Information retrieval assignment 1

Niels Van den Broeck, Stein Vandenbroeke

November 3, 2024

1 Query Processing

To easily test and change the Query Processing, we created a folder with the different used algorithms that are all subclasses of QueryProcessor. Every QueryProcessor has a tokenize function that takes an input string and produces a list of strings. Lists are easy to create:

- BasicQueryProcessor:
  We just split the input string on the space character and convert all characters to lowercase.
- RegexQueryProcessor:
  We simply split the input string based on the regex r'\ w+'. This will result in removal of all special characters. After that, we convert the characters to lowercase.
- NLTKQueryProcessor:
  The NLTKQueryProcessor will run the same regex as the RegexQueryProcessor, but in addition, it removes all english stop words. By doing this we improve the time to analyze queries by around 15%. The reason for this is that the common stop words are present in almost all documents, resulting in very large lists for those terms. By removing these stop words, we make our algorithm way faster plus a fair amount of precision increase. We also implemented the frequentWordIndexing function that allows to remove common words by analyzing only a fraction of the dataset. Since the possibility of common words are roughly the same over the files. we can take a sample and check all the most used words in these documents. These words are considered stop words and will be removed in the whole dataset.

2 Inverted-indexing

We started with a basic implementation of the inverted-indexing algorithm (DocSearch), where we used a data-structure consisting of a dictionary of dictionary's, with the first dictionary's key the term and the second dictionary's key the doc_id. Intuitively, because of the large overhead of a dictionary implementation in python, this results in full memory and a crashing computer. This is an unfortunate but expected result, and it gave us an easy way to understand everything and compare the results of the small dataset to the later implemented inverted-indexing algorithm. After some thinking, we came up with the following inverted-indexing algorithm:

1. Order the files by name.
2. Loop through all the documents and fill a dict<term, int> where the int is how many documents contain the certain term. This is done to determine the amount of storage that is needed for the array of tuples in the next step.
3. Loop again through all the documents, now a we fill a dict where the key is the term and the value is an array (with the precomputed size stored in dict from step 2) of tuples containing of the doc_id and the frequency of the term in that document.

This implementation will result in a much better space complexity. We went from around 14GB to around 6GB. This is because we replaced every term value from a dict type (with large overhead) to a static sized array of tuples. Obviously, this results in a longer indexing run time because it requires to loop twice over all files. First to get the array sizes, and second to fill them. Although, it only slows the indexing and not the search/retrieval. The reason for this is that the function does not require random access to the certain doc_ids. A disadvantages of this approach is that it is harder to multi-thread or extent the indexing by adding new documents. In this case, you need to sort the whole array all over again. This was not the case when using dictionaries to store the document frequency.

3 Document Search and Retrieval

To compute the score of a document, we process the input query with the same query processor as the documents, and than calculate the cosine similarity between the vector of the documents and the query itself. Of course, this is only useful for the documents containing the terms in the query. Otherwise, the score would always be zero.

$$\sum_{t \in q} \frac{\log\left(\frac{N}{df_t}\right)(1 + \log(tf_{t,d}))}{norm_d} \times \frac{\log\left(\frac{N}{df_t}\right)(1 + \log(tf_{t,q}))}{norm_q}$$

After the implementation, we realized there were a couple of speed improvements we could do to our function. We moved the doc_weight value to an optional function precalculate. Next to that, the $norm_q$ can be removed since all scores get divided by the same value which doesn't change the ranking order. Resulting in following formulas:

$$\text{doc\_weight} := \log\left(\frac{N}{df_t}\right)(1 + \log(tf_{t,d}))$$

$$\sum_{t \in q} \frac{docweight}{norm_d} \times \log\left(\frac{N}{df_t}\right)(1 + \log(tf_{t,q}))$$

We also replaced the normal list that is sorted at the end by a heap using the "heappush" and "nlargest" functions. This way we can retrieve the best k matching documents way faster.

4 Precalculate document weights

Because the document weight stays the same despite different queries, we can precalculate them (3). To keep the memory usage low, we can overwrite the term frequency with the calculated weights. This is only possible if we assume there are no files added after the precalculation. Otherwise, we need to keep the original term frequency to recalculate later. To do this we added a function 'pre_compute_document_weights' with a param to overwrite the inverted index or not, depending on the needs.

5 System Evaluation

5.1 Small dataset

K	1
MAP	0.5040322580645161
MAR	0.5020161290322581

For the small dataset, we only had one result per query and thus we chose the use a k value of 1. This result in a MAP@1 value of around 50% meaning of the 249 queries we run around 224 are the same as the expected result. This seems like it could be better.

5.2 Large dataset

K	3	10
MAP	0.16012065203439957	0.14451289949942434
MAR	0.03186807932386387	0.09450355209845919

The results of the large dataset seem even more disappointing. We tried to figure out what went wrong but the large time to compute the queries made it difficult to quickly see improvement for small changes that could have big changes on the results. We spent many hours on debugging the results but could not find a mistake in the implementation of our formula's.
The MAP@10 value of 0,16 means that 1 out of every 10 retrieved documents is correct.
The MAR@10 value of 0.9 means that 9% of the correct documents could be retrieved.

Link to our Github repository, including the src folder and results:
https://github.com/SteinVandenbroeke/Information_retrieval