Information Retrieval

Project Lucene
StackOverflow

M INF 2019-2020

December 20, 2019

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. [1]

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 [2].

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 via building `TokenStreams`[2]. More theoretically, it describes a policy on how items are extracted from text.

We've pointed our attention towards the `StandardAnalyzer`[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();
```

[1] From the website indicated in the assignment.

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

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

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, 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 did not freak out. Here is where we discovered a weird problem on MacOS (compared to Ubuntu LTS 18.04, that is). On Mac, the `File` class actually opens a 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);
```

For the purposes of the assignment, we indexed the fields *name* (i.e. the filename), *title*, *body*, *tags* and *answers*. Before adding these last four, they are transformed in such a way that all special characters and capitalization are removed (see also section 2). Additionally, the *answers* are all joined together, separated with two newline characters (`\n`).

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

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

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*". [3]

1.3.3 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. [2] 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:

```
// 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 [2], 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 me to emphasise that final sentence, or more specifically, that final clause. Because our solution requires running a lot of different queries on two different questions (see also section 3), we *need* to use the above-listed `searcher.search`-function. The `searcher.explain`-function that, as it is described on the docs, should allow a specific document to be scored for a query, actually changes the index, causing its results to be useless in our use-case[6].

All issues aside, we score all documents, but only remember those that are important in the context we've setup in our benchmark study (section 3).

[6] Additionally, one could argue it is quite an inefficient operation, seeing as it *does* score the entire dataset and only yields a single document thereof.

1.4.1 Similarity Scoring

From [2], 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

Using strings as normal queries in Lucene is quite finicky and obnoxious. This is why 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. This way, we can easily label some documents (see section 3). 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 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 1.3).

Finally, there is the alphanumerisation phase[7]. Here, we will remove all special characters from our queries, seeing as they will only cause issues

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

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 `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 ]", "")
```

2.2 String Separation

As you might have guessed from the previous section, we have only removed all strings from the beginning and the end of our string. All spaces in the middle are left behind and for good reason. 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. This not only allows words to be in a different word order, but also allows for words not to be next to one another. This method works wonders for Google, so why wouldn't it be in 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 is often 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 to be too precise and unable to capture all typos.

- 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 may not 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 an n-letter word, there are 2^{2n-1} possibilities for adding wildcards and even then, not all options are covered.

- Generate all possible permutations of 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 there are always more possibilities for this option, no matter the size of n.

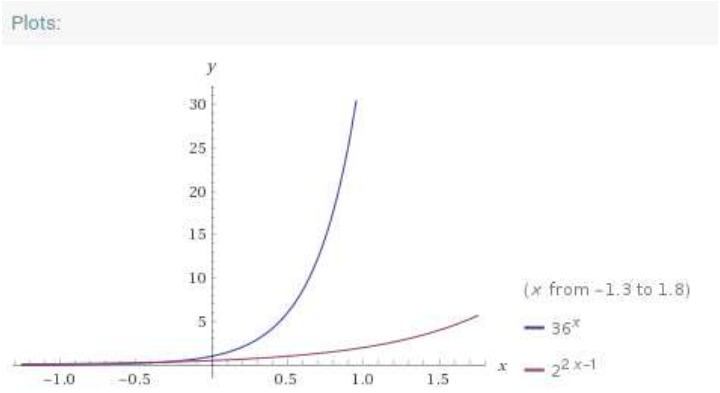

Figure 1: WolframAlpha-made plot of both $y = 36^x$ and $y = 2^{2x-1}$.

