Information Retrieval: Assignment 1

Jolan Depreter
Lander De Roeck

November 2020

1 Introduction

This report contains some of the functionality the Lucene has to offer, like the type of indices, the different score models, etc. It also presents our own implementation of an indexing, search and retrieval system build upon the Lucene library.

2 Lucene Functionality

2.1 Similarity

The Similarity class is responsible for computing the score, by default Lucene implements many different kinds:

- BM25Similarity: ranking function based on the query terms in each document, regardless of their proximity within the document
- BooleanSimilarity: Simple check if term exists or not
- MultiSimilarity: Use multiple similarity scorers and compute their norm to use as score.
- PerFieldSimilarityWrapper: allows for different scoring methods based on the document field.
- ClassicSimilarity (TFIDFSimilarity): Vector Space Model as seen in the lecture.
- Others: Axiomatic, DFISimilarity, DFRSimilarity, IBSimilarity, LMSimilarity

The implementations above are sub classes of the BaseSimilarity class, meaning you could create your own Similarity class to fit your needs.

2.2 Index Creation

2.2.1 Analyzers

The purpose of Analyzers is to extract the terms from the documents that will be indexed, Lucene offers a basic StandardAnalyzer which simply returns every term for indexing. The user can also create it's own Analyzer with a list of blacklisted terms that the analyzer won't index, for ease of use Lucene already offers a plethora of analyzers for different languages, these analyzers have a list of stop words to filter out for each language. For example: EnglishAnalyzer, DutchAnalyzer, SmartChiniseAnalyzer, JapaneseAnalyzer. There exists even a tokenizer specific for Wikipedia syntax: WikipediaTokenizer. Unfortunately, we picked the StackOverflow dump.

2.2.2 IndexWriter

The IndexWriter takes an Analyzer as input and is responsible for writing out the index to a directory, so we don't need to recompute the entire index each time we want to search. The default analyzer used is the StandardAnalyzer with an empty set of stop words. To start creating an index we need to convert our indexable files to lucene documents.

2.2.3 Documents and Fields

Lucene uses its own special Document class to compute an index. These documents can contain a wide variety of different fields. For indexing text we can use a TextField. This provides an index for full text search on a given string or file-buffer. Lucene is able to store this field for retrieval by using *Field.Store.YES*, *Field.Store.NO* will do the opposite and won't return the value on retrieval. Next we have the StoredField. This can be used to add information to this document's index without indexing the information in that field. This could be useful for storing where this documents exists, so we can retrieve is easily after querying. There are many more field types, including but not limited to:

- StringField: String indexed verbatim as a single token
- IntPoint: int indexed for exact/range queries.
- DoublePoint: double indexed for exact/range queries.
- SortedDocValuesField: byte[] indexed column-wise for sorting/faceting
- NumericDocValuesField: long indexed column-wise for sorting/faceting

The field classes can be easily extended with an own implementation. After adding at least one indexable field to a document, we can add this document to the index by calling: *IndexWriter.addDocument(Document)*. After adding all the documents we want to index we close the IndexWriter with *IndexWriter.close()*, which releases the lock on the index files.

2.3 Querying

2.3.1 IndexReader

The IndexReader can read the generated index out of a directory, it does however not update automatically when changes are made to the index. Lucene does luckily provides an efficient way of manually updating the IndexReader by calling: *DirectoryReader.openIfChanged(DirectoryReader)*, which will share resources with the previous reader. IndexReader implements two subclassses: LeafReader and CompositeReader. The CompositeReader is returned from *DirectoryReader.open()* and uses many LeadReaders to retrieve stored fields, doc values, etc.

Using the IndexReader as input we can create an IndexSearcher, this searcher provides a search method to scan through the documents and retrieve the n best scoring for a certain query.

