IR-Lucene

Senne Rosaer - Toon Meynen

Contents

1 Introduction

2 Study of Lucene

2.1 Analyzer

The first step of the whole process Lucene offers is the use of an analyzer to preprocess the input. An analyzer acts as a factory containing multiple possible ways of changing and filtering the given text to improve the results and performance of the document retrieval system. As output it should give a tokenstream of the input because the tokens are what will be used for eventual indexing.

Lucene gives us the option to use predefined analyzers and methods that can be used in analyzers, as well as the ability to create our own. We created a custom analyzer which will be discussed in section 3.2. A few important methods of processing tokens that are specifically mentioned in the Lucene documentations are: stemming, stop words filtering, synonym extension and text normalization.

For the processing methods the most basic one is the removal of stop words. These don't give much value to the search and are also quite simple to remove. The removal of stop words works by removing any token that occurs in a certain stop word list. This list can be custom specified but there is also an *EnglishParser*[**?**] which has already implemented a complete list of stop words. This specific parser also has an implementation for stemming which is the replacement of words with their stems. This can be used to broaden the search query. Since this is language dependent it's great Lucene already has one implemented for English.

As for synonym extension, which is the addition of synonyms of index/query terms, this is possible and there is support for this in Lucene but unlike the previous methods this relies on self-defined synonym mappings or external databases containing the mappings so this will not be further discussed. The same goes for normalization, changing the form of a token to a more general/normalized notation, is possible but there are no methods given by of Lucene to completely implement this.

2.2 Indexing

[6] A Lucene index is conceptually similar to a database table, it does however differ in some important ways. One of these differences is that a Lucene index does not require clear constraints, it does not have predefined columns or a primary key. This means that you can store different types of documents in one and the same index. Each document stored by Lucene has fields, which are similar to the columns in a database table.

Each index is composed of one or more sub-indexes, also called segments. These

segments are immutable and are created every time the in-memory buffer is flushed. Each index query needs to query multiple segments and merge the results, as well as handle deleted documents. A document does not actually get removed from the segment. The id is simply stored in a separate file that holds deleted IDs.

Instead of indexing page numbers to the words on that page, Lucene uses a reverse index. This means that it will keep a mapping of terms to document IDs of all documents containing these terms. The terms are extracted from the document using the analyzer as explained in the previous section. This index is stored using a skip-list, unlike a lot of other retrieval systems which work with B-Tree's.

2.3 Spell correction

[3] Lucene can be used for spell correction. It has a *SpellChecker* class that will suggest similar words to the input. By default it will use the Levenstein distance to find similar words. It is possible to require a certain accuracy before a word can be considered similar to the input word. Using *SuggestMode* it is also possible to take the popularity of the suggested words into account.

2.4 Query processing

The query processing in Lucene is a process very similar to the analyzing of documents which is logical because we need to find the same terms. It also uses an analyzer to split the input string in multiple tokens, these tokens are combined with a certain *fieldname* so we can search on specific fields. It might also be optimal to use a different analyzer to use query expansion. Being able to add synonyms and stemming could improve the hit-rate. Lucene also provides the option to indicate that some terms need to be found and some not. This way a document could have a good score but not get retrieved since it does not contain an expected term and vice versa.

2.5 Score Models

[7] Lucene score models support several pluggable information retrievel models, including: Vector Space Model, Probabilistic Models such as Okapi BM25 and DFR and Language Models.
The objects being scored in Lucene are the Documents. Each document is a collection of fields, and the field contains semantics about how it is created and stored. The scoring works on fields and combines the results for the document. Because of this it is possible for two documents with the same content, but different fields, to yield a different score.

We will briefly discuss the scoring models provided by Lucene in the following subsections.

2.5.1 TFIDF Similarity

[9] Implements the Vector Space Model for the Scoring API. In VSM documents and queries are represented as weighted vectors in a multi-dimensional space, where each distinct index term is a dimension, and weights are tf-idf values. It is possible to use VSM with other values, but tf-idf gives good results. Lucene refines VSM score in several ways.

- The normalizing vector V(d) to the unit vector can cause problems, therefor Lucene uses a different document length normalization factor. This will normalize to a vector equal or larger than the unit vector.
- When indexing a document users can use a boost value to specify that it is more important than other documents. Lucene also uses field boosts in addition to these document boosts. This field boost is the multiplication of all boosts of the separate additions of that field, because it is possible to add the same field multiple times to a document.
- At search time users can specify boosts to each query, sub-query and query term.
- A document may match a multi term query without containing all the terms of that query.

2.5.2 BM25 Similarity

[10] BM25 is a retrieval function that uses a bag-of-words model applied on the query terms. It will rank the documents based on the query terms appearing in each document, regardless of their proximity within the document.

