Assignment 3

Filipe Pires [85122], João Alegria [85048]

Information Retrieval
Department of Electronics, Telecommunications and Informatics
University of Aveiro

December 11, 2019

1 Introduction

This report was written for the discipline of 'Information Retrieval' and describes the implementation and evaluation of a ranked retrieval method that uses the indexes created with the solutions developed for the previous assignments.

We include the correction of design flaws of the delivery done prior to this one and the updates applied both to the text corpus indexation and to our class diagram. We also provide the instructions on how to run our code.

Along with the description of the solution, we also present the results of our calculations to evaluate the solution and determine its efficiency according to the metrics proposed for this last assignment (*1*). All code and documentation is present in our public GitHub project at `https://github.com/joao-alegria/RI`.

2 Re-Indexing the Corpus

In order to make query searches flexible, it was proposed to us the reindexation of the text corpus considering not only the document titles but also their abstracts. This turned out to be quite challenging due to its computational weight, as the abstracts were considerably larger than the titles. The initial index occupied about 500Mb in disk, whereas the reindexation turned out to be over 3Gb large.

There were 2 approaches when adding the abstract processing to our indexing process. The first one would consist in dealing with titles and abstracts separately, creating independent index entries for both. This approach has the advantage of simplifying the process of verification of appearance of a given term in titles, abstracts or both, since they would be stored in different structures. It has however the clear disadvantage of requiring far more disk space to store all the information and introducing redundancy and unnecessary repetitions.

With this in mind, we went for another strategy, consisting of concatenating both fields and processing the resulting text as a whole. This approach's advantage and disadvantage are inverted when compared to the previous, since the redundancy no longer exists but it becomes impossible to distinguish the field a term appears at. Our choice was based on the analysis of the requirements for the ranked retriever we wanted to implement, as there seemed to be no need to keep the notion of where each term appeared at in each document. This also made the code adaptations to the indexing pipeline more straightforward and, as we previously mentioned, the final size in disk of the index smaller. The execution time of this pipeline thankfully did not evolve as fast as the disk space, passing from 20 minutes without the abstract to approximately 1 hour and 20 minutes with it.

3 Ranked Retrieval of Relevant Documents

With an entirely functional index creator and a folder of generated indexes from the corpus of text documents, it was now time to develop a program capable of interpreting queries and returning the index entries most relevant to them. In this chapter we describe our implementation of a query results ranked retriever - that we called `Searcher`. We also explain how we prepared it for memory limitations and present the updates done to our class diagram.

3.1 Document Ranked Scores

The file `Searcher.py` contains an abstract class called `Searcher` that serves as a template for the implementations of results retriever classes. For our purposes, we developed `IndexSearcher`, a class that extends from `Searcher` and is capable of selecting which index files are required to answer a given query, assigning scores to indexed documents and returning those considered most relevant to the query according to these scores.

The step of determining which index files will be needed is done by *retrieveRequiredFiles()* and is basically a comparison of each query term with the name of each index file. As index files are named after the first and last terms it contains (separated by an underscore) and as the entire index is alphabetically ordered, the function simply determines where each term will be present and returns those index files. It is needless to say that, in order for the retrieval process to not be compromised, the same tokenizer used on the `CreateIndex.py` must be used now. A special case occurs when the necessary information is cached, i.e. when the results are already present in memory. This memory management will be detailed in section 3.3.

Calculating each document's score regarding a query is not as easy to explain. The function *calculateScores()* is the one responsible for obtaining the relevant document IDs and the respective score values. To calculate the scores, it's necessary to iterate over all the selected index files and identify the lines were the query tokens appear, so that we have access to the documents where that specific term occurred. As an execution time and memory consumption reduction mechanism, we implemented a champion list strategy to decrease our search space, meaning that we only consider the first `S` documents (`S` can be defined by the user), which is made easy by the fact that previously with our indexing process we already order the posting list (list of documents were a term appeared) by the tf-idf weight.

We then estimate the relevant documents, assuming that having at least one term of the query is enough but also considering that not all documents have the same relevance to the user because a document that has all the query terms may be more relevant that others with fewer.

$$Score_d = \sum_{t=0}^{nT} W_{qt} * W_{dt} \tag{1}$$

$$W_{qt} = 1 + \log(tf) * idf => Simplifying => W_{qt} = 1 * idf \tag{2}$$

Scores are calculated to allow the ordering of the documents considered potentially relevant. This calculation is done following the formulas above, where 1 shows that the score of a document is obtained by the sum of the multiplications of the query term weight by the weight of the term in the document. This formula, although not too complex, can be simplified, as shown in 2. Since the execution time when retrieving information is crucial, the simpler and straightforward the process is, the best for the user. The simplification that we implemented consists of assuming that each term occurs only once in the query, which doesn't compromise the performance of the search since the terms are still considered and the number of times a term appears in the query should not significantly affect the relevance of a document. This allows our process to simply skip the step of calculating the query term weights as shown in 2, which can translates to crucial processing time speedup.

