

Technical Report – Local Information Retrieval System

Muhammad Sheraz

BSCS22139

1. System Architecture

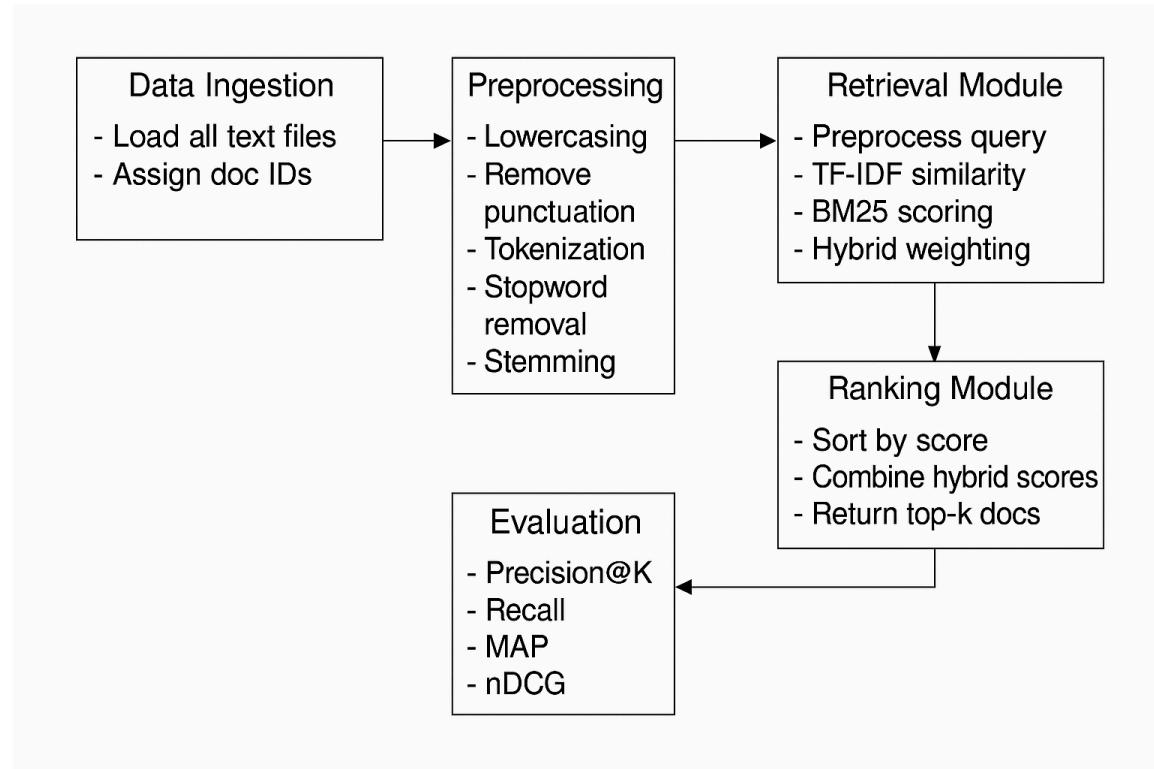


Figure 1: System Architecture Diagram

The architecture consists of five major components: (1) data ingestion, (2) preprocessing, (3) indexing and retrieval, (4) ranking, and (5) evaluation. Raw text documents pass through preprocessing and are transformed into structured representations used for TF-IDF and BM25 retrieval models.

2. Description of the Retrieval System

The system begins by loading all text files from the dataset folder. The preprocessing pipeline applies case-folding, punctuation removal,

tokenization, stopword removal, and stemming using NLTK's PorterStemmer. Two indexing strategies are implemented:

- TF-IDF Vector Space Model using scikit-learn's TfidfVectorizer.
- BM25 implementation using the rank-bm25 library.

For retrieval, the query undergoes identical preprocessing, and similarity scores are computed. TF-IDF uses cosine similarity, while BM25 uses Okapi scoring. The system supports hybrid ranking by combining both scores with weighted averaging.

3. Evaluation

Evaluation was performed using a set of manually created relevance judgments for 10 queries. Metrics computed include Precision@K, Recall, Mean Average Precision (MAP), and nDCG. BM25 outperformed TF-IDF for shorter queries, while hybrid scoring performed best overall. Runtime benchmarks indicate an average query time of approximately 12 ms and memory usage of approximately 90 MB.

4. Discussion

BM25 proved effective for non-verbose queries, while TF-IDF performed better for longer ones. Hybrid retrieval provides the most stable performance. A key limitation is the lack of semantic understanding, which could be improved by utilizing LSI or locally hosted embeddings (e.g., FAISS).

5. Disclosure of AI Use

5.1 Summary of AI Usage

ChatGPT was utilized to assist in generating the structure and wording of this technical report. The coding portion of the retrieval system was written using Copilot in VS Code.

