Information Retrieval

Project Lucene
StackOverflow

M INF 2019-2020

January 20, 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 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 to allow our operating systems not to freak if the main folder was opened. The overall size of this data 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:

```
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

[1]From the website indicated in the assignment.

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

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

character encoding system that does not map straightforwardly on 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.

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 we will bypass the use of storing these files in a directory this way.

```
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 (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.

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

[4] This method has as a downside that it will not work on Windows systems.

[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*". [4]

1.3.3 Storing the Feature Vectors

Because we will need it later on (see section 3), we will also tell Lucene it needs to store the term vectors of all fields. This way, we can also use this in section 5 to manipulate the information from the vector space model to our advantage.

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, 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

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

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

Here, we would search for a query `q` and return at most `cnt` documents that match it, ordered with 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.

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[6]. Instead, we decided to use the builtin binary query system, which is tested and optimized for searching[7].

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`.

[6] We also discarded this idea because we did not believe this was intended with the assignment.

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

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 Lucene is by default case-sensitive and via transforming all and any data to lowercase, we bypass this strange condition Lucene forces upon its users. 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[8]. 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:

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

Note that this only applies for the query itself, as far as the documents (and their fields) are concerned, we would like to refer to sections 1.3 and 4.

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?

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

Let's say we have the query "`python database setup`". Even though Lucene will probably add implicit `AND`-gates, 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 (~) 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.
  On top of that, it is unclear from the documentation if this also applies to words with different lengths, which is another reason we don't want to do 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.
For instance, in a 5-letter word "`abcde`", wildcards can be located at the letter locations (except for the `a`), or between all letters, giving us $2^{4+4} = 2^{2(5-1)} = 2^8 = 256$ possibilities.

- 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.

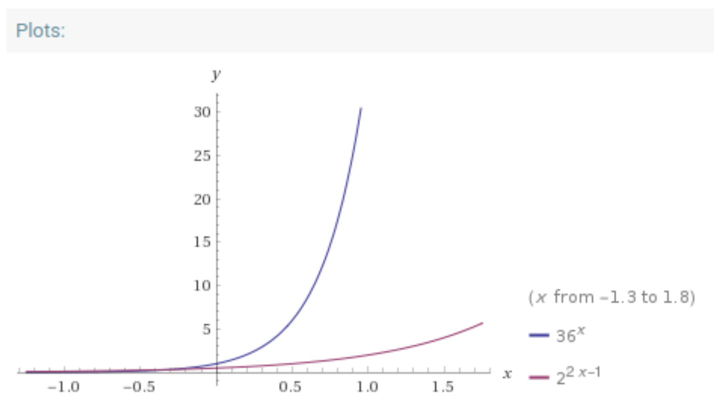

Figure 1: WolframAlpha-made plot of both $y = 36^x$ and $y = 2^{2x-1}$.

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[9]. For our pur-

[9] It's illogical that each word should match all of its possibilities. From "`bear`" to

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):

```java
// 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:

```java
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);
```

"`beard`", "`boar`" or "`beer`" are smaller changes than replacing it with "`glar`".

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:

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

3.1 Manual Labeling

Contrary to what it's name might suggest, we did not select a number of documents which we labeled by hand for a specific query. This seemed to bring us too far away from the project scope and therefore we focused on finding an acceptable algorithm to solve this issue. We were mainly inspired by this part of the assignment:

> *One acceptable approach is to use the titles of the questions as the query, and only index the remaining parts of the documents. In this way a ground truth can be generated rather easily.*

We tried different methods for achieving our goal and, at long last, we stranded on the method described below.

First things first, we have a system that can determine the score of all documents, given a set of queries. Unfortunately, this system alone is not enough in determining the amount of True Positives (TP), False Negatives (FN), False Positives (FP) and True Negatives (TN). We need a way to consistently label documents as either "*correct*" or "*incorrect*". And this can be done by exploiting the use of *tags*.

3.1.1 Tags as Queries

Each question in our dataset has a group of *tags* associated to it. These *tags* generally mark the topics or problem domains to which the questions apply, therefore we can state that, if we search for any group of tags, all documents that have these tags can be marked as "*correct*".

For instance, a question that concerns "`efficient string comparison in python`", with tags "`string-comparison`" and "`python`" must be a document that is returned when looking for "`python`", "`string-comparison`", or "`python string-comparison`"[10].

Now, how do we go about doing this? On indexing time, we will keep some sort of a query cache[11]. This cache is a list of strings that we can use as queries for benchmark purposes. For every question we have in our dataset, we create a new entry in this cache, concerning all unique possible combinations of the set of *tags* for this question. For instance, the tags "`python`", "`c`" and "`java`" become the entries {"`python`", "`c`", "`java`", "`c python`", "`java python`", "`c java`", "`c java python`"} (notice the alphabetical ordering).

Because we know there are a lot of queries that will be created with this method, we will not concern ourselves with tags that contain anything but alphabetical characters and discard those entries entirely.

Of course, this also implies that we will remove the *tags* from the fields we search on, otherwise we will always retrieve a perfect score, making an entire benchmark study useless.

