# Information Retrieval

Project Lucene
StackOverflow

*M INF 2019-2020*

August 7, 2020

**Group:** Johannes Akkermans,
Randy Paredis

**Abstract**

Lucene is a powerful Java-tool that allows for easy information retrieval of large datasets. The goal of this project was to apply Lucene on the StackOverflow data and study its feasability. Additionally, we will try to improve the overall performance of the retrieval by applying some techniques we've seen in class.

# 1 Lucene

Apache Lucene is a free and open-source search engine software library written in Java. It allows for full-text indexing and com.stackoverflow.searching making it a perfect fit for the project at hand. [2]

There are a lot of features Lucene offers, but we will not need all of them. This is why we will gradually go over our project and the Lucene features we've used to obtain the described results. All specific details on the internal workings of Lucene are described in its documentation, publicly available on [1].

## 1.1 Dataset

Of course, before we can do anything, we require a dataset on which we will execute Lucene. In order to do that, we've downloaded the massive dataset from StackOverflow[1]. Next, we extracted this dataset before running a Python script that parses this massive xml-file into 2 010 607 smaller files, containing questions and answers, all of whom had `python` or `c++` in its tags. Every 1000 files are put into a sub-folder, making sure our operating systems can handle this amount of data. The overall size approximates 7 gigabytes, which is why it's not accompanied with our submission of the project.

Now that we have this dataset, we can start looking at Lucene.

## 1.2 Analyzer and Setup

The first and foremost thing to do in Lucene is to create an `Analyzer`. As the name might suggest, it will analyze text by building `TokenStreams`[2]. More theoretically, it describes a policy on how items are extracted from text.

Lucene comes with a `StandardAnalyzer` class[3]. This subclass of the abstract `Analyzer` has a very broad use case, seeing as it correctly splits unicode text, turns everything to lowercase and removes a set of English stop words. It can be created quite easily:

```
1 StandardAnalyzer analyzer = new StandardAnalyzer();
```

Alas, the `StandardAnalyzer` does not take into account that our data consists of a lot of xml-files. On top of that, xml makes use of a special character encoding system that does not map straightforwardly on

[1] From the website indicated in the assignment.

[2] `org.apache.lucene.analysis.TokenStream`

[3] `org.apache.lucene.analysis.standard.StandardAnalyzer`

unicode. This is an issue we need to keep in mind when processing our data. Luckily, there is the possibility of creating a custom `Analyzer` (in our project, this is the `MyCustomAnalyzer` class) to solve this issue. More on the `MyCustomAnalyzer` in section 4.

## 1.3 Indexing

### 1.3.1 Index Directory

Now that we have an `Analyzer`, we need a place to keep track of the indexing. Because indexing itself is an expensive operation, it will not be done before every search and, on top of that, it will be stored in a folder on disk, making sure your indexes remain consistent (and above all: existent) between multiple executions.

In our implementation, we create a new directory, called `.search` in which we will store this information. If this directory already exists, it is cleared beforehand. Note that, this way, we will bypass the usecase of storing these files in a directory.

```
1 Directory index = new MMapDirectory(Paths.get(".search"));
```

### 1.3.2 Indexing of Files

The dataset we've used was too big to index as a whole for the scope of our project. This is why our code only indexes on a part of this data (10 000 files), which was randomly selected using Java's builtin random number generator with a fixed seed of 420 (for repeatability).

In order to do so, we had to flatten our dataset in such a way that our OS could handle it. Here is where we discovered a problem on MacOS (compared to Ubuntu LTS 18.04, that is). On Mac, the `File` class actually opens the file, causing this to be a costly operation. We solved this inconvenience via iterating over the filenames and only taking those ending in "xml"[4].

Either way, after we've obtained our files to index, we will add all these documents to an `IndexWriter`[5]. This object basically keeps track of all labeled information of the documents in our dataset.

```
1 IndexWriterConfig config = new IndexWriterConfig(analyzer);
2 IndexWriter w = new IndexWriter(index, config);
```

[4] This method has as a downside that it will not work on Windows systems. Mainly because we hardcoded the "/" as a path separator.

[5] `org.apache.lucene.index.IndexWriter`

For the purpose of the assignment, we indexed the fields *name* (i.e. the filename), *title*, *body*, *tags* and *answers*. These last four need to be transformed in such a way that all special characters and capitalization are removed (as mentioned in section 1.2, see also section 2). This is done by our `MyCustomAnalyzer` (as will be discussed in section 4). Additionally, the *answers* are all joined together, separated with two newline characters (`\n`), for ease of use.

Internally, Lucene does not search all the files, because that would be too costly. In fact, this indexing stage creates a list of indexes and the pages they refer to, similarly to something you might see in a handbook. These indexes are called "*reverse indexes*". [6]

### 1.3.3 Storing the Feature Vectors

Because we will need it later on (see section 5), we will also tell Lucene it needs to store the term vectors of all fields. This way, we hope to be able to use the Vector Space Model (VSM) Lucene uses behind the scenes.

### 1.3.4 Index File Formats

As mentioned before, Lucene stores the indexing data in a set of files. Its contents are not entirely readable by a human[6], making us suspect there is some sort compression or encoding that's being done by Lucene. Our suspicions were confirmed in [1], which also gives an extended overview of this encoding.

## 1.4 Score Models

There are numerous scoring models bundled with Lucene (as we will discuss in section 3.4). By default, Lucene uses the `TFIDFSimilarity`[7].

The fastest way to obtain a score in Lucene, would by via the following code:

```
// Search with the query
IndexReader reader = DirectoryReader.open(index);
IndexSearcher searcher = new IndexSearcher(reader);
TopDocs docs = searcher.search(q, cnt);
ScoreDoc[] hits = docs.scoreDocs;
```

[6] Using ourselves as a point of reference.

