CS6200, Information Retrieval

Team:
1. **Daniel Daskivich**
2. **Ankur Bambharoliya**

Instructor: Rukmini Vijaykumar
Semeter: Spring, 2019

Approaches to Information Retrieval

For this project, we designed and built two separate information-retrieval systems: one using the Apache Lucene framework and one implemented from scratch in Python. While our Python-based system implements a basic BM25 retrieval model, our Lucene-based system is built for flexibility, allowing users to combine various configurations among three retrieval models and two query-expansion techniques, with or without stopping. Scores and ranks are generated for each batch of queries run and saved in a text file per model/configuration; such raw results files can be found in the 'retrieval_results' directory in the project submission.

To evaluate the effectiveness of these models and query expansion techniques, we implemented four evaluation algorithms from scratch in Python: mean average precision, mean reciprocal rank, precision at k (5, 20), and recall-versus-precision comparison. For each batch of queries run for each model configuration, raw statistical results are stored in a text file and an HTML recall-versus-precision line graph is generated with one line for each query; such files and graphs can be found in the 'evaluation_results' and 'recall_v_precision_graphs' directories in the project submission. Raw recall-versus-precision data has been collated in the 'recall_v_precision_tables.xlsx' spreadsheet included in the project submission.

Finally, we implemented a snippet generator in Java, integrated with our Lucene-based system, to create an HTML document with highlighted snippets for a particular query's results; an example output file for the last of the supplied queries, called 'snippets.html', is included in the project submission.

Ankur built both retrieval systems, implementing the BM25, TFIDF, and query likelihood (with Dirichlet smoothing) retrieval models from scratch and incorporated Lucene's default retrieval model and various other custom-modified functionalities (spell-checking, stemming, pseudo-relevance, and stopping) within this framework. Daniel built the snippet generator and evaluation system, implementing all algorithms therein; he also generated most of the results, evaluations, and graphs referenced in this report. Both Ankur and Daniel contributed to writing this report.

Base Models

We chose to implement TFIDF and a query likelihood model with Dirichlet smoothing from scratch in Java within our Lucene-based retrieval system. Lucene's default retrieval model was incorporated into this system as well, and we implemented BM25 from scratch within our Python-based retrieval system.

The TFIDF score for document D given query Q is calculated as follows...

$$tf \cdot idf(D|Q) = \sum_{q \in Q} \frac{f_{q,D}}{|D|} \cdot log \frac{N}{n_q}$$

...where $f_{q,D}$ is the frequency of term q in document D, $|D|$ is the length of the document, N is the number of documents in the corpus, and n_q is the document frequency of term q.

The score for query Q given document D using our query likelihood model with Dirichlet smoothing is calculated as follows…

$$log\ P(Q|D) = \sum_{i=n}^{n} log \frac{f_{q_i,D} + \mu \frac{c_{q_i}}{|C|}}{|D| + \mu}$$

...where $f_{q_i,D}$ is the frequency of term q_i in document D, μ is the smoothing factor, c_{q_i} is the frequency of term q_i in the collection, $|C|$ is the total size of the collection in words, and $|D|$ is the length of the document. For our implementation, we used a generally accepted μ value of 2,000.[1]

The BM25 score for query Q is calculated as follows…

$$\sum_{i \in Q} = log \frac{1}{(n_i + 0.5) \div (N - n_i + 0.5)} \cdot \frac{(k_1 + 1) \cdot f_i}{K + f_i} \cdot \frac{(k_2 + 1) \cdot qf_i}{k_2 + qf_i}$$

...where N is the number of documents in the corpus, n_i is the document frequency for query term i, f_i is the frequency of query term i in the document, qf_i is the frequency of query term i in the query, and k_1, k_2, and K are parameters whose values are set empirically.

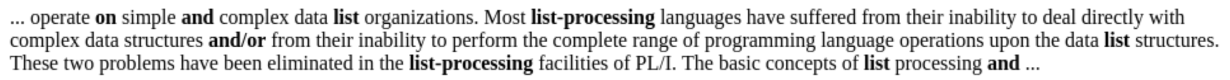
Query Enhancement

To explore possible ways to improve the effectiveness of these retrieval models, we incorporated the following three query-enhancement techniques: stopping, pseudo-relevance, and Kstemming. Stopping was implemented in our Lucene-based system by passing a list of stop words to Lucene's StopAnalyzer. Pseudo-relevance was implemented by pre-analyzing the top ten documents from an initial round of retrieval results, identifying the top ten words from these documents using Dice's Coefficient, and then expanding the original query with these additional ten words. Kstemming was implemented by using Lucene's KStemFilter class to expand the original query terms with additional whole words sharing a stem with an original query term. In particular, we chose Kstemming over Porter Stemming as our preferred method of stemming due to Kstemming's documented effectiveness[2] and its specific usefulness at query time. Porter stemming may be somewhat useful, but it can often add stemmed-version of words that aren't real words, aren't contained in a non-stemmed index, and sometimes even creates false-positive stems-- stems that mean something different from the origin word.

