HomeWork-2 Report Template

INDEX SIZE

Compressed | Stemmed	69.4 MB
Decompressed | Stemmed	143.6 MB
Decompressed | Unstemmed	148.1 MB

<u>Brief Explanation on the process used for Indexing</u>

- *Parsing Documents:*
  - *The process starts by parsing a collection of documents. Each document is extracted and divided into metadata fields (such as DOCNO, FILEID, HEAD, etc.) and the main text content.*
- *Preprocessing Text:*
  - *The text content of each document undergoes preprocessing. This includes tokenization, removal of stop words and punctuation, and stemming (reducing words to their root form). Preprocessed text is then stored in separate fields for stemmed and unstemmed versions.*
  - *Mappings are created to link each token, doc name to a unique identifier, aiding efficient retrieval and storage during indexing and querying.*
- *Batch Processing:*
  - *To handle large document collections efficiently, the indexing process is performed in batches. Each batch of documents is processed sequentially to generate the inverted index and catalog.*
- *Indexing:*
  - *The documents' preprocessed text is indexed to facilitate efficient retrieval. For each term in the text, an inverted index is created, mapping each term to the documents and positions where it occurs. This allows for quick lookup of documents containing specific terms.*
- *Catalog Creation:*
  - *Alongside the inverted index, a catalog is generated. This catalog contains metadata about each term in the index, including its offset and length in the index file. This metadata enables fast access to specific terms within the index.*
- *Merge Indices :*
  - *The merging process works without processing everything into memory which involves a sequential approach where data is processed in chunks and written directly to files without storing the entire dataset in memory simultaneously. This is done by comparing each element in the catalog and updating their offset which is explained in detail later.*
  - *During the merging process, zlib compression is applied to the document information before it is written into the merged index file. I specifically opted for level 6 compression, which offers a good balance between compression ratio and speed.*

MODEL PERFORMANCE

Index	**Model**	**Old Score**	**New Score**	**Percent (New/Old)**
Decompressed | Stemmed	TF-IDF	0.2907	0.2570	0.89 (0.004 diff)
Decompressed | Stemmed	Okapi BM-25	0.2809	0.2665	0.95
Decompressed | Stemmed	Unigram LM with Laplace smoothing	0.2316	0.2180	0.94
Decompressed | Unstemmed	TF-IDF		0.2126	
Decompressed | Unstemmed	Okapi BM-25		0.2214	
Decompressed | Unstemmed	Unigram LM with Laplace smoothing		0.1995	
Compressed | Stemmed	Okapi BM-25		0.2665	

<u>Inference on above results</u> *(Make sure to address below points in your inference)*

1. *Explain how index was created*
   a. *Parsing Documents:*
      - *This involves reading each document and extracting its content using regular expressions. Metadata such as document numbers, file IDs, headlines, bylines, datelines, and main text are identified and extracted from each document.*

   b. *Preprocessing Text:*
      - *Before indexing, the text undergoes preprocessing. This step involves tokenization, wherein the text is split into individual words or tokens. Common preprocessing tasks like removing punctuation and stop words (common words with little semantic value) are performed. Stemming is applied using a Porter stemmer to reduce words to their root form.*
      - *Mappings are created to link each token, doc name to a unique identifier, aiding efficient retrieval and storage during indexing and querying.*

   c. *Batch Processing:*
      - *To manage memory efficiently and optimize processing, indexing and catalog creation is performed in batches of 1000 documents. This approach facilitates the handling of large document collections.*

   d. *Indexing:*
      - *The indexing process creates an inverted index, associating each term (token) in the document collection with the documents it appears in and their positions within those documents. For each token, a dictionary is constructed where the key is the token, and the value is another dictionary. In this nested dictionary, the keys are document*

numbers, and the values are lists of positions within the document where the token occurs.

   e. *Catalog Generation:*
      - *Alongside the inverted index, a catalog is generated. The catalog contains metadata about each term in the index, including its offset within the index file and the length of its posting list. This metadata facilitates efficient retrieval of terms during search operations in the later use.*

   f. *Writing to Files:*
      - *Finally, the generated inverted index and catalog are written to files. Each batch of indexed terms is written to a separate inverted index file. Corresponding catalog files containing metadata are also produced. These files form the basis for efficient retrieval and querying of the indexed document collection.*