3.2 Performance Scoring

And then comes the scoring phase. For all of these queries, we will determine both the scores and its relevance of all documents as discussed above. 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 marks "*correct*" 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 [3]). Within our context, we can state that a query is a hapax if there is only one document that uses it. Or, in layman's terms, a *tag* (or a set thereof) that is only used by a single question. These do not give an accurate representation of our dataset, so we remove them on the whole. Similarly, we believe that a *tag* that's only used by two questions does not provide enough information for our data.

From the 20 710 different *tag* combinations that were used as queries, we will only look at the 2 099 queries that were able to mark more than two questions as being "*correct*".

[10] Please denote that "`string-comparison`" is used as a tag for readability of this report, but internally, in our code, this is one of the tags we won't process (see further).

[11] It is slightly more efficient to do this at indexing time, but it is perfectly doable to do this afterwards, in the actual performance checking.

3.2.1 PR and ROC

For each query in this set of 2 099 unique *tag* combinations, 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 overall information as an average of all of these curves.

3.2.2 Initial Score

Using the `StandardAnalyzer` of Lucene[12], 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 2, Lucene is not doing such a great job for the task at hand.

Do note that this graph is generated from all 20 710 queries to give a general picture, but that, for time constraints, we reduced this list to the first 1 000 queries in our further experimentation.

3.3 Scoring Models

To check whether the scoring models have any influence on the performance, the different `Similarity` measures have been used: the default similarity, `TFIDFSimilarity`[13] and `BooleanSimilarity`[14]. Remarkably, these measurements don't score/find any new documents, but rather optimize the scores of any documents that were already found with other methods.

4 Optimizations

As was clear from figure 2, 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.

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

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

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

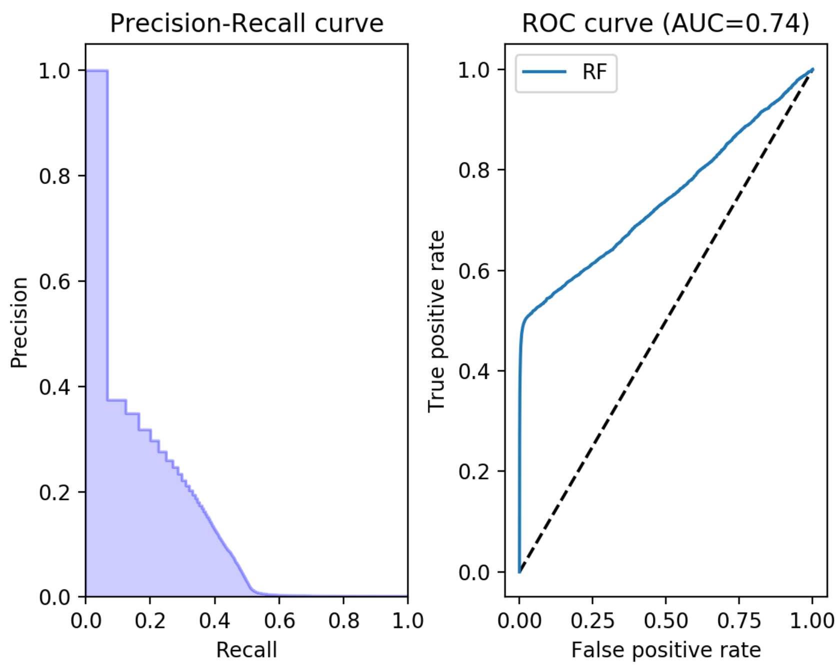

Figure 2: Results initial Benchmark Performance

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 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`[15] and a `TrimFilter`[16] in our `Analyzer`.