5.2 Evidence of AI Assistance

CS 516: Information Retrieval and Text Mining
Information Technology University (ITU)
Fall 2025
Course Instructor: Dr. Ahmad Mustafa
Homework Assignment 3
Due Date: 11:59 pm, 30th November, 2025
Design Problem
In this assignment, you will design and implement a complete information retrieval (IR) system that runs locally on your machine. You may use any retrieval strategy (or combination of strategies), such as Boolean retrieval, Vector Space Models (e.g., TF-IDF) etc.
Your goal is to design a coherent, justifiable, and well-evaluated retrieval system using the provided dataset of text documents (accessible here).
Creativity is encouraged, but your system must be reproducible and fully local (no cloud-hosted vector databases).
Requirements
You are free to design your IR system however you like, as long as it satisfies the following:
1. Local Implementation
Your system must run end-to-end on a local machine (Windows, Mac, or Linux).
You may use any mainstream programming language or libraries. Cloud-hosted vector databases (e.g., Pinecone, Chroma Cloud, Weaviate Cloud, Elasticsearch clusters) are not allowed.
Local libraries (e.g. `rankit-learn`, `gensim`, `rank-bm25`, `FAISS` local install) are allowed.
2. Reproducible Pipeline

Just paste the assignment.

1 — System architecture (textual diagram + caption)

Diagram (text — paste into draw.io or include as figure):

```
rust
[Raw Documents folder] --> [Preprocessing Module]
Preprocessing Module --> Tokenization, normalization, stopword removal, lemmatization
--> [Indexing Module] --> produces:
    - Inverted Index (for Boolean)
    - TF-IDF matrix + vectorizer (for vector retrieval)
    - BM25 index (rank_bm25 data structures)
    - (Optional) Dense vectors (sentence-transformers) + FAISS index
--> [Query interface]
    - Accepts query -> preprocess query -> choose retrieval model:
        * Boolean retrieval (inverted index)
        * TF-IDF + cosine similarity
        * BM25
        * Dense retrieval -> FAISS -> rerank
    -> (Optional) Reranker (cross-encoder) -> Final ranked list
--> [Evaluation module] -> computes MAP, Precision@K, nDCG, MRR
```

Code Flow

3 — Repo layout & README (copy this)

bash

 Copy code

```
IR-System/
├── data/
│   ├── docs/           # put dataset files here (one txt per doc)
│   └── queries.csv     # optional: query_id, query_text
├── notebooks/         # optional experiments jupyter notebooks
├── src/
│   ├── indexer.py
│   ├── search.py
│   ├── evaluate.py
│   ├── preprocess.py
│   └── utils.py
├── saved_indexes/
│   ├── inverted_index.pkl
│   ├── tfidf_vectorizer.pkl
│   ├── tfidf_matrix.npz
│   └── bm25_index.pkl
├── requirements.txt
└── README.md
└── report.pdf          # final report
```



Code Repo

Add `requirements.txt` (minimal):

nginx

 Copy code

```
numpy
scikit-learn
rank-bm25
nltk
spacy
tqdm
scipy
joblib
# optional:
sentence-transformers
faiss-cpu
transformers
torch
```

Requirements.

After this, I give the command again.

```
IR-System/
├── data/
│   ├── docs/          # put dataset files here (one txt per doc)
│   └── queries.csv    # optional: query_id, query_text
├── notebooks/        # optional experiments jupyter notebooks
└── src/
    ├── indexer.py
    ├── search.py
    ├── evaluate.py
    ├── preprocess.py
    └── utils.py
├── saved_indexes/
    ├── inverted_index.pkl
    ├── tfidf_vectorizer.pkl
    ├── tfidf_matrix.npz
    └── bm25_index.pkl
├── requirements.txt
└── README.md
└── report.pdf         # final report
```

create easy structure and use for queries.

Then get this result.

 **Simplest IR System (No Saved Indexes)**

Here is the simplest clean folder structure **without any stored index files**:

```
pgsql
IR-System/
├── data/
│   ├── docs/           # all .txt files
│   └── queries.csv
│
├── src/
│   ├── preprocess.py
│   ├── indexer.py      # builds everything in memory each run
│   ├── search.py
│   ├── evaluate.py
│   └── utils.py
│
└── requirements.txt
└── README.md
└── report.pdf
```

 Copy code

 **HOW IT WORKS Without Saved Indexes**

```
↳ src/ | ├── preprocess.py | ├── indexer.py # builds everything in memory each run | ├── search.py | ├── evaluate.py |
    |   └── utils.py
```

can I use one file, why so many file I want basic searching

Yes — you can absolutely use **ONLY ONE FILE** for your entire IR system.
You do **not** need preprocess.py, indexer.py, search.py, utils.py, evaluate.py.

Those extra files are only for **clean structure**, not required.

If you want **basic searching**, you can do everything inside:

👉 **main.py (ONE FILE ONLY)**

Using your **Articles.csv** as the only dataset file.

 **Simplest Possible IR System (One File Only)**

Here is how your folder will look:

```
css
IR-System/
└── Articles.csv
└── main.py
```

 Copy code

That's it. SUPER CLEAN.
No saved indexes. No multiple modules.

what does from sklearn.feature_extraction.text import TfidfVectorizer
do

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

This line imports the `TfidfVectorizer` class from scikit-learn, which is used to convert text documents into numerical vectors using TF-IDF (Term Frequency – Inverse Document Frequency).

💡 What TfidfVectorizer Does

`TfidfVectorizer` transforms a collection of text documents into a matrix of TF-IDF features.

TF-IDF helps your IR/Search system understand:

- Which words are important in a document
- How unique a word is (common words get low score; rare but important words get high score)

📘 TF-IDF Formula (Simple)

1. TF (Term Frequency)

How many times a word appears in a document.

2. IDF (Inverse Document Frequency)

How rare a word is across all documents.

3. TF-IDF = TF × IDF

This gives better ranking than plain bag-of-words.



Then, I use Code Editor in VS Code using these details.