Master’s Degree Artificial Intelligence and Data Engineering

MIRCV Project

2023/2024

LAURENT Thomas

INTRODUCTION

The purpose of this project is to develop a functional and efficient search engine. The project is decomposed into two main tools: the first preprocesses a collection of document and build an inverted index structure of it; the second processes queries over this inverted index.

The collection of document comes from the MSMARCO passages, composed by over 8,8 million documents. This is the minimum number of documents that the program needs to be able to index.

The whole project is developed in Python 3.12 and is available on the following public Github repository:
https://github.com/tlt-dev/MIRCV

Process to launch the two tools are available in the ReadMe of the repository.

Indexing tools

The indexing process is decomposed into two different steps:

- Preprocessing of the collection
- Build the inverted index structure.

Preprocessing phase

The preprocessing phase is decomposed in several sequential steps:

- Preparation for multiprocessing
- Convert collection to dictionary
- Build document table
- Tokenization
- Normalization

1. Preparation for multiprocessing

In a linear way, more the dataset is large more the program will be time-consuming, because we need to iterate several times on a very large memory-buffer. To reduce the time of preprocessing (and after of building the inverted index), I decided to use eight different processes, each handles a part of the dataset to process.

In order to do that, the collection is split into eight parts each representing the input of the processes. Each process has the same tasks : preprocessed the data and build the inverted index structure.

2. Convert collection to dictionary

To simplify the manipulation of the collection, I decided to convert it into a Python dictionary that offers optimization on iteration process.
Firstly, I check the format of each line of the document. In fact, each document of the collection must be formatted is the following way:

<pid>\t<text>\n

To not compromise the rest of the preprocessing operations, I check if there's no empty lines, no empty documents and if I can get a numeric pid and an alphanumeric text. If all these conditions are met, I create an entry in the collection dictionary with pid as key and text as value.

3. Build document table

The document table takes part into the inverted index structure. This document links the <pid> to the document number and store the length of the document.

Even if it’s not related to the preprocess phase, I build the document table at this moment of the program to avoid needed to reload the original collection after, as this one will be altered by the next steps. Temporary files are used to store the document table built by each process, that will be merge at the end of the indexing phase.

4. Tokenization

The tokenization consists of segmenting the documents in words. Each document is basically splitted on the spaces and all malformed/non-printable characters are removed. In the collection dictionary, texts of the documents are replaced by their respective list of tokens.

5. Normalization

To ensure that all the tokens are in the same format, a step of normalization is essential. During this phase, I operate several operations on the tokens list of each document:

- Punctuation is removed
- Tokens are converted into lowercase
- Some tokens are removed according to a stop words list : it’s a list of very common words that don’t represent a real value for a search engine
- Stemming : we keep the smallest meaningful units of each word
- Numeric tokens are converted to text

Stop words and stemming are processed using the English stop words list and the Porter Stemmer algorithm provided by the nltk python library.

Indexing phase

When a process has finished to preprocess his part of the collection, he builds the inverted index structure. This structure is composed by the inverted index, the lexicon, and the previously built document table.

The inverted index is a structure that maps a term to each document with its posting list. A posting contains a document in which the term appears and its respective frequency of apparition in the document.

The lexicon is a file used to maps a term with three information:

- The document frequency : it’s the total frequency of the term in the collection
- The offset : pointer to the position of the term in the inverted index file
- The length of the posting list

During the indexing phase, the inverted is a Python dictionary, represented as an attribute of the class Indexer(). This class also handles the lexicon structure that will be built at the construction of the final inverted index. The Indexer() class also contains the methods to build, merge, compress and load that structures.

Inverted Index dictionary format:

```
{
  « term » : {
                  doc_id : frequency,
                  doc_id : frequency
                }
}
```

When a process has terminated his task, the Indexer object is returned to the main process. Once all the process are done, we merged all the Indexer objects into the final Indexer. The temporary document table files are also merged into the final document table file. Merging the inverted indexes consists in associating the different posting lists of a same term together.

When the final index is ready, we write it into the inverted index file in a compressed or uncompressed way (depending on the launching configuration of the program). We don't need to store the term in the file as the lexicon already store the starting posting and the length of the posting list associated to each term. The compression format will be discussed in the next part of the document.

