COL764/COL7364: Information Retrieval & Web Search
Assignment 2: Vector Space model and Probabilistic Retrieval

August 29, 2025

Deadline

Submission of the complete implementation with report on the algorithms is due on September 10, 2025, 11:59 PM.

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 on Piazza.
2. All programs must be written in **Python (3.12) or 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 or other libraries 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 Piazza).
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. **No deadline extensions** will be given. Late submission is allowed with penalty as outlined in the Introductory class. Since the 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.**
6. **Submission Details**: will be shared later.

1 Assignment Description

This assignment builds upon the work begun in Assignment 1, with the following tasks:

Task 0: (**UNGRADED**) Learn to use a third party tokenizer

Task 1: Update the structure of the index (built in the first assignment) to now include both document frequency of a term as well as term frequency of the term for each document (in addition to storing the position),

Task 2: Implement boolean *phrase search* using the positional inverted index

Task 3: Implement the Vector Space Model (VSM) to retrieve the top-k results (k can vary and will be specified as a parameter)

Task 4: Implement BM-25 ranking to retrieve top-k results (k is a variable parameter),

Task 5: (**BONUS**) Improve retrieval quality using Pseudo-Relevance Feedback

Let's gauge the retrieval efficiency for each of the ranked-retrieval-models in Tasks 3, 4 and 5, by computing, Precision, Recall and F1 scores.

1.1 Document Collection (Corpus), Queries and Qrels

Training Corpus and the Queries are same as was given in Assignment 1. The **qrels** file is additionally provided which is to be used to report the retrieval-efficiency metrics.

1.2 Download Details

The training corpus & other files are available for download in this: SharePoint Folder
Use the full corpus. Put limit on the documents only for development, if needed.

1.3 Implementation Constraints

- **Allowed modules (or built-in functions)**:
  - **Python**: Only standard Python libraries, such as `os`, `re`, `json`, `math`, `collections`, `time`, `zlib`. Use **spacy** for Tokenization only.
  - **C++**: Only C++ Standard Library headers (e.g., `<iostream>`, `<fstream>`, `<sstream>`, `<string>`, `<vector>`, `<map>`, `<algorithm>`, `<cmath>`). Use **nlohmann::json** for json parsing and **Boost.Tokenizer** for tokenization only. (We will ensure that files for these are present in the evaluation environment).
- **Disallowed (third-party or non-standard libraries):**
  - **General rule**: Don't use any external library/module other than the ones which are part of standard C++ (STL) and standard Python library with the exceptions mentioned above. Implement indexing, retrieval and metric calculation logic yourself from scratch.

2 Tasks

Task 0: Tokenizer

This is an ungraded task.
Use the tokenizer from **spacy** in Python (or **Boost.Tokenizer** in C++) to genearte vocabulary file (`vocab.txt`) which should contain a single token in each line. (Don't apply stemming during tokenization.)

Task 1: Inverted Index with positions, term frequency and document frequency.

Objective: Build an in-memory positional inverted index with term and document frequencies 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 have to save the inverted index in a JSON file in the following format:

```
{
    "term1": {
      "df": 2,
      "postings": {
        "docA": { "tf": 3, "pos": [0, 7, 20] },
        "docB": { "tf": 1, "pos": [5] }
      }
    },
    "term2": {
      "df": 1,
      "postings": {
        "docC": { "tf": 2, "pos": [4, 9] }
      }
    }
}
```

- **Positions are 0-based** within each document (in the unprocessed corpus) 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 can use any efficient data structure (not necessarily JSON) as you see fit for creating the in-memory index. We require you store that in a json file.

Function:

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

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

```
    def load_index(index_dir: str) -> object
```

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 0.
   (b) The inverted index should contain, for each token, the document frequency, the documents in which it occurs, and in each document, the positions at which this occurs, term frequency[1]. 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**
5. In this assignment, you will be implementing the vector space model and the BM25 model. Whatever additional statistics are required to implement these models, you may need to collect and store during index construction/save. Name the files vsm.X and bm25.X, where X is any format of your choice. Use the same directory as the index file to store the additional files.

