UNIVERSITÀ DI PISA

Department of Information Engineering (DII)
M.Sc in Computer Engineering

Multimedia Information Retrieval and Computer Vision Project

Pietrangelo Manco

`https://github.com/PietrangeloManco/MIRCV-Project`

Academic Year 2023/2024

Contents

1 Program Goals

The program has 3 main functionalities:

- Building an Index Structure: The first thing the program can do is to process a collection of documents to build an Index Structure whose final purpose is to provide answers to user-submitted queries. The index is composed of an Inverted Index, to store term-to-document mapping with relative term frequencies, a Lexicon, to store terms with relative document frequencies, and a Document Table, to store document ids with relative document length.
- Processing user-submitted queries: The program is able to answer conjunctive and disjunctive queries, using TFIDF or BM25 scoring. At query formulation time, the user chooses the desired configuration for both cases.
- Query processing evaluation: The program is able to compute an evaluation of the performance by scoring a set of benchmark queries with relative relevance assessments and computing NDCG for each run in all 4 possible configurations of query type and scoring method.

2 Internal Structure

The purpose of this section is to explain how the program works internally in each of its 3 functionalities.

2.1 Index Building

The Index Structure building can be split into the following operations:

- Collection loading: The collection is loaded in memory in chunks. The program allows for 2 possible ways to do that: the user can choose a

chunk size or leave the default one (static chunking), or the program can choose the chunk size at each iteration, according to the memory state of the underlying system.

- Preprocessing: Before actual building of the partial structures, each document collection chunk gets preprocessed in parallel, for efficiency.
- Partial Structures Building: After preprocessing, the index builder module creates a partial inverted index, partial Lexicon, and partial Document Table by processing the list of list of tokens obtained by the chunk (there is one list for each document in the chunk). The module exploits 2 mapping structures: one to map tokens' document frequencies, that are part of the Lexicon, and one to map tokens' term frequencies (once per document), to compute them as payload of the terms' postings. In addition to that, the length of each text before preprocessing is extracted and added to the relative document's entry in the Document Table. At the end of each chunk processing, the final Lexicon and Document Table structures are updated, and a partial Inverted Index is created, compressed with p for delta compression and stored in a temporary file.
- Partial indices merge: After the processing of the whole collection, Document Table and Lexicon structures were fully created and saved on disk, together with a variable number of partial Inverted Indexes in compressed form. The merger loads them all in memory and starts merging them 2 by 2 in parallel, until there is only one left: the final Inverted Index, that gets saved on disk in compressed form.

2.2 Query Processing

The query processing can be split into the following operations:

- Query Parsing: Once the query is acquired by the system, it applies the full preprocessing pipeline to it, returning a list of tokens.
- Query Type: Based on the input query type, the program computes a list of documents to score.

  - Conjunctive: Since each document must match all the query terms, posting lists are retrieved, uncompressed and intersected.
  - Disjunctive: Each document must match at least one query term, so the operation computed over posting lists is union in this case.

- Document Scoring: Once the list of documents has been computed, the program exploits the Scoring class to score them according to the function chosen by the user: TFIDF or BM25. After the scoring of the whole list, a dict of doc ids and scores is returned, sorted and cut to top k (default is 10, but can be changed accordingly). Top k document ids are returned, together with their scores.

2.3 System Evaluation

The system evaluation can be split into the following operations:

- Queries and assessments loading: Benchmark queries and assessments are loaded into a list of tuples and a dictionary, respectively. Since only 1/4 of the benchmark queries contained in the file actually have assessments, they are filtered accordingly (only the judged ones are kept).
- Query parsing: Queries are preprocessed, like in the query processing scenario.
- Queries evaluation: The search engine is ran over the benchmark queries in all the 4 possible combinations of query type and scoring method. The ideal run is extracted from the assessments and the NDCG metric is computed for each query and each combination.
- Results computation: Finally, the average NDCG is calculated for each of the 4 combinations and it's displayed to the user.

