WikipediaMoviesRetrieval

Assignment 1

Information Retrieval Course

Academic Year 2025/2026

Group Members:

Alperen Davran	[Student ID]
Matteo Comi	[Student ID]
Shakh [Surname]	[Student ID]
Amin Borqal	s0259707

University of Antwerp
Faculty of Science
Department of Computer Science

November 1, 2025

Contents

1 Introduction

This project implements a compact, inspectable information retrieval (IR) pipeline over the Wikipedia Movies dataset. The emphasis is on a clear, minimal design that is easy to reason about and verify on a medium-sized collection. Each document is indexed as the concatenation of its `title` and `plot`; the original title is also stored as metadata for display.

Scope:

- Tokenization and normalization (lowercasing, punctuation removal, stopword filtering and stemming).
- Inverted indexing with Single-Pass In-Memory Indexing (SPIMI), plus a disk-backed variant and a small updatable mode.
- Query processing and ranking.

Non-goals:

- Distributed or industrial-scale indexing.
- Heavyweight external IR frameworks.

The implementation uses small, explicit Python modules. The design follows standard IR practice (e.g., Manning et al.) while keeping the code pragmatic for a 18k-document collection.

1.1 System Architecture

Three modules with clear contracts:

1. **Tokenizer** — Input: raw title+plot. Output: list of normalized tokens.
2. **Indexer** — Input: (doc id, tokens). Output: inverted index in memory or on disk.
3. **Query Processor** — Input: user query. Output: ranked doc ids with scores.

Data flow: `CSV data -> Tokenizer -> Indexer -> Query Processor`
Key in-memory structure (SPIMI):

```
index = { term: { doc_id: tf } }
```

with auxiliary statistics (e.g., df, cf, per-document length) maintained for scoring.

1.2 Dataset

We use the Wikipedia Movies dataset from Kaggle (https://www.kaggle.com/datasets/exactful/wikipedia-movies). It contains approximately 17.8k movies across the 1970s–2020s. The CSV schema is `title,image,plot`. The `image` field is a URL and is not used for indexing; we index only the textual fields below:

- **title** (string)
- **plot** (string)

The document text is defined as `title + " " + plot`. Each record is assigned a sequential integer document id at load time. Cleaning and normalization are handled by the tokenizer; the source CSVs are left unchanged.

2 Tokenizer [To Adjust and Complete]

The tokenizer converts raw text into a normalized sequence of terms suitable for indexing and scoring. We use NLTK (Natural Language Toolkit), a Python library for natural language processing (https://www.nltk.org/), and follow standard guidance from *Introduction to Information Retrieval* (IIR §2.2–§2.3) on tokenization and normalization.

Processing steps:

- **Tokenization (nltk py library)**: `word_tokenize` over the concatenated `title + plot` string.
- **Lowercasing**: all tokens are converted to lowercase.
- **Filtering**: keep alphabetic tokens only (punctuation, numbers removed).
- **Stopwords**: remove English stopwords (NLTK list).
- **Stemming (optional)**: Porter stemmer; enabled by default.

Design notes (IIR §2.3): stopword removal reduces noise from high-frequency function words; stemming conflates morphological variants and can improve recall at small cost to precision. For our mid-size collection, the simplicity and speed of Porter stemming are a pragmatic fit.

Current usage in code:

- **Analysis/EDA**: `Components/Tokenizer.py` is used to create tokens for statistics and exploration.
- **Indexing**: the indexer uses a minimal built-in tokenizer (lowercasing + alphanumeric split) unless an external tokenizer is injected. This keeps dependencies light for the core SPIMI loop.

If desired, the same NLTK tokenizer can be injected into the indexer to unify preprocessing (e.g., pass a tokenizer to `Indexer.init_memory(...)`).

3 Indexer

The indexer component implements three variants of inverted index construction, each suited for different use cases and scalability requirements.

3.1 Memory Index – SPIMI Approach

Reference: IIR §4.3

The memory index is based on the SPIMI (Single-Pass In-Memory Indexing) algorithm described in IIR §4.3. The book emphasizes that SPIMI avoids sorting and writes complete inverted blocks directly to memory: *"SPIMI generates a complete inverted index for a block without sorting the terms."*

In our implementation, each document is tokenized and inserted into a Python dictionary:

```
index = { term: { doc_id: tf } }
```

The dictionary grows automatically as new terms appear, thanks to Python's hash-based data structure. This design keeps insertions roughly O(1) and enables quick term lookups.

Because our dataset is small, we keep the entire index in memory. For larger collections, as explained in IIR Figure 4.4, SPIMI can be extended to write blocks to disk and merge them later. The lecture slides also visualized this "block → partial index → final merge" workflow step by step.