In figure 3, you can find the optimizations that came from the application of these filters. As you can see, there is not a lot of performance increase to be had, compared to the original graphs.

[15] `org.apache.lucene.analysis.pattern.PatternReplaceFilter`

[16] `org.apache.lucene.analysis.miscellaneous.TrimFilter`

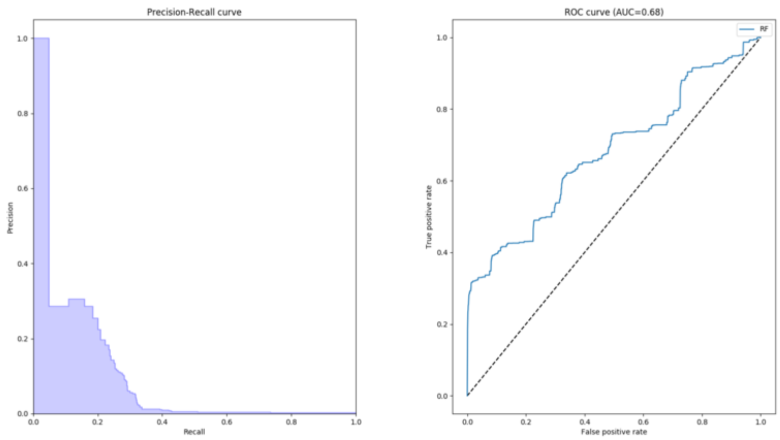

Figure 3: Results Benchmark Performance with Initial Filters

4.1.1 Word Delimiter

The `StandardTokenizer` will be extended with a `WordDelimiterGraphFilter`[17]. 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 3!

4.1.2 Stemming

Next, stemming[18] 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). This already has an impact on the performance, there is an increase of 3% that the analyzer will be able to distinguish between positive and negative class.

[17] `org.apache.lucene.analysis.miscellaneous.WordDelimiterGraphFilter`
[18] `org.apache.lucene.analysis.PorterStemFilter`

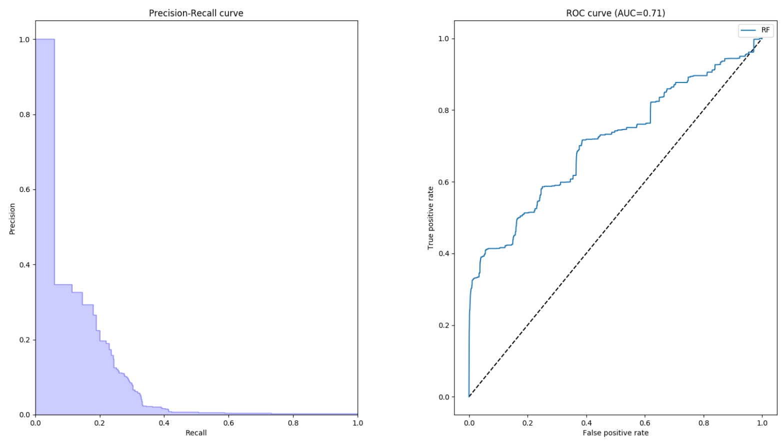

Figure 4: Results Analyzer extended with Stemming

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[19]. We tried 6 different values for $n = \{2, 3, 5, 6, 8\}$.

For all n, the results are terrible (see figures 5, 6, 7, 8, 9). We have no idea if this is due to how we've built our queries, or if it comes from the way we're using the data. Our best guess would be that n-grams are technically only interesting to use if you were to allow wildcards and the queries on which we test don't use them.

The empty plots you see are, in fact, PR-curves that have a precision close to 0, no matter the value for the recall.

It is clear we get better results without the usage of n-grams, which is why we will not include them in our optimizations.

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.

[19] `org.apache.lucene.analysis.ngram.NGramTokenFilter`

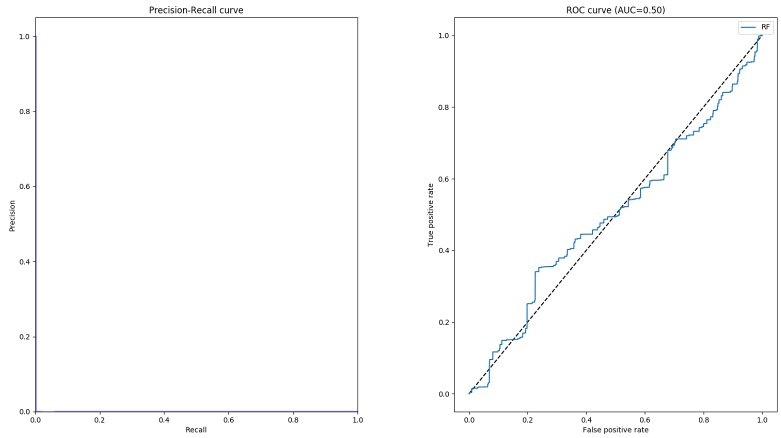

Figure 5: An overview of the extended Analyzer for n-gram filtering, where $n = 2$. $AUC = 0.5$

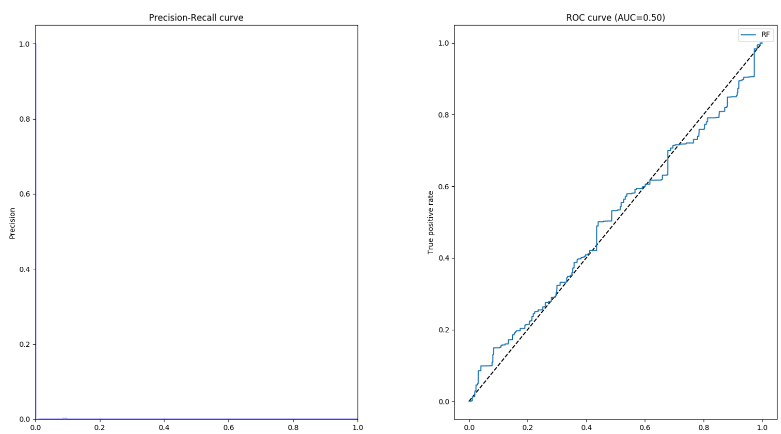

Figure 6: An overview of the extended Analyzer for n-gram filtering, where $n = 3$. $AUC = 0.5$

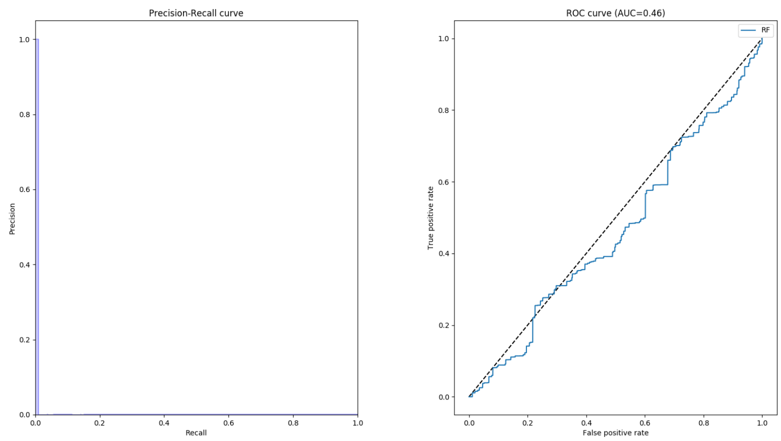

Figure 7: An overview of the extended Analyzer for n-gram filtering, where $n = 5$. $AUC = 0.46$

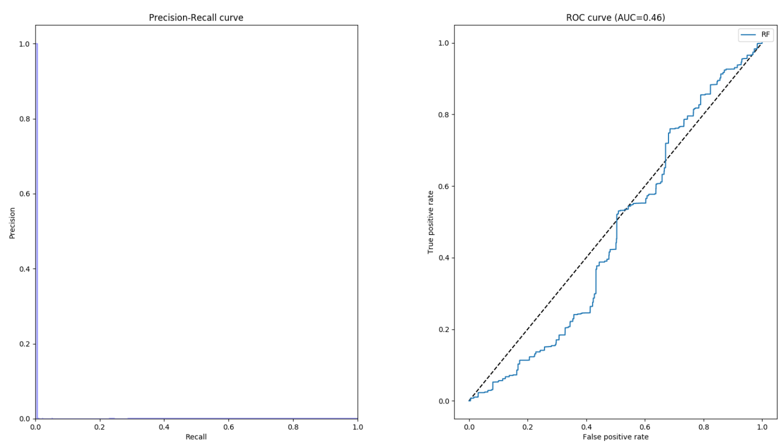

Figure 8: An overview of the extended Analyzer for n-gram filtering, where $n = 6$. $AUC = 0.46$

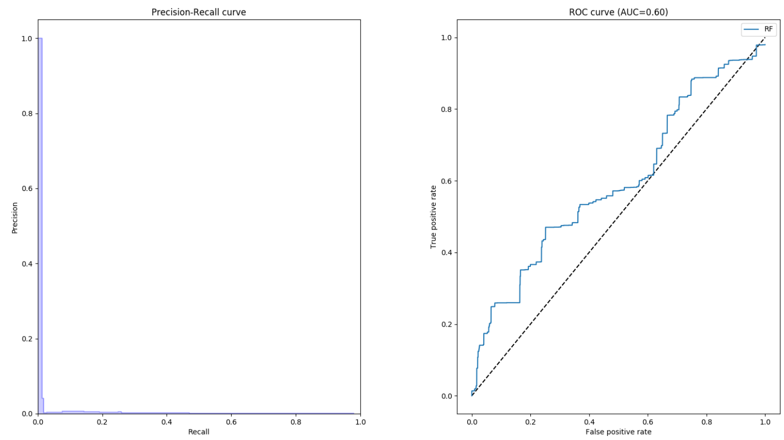

Figure 9: An overview of the extended Analyzer for n-gram filtering, where $n = 8$. $AUC = 0.60$

As a last improvement, we create such a filter[20] 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`[21]. Use used an offline dictionary of thesaurus, [5], which is built with approximately 169,000 English words.

