UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Multimedia Information Retrieval and Computer Vision

Project documentation

TEAM MEMBERS:
Fabiano Pilia
Emanuele Tinghi

Contents

Chapter 1

Introduction

This project aims at realizing a search engine performing text retrieval on the following dataset:

https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020

The data set consists of a set of *8.8 million* text documents, which will be used to build an inverted index data structure on which queries can be performed.
The source code of the project is available in the following GitHub page:

https://github.com/Fabi8997/mircv-project

We've created a test collection of *500000* documents to be used during the development of the project. It is obtained using a script that reads the collection and selects the first 500000 documents, creating *tar.gz* file at the end. The results in the report refer to the actual collection of 8.8 million documents, not the one used during the development phase.

1.1 Structure and main modules

The project consists of the following modules, each reflecting a different component:

- ***inverted-index***: from the collection, it builds the data structures that form the inverted index and writes them to disk;
- ***query-processor***: it provides an interface that allows the user to submit queries and receive the documents that are most relevant. The CLI also allows the user to select the query type, decide on the scoring function and enable debugging mode;
- ***query-evaluator***: it reads a batch of queries and performs the evaluation of each one. The results are written in the format required by the *trec_ eval* tool to evaluate the performance of the search engine. It also returns an average of the time taken to evaluate each query.

Chapter 2

Pre-processing

Initially the application decompress the file containing the collection, resulting in a tar archive containing the document collection. Each line of the file is read using UNICODE as encoding standard and contains a document in the following format:

$$<pid>\backslash t<text>\backslash n$$

To pre-process each line, the class *it.unipi.mircv.parser.Parser* is used, which takes a line as input and generates a *ParsedDocument* containing the following information:

- document id: it identifies the document during the creation of the inverted index and the query processing. It's better to use the **docId** instead of the **pid** since it creates a strictly sequential identification of the documents in which we're sure that each consecutive number is present, using the pid this isn't guaranteed due to the fact that the malformed documents are skipped, creating a hole in the sequence of identifiers.
- document length: the number of terms extracted during the pre-processing from the document.
- docNo: it contains the *pid* of the document.
- terms: the list of terms extracted from the document body during the pre-processing phase.

In the following sections are presented the steps performed during the pre-processing of each document.

2.1 Text Cleaning

To clean each document the document parser performs the following steps:

1. split the line of the collection using the $\backslash\mathbf{t}$ as delimiter obtaining the *pid* and the *document body* of the current document.

2. If the document is malformed:

   - the line is not formatted in the correct way
   - the *pid* is not present
   - the *document body* is empty

   then it is skipped.

3. convert all the characters in the document body to lower case

4. remove the punctuation and the spaces at the end and beginning of the text; The punctuation marks are substituted with a space in order to not to merge two different words:

   Two.Words -> Two Words

   The excesses of whitespaces are removed by trimming the text for the characters at the extremes and during the tokenisation phase for the characters in the middle of the text.

2.2 Tokenization

The document's body obtained from the text cleaning phase is tokenized by splitting it using a group of one or more spaces as delimiters; in this way we also remove the occurrences of two or more spaces in succession that, during the splitting, generate the empty character in the list of tokens.

2.3 Stopword removal and stemming

When the inverted index module is executed, it is possible to set the flag to apply the *stemming and stopwords removal* to the set of tokens. We've downloaded a file containing a list of english stopwords from the following page:

https://github.com/stopwords-iso/stopwords-en

This list is loaded in main memory as a list of strings by the Parser. From the array of tokens obtained in the previous phase are removed all the tokens that appear also in the list of stopwords.
The tokens are then stemmed using the Porter Stemmer exploiting the following library: *opennlp.tools.stemmer.PorterStemmer*.

Chapter 3

Indexing

3.1 SPIMI

The first algorithm to use to create all the necessary structures is the Single Pass In Memory Indexing (or SPIMI). During this phase the collection is read one document at a time and the intermediate blocks are created. Every time the memory used exceeds a threshold the following files are written to disk: one for the posting list's **frequencies**, one for the posting list's **docIds** and one for the **partial lexicon** sorted in alphabetical order. The document file is saved on disk and it updated every time a document is analyzed.

