CSE272 HW1 Report

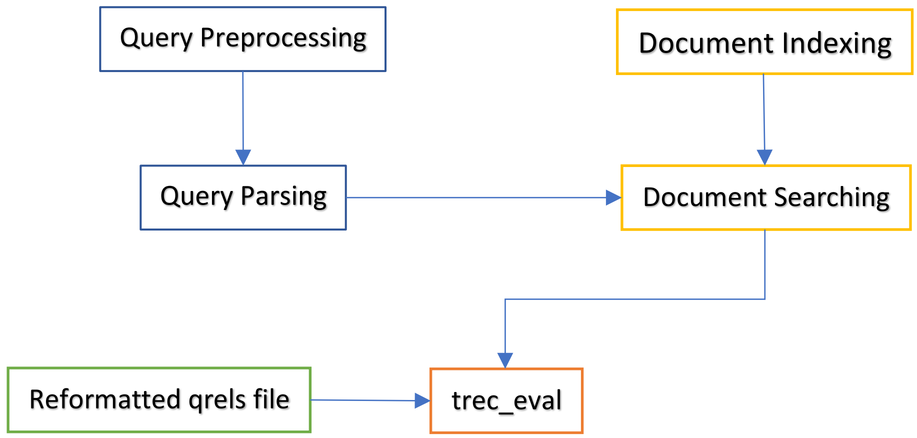
Anthony Liu

1. Software

a. Design decision and high-level software architecture

The design of this system is intended to be as intuitive as possible. At the same time, it must be easily configurable if the user wishes to customize search conditionals. As intuitiveness is the top priority, using Python to implement this is ideal because of the interpretability of this programming language.

At a lower layer, our software primarily inherits the Apache Lucene package from Java platform. I have attempted to write everything from scratch without using Lucene (as in term_frequency.py, custom_ranking.py), but the efficiency was a major tradeoff. Therefore, coming back to using Lucene, the query processing pipeline works as followed:



b. Major data structures

i. HashMap (Python dictionary) which saves the document properties accordingly.

ii. Array of HashMap, which ensures the continuity of inquiries across document cache.

iii. NumPy n-d array, which was used to implement Lucene replacement.

c. Programming tools or libraries used.

i. Python
ii. Docker
iii. PyCharm IDE
iv. PyLucene by Apache
v. NumPy
vi. NLTK.stopwords

d. Strength and weaknesses of this design, and problems this system has encountered.

i. This design allows saving in the RAM directory, which speeds up the search process by more than 3 times. Likewise, this system also allows saving in the file system in case the RAM space is insufficient.

ii. Allowance of custom search fields is a strength of this system. With query_builder.py, users can easily specify which fields to search in, which conditions (must vs. should) to satisfy, and what content to search for.

iii. Displaying the basic statistics of hits is also a strength of this system. With the last part in main.py, users can verify if the term they are searching has a proper amount of hits return. Hence, they can change the search conditionals accordingly.

iv. There are many major issues of PyLucene package. The first is that the methods are poorly implemented, but the documentations are also lacking. This cause many issues when using the standard indexing and searching methods, which were supposed to take minimal time. For example, a constructor of `BooleanClause()` in Java Lucene is wrapped as `BooleanClause.Builder()` in Python PyLucene.

Secondly, the issue comes to setting up an environment with PyLucene. To put this package in use quickly, it must be set as a Python interpreter in a Docker container. However, as solely a Python interpreter, this does not allow manual installation of other Python packages like `NumPy`, `Matplotlib`, or `scikit-learn`. This further increases the burden when performing math operations on term statistics.

v. The design of this system is less optimized for user experience. Changing the Similarity algorithm requires the user to manually input in the command line. Likewise, switching to a new document dataset requires code modification. Still, this would be addressed if it were to be a commercial product.

2. Customized new algorithm

Designing a new algorithm takes place at the same time as experimenting with existing algorithms. Specifically, we chose to modify the current TF-IDF (cosine similarity approximation) algorithm used in Lucene.