[7] `org.apache.lucene.search.similarities.TFIDFSimilarity`

Here, we would search for a query `q` and return at most `cnt` documents that match it, ordered by highest score first. To quote scoring in the words from [1], which also describes our love-hate relationship with it:

> *Lucene scoring is the heart of why we all love Lucene. It is blazingly fast and it hides almost all of the complexity from the user. In a nutshell, it works. At least, that is, until it doesn't work, or doesn't work as one would expect it to work.*

Allow us to emphasize that this quote comes unedited from their own website, basically stating that Lucene does not always works the way you want it to work, an issue we have experienced on numerous occasions during this project (especially in section 5).

### 1.4.1 Similarity Scoring

From [1], we can also conclude how its scoring actually works internally. From the course, we know there must be some similarity checking formula at work. We find that Lucene makes use of the VSM score, better known as the cosine simularity (with $V(q)$ and $V(d)$ weighed query vectors):

$$cosine\_similarity(q,d) = \frac{V(q) \cdot V(d)}{|V(q)| \cdot |V(d)|}$$

Its scoring formula therefore becomes:

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

## 2 Query Building

Even though Lucene makes use of a Vector Space Model (VSM), it is by default completely hidden. On top of that, we haven't found any information (nor in the documentation, nor anywhere online) on how to search the dataset by vector. We would have to create our own scoring system, an idea we quickly discarded because of the massive size of our data[8]. Instead, we decided to use the builtin binary query system, which is tested and optimized for com.stackoverflow.searching[9].

[8] We also discarded this idea because we did not believe this was intended with the assignment. In fact, the fast and efficient workings of the similarity checking as described in section 1.4.1 was a major influence in this decision.

[9] On top of that, it is impossible to obtain the VSM without giving such a query string.

To allow for simple manipulations of the queries we introduced the notion of a `QueryBuilder`. Based on a string with the valid search criteria, we create a query that follows the Lucene query syntax and remains conform with the inputted data. Even though this may seem as an unnecessary additional parsing step, it will become useful later on.

Let's take a deeper dive into our `QueryBuilder`.

## 2.1 String Replacements

Before we even start to build our query, we need to transform a string into a general syntax that can be easily understood without too much issues. We identify three phases:

First we have the trimming phase. Here, we remove all whitespace surrounding our query, i.e. at the beginning and the end. Although the Lucene documentation does not explicitly state this as a requirement, we wanted to make sure it does not cause any undefined behaviour.

Next, we transform our string into lowercase format. Let's call this the lowercase transformation phase. This was done because matching on capitalization is not useful within our problem domain. Of course, this also requires our documents to be transformed in a similar manner (see sections 1.3 and 4).

Finally, there is the alphanumerisation phase[10]. Here, we will remove all special characters from our queries, seeing as they will only cause issues and strange behaviour (i.e. leaving a period in the query could result in the match of any character). Admittedly, in doing so we loose the ability to match hyphenated words and other special cases like the query "`c++`". It was our hypothesis that these words would not take up too much of the language and thus are irrelevant in this context.

Assuming our query is stored in the variable `word`, these three phases look, quite elegantly, like this:

```
1 word.strip().toLowerCase().replaceAll("[^a-z0-9 ]", "")
```

Note that this only applies for the query itself. As will be discussed in section 4, for the documents themselves, this will be achieved via applying some additional filters to the analyzer.

[10] Or that's at least how we called it.

## 2.2 String Separation

As you might have guessed from the previous section, we have only removed all whitespace from the beginning and the end of our string. All spaces in the middle are left and for good reason. They tell our system that all of those words **must** appear together.

If we look at Google and how they go about spaces, you might be able to deduce from the url, that they replace all spaces with an `AND`-gate (the plus sign). This not only allows words to be in a different word order, but also allows words not to be next to one another. This method works wonders for Google, so why wouldn't it be in our system?

Let's say we have the query "`python database setup`". By default, Lucene adds an `OR`-gate between these terms, but we want to enforce that these three words occur together (not necessarily in that order, or next to one another). Thus, we will add explicit `AND`-gates between them. Therefore, we obtain the query "`python AND database AND setup`" as an input to Lucene.

For the further query building, we will be looking at this list of whitespace-less strings, separated with an `AND`. Such an individual string will be referred to as a "*term*".

```
String[] terms = word.split(" ");
String newWord = String.join(" AND ", terms);
```

## 2.3 Spell Correction

Fuzzy search is a powerful tool. It allows a search engine to find words that are similar, just in case the user made a typo in the query. Fuzzy search can be referred to as "*Did You Mean...?*" functionality.

In Lucene no spell checking is used by default, but there are multiple ways to obtain fuzzy search.

- Use the Lucene query syntax for fuzzy search. This is done via adding the tilde ($\sim$) to the back of a term. In exploring how Lucene works and what influence this syntax had, we have found it does not fully accomplish what we want to achieve.
  By default, this syntax will allow words to be at most 50% different. Within this context, the word "`bear`" will also match "`beer`", "`boar`", "`bend`", "`fear`", "`feat`", "`rear`" and "`zeal`" to name a few. All of the words mentioned above are real words, but the similarity measure does not make that assumption. Words like "`glar`", "`qeaq`"

and "`eeaa`" does not exist in the English language, but will also be matched. Hypothetically, a set of documents on a fictional alien language might use these non-existing words a lot. If there are very few documents describing bears in the dataset, I might get a lot of fictional alien language information, while I did not want this.

- Use Lucene wildcards. While at first sight, this might be a good idea, on second thought there are way too many issues with this method. First, a term is not allowed to start with a wildcard (presumably because it would make it left-recursive). Secondly, there are way too many possible locations to add wildcards. Do you add them between all letters? Do you replace all letters? Do you do a combination of the two? For this combination method and an $n$-letter word, there are $2^{2(n-1)}$ possibilities for adding wildcards. This is an upperbound, because there could be some overlap for some wildcards: `a**b` is equal to `a*b`.
  For instance, in a 3-letter word "`abc`", wildcards can be located at the letter locations (except for the `a`), or between all letters, giving us $2^{2+2} = 2^{2(3-1)} = 2^4 = 16$ possibilities. The wildcards will be added in the following form: "`abc`" $\rightarrow$ `a.b.c`, `a*b.c`, `a.*.c`, `a.b*c`, `a.b.*`, `a.b**`, `a.*.*`, `a*b.*`, `a**.*`, `a*b**`, `a**.c`, `a.**c`, `a.***`, `a*b**`, `a****`, `a***c`.[11]

- Generate all possible replacements for a term. We have an alphabet of 26 letters and 10 different digits with whom we can replace every single letter. But we hit the same wall as with the wildcards, only with a much higher set of possibilities ($36^n$ instead of $2^{2n-1}$). Figure 1 shows that, no matter the value of $n$, this option will always yield more possibilities.
  On top of that, if we say that we only replace one or two letters at a time, there are still unexplored options. Let's say we want to look for "`code`", but accidentally only type "`cod`", this method will not find the right documents.

