UNIVERSITÀ DI PISA

MIRCV PROJECT

Multimedia Information Retrieval and Computer Vision

Year 2022/2023

- Giuseppe Aniello 643032
- Edoardo Malaspina 578355
- Valerio Secondulfo 643022

Table of Contents

Project Goal

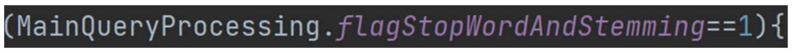
The program is composed of two main parts: the first one is focused on the creation of the inverted index structure from a set of text documents, in order to do so a preprocessing step is applied.
The second one provides a command line interface to process queries and returns top k documents ordered by relevance as result.
In the next points all the main steps will be highlighted and explained in detail.

Collection

The collection used in this program is the "passage ranking dataset" available on https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020 . It is made up of 8,841,823 documents in the format <pid>/t<passage>/n where pid is the DocNo and passage is the document text. The file size is around 2.9 GB.

Preprocessing

In the preprocessing phase the document's text are cleaned before the indexing phase.There is a flag to choose if the stemming and stop words removal should be applied or not in this phase. Setting the flag to 1 means doing stemming and stop words removal.



```
(MainQueryProcessing.flagStopWordAndStemming==1){
```

The preprocessing is applied to each document, encoded in UTF-8, passed in the "preprocess" method.

The passages that will be executed on the input document are the following:

- text cleaning
- text to lower case
- stop words removal
- stemming

Text cleaning

In the text cleaning phase the document id is eliminated, stripped off all the non-ASCII characters, are also eliminated the ASCII control characters and the non printable characters. This phase also eliminates numbers from the documents along whit words containing numbers at the beginning, at the end and in between characters of it.

Text to lower case

In the lower case phase the cleaned text is transformed by putting all the characters in lower case.

Stop words removal

In the stop words removal phase all the stop words contained inside each cleaned document are removed by using a stop words dictionary taken from:
https://www.kaggle.com/datasets/rowhitswami/stopwords?resource=download .

Only English stopwords were considered since it was the language used in the input documents.

Stemming

In the stemming phase has been applied the Porter Stemming algorithm to each word inside the document. Along with that has been used a function in order to limit the word’s characters to 20 for words longer than that and to increase the characters to 20 for shorter words by adding whitespaces at the end of them.

Indexing

In this phase the inverted index structure is built. The aim of this phase is to end up with five different files on disk:

- File with all the term frequencies of all the posting lists;
- File with all the docIds of all the posting lists;
- File of the document table;
- File of the skip info;
- File of Lexicon;

(Actually this program will generate two versions of the files above, one for the case with stemming and stopwords removal and one for the other case).

SPIMI

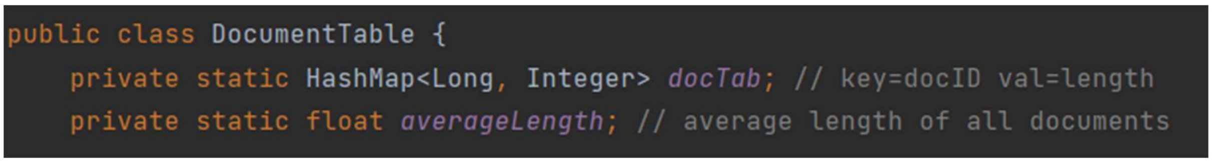
In order to do so the collection file is read, after the application of the preprocessing each token is added in the lexicon structure and consequently a posting list is created or updated. The posting list is made up of a list of term frequency-docId pair. This phase is performed exploiting SPIMI (Single Pass In Memory Indexing) algorithm, where are created blocks of terms, term frequencies and docIds checking at each step if the ram available is under a certain treshold. When this check fails the block is saved on disk exploiting a random access file, memory is cleaned and a new block is created. On disk there are two different files for term frequencies and for document ids, in this way different compression algorithms can be applied in order to optimize the compression ratio.

Merging

After the creation of all blocks a merging phase is performed, in this phase the first and the second block are merged, then the new block is merged with the third and so on until there is just one block. When two blocks are merged they are deleted in order to save space on disk.

Document table

During the creation of the SPIMI-blocks the document table is built. This structure contains information used for BM25 scoring during query processing and during computation of BM25 term upper bound.



