Artificial Intelligence and Data Engineering

Multimedia Information Retrieval and Computer Vision

Search Engine

Project Documentation

TEAM MEMBERS:
Elisa De Filomeno
Massimo Merla
Riccardo Orrù

Contents

1 Introduction

The aim of this project is to develop a search engine capable of performing text retrieval tasks on a large-scale dataset. The dataset contains 8.8 million documents and can be found in the [1]. The data is used to build an inverted index data structure, enabling efficient document retrieval based on user queries.

The source code for this project is available on GitHub at the following link:
https://github.com/ohmissam/MIRCV_project_2024.

1.1 Project Structure and Modules

The project is composed of three distinct modules, each responsible for a specific part of the search engine's functionality:

- **components:** this module is responsible for preprocessing the collection and managing the indexing phase. It constructs the inverted index and the data structures needed for the information retrieval system.
- **query_execution:** this module provides an interface for users to submit search queries and receive a ranked list of the top 20 relevant documents. It also allows users to access *settings* options to modify query modes (*conjunctive* or *disjunctive*), specify the scoring function (*BM25* or *tf-idf*), and toggle debugging mode.
- **evaluation:** this module is responsible for evaluating the search engine. It processes a batch of queries (200 queries) and measures the performance of the search engine. It calculates the average time to process a query under different indexing configurations. The evaluation results are output in the required format for use with the `trec_eval`[4] tool, which is useful for computing additional metrics for evaluation.

2 Pre-processing

The pre-processing phase is essential for preparing documents for the indexing and query processing stages. Initially, the collection file is decompressed, resulting in a tar archive containing the text documents. Each document is stored on a separate line, and the file is read using UNICODE encoding.

The Java class responsible for processing each line is:
it.unipi.dii.aide.mircv.preProcessing.DocumentPreProcessor

This class manages the pre-processing of documents, including tokenization, cleaning, and optional stemming and stopword removal. Firstly, each line is split using `\t` to extract the `pid` and the document body.

The following steps are performed to prepare each document:

- **Skip malformed documents**, such as those missing the *pid* or the document body.
- **Convert all characters to lowercase** in the document body.
- **Remove punctuation marks** by replacing them with a whitespace character. Then, trim the string to **remove leading and trailing whitespaces**.
- **Tokenize the document body** by splitting it based on one or more spaces. This step also removes consecutive spaces, which could otherwise result in empty tokens during splitting.
- **Apply optional Stemming and Stopword Removal**: Common English stopwords are removed using a file containing a list of stopwords available at [3]. The remaining tokens are then stemmed using the Porter Stemmer algorithm, which reduces words to their root forms.

3 Indexing

In this process essential data structures are built: the inverted index, the lexicon, and the document index.

There are two main phases:

1. the Single-Pass In-Memory Indexing (**SPIMI**) algorithm,
2. the **merging** phase.

3.1 SPIMI

The Single-Pass In-Memory Indexing (SPIMI) algorithm reads the documents sequentially from the collection, processes them, and creates partial lexicon and partial inverted indexes, saving the posting lists docIds and frequencies in two separate files. These partial indexes are stored on disk when memory usage exceeds a specified threshold. We decided to use 70% of the available memory as threshold. In particular, after the memory exceeds the threshold, a block is created and three files are written to disk for each block:

- **docIds file** containing the docIDs of the posting lists in the partial inverted index.
- **frequencies file** containing the frequencies of the posting lists in the partial inverted index.
- **partial lexicon file** sorted in alphabetical order.

A **document index file** is created and saved to disk, which stores information about each document.

Each document is assigned an incremental docID, starting from 1 and increasing with the number of documents processed, it correspond to: docID = docNo+1

The class responsible for the inverted index construction is:
it.unipi.dii.aide.mircv.builder.InvertedIndexBuilder which is called inside the
it.unipi.dii.aide.mircv.MainComponents class.

3.2 Merging

Once the SPIMI phase completes, the merging process begins. This phase combines all the partial inverted indexes and lexicons generated in the SPIMI phase into a final, complete index.