3 Performance

The performance evaluation has 4 key aspects:

- Index building time: Partial indexes building time + merging process time. The first operation is longer and takes around 40-50 minutes. The second operation takes around 20-30 minutes. The total indexing time varies in a range between 60 and 80 minutes, depending on the system's available resources.
- Index files size: The files generated at the end are InvertedIndex, Lexicon and DocumentTable.
  - The only one compressed is the Inverted Index, that contains the list of terms and relative Posting Lists, with document ids and term frequencies. Its size is around 872 MB.
  - Document Table contains a list of document ids and document lengths, and its size is around 103 MB.
  - Lexicon contains a list of collection's terms and relative document frequencies, and its size is around 17 MB.
- Query processing time: It's highly variable, depending especially on the query length and the query terms posting lists length, but also on the scoring method and the query type of choice.
  - For a query like "Where to eat pizza in Naples", the conjunctive processing takes 0.24s and only retrieves 3 documents, while the disjunctive one takes 0.55s and retrieves the top k. That suggests that the query terms aren't really popular in the collection, so posting lists are likely short and performance is good.
  - For a query like "best data mining algorithms", which has approximately the same length and the previous one, performance is much worse: it takes around 1s in the conjunctive case and 2.7s in the disjunctive one. That is likely caused by the abundance of related documents in the collection, so fetching, decompressing and processing the query terms posting lists takes much longer.

– For a more complex query of the first kind, like "best italian restaurants under 30 dollars" the conjunctive query processing takes around 0.8s, but that's influenced by the fact that there isn't any documents containing all the query terms. The disjunctive query processing takes around 2s, that is still high, but less than a less complex query with more popular terms.

  The system works reasonably well for short queries, with the performance degrading quickly in case of terms with long posting lists with respect to less frequent terms, and more slowly in case of long queries with respect to short ones.

– NDCG evaluation: Finally, an important evaluation metric is the quality of the retrieved results. The evaluation was carried out over a set of benchmark queries taken from the "msmarco-test2020-queries.tsv" file. Relevance assessments were taken from the "2020qrels-pass.txt" file. There were 54 queries with evaluation assessments, and each was tested with the 4 query type/scoring method combinations. The NDCG was computed for each of the 216 runs, and the final average was computed for each combination. The final results were reported in Table 1.

  The 2 scoring measures perform exactly in the same way for conjunctive queries: that's probably due to the fact that the conjunctive constraint is hard enough to leave only the actual good documents to the scoring phase. The situation changes in the disjunctive scenario, where the BM25 clearly outperforms the TFIDF, as expected. An idea to improve the quality of retrieved results is to take into account synonyms and similar terms.

Table 1: NDCG scores for the 4 combinations of query type and scoring method.

	Conjunctive	Disjunctive
TFIDF	0.742	0.751
BM25	0.742	0.795

3.1 Limitations

The biggest limitation of the system, as previously seen, is the growing query processing time in progressively complex scenarios. This issue could be addressed by implementing a dynamic pruning strategy, like Max Score or Wand, together with the related optimization strategies (skipping) and structures (skip pointers). In fact, these strategies specifically target the performance issues caused by the length of compressed posting lists, both in retrieval and processing, which have been identified as the biggest performance bottlenecks.

4 Major Functions and Modules

Major classes and functions will be briefly described in this section. Each has been tested through extensive testing modules.

4.1 Utils Package

The classes in this package have utilities purpose. Major ones are CollectionLoader, CompressionTools and Preprocessing.

4.1.1 Collection Loader

The class responsible to handle the document collection. It can process the whole collection in chunks, or it can randomly sample a number of documents of choice. Documents are represented as Pandas Data frames. Major functions are:

- *process_single_chunk*: This function is responsible to open the compressed collection file, read the input number of lines starting from the input one, and return a Data frame with 2 columns: 'index' and 'text'.

- *process_chunks*: This function iteratively calls *process_single_chunk* until the document collection is fully processed. Each chunk is yielded at each iteration. Chunk size is 500000 by class default, but can be specified here as input.
- *sample_lines*: This module builds a Data frame of random lines for testing purposes. The logic is the same as *process_single_chunk*, except it randomly generates an int at each iteration and appends the corresponding line to the final output.

4.1.2 Compression Tools

This class was used to implement methods responsible for the inverted index compression and decompression. P for delta compression was used. The functions are:

- $p\_for\_delta\_compress$: Takes as input 2 lists containing doc ids and relative frequencies and returns bytes. P for delta is used to compress doc ids in a more efficient way.
- $p\_for\_delta\_decompress$: Takes as input bytes and returns the lists of relative doc ids and frequencies.