3.2 Merging

Once the partial structures are created the *Merge algorithm* takes care of merging the intermediate blocks.
We've implemented two **k-way merge** algorithms: the first to merge each sorted *lexicon block* to obtain the final lexicon and the second to merge the intermediate posting list blocks exploiting the fact that each posting list partial block is ordered in an increasingly docid way.
In particular it analyzes one term at a time the blocks created before, taking care of merging its inverted indexes: this is done by simply appending each part of the posting list for the term in question, accessing the blocks in the order in which they were created, since in each block, for each term, the docid are sorted and each one appears in only one block.
The posting list of the term is then written to disk in two files (in *Files/docids.txt* and *Files/frequencies.txt*, possibly in compressed form) and its lexicon entry is also written in the lexicon file (Files/lexicon.txt).
The generic lexicon entry (in this project TermInfo) is composed as follows:

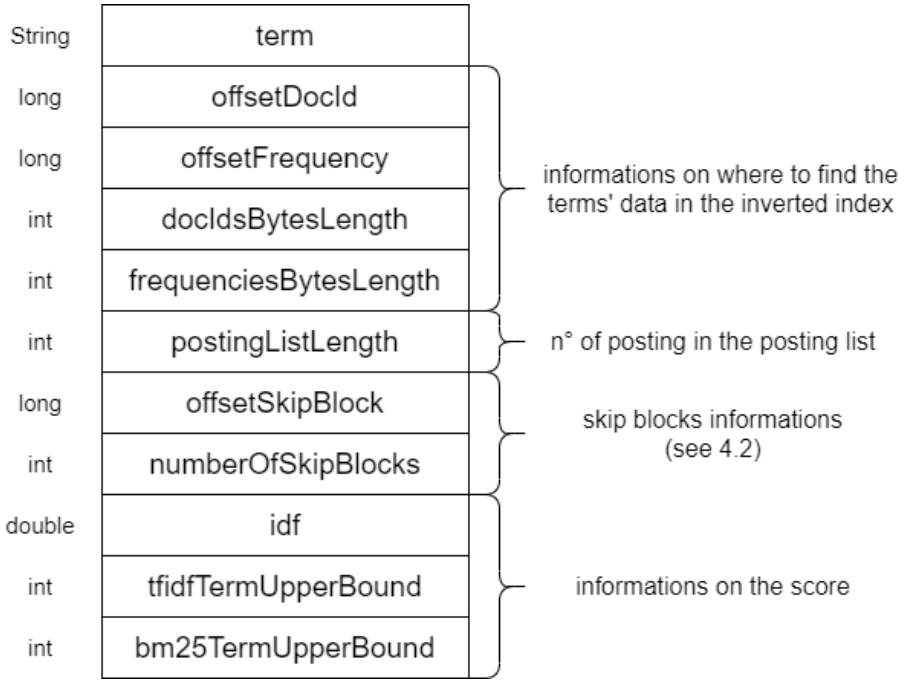

Figure 3.1: TermInfo structure

3.3 Skip Blocks

To improve the performances of the score computation, a structure called skip block is introduced. It is created during the merging phase and keeps information to enable faster traversing of the posting list. In particular, during merge, it divides every posting list in $\sqrt{n}$ blocks, where n is the length of the posting list, encoding in every block information on where to find the relative data and the max doc id present in it. The skip block structure is the following:

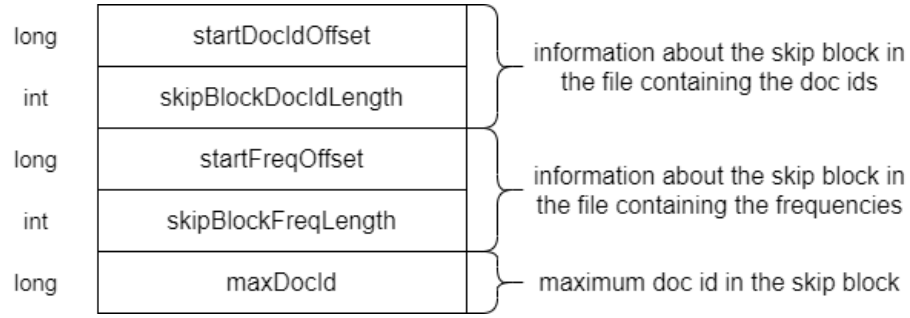

Figure 3.2: SkipBlock structure

The skip blocks for each term are written to the *skipblocks.txt* file when the merge process for a term is complete.

3.4 Compression

The merging process can be performed using compression. In this case, when the posting list of the term is merged, then the docids and the frequencies can be written to disk in a compressed format. We've chosen to use the **Variable Byte Encoding** algorithm for both.
In the table 3.1 we can see how compression affects the size of the inverted index files. Compression also has a positive effect on the speed of the merge process.

Inverted Index settings	size docids (MB)	size docids (MB)	time
-sc	654	173	1h 44min 20s
-c	1309	347	5h 42min 19s[1]
-s	1391	695	3h 9min 10s
default	2783	1391	8h 12min 34s

Table 3.1: The size and the indexing time of the inverted index files for each of the different indexing configurations

[1]During this indexing, the computer went into hibernation, thus increasing the time required to complete.

Chapter 4

Query Processing

After all the necessary data are prepared it is possible to execute the queries. The application will present the user a textual GUI in which he/she can select the configuration to use for the query. The options that can be used are:

- ***scoring function***: user can choose between using *TFIDF* or *BM25*
- ***query type***: user can query the system using the *disjunctive* or *conjunctive* mode
- ***debug mode***: user can enable *debug* mode

The first thing that the user can see is the time used by the application to execute the query.
The results are then showed to the user highlighting the position in the ranking (under the sign #), the DocNo (under the column denoted with the string DOCNO) and the score obtained for this document (below the string SCORE)

4.1 Document scoring

For what regards the scoring there are two different type of scoring function that can be used

4.1.1 TFIDF

In this case the score of the document is computed as:

$$score(q, d) = \sum_{t_i} (1 + \log(tf_i(d))) \log \frac{N}{n_i}$$

Where $\mathbf{tf_i(d)}$ is the frequency of the term t_i in the document d, $\mathbf{N}$ is the number of documents in the collection and $\mathbf{n_i}$ is the number of documents containing the term t_i.

4.1.2 BM25

This scoring function is based on **Okapi BM25**:

$$score(q,d) = \sum_{t_i} \frac{tf_i(d)}{k_1\left((1-b)+b\frac{dl(d)}{avdl}\right)+tf_i(d)} * \log\frac{N}{n_i}$$

Where **q** is the query from the user, **d** is the document that is being analyzed, $\mathbf{tf_i(d)}$ is the frequency of the term t_i in the document d, $\mathbf{k_1}$, **b** are parameters (in this project $k_1 = 1.5$ and $b = 0.75$), **dl(d)** is the document length, **avdl** is the average document length in the collection, **N** is the number of documents in the collection and $\mathbf{n_i}$ is the number of documents containing the term t_i.

4.2 Using Skip Blocks: NextGEQ and next

Using the *skip blocks* created during the indexing phase, it's possible to traverse the posting list of each term in a faster way. One of the main advantages of using skip blocks is the fact that they allow you to keep only part of the posting list in memory, and to load other parts into memory only when they're needed. These advantages are best exploited when the NextGEQ method (in the conjunctive case) is used, which takes a docid d as input. The application skips all the postings inside a part of the posting list described by a skip block with a maximum docid lower than the desired, looking for a block where maxDocId $\geq$ d. The part of the posting list that contains a docid that satisfies the requirements is loaded into memory, and it is only at this point that the postings are actually accessed to find a docid greater than or equal to d.
As we can see, this solution, called **skipping**, allows to reduce the memory consumption and to speed up the traversal of the posting lists.
The skip blocks are also used in the case of disjunctive queries when the *next* method is called, in this way is possible to load only part of the posting lists in memory.
The better improvements are achieved in combination with compression, which reduces the amount of data moved from disk.