We build the lexicon during the index write operation to retrieve easily the offset and the posting length of each term. The lexicon is then also written into the lexicon file.

Compression

1. Uncompressed file format

In the uncompressed way, each posting list is stored on one line, two postings are separated by a space and the docid and the frequency in each posting are separated by a colon.

```
1:3 4:1 5:2 9:2
1:1 3:3 2:1
```

2. Compressed file format

As the inverted index stores the posting list of more than 3 million terms, the file is very large and storing it into a text file is not memory efficient and could be expensive to open and manipulate. To reduce the size of the inverted index, each integer (docid and frequency) are converted into its binary representation. For each bytes composing the binary representation the last bit is set to 1 if the byte is the last of the representation, to 0 if it is not. It is necessary because we can't store all our integer values on one bytes but it also allows us to delete the colon separator between docid and frequency. This basic algorithm approximately halves the size of the inverted index file.

Queries Tool

Once the inverted index structure is built, the second goal of this project is to be able to use it to get the most relevant documents answering to a query. To get this answer, a score is calculated for each document to mark is relevance or not. This score is obtained, with different scoring functions. I have implemented two scoring functions: the BM25 and the TF-IDF.

1. TF-IDF: Term Frequency – Inverse Document Frequency

This scoring function is divided in two parts.

The Term Frequency (TF) determines the occurrence frequency of the term in a document.
The calculation is simple:

$$TF(t,d) = \frac{Number\ of\ times\ term\ t\ appears\ in\ ocument\ d}{Total\ number\ of\ terms\ in\ document\ d}$$

The purpose of the IDF is to diminish the weight of terms that occur very frequently across a collection of documents and increase the weight of terms that occur rarely.

$$IDF(t,D) = \log\left(\frac{Total\ number\ of\ documents\ in\ the\ set\ of\ document\ D}{Number\ of\ documents\ where\ the\ term\ tt\ appears}\right)$$

Then we multiplicate the TF and the IDF to get the relevance score of the document for the current term. By summing the score of each term in the query, we got the relevance score of the document. More this score is high, more the document is relevant for the considered query.

2. BM25: Best Matching 25

This algorithm introduced two concepts in comparison to the TF-IDF function. The first is the term frequency saturation that is the consideration that beyond a certain point, additional occurrences of the term do not significantly contribute to the document's relevance. The second is the document length normalization that allows to prevent the longer documents to be more likely retrieved just because they have more words and so, more chance to contains the query's terms.

$$\boldsymbol{Score(d,Q) = \sum_{t\in Q} IDF(t) \times \frac{f(t,d) \times (k_1 + 1)}{f(t,D) + k_1 \times (1 - b + b \times \frac{|d|}{avgdl}}}$$

Where:

- $f(t,d)$ is the frequency of term *t* in *d*.
- $|d|$ is the length of document d.
- *avgdl* is the average document length in the collection.
- k_1 and *b* are free parameters (k_1 between 1.2 and 2, b usually equal to 0.75)

Once the score is calculated for each document in the collection, I return the 10 documents with the highest relevance score.

Performances

1. Time Performance

Purpose	**Time**
Build Inverted Index	Betwenn 2500 and 2700s (41-45 min)
Load Lexicon	2.3s
Load Document Table	7.2s
Avg time per query (TFIDF)	0.42
Avg time per query (BM25)	0.83s

2. Files space

File	**Space**
Inverted Index	544 Mo
Lexicon	76 Mo
Document Table	200 Mo

3. Precision

TF-IDF	BM25
0.1278	0.1944

LIMITATIONS

As we can see, the precision and the speed is not optimal. Several solutions should be implemented to increase the precision, the speed, and the memory management.
First, the preprocessing of data is very basic and consider just each word as a term. Some algorithms, like the Byte Pair Encoding permit to reduce the vocabulary size. Also, better compression algorithms should be implemented to reduce the files space even more.
The posting lists are currently traversed Term-At-A-Time that is memory and time-consuming. Using Document-At-A-Time technics would allow to traversed the posting lists in parallel. Implementing such methods like NextGEQ() would increase the search speed while going through the posting lists. Maybe, processing also each term of a query in parallel/different processes would increase a lot the speed of search.