Information Retrieval Assignment 2:

Document Search and Retrieval with Lucene

Philippe Voet s0183222

GitHub: https://github.com/Triple1312/IR_assignment2

Background on Lucene:

Lucene is an open-source library by Apache software primarily focussed on Search and retrieval on documents. It's a very powerful engine as I've seen that is both fast and very very efficient for space management. It has a unique indexing system that is very complicated (trust me, I tried to learn how exactly it works and failed) but very efficient. This is what makes Lucene so unique. Whilst it indexes text so efficiently, it tries to minimize the overhead of indexing by writing the index to files. It is also possible to make a virtual index on RAM. You'd expect that such an efficient index would take up a lot of space on disk, but it was far less than I expected. Indexing 3.14 GB of files was indexed in about 3 GB of indexed files. Lucene is written in Java but also has a python version. For everything in this document I will be talking about and using the Original Java implementation.

Functionalities:

Indexing

Lucene consists of many different classes with lots of functionality. By far The most important classes are *IndexWriter* and *Analyser*. The classes are used in the **indexing** steps. You start by creating an *IndexWriter* instance. This class will write the index to multiple files. The index gets created by adding a Document to the *IndexWriter* using *IndexWriter.addDocument(Document document).* During this indexing tokens are extracted to create an inverted index of the document. This makes it so that fast lookups of terms are possible. For the tokenization process another class is used. The *Analyser* class is responsible for tokenization of the documents. The *Analyser* can use *Filters* to process the text. There are also many subclasses of *Analyser* to use. The most notable for our usage was the *EnglishAnalyser* class. This class already has some filters built in, like the *PorterStemmerFilter* and the *StopWordFilter* .

The Lucene indexer is very complicated and hard to understand. I will explain everything I learned of how it works and what it can do. The index is not just an inverted index, but also saves positional information for each term. The index creation works like SPIMI, every document that gets added, another segment is made and stored on the disk. Over time these segments get merged. When segments are merged, they become larger segments for efficiency reasons.

The index consists of many file types each with their own function that are binary encoded for space efficiency. *.cfs* files contain the index values. *.fdt* files store the actual data of the documents. To have these saved in the file, you create a Textfield tag with parameter *Field.Store.Yes* as happens in my code later. *.tip* files save the dictionary. *.frq* files save how often terms occur. *.pos* files save the positions of terms in documents. All these files make for an efficient search engine.

Search

When all documents are indexed the next thing we probably want to do is search for terms. There are various types of queries that can be handled by Lucene. Lucene can handle term queries using *TermQuery* , phrase queries which I used using *PhraseQuery* , Boolean queries, Wildcard queries …. The class used for this is *IndexSearcher* . The class needs the index as a parameter. The IndexSearcher uses a similarity measure that can be set with *IndexSearcher.setSimilarity()* with a default-value of *BM25Similarity(k1: 1.2F, 0.75F)* . I both used BM25Similarity with lots of variables and the VSM similarity that has the name *ClassicSimilarity().*

When a similarity is selected we make the documents ready to query on. We create a QueryParser class that make the documents searchable. This is done using the *Analyser*. We then create a *Query* by parsing the *QueryParser* we prepare a *Query* to search. we call *IndexSearcher.search(Query query, int n)* which gives us a *TopDocs* class that contains the top n document indices for our query sorted. Since TopDocs has id's that don't correspond to the order we indexed the files, we still have to ask what documents the indices correspond to. This is done by calling *IndexSearcher.storedFields().*

Implementation of my system:

Last time I wrote my code in dartlang because I hate python. I thought about doing it again, but since I need Lucene in a language other than dartlang, My only option was using ffi which would basically mean I'd write everything in Java anyway, so I just went for pure Java. With the code on GitHub the gradle files are not included, so if you want to run it, you'll have to link Lucene. This did mean I had to write some code again in Java. For this project I used Lucene 10.0.0 .

The classes I had to write again are DataFrame, since it doesn't exist in Java and the Map and Mar evaluator. The DataFrame is defined in the *MiniDataFrame* class. It can basically only read csv files and extract the data. It has no interesting methods. The Map and Mar are defined in *SystemEvaluator* class. It is basically my previous class but translated and has Map and Mar as static methods.

RetrievalSystem

My Retrieval system is very easy, if I may say so myself. Everything is encapsulated in one class called *RetrievalSystem.* It is an abstract class with 2 subclasses. One is called *IndexRetrievalSystem* which creates the normal Lucene index on the drive. This class doesn't open the index on creation, it waits until you fit it. The other one is the *RAMRetrievalSystem* which uses *ByteBufferDirectory* (formerly *RAMDirectory*) to put its index. Even though these subclasses exist and have to be instantiated, all functionality of *RetrievalSystem* is in the Superclass.