Task 2: Boolean Phrase Search

Objective: Do phrase search using the positions saved in the inverted index. (It's a boolean retrieval task, tf-idf is not to be used here).
Function: Note that you have to write two functions with different signatures, one which takes a single query and returns a list of results, and another which takes a query file (in the format described earlier) and returns a qRel file suitable to compute the accuracy numbers. (format given in 3.2)

```
def phrase_search_query (query: str, index: object) -> list #given the
    query string, return all the matching documents
```

```
def phrase_search (queryFile: str, index_dir: str, stopword_file: str,
    outFile: str) -> None #given the file containing queries, write
   the results for each query in the output file
```

Task 3: Retrieval using Vector Space Model

Objective: Implement the Vector Space Model.

Function: Note that you have to write two functions with different signatures, one which takes a single query and returns a list of results, and another which takes a query file (in the format described earlier) and returns a qRel file suitable to compute the accuracy numbers.

```
def vsm_query (query: str, index: object, k: int) -> list #given the
   query string, return the top-k documents with score
```

```
def vsm (queryFile: str, index_dir: str, stopword_file: str,, k: int,
   outFile: str) -> None #given the file containing queries, number of
    results k per query, write the results for each query in the
   outFile in the output file
```

[1]Note that explicitly storing term frequency information is likely to speed up query time.

Task 4: Probabilistic retrieval

Objective: Implement the Okapi BM25 model. The model requires hyper-parameters. Study this model carefully and choose the parameters that you think are best.

Function: Note that you have to write two functions with different signatures, one which takes a single query and returns a list of results, and another which takes a query file (in the format described earlier) and returns a TREC-eval file suitable to compute the accuracy numbers.
Fine tune the hyperparameters to get the best results and hard code the finally selected hyperparameters in the code.

```
def bm25_query (query: str, index: object, k: int) -> list #given the
    query string, return the top-k documents with scores
```

```
def bm25 (queryFile: str, index_dir: str, stopword_file: str, k: int,
    outFile: str) -> None #given the file containing queries, number of
     results k per query, write the results for each query in the
    output file
```

Task 5: BONUS/OPTIONAL Pseudo-relevance feedback

Objective: Implement *ONE* round of pseudo-relevance feedback for the VSM retrieval.
It is required that the feedback loop should result in an **improved precision and recall, both**, as compared to what is obtained in Task3. If there is no improvement, then no marks will be awarded. You are free to choose the algorithm and hyperparameters here. Hard code the finally selected best hyperparameters.

NOTE: This is an optional component that you may choose to implement for a maximum of 5% additional marks at course level (that is, 5 marks if the total is 100 for the course). The team that reports the best improvement will receive the full 5%. The marks will be scaled down upto a minimum of 3% for the remaining teams.

Function: Note that you have to write two functions with different signatures, one which takes a single query and returns a list of results, and another which takes a query file (in the format described earlier) and returns a TREC-eval file (format mentioned in 3.2)

```
def prf_query (query: str, index: object, k: int) -> list #given the
    query string, return the top-k documents with scores AFTER one
    round of pseudo-relevance feedback
```

```
def prf (queryFile: str, index_dir: str, stopword_file: str, k: int,
    outFile: str) -> None #given the file containing queries, number of
     results k per query, write the results for each query in the
    outFile in the qRel 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.

Task 0: Standard Tokenizer

The program for this task should be named as `tokenize_corpus.{py|cpp}`.

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_STOPWORD_FILE> <VOCAB_DIR>
```

where:

`<CORPUS_DIR>`: absolute path of the directory containing all the training documents only (in the same format as in Assignment 1). (you must use the entire set of documents here).

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

`<VOCAB_DIR>`: 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.

Task 1: Inverted-Index :

The program should be named as `build_index.{py|cpp}` and should contain the `build_index()`, `save_index()` and `load_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>
```

where:

`<CORPUS_DIR>`: Same as in Task 0.

`<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

Output: Generate one file `index.json` which contains the inverted index postings with tf-idf values.

Task 2: Boolean Phrase-Search:

The program should be named as `phrase_search.{py|cpp}` and should contain the `phrase_search_query()` and `phrase_search()` functions.

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

```
phrase_search.sh <INDEX_DIR> <QUERY_FILE_PATH> <OUTPUT_DIR> <
   PATH_OF_STOPWORDS_FILE>
```

where:

`<INDEX_DIR>`**:** absolute path of the directory where **index.json** is saved

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

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

`<PATH_OF_STOPWORDS_FILE>`**:** Same as in Task 0

Output: Genearte one file `phrase_search_docids.txt` (format mentioned in 3.2)

Task 3: Vector Space Model:
The program should be named as `vsm.{py|cpp}` and should contain the `vsm_query()` and `vsm()` functions.

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

