Project of information retrieval

GIANLUCA PULERI∗†, SIMONE MONTI∗†

∗UNIVERSITY OF ANTWERP, BELGIUM †UNIVERSITY OF MILANO – BICOCCA, ITALY

EMAIL: GIANLUCA.PULERI@STUDENT.UANTWERPEN.BE, SIMONE.MONTI@STUDENT.UANTWERPEN.BE

GIT REPOSITORY: https://github.com/montis96/InformationRetrieval2019-2020

1 ABSTRACT

The explosion of internet and digital repositories make available for everyone a big amount of data that is growing by time. It led us in the direction to need different and more dynamic ways to find information. This problem has attracted several research efforts in the Information Retrieval society.

In this paper we will deal with Lucene, an open source Java library that lets you add search to any application. We will present the functionality offered by Lucene and we will investigate on the feasibility of using Lucene to add retrieve capabilities. We will also analyse the impact on retrieval performance when implementing Rocchio algorithm for relevance feedback and query expansion using an automatically generated thesaurus (Wordnet).

2 INTRODUCTION

For thousands of years people have realized the importance of archiving and finding information. With the advent of computers, it became possible to store large amounts of information; and finding useful information from such collections became a necessity. Usually documents were given in various formats without following any schemas (unstructured documents). For this reason, to identify and retrieve valuable information from these documents became very difficult. Lucene is an open source Java search library which allows you to create an index and apply query to that, and it is meant to be a concrete solution for this problem. It is a project developed in the Apache Software Foundation and it is under the liberal Apache Software License. Thanks to its open source license and its scalability, nowadays Lucene is become powerful and one of the most used libraries for the information retrieval. Lucene presents also several integrations to other programming languages such as C++, Python etc. but is preferable to use the java version since it has more extensive documentation and is used by many more developers.

3 LUCENE CORE

Lucene library allows to create an index and make searchable data that you can extract text from, without caring about the source, the format or the language of data. Moreover, it manages all the complexity of indexing and searching using its own API.

3.1 INDEXING

3.1.1 INDEX CONSTRUCTION

Lucene provides a mapping of terms that is called an inverted index, that consists of indexing each term of documents with the list of all documents that contains this term and, optionally, also the positions of the occurrences can be stored. Lucene first analyses the text, a process that splits the textual data into a stream of tokens and performs an optional pre-processing on them. An index may store a heterogeneous set of documents and you must extract real text from your data, it means that you should pay attention when you are indexing XML or HTML files because you need to avoid the indexing of tags. Pre-processing consists in the transformation from natural language text to a computer-friendly one; Lucene performs this through the using of filters. Easy examples of pre-processing-technics are:

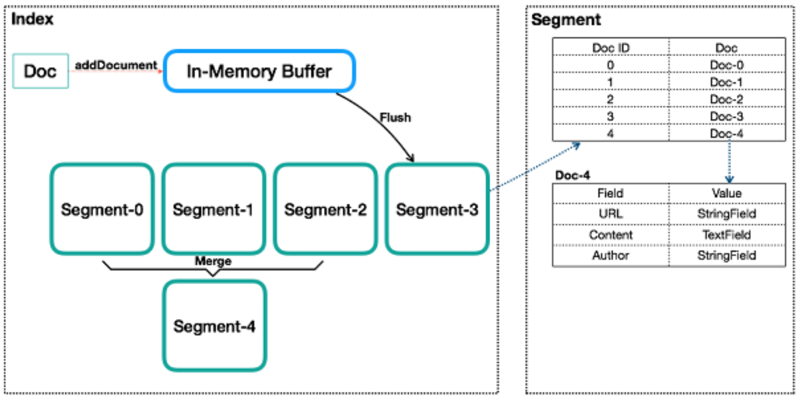
- Tokenization (*StandardTokenizer())*
- Stop word removal (*StopFilter()*)
- Stemming (*PorterStemFilter()*)
- Lemmatization (with external libraries)
- Spell correction (SpellChecker() using n-gram method and Levenshtein distance)