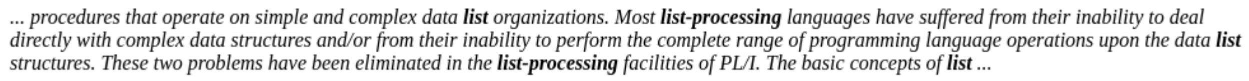
Snippet Generation

Snippet generation was implemented by adapting H.P. Luhn's significance factor approach[3] as follows. First, we defined a snippet to be the most significant stretch of 60 consecutive words from a document. We felt that longer snippets would lose relevance to the user, presenting too much information to sift through; conversely, snippets shorter than 60 words would not provide enough context to distinguish among the choices presented. To identify a snippet, we considered up to 30 of the most significant terms from the query. Query-term significance was determined by comparing each query term's inverse document frequency; the higher the IDF, the higher the significance.[4] Each possible 60-word window in a document was then scored by summing the IDF for each significant query term within that window. The earliest occurrence of the highest-scored window in the document was used as the snippet.

Our approach differed from Luhn's approach in that ours was not intended to identify topic sentences to summarize a document, as was Luhn's, but to identify the portion of the document that will best satisfy a searcher's specific information need-- using the only direct insight into the user's need that we have, the query terms themselves. For the purpose of displaying a fixed-length snippet, as most web search engines do (to present a uniform graphic interface to the user), the density of significant words was not as important as the overall significance within that fixed length. Furthermore, the complexity of parsing documents into sentences, as is done in Luhn's approach, did not seem to significantly to the contextual comprehension of the searcher; most internet searchers these days can extrapolate meaning from incomplete sentences.


... operate **on** simple **and** complex data **list** organizations. Most **list-processing** languages have suffered from their inability to deal directly with complex data structures **and/or** from their inability to perform the complete range of programming language operations upon the data **list** structures. These two problems have been eliminated in the **list-processing** facilities of PL/I. The basic concepts of **list** processing **and** ...

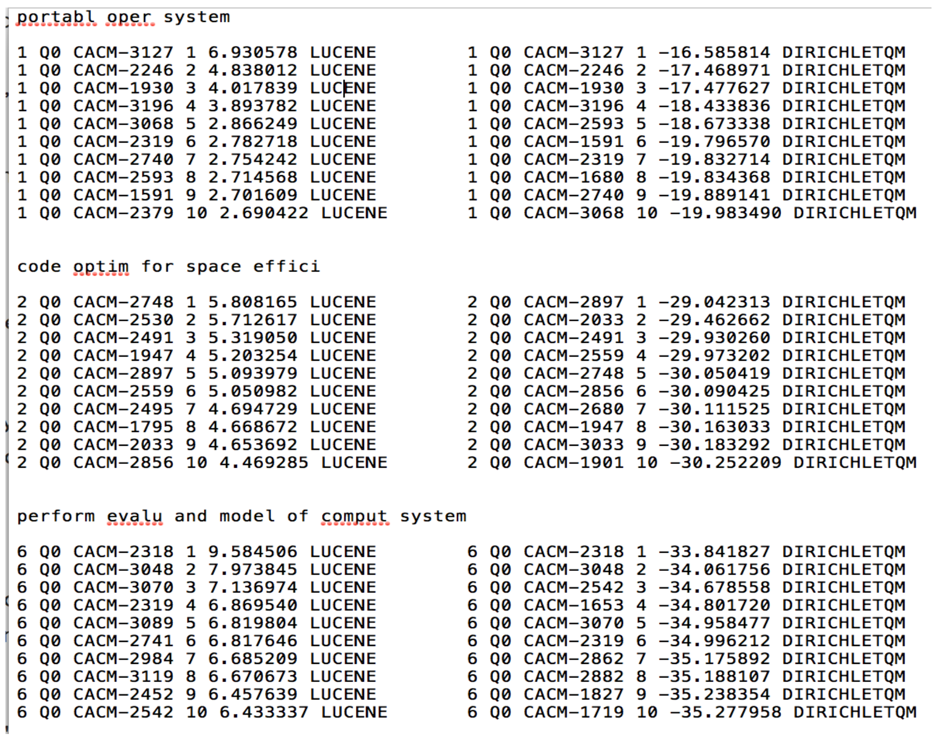
snippet before removing stop words


... procedures that operate on simple and complex data ***list*** *organizations. Most* ***list-processing*** *languages have suffered from their inability to deal directly with complex data structures and/or from their inability to perform the complete range of programming language operations upon the data* ***list*** *structures. These two problems have been eliminated in the* ***list-processing*** *facilities of PL/I. The basic concepts of* ***list*** *...*

