Assignment 3

Filipe Pires [85122], João Alegria [85048]

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

December 10, 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 (*I*). 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 the abstract template and is capable of selecting which index files will be required to answer a given query, assigning scores to documents from the index and returning the documents considered most relevant to the query according to these scores.

The step of determining which index files will be needed to answer the query is done through the function *retrieveRequiredFiles()* and is basically a comparison of each query term with the name of each index file. As each index file is 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.

Calculating the score of each document regarding a query is not as easy to explain. ..... explain *calculateScores()* .....

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

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 transform query vectors (consisting in the weights of each query term in the query) into new vectors closer to the actually relevant documents and further away from the remainder. The way we implemented this form of attempting to improve results was through a well known algorithm named Rocchio algorithm.

..... explain rocchio .....

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:

$$Fs = \frac{2 \times P \times R}{P + R} \tag{1}$$

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:

$$DCG = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)} \tag{2}$$

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

Lorem ipsum ...

4.3 Discussion

Lorem ipsum ...

4.4 Implementation Limitations

Lorem ipsum ...

5 Conclusions

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

The biggest challenge we faced was .....

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.