As analyser I just use the *EnglishAnalyser* because it already stems using the *PortereStemmerFilter* and removes stopwords with the *StopWordsFilter*. Something to note is that the amount of stopwords it filters is way less than the amount of stopwords I filtered in my previous project. I just used the stopword list used by Lucene which I think helped me in my implementation, more on that later.

Fitting

To start using the class we obviously have to add some documents. This can be done by calling *RetrievalSystem.fit_corpus(String folderpath)*. The method takes in a folderpath , loads all txt files and creates a IndexWriter that writes them to the index after being tokenized by the *EnglishAnalyser*. It creates a *Document* of every file with 2 tags, one with the contents of the file and one "id" with the filename which will be very important later on because the index doesn't preserve the order at which the files are added. Another way to fit is to call *RetrievalSystem.addDocument(Document doc)* to add individual documents to the index.

Search

When the system is fitted we can start to search on it. There are 3 types of query options. There is the default *query(String input, int k)* method that uses the *BM25Similarity* with k1=1.2F and b=0.75F, the *query_vsm(String input, int k)* method that uses the *ClassicSimilarity* class to use Tf-Idf and a *query_bm(String input, int k, float k1, float b)* to alter the parameters of the default *BM25Similarity* where *input* is the query string and *k* is the top-k results to be returned. All these methods return a List of tuples of a <Document, Float> . Since Java doesn't have a Tuple class I used *Map.Entry* class which does the same thing.

The methods all work the basically the same. First open the index and instantiate an *IndexSearcher* instance and sets its similarityType. Then we instantiate a *QueryParser* with the tag of the documents we want to query on. We use this *QueryParser* to create a *Query* with the query string. With this done, we can finally use this to search the index with *IndexSearcher.search(Query, int k)*. This gives us the top-k documents to be returned. We use only Phrase queries. Others are not allowed. This is why in the main function I remove all characters that contain to any other type of query.

Evaluation:

VSM

Since in the previous project we had to write our own Tf-Idf I was very interested in how bad my implementation would be. However, to my surprise this wasn't the case. These were the scores for Lucene's VSM:

- Map@1: 1.9256691700365877E-4
- Map@3: 0.30682328775916146
- Map@5: 0.3290583477758595
- Map@10: 0.3013287117273247
- Mar@1: 7.406419884756107E-6
- Mar@3: 0.061178279978662604
- Mar@5: 0.10869315846828119
- Mar@10: 0.19755062706603205

In comparison to my results the previous assignment:

- Map@3: 0.44444444444444486
- Map@10: 0.2976121702291529
- Mar@3: 0.0884864514917354
- Mar@10: 0.19470504639887443

My System scored higher or the same on every test, which really surprised me. My system only took 10 hours to complete which isn't ideal. Everything should be the same. I think the biggest factor is as I described earlier that I removed way more stopwords than Lucene and also, Lucene accepts if a query-word is part of a word in the document whilst my implementation needs an exact match. Although I thought the latter would help Lucene. Now I don't feel so bad about my implementation anymore.

BM25

We didn't have to implement this in the previous assignment and after the "bad" scores of VSM I was very interested in if BM25 would do better, and it did:

- Map@1: 1.9256691700365877E-4
- Map@3: 0.5685859169394567
- Map@5: 0.6335451569420532
- Map@10: 0.5968226458694431
- Mar@1: 7.406419884756107E-6
- Mar@3: 0.11548215033529591
- Mar@5: 0.21286484430741137
- Mar@10: 0.395485397101632

This is significantly better than the VSM scores although an average of 0.6 is not that good either. This is also better than my Tf-Idf. This was BM25 using its default parameters of k1=1.2F and b=0.75F, maybe these are just not great parameters for this dataset.