You can use built-in Analyzer with standard implemented in filters or create your own Analyzer by chaining together Lucene's token and filters in customized ways. After the analysis the input is ready to be added to the inverted index, Lucene has a rich and detailed index file format that could be optimized with your purposes. A Lucene index consists of one or more segments that is a standalone index containing a subset of all indexed documents. At search time each segment is visited

separately, and the results are combined.
Very important when indexing documents is the *Field* class. When you create a field, you specify a lot of options and controls about what Lucene must do during the addition of a document to the index, for example by changing a field's boost factor, you can instruct Lucene to consider it important with respect to other fields. Documents are collections of fields and Lucene refers to them through an integer *document number* starting from zero. Considering that each document number is unique within the segment, at each of them is assigned a range of values, to avoid problem at merge-time. They are the atomic units of indexing and writing.

Each segment is composed by:

- *Segment info*: this contains metadata about a segment, such as the number of documents, what files it uses.
- *Field names:* this contains the set of field names used in the index.
- *Stored Field values:* this contains, for each document, a list of attribute-value pairs, where the attributes are field names. These are used to store auxiliary information about the document, such as its title, URL, or an identifier to access a database. The set of stored fields are what is returned when you ask Lucene to give you back a document. This is keyed by document number.



Doc ID	Doc
0	Doc-0
1	Doc-1
2	Doc-2
3	Doc-3
4	Doc-4

Field	Value
URL	StringField
Content	TextField
Author	StringField

Figure 1 – Basic internal structure of an index

- *Term dictionary*: a dictionary containing all the terms used in all the indexed fields of all the documents. The dictionary also contains the number of documents which contain the term, and pointers to the term's frequency and proximity data.
- *Term Frequency data*: for each term in the dictionary, the numbers of all the documents that contain that term, and the frequency of the term in that document, unless frequencies are omitted (IndexOptions.DOCS_ONLY).
- *Term Proximity data*: for each term in the dictionary, the positions where the term occurs in each document. Note that this will not exist if all fields in all documents omit position data.

- *Normalization factors*: for each field in each document, a value is stored that is multiplied into the score for hits on that field.
- *Term Vectors*: for each field in each document, the term vector (sometimes called document vector) may be stored. A term vector consists of term text and term frequency. To add Term Vectors to your index see the Field constructors.
- *Per-document values*: like stored Field values, these are also keyed by document number, but are generally intended to be loaded into main memory for fast access. Whereas stored Field values are generally intended for summary results from searches, per-document values are useful for things like scoring factors.
- *Live documents*: an optional file indicating which documents are live.
- *Point values*: optional pair of files, recording dimensionally indexed fields, to enable fast numeric range filtering and large numeric values like BigInteger and BigDecimal (1D) and geographic shape intersection (2D, 3D). [1]

3.1.2 INDEX COMPRESSION

The index has many integer data, whose representation without compression is obviously unreasonable. Index compression could reduce storage space, furthermore, comparing with uncompressed index, transfer more information per second on the same I/O bandwidth. Lucene naturally uses Variable-Byte to encode integer. Variable-Byte Coding uses 8 bits as the minimal storage unit; an integer n is encoded as $\lfloor \log_{128}(n) \rfloor + 1$ bytes. For the binary expression of n, from low-bit to high-bit, the essence of the method puts every 7 bits in one byte's low 7 bits, the highest bit of byte is regarded as flag-bit, "0": current byte is n's last byte; "1": next byte must be fetched. Method circularly processes n, until there is a byte with a "0" flag-bit.

3.2 QUERY

Lucene's search methods require a *Query* object, the library implements a *QueryParser* class that allows you to translate text entered by the user into an appropriate equivalent *Query*. Lucene provides several built-in Query types, described in the *Table 1*, but also it supplies to construct queries programmatically.