Once the scores are found, *sortAndWriteResults()* does what its name suggests: sorts the documents by score and writes to a results file the first `limit` documents, where `limit` is passed as argument.

3.2 Relevance Feedback

We knew `IndexSearcher` was a very limited solution, as it considers documents as relevant only for the presence of query terms in their titles and/or abstracts. So the idea of attempting to expand queries was introduced through feeding back to the Searcher information that would help it fine tune its results.

The aim of relevance feedback is to try to steer the search process behavior in a way that guides the results to more relevant document. This is accomplished by transforming the query vectors (consisting in the weights of each query term in the query) into new vectors closer to the trully relevant documents and further away from the remainder. The way we implemented this form of attempt to improve results was through the well known algorithm Rocchio algorithm.

Rocchio was developed using the Vector Space Model like many other relevance feedback approaches. Descending from the SMART Information Retrieval System, the algorithm assumes that most users have a general conception of which documents should be denoted as relevant or non-relevant. With this information in mind, an improvement over the classic score values calculation is proposed by the following the formula:

$$\vec{q}_m = \alpha\vec{q}_0 + \beta\frac{1}{|D_r|}\sum_{\vec{d}_j \in D_r}\vec{d}_j - \gamma\frac{1}{|D_{nr}|}\sum_{\vec{d}_j \in D_{nr}}\vec{d}_j \tag{3}$$

Where $\vec{q}_0$ is the original query vector, i.e. the weights of each term in the query, that in our context is simplified to $1 \times idf$; D_r is the list of known relevant documents; D_{nr} is the list of known non-relevant documents; finally α, β and γ are parameters that can be altered by the user to improve the performance of the algorithm.

Developing the formula we end up with the addition of three vectors: the original query, the sum of all relevant document vectors and the sum of all non-relevant vectors, which when applying the formula and summing all the values of the resulting vector we achieve the new score of the document.

Taking a closer look at what the algorithm attempts to do, in a first analysis it appears to do the same for all documents, which could translate in maintaining the relations between them, but it's from our understanding that although true, (referring to the vector space model that the algorithm is based on) what's really happening is that the algorithm is trying to shift the query closer to the relevant documents by the addition of its middle point ($\frac{1}{|D_r|}\sum_{\vec{d}_j \in D_r} \vec{d}_j$) and distance it from the non-relevant documents through the substation of its middle point ($\frac{1}{|D_{nr}|}\sum_{\vec{d}_j \in D_{nr}} \vec{d}_j$). This algorithm had a considerable impact in our system, which will be further explained in sections 4.2 and 4.3.

We decided to implement 2 distinct approaches of providing feedback. The first was a simulation of the user feedback and the other was a pseudo feedback. To achieve the first strategy we executed the information retrieval normally, obtaining the best results our system could supply and compared the obtained top K results with the gold standard supplied to us for the assignment. From those, the results that checked with the gold standard were considered as relevant documents and the remaining as irrelevant; this information was partially processed and persisted in files to make its further usage easier. A similar process was made to the pseudo feedback, being this one quite simpler since it only considers the relevant documents and defines them by selecting the top K values returned.

A Python script was developed to perform all of this automatically. As it served only as an auxiliary function, we didn't took in consideration the execution time nor the memory usage, since the values it produces should ideally, in a real world situation, be selected and verified in a more controlled and reliable way.

3.3 Memory Management

When developing `Searcher`, we decided to implement an intelligent memory usage strategy to take advantage of the unused memory that the system has available. By storing the terms and the respective champions needed to answer the queries performed in the system and the number of times the term is used, we can manage which terms and documents are in memory and which ones should we maintain.

The workflow that we adopted consisted in: if the program has memory available, it will store the terms used to answer the query being processed and the respective champion lists; if it detects that the memory limit is close to be surpassed, the internal cache is cleaned, leaving only one forth of what was originally there (the most used terms until then). This ensures that the most relevant terms are probably in memory and consequently will decrease the execution time, due to less I/O operations. After cleaning the internal cache the process verifies that memory was freed and continues writing the terms used which may become the new most used terms.

Below we present the diagram of all developed classes, including those for index creation and those for index querying.