snippet after removing stop words

Stemmed Queries & Corpus

The following side-by-side comparisons of three stemmed queries of the stemmed corpus using Lucene's default retrieval model and our query likelihood model with Dirichlet smoothing show that many queries result in very similar top rankings for both small (the first) and larger (the third) queries, with some variation further down the list. There can also be variation at even the highest ranks, however, as demonstrated by the second query below.


```
portabl oper system

1 Q0 CACM-3127 1 6.930578 LUCENE        1 Q0 CACM-3127 1 -16.585814 DIRICHLETQM
1 Q0 CACM-2246 2 4.838012 LUCENE        1 Q0 CACM-2246 2 -17.468971 DIRICHLETQM
1 Q0 CACM-1930 3 4.017839 LUCENE        1 Q0 CACM-1930 3 -17.477627 DIRICHLETQM
1 Q0 CACM-3196 4 3.893782 LUCENE        1 Q0 CACM-3196 4 -18.433836 DIRICHLETQM
1 Q0 CACM-3068 5 2.866249 LUCENE        1 Q0 CACM-2593 5 -18.673338 DIRICHLETQM
1 Q0 CACM-2319 6 2.782718 LUCENE        1 Q0 CACM-1591 6 -19.796570 DIRICHLETQM
1 Q0 CACM-2740 7 2.754242 LUCENE        1 Q0 CACM-2319 7 -19.832714 DIRICHLETQM
1 Q0 CACM-2593 8 2.714568 LUCENE        1 Q0 CACM-1680 8 -19.834368 DIRICHLETQM
1 Q0 CACM-1591 9 2.701609 LUCENE        1 Q0 CACM-2740 9 -19.889141 DIRICHLETQM
1 Q0 CACM-2379 10 2.690422 LUCENE       1 Q0 CACM-3068 10 -19.983490 DIRICHLETQM


code optim for space effici

2 Q0 CACM-2748 1 5.808165 LUCENE        2 Q0 CACM-2897 1 -29.042313 DIRICHLETQM
2 Q0 CACM-2530 2 5.712617 LUCENE        2 Q0 CACM-2033 2 -29.462662 DIRICHLETQM
2 Q0 CACM-2491 3 5.319050 LUCENE        2 Q0 CACM-2491 3 -29.930260 DIRICHLETQM
2 Q0 CACM-1947 4 5.203254 LUCENE        2 Q0 CACM-2559 4 -29.973202 DIRICHLETQM
2 Q0 CACM-2897 5 5.093979 LUCENE        2 Q0 CACM-2748 5 -30.050419 DIRICHLETQM
2 Q0 CACM-2559 6 5.050982 LUCENE        2 Q0 CACM-2856 6 -30.090425 DIRICHLETQM
2 Q0 CACM-2495 7 4.694729 LUCENE        2 Q0 CACM-2680 7 -30.111525 DIRICHLETQM
2 Q0 CACM-1795 8 4.668672 LUCENE        2 Q0 CACM-1947 8 -30.163033 DIRICHLETQM
2 Q0 CACM-2033 9 4.653692 LUCENE        2 Q0 CACM-3033 9 -30.183292 DIRICHLETQM
2 Q0 CACM-2856 10 4.469285 LUCENE       2 Q0 CACM-1901 10 -30.252209 DIRICHLETQM


perform evalu and model of comput system

6 Q0 CACM-2318 1 9.584506 LUCENE        6 Q0 CACM-2318 1 -33.841827 DIRICHLETQM
6 Q0 CACM-3048 2 7.973845 LUCENE        6 Q0 CACM-3048 2 -34.061756 DIRICHLETQM
6 Q0 CACM-3070 3 7.136974 LUCENE        6 Q0 CACM-2542 3 -34.678558 DIRICHLETQM
6 Q0 CACM-2319 4 6.869540 LUCENE        6 Q0 CACM-1653 4 -34.801720 DIRICHLETQM
6 Q0 CACM-3089 5 6.819804 LUCENE        6 Q0 CACM-3070 5 -34.958477 DIRICHLETQM
6 Q0 CACM-2741 6 6.817646 LUCENE        6 Q0 CACM-2319 6 -34.996212 DIRICHLETQM
6 Q0 CACM-2984 7 6.685209 LUCENE        6 Q0 CACM-2862 7 -35.175892 DIRICHLETQM
6 Q0 CACM-3119 8 6.670673 LUCENE        6 Q0 CACM-2882 8 -35.188107 DIRICHLETQM
6 Q0 CACM-2452 9 6.457639 LUCENE        6 Q0 CACM-1827 9 -35.238354 DIRICHLETQM
6 Q0 CACM-2542 10 6.433337 LUCENE       6 Q0 CACM-1719 10 -35.277958 DIRICHLETQM
```

Spell Checker

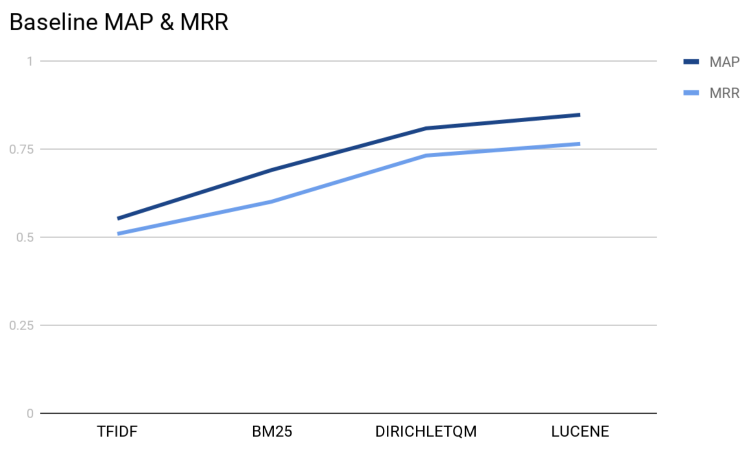
For extra credit, we implemented spell checking using Lucene's built-in SpellChecker class, which uses Levenshtein distance to find the six nearest words for every query term not present in the index. We did not use any external dictionary; rather we used our index as a dictionary to generate suggestions, since our corpus has many domain-specific terms like, TSS and EL1, which might not be available in a general-purpose dictionary.

