MIRCV Project

2023-2024

RAYNAUD Lucas

1. Introduction

Search engines are parts of everybody's life at any moment. This project includes multiple steps to create a functional search engine. These steps involve computing a document collection to obtain usable data, indexing this collection for faster search capabilities, and performing queries over the collection using the created index. The project appears to focus on the development and implementation of an efficient search engine, detailing the processes and methodologies used in its creation.

2. Overview

This project can be decomposed into multiple steps to achieve a working search engine. The first one is to compute the document's collection to get usable data. The second one is indexing this collection to make it faster to search. The last one is to perform queries over the collection using the index created before.

2.1. Preprocessing

The collection used is, as specified before, the msmarco collection, composed of 8,841,823 documents gathered on a tsv file of the format : <pid>\t<text>\n. The collection is stored in a compressed file, and it is uncompressed on read to be converted as a dictionary using the pid as key and the text as value. Here is an example of a line of the collection :

The first step is to tokenize the passage, the string is just split using spaces. The second step is to normalize the tokens by lowering all the characters. The third step is to remove the punctuation, it is replaced by nothing.

The fourth step is to remove the stop words, which are insignificant words. Those words are filtered out because they are too common, and they don't offer so useful information. There is not a unique stop words list. The stop word list of the nltk package is used, which is the Natural Language Toolkit available in Python. Also, when the words are stemmed, it means that the similar words are gathered to their stem, like for words "programmers", "programs" or "programming", they will be reduced as "program". It is useful because it gathers variations of a word, it won't lose so much information because initial words are from the same topic, it will especially reduce the size of the lexicon, in this example, three words are reduced to only one. The result is the following :

The documents are now ready to be indexed.

2.2. Indexing

After preprocessing the collection, I created the inverted index, the lexicon and the document table.

2.2.1. Saved structures

The inverted index is a data structure mapping a term (word) to his presence into documents. In this case, the number of occurrences of a word in a document is also saved. The structure is the following : <termid>\t<docid>:<occurence>,...,<docid>:<occurence>

To create the inverted index, each text is computed and for each term, an entrance in his posting list is added and the occurrence count increased.

Also, if the term never appeared before, it is added to the lexicon. The lexicon is mapping terms to a unique id to reduce the size of the inverted index and also to allow integer compression.

To save space, some compression or optimizations can be performed, like gap compression. The gap compression changes the way docid are stored in the posting list. The docid are not saved using their real id but using the gap with the previous docid stored in the posting list. It is useful when the docid is too big, and even more so when the term is common.

Also, to speed up the query processing, it is useful to save the size of each document. Like we will see in the next part, score computing needs to know this data, and it is not easy to retrieve it without charging the collection.

2.2.2. Binary saving

The inverted index is taking up a huge amount of memory space, to reduce it, it is possible to transform it from plain text to binary, which will allow it to perform a variable byte encoding. The principle is to encode an integer into a smaller number of bytes than the language. For this, the integer is converted to its binary representation. Then, we take his first 7 lowest bits, and we append 0 to indicate this isn't the last byte. We take the 7 lowest bits of the remaining bits, and we add a 1 to signify this is the last byte. This added bit is used to know if this is the last byte of the integer. For example, take 1234 with its binary representation 100 11010010. We take the lowest 7 bits : 1010010, and we add a 0, as it is not

the last byte. We obtained 10100100. Then, we take the next 7 bits and add 1, obtaining 10011. So, the encoding of 1234 is 10011 10100101. 1234 is encoded in 2 bytes instead of 4 bytes for Python.

2.3. Querying

After processing the documents and computing the different indexes, The main goal is to make queries and rank documents by relevance. To achieve this, there are many different methods, among which are the TF-IDF and BM25 scoring functions. For each one, the score of a document is calculated by summing the score obtained by each term of the query about the document.

2.3.1. TF-IDF

The TF-IDF (Term Frequency-Inverse Document Frequency) is a ranking function used to measure the importance of a term considering a document. It can be decomposed as two parts : TF and IDF.

The TF (Term Frequency) part of the algorithm calculates how frequently a term occurs in a document. This is done by dividing the number of times a specific word appears in the document by the total number of words in that document. The idea is that the more frequently a term appears in a document, the more important it is for that document. However, if a word appears frequently in many documents, it's not a useful metric for distinguishing one document from another.

This is where the IDF (Inverse Document Frequency) part comes in. IDF diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. IDF is calculated by taking the total number of documents, dividing it by the number of documents that contain the word, and then applying the logarithmic scale to this quotient.