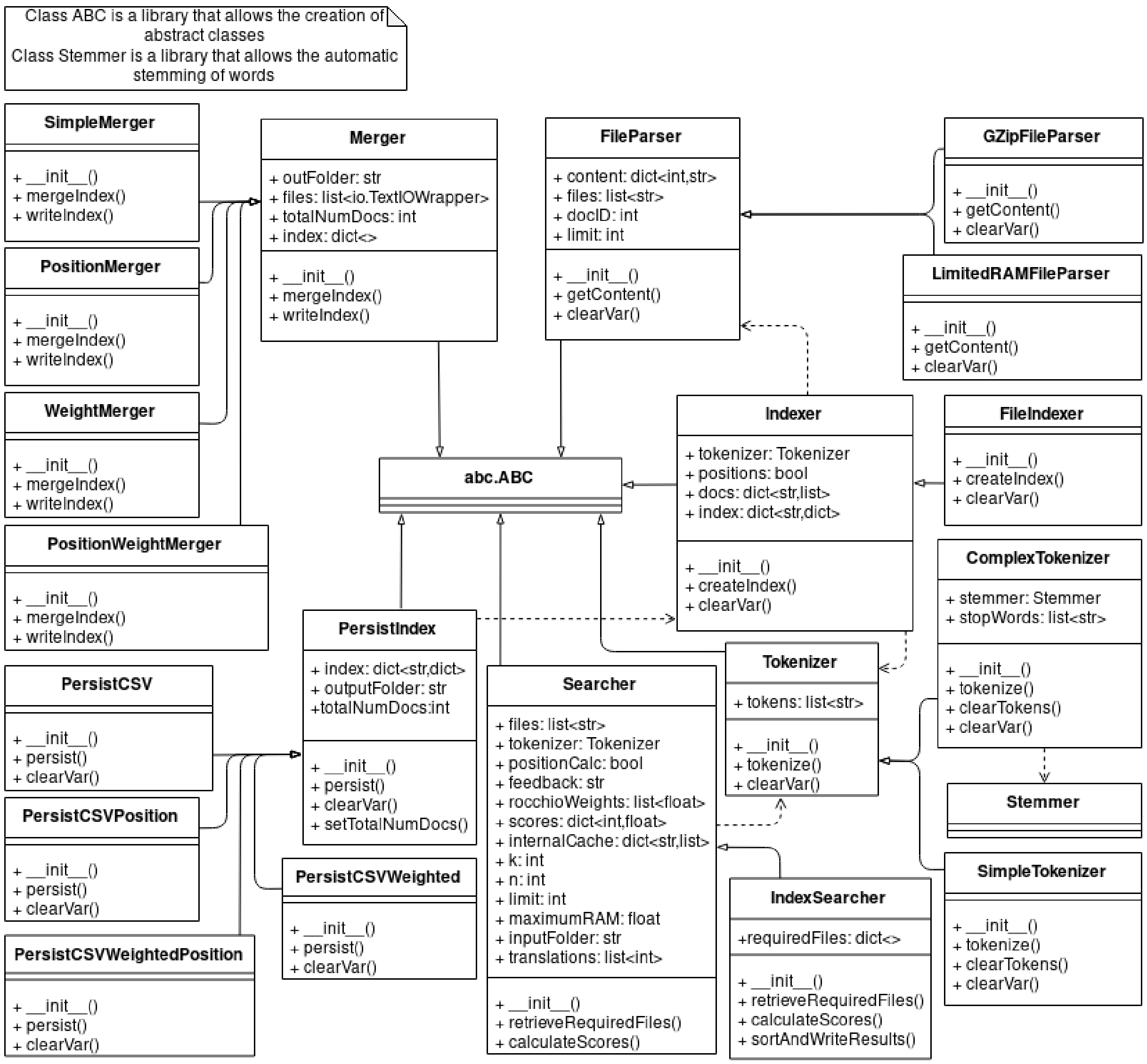

Figure 1: Program's class diagram.

4 Evaluation and Results Discussion

In order to evaluate the quality of our solutions, with and without relevance feedback, we calculate an assortment of performance metrics. The chosen metrics were: Precision, Recall, F-Measure, Mean Precision at rank 10, Mean Precision, Normalized Discounted Cumulative Gain, and all the averages in between the queries performed for all the previous metrics. The implementation of the calculations was done in `QueryAnalyzer.py`. Time related metrics were also considered, such as: Query Throughput and Median Query Latency, but for these we chose to use an auxiliary linux command line program called `time` which is able to summarize the system resources used in a program's execution.