Results

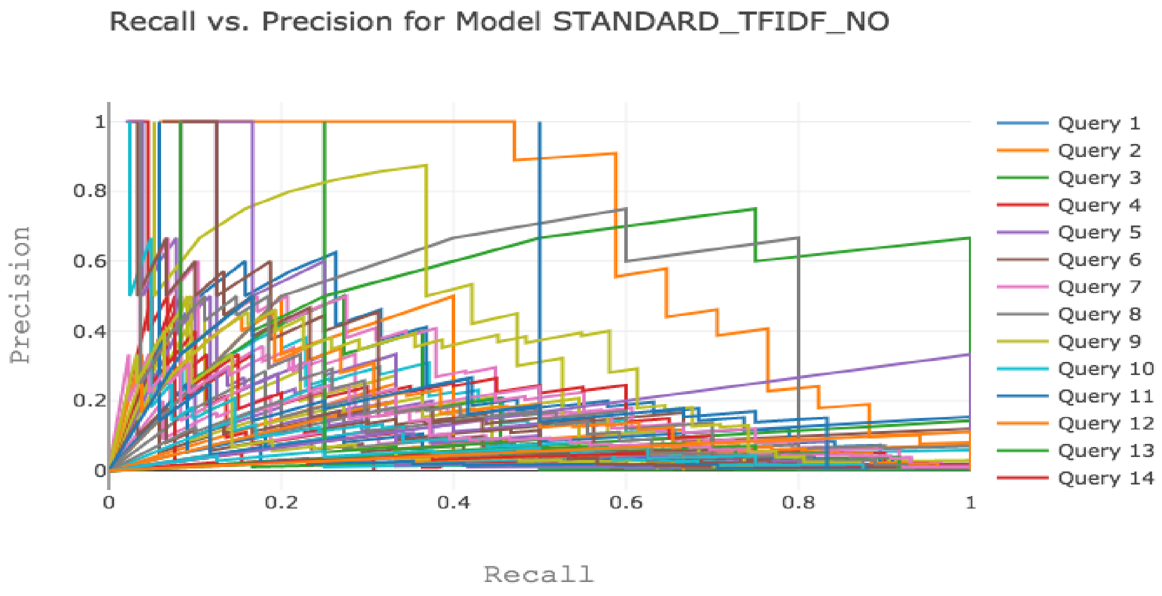
For evaluation, we implemented MAP, MRR, precision at k=5 and k=20, and recall-versus-precision comparisons from scratch in Python. Relevance judgements and model rankings were inhaled in standard TREC format. The resulting raw output files include in their names the model used, query expansion used, and whether the index was stopped or not. Such files list the MAP score, MRR score, precision at k=5 and k=20 for each query, and the recall-versus-precision tables for each query. Much of this data is coalesced in the charts below for easier reference and analysis.

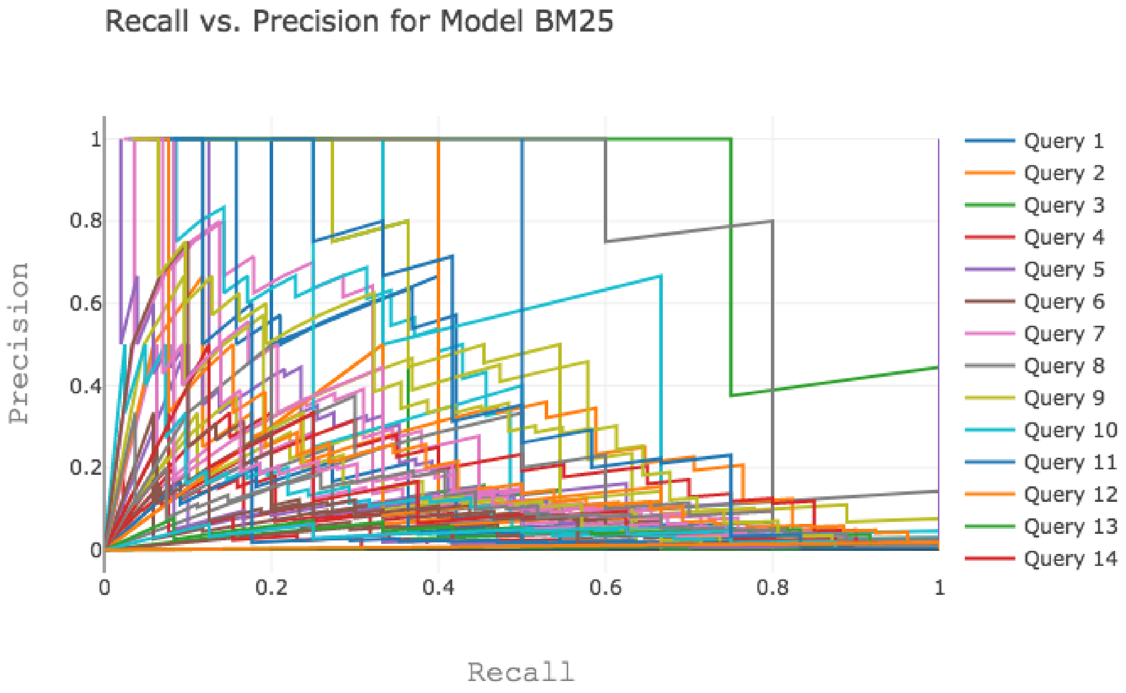
MAP and MRR scores are shown for each of our baseline models in the 'Baseline MAP & MRR' graph. In choosing which models to implement, we wanted to see a range of effectiveness. We figured that TFIDF would be the least effective, as it is the simplest model. We thought that BM25 would be more effective than TFIDF as TFIDF can be skewed by long documents. We anticipated that a query likelihood model with Dirichlet smoothing would be more effective than either of these, as augmenting statistical models probabilistically tends to improve them.[5] And we were unsure of how Lucene's default model would perform, but given its high adoption in industry, we expected it to outperform all other models.

