COL764/COL7364: Information Retrieval & Web Search
Assignment 1: Indexing and Boolean Retrieval

August 19, 2025

Deadline

Submission of the complete implementation with report on the algorithms is due on August 27, 2025, 11:59 PM. All the submissions are to be made on Moodle.

Instructions

Follow all the instructions. Not following these instructions will result in penalty.

1. **The assignment is to be done individually or in pairs.** Do **not** collaborate by sharing code, algorithms, or any other details. Discussion for clarification is allowed but it must happen in the "Assignment 1" channel on Microsoft Teams.
2. All programs must be written in **Python (3.12), Java (SE) 22, C, C++ (gcc12.3) only**. Any other language requires prior instructor approval. We recommend you use the same versions or ensure compatibility with these.
3. **Implement from scratch.** You are allowed to use only the standard library with exactly the exceptions mentioned (in 1.4). Use of any other library is not allowed and will immediately result in 0 marks. If your program depends on any special libraries, get approval from the instructor beforehand (or ask on Teams).
4. All submissions will be evaluated on **Ubuntu Linux**. Ensure that file names, paths, and argument parsing are Linux-compatible. Each team will be given an account on the baadal machine. The code needs to be tested there as we will be running evaluation scripts on baadal only.
5. **Submission Details:** Submisssion details will be shared later.
6. **No deadline extensions** will be given. Late submission is allowed with penalty as outlined in the Introductory class. Note that this assignment requires substantial implementation effort, as well as some manual tuning for performance, so it is advised to start early. **Do not wait until the last moment.**

1 Assignment Description

This assignment is designed to help you gain practical experience with the core Information Retrieval (IR) concepts - tokenization, inverted indexing and query parsing, and query evaluation in the context of **Boolean retrieval**. It also introduces you to standard retrieval benchmarks, such as the TREC Robust 2004 dataset. Additionally, students are expected to implement index compression techniques as part of the assignment.

1.1 Document Collection (Corpus)

You are given a document collection – extracted from a benchmark collection in TREC (**CORD-19**[1]), consisting of publicly available collection of scholarly articles related to COVID-19, SARS-CoV-2, and other coronaviruses. Each document is represented as a JSON fragment (as shown in the box below) and has a unique document-id. All content except the doc-id can be indexed, and this indexable portion of the document will be referred to as **'document-content'** in the rest of this document.

The document collection itself is specified as a single JSON file, containing one or more documents (each is a JSON fragment).

An Example of JSON Fragment Representing a Document.

Each document is enclosed in curly braces, contains a "doc id" field, and remaining contents under different (collection specific) set of JSON fields.

```
{
"doc_id"  : "2b73a28n",
"title"   : "Role of endothelin-1 in lung disease",
"doi"     : "10.1186/rr44",
"date"    : "2001-02-22",
"abstract": "Endothelin-1 (ET-1) is a 21 amino acid peptide with diverse
             biological activity that has been implicated in numerous diseases.
             ET-1 is a potent mitogen regulator of smooth muscle tone, and
             inflammatory mediator that may play a key role in diseases of the
             airways, pulmonary circulation, and inflammatory lung diseases"
}
```

You may safely assume that all documents are well-formed (i.e., no broken JSON fields/formats), and contain no arbitrary non-ASCII characters.

1.2 Test Queries

Queries are provided in a json file named **test_queries.json**. Within each query, fields for **query_id**, **title** and **description** are present. **query_id** is the unique identifier for a query and is not necessarily an integer. **title** contains the query and **description** is an explanatory description of the query. A fragment of the file for one query is shown below:

[1]Read about CORD-19 here

Format of each Query in `test_query.json` file

```
{
"query_id"    : "1",
"title"       : "coronavirus origin",
"description" : "what is the origin of COVID-19",
"narrative"   : "seeking range of information about the SARS-CoV-2 virus's
                 origin, including its evolution, source, and transmission."
}
```

The above is the default format of query files in TREC. You are however expected to make use of the content under the "title" field only (as the scope of this assignment is Boolean Retrieval). Note that, **if no connective is present between any two terms**, assume `AND` between them. For example: `query_id 1` is `coronavirus AND origin`. Ensure that the title is (carefully) tokenized using your tokenizer before retrieval.