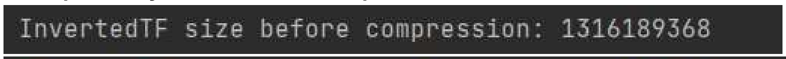
```java
public class DocumentTable {
    private static HashMap<Long, Integer> docTab; // key=docID val=length
    private static float averageLength; // average length of all documents
```

The docTab field is an HashMap with key docId and value the length of the corresponding document. There is also another field, averageLength, that is the average length of all document across the collection.
The document table after creation is saved on disk. When the query processing program is started the document table is loaded in main memory. The user can choose to apply or not stemming and stopword removal and consequently the right document table is applied.
Document table should contain also docId to docNo mapping, in this program in order to save memory (since this structure has to be loaded in memory during query processing) has been chosen to not add this field given the fact that $docId = docNo + 1$. This situation is handled adding one to each docId during indexing and subtracting one before showing the results during query processing. In this way the user only see docNo.

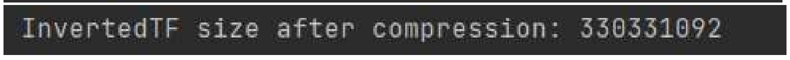
Compression

The posting lists saved on file (divided in two different files for docIds and for term frequencies) have size of 4 bytes of each term frequency and 8 bytes for each docId, in order to save space on disk a compression phase is performed. DocIds and term frequencies have been saved in two different files in order to apply two different compression algorithms.

- Compression of term frequencies: unary compression is applied because a significant number of term frequencies were lesser or equal to 7 so with this kind of compression the size of the file on disk has been reduced from 4 bytes to 1 bytes for each term frequency so the compression ratio is around 4.

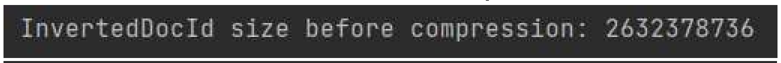


```
InvertedTF size before compression: 1316189368
```

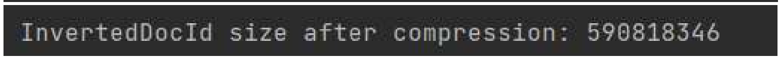


```
InvertedTF size after compression: 330331092
```

- Compression of docIds: before applying the compression all the docIds are transformed in d-gaps and then gamma compression is applied. Also here is observed that the size of the compressed file is ¼ of the size of the uncompressed file.



```
InvertedDocId size before compression: 2632378736
```



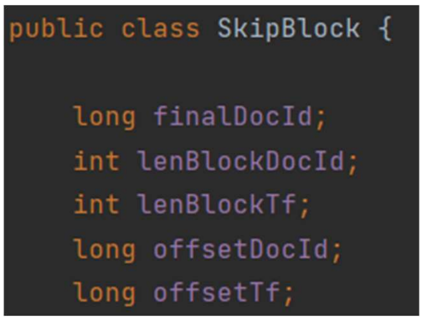
```
InvertedDocId size after compression: 590818346
```

Skip Info creation

During the process of compression is created the skip info file that allows to split the posting lists in different blocks with small size

(size of block $\simeq$ number of blocks $\simeq \sqrt{Document\ frequency}$).

Skip infos are made up of SkipBlocks, each one has size of 32 bytes.



```
public class SkipBlock {

    long finalDocId;
    int lenBlockDocId;
    int lenBlockTf;
    long offsetDocId;
    long offsetTf;
```

Where:

- finalDocId is the last docId of the block;
- lenBlockDocId is the size in bytes of the docIds compressed on file.
- lenBlockTf is the size in bytes of the term frequencies compressed on file.
- offsetDocId is the starting point of the block in the docIds compressed file.
- offsetTf is the starting point of the block in the term frequencies compressed file.

The creation of skip info is necessary in order to implement skipping in the query processing phase, because it allows a faster implementation of next() and nextGEQ() methods.