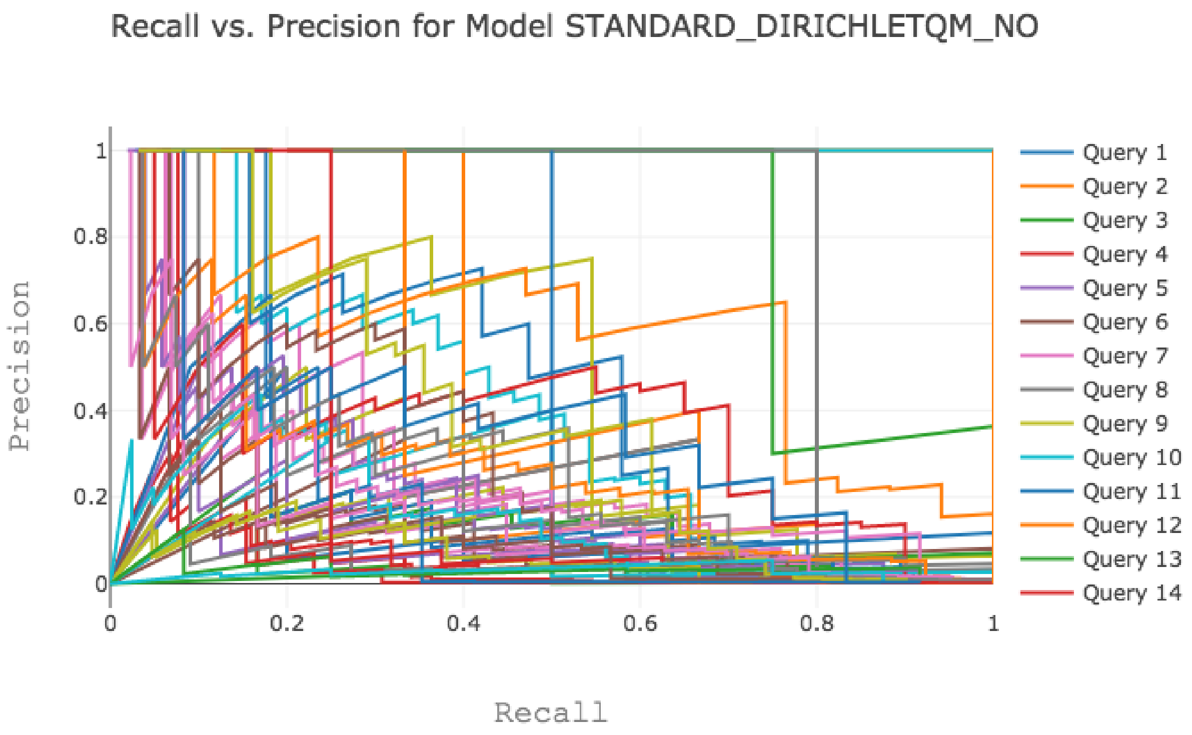
These expectations were corroborated through evaluation, with each model increasing effectiveness in order, as evidenced in MAP and MRR scoring: TFIDF with the lowest effectiveness, BM25 improving upon TFIDF, Dirichlet improving upon BM25, and Lucene improving upon Dirichlet.

