Università di Pisa

Master Of Science in Computer Engineering

artificial intelligence and data engineering

Search Engine MIRCV Project

Project members:
Massimo Valentino Caroti
Simone Landi

INDEX

INTRO

In this project, a search engine was developed based on the MSMARCO Passages collection, which is available on https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020. The project consists of three main parts:

• **Inverted Index Creation**: the first part involves creating an inverted index structure from a set of text documents.

• **Query Processing**: the second part focuses on processing queries over the inverted index through a terminal interface.

• **Search Engine Evaluation**: the final part involves evaluating the search engine's performance using standard evaluation tool TrecEval.

The project is implemented in Java, and the source code is available in the following GitHub repository https://github.com/max423/MIRCV-Project

Search Engine Configuration

For the Search Engine configuration, we use the Configuration class, which encapsulates various configuration fields. These parameters are set through a JSON file located in the resources.

< src/main/java/it/unipi/dii/aide/mircv/resources/configuration.json >

In the Configuration.json file, it is possibile to define the parameters for building the inverted index. In addition, query processing options can be specified via the command line through the terminal interface.

Project Organization

The project has been developed in Java, and its structure is organized as follows:

• **Indexer:** It contains the Spimi algorithm and the Merger component responsible for generating the index structure.
• **Compression:** Unary and VariableByte compression techniques have been employed to reduce the size of posting lists.
• **Models:** This section contains all necessary data objects for the implementation.
• **Query:** It handles the execution of ranked queries.
• **Resources:** This section includes the JSON configuration file, files for evaluation, the test collection, and the stopword list.
• **Text Processing:** Stemming and stopword removal operations are implemented.
• **Utils:** This section contains scripts used for evaluation and other utility functions.
• **Testing:** This section incorporates test code designed to validate the correctness of the implementation. The tests cover compression, indexing, query handling, and text processing.

TEXT PROCESSING

All transformations are performed through the usage of regular expressions. The steps we took to process the documents and queries consisted of:

- Preprocessing
- Tokenization
- Stop-word removal
- Stemming

Preprocessing consists of 'cleaning' the text by removing html tags and replacing punctuation signs, strange characters and multiple spaces with a single space and turning the text into lower case.
In the tokenization phase, we first divided the text into tokens according to whitespaces and then truncated tokens longer than 20 bytes.
The next step was to remove the stopwords using a downloaded list. As far as stemming is concerned, we used a library that implements Porter stemming.

INDEXER

Spimi

The system implements the SPIMI (Single Pass In-Memory Indexing) algorithm to create structures for the search engine. Depending on the configuration, the collection is read and processed with Text Processing.
Two HashMaps hold the partial Vocabulary and partial Posting Lists in memory. The documents in the collection are processed one at a time, writing the corresponding document index entry directly to disk. We perform the following procedure for each token in each document:

- If a token is not in the Vocabulary, new entries are created for the Posting List and Vocabulary, calculating the term frequency.
- If a token is already present in the Vocabulary and has been processed in a previous document, we update the related vocabulary entry (collection frequency and document frequency) and insert a new posting in its posting list.

Through MEMORYFREE_THRESHOLD, we ensure that 15% of free memory is maintained during the process. When this threshold is exceeded, the Vocabulary is sorted; and together with the Posting Lists saved to disk in partial files containing the assigned blockNum in the filename. Subsequently, an attempt is made to force the execution of the garbage collector to obtain additional free memory. The process ends when we have processed all the documents in the collection.

RandomAccessFile

For file accesses, we use a HashMap<Index, ArrayList of RandomAccessFiles> called skeleton_RAF. This structure manages the RandomAccessFiles for the various blocks.
Indexes >= 0 refer to partial files generated during Spimi, while index -1 is reserved for final files.

Each ArrayList<RandomAccessFile> contains:
i= 0: Vocabulary
i= 1: Docid (posting list)
i= 2: TermFreq (posting list)
i= 3: collection statistics (only present in index -1)

Merger

Once the entire collection has been processed, the Merger takes care of merging the partial structures to generate the final index.
This process uses a heap-based approach, which is populated with the first term of each partial vocabulary together with the identifier of the partial file (BlockNum). In this way, we can extract <Term, BlockNum> pairs sorted alphabetically based on the term and in increasing order based on the BlockNum.

The process follows the following flow:

- Extract and remove a <Current Term from Partial Vocabulary, BlockNum> pair from the heap (heap.poll)
- Insert a new pair into the heap from that partial vocabulary
- Check if in the first element of the heap (heap.peek()) there is the same term:
- If no, write the term in the final vocabulary and the corresponding posting list in the final posting list, following the compression configuration
- If yes, merge the information of the same term using the Partial Vocabulary and Partial Posting List, these will be written to disk when we will find a different term in the heap

At the end, the partial files are deleted, the Collection Statistics is created and the execution times are recorded in a log file.