Together, TF-IDF allows for the evaluation of how relevant a word is to a document in a collection, with higher scores indicating greater relevance.

2.3.2. BM25

BM25, short for "Best Matching 25," is an algorithm used in information retrieval for ranking documents based on the queries. It is an evolution of the TF-IDF (Term Frequency-Inverse Document Frequency) model and is known for its effectiveness in handling the relevance of documents to a given search query.

The BM25 algorithm differs from TF-IDF by introducing two key concepts: term frequency saturation and document length normalization. Term frequency saturation means that after a certain frequency, the importance of a term's frequency in a document plateaus, addressing the issue where TF-IDF linearly increases the weight with term frequency, which can be unrealistic. BM25 handles this by using a non-linear function that grows quickly for lower term frequencies and slows down as frequency increases, preventing overemphasis on higher term frequencies.

Document length normalization is another crucial aspect of BM25. It adjusts the document's score based on its length, ensuring that longer documents do not inherently have an advantage over shorter ones. The algorithm introduces two parameters, usually denoted as k1 and b, which control how term frequency and document length are factored into the scoring. The parameter k controls term frequency saturation, and b controls the degree of length normalization.

3. Performances

The performances are divided into three parts : time, space and precision. The time for indexing, creating, and saving the structures used and the time taken for each query. The space taken by those structures in memory. The precision of the performance of the model according to a standard test collection.

3.1 Time

The first part is to create the inverted index, the lexicon and the document table. Those three structures are created in the same time, the total time for the full collection of 8841823 documents is 4968.33 sec, approx 82.80 min. The speed of processing is almost 1780 documents per second.

The second part is to load the different structures to process queries. The inverted index load in 113.72 sec, the lexicon in 0.13 sec and the document table in 7.13 sec.

The third part is the time taken to process a query. The velocity of the model was tested during the evaluation, with TF-IDF and BM25 ranking functions and the same amount of 54 queries, the test was performed two times for each on the same queries. For the TF-IDF, the time taken is approximately 66.22 sec, or 1.23 sec per query. For the BM25, the time taken is approximately 106.87 sec, or 1.98 sec per query.

3.2 Space

As the indexing is taking a long time, it is necessary to store it. The space taken by the inverted index is 436785 Ko, this space is lowered by the variable byte encoding seen previously. There is no comparison with the space taken if this encoding wasn’t performed because the program was crashing my RAM during the running. We can consider the space taken to be bigger than the space taken after compression. The lexicon is taking 494 Ko as a plain text file, and the document table is taking 102 433 Ko, also as a plain text file. The document table is taking up a really huge amount of space because it is storing document ids that are in the range of 0 to 8841822. It could be a possibility to perform a variable byte encoding or just not store document ids but only length and consider lines as index, but it is only taking 7 seconds to load it, and it is performed only once per script run.

3.3 Precision

The final goal of this project is to query the collection and get the most relevant documents. For testing the system, there is a list of test queries with the expected documents to be returned. The expected documents are rated by relevance from 0 (not relevant) to 3 (totally relevant). For comparing the computed results by the TF-IDF and the BM25 to the expected results, we can use the nDCG (Normalized Discounted Cumulative Gain). The nDCG is considering two assumptions : the totally relevant documents are more useful in the first results returned, and the totally relevant documents are more useful than partially relevant documents that are more useful than not relevant documents. As his name mentions, the nDCG is the sum relevance value of all results in a search, but those values are logarithmically reduced proportionally to their position in the result list. Finally, it is normalized by dividing this gain by the gain obtained for the expected results. Furthermore, it is possible to separate patient users from impatient users, the first will be more inclined to consult less well-classified results than the second, so the logarithm used to lower the relevance score will not be the same, log10 for patient users and log2 for impatient users. The score calculated by the nDCG is in the range from 0 to 1, as it is normalized.

	TF-IDF	BM25
Impatient user	0.11912212034693231	0.18249851524083632
Patient user	0.1080524083016866	0.16130841665728948

Table of nDCG results for patient and impatient user and for TF-IDF and BM25 ranking functions

4. Conclusion

To conclude this paper, it is important to note the limitations of the system. It is hard to have the fastest system without consuming huge amounts of space and having good querying precision. During the implementation step, I had to choose collection processing methods that reduced the lexicon size, spelling and stemming, that maybe reduced too much the size of the lexicon. It was a good thing for space saving and for the time of processing but the precision was negatively affected for sure. The main part is to judge useful and prioritize the optimizations to obtain the best search engine. This system also has limitations about the language and the topics of the msmarco collection.