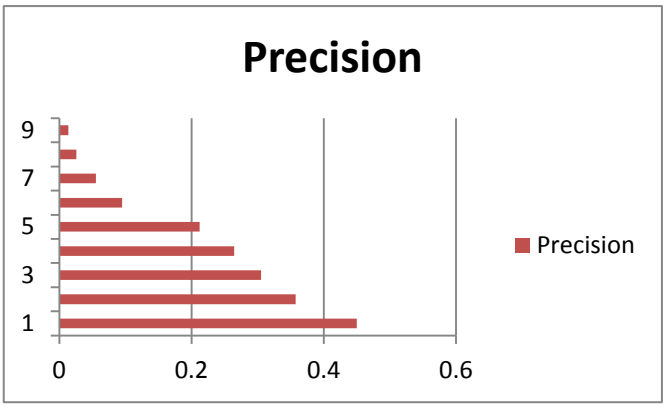
6. Metasearch of combiniation of 2 best models: BM25 + Vector weighted by IDF

At x Docs	Precision
5	0.4346
10	**0.3654**
15	0.3154
20	0.2692
30	**0.2128**
100	0.096
200	0.0555
500	0.026
1000	0.0136



7. Metasearch combo of 4 miodels. BM25 + Vector (weighted) + Jelinek + Laplace

At x Docs	Precision
5	0.45
10	**0.3577**
15	0.3051
20	0.2644
30	**0.2122**
100	0.095
200	0.0554
500	0.026
1000	0.0136



Observations:
I observed that again this system using the CACM corpus fared much better than the previous TRC corpus.

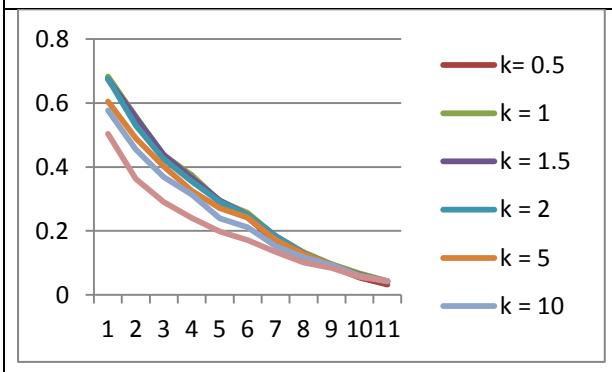
The systems achieve a much higher amount of precision even at very high recall whereas in the TREC systems this would be 0.

In my opinion this is because the TREC was a very large Corpus and as a result there were several thousands of documents that were relevant to each query. As a result, as the recall increased, the precision kept reducing rapidly.

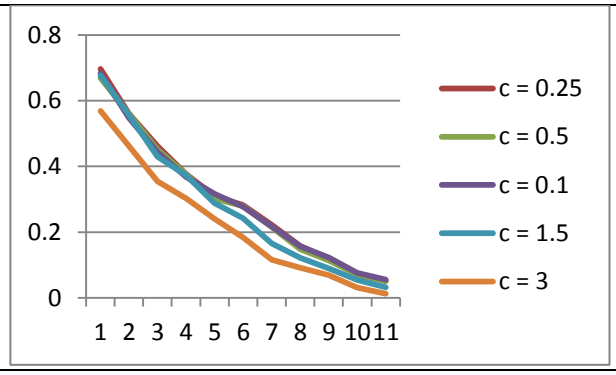
Further analysis of each system by changing the values of the constants:

1. Vector Space model without weighting.

For different values of k

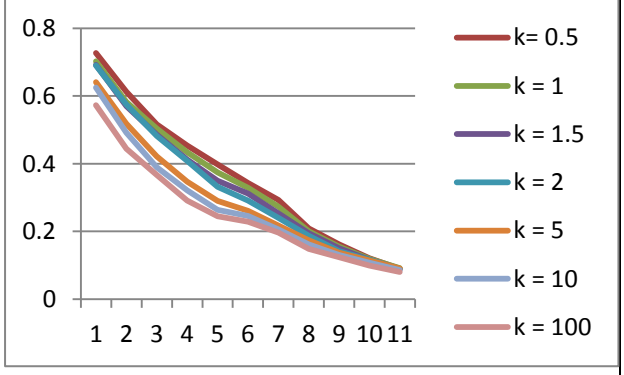


For different values of c

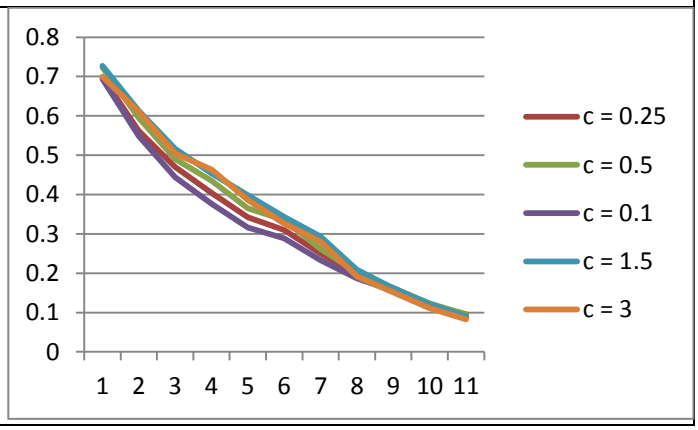


2. Vector space model using okapi * IDF weighting

For different values of k

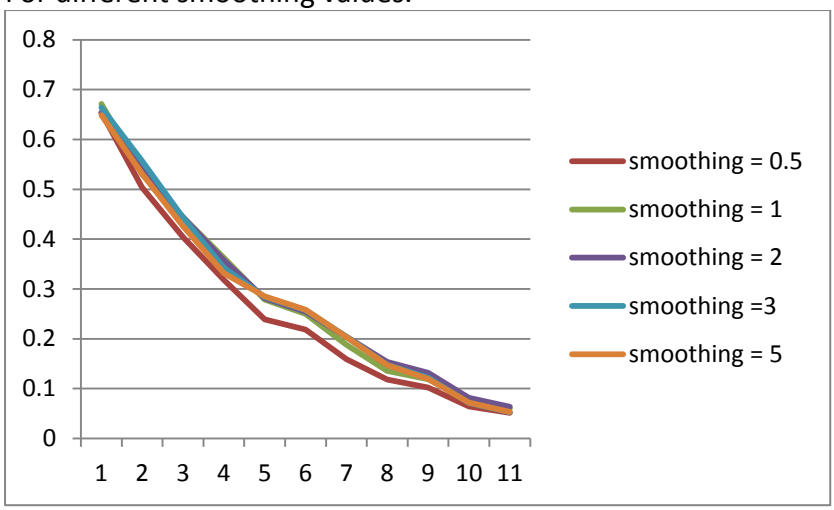


For different values of c



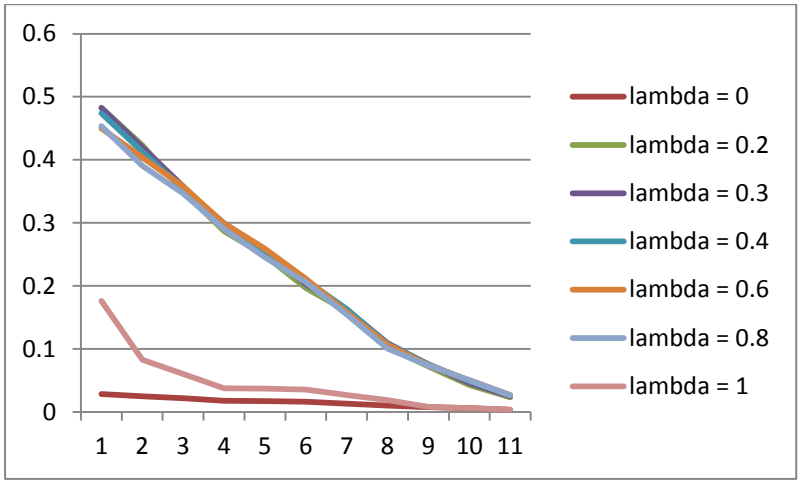
Observations in Vector Space Models: For small changes in k, no significant changes are noticed. However for large values of k, Precision decreases since the denominator dominates and all documents start reaching the same range of scores.

3. Language modelling, maximum likelihood estimates with Laplace smoothing.
   For different smoothing values.

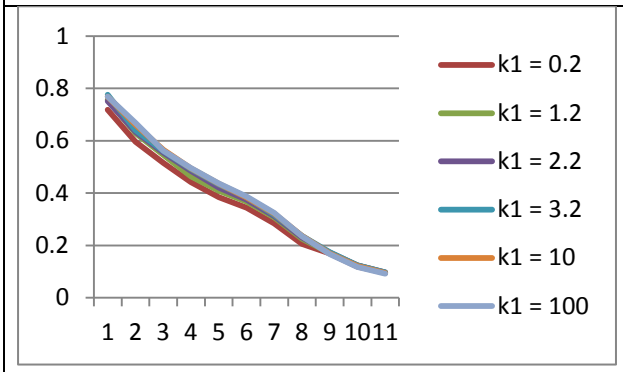
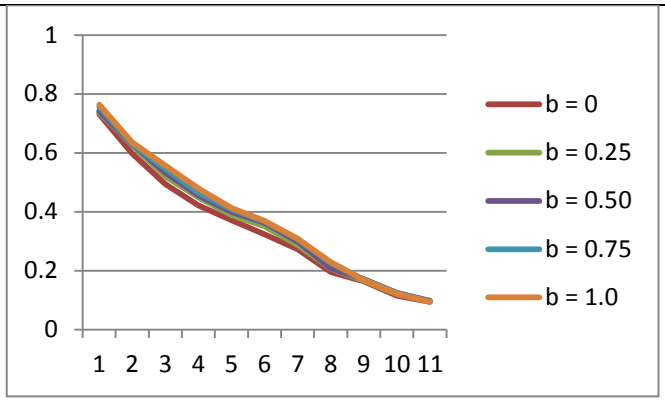


Observations: It is noticed that for very small and very large values of smoothing performance degrades. I observed that the smoothing should ideally be around the value of “1” or the average term frequency for all the terms in the corpus.