Term upper bounds

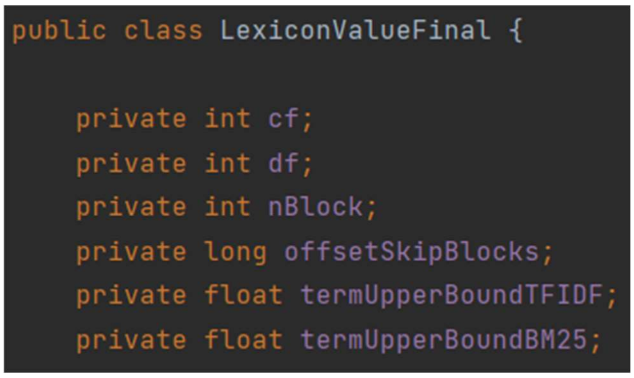
In order to apply dynamic pruning strategies (in this case MaxScore) during query processing, a phase in which term upper bounds are computed is necessary. In this program there are two term upper bounds, one computed exploiting TFIDF scoring function and one exploiting BM25. In this way when a query is processed is possible to choose between the two different scoring function.

The term upper bound is calculated in this way:

- For each term of the Lexicon a query, containing only the term itself, is made.
- For each document of the posting list is computed the score exploiting one of the two possible scoring functions (TFIDF or BM25).
- The term upper bound is the higher between all those scores.

Lexicon Final

Given the fact that Lexicon data structure is made up of many field exploited during the indexing phase but useless during the query processing in this program in created another data structure, the LexiconFinal, a smaller version of Lexicon that allows to save memory and avoid to load useless information during the query processing. LexiconFinal is a TreeMap with Term as key and LexiconValueFinal as value.



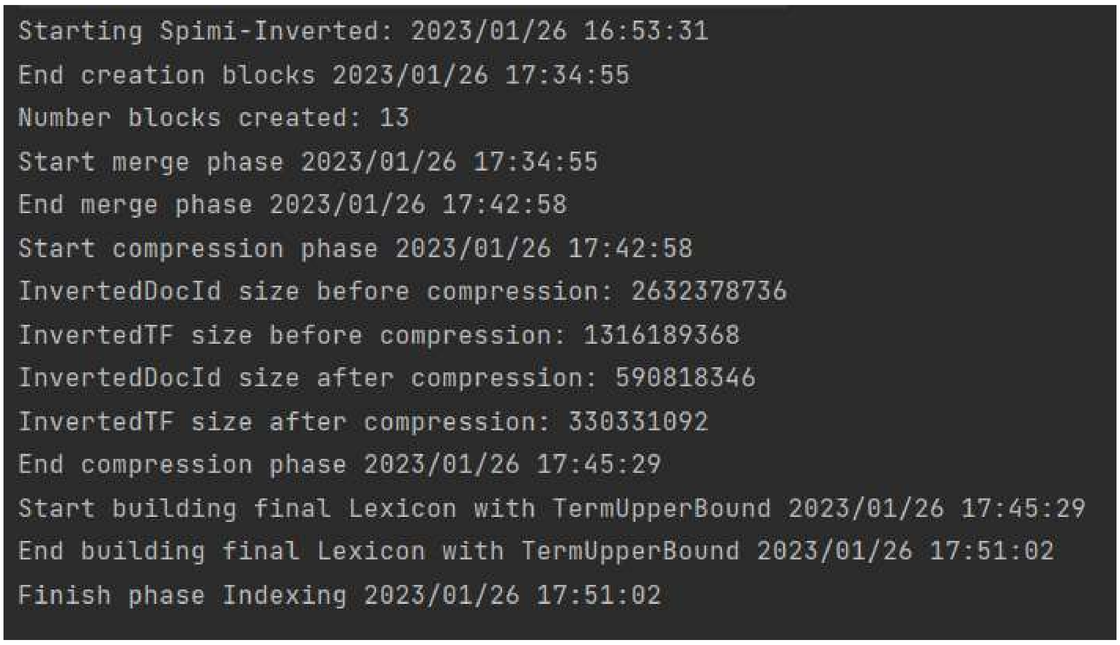
```
public class LexiconValueFinal {

    private int cf;
    private int df;
    private int nBlock;
    private long offsetSkipBlocks;
    private float termUpperBoundTFIDF;
    private float termUpperBoundBM25;
```

Where:

- Cf is the collection frequency of the term;
- Df is the document frequency of the term;
- nBlock is the number of skip blocks in which is split the posting list of the term;
- termUpperBoundTFIDF is the field in which is saved the term upper bound computed using TFIDF scoring function;
- termUpperBoundBM25 is the field in which is saved the term upper bound computed using BM25 scoring function;

Indexing time
Number of blocks created is 13.