- Lucene has this `SpellChecker` class that can suggest the $n$ closest words in the dictionary of all files. This method allows us to resolve typos and tweak the amount of possible replacements[12]. For our pur-

[11] The dot in the wildcard strings is just a seperation to have a better visual representation of the possibilities.

[12] It's illogical that each word should match all of its possibilities. From "`bear`" to "`beard`", "`boar`" or "`beer`" are smaller changes than replacing it with "`glad`".

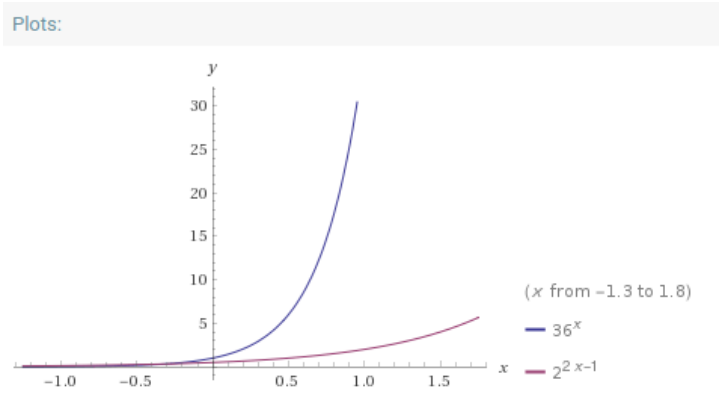



Figure 1: WolframAlpha-made plot of both $y = 36^x$ and $y = 2^{2x-1}$.

poses, we assumed 5 similar words (and the current one, making it 6 checks) should do the trick. We merge them all together in one big `OR`-gate for each term.

As you can guess, we have decided on the latter option, which can be implemented as follows (assume `term` is our term):

```
// Did you mean? https://www.javacodegeeks.com/2010/05/did-you-mean-feature-lucene-spell.html
SpellChecker spellchecker = new SpellChecker(spellIndexDirectory);
// To index a field of a user index:
spellchecker.indexDictionary(new LuceneDictionary(my_lucene_reader, a_field));
// To index a file containing words:
spellchecker.indexDictionary(new PlainTextDictionary(new File("myfile.txt")));
String[] suggestions = spellchecker.suggestSimilar(term, 5);
String newTerm = "(" + String.join(" OR ", suggestions) + ")";
```

Combined with String Separation, we get:

```
SpellChecker spellchecker = new SpellChecker(spellIndexDirectory);
spellchecker.indexDictionary(new LuceneDictionary(my_lucene_reader, a_field));
spellchecker.indexDictionary(new PlainTextDictionary(new File("myfile.txt")));
String[] terms = word.split(" ");
ArrayList<String> newTerms = new ArrayList<>();
for(String term: terms) {
```

```
    // Just in case there are multiple spaces
    if(term.isEmpty()) { continue; }

    List<String> suggestions = new ArrayList<>(Arrays.asList(
    spellchecker.suggestSimilar(term, 5));

    // Make sure the actual term is part of the query
    if(!suggestions.contains(term)) {
        suggestions.add(term);
    }
    newTerms.add("(" + String.join(" OR ", suggestions) + ")")
}
String newWord = String.join(" AND ", newTerms);
```

# 3 Benchmark Study

To evaluate the retrieval performance of Lucene, the PR- and the ROC-curve will do great at visualizing this performance. To plot these curves, the Precision ($P$), Recall ($R = TPR$) and Fallout ($F = FPR$) need to be computed, which will be realized by *manual labeling*. Formulas for these numbers are given as follows:

$$P = \frac{TP}{TP + FP} \qquad R = \frac{TP}{TP + FN} \qquad F = \frac{FP}{FP + TN}$$

Unfortunately, for our use-case, the ROC (and AUC) are not the best measures to use, seeing as they only show the quality of the order compared to a random order. In this document retrieval system, the top-$k$ is the most important part of the returned documents. To measure the quality of this top list, we will calculate the Precision@k, Recall@k and Accuracy@k values. Formulas for these numbers are given as follows, using $R$ is the total relevant documents with $r \subseteq R$ documents in the top-$k$ and $t$ are the TN documents in that top-$k$:

$$P@k = \frac{r}{k} \qquad R@k = \frac{r}{R} \qquad A@k = \frac{r}{r + t}$$

## 3.1 Manual Labeling

First, we randomly choose 500 documents out of the complete data set. Then, 35 different queries are manually created based on the content of these documents. The queries are 'human real life' created such that the

use case is matching the reality. Some examples of the queries are: *"How to read and write to a file"* and *"Beautifulsoup web scrapers"*. Next to these queries, there are also two test queries to check if the algorithm works, namely the queries *"Python"* and *"C++"*.

Each document is deeply looked into and manually mapped to which query/queries should linked to this document.

## 3.2 Performance Scoring

And then comes the scoring phase. For all of these queries, we will determine the scores of all documents. This gives us two general information files.

The first one is a results file that states for each question how good it performed on which queries. Secondly, there is a relevance file where each query maps to a list of question ids. These ids represent the questions that were marked as relevant/correct document for the corresponding query.