2.5.3 Boolean Similarity

[11] The Boolean Similarity class will give terms a score equal to their query boost. This is typically used with disabled norms.

2.5.4 Multi Similarity

[12] The Multi Similarity Class implements the CombSUM method described in Joseph A. Shaw, Edward A. Fox. In Text REtrieval Conference (1993), pp. 243-252.

2.5.5 Axiomatic Similarity

[14] This class implements a family of models based on BM25 as described in Hui Fang and Chengxiang Zhai 2005. An Exploration of Axiomatic Approaches to Information Retrieval. In Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '05). ACM, New York, NY, USA, 480-487.

2.5.6 DFI Similarity

[15] This class implements the Divergence from Independence model based on Chi-square statistics. DFI does not require any parameter tuning or training nor does it make any assumptions about word frequency distributions on document collections. With this similarity it is highly recommended not to remove stopwords.

2.5.7 DFR Similarity

[16] This class provides a fromework for the Divergence from Randomness as introduced in Gianni Amati and Cornelis Joost Van Rijsbergen. 2002. Probabilistic models of information retrieval based on measuring the divergence from randomness. ACM Trans. Inf. Syst. 20, 4 (October 2002), 357-389.

2.5.8 IB Similarity

[17] This class provides a framework for the family of information-based models, as described in Stéphane Clinchant and Eric Gaussier. 2010. Information-based models for ad hoc IR. In Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval (SIGIR '10). ACM, New York, NY, USA, 234-241. The retrieval function used in this framework is of the form $RSV(q,d) = \sum -\chi_w^q log(P(X_w \geq t_w^d | \lambda_w))$ where χ is the query boost, X counts the occurences of word w, t is the normalized term frequency and λ is a parameter. This framework closely resembles the DFR framework.

2.5.9 LM Similarity

[18] The LM similarity is an abstract class for language modeling similarities, of these two are implemented:

- Dirichlet Similarity, This implements bayesian smoothing using Dirichlet priors. From Chengxiang Zhai and John Lafferty. 2001. A study of smoothing methods for language models applied to Ad Hoc information retrieval. In Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '01). ACM, New York, NY, USA, 334-342.

- Jelinek Mercer Similarity, This implements a language model based on the Jelinek-Mercer smoothing method. From Chengxiang Zhai and John Lafferty. 2001. A study of smoothing methods for language models applied to Ad Hoc information retrieval. In Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '01). ACM, New York, NY, USA, 334-342.

2.6 Document retrieval

To retrieve a document in Lucene queries are used. Each query will use one of the previously described score models, or a custom model, to score all documents. It will then return the n highest scoring documents, where n is a parameter in the search function. The Lucene implementation will only count top hits accurately up to 1000 documents. This happens because it would take longer to count the hits than to retrieve them for larger amounts of found documents.

3 Implementation document retrieval system

3.1 Preprocessing data

Since we got a pretty large dataset to work with that is just a single XML file we needed some way of filtering this. For this we created a python script that created a single file for every question where the filename is the ID and all the answers are seperate lines in these files. We use the fact that we create

Algorithm 1: Pseudo-code preprocessing algorithm

```
for row ∈ XMLfile do
    if rowType == question then
        file ← createFile(rowID);
        file.write(row.content);
    else if rowType == answer then
        file ← openFile(parentID);
        file.write(row.content);
    else
        pass
```

a separate file with as title the ID and every answer contains this ID so we can find the file based on that and add more data. For most of this project we used a smaller subset of this data since we wanted to be able to have faster computations while figuring out how things worked.

3.2 Document analysis and indexing

The implementation for document analysis and indexing mostly follows the structure of the documentation [19]. We first need an analyzer for which we originally planned on using the *StandardAnalyzer* but after analyzing we stumbled upon a few issues. This analyzer completely filtered any special characters from the input which would be fine in most cases but we did not feel it was appropriate for indexing on code questions. Some examples on why we saw this as an issue.

1. $function() \rightarrow function$: Having function calls without the indication that it is effectively a call is not something we want.

2. $Class.Object \rightarrow Classobject$: It would also remove the instances of notation in some languages.

3. $c++ \rightarrow c$: It would make us enable to index language names that include symbols.

