University of Pisa

Department of Information Engineering
Artificial Intelligence and Data Engineering

Multimedia Information Retrieval & Computer Vision course

Textual Search Engine

Di Donato Mattia, Giorgi Matteo

Summary

1 Introduction

This project is dedicated to the development of a search engine capable of performing text retrieval across an extensive collection comprising 8.8 million documents. The project is organized in two main phases:

1. **Document Indexing:** in this initial phase, we focus on the creation of essential data structures required for efficient retrieval.
2. **Query Processing:** this second phase involves the actual retrieval of pertinent documents based on user-provided queries.

After the explanation of these two steps in more detail, we will carefully analyse how well they work. We'll evaluate disk occupancy and indexing time for the first step. In the second step, we'll evaluate efficiency and effectiveness of our search engine.

The collection of documents used for the creation of the index and the queries used for testing the performance can be found here.

The source code for this project is openly available on GitHub via the following link: TextualSearchEngine.

2 Project structure and main modules

The Java project is structured around several main modules:

- **Index:** This module is responsible for carrying out the indexing process, in which lexicon, inverted index, document index structures are generated.
- **Prompt:** This module acts as a user interface to access the search engine. It allows to query the system and to evaluate it by submitting a batch of test queries.

2.1 Index

This module is composed by different packages but the most important are the structures and the algorithms ones that contains, respectively all the classes for the realization and manipulation of inverted index, and the Spimi and Merger algorithm used for the processing indexing of document collection. This module also contains the class used for the pre-processing of documents and queries.

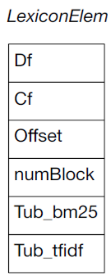
2.1.1 Preprocessing

Before building the index for our search engine, a preprocessing step is performed on the entire collection of documents. This preprocessing is intended to improve the quality and efficiency of the indexing process. It involves the systematic removal of stop-words, words containing symbols, and stand-alone symbols from each word within documents. By eliminating these elements, we ensure that the resulting index is more focused on meaningful content, thus improving the accuracy and relevance of search results.

2.1.2 Structures

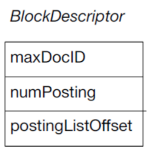
The foundation of our search engine's indexing process lies in the creation of several essential data structures. These structures collectively enable efficient information retrieval from the extensive document collection.

- **Lexicon**: it is a central component that collects all the unique terms encountered in the processed documents. Each term is associated with essential information that describes the term itself within the processed collection of documents.
- **LexiconElem**: each term with its related information forms an object called LexiconElem.



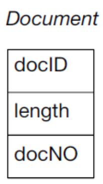
- *Df:* document frequency.
- *Cf:* collection frequency.
- *Offset:* location in the file.
- *numBlock:* number of blocks into which the PostingList is divided.
- *Tub_bm25:* tub obtained with bm25 score.
- *Tub_tfid:* tub obtained with TFIDF score.

- **PostingList**: for each term, a list of objects is created containing information regarding the documents in which the term appears and its relative frequency. Each posting list is divided into blocks thanks to the BlockDescription structure analysed subsequently. The number of Postings in each block is variable and is calculated through the root of the total length of the PostingList. If the PostingList has less than 1024 Postings, then no division into blocks is made.
- **Posting**: this object stores information relating to the document and its frequency within it.
- **BlockDescriptor**: it is a fundamental object that holds the necessary information when searching for a term in the entire index. It allows you to implement the search algorithm efficiently by using block loading of the posting lists.



- *maxDocID:* is the max ID contained into the block.
- *numPosting:* is the number of postings contained into the block.
- *postingListOffset:* location in the file.

- **Document**: structure that stores information relating to a processed document.



- *docID:* is the ID of the document.
- *lenght:* is the of term into the document.
- *docNO:* identifying title of the document.

2.1.3 Spimi

To create a robust and efficient index for our search engine, we implemented the SPIMI (Single Pass In-Memory Indexing) algorithm. SPIMI is renowned for its excellent handling of large document collections while significantly reducing memory overhead.

SPIMI proceeds by processing documents one at a time, transforming them into words, and performing preprocessing tasks such as eliminating stopwords, filtering words with symbols, and removing stand-alone symbols. These cleaned and refined terms are then organized into a data structure called a Lexicon.

One of the main advantages of SPIMI is its ability to minimize memory usage. This is done by partially processing the collection of documents, thus creating partial index structures. From time to time these partial structures are saved to disk to free up memory. Once this is done, the processing of the documents continues until they are finished. What is then produced by SPIMI will be a collection of partial index structures which will subsequently be merged by the Merger.

