TIM 260 2015

Information Retrieval Homework 1 Sanjana Maiya

Introduction

As part of Homework 1

1. I used Lucene to index the medline corpus and implement standard search ranking algorithms: Boolean search, TF ranking, TF-IDF ranking.
2. I built a custom search ranking algorithm, which I call Medsearch in the rest of this report as well as the code.

Documented below are the high level details of the design and implementation as well as experimental results running the algorithms for the standard set of queries.

Software

Design decisions, and high-level software architecture

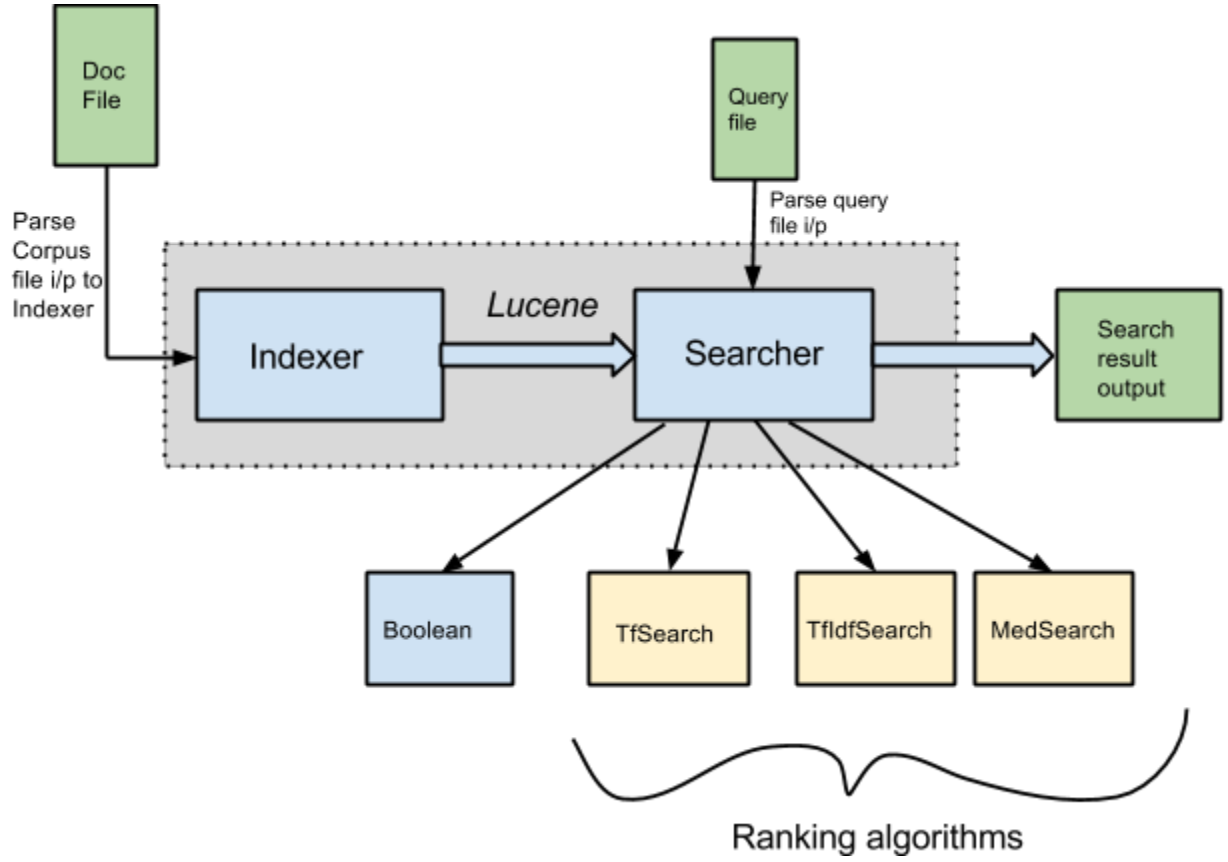
Lucene VS custom index

There are a few advantages of building a custom index, such as more flexibility and less time spent understanding the framework. As such, I tried building a inverted index for the search engine using Java HashMaps, but due to the size of the corpus and limited compute resources in my machine, the program failed to index the complete corpus and ran out of memory after indexing around 50% of the corpus.

Lucene did not encounter this problem as Lucene's indexes are very compact. This is definitely a strong reason to use the same.