Because of this reason we ended up creating our own analyzer that starts with a tokenizer that splits on whitespaces and *;* since that is also a very frequent line ending. We also used some token filters and the easiest one is a stopword filter. As said earlier this needs a list of words we want to filter out but luckily apache has an english analyzer [20] which gives us the ability to get it's list of stopwords. When testing around with the output of this we got tokens like *opacity.* which has a normal end of sentence symbol at the end of a word. Since we decided to not remove every symbol it's obvious we get tokens like this. This is not really what we want so we created another token filter that removed some symbols that occur in a list if they get matched at the end of a token. We also included a filter for single symbol tokens since those are most likely not very usefull. Finaly we added lowercase filter to ensure we can find all documents. We will not show any pseudo-code or normal code about every filter since these quite literally do what was explained and some are implementations from Lucene itself. As for the rest of the implementation part we will give code but since we do not want to cut any information from this part it won't be pseudo-code.

```
public class OwnAnalyzer extends Analyzer {
    @Override
    protected TokenStreamComponents createComponents(String fieldname){
        final Tokenizer source = new OwnTokenizer();
        TokenStream result = new StopFilter(source, EnglishAnalyzer.getDefaultStopSet());
        result = new RemoveSignAtEndTokenFilter(result);
        result = new RemoveSingleSymbols(result);
        result = new LowerCaseFilter(result);
        return new TokenStreamComponents(source, result);
    }
}
```

Figure 1: Custom created analyzer

For the next part we need to index our documents. This starts with a few simple steps.

```
Analyzer analyzer = new OwnAnalyzer();
Path indexPath = Paths.get( first: "indexDirectory");
Files.createDirectory(indexPath);
Directory directory = FSDirectory.open(indexPath);
IndexWriterConfig config = new IndexWriterConfig(analyzer);
IndexWriter iwriter = new IndexWriter(directory, config);
```

Figure 2: First part of indexing

Here we create a directory that is used to write the index file to, create an *IndexWriterConfig* which is the configuration class that has our analyzer as a parameter and then we create an *indexwriter* with the directory and config. The second part is actually adding documents that have to be indexed. Every document needs to be represented with a *Document* class object from Lucene. Since every file we create by preprocessing contains questions and answers that are linked, this step does not take much more work. A document contains multiple fields and later when we are querying we always query based one or more fields. Every row from the XML file contains a lot of information but we decided to focus on a few simple parts like *Title*, *Body*, *Tags*, *CreationDate*. These are also the different fields we had for every document. Every created document was then added to the *IndexWriter* so it could be indexed which concludes this step.

3.3 Query processing

This is a rather short section since Lucene makes this step fairly easy.

```
QueryParser qparser = new QueryParser( f: "Title", analyzer);
Query query = qparser.parse( query: "conversion c++ ;from decimal");
```

Figure 3: Query processing

First we need a *QueryParser* which, as the name suggests, parses a query. This class needs a *fieldname* as parameter and an analyzer. In this case we have the *Title* but in most cases it's probably better to use *Body*. The analyzer is our custom analyzer, as used in the indexing. In some cases it might be better to use different analyzers for indexing the documents and parsing the query but we felt that both needed the same because of the reasons we mentioned in 3.2. We could've used another analyzer that had some form of query expansion and we tried to use stemming implemented by a Lucene *PorterStemmer*. This sadly provided some unexpected results, for example $conversion \rightarrow convers$, which is not something we want. It might be possible to say that if we also use stemming on the indexing that we will still get the expected results but this opens up the possibility of mapping multiple different words to the same.

3.4 Document search and retrieval

```
DirectoryReader ireader = DirectoryReader.open(directory);
IndexSearcher isearcher = new IndexSearcher(ireader);
isearcher.setSimilarity(new BM25Similarity());
ScoreDoc[] hits = isearcher.search(query,  n: 10).scoreDocs;

Document hitdoc = isearcher.doc(hits[0].doc);
System.out.println(isearcher.explain(query, hits[0].doc));
System.out.println(hitdoc.get("Body")+ "\n");
```

Figure 4: Document search and retrieval

The document search and retrieval starts by defining an *IndexSearcher* instead of an *IndexWriter* which reads from the directory where we wrote the index file. Then there is a possibility to set the similarity which is the score model we want to use. In our case we used BM25 since it is a very simple one, but as described earlier there are a lot of different models implemented in Lucene. We then searched with a certain query and the amount of best documents we want. This will search every document, score them and give us the best options. Since this gives us json objects that contain the document ID and the score we still need to retrieve the document itself. Lucene also provides the option to explain the scoring of a certain query of a document which is a handy feature to test multiple analyzers and find whhich analyzer works best for your needs.