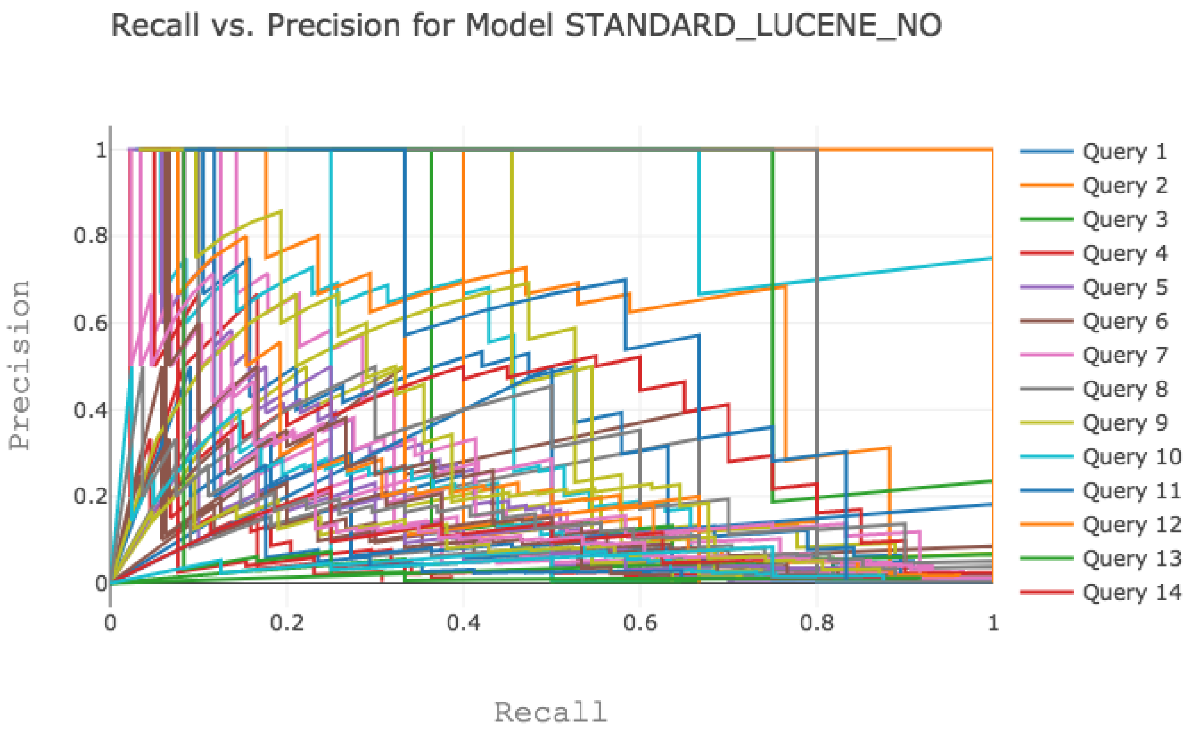


Recall-versus-precision graphs that map recall and precision for each query/run reveal these trends as well, though somewhat more obscurely. In general, the TFIDF recall-versus-precision graph is denser in the lower left-hand corner, demonstrating less precision and overall effectiveness, than the other models, which tend to be less and less focused in the lower left-hand corner with increased effectiveness: BM25 to Dirichlet to Lucene.

