MIRCV Project Description

In this project, you are asked to write a program that creates an inverted index structure from a set of text documents and a program that processes queries over such inverted index.

It is strongly recommended to implement this project by decomposing it into multiple components. For example, indexing can be carried out by one component for parsing the documents, generating intermediate postings and writing these postings out in unsorted or partially sorted form, one component for sorting or merging the postings, and one component for reformatting the sorted postings into the final index and lookup structures.

For developing this project, it is strongly advised to create a test collection with a few documents, and implement regression tests to check that every component/function/method behaves as expected on the test collection.

In order to positively evaluated, the project *must* fulfil the following *mandatory* requirements:

Programming Language. You *must* use C++ or Java for this project, as well as a mix of these languages. Make sure you know how to "operate on a binary level" when it comes to input, output, or compression in your chosen language(s). You *must* use Github to manage your project development.

Dataset Specification. You *must* use the document collection available on this page: `https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020`. Scroll down the page until you come to a section titled "*Passage ranking dataset*", and download the first link in the table, `collection.tar.gz`. Note that this is a collection of 8.8M documents, also called passages, about 2.2GB in size. The uncompressed file contains one document per line. Each line has the following format:

```
<pid>\t<text>\n
```

where `<pid>` is the docno and `<text>` is the document content. These 8.8M documents *must* be the minimum data set you should be able to index.

Document Processing. Note that the data may contain text from many different languages, including text that uses non-ASCII character sets. You *must* read the data using UNICODE instead of basic ASCII. Also, the data may be provided in a fairly raw form with various errors (empty page, malformed lines, malformed characters) that you *must* to be able to to deal with. You *must* remove the punctuation signs.

Inverted Index. You *must* create an inverted index structure, plus structures for the lexicon (to store the vocabulary terms and their information) and for the document table (to store docid to docno mapping and document lengths). You *must*

not build an index by simply shoving all the posting data into one of the high level data structures provided by many languages. Also, you *must not* load the posting data into a relational database. At the end of the program, all structures *must* be stored on disk in some suitable format.

Query Execution. Your program *must* execute ranked queries by using something similar to the simple interface presented in class based on operations `openList()`, `closeList()`, `next()`, `getDocid()`, `getFreq()` on inverted lists. This way, issues such as file input and inverted list compression technique should be completely hidden from the higher-level query processor. Your program *must* return the top 10 or 20 results according to the TFIDF scoring function. You *must* implement conjunctive and disjunctive queries.

Demo Interface. Upon startup, your program may read the complete lexicon and URL table data structures from disk into main memory, which might take some seconds or even a minute. However, you *must not* not read the inverted index itself into memory. Your program *must* read user queries via simple command line prompt, and *must* return the `<pid>` of each result. After the user inputs a query, your program *must* perform seeks on disk in order to read only those inverted lists from disk that correspond to query words, and then compute the result. After returning the result, your program should wait for the next query to be input.

Documentation. Your program source code *must* be well-commented, and *must* include a `README` file explaining what your program can do, and how to compile & run the program.

In order to receive higher marks, the project *should* fulfil (some of) the following *optional* requirements:

Compressed Reading. The document collection *should* be uncompressed during parsing, by loading one compressed file into memory and then calling the right library function to uncompress it into another memory-based buffer.

Stemming & Stopword Removal. Stemming and stopword removal *should* be implemented. For stemming, you can use any third-party library providing the Porter stemming algorithm or similar. For stopword removal, you can use any english stopwords list available on the Web. Ideally, your program should have a compile flag that allows you to enalbe/disble stemming & stopword removal.

Index Compression. You *should* implement at least some basic integer compression method to decrease the index size. Do not use standard compressors such as `gzip` for index compression. Ideally, your program should have a compile flag that allows you to use ASCII format during debugging and binary format for performance.

Scoring Function. You *should* implement the BM25 or other scoring functions such as those based on language models.

Query Processing Efficiency. You *should* implement skipping in your inverted lists, you *should* implement the `nextGEQ()` operation in your posting interface, and you *should* implement a dynamic pruning algorithm.

The project can be carried out in groups of 1-3 participants, with no exceptions. For the project discussion, hand in the following in a *public* Github repository link:

1. your well-commented program source;
2. a 10-pages paper (written in Word or Latex or similar) explaining
   - what your program can do;
   - how it works internally;
   - how long it takes to index and to process queries on the provided data set and how large the resulting index files are;
   - what limitations it has;
   - what the major functions and modules are.

The *final evaluation* is composed of (1) the *project discussion* and (2) the *oral examination*, and it must be carried out according to the following rules:

1. every student *may* perform the *project discussion* independently from the other members;
2. the *project discussion* and the *oral examination must* be carried out together, in the same evaluation appeal;
3. a student failing a *final evaluation must* perform the *project discussion* and undertake the *oral examination* again in a future evaluation appeal;
4. the project link *must* be communicated via email no later than 7 (seven) days before the official schedule of the evaluation appeal in which the project should be presented. Late communications will not be taken into account.
5. No changes to the repository will be taken into account after the above deadline.