A Hapax Legomenon (or a hapax for short) is a word or a concept that only occurs only once in a given context (see [5]). Within our context, we can state that a query could a hapax if there is only one document that uses it and will match. Or, in layman's terms, a query (or a set thereof) that is only used by a single question. These do not give an accurate representation of our dataset, so we made the queries such that it has at least 2 words. Similarly, we believe that a query will only match to only one document is not an accurate choice, so we removed these queries. As a result we only have queries that will match to at least 2 documents.

### 3.2.1 PR and ROC

For each manual created query, we can now determine the top-$k$ precision, recall and fallout (as is shown in the lecture slides), which allows us to create the Precision-Recall (PR) Curve and the ROC-Curve for each query.

Because this will yield a lot of curves, most of which without any significant features, we have decided to represent the PR and ROC as an upper bound (i.e. the curve that is generally higher than all values). Note that this may give a skewed view of our results, but it seemed the most doable option. We tried using (weighed) averages, minimals, differences..., but unfortunately all of these resulted in plots with strange artifacts. An example of such a plot is shown in figure 2.

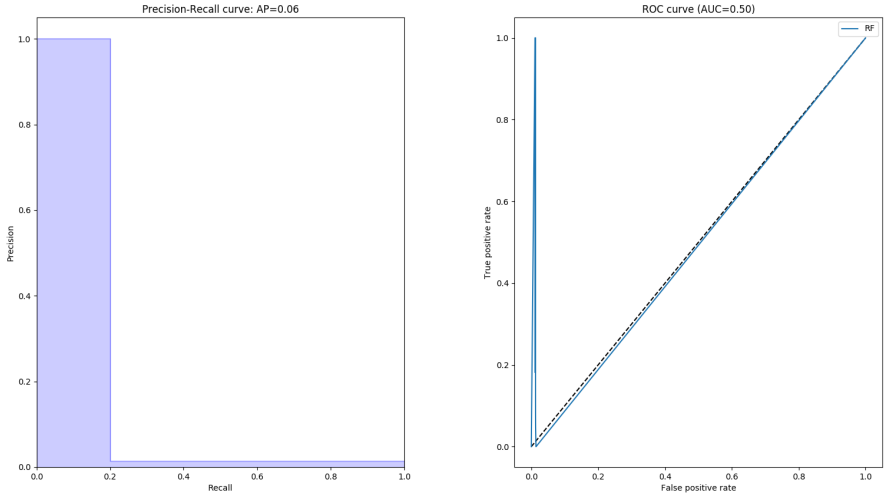



Figure 2: PR/ROC plots with artifacts due to maximization

### 3.3 Precision@k and Recall@k

For document retrieval, the top-$k$ returned documents are really interesting. Usually, a user types in the query and they only look at the first 10, 20 results (read: the top-$k$). So the top results are really important and should (ideally) all consist out of relevant documents. The precision@k is a way of measuring how good the top-$k$ is. It calculates the percentage of the relevant documents in the top listed documents.

Beside, the recall@k is also calculated: how many documents are there in the top-$k$ out of the relevant, correct documents. Unfortunately it doesn't give that much extra information for the performance of our system (similar for the accuracy@k), but it is calculated for its completeness.

#### 3.3.1 Initial Score

Using the `StandardAnalyzer` of Lucene[13], the scores are computed as described above. This gives us an idea how well Lucene initially performs.

Using the averaged PR- and ROC-curves to visualize the results, we can conclude from Figure 3, Lucene is doing a good job for the task at hand.

[13] Which does a lot behind the scenes, including (but not limited to) removing stopwords and lowercase transformations

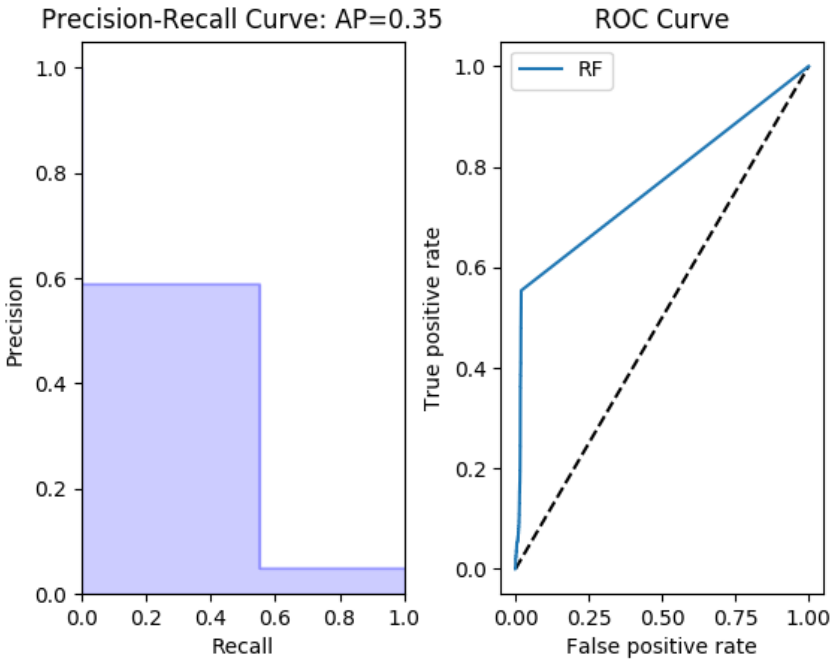



Figure 3: Results initial benchmark performance.

Next to these curves, the Precision@k and Recall@k are even more important to have an idea about the top results. Note: for the entire project, we have taken $k = 20$.

| | |
|---|---|
| Weighted Precision@20 | 0.327056277056277 |
| Weighted Recall@20 | 0.24085364440294335 |
| Weighted Accuracy@20 | 0.6367911543988225 |

Table 1: Results top-20 for the initial benchmark performance.

### 3.4 Scoring Models

To check whether the scoring models have any influence on the performance, the different `Similarity` measures have been used: `TFIDFSimilarity`[14] and `BooleanSimilarity`[15].