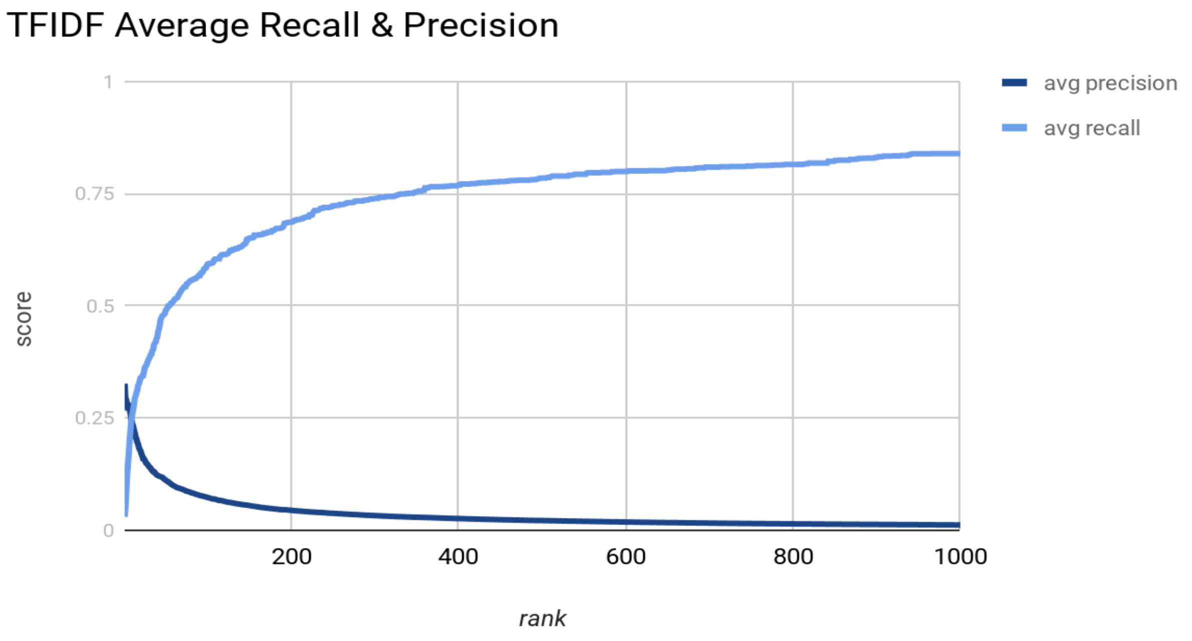


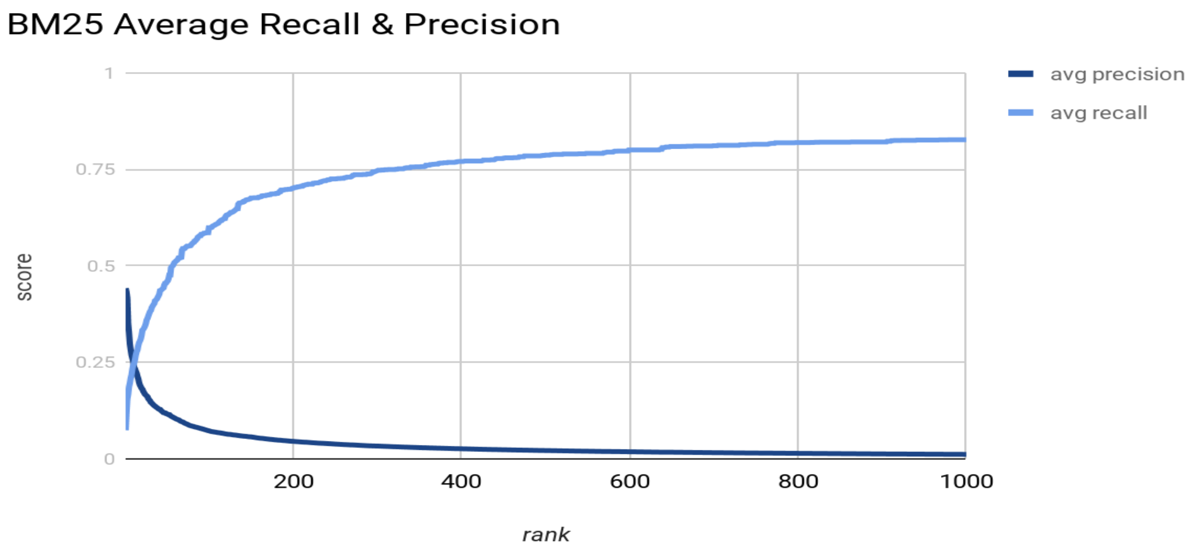


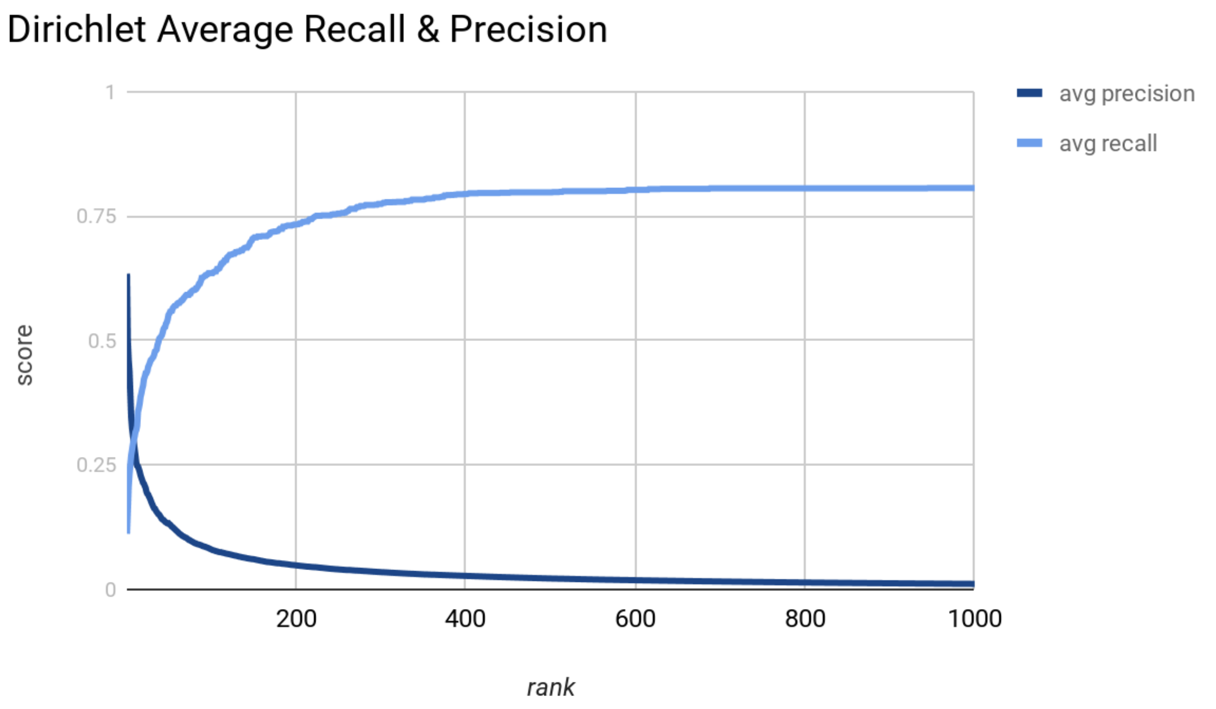


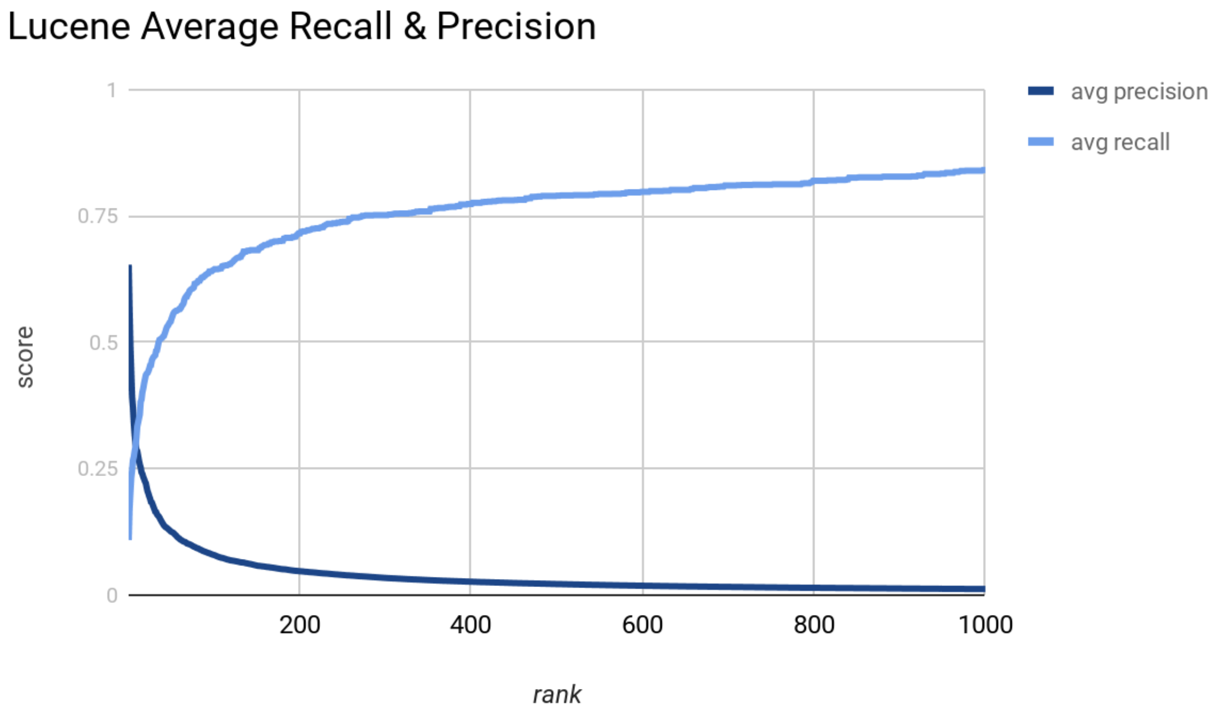


In the following graphs, recall and precision are averaged across all queries for baseline model runs and plotted again rank, allowing for a clearer presentation of information. Precision at early ranks (often the most important to the searcher) can be seen to improve from TFIDF to BM25 to Dirichlet to Lucene. The early slope of the recall line also steepens in this order as well, reiterating this order of increasing effectiveness.

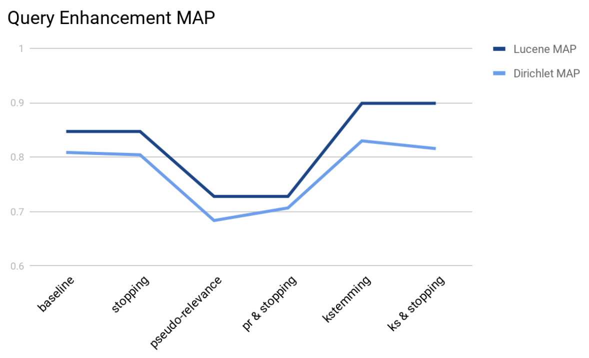


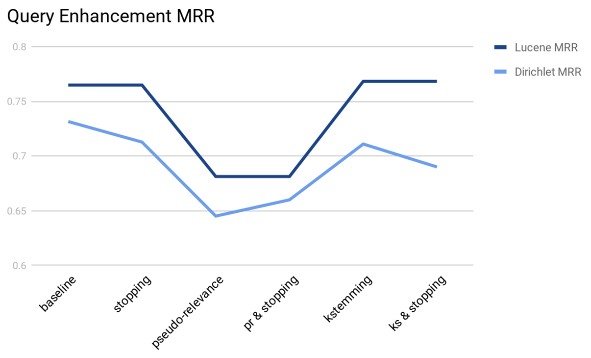






Drilling down, we examined the effects of Kstemming, pseudo-relevance feedback, and stopping on the two best baseline models. Results of these evaluations can be seen in the ‘Query Enhancement MAP’ and ‘Query Enhancement MRR’ graphs below.





With respect to MAP, stopping had a negligible effect, while pseudo-relevance feedback dramatically reduced effectiveness for both Dirichlet and Lucene. Combining stopping with pseudo-relevance helped mitigate the losses for Dirichlet attributed to pseudo-relevance feedback but left Lucene unaffected. Kstemming dramatically improved both Dirichlet and Lucene. Combining stopping Kstemming lessened the gains from Kstemming for Dirichlet but left Lucene unaffected. Precision at k=5 and k=20 results for Dirichlet and Lucene show that both had low precision at early ranks; as such, analysing mostly non-relevant documents (the top 10 in our case) to extract query expansion terms can thus be seen to harm effectiveness, pulling common words from documents irrelevant to the searcher's underlying information need. Such trends are mimicked in the MMR graph as well, reiterating the need for high precision in high-ranked results to implement pseudo-relevance feedback effectively.