2.1.4 Merger
This component already mentioned previously is responsible for completing the processing started by SPIMI. Specifically, it collects the partial files that structure the index and combines them, creating unique files for the lexicon and posting lists.

2.2 Prompt
The core of this module is the class Searcher which provides the DAAT (Document At A Time) and MaxScore methods. Among the other functionalities, there is the evaluation package which allows to perform an effectiveness evaluation of the search engine by also using the line command trec_eval. Moreover, there is a functionality, dedicated to lexicon and document expansion, for the computation of TUB and DUB used in the searching process if the dynamic pruning option is active.

2.2.1 Term Upper Bound & Document Upper Bound
TUBs and DUBs are precomputed only if dynamic pruning is used, otherwise they are not necessary for the search engine. A TUB is related to a term, and it is the maximum score a single query term can obtain. On the other side a DUB is related to a document, and it is the sum of all TUBs of the terms contained in the document.

2.2.2 Searcher
In this class are contained the main methods for document scoring and retrieval. Before searching the terms in the lexicon, a query object is created, in this way the query is pre-processed as our document collection and then submitted to the search engine by using one of the two following methos.

2.2.2.1 DAAT
This method checks the presence of each query terms in the lexicon and load in memory each related posting list. The loading of the posting list is the same in the MaxScore, it is performed by loading all the blockDescriptors of each term, from which the posting list offsets and dimensions are retrieved and then for each term is loaded the first posting list block. The posting lists are all open at the same time and the least docid is retrived by calling the *getNextDocIdDAAT*. The scoring function, for the least docid, is computed for all the terms that contains that document in their posting. The document with the relative scoring is added to the query results. When the posting lists are completely fetched the results are sorted by the scoring value and limited to the top K results.

2.2.2.2 MaxScore
The MaxScore function is a bit more complicated than the DAAT one and it is based on the concept of essential and non-essential posting list, which is given by the TUB (term upper bound) and the current threshold. The loaded posting lists are ordered by decreasing value of TUB. The first essential posting list is calculated by summing the TUBs until they become greater or equal to the current threshold. Initially the current threshold is set to 0 and in this way all the posting lists are essential. When the query results are fulfilled with K results the current threshold is becoming equal to the least value of scoring in the query results.

The essential posting lists are processed in a DAAT manner, the least docid is processed only if its DUB is greater or equal to the current threshold, otherwise it is skipped. The scoring of a document is computed by using the following functions:

- *computeEssentialPS*: used to calculate the partial score and update the iterator of the essential posting list.
- *sumNonEssentialTUBs*: used to add the TUBs of the non-essential posting lists to the partial score to obtain the actual document upper bound if it is less or equal to the current threshold the computation is interrupted and a new docid is processed.

- *computeDUB*: used to calculate the actual document upper bound and partial, score processing the non-essential posting lists from the most to the least significative (respect to TUBs order).

At the end if the partial score is greater than the threshold the docid is added to the query results, they are sorted in decreasing order and the current threshold and essential posting lists are updated.

2.2.2.3 TFIDIF

It is a simple but effective scoring function based on the term frequency and the inverse document frequency.

$$TFIDF = (1 + \log(tf)) * \log\left(\frac{N\_docs}{df}\right)$$

Where:

- tf: represents the term frequency in the current document.
- df: represents the document frequency, which is the number of documents that contain the term.
- N_docs: represents the total number of documents in the corpus.

2.2.2.4 BM25

This is a more sophisticated scoring function which take in consideration also the document length and the average document length.

$$BM25 = \left(\frac{tf}{(k1 * B + tf)}\right) * idf$$

With:

$$B = ((1 - b) + b * \left(\frac{docLength}{avgDocLength}\right))$$

$$idf = \log\left(\frac{N\_docs}{df}\right)$$

Where:

- k1: is a constant (usually set to 1.2).
- b: is a constant (usually set to 0.75).
- B: is a term that accounts for document length normalization
- N_docs: is the total number of documents in the collection.

2.2.3 Evaluation and TrecEval

The evaluator is a class contained in trec_eval package which is devoted to the execution of test queries, the processing time are recorded by the log to obtain an avg execution time and the query results are written in a file according to trec eval format, ready to be evaluated with the relevant results using the command-line tool:

```
trec_eval -q -c qrels.test results.test
```

Where:

- q: indicates whether to output results for individual queries as well as the averages over all queries.
- c: used if results.test doesn't have results for all queries, this flag ensures correct evaluation.

