CS7 IS3 | Assignment 1 Report
Lucene Search Engine

Thomas Kelly - tkelly2@tcd.ie
16323455

Introduction

While a challenging assignment, after climbing the learning curve with the project technologies, I managed to get the functionality for indexing, searching and evaluation working correctly, having learned a lot about lucene, java, maven and search related concepts. I used the tutorial here to get set up with maven and Java.

Github Repo

Upon beginning work on the assignment, I created a GitHub repository, which stores the java-maven project under '/my-app/', the cranfield collection at /cran, and the trec_eval tool. Please refer to the ReadMe file there for more information on how to run the different aspects of the project.

Access to EC2 Instance & Running Functions

The EC2 instance can be accessed through ssh. Please find the attached '*.pem'* file and run the following commands;

```
$ chmod 400 LUCENE_SERVER.pem
$ ssh -i "LUCENE_SERVER.pem" \
ec2-user@ec2-34-247-165-13.eu-west-1.compute.amazonaws.com
```

Upon entering the *my-app* directory, use '*mvn package'* to compile it.

More detailed instructions on how to run the indexing, searching and evaluation commands are outlined in the project's GitHub ReadMe.md file.

In short, the commands to run indexing, search and evaluation are listed below;

→ *cd my-app/*
Indexing → *./index*
Searching → *./search -s bm25 -q ../cran/reformatted_cran.qry*

- For Vector Space scoring use *'-s vsm'*
- If the -q flag is omitted, querying resorts to user input mode.

Evaluation
→ *cd trec_eval-9.0.7*
→ *.//trec_eval ../cran/reformatted_cranqrel ../cran/results_for_trec_eval*
Note: Ensure that for searching, the *reformatted_cran.qry* file is used, and in the case of evaluation, *reformatted_cranqrel* is used.

Implementation

Indexing

After setting up an analyzer, index writer and configuring directories for where the index will be stored (/index) - and where the documents will be read from (/cran/cran.all.1400) - a buffered read is used to

read the input cranfield file. The code loops through this file, parsing each based on the following fields; Title (.T), author (.A), b_field (.B) and content (.W).
The index writer then adds each document to the index at /index.

Searching

The SearchTest.java file firstly handles the input arguments, setting the scoring and input query file as required.
Looping through the query file, the query term is parsed out for each query.

The CustomAnalyzer file extends the Lucene Analyzer function to incorporate stop words and stemming. This is then used to perform a multi field query based on the search term (either read from the query file or input by the user). The results are written to the *'/cran/results_for_trec_eval'* in the correct format specified here.

Scoring

The scoring strategy turned out to be quite trivial to implement;

```
if (scoring == 0) {
     searcher.setSimilarity(new ClassicSimilarity());
   }
 if (scoring == 1) {
   searcher.setSimilarity(new BM25Similarity());
 }
```

Noting that in Lucene ClassicSimilarity is a subclass of TFIDFSimilarity which implements the Vector Space model for scoring. (ref)

Results

After running trec_eval for each search case, the results could be analysed and both scoring approaches were compared. I focused on a set few metrics to discuss my results.

MAP - Mean Average Precision

→ 'Precision measured after each relevant doc is retrieved, then averaged for the topic, and then averaged over topics (if more than one)' - *trec eval m_map.c file*

MAP for BM25 ⇒ 0.3126
MAP for VSM ⇒ 0.2708

While I initially thought this looked quite low, trying to find a point of reference I compared these results to the MAP value found in trec_eval's test output files (MAP = 0.1785) - this engine seems to perform better in at least that test case. This however appears to be a pretty average result in both scoring cases.

Success

Success being defined as a relevant document having been retrieved at different cutoffs, I thought this might be a good metric to quantify how successful the engine is.
BM25

- Success at cutoff → 5 = 0.7911
- Success at cutoff → 10 = 0.72

VSM

- Success at cutoff → 5 = 0.8622
- Success at cutoff → 10 = 0.8267

The success rates here in all cases are pretty good.