1.3 Download Details

The training corpus and other files are available for download at the following path: SharePoint Folder

Use the full corpus. Each document has a `doc_id` and `text`. Limit to the first 10,000 documents for development if needed.

1.4 Implementation Constraints

- **Allowed modules (or built-in functions**:
  - **Python**: Only standard Python libraries, such as `os`, `re`, `json`, `math`, `collections`, `time`, `zlib`.
  - **Java**: Only classes from the Java Standard Library (e.g., `java.io`, `java.nio`, `java.util`, `java.lang`, `java.math`).
  - **C++**: Only C++ Standard Library headers (e.g., `<iostream>`, `<fstream>`, `<sstream>`, `<string>`, `<vector>`, `<map>`, `<algorithm>`, `<cmath>`).
  - **C**: Only the C Standard Library headers (e.g., `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<math.h>`).
- **Disallowed (third-party or non-standard libraries):**
  - **Python**: `nltk`, `spacy`, `tokenizers`, `regex` (third-party version), `pandas`, `numpy`, `whoosh`, `elasticsearch`, etc.
  - **Java**: Any external search/NLP libraries such as Apache Lucene, OpenNLP, Stanford CoreNLP, or LingPipe.
  - **C++**: Any external search/NLP libraries such as Xapian, SphinxSearch, or Lemur/Indri.
  - **General rule**: You may use only the standard string and I/O utilities provided by the language (e.g., Java's `java.util.regex`, C++'s `<regex>`), and you must implement tokenization, indexing, and retrieval logic yourself.

2 Tasks

This section specifies the artifacts and behaviors required to implement a Boolean retrieval system over the provided corpus. All tokenization and query handling must be **deterministic** and **reproducible** across runs.

2.1 Task 1: Custom Tokenizer

Objective: Build a simple, reproducible tokenizer and generate the vocabulary.

Function:

```
def build_vocab(corpus_dir: str, stopwords_file: str, vocab_dir: str) ->
    None
```

Inputs:

1. **corpus_dir**: Path of the corpus directory. (file(s) present here follow the structure mentioned in 1.1.)
2. **stopwords_file**: Path of stopwords.txt. It contains one token per line. Exact matches are to be removed.
3. **vocab_dir**: Absolute path of directory where vocab.txt is to be saved.

Output:

1. Save your vocabulary in the file **vocab.txt** in **vocab_dir**

Tokenization Process:
Do the following on the **document content** (as defined in 1.1) of each document:

1. **Preprocessing**: Lowercase, Remove digits 0–9. (retain all other characters i.e. punctuation and symbols).
2. **Split on whitespace** to obtain raw tokens.
3. Remove any token that **exactly matches** a stopword (from stopwords.txt).

Example

stopwords.txt:

```
{the, at}
```

Raw text:

```
The co-op reopens Room-101 at #3 - $20 off today!
```

Preprocess (lowercase, remove digits)

```
the co-op reopens room- at # - $ off today!
```

Split on whitespace

```
["the","co-op","reopens","room-","at","#","-","$","off","today!"]
```

Remove stopwords (exact matches)

```
["co-op","reopens","room-","#","-","$","off","today!"]
```

⇒ The entries in the output file (named `vocab.txt`) include:

```
#
$
-
co-op
off
reopens
room-
today!
```

Behaviour(Vocabulary):

1. Accumulate tokens from all documents into a set.
2. When you export or print the vocabulary for inspection, ensure it is **lexicographically sorted**, one token per line, no duplicates, no blank lines.

2.2 Task 2: Inverted Index

Objective: Build a in-memory positional inverted index over the corpus limited to the tokens present in your vocabulary (i.e. `vocab.txt`). Use the best data structures and algorithms that you can think of (recall the discussion from class).