Query Implementation	Purpose	Sample expressions
`TermQuery`	Single term query, which effectively is a single word.	reynolds
`PhraseQuery`	A match of several terms in order, or in near vicinity to one another.	“light up ahead”
`RangeQuery`	Matches documents with terms between beginning and ending terms, including or excluding the end points.	[A TO Z] {A TO Z}
`WildcardQuery`	Lightweight, regular-expression-like term-matching syntax.	j*v? f??bar
`PrefixQuery`	Matches all terms that begin with a specified string.	cheese*
`FuzzyQuery`	Levenshtein algorithm for closeness matching.	tree~
`BooleanQuery`	Aggregates other Query instances into complex expressions allowing AND, OR, and NOT logic.	reynolds AND “light up ahead” cheese* -cheesewhiz

Table 1 - https://www.javacodegeeks.com/2015/09/advanced-lucene-query-examples.html

If the Query is created by the *QueryParser* the free text is converted into each of the Query types described.

3.3 Searching

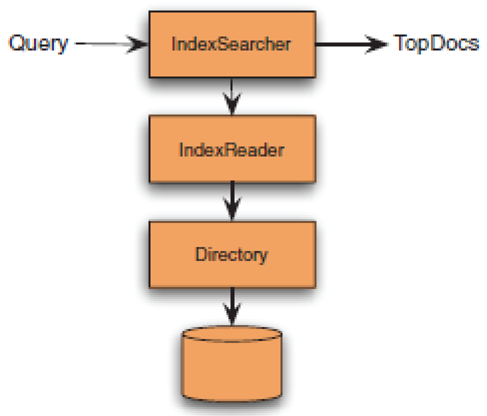


Figure 2 - Relationship among the classes used for searching

Lucene implements an easy and efficient way to retrieve documents, using only few classes you can make your own basic SearchFile. The main one is the *IndexSearcher* that can perform a search through the *search()* methods. First you need to instance an IndexReader object, this is an abstract class, providing an interface for accessing a point-in-time view of an index, you must pay attention that any changes made to the index via IndexWriter will not be visible until a new IndexReader is opened. The search methods visit every document that is a candidate or that satisfy every constraint of the query and once got the top results it will return them to you, it does a wide amount of work and performs it very quickly. It returns a *TopDocs* object that contains the top hits, or *ScoreDoc* for the query, ordered by relevance. The terms passed to the indexSearcher must be consistent with the terms produced by the analysis of the source document during the indexing, which means to use the same Analyzer for indexing and searching.

3.4 Scoring

For every document that matched during a search, it is assigned a score that points the similarity between the document and the query. The higher is the score, the stronger is the similarity. Lucene scoring supports several pluggable information retrieval models, including:

- *Vector Space Model (VSM)*
- *Probabilistic Models*, such as *BM25* (the default scoring model)
- *Language based model*

These models can be plugged in via the *Similarity API.* Generally, the Query determines which documents match (a binary decision), while the *Similarity* determines how to assign scores to the matching documents. Changing Similarity is an easy way to influence scoring, this is done at index-time and at query-time. We have Classic*Similarity* class for VSM, *BM25Similarity* for BM25 and *LMSimilarity* for Language-based model.

3.4.1 VECTOR SPACE MODEL

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. *VSM score* of document d for query q is the Cosine Similarity of the weighted query vectors V(q) and V(d).

$$\text{cosine-similarity}(q,d) = \frac{V(q) \cdot V(d)}{|V(q)|\ |V(d)|}$$

Figure 3 – VSM score

Lucene refines *VSM score* for both search quality and usability. Therefore, making the following assumptions we get the *Lucene's Conceptual scoring formula*:

- A query-boost(q) is used by the users to give importance to documents.
- A doc-len-norm(d) is used to normalize to a vector equal to or larger than the unit vector, since normalizing to the unit vector can be problematic because it removes all document length information.
- A doc-boost(d) is used by the users to give importance to documents.