- 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. For our purposes, 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):

```
1 // Did you mean? https://www.javacodegeeks.com/2010/05/did-you-
      mean-feature-lucene-spell.html
2 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[8], Recall[9], True9[0] and False Positive Rates[10] need to be computed, which will be realized by *manual labeling*.

[8] Precision $= \frac{TP}{TP+FP}$
[9] Recall $=$ TPR $= \frac{TP}{TP+FN}$
[10] FPR $= \frac{FP}{FP+TN}$

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. Instead we allowed us to be inspired by the following 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.*

First of all, out of all the documents, 9999 documents have been randomly selected[11]. These documents are indexed by Lucene (see section 1.3), let's call these documents $D = \{D_1, D_2, \ldots, D_{9999}\}$. The titles of these documents are used to build the queries Q_i, where Q_i is the title of D_i. We know that each document must (at least) find itself and therefore we label D_i for Q_i to be `true` (indicated with **T**), for all i (see table 1). We've called the comparing a score against this data the "*self-scoring*" of a document D_i.

Now, we have a so-called *positive space* (PS), which can be used as a point of reference. Unfortunately, this does not give us any information on the amount of False Negatives (FN) our algorithm will yield. We need a way to unambiguously indicate that all D_i will score `false` (indicated with **F**) for a query Q_{10000}. Additionally, we want both our *positive space* and our *negative space* (NS) to be approximately the same in size, so we can make acceptable estimates.

We solve this issue via creating a dummy file such that the title of this file will never match with all other documents. This dummy file (called `question_NS.xml`) has been generated by filling the *body* and *answers* with a gibberish sample text[12]. Now, as far as the the *title* is concerned, we cannot use some other gibberisch for the same goal. If we did, we would not be able to match this file's contents to its title. We took three words from the *body* and used them as a *title*. Furthermore, for its *tags*, we did the same, which made sure this document would be equivalent to any StackOverflow document. Let's call this document D_{10000} and the title is this document will be query Q_{10000} (table 2)[13]. Because we use a dummy document, we refer to this part as the "*dummy-scoring*".

