University of Aveiro, Information Retrieval, 2024/2025

Assignment 1

In this assignment, you will implement a complete **information retrieval system** capable of **indexing** and **searching** a given document collection. This project will give you hands-on experience with core IR concepts, including document parsing, tokenization, indexing algorithms, and search ranking models.

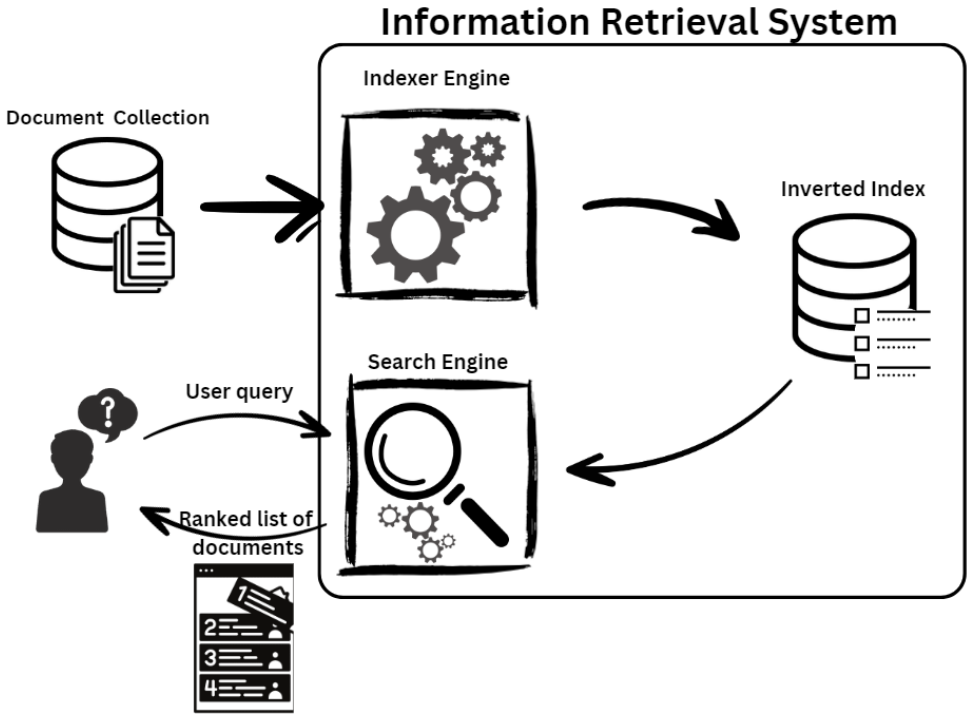

Figure 1: Overview of the components in an Information Retrieval System.

The Figure above gives you an overview of the major components in an IR system. For this assignment, you will need to implement from scratch an indexer and search engine following the requirements listed below.

1. Document collection

You will work with a compiled version of the annual 2024 MEDLINE baseline as your corpus. This collection consists of biomedical and life sciences journal citations and abstracts. We also provide a question dataset to help the development of the search engine.

Corpus (available in elearning "*Dossier/Assignment1/ MEDLINE_2024_Baseline.jsonl*"):

- 500 000 documents.
- 1455 average characters length.
- 719 MB in disk.

Questions ("*questions.jsonl*"):

- 100 evaluation questions.
- Average of 9.81 relevant documents per question.

2. Indexer Engine

The indexer engine has the responsibility of parsing a given document collection and build an inverted index. Your task is to implement the Single-Pass In-Memory Indexing (**SPIMI**) algorithm to create an inverted index for the provided collection. The index should be a **positional index**, where the posting list includes the term position in the corresponding document. When the indexing is finished, all index structures (dictionary, posting lists, document mapping, and other necessary information like metadata) should be **saved on disk** so the Searcher can read it.

Although the students have the freedom to implement and design the indexer architecture, we at least expect to see the following components:

- **Corpus Reader**: Implement a dedicated parser to read and process the MEDLINE documents.
- **Tokenizer**: Implement a flexible tokenizer with configurable options. Users should be able to set a **minimum token length**, choose whether to normalize to **lowercase**, filter **stop words** based on a provided list, and apply **stemming**.
- **SPIMI Indexer**: Implement the SPIMI algorithm, creating positional posting lists that include term positions (positional index).
    - Although the provided document collection is small enough to fit in memory, we want you to simulate working with larger datasets. We will value an implementation that forces your SPIMI algorithm to create partial indexes. Set a limit of 10,000 documents for each partial index. Once this limit is reached, write the current partial index to disk, clear the in-memory structures, and start a new partial index for the next batch. After processing all documents, merge these partial indexes into a single, coherent inverted index.

3. Searcher Engine

The searcher engine is where your system will demonstrate its retrieval capabilities. Develop a search engine that can **load the created index** from disk and provide ranking functionality for user queries. Note that your query processing should use the same tokenization rules as the indexer to ensure consistency. Tip: Take advantage of the index metadata to store the tokenizer configuration that was used to create that index.

By default, configure your searcher to **return a maximum of 100 documents** for each query. Optionally, design this as a flexible parameter that users can adjust when interacting with the searcher, allowing them to customize the number of results based on their needs.

Ranking Model:

- BM25 ranking: Implement the BM25 ranking algorithm and allow for parameterization of k1 and b parameters (assume by default, the values k1=1,2; b=0,75).
- In addition, you may implement a vector space model with tf-idf weights, allow for a parameterization following the SMART (for instance lnc.ltc).

The output format of the ranking list should follow the same structure as the question input file:

File name: ranked_questions.jsonl

{“id”:“<question_id>”,
“question”: “<question_text>”,
“retrieved_documents”: [“<doc_id_pos_1>”, “<doc_id_pos_2>”, ..., “<doc_id_pos_100>”,]},
...

4. Implementation Notes

To interact with your system, develop a comprehensive Command-Line Interface (CLI) that allows users to control both the indexer and searcher components. The CLI should provide options for specifying input and output directories, setting tokenizer options, loading specific indexes, entering queries interactively or processing batches (questions.jsonl), and customizing ranking parameters. Furthermore, we recommend exploring the best configuration of the indexer and searcher parameters. To measure the efficacy of your system you should run the given questions and compute the nDCG@10 metric.

Moreover, the programming language is free of choice. However, we recommend python or java for a better support.

5. Submission Requirements

Your submission should include:

- README file details what was implemented and the possible configurations for executing the system.
- Well-documented source code for all components.
- The obtained ranked lists for the 100 evaluation questions. (called it ranked_questions.jsonl)
- The code needs to be **reproducible**. So do not forget to add a **setup script** that installs and configures all dependencies. Also, include an **entry point script** with examples of how to run your indexer and searcher.

6. Evaluation Criteria

Your assignment will be evaluated based on the following criteria:

- Work progression along the time
- Presentations
- Information retrieval components
    1. Correctness and completeness of implementation
    2. Final ranking metrics (nDCG@10)
- Software engineering components
    1. Quality of the code, organization, design patterns and structures
    2. Overall computational efficiency