$$\text{score}(q,d) = \text{query-boost}(q) \cdot \frac{V(q) \cdot V(d)}{|V(q)|} \cdot \text{doc-len-norm}(d) \cdot \text{doc-boost}(d)$$

Figure 4 – Lucene's Conceptual scoring formula

Some scoring components are computed and aggregated in advance for efficient score computation and making some assumptions we can derive the *Lucene's Practical Scoring Function*:

- *query-boost(q)* is known when search starts.
- The normalization of *V(q)* can be computed when search starts, as it is independent of the document being scored.
- *doc-len-norm(d)* and *doc-boost(d)* are known at indexing time. They are computed in advance and their multiplication is saved as a single value in the index: *norm(t,d).*

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

Figure 5 – Lucene's Practical scoring function

Where:

- *tf(t in d)* correlates to the term's frequency
- *idf(t)* stands for Inverse Document Frequency. idf(t) appears for t in both the query and the document, hence it is squared in the equation
- *t.getBoost()* is a search time boost of term *t* in the query *q* as specified in the query text
- *norm(t,d)* is an index-time boost factor that solely depends on the number of tokens of this field in the document, so that shorter fields contribute more to the score. [2]

3.4.2 PROBABILISTIC MODELS – BM25

BM25 has its roots in probabilistic information retrieval. Basically, it casts relevance as a probability problem. A relevance score, according to probabilistic information retrieval, ought to reflect the probability a user will consider the result relevant. Lucene can score documents through the classic *BM25 formula*:

$$RSV_d = \sum_{t \in q} \log \left[\frac{N}{\text{df}_t} \right] \cdot \frac{(k_1 + 1)\text{tf}_{td}}{k_1((1-b) + b \times (L_d / L_{\text{ave}})) + \text{tf}_{td}}$$

Figure 6 – BM25 formula

where:

- tf_{td} is the term frequency in document d
- L_d and *Lave* are respectively the length of document d and the average document length in the entire collection

- k_1 is the tuning parameter controlling the document term frequency scaling (default 1.2)
- *b* is the tuning parameter controlling the scaling by document length (default 0.75)

3.4.3 LANGUAGE BASED MODEL

Lucene implements two language models, *LMDirichletSimilarity* and *LMJelinekMercerSimilarity*, based on different distribution smoothing methods.

3.5 RELEVANCE FEEDBACK

The idea of relevance feedback is to involve the user in the retrieval process to improve the result set. In particular, the user gives feedback on the relevance of documents in an initial set of results. The basic procedure is:

- The user issues a (short, simple) query.
- The system returns an initial set of retrieval results.
- The user marks some returned documents as relevant or nonrelevant (this point could be automated).
- The system computes a better representation of the information need based on the user feedback.
- The system displays a revised set of retrieval results. [3]

There are three types of relevance feedback:

1. Explicit feedback: user indicates the relevance of documents retrieved by a query. The problem is that users could be reluctant to provide explicit feedback.
2. Implicit feedback: it consists of implicitly taken feedback from actions of user like the time spent on viewing a document or the kind of document they view or not.
3. Pseudo relevance feedback: it automates the manual part of user feedback, it assumes that the top k documents are relevant, and we use them to do the relevance feedback.

3.6 QUERY EXPANSION

Query expansion is the process of reformulating the initial query by adding additional meaningful terms with similar significance, acted to improve retrieval performance. Words are considered similar if they co-occur with similar words or if they occur in a given grammatical relation with the same words. Thanks to QE we could retrieve documents even if there is no term match with the original query. Normally it leads to an improvement of recall but decreasing the precision. Main techniques of QE:

- Manual thesaurus (https://www.thesaurus.com/).
- Automatically derived thesaurus.
- Query-equivalence based on query log mining (used normally in the web).

3.7 Link analysis

In Lucene there are not present classes that implement link analysis directly, like PageRank or Hub and Authorities but there exist many java implementations developed by the users. One of the most popular is the class PageRank<V,E>. [8]

4 Description of our project

Our developing started from the standard Lucene's demo *org.apache.lucene.demo.SearchFiles* and *org.apache.lucene.demo.IndexFiles [4].* This paragraph will be focused on the code changes acted to adapt them to our project solutions.