Different similarity measurements have an impact on the performance of the system. The results give an overview of the boolean similairty where the terms score is equal to their query boost. The other measurement is the Tf-Idf similiartity; this scores the documents based on the cosine similarity in a VSM as described in section 1.4.1. The Tf-Idf and the Boolean similarity have exactly the same curves as the initial benchmark (see figure 3 and the top-$k$ results in table 1). We have chosen to continue the project with the Tf-Idf similiarity measure.

## 4 Optimizations

As was clear from figure 3 and table 1, we can do better. This is why we will try to optimize our algorithms, in the hope of obtaining better results. In order to achieve this goal, we will reimplement the `StandardAnalyzer` as a basis for our `MyCustomAnalyzer` and work from there.

### 4.1 Removing Special Characters and Whitespace

As discussed in depth in section 2, we will transform our fields so all specialities are discarded. If we base ourselves on the `StandardAnalyzer` and on the transformations we want to apply, as per section 2, all that's left to do

[14] `org.apache.lucene.search.similarities.TFIDFSimilarity`
[15] `org.apache.lucene.search.similarities.BooleanSimilarity`

in order to have a valid transformation is to remove all non-alphanumerical characters and additional whitespace from our text.

This is done via adding a `PatternReplaceFilter`[16] and a `TrimFilter`[17] in our `Analyzer`.

In figure 4, you can find that there was no significant difference compared to the original plot (figure 3) and to the Precision@k value.

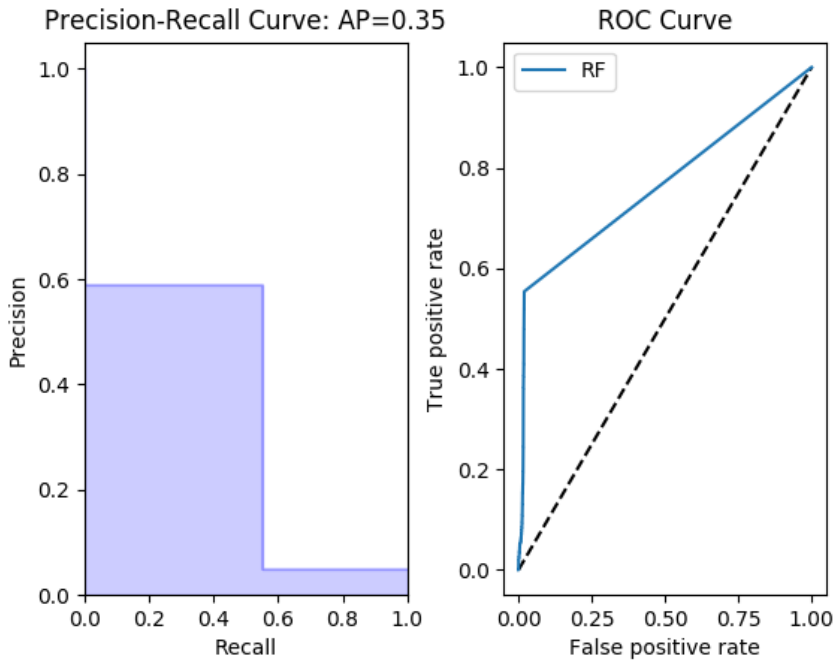



Figure 4: Results Benchmark Performance with Initial Filters.

| | |
|---|---|
| Weighted Precision@20 | 0.327056277056277 |
| Weighted Recall@20 | 0.24085364440294335 |
| Weighted Accuracy@20 | 0.6367911543988225 |

Table 2: Results top-20 with Initial Filters.

[16] `org.apache.lucene.analysis.pattern.PatternReplaceFilter`
[17] `org.apache.lucene.analysis.miscellaneous.TrimFilter`

### 4.1.1 Word Delimiter

The `StandardTokenizer` will be extended with a `WordDelimiterGraphFilter`[18]. This will split the words into multiple subwords and performs optional transformations on subword groups. [1]

For example, if someone wants to search for "`Macbook`", but the user types "`Mac Book`". The WordDelimiter will catch this query because the word "`Macbook`" will internally saved as {"`Mac`", "`Book`", "`Macbook`"}. This way, we can cope with different cases of compound words.

At first, it sounds like a good idea, but, unfortunately, it doesn't have a huge impact on the performance. It even results in the same PR- and ROC-cuves as Figure 4 and the same for Table 2! We believe this is because this kind of filter does not apply for our current set of queries.

### 4.1.2 Stemming

Next, stemming[19] has been added to the analyzer such that multiple variants of a word are mapped to the same word (i.e. plurals are mapped to its singular). There is a performance increase on this front, as you may be able to deduce from figure 5. (Yes, it looks once more like the very same plot, possibly because of the way we generate the plots.) More importantly there is a positive impact on the Precision@k, which is increased after applying the stemming (Table 3).

| | |
|---|---|
| Weighted Precision@20 | 0.3775088547815821 |
| Weighted Recall@20 | 0.22292070811981587 |
| Weighted Accuracy@20 | 0.6506418868487834 |

Table 3: Results top-20 Analyzer extended with Stemming.

### 4.1.3 $n$-grams

Now, let's see what happens if we divide the words into tuples of length $n$. This is called $n$-gram filtering[20]. We tried 7 different values for $n = \{2, 3, 4, 5, 6, 7, 8\}$.

