COMP 479

Information Retrieval and Web Search

Dr. Sabine Bergler

Project 1

Report

October 13th, 2024

Louis-Charles Marquis 40177137

Gina Cody School of Engineering and Computer Science

Concordia University

1. PreProcessor

The design is efficient and modular since each pre-processing step is encapsulated in separate functions. This modularity enhances readability and maintainability.

2. PrimaryIndex

This file’s purpose is to tokenize documents from the Reuters dataset and create a dictionary (the index) where each token is associated with the documents it appears in. Once the index is built, it is stored using pickle, a Python module used to serialize and deserialize Python objects. This module is useful for saving complex data structures, such as dictionaries, to a file or sending them over a network, and then later loading them back into memory without having to reconstruct them from scratch every time.

The indexing process is efficiently implemented using a dictionary to store term-document mappings. The use of defaultdict(list) simplifies appending document IDs and avoids key errors. While this is a good start for handling Boolean queries (e.g., single term, AND, OR queries), the logic could benefit from more complex token processing (e.g., handling phrases or proximity queries). The modularity makes it easy to integrate with the QueryProcessor for retrieval tasks.

3. PositionalIndex

The PositionalIndex builds on the concept of the primary inverted index but adds positional information, allowing retrieval systems to perform proximity searches. In this case, each token maps not only to the document IDs but also to the positions within those documents where the token appears.

The positional information is stored and retrieved logically, but the memory footprint could be large for extensive datasets. Future improvements might include compression techniques to reduce storage needs.

4. lossyCompression

This file focuses on lossy compression techniques to reduce the size of the index. It applies various filters such as number removal, case folding, stopword removal and stemming to minimize the size of the dictionary before the indexing process begins.

The compression implementation is effective at reducing redundancy and index size by normalizing the token set. The modular approach in applying filters ensures that the compression is flexible. Users can easily apply different levels of compression based on their needs. The impact of these compression techniques on retrieval accuracy, however, may vary and should be further explored in an evaluation stage to ensure performance is not significantly affected.

5. QueryProcessor

This file processes single-term, AND and OR queries and fetches the relevant documents by interacting with the primary index file that was saved upon its creation.

The Boolean retrieval implementation is straightforward and efficient. The design supports flexible querying while maintaining ease of integration with the indexing system. The use of set operations optimizes query processing by enabling quick intersection of document lists. However, it could be expanded to support more complex queries (e.g., phrase searches). Additionally, the program could benefit from more robust error handling for edge cases.

6. NearOperator

Two versions of the operator are provided: one optimized using a two-pointer technique and a brute-force version.

The use of a two-pointer technique in the optimized near\_operator function improves the performance of proximity queries compared to the brute force approach. Instead of iterating over all combinations of document IDs and positions, which takes O(n\*m) time, we only move the pointers in one list when necessary, making the algorithm linear in the total number of documents containing the terms (O(n + m)); where n is the number of documents where term1 appears and m is the number of documents where term2 appears. This design will scale well for large datasets.

The near operator function uses the positional index file that was saved upon its creation.

7. ConcordanceOperator

My implementation of the concordance query achieves the desired functionality by efficiently leveraging a positional index for document retrieval and query processing. While it performs well for individual queries, further optimizations such as token caching and parallelism could enhance performance for repeated queries or larger datasets.