4.1 Dataset

The data to be used in the project consist of a dump of StackOverflow questions and answers saved in the XML format following these rules:

- Every file starts and finishes with the root tag (*<root> data </root>*).
- A question is included between *<question>* and *</question>*.
- The question is composed by the title, the body of the question and the tags.
- We have one replies tag that contains all the answers to that question.
- Finally, we have for each answer two tags (answer and body) that contain the text of the answer.

Format:

```
<root>
<question>
<Title> Title of the question </Title>
<Body> Body of the question </Body>
<Tags>Tags of the cquestion</Tags>
</question>
<replies>
<answer><Body> Text of the answer </Body></answer>
 [...]
<answer><Body> Text of the answer </Body></answer>
</replies>
</root>
```

4.2 IndexFiles

We create a Lucene index using the *StandardAnalyzer()* that applies to every document the *StandardTokenizer* with *LowerCaseFilter* and *StopFilter*. We used the Oracle Digester class [5] that provide a common implementation to map an XML file to a Java object, in our case the *Question* class, that contains one attribute for each XML tag (*String title, String body_question, String tags, List<String> answers*) and the standard methods. We have done three kinds of Index:

1. *IndexFilesBM25()*: used to index all the documents that we want to retrieve with the standard similarity class (BM25). We indexed every document tag in separated fields, but we did not index the title (later we will explain the reason).
2. *IndexFilesTFIDF()*: as above but the documents are retrieved with the Lucene TFIDF similarity (ClassicSimilarity).
3. *IndexTitles()*: we indexed only the title and the tags for every document, a smaller version of *Question class* is used (*QueryXML class*).

During the indexing process we stored some titles in an XML file that have been used as queries for the evaluation.

4.3 SearchFiles

The SearchFiles classes search for a specified query into the index built by Index-Files classes. They use an IndexReader, IndexSearcher, StandardAnalyzer (which is used in the IndexFiles classes as well) and a QueryParser that allows us to construct queries for multiple fields. We have these kinds of SearchFiles:

- *SearchFilesTFIDF()* and *SearchFilesBM25()*: they are classes very close to the SearchFile demo provided by Lucene. The main difference between these files is the Similarity class used for the scoring, the ClassicSimilarity for the first one and the BM25Similarity for the second one.
- *SearchFilesRocchio()*: it is one of the most popular and widely applied learning methods, Rocchio proposed a classical query expansion algorithm based on the Vector Space model:

  $$qtf_m = \alpha . qtf + \beta . \sum_{d \in R} tf - \gamma . \sum_{d \in N} tf$$

  We have implemented a Pseudo-relevance feedback using Rocchio's and we decided to put K equal to 7, so we consider the top 7 documents retrieved as relevant.
- *SearchFilesWordnet()*: WordNet is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. [6] We implemented the algorithm using the *WordnetSynonymParser()* provided by Lucene.
- *SearchTitles()*: class similar to the SearchFileBM25 but it uses only title and the tags fields, we use this in order to find relevant documents for every query.

5 Study Impact

From the initial dataset we indexed around 270000 files. During the creation of the index we saved in XML files the title of the documents every 1800 files, each title is used as query (it means around 150 queries) and this is the reason why we did not index it. We know that at least one document is relevant (the document that has the title matching with the query), in order to find other relevant documents we created a new index (using *IndexTitles()*) in which we stored only titles and tags. For each query we performed another search through *SearchTitles()* and using a threshold on the scores of the documents retrieved with this algorithm we found the relevant ones. The first document retrieved by *SearchTitles()* was always the matching file so, after a manual checking, we decided to consider relevant the documents that have scores higher than the first one divided by 2.4.