4.1.3 Preprocessing

The preproccesing class has the usual functionalities. There is special attention to URLs and code-like removal as there are a lot of documents in the collection that are composed exclusively by them, making them useless.

The only noticeable function is *vectorized_preprocess*, that applies the full preprocessing pipeline in a parallel way, to fully exploit the multi-core capacity of the system. It uses all the cores except one, left for other operations.

4.2 Index Package

The Index Package contains all the classes related to the Index: InvertedIndex, CompressedInvertedIndex, Lexicon, DocumentTable and classes responsible of Index Building.

4.2.1 CompressedInvertedIndex and InvertedIndex

Since the inverted index is kept in memory fully compressed, this class is used to manage it in this form. The InvertedIndex class is used to manage the partial indices during building and to write the uncompressed inverted indexes on a file in compressed form.

The most important functions are:

- *write_compressed_index_to_file*: Used during merging: partial indices are saved on disk in compressed form, so this method is used to write the final compressed index to disk.
- *load_compressed_index_to_memory*: Used both during merging and to initialize the index structures during query processing.
- *write_index_compressed_to_file*: This is the method used to write an index that's uncompressed in memory to a file in compressed form.

4.2.2 InvertedIndexBuilder

This class is used to build the index structures and partial index structures for testing. Main methods are:

- *process_chunk*: Main processing module, it takes as input a collection chunk provided by the CollectionLoader, preprocesses it to extract a list of list of tokens (one per document of the chunk), and creates 2

maps to track term frequency for each term, document pair, and overall document frequency for each term. Final Lexicon and DocumentTable are progressively built by this method, while a partial inverted index is returned.

- *build_partial_indices*: This method is used to coordinate the building of partial indices. Its main responsibility is to implement the dynamic chunking logic, if it's selected: for each iteration, it makes an estimate of the maximum chunk size based on the available memory. It also saves partial index files to disk and ensures memory is freed up properly after each iteration.

- *build_full_index*: This method combines together the logic described to build the partial indexes with the merging process, to build the final Inverted Index. It also writes to file the Lexicon and the DocumentTable.

- *build_partial_index*: implements the same logic over the lines randomly sampled by the relative method in the CollectionLoader.

4.2.3 Merger

This class is used to merge the partial inverted indexes saved on disk in compressed form by the InvertedIndexBuilder. Its methods are:

- *_merge_compressed_postings*: Main processing unit, simply takes the compressed postings of 2 instances of the same term, coming from different partial inverted indexes, decompresses and merges them, and returns the final posting list compressed again. It's general enough to handle empty postings.

- *_merge_two_indices*: Method to merge two partial inverted indexes. Necessary because the merging of the indexes is carried out in parallel, two by two. It iteratively calls the *_merge_compressed_postings* over the two files.

- *merge_multiple_compressed_indices*: Method responsible to manage the parallel merging process, goes on with the merge of partial indexes 2 by 2 until one is left, and it's returned.

4.3 Query Package

This package's classes are Scoring, QueryParser and QueryProcessor.

- The QueryParser class is just an embedding of the Preprocessing, adjusted to handle query terms.
- The Scoring class is simple as well: its main methods are simple implementations of TFIDF and BM25 scoring.

4.3.1 QueryProcessor

The class that actually executes query processing. Main functions are:

- *process_query*: method that coordinates the execution. It parses the query, calls the correct method according to the query type (conjunctive or disjunctive), performs ranking of the returned list of documents using the Scoring class, and returns the top k in descending order of relevance, according to the scoring measure of choice.
- *rank_documents*: The scoring functions can only perform the scoring of a term against a document, this method manages the process by executing the scoring process for each retrieved document against each query term.
- *execute_conjunctive_query*: This method retrieves the document ids of the documents that match all of the query terms. For efficiency, it starts from the shortest posting list.

- *execute_disjunctive_query*: This method retrieves the document ids of the documents that match at least one of the query terms, so it just computes the union of the query terms' posting lists.

4.4 EvaluationMain

This module is a runnable that performs the evaluation of the system computing NDCG, using benchmark queries and assessments. Its main functions are:

- *compute_ndcg*: This function performs the ndcg computation after some preprocessing operations over relevance scores and judgments.
- *process_queries*: Extracts query id and text from the query file.
- *evaluate_all_queries*: Actual processing method, combines the previous methods to compute NDCG scores for each query in each combination of query type and scoring method.