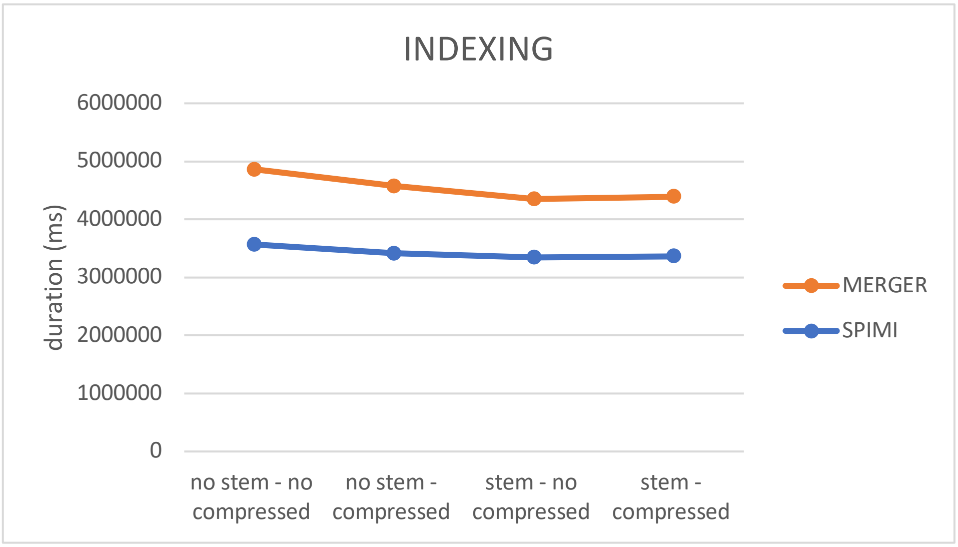
3 Performance

We conducted several experiments on our search engine to evaluate the achieved performance. The tests were performed across various scenarios based on the combination of available parameters.

3.1 Indexing

Options	Spimi duration (mm:ss)	Merger duration (mm:ss)	Index size (GB)
Not compressed not stemmed	59:28	22:35	1.406
Compressed not stemmed	56:57	19:19	
Stemmed not compressed	56:44	17:47	1.356
Compressed and stemmed	56:03	17:10	

Compression noticeably improves both Spimi and Merger durations, leading to reduced processing times for these phases. Additionally, the implementation of stemming demonstrates advantages in terms of shorter durations and contributing also to a reduction in the index size when compared to the non-compressed option.



If we want to use the dynamic option, enabling the query processing using the MaxScore algorithm, we must spend more time in indexing, adding TUBs and DUBs values for TFIDF and BM25. More precisely about 15 minutes for TUBs processing and 58 minutes for DUBs processing, leading also to an increase of indexing structures, in particular documents.bin increase of 36 MB and lexicon.bin increase of 4 MB.

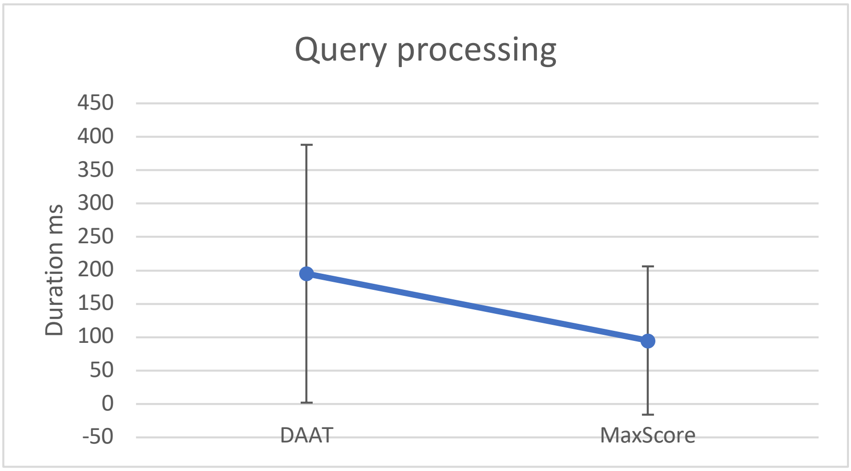
3.2 Query processing & trec_eval

We conducted experiments to evaluate query processing using the query collection available at this link. The collection consists of 101093 different queries that we submitted to our search engine. We submit disjunctive queries, asking to our search engine for the top 100 results. Then we compared the scoring algorithms, the scoring functions, and the use of the stemmer for pre-processing.