Starting with TF, the modified TF calculation is as followed. This further punishes the redundancy of each term.

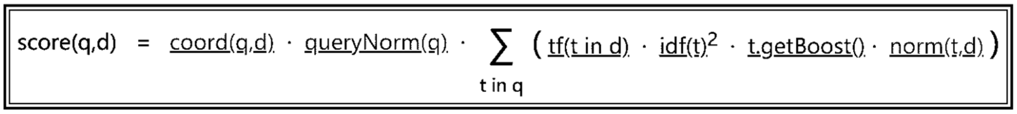
$$tf(t,d) = \sqrt[3]{freq}$$

Then, for the IDF, the equation evolved from both a boosted native IDF and the IDF from BM25 algorithm.

$$\mathrm{idf}(t) = \left[1 + log\left(\frac{\mathrm{numDocs} - \mathrm{docFreq} + 1}{\mathrm{docFreq} + 1}\right)\right]^2$$

Lastly, we fixed the LengthNorm to 1.0 so that longer documents are not penalized accordingly.

Together, these better calculated values are put into Lucene's practical scoring formula, which is shown below.



$$\mathrm{score(q,d)} = \underline{\mathrm{coord(q,d)}} \cdot \underline{\mathrm{queryNorm(q)}} \cdot \sum_{\mathrm{t\ in\ q}} \left(\underline{\mathrm{tf(t\ in\ d)}} \cdot \underline{\mathrm{idf(t)}}^2 \cdot \underline{\mathrm{t.getBoost()}} \cdot \underline{\mathrm{norm(t,d)}} \right)$$

Lucene Practical Scoring Function

3. Experimental results

With the rest remained unchanged, “T”, “W”, and “M” fields parsed, and query formatted with “<desc>”, there are some major improvements on the performance scores.

a. Precision

   The MAP value increased from the 0.08 of Boolean Similarity, and the 0.1257 of Lucene’s native Similarity, to 0.1478, which is almost on par with 0.1524 of BM25 Similarity.

b. AvgPrec

   In trec_eval this is the gm_map (geometric mean, mean average precision). Our model excels over Boolean Similarity (0.0063), and it achieves (0.0295) nearly equivalent gm_map as Lucene’s native Similarity (0.031)

c. Running time statistics

   The time spent on indexing the documents using our Similarity is nearly identical to the time using others (~57 secs for 293856 documents). The searching time is at an acceptable level of (~30 secs for 63 queries to 293k docs).

d. Patterns observed across a set of experiments.

   i. Stop words in queries are less relevant so it is better to remove for efficiency.

   ii. Punctuation removal does not guarantee an improvement in the precision scores.

   iii. TF is less important than IDF in these particular medical documents.

   iv. Boosting the more important terms in Similarity is a nice-to-have.

4. Things learned from this assignment

a. Adding the parsing fields or broaden the search parser conditionals does not ensure an improved match. In contrast, the precision could decrease due to added False Positive. For example, adding query title to the search could decrease the precision even with its optional conditional.

b. Removing stop words from the queries helps only a little. Removing punctuations from the documents slightly worsen the performance.

c. Lucene may be helpful in Java platform, but as someone unfamiliar with Java, PyLucene is the ideal choice. However, it was a regret to choose PyLucene as it costs strenuous effort to *properly* implement the methods.

   Nevertheless, as I attempted to bypass Lucene and implement the whole pipeline by myself, I found that my program will run out of memory or spend too long to parse the document. These drawbacks have pushed me to sacrifice the flexibility of my own program for the efficiency Lucene provides.

d. The ranking algorithm has evolved much over time, specifically when comparing TF-IDF similarity with the Okapi BM25 similarity. The MAP performance increased from 0.096 to 0.126, which is surprisingly promising.

e. Is a 12.6% MAP accurate or anything was wrong in the system, I still do not know the answer, unless I refer to the results from other students. Was my output formatting wrong? I hope someone could resolve my confusion so that I can learn and get prepared for the future.