Total time to build the inverted index structure in the case of stemming and stopword removal is 57 minutes and 32 seconds.



```
Starting Spimi-Inverted: 2023/01/26 16:53:31
End creation blocks 2023/01/26 17:34:55
Number blocks created: 13
Start merge phase 2023/01/26 17:34:55
End merge phase 2023/01/26 17:42:58
Start compression phase 2023/01/26 17:42:58
InvertedDocId size before compression: 2632378736
InvertedTF size before compression: 1316189368
InvertedDocId size after compression: 590818346
InvertedTF size after compression: 330331092
End compression phase 2023/01/26 17:45:29
Start building final Lexicon with TermUpperBound 2023/01/26 17:45:29
End building final Lexicon with TermUpperBound 2023/01/26 17:51:02
Finish phase Indexing 2023/01/26 17:51:02
```

This time is also inclusive of the skip info creation and of the computation of terms upper bound.

The time took for the two types of indexing (with and without stemming and stopword removal) is approximately the same, both took less than an hour.

Query Preprocessing

The second program is divided into two main phases: the first one is the phase in which the document table is loaded in memory, the second one is the phase in which the queries are actually processed.

First phase

In this phase the program asks if you want to use stemming and stopword removal or not. After the user answers the program loads the appropriate document table in memory. In order to save memory this program does not load the entire lexicon in memory in this phase, the only data structure that is completely loaded is the document table. This phase could take some seconds.

Second phase

This phase starts with two questions to the user. In the first question the user can choose if he wants to perform conjunctive or disjunctive queries, in the second the user can select the scoring function (TFIDF or BM25).

- TFIDF:

$$w_{t,d} = \begin{cases} (1 + \log(tf_{t,d}))\log \frac{N}{df_t} & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases} \qquad s(q,d) = \sum_{t \in q \cap d} w_{t,d}$$

- BM25:

$$RSV^{\text{BM25}}(q,d) = \sum_{t_i \in q} \frac{tf_i(d)}{k_1\left((1-b) + b\frac{dl(d)}{avdl}\right) + tf_i(d)} \log \frac{N}{n_i}$$

After the two questions the program asks the user to input the query. DocIds of the top k document are printed. If the queue is not full the remaining positions will be filled with “-1” (for example if a conjuctive query is made and there are not enough documents containing all the

query terms) and the program will show in those position “no result”.
After the result the program will print the time took by the query processing measured in milliseconds (the computation of this time stars when the user press enter after inserting the query and stops when the program prints the results).
In order to save memory this program does not load the entire posting list of a query term in RAM: initially the first skipblock for each query term is loaded, then during next() or nextGEQ() operations another block could be loaded and the first one is discarded. During nextGEQ() before loading a new block skip info are exploited in order to find if the new block is the right one or if it has to be skipped. In this way in main memory is always loaded just one block for each query, so this program could perform even with long posting lists and small amount of memory (however there are some limitations due to the fact that the dimension of the block is related to the total size of the posting list so at a certain point memory could not be enough even for a single block).

Performances

Here is showed a comparison of some of the combinations of flag using an example query taken from some documents in the collection:

Query: “manhattan project”

Conjunctive	BM25	Stemmed/Sw remov	79 ms
Conjunctive	BM25	Not stem/sw remov	52 ms
Conjunctive	TFIDF	Stemmed/Sw remov	64 ms
Conjunctive	TFIDF	Not stem/sw remov	44 ms
Disjunctive	BM25	Stemmed/Sw remov	68 ms
Disjunctive	BM25	Not stem/sw remov	61 ms
Disjunctive	TFIDF	Stemmed/Sw remov	66 ms
Disjunctive	TFIDF	Not stem/sw remov	49 ms

As expected queries on index without stemming and stopword removal are faster because posting lists are shorter and is also observed that queries exploiting BM25 are a little slower than ones exploiting TFIDF. Is also noticeable that conjunctive queries in general are faster than disjunctive, this is reasonable because in conjunctive queries more postings are skipped (altought there is some exception). Confirming this is easy to observe that the fastest combination in the example is Conjunctive-TFIDF-not stemmed/Sw removed. Even considering the different combinations the difference in time is really small and results vary a bit reproducing tests many times (this also happens during the indexing phase but in that case the duration is a bit less than one our so a small change is less noticeable).