3.2 Disk Index – Term-per-File and Lazy Loading

Reference: IIR §5.2, §5.3

The disk version focuses on storing the index persistently. Each term is stored in a separate file, making it easy to test and inspect. To avoid naming conflicts, we used MD5 hashing for term filenames:

```
path = index_dir + "/terms/" + hashlib.md5(term.encode()).hexdigest() + ".pkl"
```

Alongside term files, we keep a `lexicon.pkl` file with metadata such as `df` (document frequency), `cf` (collection frequency), and file path. This matches the structure described in IIR §5.2: *"An inverted index consists of a dictionary and a set of postings lists stored on disk."*

When querying, only the relevant term's file is loaded — this lazy-loading design aligns with the assignment goal of reading only the necessary index parts into memory. In class, this was highlighted under the slide "Efficient Index Storage and Access."

3.2.1 Compression Implementation

Reference: IIR §5.3 To make the disk index smaller, we implemented both **gap encoding** and **Variable-Byte (VB)** compression. According to IIR §5.3, "gap encoding followed by variable byte coding gives a good balance between compression and speed." Our implementation compresses each postings list as follows:

```
def _compress_postings(postings):
    gaps = _gap_encode(postings)
    compressed = bytearray()
    for gap, tf in gaps:
        compressed.extend(_vb_encode(gap))
        compressed.extend(_vb_encode(tf))
    return bytes(compressed)
```

This combination reduced file size noticeably while keeping decoding fast. We chose VB instead of gamma codes because it is simpler, faster to decode in Python, and more suitable for small to mid-size datasets — as also explained in the "Compression Techniques" lecture slide.

3.3 Updatable Index – Auxiliary Index and Merge

Reference: IIR §4.5

The third mode adds support for updates, insertions, and deletions. We followed the dynamic indexing model from IIR §4.5, which recommends keeping a small auxiliary index in memory and merging it periodically with the main on-disk index. This approach supports incremental updates efficiently without rebuilding the entire index.

3.3.1 Structure and Components

The updatable indexer maintains three main parts:

- **Main on-disk index:** The compressed postings built earlier.
- **Auxiliary in-memory index (`aux`):** Stores new or modified documents before merging.
- **Tombstone set (`deleted`):** Tracks document IDs marked as deleted.

Initialization example:

```
def init_upd(index_dir="updindex", tokenizer=None):
    base = init_disk(index_dir, tokenizer)
    load_disk_min(base)
    base.update({
        "aux": defaultdict(dict),
        "deleted": set()
    })
    return base
```

3.3.2 Adding and Deleting Documents

New documents are first indexed into the auxiliary memory index:

```
def add_upd(st, title, text):
    did = st["next_id"]
    st["next_id"] += 1
    toks = re.findall(r"[a-z0-9]+", text.lower())
    for t, tf in Counter(toks).items():
        st["aux"][t][did] = tf
    return did
```

To delete a document, we simply mark it using a tombstone (instead of rewriting the entire index immediately):

```
def delete_upd(st, doc_id):
    st["deleted"].add(doc_id)
```

This idea matches the explanation in IIR §4.5: *"We maintain a small auxiliary index in memory and occasionally merge it with the main index."* In the lecture on dynamic indexing, this approach was illustrated as *"Fast updates using auxiliary memory and periodic merges."*

3.3.3 Merging Step

When a merge is triggered, the auxiliary postings are merged with the main index, deleted documents are removed, and the postings are re-compressed:

```
def merge_upd(st):
    for term in st["aux"].keys():
        main = postings_disk(st, term) if term in st["lex"] else {}
        main.update(st["aux"][term])
        for d in list(main.keys()):
            if d in st["deleted"]:
                del main[d]
        compressed = _compress_postings(main)
        with open(_term_path(st["dir"], term), "wb") as f:
            f.write(compressed)
```

This mirrors the merge mechanism described in both the book (IIR §4.5) and the "Dynamic Indexing" lecture slides.

3.3.4 Design Rationale

We decided not to implement the more complex *logarithmic merging* model (IIR §4.5) because it is meant for large-scale systems. Our single-level merge model is simpler and demonstrates the same concept clearly for small datasets. Logarithmic merging reduces the overall update cost from $O(T^2)$ to $O(T \log(T/n))$, but for this project, simplicity and clarity were our priorities.

3.3.5 Implementation Outcome

This version completes the full indexer lifecycle — from initial indexing to compression and dynamic updates — fully reflecting the concepts taught in both the book and the course slides.

4 Query Processing

[To be completed]

4.1 Vector Space Model (VSM)

[To be completed]

4.2 BM25 Ranking

[To be completed]

4.3 Language Models

[To be completed]

5 Conclusion

[To be completed]

References

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.
Available at: https://nlp.stanford.edu/IR-book/