For all $n$, the ROC curves are all different and some have a good opportunity to be extended in our custom analyzer. Especially $n = 8$ has a

[18] `org.apache.lucene.analysis.miscellaneous.WordDelimiterGraphFilter`
[19] `org.apache.lucene.analysis.PorterStemFilter`
[20] `org.apache.lucene.analysis.ngram.NGramTokenFilter`

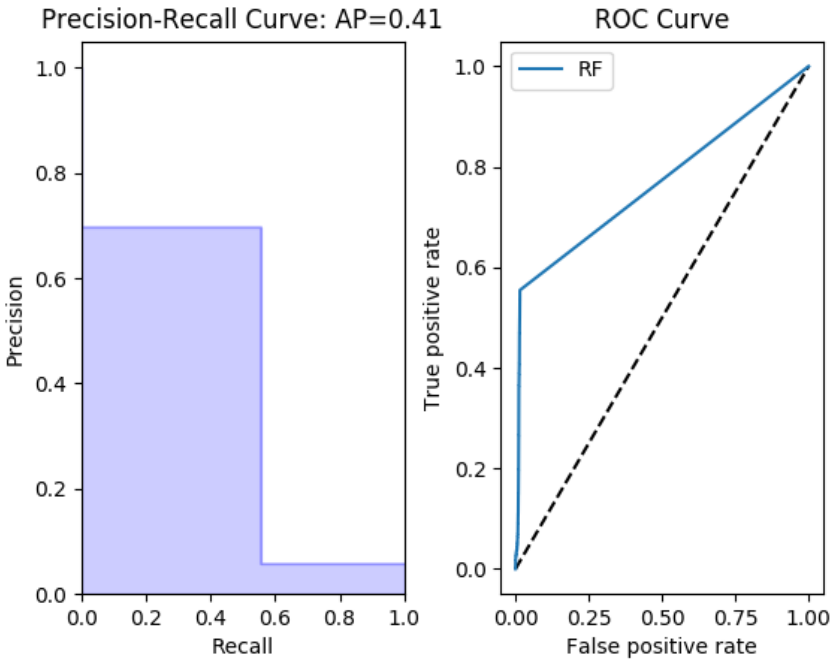



Figure 5: Results Analyzer extended with Stemming

potential ROC curve for all returned documents. Beside, the Precision@k has the conclusion - which is more important to be considered for the custom analyzer. For the top-$k$ results the Precision@k has increased a lot, namely to almost 0.42. Because of this huge improvement, we will include the $n$-grams (for $n = 8$) in our optimizations.

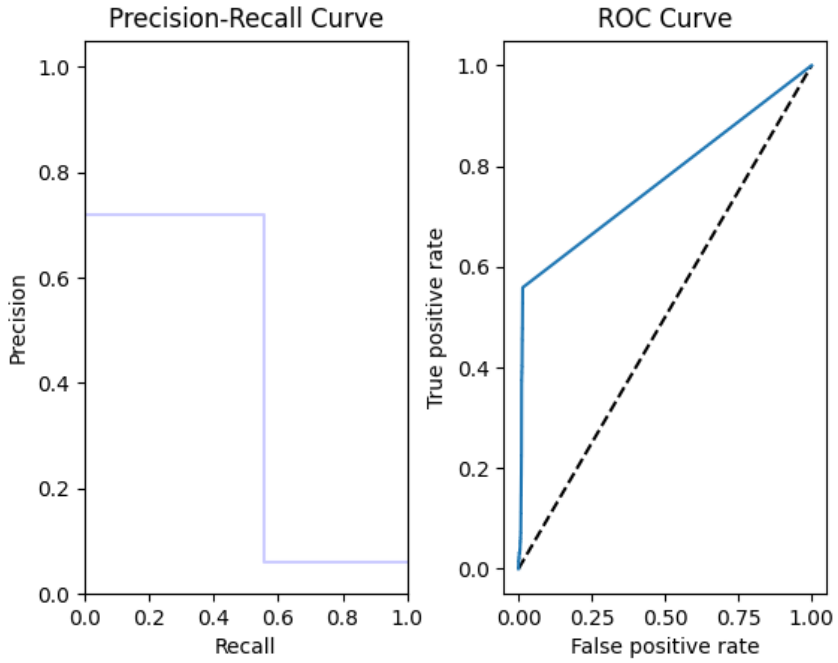



Figure 6: Results Analyzer extended with $n$-grams (for $n = 8$)

| | |
|---|---|
| Weighted Precision@20 | 0.417955060812204 |
| Weighted Recall@20 | 0.233950930066487 |
| Weighted Accuracy@20 | 0.680485720858392 |

Table 4: Results top-20 Analyzer extended with $n$-grams (for $n = 8$).

### 4.2 Manual Thesaurus-Based Query Expansion

A thesaurus is a list of synonyms. For each term in our dictionary, we may create such a list to associate the words with. Theoretically, this should

increase our general recall, but may decrease precision, especially when we're talking about homonyms and such.

As a last improvement, we create such a filter[21] and add it to the analyzer. This way, for each word we see, a set of synonyms will be associated with it, using a `SynonymMap`[22]. Use used an offline dictionary of thesaurus, [7], which is built with approximately 169 000 English words and is two-directional.

To our surprise there was an enormous decrease in performance (approximately 12.7% in terms of Precision@k), meaning that the synonyms get in the way of the analyzer and scores only worse. This is shown in figure 7 and table 5. That's why we decided to not extend the analyzer with the synonym filter.

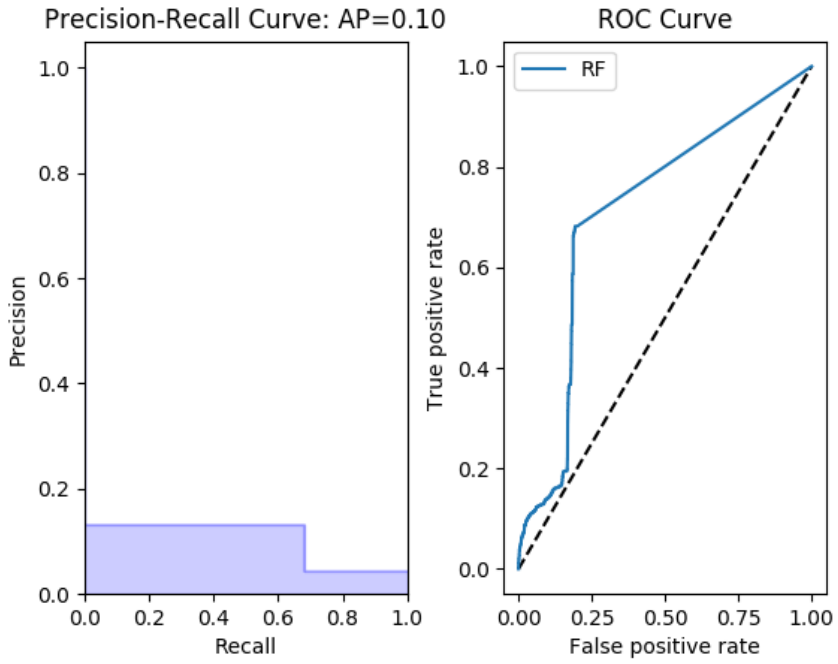



Figure 7: An overview of the extended Analyzer for manual thesaurus-based query expansion.