The class responsible for the merging is:
it.unipi.dii.aide.mircv.merger.IndexMerger

The merging algorithm operates in two stages:

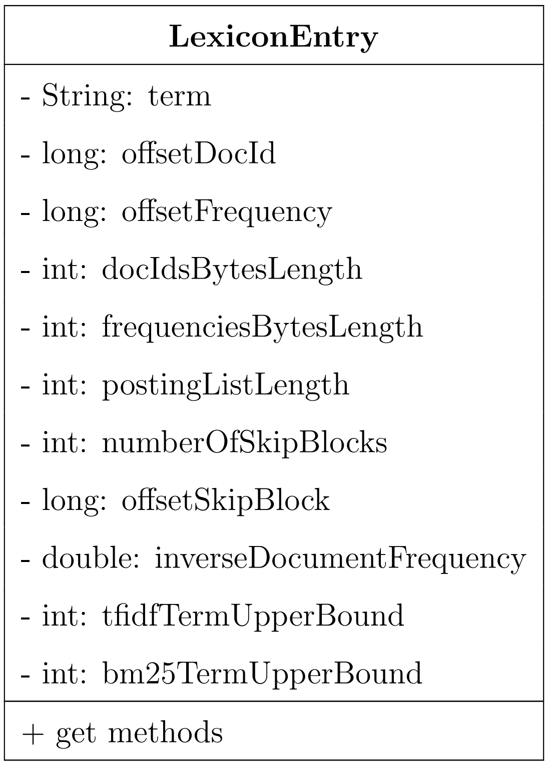
1. **Merging the Lexicons:** The lexicons are merged using a **k-way merge** algorithm, where each sorted partial lexicon is combined into one final lexicon.
2. **Merging the Posting Lists:** The posting lists for each term are merged by iterating through each partial index in order, one term at a time, using **k-way merge** algorithm. Since the docIDs are already sorted within each partial index, this step involves concatenating the posting lists. The merged posting list is then written to

disk, storing docIDs and term frequencies in separate files, inside *data* directory of the project. There is an optional option to store them in a compressed way.

The final files written during the merging process are:

- `InvertedIndexDocIds.txt`: Contains all the docIDs.
- `InvertedIndexFrequencies.txt`: Contains the corresponding frequencies.
- `Lexicon.txt`: Contains the final lexicon with term entries.

In the lexicon data structure each lexicon entry is represented by the java class *it.unipi.dii.aide.mircv.model.LexiconEntry*, composed as the following picture:



LexiconEntry

- String: term
- long: offsetDocId
- long: offsetFrequency
- int: docIdsBytesLength
- int: frequenciesBytesLength
- int: postingListLength
- int: numberOfSkipBlocks
- long: offsetSkipBlock
- double: inverseDocumentFrequency
- int: tfidfTermUpperBound
- int: bm25TermUpperBound

+ get methods

Figure 1: UML Diagram for LexiconEntry Class

Each lexicon entry contains the *term* it is referring to, the offset to the inverted index (*offsetDocId* and *offsetFrequency*) with the number of bytes occupied (*docIdsBytesLength* and *frequenciesBytesLength*). It also includes the length of the associated posting list. Posting lists are divided into skip blocks, so the class contains additional information for the *Skip Block*. Lastly, it contains the *idf score* and two upper bounds scores.

3.3 Skip Blocks

To improve the efficiency of query processing, *skip blocks* are introduced during the merging phase. A *skip block* is a structure that enables faster traversal of posting lists.

During the merging process, the posting list associated to each term is divided into blocks. Each block contains a predefined number of postings, specifically $\sqrt{n}$, where n is the number of postings in the list.

After merging the posting lists for each term, these skip blocks are written to a separate file, always inside the *data* directory, named `SkipBlocks.txt`.

the *SkipBlock* java class is:
it.unipi.dii.aide.mircv.model.SkipBlock, composed as the following picture:

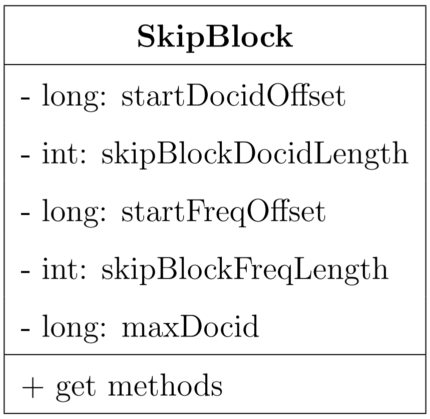


Figure 2: UML Diagram for SkipBlock Class

The *startDocidOffset* specifies the starting offset of the corresponding block of documentIDs. The *skipBlockDocidLength* represents the length of the document ID block, either as the number of postings (if uncompressed) or as the number of bytes (if compressed). Similarly, the *startFreqOffset* indicates the starting offset of the corresponding frequency block, while the *skipBlockFreqLength* denotes its length, either in postings or bytes, depending on compression. Lastly, the *maxDocid* holds the maximum document ID present within the block.