4.3 Dynamic pruning algorithm

To further increase the speed of the scoring process MaxScore was used as pruning algorithm. For every term two new variables are introduced, one to handle the TFIDF case, one for the BM25 one.
In both cases an additional variable is used, named **idf**, calculated as:

$$idf = \log\frac{N}{n_i}$$

The **idf**, the **TFIDF term upper bound** and the **BM25 term upper bound** are calculated during the indexing phase. In the trade-off we've chosen to increase the indexing time to reduce the scoring time.

4.3.1 Term upper bound for TFIDF

For every term t_i a new variable is introduced, **tfidfTermUpperBound**:

$$tfidfTermUpperBound = (1 + \log(maxTf_i)) * idf$$

where $\mathbf{maxTf_i}$ represent the max document frequency of term t_i in the collection

4.3.2 Term upper bound for BM25

Similarly, in this case, for every t_i, **bm25TermUpperBound** is calculated, exploiting the general BM25 formula:

$$\frac{tf_i(d)}{k_1 \left((1-b) + b\frac{dl(d)}{avdl} \right) + tf_i(d)} * idf$$

This is calculated for every document in the posting list of a term and only the highest bm25 score is kept and saved in the above mentioned variable.

4.3.3 MaxScore

The chosen dynamic pruning algorithm is **MaxScore** and it's used in both *conjunctive* and *disjunctive* cases.
During the scoring of a document d, we sort in ascending order each posting list whose current docid is d according to its current term upper bound. The list is divided into **essential** and **non-essential** lists: the *essential* are the posting lists, starting from the list with the maximum term upper bound, whose sum reaches the threshold required to enter in the top k *priority queue*, the other posting lists are considered non-essential. The posting lists are traversed using a **DAAT** algorithm: if a docid doesn't appear in at least one of the essential posting lists, it has no chance of being in the top k documents and its scoring is skipped (document pruning). This approach only makes sense in the disjunctive case: in the conjunctive, we score documents that appear in all the query term's posting lists, an essential is always present.
The term upper bounds are used also to prune the document during its scoring: if the *partial score* added to the *sum* of the *remaining* posting lists term upper bounds is lower than the *threshold* to enter in the top k documents, then the scoring is stopped and the document is pruned (this method is used for both conjunctive and disjunctive queries).

Chapter 5

Performance

We've evaluated the performance of our search engine using a set of queries downloaded from the same website from which we downloaded the collection:

https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020

The file we've downloaded is called Test(2020). It's a *tsv* file with two attributes: the **qid** (query id) and the **query** itself.
The qrel file is a txt file, it is associated with the same set of queries in the Test(2020) file and contains the following fields: **qid**, **q0** (constant for all the results), **docno** and **rating** in terms of the relevance of the docno with respect to the query.

The program in the *query-evaluation* module reads each line of the file, exectues the query, stores the completion time of the ranked retrieval (top 1000 documents) and appends to the *query_results* file a line for each retrieved document composed as follows (q0 and runid are strings with the same value for each query):

qid q0 docid rank score runid

In table 5.1 we can see the average completion time obtained by executing the **200** queries in the test set for different configurations. As mentioned before, **-sc** stands for *stemming and stopwords removal and compression*, while **-c** stands for *compression only*, so the stopwords are kept and no stemming is applied.