DATA STRUCTURES

Posting List

The PostingList class represents a posting list associated with a term.

Type	Name	Byte	Description
String	term	20	Represents the term associated with the posting list
ArrayList<Posting>	postingList	-	A list of posting objects, where each object represents a posting containing information such as the document ID and the frequency of the term in that document.

Posting

The Posting class represents a single posting associated with a term.

Type	Name	Byte	Description
Int	docID	4	Represents the ID of the document in which the term occurs
Int	termFreq	4	Represents the frequency of the term in the corresponding document

VocabularyElem

The VocabularyElem class represents an element in the vocabulary.

Type	Name	Byte	Description
String	term	20	Represents the term in the vocabulary
Int	DocFreq	4	Represents the number of documents in which the term appears
Int	CollFreq	4	Represents the total number of occurrences of the term in the collection
Long	docIdsOffset	8	Represents the offset of the first docId in the document posting list
Long	termFreqOffest	8	Represents the offset of the first termFreq in the term frequency posting list
Int	docIdsLen	4	Represents the length in bytes of the document posting list
Int	termFreqLen	4	Represents the length in bytes of the term frequency posting list
Double	Idf	8	Represents the inverse document frequency of the term

DocumentIndexElem

The DocumentIndexElem class represents a document index element.
The docId is not stored as it is implicitly defined by the position of the entry in the file. The docNo is a string with a fixed, configurable size (20 bytes in the default configuration).

Type	Name	Byte	Description
String	docNo	20	Represents a unique identifier of the document in the collection
Int	docLen	4	Represents the length of the document

CollectionStatistics

The CollectionStatistics class represents the statistics of the collection.

Type	Name	Byte	Description
Int	docCount	4	Represents the total number of documents in the collection
Long	totalLength	8	Represents the total length of the documents in the collection

Configuration

The Configuration class is a configuration class that reads settings from a JSON file.

Type	Name	Description
Boolean	compressionON	Indicates whether compression is enabled while reading
Boolean	stemming_stopwordON	Indicates whether stemming and stopword removal are enabled
Boolean	index_compressionON	Indicates whether index compression is enabled
Boolean	testingON	Indicates whether to use testing collection
Boolean	scoreON	Indicates the type of score used (BM25=true or TFIDF=false)
Boolean	conjunctiveON	Indicates the type of query retrieval (conjunctive=true or disjunctive=false)
String	Collection Compressed_path	Indicates the configurable path of compressed collection
String	Collection Unompressed_path	Indicates the configurable path of uncompressed collection

COMPRESSION

We have opted for the following compression algorithms to reduce the memory usage of the posting lists:

- Variable Byte compression for DocIds
- Unary compression for Term frequencies

Unary

The unary algorithm is suitable when we have low numerical values, which is the reason why we decided to implement it to compress the term frequency of posting lists.
Another factor to consider is the simplicity of the algorithm's encoding and decoding implementation, which simplifies the process of accessing and retrieving information.

x	$U(x)$
1	0
2	10
3	110
4	1110
5	11110
6	111110
7	1111110
8	11111110

Variable Byte

Since unicode compression is only good for small values, we opted for Variable-Byte compression for DocId knowing that these take larger numeric values.
In the Variable-Byte algorithm, the binary representation of x is divided into a set of bytes. Each byte is allocated 7 bits for representing x itself (data bits), and 1 bit (control bit) is employed to indicate the continuation or end of the byte stream. This design allows for a flexible and space-efficient representation of integer values, with the control bit serving as a signal for whether more bytes are needed to represent the entire value or if it concludes the encoding.

Example for $x = 67822$, $bin(67822,17) =$ 10000100011101110

(1) 100 0010001 1101110

(2) xxxxx100 x0010001 x1101110

(3) 00000100 10010001 11101110

QUERY PROCESSING

Before the actual query execution begins the user is asked for the type of query to be executed (conjunctive or disjunctive), the scoring function to be used (BM25 or TFIDF) and the desired number of results. Once the user has entered the query, it is preprocessed by text cleaning, tokenization, stop word removal and stemming. After preprocessing, query execution is handled by initially removing duplicates from the supplied tokens, creating a list without duplicates.
Before starting the search, a check is made in the cache to see if the query has already been made in the current session, so if it is present the results are displayed, otherwise the query is executed: using a binary search, the vocabulary entries of the terms are loaded into memory along with the corresponding posting list. If a posting list is empty, we signal that the token is not present in the vocabulary.
Next, we initialize a priority queue of 'scoreDoc' objects that will contain the top k results.