2.3.2 Query Types

Queries in Lucene are a string with a certain syntax, comparable to SQL queries. While it is possible to manually create such a query string, Lucene strongly recommends to use the provided wide array of different QueryBuilders. These queries get parsed through the QueryParsers. Given an input string and, if needed, extra options, it builds the query for us. Every Parser also requires an Analyzer, it is best practice to use the same one you used to create the IndexWriter.

The simplest query is the TermQuery, it matches the document against a single Term. Next we take a look at the PhraseQuery, this matches a document against a sequence of terms. The terms must match in order! That makes this query very specific. There exists a slight variant on this query: the MultiPhraseQuery. This is a more advanced implementation, which allows for multiple terms at a single position in the phrase.

We also have the BooleanQuery, these can match a document against boolean combinations of a list Queries, even other BooleanQueries. This can be used to match for multiple terms using TermQueries without the terms having to be in a certain order like the Phrase Query. It can also be used to exclude certain terms or to force a certain term / query appears. Next we have the DisjunctionMaxQuery, which takes a list of Queries, like the BooleanQuery but does not combine the score, instead just using the maximum score of all given queries.

To boost the scores of a Query we can use the BoostQuery. In combination with the BooleanQuery, this could be used to prefer certain kinds of queries over others. Finally we have the ConstantQuery. This Query simply returns one or zero if the document does or does not match the query. We did not find a use for this Query.

To query over multiple fields at once, we can use a MultiFieldQueryParser. This parser also supports different weights for different fields. For example: Let's say we have the fields title and body, then if I were to search for a query that matches to different documents. In one document it matches in the title

and for the other in the body. Then the document that got matched on the title should be more relevant. So using this parser, we can add a higher weight to the score of the title field.

2.4 Spell correction and auto-completion

Lucene supports a build in spelling checker, the spelling checker uses a string distance method to compare input to the provided dictionary. The spellchecker can be used to correct mistakes in the input query terms and provide more accurate results. The build-in options for string distance algorithm are:

- JaroWinklerDistance
- LevenshteinDistance *(default)*
- LuceneLevenshteinDistance
- NGramDistance

On top of the spelling checker, Lucene also offers a suggestion class, which can provide auto-complete functionality.

3 Project

3.1 Document pre-processing: XML parsing and splitting

For the retrieval part of the assignment we needed to split up the dump file into multiple smaller files, each containing a question and it's answers. There are multiple way to do this, since the dump file is way too big to fully read into memory we need to create a stream based reader, which only loads a part of the file in memory at one time. This is the most efficient way to read out the file, but even reading out the file sequentially will take a very long time due to the sheer size of it. Because we wanted to work with the newest data, which likely be more relevant than the old, we also had an additional issue that we would need to skip over the first data. Reading through this data, even sequentially, would take too long. To tackle these problems we created a file reader in .Net Core, this file reader is optimised to skip over a variable portion of the dump file, and sequentially read out the remaining part, saving the contents to smaller XML files which contain a question and it's answers.

The basic logic of the reader isn't too hard to follow, we create a filestream and use the file size to calculate a position in the file, after which we jump to said location. Since we are in a random location inside the file we will search for the first line-break we find. After finding the line-break we know each line will contain a valid XML structure containing one message, so we start sequentially reading the file, parsing each line to get the message contents. When we come across a question, we create a XML file with name *id*.xml. When we come

across an answer we check if the question is in our data set, if it isn't we ignore the answer, if it is we append the answer to the corresponding question XML. Alternatively we could store it to a database file, which would increase performance as it doesn't need to write multiple small files to disk, we however chose for the small xml files so we could do the retrieval part of the exercise.

3.2 Indexing

Only a few Field classes are needed for our use-case. For indexing text we can use a TextField. This provides an index for full text search on a given string or file-buffer. In our case it will be strings, since we will be parsing the XML files first. We can ask Lucene to also store this string, using Field.Store.YES. This will be useful later on. The last Field we used is a StoredField. This can be used to add information to this document's index without indexing the information in that field. We used it to store the path to this document, so we can easily retrieve it after querying. Our documents are set up in a way, such that it will be easier to score our queries later on. First of all, we store the path to the document without indexing it (StoredField).