3.4 Compression

The compression is optional, it is useful to reduce the storage space required for the inverted index.

We decided to use the **Variable Byte Encoding** algorithm as the compression algorithm. This algorithm is applied to both the docIds and the frequencies.

4 Query Execution

Once the necessary data was prepared, we proceeded to the query processing phase. The user can use a interface to search a query in the information retrieval system to obtain the top 20 ranked relevant documents, with their score. The user can also change the *settings*.

In the *settings* option the user can:

- **change the scoring function** between TFIDF or BM25.
- **change the query mode**, either the disjunctive or conjunctive query mode.
- **toggle the Debug Mode flag** to see more info of the execution.

Upon submission of a query, the system will display the time taken to process the query.

4.1 Document Scoring

There are two available scoring functions the user can choose to use: **TFIDF** and **BM25**.

4.1.1 TFIDF Scoring

The forumla used to compute the TFIDF scoring is:

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

where q is the query of the user, $tf_i(d)$ is the term frequency of term t_i in document d, N is the total number of documents in the collection, and n_i is the number of documents containing the term t_i.

4.1.2 BM25 Scoring

The formula for the BM25 score is:

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

where $k_1 = 1.5$ and $b = 0.75$ are the values chosen for the project, $dl(d)$ is the length of document d, $avdl$ is the average document length, and the other variables are as defined for TFIDF.

4.2 Efficient Query Traversal with Skip Blocks

Skip blocks are a key optimization that enables faster traversal of posting lists.

In the context of **conjunctive queries** the ***NextGEQ*** method is employed. In this method, a docid d is provided as input, and the algorithm skips over parts of the posting list where the maximum docid is less than the desired d, loading only the block where *maxDocId* $\geq d$. The relevant part of the posting list is then loaded into memory, and the algorithm continues by accessing only the postings of this part of posting list.

The skip blocks are also used in **disjunctive queries** when the ***next*** method is invoked, in this way only a part of the posting list is loaded in memory, saving memory and improving the performance.

5 MaxScore: Dynamic Pruning algorithm

The **MaxScore pruning algorithm** is employed. The goal of this algorithm is to reduce unnecessary scoring by pruning documents that cannot possibly be in the top k results.

5.1 Term Upper Bounds for Scoring

For both TFIDF and BM25, their upper bounds are calculated and stored in the lexicon during the indexing phase. These bounds are then used to prune documents during the query scoring phase, improving the overall efficiency.

5.1.1 TFIDF Term Upper Bound

For each term t_i, the formula for the term upper bound for TFIDF is:

$$\text{tfidfTermUpperBound} = (1 + \log(\text{maxTf}_i)) \times \text{idf}$$

where *maxTfi* is the maximum term frequency for t_i across all documents in the collection.

5.1.2 BM25 Term Upper Bound

For BM25, the term upper bound is calculated with the following formula:

$$\text{bm25TermUpperBound} = \frac{tfi(d)}{k_1\left((1-b) + b\frac{dl(d)}{avdl}\right) + tfi(d)} \times \text{idf}$$

For each term t_i the highest score from the document's posting list is retained.

5.2 MaxScore Algorithm

The MaxScore algorithm is implemented in its full form in the *disjunctive* query type. A variant of it is also used in *conjunctive* query types.

The class responsible for thi algorithm is:
it.unipi.mircv.scorer.ScorerConjunctiveAndDisjunctive

During the scoring phase of a document, the posting lists are sorted in ascending order of their term upper bounds. Based on this ordering, lists are classified as **essential** or **non-essential**. The algorithm used to traverse the posting lists id **DAAT**. In the disjunctive case if a document does not appear in any of the essential posting lists, it is skipped, since it's impossible it will appear in the final top 20 ranking. This method is effective in pruning irrelevant documents from the ranking.

