UNIVERSITÀ DI PISA
SCUOLA DI INGEGNERIA

886II
Multimedia Information Retrieval And Computer Vision

Project report

Students
BOVEROUX Laurie (653961)
HIMMICHE Aida (648540)
FABBRI Anna (xxxxxx)

Professor
TONELLOTTO Nicola

Academic year
2022-2023

1 Overview

The program described in this report is a search engine (SE) built based on the document collection provided by TREC(2020) which contains 8.8 millions documents– of which the totality was indexed and made ready to perform queries on, then tested using the query examples provided by TREC as well.

Moreover, the demo application offers flexible functionalities to a user by allowing them to choose a number of parameters before building the indexes used in the SE, and provides two different scoring functions (TF-IDF and BM25 Okapi) to process queries.

2 Program Features

2.1 Indexing

The index is the data structure storing information about the terms, their appearance in documents and other related information. The inverted index more specifically is an implementation of said data structure which associates documents to terms; each document has a docId and is added in the posting list of a term if that terms appears in it.

This program uses the Inverted Index, and with large amounts of data as in our case, scaled in-memory indexing relies on Blocked Sort-based indexing where each block is sorted by terms, and each term's posting list is organized within by increasing docIds. Each block is written into disk and a new dictionary is started for the subsequent block without resetting the docId counter. The algorithm described is called Single Pass In Memory Indexing (SPIMI).

2.2 Stop-Word Removal

Stop-words are words that don't hold value other than connecting semantically relevant words around them; this includes articles, pronouns, prepositions, conjunctions, etc. Removing such words saves space when indexing the terms present in a document collection, and usually doesn't affect the assimilation of the user's information need, as they don't hold individual semantic value.

The removal of stop words is optional in this program and is carried out by excluding terms that overlap with an English stop-word list sourced from the web. Queries are processed accordingly.

2.3 Stemming

The stemming algorithm used in this project is the Porter Stemmer, which removes suffixes from words in a number of different patterns. This is particularly useful to reduce the dimentionality of the vector space by only keeping the essential part– or *stem* of the term. The result of queries when using stemming is usually more comprehensive and includes documents that mention the query terms in different shapes.

Stemming is optional, but if chosen, the application uses Apache's Maven dependency "opennlp" to import a PorterStemmer object, and calls the stem() method to transform the term at hand. Query terms are processed accordingly.

Example:

$$Connecting \Rightarrow Connect$$

$$Connected \Rightarrow Connect$$

2.4 Index Compression

The size of the inverted index can become very large when we have a large document collection. A way to reduce the memory it takes is to use compression algorithms.
We used the same compression algorithm, variable-byte encoding, for the document id and the frequency.

A cleverer encoding for the frequency would be the unary encoding. Indeed, the frequencies take small values; they are more or less between 1 and 10, since we are dealing with real documents (which do not contain only the same word many times), and the average length of the documents is more or less between 30 and 50 depending on stopword removal and/or stemming. We tried to implement it but we faced some difficulties.

The unary encoding scheme is bit-oriented, as the length of the unary representation of a posting varies and isn't necessarily a multiple of 8 bits. And the minimum unit we can set the offset to read in a document at a specific place is the byte. Therefore, encoding such posting lists requires of us to keep track of all previous unary representations in the current posting list then write them byte-by-byte. The way we implemented this was not efficient and made the construction of the inverted index longer than 30 minutes.

2.5 Rank-based Information Retrieval

In contrast with boolean information retrieval where the results of a query are a set of documents that strictly contain the query term, regardless of any other comparative qualities, rank-based information retrieval allows to rank those results base on different criteria.

Those criteria(frequency, document length, etc.) are exploited by scoring functions which return a quantitative way to designate which documents are "more relevant" to the user issuing the query.

Our program uses two different scoring functions: TFIDF (Term Frequency and Inverted document Frequency) and BM25 Okapi.

TFIDF: $w_{i,j} = tf_{i,j} \times log(\frac{N}{df_i})$

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

3 How our program works internally

3.1 Class App

The class App is the class containing the main function. It is responsible for the creation of the demo interface. It calls functions from the other modules accordingly during the program-user interaction.