Logical Format: You will be asked to save your inverted index into a JSON file in the following format:

```
{
  "term1": {
    "docA": [p1, p2, ...],
    "docB": [p3, p4, ...]
  },
  "term2": {
    "docC": [p5, p6, ...]
  }
}
```

- Keys at the top level are terms (strings).
- For each term, the value is a map from **"doc_id"** (string) → **list of positions** (integers).
- **Positions are 0-based** within each document and must be sorted ascending.
- For any exported/printed JSON rendering:
  - Terms must be sorted lexicographically.
  - For each term, document IDs must be sorted lexicographically ascending
  - **NOTE:** You do not have to create a in memory json object, you can use any efficient data structure as you see fit. We require you to only create the in memory index and store it in a json file. Also the positions created during the index creation will not be used to do retrieval.

Function:

```
def build_index(collection_dir: str, vocab_path: str) ->
    inverted_index : object
```

```
def save_index(inverted_index : object, index_dir: str) -> None
```

Behavior:

1. Load your vocabulary into a lookup set V.
2. For each document d in `<COLLECTION_DIR>`:
   (a) Tokenize its text exactly as in task 1.
   (b) The inverted index should contain, for each token, the documents in which it occurs, and in each document, the position at which this occurs. The very first position in the document is the 0^{th} position.
3. You may use *any data structure* that you think is efficient.
4. Store the index in a json file, named **index.json** as per the format mentioned above in directory **index_dir**

2.3 Task 3: Index Compression

Objective: Reduce the on-disk footprint of your inverted index using variable-length encoding, while preserving exact retrieval semantics.

Function:

```
def compress_index(path_to_index_file : str,
    path_to_compressed_files_directory : str) -> None
```

Notes

1. **File layout:** The compressed index file(s) will have to be saved on disk. It is important that you think carefully about what the layout of the file(s) should be. The evaluation is based not just on the compression methods used, but also the final on-disk size reduction.
2. **DocID mapping:** You may map the string document IDs to integers. This extra information will need to be stored when you save your *compressed index.* Choose a layout for your file that will allow you to store the mapping.
3. **Encoding:** Use variable-length encoding for integer sequences (e.g., variable-byte). Delta/-gap encoding is allowed but not required. You may encode positions as you see fit. Note that there are potentially two integer sequences – the docIds (which have been mapped to integers) and positions. You may choose to compress either one or both in any way you like.
4. **Output:** Use the **<COMPRESSED_DIR>/** directory to save all the compressed index artifacts (uncompressed structures, compressed byte streams, internal metadata, etc.). You pick the filenames and layout. The only requirement is that your retrieval component (Task 4) can load the compressed representation and answer Boolean queries correctly
5. **Losslessness:** The saved compressed index will be subsequently loaded. The loaded index must reconstruct your original index exactly.

2.4 Task 4: Boolean Retrieval

2.4.1 Decompression

Objective: Reconstruct the logical inverted index (Task 2 format) from your compressed representation so that Boolean retrieval funtion can decompress the on-disk compressed index into uncompressed in-memeory index.

Function:

```
def decompress_index(compressed_index_dir: str) -> inverted_index:
    object
```

Conceptual Requirements:

- Decode the variable-length encoded integers (and any gaps, if you used deltas).
- Map integer DocIDs back to string DocIDs using the inverse of your DocID mapping.
- For each term, reconstruct the postings with docIDs lexicographically ordered and positions sorted ascending.
- Save the decompressed index in a json file.

2.4.2 Query Pre-processing

Objective: transform the raw query text into a cleaner, more standardized form so that it can be matched effectively against the indexed documents.
Input: Query file has the format as specified in the Assignment Description (section 1). Use only the title field to form the Boolean query string. If no explicit connectives appear between two successive tokens, assume those to be connected with `AND`.

Examples *(titles → interpreted Boolean)*:

- *jaguar NOT car → jaguar AND NOT car*
- *(information retrieval) OR indexing → (information AND retrieval) OR indexing* (because the parenthesized phrase has no operator; apply the same "implicit AND" rule inside parentheses.)

Tokenization of Queries:

1. Apply the same tokenizer and normalization used for documents in Task 1 (lowercasing, digit removal, whitespace split, stopword removal).
2. Connectives are the reserved words: AND, OR, NOT. They are case-insensitive in parsing, but you should canonicalize them (e.g., uppercase) during parsing.
3. Parentheses *( )* are supported.