Precision

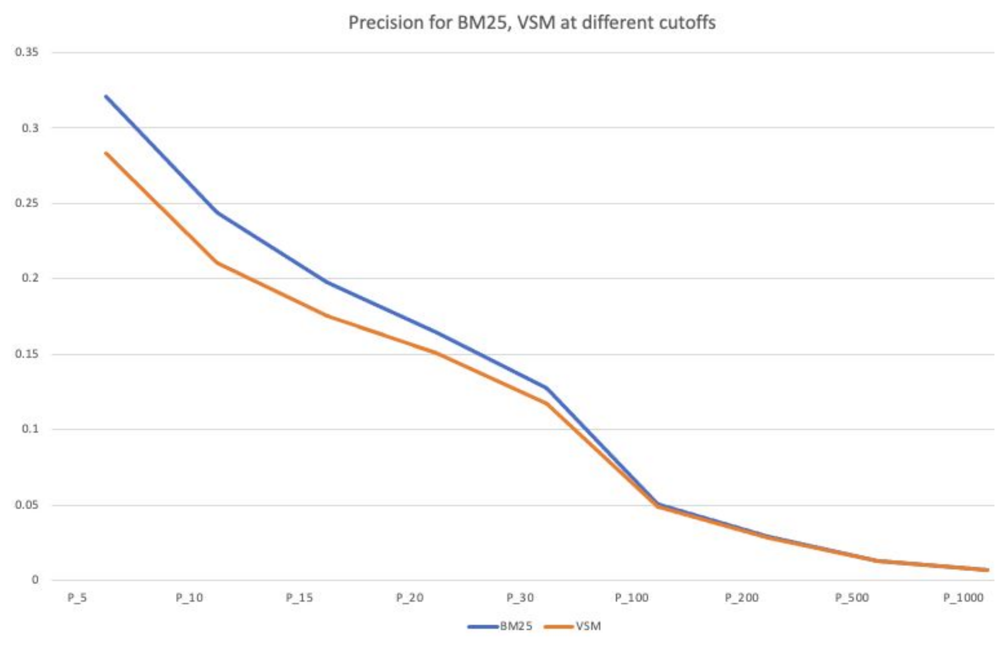

Figure 1; Precision for BM25, VSM at different cutoffs

Precision behaves as expected - reasonably for the highly ranked results, but drops as we include lower ranking results.

Recall

Recall was firstly plotted for documents 5 → 1000, comparing BM25 and VSM scored searches.

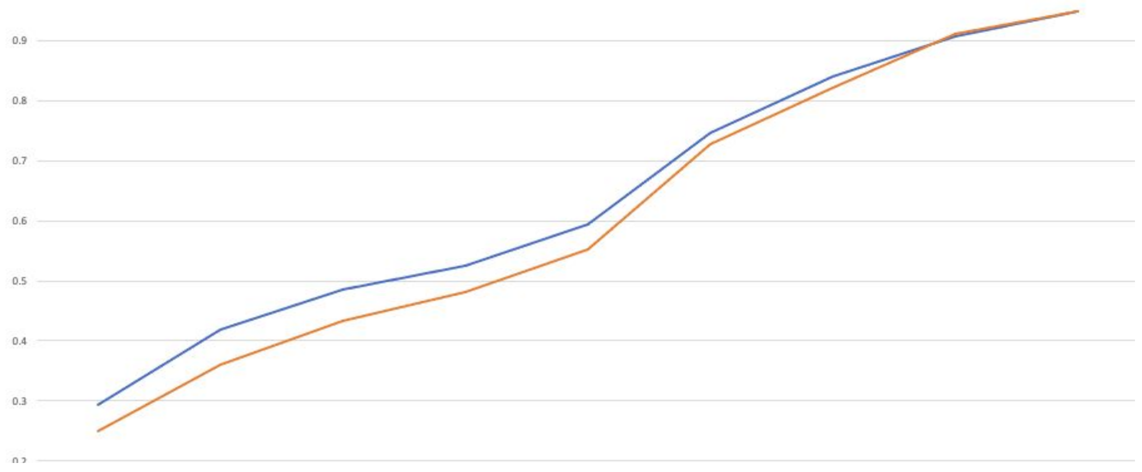

Figure 2; Recall @ cutoffs 5 → 1000; (BM25 = blue, VSM = orange)

Recall is the fraction of relevant results returned and as we can see it expectedly increases as we increase the cutoff size towards 1000 results.
A full list of results are included in the project trec_eval_results.txt file.