Conclusion & Outlook

From the results above, we can conclude that Lucene's default model does, in fact, perform marginally better than our query likelihood model with Dirichlet smoothing and Kstemming. However, we know that Lucene improves upon the basic BM25 ranking model with query boosting. We therefore believe that if we could integrate Lucene's query boosting with our query likelihood model with Dirichlet smoothing and KStem expansion, then that would outperform Lucene's default.

In our implementation of query expansion, we just added the stemmed version of the query words. However, we might improve the performance of the query expansion by adding the whole stem class of the query word rather than just the stemmed word. Secondly, we could improve our spell correction by using a more context-aware algorithm like the noisy channel model of spelling correction. Finally, as throughout this report, we focused on the effectiveness of the ranking models and hardly paid any attention to their efficiency. So we believe it would be really interesting to see how these models scale to large applications like a web search engine.

Bibliography & References

1. Bennett, Graham; Scholer, Falk; Uitdenbogerd, Alexandra, *A Comparative Study of Probabilistic and Language Models for Information Retrieval*, 2008. Section 2.3

2. http://lexicalresearch.com/kstem-doc.txt
3. Croft, W. Bruce; Metzler, Donald; Strohman, Trevor, *Search Engines: Information Retrieval in Practice*, 2010. Pages 216-218
4. Overwijk, *Arnold, Generating Snippets for Undirected Information Search*, 2008.
5. https://nlp.stanford.edu/IR-book/html/htmledition/language-modeling-versus-other-approaches-in-ir-1.html