2.4.3 Boolean Query Parser

Objective: You must write a parser that:

- Converts infix Boolean queries to a postfix format
- Builds an operator tree
- **Order of precedence of explicit operators that we are going to follow**:

```
highest    () > not > and > or    lowest
```

- Evaluates the tree recursively

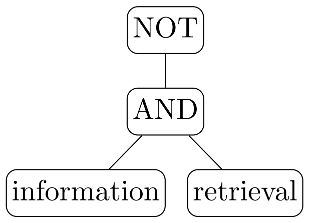
Funtion

```
def query_parser(query_tokens: vector) -> AST/Postfix
```

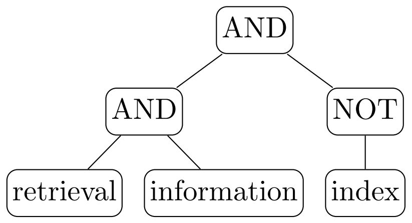
Example Query:

```
(a) not (information and retrieval)
(b) retrieval information not index
```

are parsed as follows:



(a) Parse tree for: `not (information and retrieval)`



(b) Parse tree under implicit AND: `retrieval AND information AND NOT index`

Figure 1: Example parse trees for two Queries

2.4.4 Boolean Retrieval

Implement a function that will take in a set of queries in the json format specified earlier and outputs a result file (named **docids.txt**).
Function:

```
def boolean_retrieval(inverted_index : object, path_to_query_file: str
    , ,output_dir: str) -> None
```

Output: Result file in TREC-eval format Produce a file named **docids.txt** in the **output_dir** dirctory. where each line follows the following format (header is not to be written):

Output Format

```
qid          docid          rank   score
Q         SZF08-175-870       5       1
```

- **qid**: Query Identifier present in the test_query file.
- **docid**: a document ID from the corpus (string).
- **rank**: In ranked retrieval, **score** (e.g., TF-IDF or another ranking metric) determines the rank assigned to each document. For the boolean retrieval use lexicographic order of docIDs as the tiebreaker and assign ranks 1..k in that order.Non-matching documents are not to be listed.
- **score**: a numeric score (for Boolean retrieval, use a constant like 1 for retrieved docs).

Behavior:

1. Tokenize the title string using Task 1 rules (tokens and punctuation handling).
2. Insert implicit ANDs where two tokens appear without an explicit operator.
3. Convert infix to a machine-evaluable form (e.g., postfix) or build an operator tree.
4. Evaluate the expression using postings lists from the index; use set operations as specified.
5. Generate runfile lines in TREC format.

3 Program Structure and Submission Plan

3.1 Code Structure

In order to complete the above tasks, you are required to write the following programs.

1. **Task 1: Custom Tokenizer** The program for this task should be named as `tokenize_corpus.{py|c|cpp|C|java}` and should contain the `build_vocab()` function.

   This program will be executed using a shell script named `tokenize_corpus.sh` (to be provided by you). The shell script will be invoked via terminal as follows:

   ```
   tokenize_corpus.sh <CORPUS_DIR> <PATH_OF_STOPWORDS_FILE> <VOCAB_DIR>
   ```

   where:

   `<CORPUS_DIR>`: absolute path of the directory containing all the training documents only (in the same format as mentioned in 1.1). (you must use the entire set of documents here).

   `<PATH_OF_STOPWORDS_FILE>`: Absolute path of the file having List of stopwords (one per line).

   `<VOCAB_PATH>`: absolute path of the directory where output file (`vocab.txt`) is to be saved.

   Output: The program should generate one file `vocab.txt` which contains the tokens generated as the result of tokeniztion of the corpus.

2. **Task 2 and 3: Inverted-Index and Index-Compression:** The program should be named as `build_index.{py|c|cpp|C|java}` and should contain the `build_index()`, `save_index()`, `compress_index()` and `save_compressed_index()` functions.

   This program will be executed using a shell script (to be provided by you) named `build_index.sh`. The shell script will be invoked via terminal as follows:

   ```
   build_index.sh <CORPUS_DIR> <VOCAB_PATH> <INDEX_DIR> <COMPRESSED_DIR>
   ```

   where:

   `<CORPUS_DIR>`: Same as in part 1.