5.1 People Search Behaviour

Most people, after a web research, will click on one of the first few results, different reasons are that they have found what they were looking for, they have only short time, or they do not want to scroll further. As you can see in the web page https://www.advancedwebranking.com/ctrstudy/ the

CTR of the first page retrieved is around the 71% while the second and the third page combined reached the 6%. If we go further, we find that on the first page alone, the first 5 results account for 67.60% of all the clicks and the results from 6 to 10 account for only 3.73% [7]. This project is based on the query-search in the StackOverflow portal, given that the middle user of StackOverflow is a programmer and they will not stop their searching only at the first three or four results, for our analyses we took this assumption: the user will examine the top-10 results (first page).

5.2 AVERAGE, MODE, MEDIAN

	Average	**Mode**	**Median**	**notNull(%)**
SearchFilesBM25	2.164	1	1	48.667
SearchFilesTFIDF	2.348	1	1	30.667
SearchFilesRocchio	2.275	1	1	26.667
SearchFilesWordnet	3	2	1	33.3

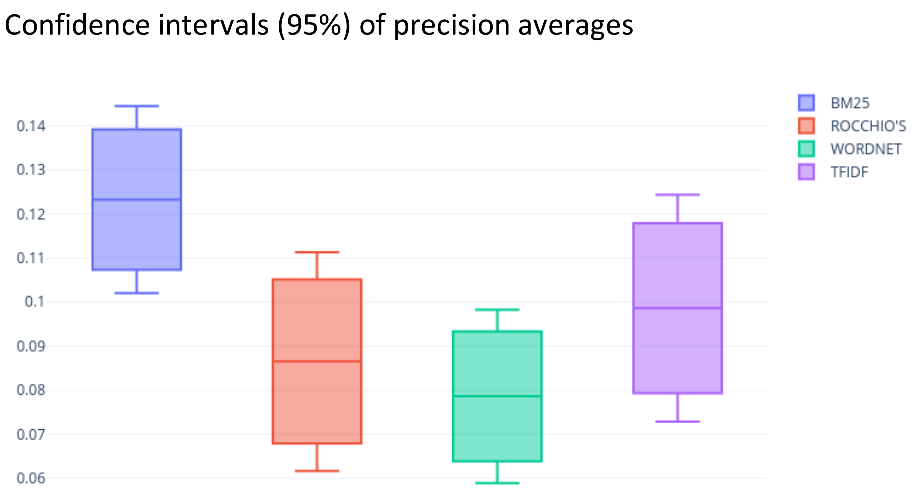
Table 2 - Average, mode and median

After the query searches we saved the position of the correct answer only if it was in the first 10 retrieved documents. We computed the average, the mode, the median and the percentage of not null values of the positions. *SearchFilesBM25* has the best scores, it is logical because BM25 is generally considered better than TFIDF and the main purpose of Rocchio algorithm and query expansion algorithm is improving the recall, so it justifies that the matching document is in a worse position than BM25.

5.3 PRECISION @10 AND RECALL @10

5.3.1.1 Precision@10

We have plotted the confidence intervals of the averages of precision and recall for every model.