[21] `org.apache.lucene.analysis.synonym.SynonymFilter`
[22] `org.apache.lucene.analysis.synonym.SynonymMap`

| | |
|---|---|
| Weighted Precision@20 | 0.29145803456949 |
| Weighted Recall@20 | 0.41720735195784 |
| Weighted Accuracy@20 | 0.505901510762118 |

Table 5: Results top-20 Analyzer extended with manual thesaurus-based query expansion.

## 4.3 Final Results

A final result of our benchmarking study can be found in figure 6 and table 4, where everything comes together: lower case conversion, trimming, removing of non-alphanumeric values, word delimiters, stemming, and $n$-grams (for $n = 8$).

Note that manual thesaurus-based query expansion is not included in this result.

### 4.3.1 Summary Precision@k

The Precision@k values were decisive in creating the optimal custom analyzer. By adding the stemming, the results were better in the top-$k$. By combining it with the $n$-grams, the results were astonishing.

To give an idea about the impact, we have put all the values next to each other in a histogram (figure 8). It is clearly to see that 'C N7' and 'C N8' are the best scoring custom analyzers (focusing at the Precision@k).

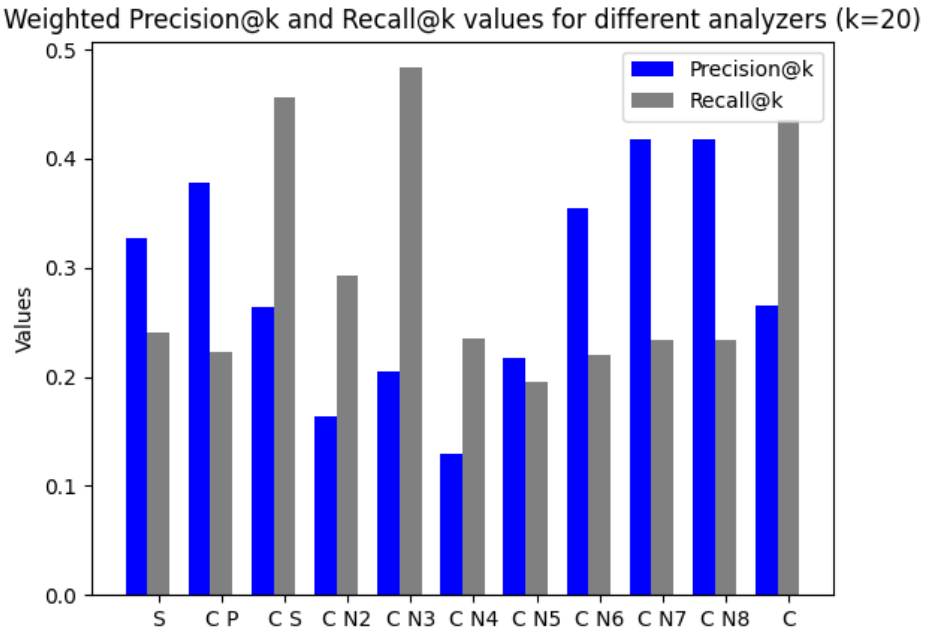



Figure 8: An overview of the impact to the custom analyzers of the Precision@k and Recall@k. Where the 'S' stands for Standard Analyzer, the 'C' for Custom, 'P' for adding Porter stemming (also trimming and word delimiter), 'S' for adding Synonyms, the 'N' for $n$-grams (this is including Porter, excl. synonyms) and the final 'C' is the total Custom with all the elements active.

# 5 Rocchio Algorithm

One possible optimization can come from the Rocchio algorithm for pseudo-relevance feedback. It's principle is simple: each document is located in an $n$-dimensional space, we search for a query (in the same space), mark some documents as "*relevant*" and change the angle of our query vector so it is located in the center of the cluster of relevant documents. Note that this method assumes all relevant documents are located in the same cluster.

To apply the algorithm, the following formula can be used to alter our original query $\overrightarrow{Q_o}$:

$$\overrightarrow{Q_n} = \alpha\overrightarrow{Q_o} + \beta\frac{1}{|D_R|}\sum_{\overrightarrow{Q_j}\in D_R}\overrightarrow{Q_j} + \gamma\frac{1}{|D_{NR}|}\sum_{\overrightarrow{Q_k}\in D_{NR}}\overrightarrow{Q_k}$$

In section 3, we described a way to easily and uniformly determine if a document was "*correct*" (read: *relevant*) or not. So we have a query $q$, a set of relevant documents $D_R$ and a set of irrelevant documents $D_{NR}$ (which can be obtained from the simple truth that $D_R + D_{NR} = D$, where $D$ represents the set of all documents). Because we remembered the frequency vectors (see section 1.3.3), we can easily transform all these variables into their corresponding vectors and apply the formula.

Unfortunately, while we tried to come up with a viable algorithm to compute $\overrightarrow{Q_n}$ from the information Lucene provided on the query, we failed in doing so. Instead, we pointed our attention to already-existing library for computing pseudo-relevance feedback [4] and its dependency [3]. The last activity in either of these repositories dates from 2017, hence a lot of functions and features follow an outdated Lucene version, compared to our project. Therefore, instead of adding these repositories as submodules, they were added by copying the files. This way, we could update the dependencies to work with our code. Additionally, we changed our build system from a single Java-command to Maven.

Exercising Rocchio for the `StandardAnalyzer` yields the results shown in figure 9. Compared to figure 3, we can see clearly that the AUC is increased, resulting in a better ROC curve. However, the Precision@k value is decreased by approximately 16.4% which isn't a good result. Therefore, the Rocchio algorithm combined with the standard analyzer performs worse than when using our custom analyzer.