Obviously, in the scoring algorithm, we don't look at the *title*s of the documents.

[11] To allow for repetition, we've set the seed of the random number generator to 420.
[12] We used [4] for these samples.
[13] Please note that our NS is slightly larger in comparison to our PS

	D_1	D_2	$\dots$	D_{9999}
Q_1	**T**			
Q_2		**T**		
$\vdots$			$\ddots$	
Q_{9999}				**T**

Table 1: Manual labelling for titles as query

	D_1	D_2	$\dots$	D_{10000}	D_{10000}
Q_{10000}	**F**	**F**	$\dots$	**F**	**T**

Table 2: Manual labelling for dummy title as query

3.2 Scoring

Using the `StandardAnalyzer` of Lucene[14], the self-scoring and the dummy-scoring methods are performed. This gives us an idea how well Lucene initially performs. The PR- and ROC-curve are used to visualize the results. We can conclude from Figure 2, Lucene is doing an okay job for the task at hand. From our PR-curve we can see that, purely based on the positives (and their prediction) we're not doing too good. The ROC curve shows that Lucene is almost as good as a Random Classifier.

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. To do that, we will first do some custom optimizations to get better standardized results and afterwards we will take a look at some optimization algorithms.

4.1 Different PS/NS Ratios

One of the easiest optimizations to do is tweaking the PS/NS ratio. At this current point in time, we have approximately a 50/50 split, but this assumes our overall data is also divided this way. I.e. we assume that for a query Q_i, we can find 50% of all documents (in our entire dataset).

Obviously, this is a major assumption, seeing as we're more likely to find less documents with any given query. We will decrease the size of our positive space and redraw our curves to get a better view of how our data is divided.

In figure 3, there is an overview of four differently split datasets, respectively 10/90, 25/75, 75/25 and 90/10. From these graphs it is clear that the