Unfortunately, the performance results give a terrible result (see also figure 10).

4.3 Final Results

A final result of our benchmarking study can be found in figure 11, where everything comes together: lower case conversion, trimming, removing of non-alphanumeric values, word delimiters, stemming, and thesaurus-based manual query-expansion.

Note that n-grams are not included in this result.

5 Rocchio Algorithm

Next, we wanted to apply the Rocchio algorithm for pseudo-relevance feedback. It's principle is simple: we search for a query, mark some documents

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

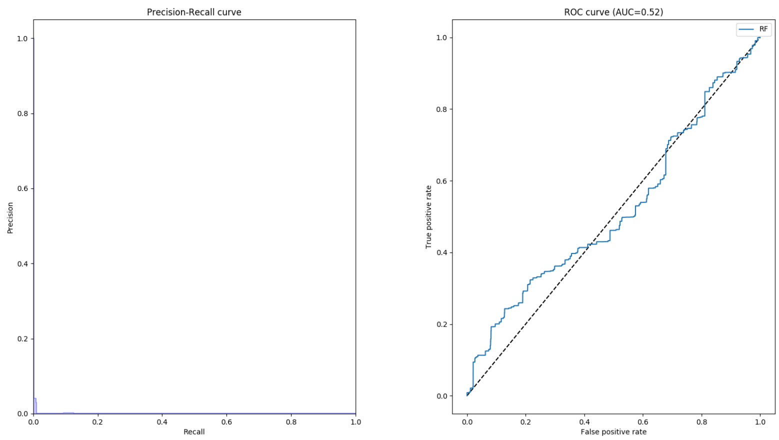

Figure 10: An overview of the extended Analyzer for manual thesaurus-based query expansion. $AUC = 0.52$

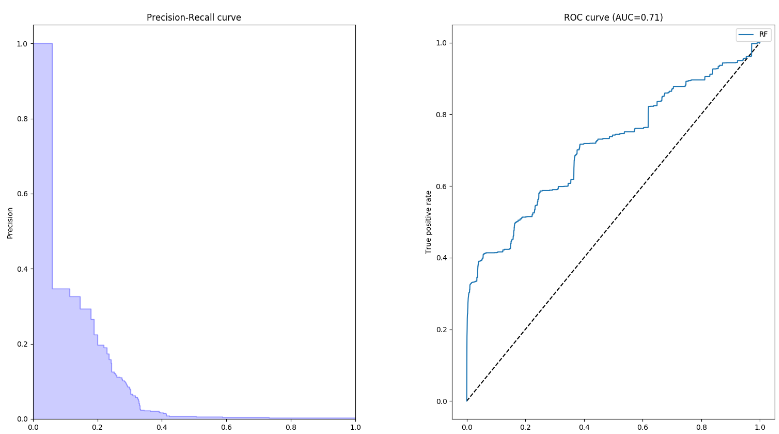

Figure 11: Results Benchmarking with word delimiters and stemming.

as "*relevant*" and change the angle of our query vector so it is located in the center of the cluster of relevant documents[22].

Unfortunately, we hit a brick wall. And we hit it hard. In theory, there

[22] Note that this method assumes all relevant documents are located in the same cluster.

is no problem. We apply the Rocchio formula as given below to our original query Q_o:

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

Let's ignore the scale factors α, β and γ for now. 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.

As you might recall, we described in the beginning of section 2 that Lucene does not work nicely with manipulating the VSM and searching by queries. Yet, we decided not to create a custom system for searching the documents, due to the sheer size of our data. This decision sort of came back to haunt us.

Because we did not develop such a system and thus do not have an optimized method for searching (and sorting) our documents, we have to map our custom feature vectors onto a Lucene boolean query string. Lucene's query syntax (found in [1]) provides a way to add a *boost* to each term in the query. Hence, for each term in our resulting query vector $\overrightarrow{Q_n}$, we give it a boost of its absolute value and negate it if the original value in $\overrightarrow{Q_n}$ was negative.

For instance, if $\overrightarrow{Q_n}$ states that the term "`python`" has a score of 5.6 and "`abstract`" has a score of -9.123, our resulting query string would be "`python^5.6 -abstract^9.123`".

Now, for each word in our vocabulary (which counts approximately 20 000 words), we have a term in our query string[23]. Also, Lucene cannot handle boolean queries that count over 1 024 words (by default), so having almost 20 times that size is difficult.

We can solve this issue by saying that Lucene must ignore the maximal clause count as follows:

```
BooleanQuery.setMaxClauseCount(Integer.MAX_VALUE);
```

[23] Since we drop all terms that would have a *boost* of 0, this is slightly less than our entire vocabulary, but the difference is nullable.

In hindsight, Lucene put that number by default to 1 024 for a good reason. As soon as this value is exceeded, it takes notably longer[24] to execute a simple search, so this is not the solution we were aiming for.

On top of that, notice how we must apply a mathematical formula to vectors of size 20 000. This formula takes an incredibly long time to compute, causing our benchmarking to be estimated to take 80 hours to run (if we were to limit the query generation to a mere 500 terms, which is a massive oversight in itself). And this is even ignoring the fact that we wanted to test it out for different values of α, β and γ.

Long story short, the way we have approached Rocchio is not the way to go. An optimization would be to set both β and γ to 0, but then we technically don't do anything. Another solution would be to limit our dataset to very few documents, but here the issues remain the same (with still an estimate of a couple of days).

Due to time constrains and the lacking documentation of Lucene on this front, we believe Rocchio (or other kinds of pseudo-relevance feedback) not to be possible with the current setup of our system.

[24] In fact, the process was sooner killed, due to "hanging" than it finishing for a single query.

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] Hapax legomenon. https://en.wikipedia.org/wiki/Hapax_legomenon.

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

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