In this chapter we explain the metrics used to evaluate the implemented ranked retrieval, present the results of our evaluation and our discussion regarding them and attempt to understand what exactly are the solutions limitations.

4.1 Evaluation Metrics

Precision and recall are one of the most used metrics in information retrieval. The first is characterized by dividing the correct retrieved documents by the totality of retrieved documents. This metric provides the percentage of the retrieved documents that are really relevant for the user. The second consists of dividing the correct retrieved documents by the totality of ideal correct documents. The result is the percentage of correct documents the system can present.

F-Measure (or F-Score) is a metric many times used to represent the system performance with only one value. This is accomplish by the combination of the two previous metrics, performing the harmonic mean between the two metrics through the following equation:

$$Fs = \frac{2 \times P \times R}{P + R} \tag{4}$$

Mean Precision is a variation of the Precision metric. Its formula is similar, the difference is that in this case the precision is calculated in the various ranks, i.e. it is calculated for every newly retrieved document and then the average of all the intermediate precisions is achieved, resulting in the mean of the precisions of the query result (hence the name). Mean Precision at rank 10 is described as the Mean Precision calculated just until rank 10 (unitl the 10^{th} value).

Normalized Discounted Cumulative Gain is a more complex metric that uses a graded relevance scale of documents to evaluate usefulness/gain of the returned results. The core idea with this metric is that relevant documents that are lower in the list should be penalized, since the important documents should be the first results provided. This is accomplished by the formula:

$$DCG = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)} \tag{5}$$

The normalization is then achieved by dividing this value by the ideal one, i.e. having the relevances in order, the first documents considered the most relevant and so on.

Query Throughput is a time related metric that indicates how many queries can be processed in one second. It's obvious that it's ideal that the information gathering and ordering should take the least amount of time possible, so the higher the query throughput the better.

Finally, the Median Query Latency, as the name suggests, is the average time it takes to process a query. Following the same logical line of thought of the previous metric, the lower this number is the better, since it means that more queries can be processed in less time.

4.2 Results

The tests phase of the project was conducted with the help of `QueryIndex` to execute the ranked retrievals and `QueryAnalyzer` to evaluate the results. They considered the original dataset indexed with the updates mentioned in chapter 1 and all 50 given queries.

`QueryIndex.py` works as the script that creates an instance of the `IndexSearcher` and, for each query, tells it to retrieve the best results. The command works as follows:

```
$ python3 QueryIndex.py [-h] [-p] [-o outputFile] [-t tokenizer]
  [-r limitRAM] [-f feedback] [-s rocchioScope] [-c numChamps]
  [-l limit] queryFile indexFolder [a b g]
```

Here, `-h` is the option that presents the manual for the command usage. Option `-p`, if present, tell the program that the index has the positions of each term in each document along with the weights. Option `-o` allows the definition of the output folder's name where the results will be stored. Option `-t` makes possible for the user to choose the type of tokenizer to be used, and the alternatives are: 'simple' for the use of the `SimpleTokenizer` class, and 'complex' for the `ComplexTokenizer` class. The chosen tokenizer must be the same used when indexing the text corpus. Option `-r` allows the user to define the maximum amount of memory that can be used by the process running the program. Option `-f` makes possible for the user to choose a form of relevance feedback, so that there is a greater assurance that the returned documents are actually relevant to the query. The alternatives are: 'pseudo' and 'user'. Options `-s`, `-c` and `-l` allow the definition of the number of retrieved documents considered for the Rocchio algorithm (applied in the relevance feedback), the size of the champions list and the number of scores to return (to store in the output files). The previous arguments are all optional and the actual values for these arguments must appear right after the respective options. `QueryFile` and `indexFolder` are the only obligatory parameters, as they tell the program where are the queries and the index files. `A`, `b` and `g` are optional parameters that must be defined if the relevance feedback is activated (if `-f` and `-s` are also defined), as they correspond to the Rocchio algorithm's parameters α, β and γ.

We executed `QueryIndex.py` for different combinations of parameters to ensure its correct functioning and to search for the configuration with highest performances. It was initially tested with indexes with positions and both types of tokenizers, although the remaining tests were all using only weights to reduce execution times and only the complex tokenizer to remove noise and increase precision. Then we tested the program with and without champions list (although this was not done through the command's parameters) and for champions lists of different sizes. Following we also varied the number of documents the program returned. And finally we tested the program with relevance feedback, with both pseudo and simulated user feedbacks, and with different Rocchio parameters values.