Firstly, the user can ask to extract the document collection from the *tar.gz* file included in the data folder, if not already done. The resulting file is *collection.tsv*. Next, the user must specify whether the necessary indexes (i.e. DocumentIndex.txt, InvertedIndexDocID.txt, InvertedIndexFreq.txt and Lexicon.txt, metaDataCollection.txt) are already present in their directory. If not, they may choose to generate them from scratch. This step takes into consideration 3 input choices from the user including the number of documents to index if not the entire 8.8 millions, whether to use stemming and whether to remove stop-words. The last two choices are saved in Boolean flags to be used when processing the queries.

The class App also prompts the user to enter their query, specify the type; conjunctive or disjunctive, then choose the scoring function to assess the documents. The scoring functions available in this program are TF-IDF and BM25 Okapi.

3.2 Class Indexer

This class is used to construct the inverted index, the lexicon,the MetaData collection, and the document index.
To construct the inverted index, we implement the Single-pass In-memory Indexing (SPIMI) algorithm.

3.2.1 ParseTsvFile

The first part of this algorithm is done by the function *parseTsvFile*. This function takes as argument the path of the collection uncompressed, the number of documents to read (-1 is we want to read all the collection) and a boolean value *stemFlag* that is true if we want to stem the words, false otherwise. The function *parseTsvFile* creates several inverted indexes because we do not have enough cache memory to process the whole collection in one time. It works as follows:

While we have free memory in cache and until the number of documents given is read, we process the documents. We use a dictionary data structure where the key is the term (limited to 64 characters, i.e. 64 bytes) and the value is the posting list. Every time we encounter the term, we add the document id to the corresponding posting list. Once the memory is full, we sort the dictionary and write it to a file named *block*α*.txt* where α is the number of the block. We write strings into the blocks to simplify debugging.

In the blocks, the inverted index is written "term : [0 0 0 1 2 ...]". The blocks are only temporary files and are not the final inverted index. To process a document, we decode it with UTF-8. Then we remove all the punctuation, numbers, multiple spaces and stop words and convert all the text to lower case. We stem the words if the *stemFlag* is true.

We assign a document id (docid) to each document. The number of the document from the collection (docNo) and its length are written in a byte buffer with four bytes each one. When the buffer is full, it is written in the file *DocumentIndex.txt*. We also keep track of the number of documents processed [1] and the average length. They are written in the file *metaDataCollection.txt*.

3.2.2 MergeBlocks

The second part of the SPIMI algorithm is the merging part. Once all the documents are processed, we merge the blocks with the function *mergeBlocks*.
The terms of each block being in alphabetical order, we can advance sequentially and simultaneously in each block. For each new term, we reduce the posting list, i.e. from "term : 0 0 0 1 1" we obtain the document ids 0 (respectively 1) with the frequency 3 (respectively 2). We write the document id and the frequency in two different byte buffers.
The reducing and the writing of the posting lists is done in the function *reduceAndWritePostingList*. The document id and the frequency are encoded in bytes with the variable-byte encoding schemes. The buffers are written in files *InvertedIndexFreq.txt* and *InvertedIndexDocid.txt* when one of them is full.

[1] When the user ask to process the whole collection, we do not know how many documents there are.

The lexicon is constructed during the merging step. When we encounter a term in the blocks, we write it in the lexicon with 64 bytes. We also write two offsets: the number of bytes from which its posting list started in the inverted index of document id's and the one of frequencies. We decided to have two offsets because the document id's and the frequencies are type of integer: as the average length of the documents of the collection is 32 (with our stop words removal), the frequency is a small integer whereas the document id can be a very large integer, from 0 to more than 8 millions. So it is not wise to encode the pairs (docid, frequency) with the same number of bytes. In the lexicon, what takes the more space is the term (64 bytes) and not the offset (4 bytes). So, at the end, even if the lexicon is a bit bigger, the inverted index with the frequency is considerably smaller.

3.3 Class QuerySearch

This class is used to process a query.
Firstly, we process the query in the same way than the documents and we split it into terms. For each term, we get its posting list. The posting list is obtained with the *openList* function. This function searches the term in the lexicon with a binary search. The implementation of this search algorithm is done in the class *BinarySearch*. The lexicon being large, we cannot download it into cache memory. To solve this problem, we use a random access file that allows us to read any line of the file. The size of a line is 72 bytes (64 bytes for the term, 4 bytes for each offset).