Conjunctive

In the conjunctive method, two priority queues (scoreDocsDecreasing and scoreDocsIncreasing) are initially created to keep the documents sorted according to the calculated score. Both queues are initialized with a comparator that takes the scores into account, allowing them to be sorted in descending and ascending order respectively. The use of the two queues was necessary in order to more efficiently handle the removal from the descending queue (the one with the top k scores). Having a queue sorted in ascending score order allows us to exploit the poll function and thus obtain the score of the first document (the one corresponding to the k-th in the descending queue). Using the getMinDocID() method, the document with the lowest DocID among the token posting lists in the query is obtained. Document processing is performed until there are no more documents to process.
Since it is a conjunctive query, the document score will be calculated by summing the partial scores obtained from each posting list only if the document is present in all the posting lists for the tokens in the query. This check is implemented by incrementing a variable if the token is present within the document. If, at the end of document processing, the variable 'present' has a value equal to the number of query tokens, the document's score will be calculated by summing the partial scores obtained from each posting list.
The documents and their scores will be added to the priority queues scoreDocsDecreasing and scoreDocsIncreasing. If one of the queues exceeds the desired size k, only the top-k set is retained. At the end of the process, the queue sorted in descending order will be returned.

Disjunctive

The disjunctive method follows a similar approach to the conjunctive method, but with a key difference to handle queries. In particular, there is no present variable check, since disjunctive queries do not require a document to be present in all posting lists, but rather in at least one of them to be considered relevant. The process consists of initializing the priority queues, iterating over the documents, calculating the scores, updating the priority queues and returning the results, all without requiring the document to be present in all the query's posting lists.

PERFORMANCE

This paragraph shows the performance obtained by our search engine. All tests are carried out with the cache disabled.

Indexer Time and Size

As regards indexing, this can take place with or without compression and with or without stopword removal and stemming.
The times of the indexing process and the file sizes of the inverted index and vocabulary are shown below.

Configuration		SPIMI (m)	Merge (m)	Total (m)
Compressed	+ Stemming & Stopword Rem	11,68	2,30	13,98
Compressed	+ No Stemming & Stopword Rem	11,30	3,98	15,30
Uncompressed	+ Stemming & Stopword Rem	11,44	1,16	12,60
Uncompressed	+ No Stemming & Stopword Rem	10,69	1,44	12,13

Table 1: Execution times for each configuration.

Configuration		Vocabulary	Term Freq	DocId	Total
Compressed	+ Stemming & Stopword Rem	76,2	34,1	742,2	1049
Compressed	+ No Stemming & Stopword	88,2	65	1340	1660
Uncompressed	+ Stemming & Stopword Rem	76,2	789	789	1813
Uncompressed	+ No Stemming & Stopword Rem	88,2	1400	1400	3038

Table 2: Files Size for each configuration.

What is achieved with compression is a significant reduction of the frequency file size, but at the cost of a negative change in timing. As one might expect, pre-processing allows us to reduce file size.

Query Time

We will now analyze the average response time using queries from TREC DL 2020 and K=1000. All results are obtained with the cache disabled and performed with stopwords removal and stemming active.

The results with index compression on and off are shown below:

Configuration	Conjunctive		Disjunctive	
	BM25	TFIDF	BM25	TFIDF
Compression ON	59	29	62	32
Compression OFF	50	22	58	28

Table 3: Average response time (ms).

Effectiveness

The evaluation results using a standard collection TREC DL 2020 queries and TREC DL 2020 qrels are shown below:

Configuration	Map@10	nDCG@10	Precision@10	Precision@20	Recall@1000
Conjunctive-BM25	0.1195	0.4119	0.4766	0.3351	0.3040
Conjunctive-TFIDF	0.1228	0.4169	0.4787	0.3277	0.3029
Disjunctive-BM25	0.1393	0.4647	0.5593	0.4852	0.7533
Disjunctive-TFIDF	0.1277	0.4424	0.4416	0.4639	0.7359

Table 4: Evaluation of Search Engine Effectiveness.

LIMITATIONS

- Implementation of skipping and a dynamic pruning algorithm in order to improve query processing efficiency.
- Test of our collection against other preprocessing and compression techniques and algorithms to evaluate whether the performances increase or not.
- Extend the compression to the others data structures.
- BM25 parameters were not evaluated.
- This search engine does not consider the order of the query terms and their proximity within the documents.
- This search engine does not consider synonymous or related terms.