Next we divide the StackOverflow post in different sections and put them in TextFields:

- Question Title (stored)
- Question Body
- Accepted Answer
- Other Answers

This document is then passed to an IndexWriter.

3.3 Querying

To query our different fields, we used the MultiFieldQueryParser. This generates a Query for an input string, a list of fields and score booster per field. We used this to give the question title and the accepted answer a bigger weight. Since that part of the document seemed most important to us. However we wanted more control over our query syntax and this could not be achieved using MultiQueryParser. Luckily, we can implement multi-field search ourselves. For better relevancy we could have also taken the score and the date into account, by boosting higher up-voted and newer questions.

3.3.1 Our Query Syntax

Our query syntax is a subset of the google search syntax. You can use "" quotation marks, so a certain term must appear in the document and - negation to exclude a term from the documents. To achieve this, we made use of a combination of boolean queries, boost queries and the standard query

builder. “” is achieved using *BooleanClause.Occur.MUST* and - using *BooleanClause.Occur.MUST_NOT*.

3.4 Comparing Implementations & Score

3.4.1 Analyzer

Here we investigate the influence the analyzer has on the score of a query: In this comparison we look at the query: const. We always used the default similarity (BM25Similarity) for scoring.

Limited subset of 10.000 files

Document	English Analyzer	Simple Analyzer	Standard Analyzer
63276920	14.546543	16.41507	15.166318
63281043	12.426927	12.009628	12.3171
63265899	5.514491	4.845581	5.422535
63270674	NOT FOUND	4.845581	NOT FOUND
63267228	NOT FOUND	3.3458996	NOT FOUND
63268055	2.8042016	2.485289	2.5694547

Shockingly, both the Standard and English analyzer missed some documents. Upon closer look, it becomes clear that for terms like “lucene” there exists a small issue: These analyzers do not split terms on a dot before indexing. Therefore *‘en.lucene’* or *‘org.apache.lucene.store.MMapDirectory...’* does not get recognised on the termquery “lucene”.

Full subset of $>$ 210.000 files

	English Analyzer	Simple Analyzer	Standard Analyzer
Total hits	39	69	39

Sorted on descending score Simple Analyzer.

Document	English Analyzer	Simple Analyzer	Standard Analyzer
63379581	21.823944	20.852154	21.629093
63476621	21.261738	20.305462	21.074787
63333920	21.362043	20.237164	20.871498
63509841	15.278131	19.295286	15.7742195
63276920	16.476843	17.929491	17.200945
63374039	9.48103	17.42485	9.856176
63642733	15.417741	16.519659	15.781178
63415590	15.188727	16.503597	14.6678505
63735582	15.700655	15.581157	15.7705145
63281043	14.005995	13.548944	13.895534

The big difference in score in the highlighted column can be attributed to the same problem explained above. This also explains the 40 missing document hits.

3.4.2 Similarity

Limited subset of 10.000 files. Using EnglishAnalyzer.
SearchQuery: lucene

Document	BM25Similarity	BooleanSimilarity	ClassicSimilarity
63276920	14.546543	3.0	9.304433
63281043	12.426927	3.0	7.735965
63265899	5.514491	1.0	1.5599034
63268055	2.8042016	1.0	0.4757661

As expected, the BooleanSimilarity is mostly useless. It just detects if the term is present in a field or not. It does not distinguish between them.