Design using Lucene

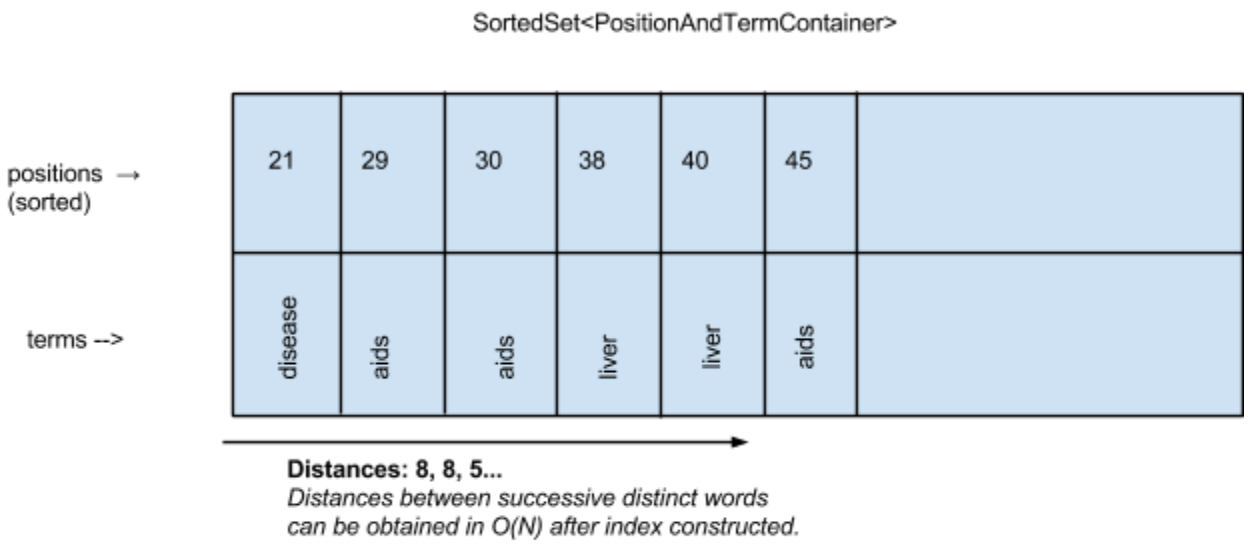
The project consists of two parts at a high level: The indexer, and the searcher. The indexing has been done through Lucene, while the search has been implemented using Boolean search, and ranked using tf, tf*idf and a custom search (MedSearch).



Major data structured, if any

Since I'm using Lucene for indexing, I did not create any data structures for indexing.

For MedSearch, I did use a custom data structure in order to obtain relative locations of query terms in the document in O(N). It is essentially a SortedSet of Position and Terms (see PositionAndContainer.java in code), where the sorting is done using the position. This data structure once built for a query document pair can be used to give relative positions between consecutive distinct terms, which turned out to be a good signal for ranking (see Section about MedSearch.)



Programming tools or libraries used

The following libraries/tools were used:
-- Apache Lucene 5.1.0
-- Luke, for viewing the index created. However, for Lucene 5.1, the recent Luke version was not compatible anymore.
-- Eclipse for IDE and debugging.
-- Junit for testing.
-- Google docs spreadsheet for generating charts.
-- Python for running scripts to get wordcounts on query set.

Strengths and weaknesses of your design, and any problems that your system encountered

Using Lucene for indexing can be a double edged sword, with pros and cons.

Strengths

- Lucene provides many features out of the box.
- Lucene can be extended and customized to implement scoring for search.
- Lucene has a much smaller memory footprint to hold the index as compared to a simple HashMap based index implementation, which can prove to be critical when corpus size is large and machine resources are constrained, like in this homework. I was closely watching the memory footprint of the Lucene based java program, and found it to be well within the limits of my system. For testing and iterating, I was saving

the index into disk, and then loading the index from disk for search. I found no problem with that either.

Weaknesses

- Lucene has a steep learning curve and it takes time to learn.
- Lucene has different sets of APIs to implement the same feature in different Lucene versions, making it very difficult to use usual online resources such as stack overflow.
- While Lucene APIs are documented, there is a shortage of example usage of APIs which take a lot of trial and error to get right.

Medsearch: Customized new algorithm.

Since the nature of the corpus is known, we can customize the search ranking algorithm to potentially do a better job than standard ones. These were the following ideas I tried, some of which improved the ranking while some had the opposite effect.