```
vsm.sh <INDEX_DIR> <QUERY_FILE_PATH> <OUTPUT_DIR> <
   PATH_OF_STOPWORDS_FILE> <k>
```

where:

`<INDEX_DIR>`**:** absolute path of the directory where **index.json** is saved

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

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

`<PATH_OF_STOPWORDS_FILE>`**:** Same as in Task0

`<k>`**:** Count of the k-most relevant documents to be retrieved for each query

Output:

(a) The program should generate one file `vsm_docids.txt` (format mentioned in 3.2)

(b) Based on the qrels file provided, report the precision, recall and F1 score.

Task 4: BM-25 Retrieval :
Your submission should also consist of another program called `bm25_retrieval.{py|cpp}` . It should contain the `bm25_query()` and `bm25()` functions.

This program will be executed using a shell script (to be provided by you) named `bm25_retrieval.sh`. The shell script will be invoked via terminal as follows:

```
bm25_retrieval.sh  <INDEX_DIR> <QUERY_FILE_PATH> <OUTPUT_DIR> <
   PATH_OF_STOPWORDS_FILE> <k>
```

where:

`<INDEX_DIR>`**:** absolute path of the directory where **index.json** is saved

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

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

`<PATH_OF_STOPWORDS_FILE>`: Same as in Task0

`<k>`: Count of the k-most relevant documents to be retrieved for each query

Output:

(a) The program should generate one file `bm25_docids.txt` (format given in 3.2)

(b) Based on the qrels file provided, report the precision, recall and F1 score.

Task 5: Relevance Feedback:

Your submission should also consist of another program called `feedback_retrieval.{py|cpp}` .

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

```
feedback.sh  <INDEX_DIR> <QUERY_FILE_PATH> <OUTPUT_DIR> <
   PATH_OF_STOPWORDS_FILE> <k>
```

where:

`<INDEX_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.

`<PATH_OF_STOPWORDS_FILE>`: Same as in Task0

`<k>`: Count of the k-most relevant documents to be retrieved for each query

Output:

(a) Generate one file `feedback_docids.txt` (format given in 3.2)

(b) Based on the qrels file provided, report the precision, recall and F1 score.

3.2 Output Files format

Each line is a valid TREC run record as shown (header is not to be written):

Output Format

qid	docid	rank	score
Q	SZF08-175-870	5	0.31

- **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).

3.3 Checklist

Shell Scripts	Python/CPP Programs	Output Files
`build.sh`	—	—
`tokenize_corpus.sh`	`tokenize_corpus.{py|cpp}`	`vocab.txt`
`build_index.sh`	`build_index.{py|cpp}`	`index.json`,
`phrase_search.sh`	`phrase_search.{py|cpp}`	`phrase_search_docids.txt`
`bm25_retrieval.sh`	`bm25_retrieval.{py|cpp}`	`bm25_retrieval_docids.txt`
`vsm.sh`	`vsm.{py|cpp}`	`vsm_docids.txt`
`feedback.sh` (optional)	`feedback_retrieval.{py|cpp}`	`feedback_docids.txt`

Table 1: Source code to be submitted

In addition to the above code files (only, not output 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 `<team_name>.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.
- Ensure before the submission that all your shell scripts and programs execute successfully in the baadal-environment and produce the required output files.

4 Evaluation Plan

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 <team_name>.zip
$ cd <team_name>
$ bash build.sh
$ bash tokenize_corpus.sh <arguments>
$ bash build_index.sh <arguments>
$ bash phrase_search.sh <arguments>
$ bash vsm.sh <arguments>
$ bash bm25_retrieval.sh <arguments>
$ bash feedback.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 (`build_index.sh`)	45 minutes
3	Boolean Phrase Search (`phrase_search_retrieval.sh`)	25 minutes (for 25 queries)
4	Vector Space Model (`vsm.sh`)	25 minutes (for 25 queries)
5	BM-25 Retrieval (`bm25_retrieval.sh`)	25 minutes (for 25 queries)
6	Relevance Feedback (`feedback.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 ranked-retrieval (task 3 and 4) is competitive. The top-3 will get full marks (allotted to these two tasks), next 5 will get 90% of marks, and so on.
3. Don't do stemming and lemmatization during tokenization (it will result in 0 marks).
4. In Task 3, all the Queries will be mono-phrase. Hence, connectives (AND, OR, NOT) are to be treated as stopwords only.