Subset of $>$ 210000 files. Using SimpleAnalyzer. SearchQuery: lucene

Document	BM25Similarity	ClassicSimilarity
63379581	20.852154	10.072334
63476621	20.305462	9.827312
63333920	20.237164	8.963651
63509841	19.295286	13.607705
63276920	17.929491	11.0132065
63374039	17.42485	9.2265415
63642733	16.519659	9.222331
63415590	16.503597	8.823314
63735582	15.581157	10.231661
63281043	13.548944	7.4191236

The results are quite interesting, the scorers do not agree on the order on the documents. Let's take a closer look at the two top scoring documents for each similarity (highlighted). The top scoring document for BM25Similarity contains the term lucene in the title, title body, and answer. There are only a few occurrences, but the document is quite short. (Stackoverflow link). On the other hand, the ClassicSimilarity has more references in the document but the document is longer (Stackoverflow link). This is because BM25Similarity takes the relative length of the document into account [Wik]. In general BM25 is considered superior to TF-IDF [Foua]. From now on, we will use BM25 for our scorer.

3.4.3 Query

Subset of $>$ 210000 files. Using SimpleAnalyzer, BM25Smimilarity. SearchQuery: lucene

Document	Custom Query	MultiFieldQuery
63379581	20.852154	20.852154
63476621	20.305462	20.305462
63333920	20.237164	20.237164
63509841	19.295286	19.295286
63276920	17.929491	17.929491
63374039	17.42485	17.42485
63642733	16.519659	16.519659
63415590	16.503597	16.503597
63735582	15.581157	15.581157
63281043	13.548944	13.548944

For a simple query, our implementation is equivalent to the builtin multifield-query.

3.5 Command line arguments

3.5.1 Indexer

The indexer can accept two cli arguments, namely the input and output folder. The input folder needs to contain the XML files that need to be indexed, while the output file will be where the index is stored.

```
> java -jar Interface.jar indexer <input_directory> <output_directory>
```

3.5.2 Querier

The querier accepts multiple arguments, the first argument needs to be the index directory, the rest of the arguments will be parsed as search terms.

```
> java -jar Interface.jar querier <index_directory> <search terms>
```

3.5.3 Example

We will give some examples as to how to run the code. First we need to pre-process the StackOverflow dump, after which we can index these files with the indexer. In this example we will be reading 1% of the file (which means we will skip over 99%).

```
> BigXMLReader.exe 0.99 ./dump/Posts.xml ./dump/xml/
> java -jar Interface.jar indexer ./dump/xml/ ./index/
```

With the files indexed, we can now start searching in them, we will give some example scenarios.

Search for the term “lucene”:

```
> java -jar Interface.jar querier ./index/ lucene
```

Searching on multiple terms is also possible, like “stack” and “overflow”:

```
> java -jar Interface.jar querier ./index/ stack overflow
```

We can also search on an exact match of a word, like “lucene”. This will search for documents that have an exact match of the word. It can of course be multiple words if you want an exact match of a sentence.

```
> java -jar Interface.jar querier ./index/ '"lucene"'
```

Lastly we can also exclude a term from the results, let's say we want all occurrences of the word “lucene” but never with “python”

```
> java -jar Interface.jar querier ./index/ lucene -python
```

References

[Foua] Apache Software Foundation. *ClassicSimilarity*. [Online; accessed 6-November-2020]. URL: `https://lucene.apache.org/core/8_6_3/core/org/apache/lucene/search/similarities/ClassicSimilarity.html`.

[Foub] Apache Software Foundation. *Lucene Documentation*. [Online; accessed 6-November-2020]. URL: `https://lucene.apache.org/core/8_6_3`.

[Tut] Tutorialspoint. *Java DOM Parser - Parse XML Document*. [Online; accessed 29-October-2020]. URL: `https://www.tutorialspoint.com/java_xml/java_dom_parse_document.htm`.

[Ven] Sripriya Venkatesan. *Apache Lucene Hello World Example*. [Online; accessed 6-November-2020]. URL: `https://examples.javacodegeeks.com/core-java/apache/lucene/apache-lucene-hello-world-example/`.

[Wik] Wikipedia. *Okapi BM25*. [Online; accessed 6-November-2020]. URL: `https://en.wikipedia.org/wiki/Okapi_BM25`.