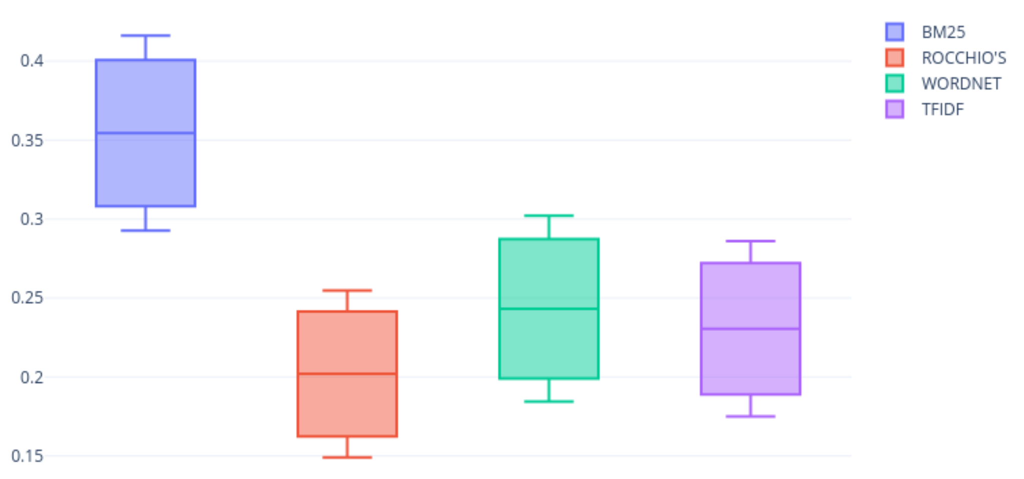


Precision	**BM25**	**Rocchio**	**Wordnet**	**TFIDF**
Average	0.123256	0.08651	0.07860	0.09861
Standard Dev.	0.131556	0.153934	0.121931	0.15958

The precision scores are as expected, the highest score is owned by the BM25 algorithm while the worst scores are owned by the Rocchio and the Wordnet algorithm, we have already explained in this paper the reasons of this behaviour. In this analysis we think that the precision is more important than the recall because normally the user start to see documents from the first one.

5.3.2 Recall@10

Confidence intervals (95%) of recall averages



Recall	**BM25**	**Rocchio**	**Wordnet**	**TFIDF**
Average	0.354446	0.201932	0.24326	0.230537
Standard Dev.	0.38239	0.327479	0.36506	0.344140

As we can see from the Recall's plot, the best model results to be the BM25. Normally the Rocchio and Wordnet algorithms should improve the recall but, in this case, there is a decreasing of it. The main reason for the decreasing score of the Wordnet algorithm is that the language used in this kind of documents is very precise and searching related words is not the most effective way to improve the amount of relevant retrieved documents. While regarding the Rocchio's algorithm the reason is inside the nature of the pseudo-relevance feedback: we are considering as relevant the first 7 documents without manually checking if they effectively are.

5.4 LIMITS OF THE EVALUATION

To find the relevant documents we tried to use different similarities inside *IndexTitles()* and *SearchTitles()* classes, manually comparing the results (how many and in which position relevant documents were retrieved) the best one seemed to be the BM25Similarity. The threshold used to split relevant from nonrelevant was obtained doing an average of the ratio between the score of the "correct" document and the last relevant one (manually checked) of ten queries. For these queries we found that before of this threshold all the documents were relevant while after that they were nonrelevant, but we are not sure that we can say this for all the queries. The scores of the different analysed algorithms were not influenced by the kind of similarity used to split relevant/nonrelevant documents and BM25 resulted always the best model.

6 REFERENCES

I Text:

[1] https://lucene.apache.org/core/8_3_1/core/org/apache/lucene/codecs/lucene80/package-summary.html#package.description

[2] https://lucene.apache.org/core/8_3_1/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html

[3] https://nlp.stanford.edu/IR-book/html/htmledition/relevance-feedback-and-pseudo-relevance-feedback-1.html

[4] https://lucene.apache.org/core/8_3_0/demo/index.html

[5] https://commons.apache.org/proper/commons-digester/

[6] https://wordnet.princeton.edu/

[7] https://moz.com/blog/google-organic-click-through-rates-in-2014

[8] http://jung.sourceforge.net/doc/api/edu/uci/ics/jung/algorithms/importance/PageRank.html

II Code:

[1] https://github.com/eric-cho/LuceneQueryExpansion/blob/master/src/main/java/in/student/project/queryexpansion/SearchFilesRocchio.java

[2] https://github.com/uiucGSLIS/ir-tools/blob/master/src/main/java/edu/gslis/lucene/expansion/Rocchio.java

[3] https://gist.github.com/gtke/5833889

[4] https://gist.github.com/guenodz/d5add59b31114a3a3c66