Document pruning is performed during the scoring phase. If the accumulated score for a document, considering the partial scores of the remaining posting lists, is below the threshold required to be in the top k documents, the document is pruned from further consideration.

6 Performance

6.1 Indexing

Indexing Type	**Time**	**DocId Size**	**Freq Size**	**Lexicon Size**
default	3h 55min 32s	2715 MB	1359 MB	148 MB
Variable Byte Compression + Stemming and Stopword removal	1h 8min 12s	687 MB	182MB	128 MB
Only Variable Byte Compression	4h 7min 1s	1296 MB	344 MB	156 MB
Only Stemming and Stopword Removal	1h 38min 38s	1444 MB	722 MB	128 MB

Table 2: Comparison of indexing types by size and time.

From the table, we can observe that compression significantly reduces the storage space required for the inverted index, affecting both the DocIDs and the frequencies file sizes. In terms of processing time, there is no substantial difference compared to the default indexing method. When examining the timings, *Stemming and Stopword Removal* improve performance.

6.2 Evaluation

The class responsible for the evaluation phase is:
Evaluation

We used a bunch of queries, a total of 200 queries, downloaded from [1], from the file *Test(2020)*. In the following table, we report the average times with different inverted index settings.

Inverted Index Settings	**BM25 (ms)**	**TF-IDF (ms)**
Compression (Disjunctive)	592.77	602.485
Compression (Conjunctive)	113.21	116.55
Compression + Stemming/Stopword Removal (Disjunctive)	35.625	25.57
Compression + Stemming/Stopword Removal (Conjunctive)	11.96	10.435

Table 3: Performance comparison of inverted index settings by retrieval models (BM25 and TF-IDF).

We can see that the times with *Stemming and Stopword Removal* decrease significantly compared to using only compression. On the other hand, no substantial differences in terms of time are observed between the two metrics.

6.3 Trec Eval Tool

For the evaluation, we used the *Trec Eval Tool*, downloaded from [2]. We decided to compare our IR system with different configurations, reporting the rank-based metrics in the following tables.

The following table shows the results with only *Compression*.

Scoring Settings	**MAP**	**MRR**	**nDCG@10**	**P@5**	**P@10**
Conjunctive + BM25	0.0314	0.4375	0.1478	0.2000	0.1000
Conjunctive + TF-IDF	0.0316	0.3750	0.1427	0.2000	0.1000
Disjunctive + BM25	0.1549	0.7168	0.4267	0.5519	0.4796
Disjunctive + TF-IDF	0.0919	0.5082	0.2742	0.3519	0.3444

Table 4: Performance comparison of scoring settings using various metrics.

The following table shows the results with both *Compression* and *Stemming/Stopword Removal*:

Scoring Settings	**MAP**	**MRR**	**nDCG@10**	**P@5**	**P@10**
Conjunctive + BM25	0.0774	0.5227	0.2393	0.3609	0.2783
Conjunctive + TF-IDF	0.0750	0.4506	0.2254	0.3435	0.2652
Disjunctive + BM25	0.1932	0.8011	0.4648	0.6000	0.5500
Disjunctive + TF-IDF	0.1175	0.6601	0.3389	0.4741	0.4222

Table 5: Performance comparison of scoring settings using various metrics.

7 Limitations and Improvements

- For the compression phase only the Variable Byte Encoding algorithm has been used for both docIDs and frequencies. A valid alternative could have been using Unary encoding for frequencies, as frequency values are generally small.
- To modify the configuration of Compression, Stemming and Stopword Removal, or the Upperbounds, the inverted index must be rebuilt, consuming lots of time.
- A possible improvement is to implement caching systems. For example, it would be useful to cache frequently requested posting lists to reduce access times to the inverted list.

References

[1] Trec deep learning track 2020. https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020, 2020.

[2] National Institute of Standards and Technology (NIST). Trec evaluation.

[3] stopwords-iso. stopwords-en. Accessed: 2025-01-15.

[4] Text REtrieval Conference (TREC). Trec - text retrieval conference. https://trec.nist.gov/. Accessed: 2025-01-13.