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

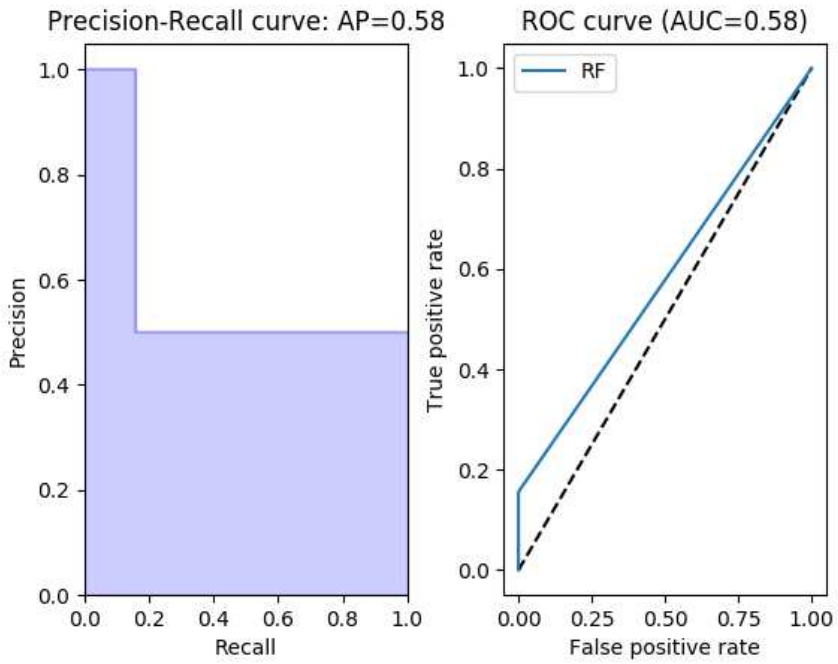

Figure 2: Results initial Benchmark Performance

data ratio does not matter for our scoring to be accurate[15].

4.2 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`[16] and `BooleanSimilarity`[17]. 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.

[15]The difference in PR-curve can be explained quite easily if you notice that it corresponds to the exact percentage of the PS used.

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

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

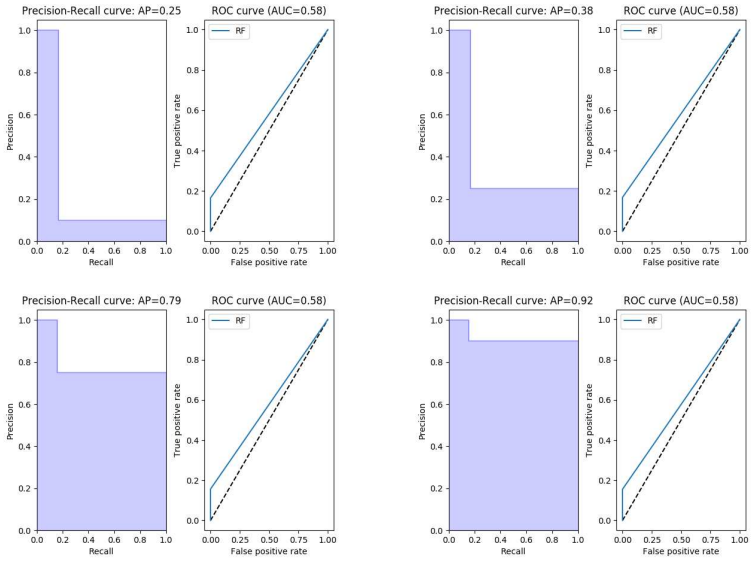

Figure 3: An overview of the different PR- and ROC-curves, resp. for 10/90, 25/75, 75/25 and 90/10 PS/NS Ratios.

4.3 Custom Analyzer

To increase the performance, the analyzer will be adapted in such a way that different aspects will be taken into account. In this section, the different approaches will be analyzed.

4.3.1 Word Delimiter

The `StandardAnalyzer` will be extended with a `WordDelimiterGraphFilter`[18]. This will split the words into multiple subwords and performs optional transformations on subword groups. [2]

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 2.

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

4.3.2 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). This already has an impact on the performance, there is an increase of 5% chance that the analyzer will be able to distinguish between positive and negative class.

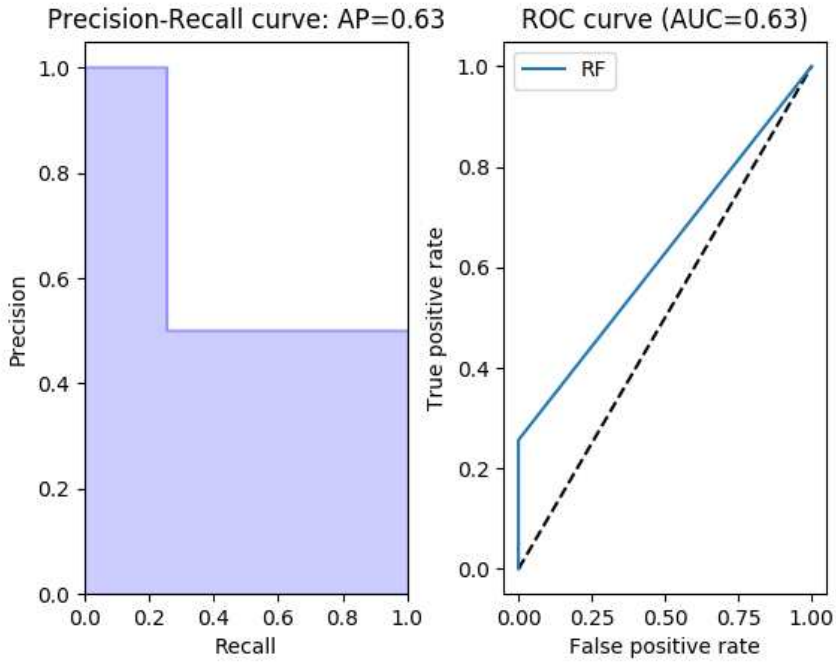

Figure 4: Results Analyzer extended with Stemming

4.3.3 n-grams

Then, the words will be divided into tuples of length n, this is called the n-gram filtering[20]. We tried 6 different values for $n = \{2, 3, 4, 5, 6, 8\}$. The performance results for $3 \leq n \leq 6$ are very good with a AP and AUC value of at least 0.78. This is a huge increase of the performance. See also figure 5.

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

The only disadvantage is that it will influence the run time (see table 3). Nevertheless, $n = 4$ yields the fastest results, therefore we've chosen this as the standard value.