3.3 DAAT vs MaxScore

Scoring Algorithm	Mean and STD Response time (ms)
DAAT	195 ± 193
MaxScore	95 ± 111

The results show that MaxScore algorithm is significantly faster than DAAT, with both the average response time and standard deviation being roughly half compared to DAAT.



However, it's important to note that both algorithms have significant variations in response times, as indicated by their high standard deviations and this is caused by the length of the query terms and the length of the associated posting lists.

3.4 TFIDF vs BM25 without Stemming

Scoring Function	Mean and STD Response time (ms)	MAP	Rprec	P@5
TFIDF	64 ± 78	0.1256	0.0663	0.0381
BM25	56 ± 67	0.1795	0.0970	0.0555

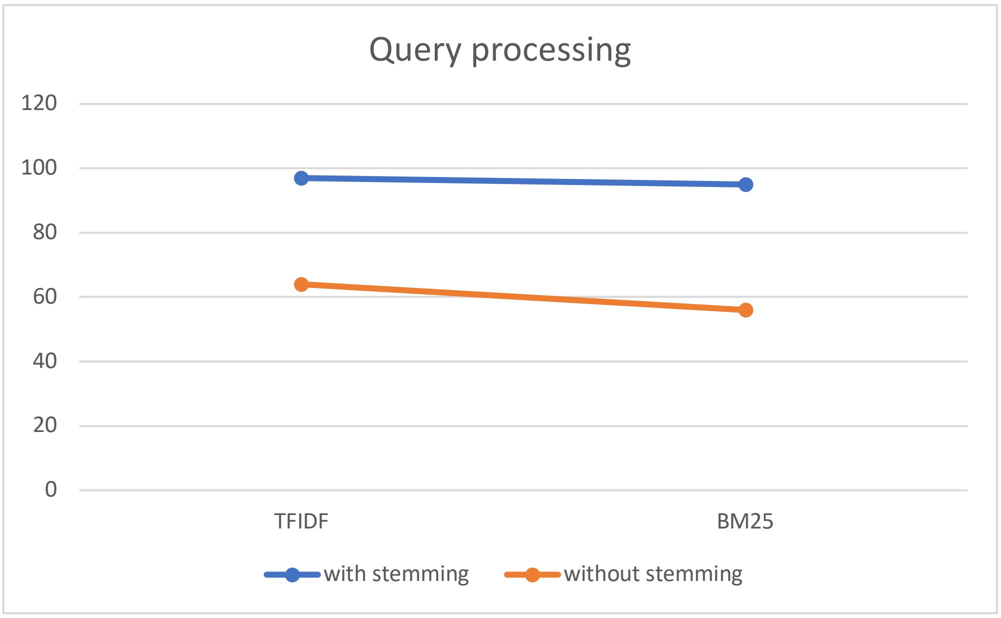
Both TFIDF and BM25 exhibit similar mean response times. However, BM25 appears to be the superior scoring function offering slightly faster response times on average and consistently provides higher-quality search results, as indicated by its higher MAP, Rprec, and P@5 values.

3.5 TFIDF vs BM25 with Stemming

Scoring Function	Mean and STD Response time (ms)	MAP	Rprec	P@5
TFIDF	97 ± 108	0.1260	0.0656	0.0379
BM25	95 ± 111	0.1842	0.0976	0.0574

Similarly, to the results obtained without stemming, also in this case BM25 outperform the TFIDF scoring function.

For TFIDF, stemming has minimal impact on search evaluation metrics, but it results in a slightly higher average response time. For BM25, stemming also has limited effects on search evaluation metrics, but in the case without stemming, it significantly reduces the response time while maintaining similar performance in search quality.



BM25 without stemming stands out with significantly faster response times, making it a potentially more efficient choice for our search engine. However, if we want to obtain a more precise response, we should use stemming.

4 Future Improvements

One of the key enhancements we could introduce is caching. There are two possible level of caching: one involves storing search results associated with a list of query terms, which can significantly improve the response time of the search engine, the other involve the in-memory storage of posting lists to avoid the need for disk access for frequently occurring terms.

Another significant improvement involves the compression of the search index. This process aims to reduce the size of the index, making it more easily to be stored in-memory. The index should be divided into two different files: docIds and termFrequencies. Each of these components could be compressed using suitable compression techniques. For frequencies, considering that these values are typically small, it is possible to leverage the distribution of values and apply unary compression to minimize storage space. For docIds, especially as it exhibits an incremental pattern as in our case, an effective compression algorithm could be the gamma compression.