Information Retrieval Assignment Report

Naveeta Maheshwari (2024AIZ8309)
Sagar Singh (2025SIY7574)

Course: Information Retrieval and Web Search (COL764/COL7341)
Assignment 1

Introduction

The goal of this assignment was to implement an end-to-end Boolean retrieval system for a large collection of documents. The pipeline involved tokenization and vocabulary construction (`tokenize_corpus.py`), inverted index construction (`build_index.py`), index compression using Variable-Byte with Gap Encoding, and finally, decompression and query evaluation (`retrieval.py`). The report concludes with experiments analyzing the system's performance, including index construction speed and query execution time.

Tokenization & Vocabulary Construction

File: `tokenize_corpus.py`

Algorithm

For each JSON document in the corpus, non-ASCII characters and digits are removed, text is converted to lowercase, split into tokens, and filtered against a provided stopword list.

Implementation Details

- Used Python's `re.sub` for efficient removal of digits and non-ASCII characters.
- Tokens are stored in a Python `set` to ensure uniqueness and efficient lookup.
- The final vocabulary is sorted alphabetically and written to `vocab.txt`.

Output

A `vocab.txt` file containing the sorted list of unique tokens.

Inverted Index Construction

File: `build_index.py`

Algorithm

A positional inverted index is constructed by iterating through all documents in the corpus. For each token, its document ID and position(s) are added to a postings list.

Data Structure

```
inverted_index[token][doc_id] = [pos1, pos2, ...]
```

Saving

The complete index is saved as a JSON file (`index.json`) with alphabetically sorted terms, lexicographically sorted document IDs, and numerically sorted positions.

Tuning

Utilized Python's `collections.defaultdict` to simplify the code and slightly improve performance by reducing the need for key existence checks.

Compression with Variable-Byte (VB) + Gap Encoding

File: `build_index.py`

Algorithm

The system employs a two-stage compression pipeline to reduce the index size significantly.

1. **Gap Encoding:** Instead of storing absolute values for document IDs and positions, we store the *difference* (gap) between successive entries. For example, a sorted list of document IDs '[103, 110, 115]' becomes '[103, 7, 5]'. This results in a sequence of smaller integers, which are highly compressible.

2. **Variable-Byte (VB) Encoding:** Each integer is then encoded using a variable number of bytes. Numbers are broken into 7-bit chunks, and the most significant bit of each byte is used as a continuation flag (0) or a stop flag (1). This allows small numbers, which are frequent after gap encoding, to be stored in a single byte.

Output Files

- **postings.bin:** A binary file containing the compressed postings lists.
- **lexicon.tsv:** A lookup file mapping each term to its document frequency, byte offset, and data block length in `postings.bin`.

- **docmap.tsv:** A mapping from the original string-based document IDs to the internal integer IDs.
- **meta.json:** A file describing the format and structure of the compressed files.

Decompression and Query Processing

File: `retrieval.py`

Decompression Algorithm

1. The `docmap.tsv` and `lexicon.tsv` metadata files are loaded into memory.
2. The entire `postings.bin` file is read.
3. For a given term, its byte offset and length are retrieved from the lexicon.
4. The corresponding byte slice is extracted and decoded by reversing the VB and Gap encoding processes to reconstruct the original postings list.

Query Processing Steps

1. **Tokenization:** Queries are tokenized, lowercased, and filtered using the stopword list. Implicit `AND` operators are inserted between adjacent terms.
2. **Parsing:** The infix query is converted to a postfix (Reverse Polish Notation) expression using the Shunting-yard algorithm, which respects operator precedence (NOT > AND > OR).
3. **Evaluation:** The postfix expression is evaluated using a stack and set operations: `AND` maps to set intersection, `OR` to union, and `NOT` to set difference against the set of all documents.
4. **Results:** The final set of matching document IDs is sorted and written to `docid.txt`.

Experiments

Experiments were conducted on the CORD-19 dataset (˜192,000 documents). Performance times were measured for each stage of the pipeline, and the final index size was compared to the uncompressed version.

Performance Timings

Index Size Comparison

Conclusion

In this assignment, a complete Boolean retrieval engine was successfully implemented. The pipeline included vocabulary construction, inverted index construction, efficient index compression using Variable-Byte and Gap Encoding, Decompression, and a robust query evaluation system.

Stage	**Time Taken (seconds)**
Vocabulary Construction	˜14.65
Inverted Index Build (JSON)	˜107.00
Compression	˜69.27
Decompression	˜30.46
Query Retrieval (50 queries)	˜1.57

Table 1: Performance timings for each stage of the pipeline.

File / Format	**Size**
Uncompressed (`index.json`)	501 MB
Compressed Total	**˜70 MB**
`postings.bin`	52 MB
`lexicon.tsv`	12.52 MB
`docmap.tsv`	2.9 MB

Table 2: Comparison of index size before and after compression.

The experimental results demonstrate the system's efficiency. The entire indexing and compression process for approximately 192,000 documents was completed in under 3.5 minutes. The compression strategy proved highly effective, reducing the index size from 501 MB to approximately 70 MB, achieving a space saving of over 85% (a ˜7x reduction). This significant reduction in size contributes to faster load times and query evaluation, with 50 queries being processed in just over 1.5 seconds. Overall, the system is efficient, scalable, and well-suited for handling large document collections.