4. Language modelling, Jelinek-Mercer smoothing.
   For different values of lambda.



5. BM25

For different values of k1	For different values of b
	

EXTRA CREDIT.

I also parsed the corpus and queries so as to separate the Authors and create a different inverted index for it.

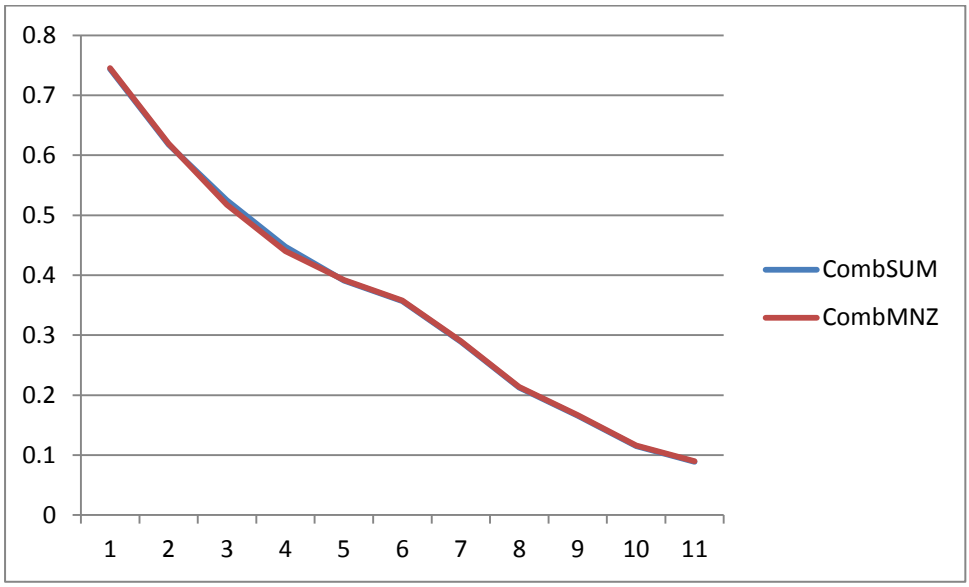
I then combined both these models using CombSUM and CombMNZ.

However, since there are only 8 queries that have author relevant information, there wasn't a significant increase in the overall scores of all the queries.

A few things that I did differently here was, I did not stop and stem the Author corpus or the author queries.

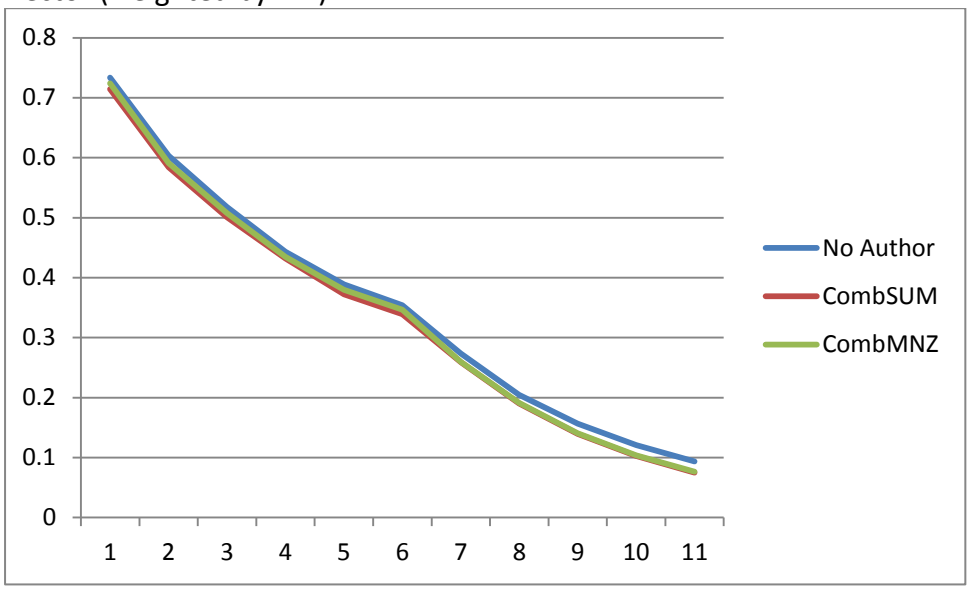
I had tried stopping the initials of authors but noticed that I got poorer scores when removing the initials. I thus came to the conclusion that even the initials of each author contribute to the information of the corpus.

1. BM25. Author + Text.



This is the BM25 model where the Author is separated from the data and a separate query is made for authors.
Since only 8 queries have authors out of the 64 there is a very small difference.
Thus, the CombSUM and CombMNZ scores are very similar to each other and their graphs almost overlap.

2. Vector (weighted by IDF)



On applying it using the weighted IDF Vector model, we notice that the system actually does better without the Authors.

Additionally, it is seen here that CombMNZ fares a little better than CombSUM in this case.

I believe this is so because as opposed to BM25, the background probability is not considered here.