The search function returns two lines: the term and the next term with their offsets. Indeed the offsets of the first (resp. second) line are the beginning (resp. end) of the posting list of the term in the inverted indexes. For the last term in the lexicon, we get only one line. This way we know that the term is the last term and that the posting list ends in the end of the file. Once we have all the needed information about the posting list, we create an object *ListPointer* from the class *ListPointer* for each posting list. To get the list of the document id's and of the frequencies of the posting list, we read the corresponding bytes in the inverted index for document id's and frequencies and decode them.

All the implementation of the query processing is based on operations openList(), closeList(), next(), getDocid(), getFreq() on inverted lists implemented in the class *ListPointer*. Once we have all the posting lists of the terms of the query, we sort them based on their length and we process the posting lists Document at a Time (DAAT).

For the conjunctive query, we get the minimum document id among the maximum document id of the posting lists in order to stop the computation as soon as possible. Once we find a document id present in the posting list of each query term, we compute the score. For the disjunctive query, we check for each document id (starting from 0 to

the maximum document id present in the posting lists) at a time if it is in the different postings lists and we compute the score if the document id is a list in one posting list. We do not start the checking each time from the beginning, we keep track of the index we reached.

The score can be TF-IDF or Okapi BM25 depending on the choice of the user. The scores are in a list of size 10, we keep only the top 10. The scores are given to the user in decreasing order with the corresponding document number. To get the document number, we read in the random access file *documentIndex.txt* the first 4 bytes from the byte $documentId * 8$ (Where 8 corresponds to the number of bits to skip). Indeed, we did not save the document id in the inverted index because it just represented the number of the line.

4 How long our program takes to index and to process queries on the provided data set and how large the resulting index files are

The time to index, to process queries and the size of the resulting index files are given in the table 1. The query time (conjunctive and disjunctive) is done with the query collection available on this page: `https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020`. We downloaded the file msmarco-test2020-queries.tsv. This is a collection of 200 queries. The uncompressed file contains one query per line. Each line has the following format:

<qid>\t<query>\n

where <qid> is the query identifier and <query> the query content. The query time (conjunctive and disjunctive) reporting in the table 1 is the mean of the time to process these 200 queries. These times are done with the file *QueryTest.java*. We observed that the query time is well below the second when is use stopword removal. Without the stopword removal, the search takes in mean more than a second. Indeed, when we observe the concerned queries, we see several stopword such as *what, where, is, of*. The posting list of these word are long and we do not implement any skipping like *p.nextGEQ(d)* that advances the iterator forward to the next posting with a document identifier greater than or equal to d. We only move sequentially the iterator to the next posting and we uncompress the whole posting list, which takes time.
The extraction of the collection from the file *collection.tar.gz* to the file *collection.tsv* takes 35 seconds.

Stopword removal	No	No	Yes	Yes
Stemming	No	Yes	No	Yes
Parsing time	5min 30s	7min 22s	6min 46s	10min 48s
Merging time	4min 38s	4min 35s	2min 52s	2min 55s
Total index time	10min 8s	11min 57s	9min 38s	13min 43
Document index size [MB]	67.4	67.4	67.4	67.4
Inverted index docId size [GB]	1.19	1.16	801	774
Inverted index frequencies size [MB]	324	326	212	205
Lexicon size [MB]	85.3	71.6	85.2	71.6
Conjunctive query time [ms]	2906	2870	501	805
Disjunctive query time [ms]	2889	2567	506	437

Table 1: Time to index, to process queries and the size of the resulting index files depending on stopword removal and stemming

5 What limitations our program has

The main limitation of our program is about the blocks. Without (resp. with) the stopword removal, the directory that contains all the blocks has the size 4.11 (resp. 2.43) GB, which is equal or bigger to the size of the collection. At a larger scale, we might not have enough space to store the blocks. A solution would be to store their content into bytes instead of characters.

The second limitation is also about the space. The compression algorithm we use, the variable byte code, is not the most relevant one. Indeed, we have to type of integer: the docid and the term frequency. The docid are from 0 to 8.8 millions. A better compression algorithm would be Delta code, that is used for large numbers. The term frequency are in general very small integers, like 1, 2 or 3. A better compression algorithm would be the unary code as discussed in a previous section.