The presence of the champions list seemed to have no relevant impact on the overall performance, with tests at the time rounding 19% of precision (with a champions list of size 10^4). This feature was important for the execution times, as it reduced the search space of the program for each query without loosing valuable information. What we later understood was that, if its size was reduced, the lost of useful information and relevant documents started to have a big impact on the performance, as for a list of size 10^3 the precision dropped to about 16%.

With the champions list at size 10^4, we tried to reduce the number of results returned from 100 to 50, 20 and 10. The later reached a precision of 26%, which was good news to us.

Although pseudo feedback was also part of our tests, we knew *a priori* that this would not influence the results at all, since it assumed that the first results the program already considered more relevant were in fact relevant and increased their final scores. User feedback, on the other hand, proved to have great value when tested. To test this form of feedback, we varied the parameters `n` (number of retrieved documents considered for the Rocchio algorithm) between 5, 10 and 20 with the `a`, `b` and `g` parameters fixed at 1.0, 0.5 and 0.25 respectively. The final precision for each `n` was: 28%, 29% and 29%. At last we inverted the parameters variation in order to study the effects of α, β and γ. To do this, we fixed `n` at 10 and tested the 3 parameters for values between 0 and 1. The values with the best precision were α `= 1.0`, β `= 1.0` and γ `= 0.1`, with a final precision of 31%. Table .... (located at the end of this document) presents the queries results and statistics of the best performance achieved for further analysis.

4.3 Discussion

So the tests results were presented in the previous section, and the seemingly best combination of parameters was achieved with the available resources and developed features. However, we did not content with accepting the facts and looked for logical explanations to the influence of each factor on the final performance. Here we share our thoughts on each observation made and attempt to explain why the final performance is as it is.

The reason why champions lists reduce execution times has already been slightly discussed. This is mainly due to the fact that, before executing any computation on the index, a cheap verification is done to determine whether it is worth executing them, so it makes sense that, if a sufficient amount of unnecessary computations are avoided, the time required to answer a query is significantly reduced. The key aspect here is the condition that determines whether de calculation must be done or not and the goal is to avoid loosing relevant information as much as possible while speeding up the retrieval process. This is also why champion lists too small have a negative impact on the overall performance, since they have a higher probability of discarding index entries that in reality would benefit the results.

The fact that the number of results returned directly influences the results precision is also relatively simple to understand. As the performance metrics are only applied to the returned documents, ...

The way the user feedback through the rocchio algorithm works is: scores from documents considered not relevant to a given query by users must be lowered and scores from documents considered in fact relevant must be increased. Well, it is only natural that, when applied with appropriate scores regulation, this feedback will have a positive influence on the program's precision because it gives the program "inside information" on what the user making the query is probably looking for. And the more documents considered on the algorithm, the more the results will be closer to the truly relevant documents and further from the non relevant as more documents from the results are affected by this regulatory feature. ....

In our analysis we only mention precision as a means of determining whether the variation of parameters was good or bad, but in reality our observations are also verified by the other metrics, only their evolution is not as straightforward to explain as precision's. Nevertheless, we reached a few conclusions regarding other performance metrics worth mentioning.

A low recall is common amongst all tests and, at first sight, it seems a bad sign. However, it is actually expected to be low, since the amount of returned documents is usually very low compared to the entire list of documents from the index.

....

Precision

Explain why:

X champions list reduces execution time

X champion lists too small have worst precision

- the less results are returned the higher the precision

X simulated user feedback increased precision

X the more documents considered on the rocchio algorithm the higher the precision

- some rocchio parameters values result in higher precision

Other Observations

Explain why:

X low recall is expected entire universe of documents)

- .....

4.4 Implementation Limitations

So far we have explored ways of improving the program's performance through limited, although relevant, strategies. But the hard truth is that a precision of 30% on a sort of search engine is not exactly something to be very much proud of, or, at least, not the ideal value a regular user expects from such system. In this section we attempt to understand why our solution is so limited and what could be done to greatly increase its quality.

....

Two major factors limit retrieval speed.

- if index files were smaller, retrievals would be faster we kept their sizes as they were as we considered them more realistic than reducing them to small amounts of Mb

- ram limitations also affect execution time we found that the minimum amount of memory required for retrievals to work reasonably well is 500Mb

There are some reasons that explain why a high precision is never achieved.

- synonyms and similar/related words are not considered when retrieving documents

- query log mining

....

5 Conclusions

After completing the assignment, we drew a few conclusions regarding our solutions and the whole concept of efficient indexing and information retrieval.

The biggest challenge we faced was .... rocchio, memory

From this assignment, we take ....

The overall perspective of our performance regarding the project is ....

References

1. S. Matos, *IR: Assignment 3*, University of Aveiro, 2019/20.