3.5 Example

To test that we had a working document retrieval system we made our own document and added it to the list.


```
{"id": -1, "Body": "I have a question about something something in c++. There is this class.object and I want to do function x() on it.", "Title": "C++ issue class.object"}
{"id": -2, "Body": "I think you should try something something function z()"}
{"id": -3, "Body": "You should never use function x() on class.object, always use function z() in this case"}
{"id": -4, "Body": "This question is a copy of question something something href so please look there and delete this"}
```


Figure 5: Custom document

This was the code used to retrieve it.

```java
public static void example() throws IOException, ParseException {
    Analyzer analyzer = new OwnAnalyzer();

    Path indexPath = Paths.get( first: "indexDirectory");
    Directory directory = FSDirectory.open(indexPath);


    QueryParser qparser = new MultiFieldQueryParser(new String[]{"Title", "Body"}, analyzer);
    Query query = qparser.parse( query: "c++ class.object");


    DirectoryReader ireader = DirectoryReader.open(directory);
    IndexSearcher isearcher = new IndexSearcher(ireader);
    isearcher.setSimilarity(new BM25Similarity());

    ScoreDoc[] hits = isearcher.search(query,  n: 10).scoreDocs;

    Document hitdoc = isearcher.doc(hits[0].doc);
    System.out.println(isearcher.explain(query, hits[0].doc));
    System.out.println(hitdoc.get("Title")+ "\n");
    System.out.println(hitdoc.get("Body")+ "\n");
}
```

Figure 6: Example code

This works very similar to the previously explained parts. The only big difference is the use of a *MultiFieldQueryParser*. Using this instead of the standard parser allows us to search multiple fields at once which gives us the possibility of searching *title* and *body* at the same time. We then queried using *c++ class.object* as the query since giving the language and the part where the problem occurs is quite common. This gives us the expected result.

References

[1] https://Lucene.apache.org/solr/guide/6_6/understanding-analyzers-tokenizers-and-filters.html

[2] http://www.Lucenetutorial.com

[3] https://Lucene.apache.org/core/8_6_3/suggest/org/apache/Lucene/search/spell/SpellChecker.html

[4] https://Lucene.apache.org/core/8_6_3/analyzers-common/index.html

[5] https://en.wikipedia.org/wiki/Stemming

[6] https://alibaba-cloud.medium.com/analysis-of-Lucene-basic-concepts-5ff5d8b90a53

[7] https://Lucene.apache.org/core/8_6_3/core/org/apache/Lucene/search/package-summary.html#package.description https://Lucene.apache.org/core/8_6_3/core/org/apache/Lucene/search/similarities/Similarity.html

[8] https://d1wqtxts1xzle7.cloudfront.net/30836579/FullProceedings.pdf?1362510774=&response-content-disposition=inline%3B+filename%3DFrom_Puppy_to_Maturity_Experiences_in_De.pdf&Expires=1604499920&Signature=K0yH1qZrrEmBU4H-hM564m4Q3qWS6~lGTmVDi7uyFbdTJn6KO~Ai2kPdO-Q-5v6PfjAdBY-~MJM805FXYfOxfvGehh2uykWqPgO95L2l5-VQFU-IDi66KriWbIA7yhLa4D1NcKs5C-8teuwUeymRZdZs5ak1rrw-GAPN3asuxTyRPx5bH9dfsK2DN70qzM7so-P-M7F26JqcUxzrgswvdBgX9VQO0bm7qFG~yLONzHlPA37ug0CinVK98YpCl8UcZmCTO~GtIboz0PfAlYa6X4~N-D7x8WJIr~X5pVKuLJjAMgFt5jqxAc2Sjgi1MvWu78fyWPsRT9D46eiQ~pmtSA__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA#page=22

[9] https://Lucene.apache.org/core/8_6_3/core/org/apache/Lucene/search/similarities/TFIDFSimilarity.html

[10] https://Lucene.apache.org/core/8_6_3/core/org/apache/Lucene/search/similarities/BM25Similarity.html https://en.wikipedia.org/wiki/Okapi_BM25

[11] https://Lucene.apache.org/core/8_6_3/core/org/apache/Lucene/search/similarities/BooleanSimilarity.html

[12] https://Lucene.apache.org/core/8_6_3/core/org/apache/Lucene/search/similarities/MultiSimilarity.html

[13] https://Lucene.apache.org/core/8_6_3/core/org/apache/Lucene/search/similarities/PerFieldSimilarityWrapper.html

[14] https://Lucene.apache.org/core/8_6_3/core/org/apache/Lucene/search/similarities/Axiomatic.html

[15] https://Lucene.apache.org/core/8_6_3/core/org/apache/Lucene/search/similarities/DFISimilarity.html

[16] https://Lucene.apache.org/core/8_6_3/core/org/apache/Lucene/search/similarities/DFRSimilarity.html

[17] https://Lucene.apache.org/core/8_6_3/core/org/apache/Lucene/search/similarities/IBSimilarity.html

[18] https://Lucene.apache.org/core/8_6_3/core/org/apache/Lucene/search/similarities/LMSimilarity.html

[19] https://Lucene.apache.org/core/8_7_0/core/overview-summary.html

[20] https://Lucene.apache.org/core/8_7_0/analyzers-common/index.html