Inverted Index settings	BM25 (ms)	TFIDF (ms)
-sc disjunctive	18.27	10.245
-sc conjunctive	5.785	5.725
-c disjunctive	329.7	302.04
-c conjunctive	115.5	132.435

Table 5.1: Average search engine's completion time for different configurations

To evaluate the results of the queries, we've also downloaded the trec_eval tool for the following website:

https://trec.nist.gov/trec_eval/

The program takes as input the **qrel** file and the **query_results** file and computes several metrics, in table 5.2 and table 5.3 we can see the results (the tables are split for clarity).

Inverted Index settings	map	MRR	p@5	p@10
-sc disjunctive (BM25)	0.3568	0.7759	0.5926	0.5500
-sc disjunctive (TFIDF)	0.2015	0.6634	0.4667	0.4185
-sc conjunctive (BM25)	0.1323	0.4191	0.2333	0.1722
-sc conjunctive (TFIDF)	0.1286	0.5157	0.2444	0.1889
-c disjunctive (BM25)	0.2980	0.8284	0.6370	0.5593
-c disjunctive (TFIDF)	0.1627	0.5744	0.4296	0.4093

Table 5.2: Search engine's performances obtained from the trec_eval tool

Inverted Index settings	iP@0.00	iP@0.10	iP@0.20
-sc disjunctive (BM25)	0.8325	0.6463	0.5684
-sc disjunctive (TFIDF)	0.6935	0.4760	0.3719
-sc conjunctive (BM25)	0.4873	0.2262	0.2150
-sc conjunctive (TFIDF)	0.5504	0.2183	0.2171
-c disjunctive (BM25)	0.8474	0.6612	0.5132
-c disjunctive (TFIDF)	0.6258	0.4283	0.3120

Table 5.3: Search engine's performances obtained from the trec_eval tool

The files containing the results of the evaluation script and the trec_eval results can be seen at the following link:

https://github.com/Fabi8997/mircv-project/tree/main/Docs

The file containing the indexes for each configuration can be downloaded from this link.

Chapter 6

Conclusions

6.1 Major functions and classes

The major function of the project are the following:

- *it/unipi/mircv/parser/Parser.processDocument*: it parses the document in input, tokenises it and applies the pre-process discussed in Chapter 2.
- *it/unipi/mircv/builder/InvertedIndexBuilder.insertDocument*: It inserts the document's tokens into the lexicon and updates the inverted index data structures, it's part of the implementation of the SPIMI algorithm.
- *it/unipi/mircv/compressor/Compressor*: class containing the methods to compress/decompress a list of integers using the Variable Byte Encoding.
- *it/unipi/mircv/Indexer*: the main class of the *inverted-index* module, it handles all the aspects discussed in Chapter 3. The implementation of these aspects is grouped together to make better use of the **single pass strategy**.
- *it/unipi/mircv/EvaluateQueries*: main class for the query evaluation module, it provides the methods for loading a batch of queries, which are used to generate a file containing the results of the queries in a format that can be used to calculate the metrics using the trec_eval tool.
- *it/unipi/mircv/scoring/Score*: class containing the implementation of the scoring process using the DAAT algorithm.
- *it/unipi/mircv/Main*: class in which is implemented the Demo Interface.
- *it/unipi/mircv/beans/PostingList*: It implements the interface provided to operate on a posting list, the main methods are *OpenList(termInfo)*, *next()*, *nextGEQ(docid)*, *closeList()*, *hasNext()*, *getDocId()*, *getFreq()* and *getTermInfo()*. All aspects such as decompression, loading and changing the skip blocks are handled internally.

6.2 Limits

In the following list we will explain some of the limits of our project:

- We've used the *same* algorithm for integer compression for docids and frequencies, we may get better results using unary coding for frequencies, since many of them are numbers close to 1.

- At the start of the query processing program, a few minutes are required to load the data structure into memory; the slower part is loading the document index, due to the limited memory allocated to the JVM.

- The execution of queries in the default indexing configuration (no compression and no stemming and stopword filtering) is too slow, probably due to the limitation of the memory allocated to the JVM; in fact, the structures in memory in the case of no compression are larger than those with compression, causing the JVM to free memory frequently.