Each query has two fields: title and description, and I use the description to run the search. The custom algorithm (MedSearch) uses the following methods:

(a) Removal of StopWords :

From the queries list, we obtains the histogram of the words used :

and	28
of	21
management	12
diagnosis	10
therapy	7
for	7
in	7
treatment	6
differential	6
review	5
with	5
to	4

pathophysiology	4
on	4
cancer	4
TREATMENT	4
disease	4
radiation	3
breast	3
advanced	3
article	3
chronic	3
complications	3

The diagram lists the frequency of the top words. Some of these words are not of use in the query and can be removed. As a first step, words like “and”, “in”, “treatment” are removed from the query. While selecting these terms, I have tried not to overfit to the test data set while at the same time keeping in mind that this is a medical domain search.

(b) Tf * idf

The term frequency(tf) is calculated for each term in a document. This term frequency is multiplied by the inverse document frequency (idf), where

$$\text{idf (term)} = \log \left[\frac{\text{Number of Docs}}{\text{docFreq} + 1} \right] + 1$$

(c) Number of matches of query terms

If x number of terms of the query are found within a document, then x is multiplied with the tf*idf
So, the equation become:

$$\text{Score} = \text{numTermsFoundInQuery} * \sum_{\text{t in query}} \text{tf(t)} * \text{idf (t)}$$

(d) Boost based on Proximity of found terms

The final method applied is to boost the score based on the proximity of the distinct query terms found in the document. If distinct terms are found closer together than a threshold, then higher weightage is given to that document. We only consider pairs of consecutive terms that are closer than 6 terms apart, and the boost factor is a multiplicative factor for each of the found pairs. This choice of parameters (1.3 and 6) is heuristic and based on running some experiments. The idea is that when query terms are found close together in the doc, it would likely be a more relevant doc. Phrases such as lung cancer, liver infection are treated as a bag of words in the query, this boost term would undo some of that information loss. This terms improved the number of relevant documents retrieved by around 35 (from 668 to 704).

$$\text{Score} = \text{numTermsFoundInQuery} * \text{proximity boost} * \sum_{\text{t in query}} \text{tf(t)} * \text{idf (t)}$$

$$\text{proximity boost} = 1.3 \text{ ^ (numOf(DistinctTermDistance < 6))}$$

A SortedSet of all found term positions and locations was used to implement this in O(N), the details are in section about special data structures.

Unsuccessful ideas

Not all ideas tried out improved the ranking. A few of the ideas which were tried out and discarded as they did not give desirable results are as follows:

(a) **Weighted term frequency:** Tried changing the weight of more common terms in term frequency, to decrease their term frequency by a factor, but this decreased the number of relevant documents retrieved. For example, the tf of the word “treatment” was re-adjusted to **tf(treatment) *= 0.25.**

(b) Tried to **square the idf in the tf * idf metric** (got this idea from Lucene, as I tried to match their relevance results), but this reduced number of relevant docs retrieved

(c) **Generalization of number of terms found**: Tried to factor in the occurrence of pairs of terms and factor that into the score. The rationale behind this being, for example, if a query has two terms "autoimmune" and "symptoms", then the occurrence of both words multiple times is better than if "symptoms" occurs multiple times and "autoimmune" occurs only once. However, while this did not significantly decrease the performance, this did not increase the number of relevant documents retrieved either. It was a neutral change and was discarded in favor of simple design.