BM25 with parameters

$$\text{score}(D,Q) = \sum_{i=1}^{n} \frac{\text{IDF}(q_i) \cdot f(q_i,D) \cdot (k_1 + 1)}{f(q_i,D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgD}}\right)}$$

This is the formula for BM25. As you can see there are 2 undefined variables in the formula: k1 and b. We call k1 the termFrequency parameter. It controls how much that reoccurring terms increase the document score with. Its value is mostly set between 1.2F and 2.0F. We call b the normalization parameter. It controls the impact of the document length. A value between 0 and 1 is used with its default being 0.75F. I played with these parameters by firstly playing with their extremes and then looking what it does.

k1=2 , b=0.75

I started By setting k1 to 2.0 and keeping b on 0.75. These are the results:

- Map@1: 1.9256691700365877E-4
- Map@3: 0.5803324988766765
- Map@5: 0.6516849605238012
- Map@10: 0.6141536683997748
- Mar@1: 7.406419884756107E-6
- Mar@3: 0.11793322825244203
- Mar@5: 0.2190961251619986
- Mar@10: 0.40706366470487443

These results are slightly better in every way than with its default parameters. Apparently the more the relevant terms are present in the document the more relevant the document is.

k1=1.2 , b=0.5

- Map@1: 1.9256691700365877E-4
- Map@3: 0.5857243725527772
- Map@5: 0.6583477758521311
- Map@10: 0.6274407856730291
- Mar@1: 7.406419884756107E-6
- Mar@3: 0.1191112740483648
- Mar@5: 0.22172697957266035
- Mar@10: 0.41674312481138465

This is again a little bit better than the default BM25. This means that if the terms get found inside the documents , their length seems to not make them less relevant. I wanted to check if normalization was needed for good results and tried with b=0

k1=1.2 , b=0

- Map@1: 1.9256691700365877E-4
- Map@3: 0.265421400603375
- Map@5: 0.29374157519738586
- Map@10: 0.2896398998652019
- Mar@1: 7.406419884756107E-6
- Mar@3: 0.05311432303856303
- Mar@5: 0.0974484002093959
- Mar@10: 0.1908804480972425

These results are far worse than the previous. This means that normalization in fact does play a major role in ranking the documents. The best score is probably somewhere in the middle.

k1=2 , b=0.5

- Map@1: 1.9256691700365877E-4
- Map@3: 0.5937479940945939
- Map@5: 0.6711727325245771
- Map@10: 0.64373194685154
- Mar@1: 7.406419884756107E-6
- Mar@3: 0.12077977564544665
- Mar@5: 0.22622073529425954
- Mar@10: 0.42795317148874024

This again Is a little bit better than all previous results. I know I cant push my b down anymore, but maybe I can push my k1 higher for a better result ?

k1=3 , b=0.5

- Map@1: 1.9256691700365877E-4
- Map@3: 0.5957378522369642
- Map@5: 0.6715963797419856
- Map@10: 0.64261505873292
- Mar@1: 7.406419884756107E-6
- Mar@3: 0.12123333907957023
- Mar@5: 0.2261511055115861
- Mar@10: 0.42707102353392434

Even better than all other. I think it’s safe to say that the more times a term is present.

Whilst my implementation had a totally different score for Map@3 as for Map@10 something that has been consistent is that in these examples Map@3, Map@5 and Map@10 always kind of have the same score.

StandardAnalyser

We have always tried using an *EnglishAnalyser* but does this really matter ? I already thought my code got a better result because I removed more stopwords, so what if we don’t remove any ? We now use the *StandardAnalyser* .

VSM

- Map@1: 1.9256691700365877E-4
- Map@3: 0.2738943449515364
- Map@5: 0.2936260350471867
- Map@10: 0.27357981898709777
- Mar@1: 7.406419884756107E-6
- Mar@3: 0.05433948007767949
- Mar@5: 0.09679153526491231
- Mar@10: 0.17891534412021504

The scores are definitely worse than with the *EnglishAnalyser* but are not that much worse to my surprise.

BM25 k1=3 , b=0.5

The Tf-Idf with the normal analyser wasn't a success. What if I choose the best values for BM25, would it then have a good score without stemming and filtering ?

- Map@1: 1.9256691700365877E-4
- Map@3: 0.6043391745297929
- Map@5: 0.6735605622954242
- Map@10: 0.625726940111698
- Mar@1: 7.406419884756107E-6
- Mar@3: 0.12276431617048314
- Mar@5: 0.22651436372485964
- Mar@10: 0.4146797690890953

I don't know how, but the values are around the same values as with the *EnglishAnalyser* which it shouldn't be. I think its gets such a good result because of the way that Lucene handles phrase querries.

Conclusion

This project was a lot of fun to do and I'm glad I had so much fun with it. I also learned again that I have to read the documentation more clearly especially if you get all your info from older versions that use methods that don't exist anymore in lucene 10. I was extremely interested in its indexing system and I might want to try to make it myself for my other project in dart.

I will make more than just one results.csv file. The results.csv file is the system with BM25 using its default values, results_vsm.csv are the results using lucene's *ClassicSimilarity* and results_best.csv will be the results of BM25 with k1=3 and b=0.5 . All files will use the *EnglishAnalyser*.