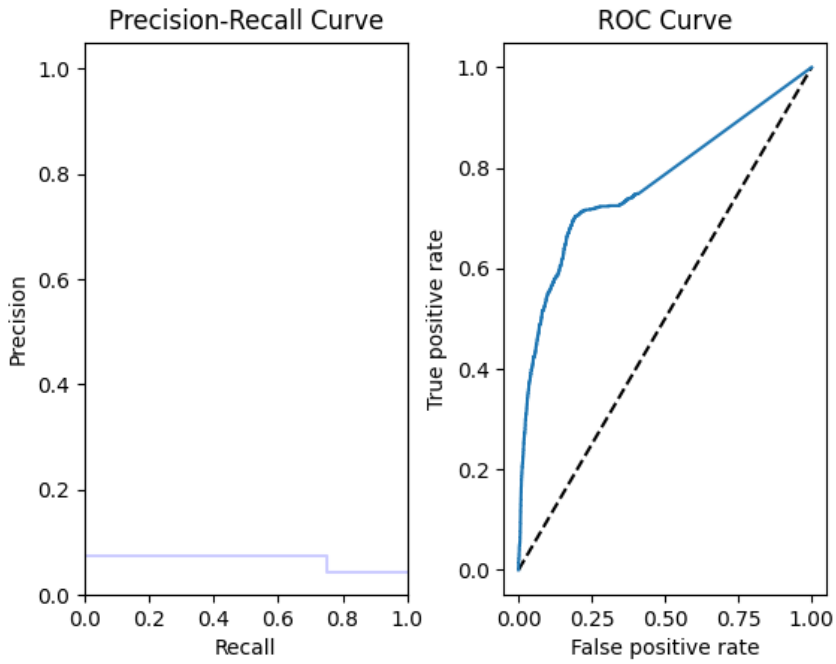


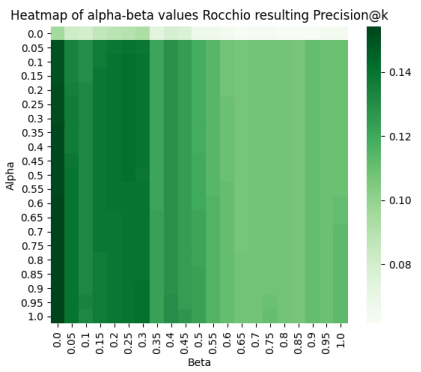

Figure 9: The standard analyzer combined with the Rocchio algorithm for pseudo-relevance feedback.

| | |
|---|---|
| Weighted Precision@20 | 0.2542857142857142 |
| Weighted Recall@20 | 0.6373501077545152 |
| Weighted Accuracy@20 | 0.0.10271136159409343 |

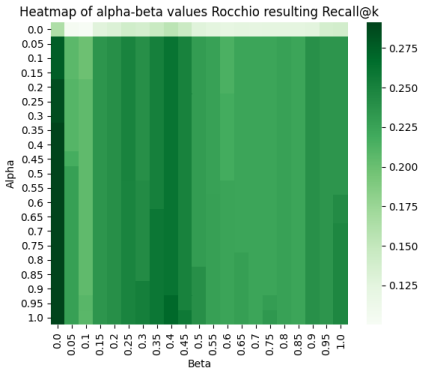
Table 6: Results top-20 standard analyzer combined with the Rocchio algorithm for pseudo-relevance feedback.

## 5.1 Optimizations

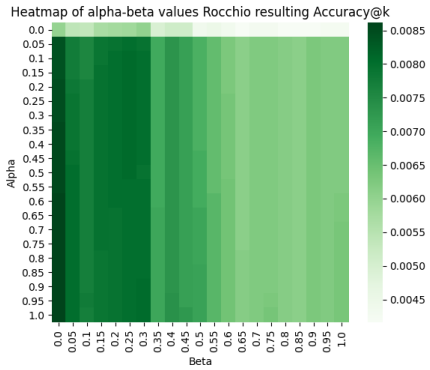
As described above, the Rocchio algorithm is parameterized with an $\alpha$, $\beta$ and $\gamma$ value. In the implemented algorithm of the used library [4], there are only 2 parameters for Rocchio ($\alpha$ and $\beta$) and 2 other parameters for the BM25-similarity measure. We have used a Gridsearch to find the optimal $\alpha$ and $\beta$. To visualize this, the precision@k, recall@k and accuracy@k values are showed in a heatmap for the different parameter values.




(a) Precision@k




(b) Recall@k




(c) Accuracy@k

Figure 10: Heatmap

We mentioned that for this use case - with the 500 manually labeled documents - the best parameters must be set to $\alpha = 1.0$ and $\beta = 0.0$. The relevance feedback algorithm isn't working in its ideal environment, because

of the small use case. Although, for a bigger use case - with lots of more documents - the Rocchio algorithm will perform much better. As a result, the $\beta$ parameter will also have an impact to the document retrieval (in contrast to our use case). Unfortunately, the optimal Rocchio parameters will also perform with an extreme worse Precision@k namely equal to 0.15428.

## References


[1] Apache Lucene Homepage. https://lucene.apache.org/. Accessed: 2019-12-18.

[2] Apache Lucene Wikipedia Page. https://en.wikipedia.org/wiki/Apache_Lucene. Accessed: 2019-12-18.

[3] GLIS ir-tools repository. https://github.com/uiucGSLIS/ir-tools. Accessed: 2020-06-05.

[4] gtsherman Rocchio repository. https://github.com/gtsherman/lucene. Accessed: 2020-06-05.

[5] Hapax Legomenon. https://en.wikipedia.org/wiki/Hapax_legomenon.

[6] Lucene Basic Concepts. http://www.lucenetutorial.com/basic-concepts.html. Accessed: 2019-12-18.

[7] Offline database of synonyms/thesaurus. https://github.com/zaibacu/thesaurus. Accessed: 2019-12-19.