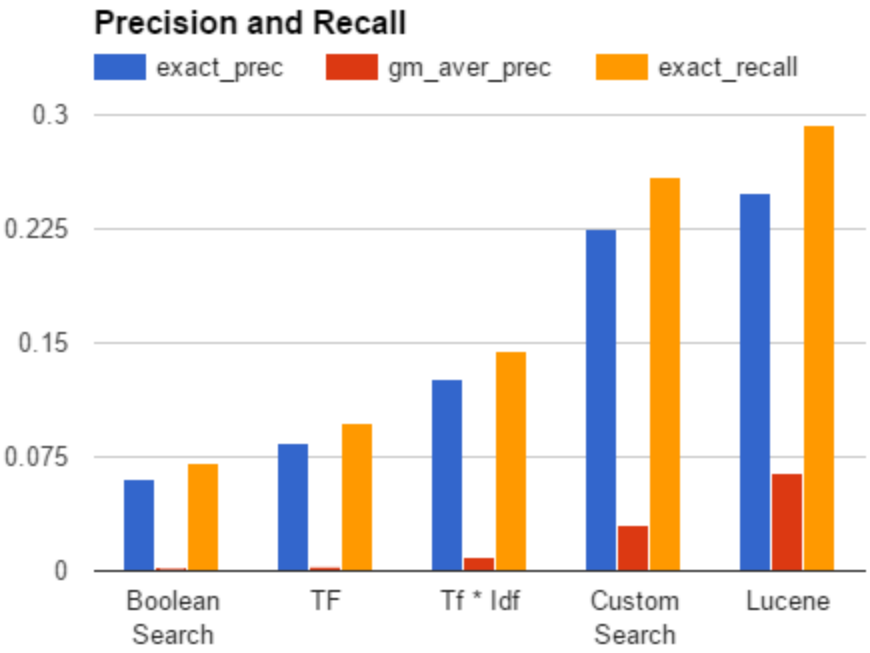
(d) **Plural Words:** To test whether the relevance score was being damaged by the presence of plural words in the query(for example, “diseases” versus “disease”), I tried including both singular and plural words for each word in the query. If this had improved the relevant documents retrieved, then I had planned to pursue this further by making use of a Stemming while indexing and retrieving. However, this approach reduced the number of relevant documents retrieved, so I did not investigate stemming.

(e) **Non-linear numTermsFoundInQuery:** Another approach tried was to tune the factor based on the number of query terms found in the document. While the “numTermsFoundInQuery” above is a linear indicator, I tried to make this asymptotic **(1 - e^(-numTermsFoundInQuery/(QueryLength/2)))**. However, this slightly decreased the number of relevant documents found.

Experimental results

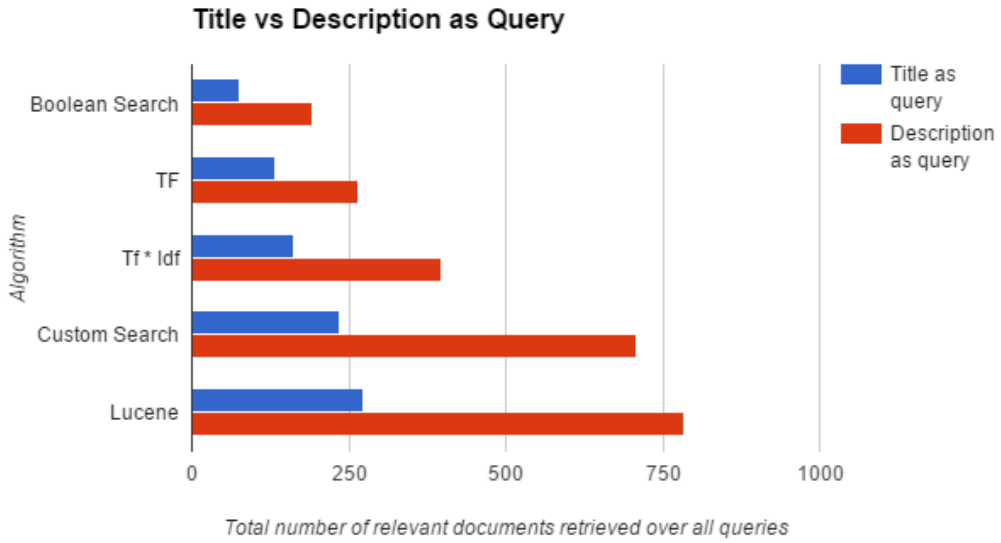
the Precision, AvgPrec, and running time statistics; any patterns you observed across the set of experiments, for example, about which algorithms or representations were effective; what worked well, what did not not;

The following graph shows the exact precision, average precision (geometric mean), and exact recall over all 63 queries run using the query description.