   `<VOCAB_PATH>`: absolute path of the vocabulary file generated above

   `<INDEX_DIR>`: absolute path of directory where json file for index (**index.json**) is to be saved

   `<COMPRESSED_DIR>`: absolute path of the directory where compressed file(s) is/are to be saved

   Output: The program should generate one file `index.json` which contains the inverted index postings and compressed file(s) in the `<COMPRESSED_DIR>` directory.

3. **Task 4: Boolean Retrieval :** Your submission should also consist of another program called `retrieval.{py|c|cpp|C|java}` for performing boolean retrieval using the index you have just built above. It should contain the `decompress_index()`, `query_parser()` and `boolean_retrieve()` functions.

   This program will be executed using a shell script (to be provided by you) named `retrieval.sh`. The shell script will be invoked via terminal as follows:

```
retrieval.sh  <COMPRESSED_DIR> <QUERY_FILE_PATH> <OUTPUT_DIR>
```

where:

`<COMPRESSED_DIR>`: absolute path of the directory where compressed file(s) is/are saved

`<QUERY_FILE_PATH>`: absolute path of the query file

`<OUTPUT_DIR>`: absolute path of the directory where **docids.txt** is to be saved.

Output: The program should generate one file `docids.txt` in which each line is a valid TREC run record

3.2 Checklist

Shell Scripts	Python Programs	Output Files
`build.sh`		
`tokenize_corpus.sh`	`tokenize_corpus.py`	`vocab.txt`
`build_index.sh`	`build_index.py`	`index.json,` `compressed-index-files`
`retrieval.sh`	`retrieval.py`	`docids.txt`

Table 1: Source code to be submitted

In addition to the above code files, submit the following also:

1. **Report file** which includes the implementation details as well as any tuning you may have done. The PDF report should also contain details of how the experiments were conducted, what the results are –speed of construction, performance on the released queries, query execution-time etc. The report should be named as 20XXCSXX999.pdf

2. **README.txt**: Instructions to compile/run the code and reproduce outputs.

Please note the following:

- All your submissions should strictly adhere to the formatting requirements given above. You might chose to add additional files/directories/functions but it is your responsibility to integrate them and make them work correctly. You can also generate temporary files at runtime, if required, only within your directory.

- Any missing or incorrectly named files will result in penalty marks.

- All allowed dependencies of libraries that you require should be handled in your build script. You can assume that there is **no internet connectivity** during the build.

4 Evaluation Plan

Submissions will be evaluated on the same Virtual Programming Lab (VPL) environment, configured with Ubuntu Linux, which is provided for the course. It has all the allowed libraries installed on it. You must verify that all your shell scripts and programs execute successfully in VPL and produce the required output files before submission.

4.1 Evaluation Steps

The set of commands for evaluation of your submission roughly follows -

Tentative Evaluation Steps (which might be used by us)

```
$ unzip 20XXCSXX999.zip
$ cd 20XXCSXX999
$ bash build.sh
$ bash tokenize_corpus.sh <arguments>
$ bash build_index.sh <arguments>
$ bash retrieval.sh <arguments>
```

- Note that all evaluations will be done on not only the queries that are released, but on a set of held-out queries that are not part of the training set. We will use a different query file than test query.txt given in the shared folder, although the format of the query file will remain the same. Thus, your implementation should be able to handle new terms in the query.
- It is important to note that instructor / TA will not make any changes to your code – including changing hardcoded filenames, strings, etc. If your code does not compile and/or execute as expected, you will get no marks on this assignment
- There are timeouts at each stage exceeding which the program will be terminated –irrespective of whether it has completed or not. You may assume the following timeouts:

Stage	Description	Timeout
1	Tokenization and vocab construction (`tokenize_corpus.sh`)	5 minutes
2	Inverted index construction and compression (`build_index.sh`)	45 minutes
3	Decompression and Boolean Retrieval (`retrieval.sh`)	25 minutes (for 25 queries)

NOTE:

1. Evaluation may be done on a separate set of documents and queries. So ensure that your code is properly tested for all the given specifications.
2. The evaluation for ”Index-Compression” and ”Retrieval Efficiency” is competitive. The top-3 will get full marks (allotted to these two tasks), next 5 will get 90% of marks, and so on.