Condition	**Run Time** in *mm:ss,hh*
$n = 2$	49:26,44
$n = 3$	11:08,89
$n = 4$	06:46,16
$n = 5$	09:49,41
$n = 6$	11:47,20
$n = 8$	11:51,95
$n = 4$, with synonyms (sec 4.4)	1:33:41,80

Table 3: Different run times when using n-grams, on a MacBook Pro Retina (2.5 GHz Quad-Core Intel Core i7).

4.4 Synonyms

The last improvement is by adding the synonyms[21] to the analyzer. The `SynonymMap` will be built with approximately 169,000 english words and their synonyms [5]. The performance results are amazingly good (see also figure 6).

The duration of this method, combined with n-grams (where $n = 4$) is also denoted in table 3.

4.5 Custom Analyzer

All these components are combined together in our `CustomAnalyzer`, graphically explained in figure 7. Conceptually, we do the following. First, we apply a `LowerCaseFilter` before adding a `StopFilter` of all English words[22] (similarly to the `StandardAnalyzer`). Next, we apply the word delimiter as described in section 4.3.1. We follow up with the stemming (see section 4.3.2) and a `SynonymFilter` (section 4.4). Finally, we apply the n-grams (with $n = 4$) as discussed in section 4.3.3.

The overall performance of our `CustomAnalyzer` is shown in figure 6.

[21] `org.apache.lucene.analysis.synonym.SynonymFilter`
[22] `org.apache.lucene.analysis.en.EnglishAnalyzer.ENGLISH_STOP_WORD_SET`

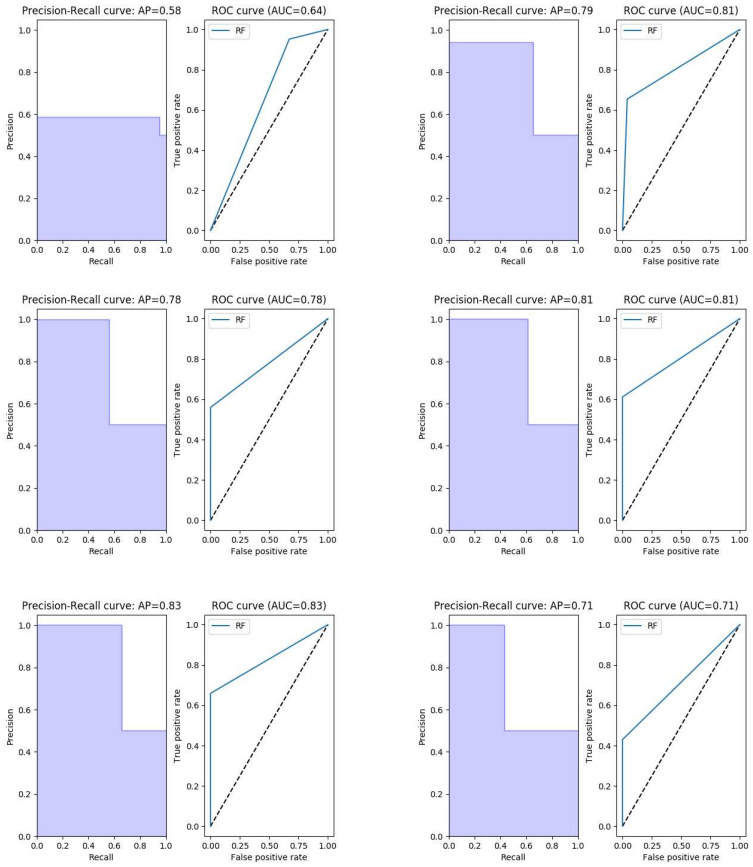

Figure 5: An overview of the extended Analyzer with different values for n for n-gram filtering. From left to right, top to bottom the values for n are 2, 3, 4, 5, 6 and 8.

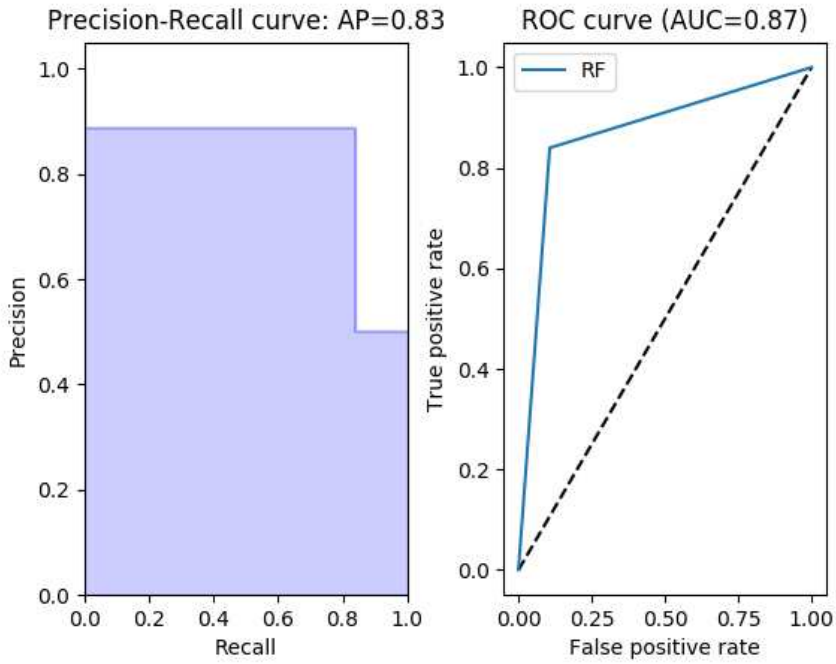

Figure 6: Results Analyzer extended with synonyms

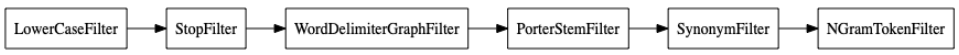

Figure 7: Custom Analyzer

References

[1] "Apache Lucene Wikipedia Page," https://en.wikipedia.org/wiki/Apache_Lucene, accessed: 2019-12-18.

[2] "Apache Lucene Homepage," https://lucene.apache.org/, accessed: 2019-12-18.

[3] "Lucene Basic Concepts," http://www.lucenetutorial.com/basic-concepts.html, accessed: 2019-12-18.

[4] "Random Gibberish Generator," http://www.weirdhat.com/gibberish.php, accessed: 2019-12-18.

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