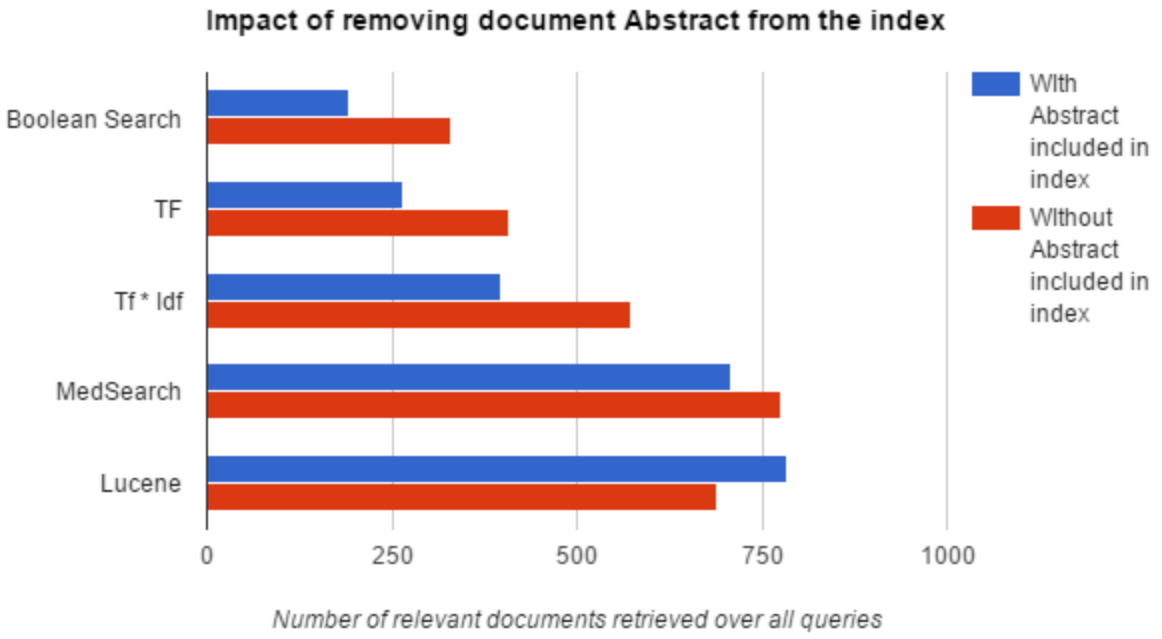


Some general observations regarding the dataset and queries:

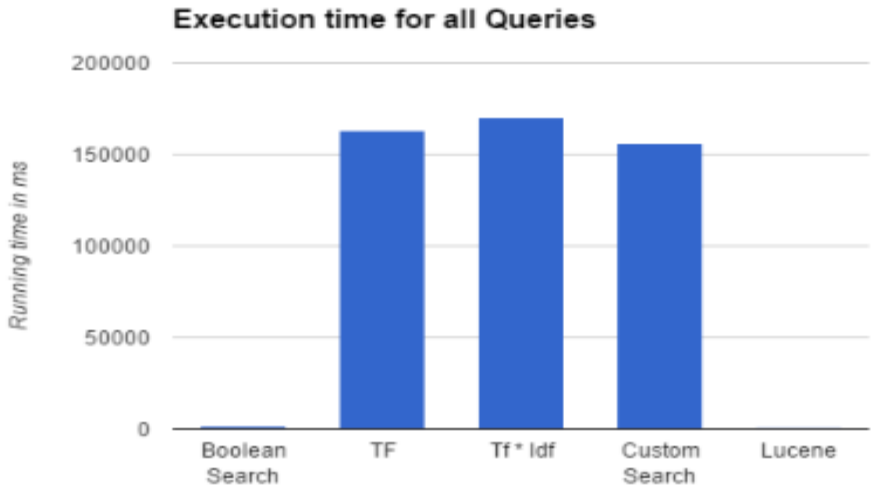
(a) While searching, using the query description instead of the query title drastically improved results for each algorithm used. In the following graph,
Total number of documents retrieved over all queries = 3150 (50 per query, 63 queries in total)
Total number of relevant documents over all queries = 3205



(b) For the program, I have added the terms in the fields “MeSH terms”, “Title” and “Abstract” from the OHSUMED document to the index. However, adding words from only “Title” and “MeSH” to the index increases the number of relevant documents retrieved. This true for all algorithms, but not for Lucene’s default Search.



(c) Customizing the search through Lucene’s CustomScoreQuery significantly increases the run time of the program while retrieving the results of the queries



The time mapped in the above graph includes time for writing the query results to the log file.

Learnings:

What you have learned from this assignment.

- Search results are tricky, changing small parameters in the scoring can change results in a manner not previously predicted. Adding more complexity to the scoring may not improve the search results, keeping it simple is better. However, achieving gains after a certain point is very difficult.
- Indexing data structured require a lot of thought and optimization to scale. Especially with the limited ram on my device, I learnt this the hard way by spending time building a custom index.
- Eyeballing queries and documents is very helpful, especially in a domain specific search, like in this case, a medical search engine
- Using Lucene is not easy. With the number of APIs that change with every release, it is not trivial to implement advanced features. However using Lucene and trying to figure out how it works has given me a better idea of how scoring is done.