2. *Pseudo algorithm for how merging was done*
   a. *Without Compression*
      - *Working of a single merge*

```
1. Input: Paths for merging and numbering, catalogs, and index files for both sets
   of data.
2. Open new catalog and index files for writing.
3. Initialize pointers i and j to 0 and offset to 0.
4. Loop while i is less than the length of the first catalog and j is less than the
   length of the second catalog:
    a. Read terms and corresponding start and length from both catalogs.
    b. If the terms from both catalogs match:
        i. Read corresponding document information from both index files.
        ii. Concatenate the document information.
        iii. Write the merged term, offset, and length to the merged catalog
             file.
        iv. Write the merged document information to the merged index file.
        v. Increment i, j, and update the offset.
    c. Else, if the term from the first catalog is greater than the term from the
       second catalog:
        i. Read document information from the second index file.
        ii. Write the term, offset, and length to the merged catalog file.
        iii. Write the document information to the merged index file.
        iv. Increment j and update the offset.
    d. Else (the term from the first catalog is less than the term from the second
       catalog):
        i. Read document information from the first index file.
        ii. Write the term, offset, and length to the merged catalog file.
        iii. Write the document information to the merged index file.
        iv. Increment i and update the offset.
5. After the loop, if there are remaining terms in the first catalog:
    a. Read document information from the first index file.
    b. Write the remaining terms, offsets, and lengths to the merged catalog
       file.
```

        c. *Write the remaining document information to the merged index file.*
    6. *After the loop, if there are remaining terms in the second catalog:*
        a. *Read document information from the second index file.*
        b. *Write the remaining terms, offsets, and lengths to the merged catalog file.*
        c. *Write the remaining document information to the merged index file.*
    7. *Close all opened files.*

- *Working of a single merge with compression*
    1. *Input: Paths for merging, catalogs, and index files, merge_number for both sets of data.*
    2. *Open new catalog and index files for writing.*
    3. *Initialize pointers i and j to 0 and offset to 0.*
    4. *Loop while i is less than the length of the first catalog and j is less than the length of the second catalog:*
        a. *Read terms and corresponding start and length from both catalogs. The mode of reading is changed accordingly throughout.*
        b. *If the terms from both catalogs match:*
            i. *Read corresponding document information from both index files.*
            ii. *Concatenate the document information.*
            iii. *Compress the merged document information using zlib.*
            iv. *Write the merged term, offset, and length to the merged catalog file.*
            v. *Write the compressed merged document information to the merged index file.*
            vi. *Increment i, j, and update the offset.*
        c. *Else, if the term from the first catalog is greater than the term from the second catalog:*
            i. *Read document information from the second index file.*
            ii. *Compress the document information using zlib.*
            iii. *Write the term, offset, and length to the merged catalog file.*
            iv. *Write the compressed document information to the merged index file.*
            v. *Increment j and update the offset.*
        d. *Else (the term from the first catalog is less than the term from the second catalog):*
            i. *Read document information from the first index file.*
            ii. *Compress the document information using zlib.*
            iii. *Write the term, offset, and length to the merged catalog file.*
            iv. *Write the compressed document information to the merged index file.*
            v. *Increment i and update the offset.*
        e. *After the loop, if there are remaining terms in the first catalog:*
            i. *Read document information from the first index file.*
            ii. *Compress the document information using zlib.*
            iii. *Write the remaining terms, offsets, and lengths to the merged*

                catalog file.
            iv. *Write the compressed remaining document information to the merged index file.*
        f. *After the loop, if there are remaining terms in the second catalog:*
            i. *Read document information from the second index file.*
            ii. *Compress the document information using zlib.*
            iii. *Write the remaining terms, offsets, and lengths to the merged catalog file.*
            iv. *Write the compressed remaining document information to the merged index file.*
    5. *Close all opened files.*

b. *Merging Orchestrator*
    - *Input: Path configurations for catalogs, indices, merging, and compressed directories.*
    - *Clean up the merging directory, if files are there.*
    - *List catalog and index files, sort them.*
    - *Set initial catalog and index files as the first file in those directory.*
    - *Initialize variables for merge count.*
    - *Loop through the catalog and index files for merging other than the first file:*
        1. *Load partial catalog and index files.*
        2. *Merge indices and catalogs using the merging algorithm.*
        3. *Update the current merged catalog and merged index files as initial for the next iteration.*
        4. *Increment merge count.*
    - *Copy the final merged files to the compressed directory.*
    - *End merging process.*

3. *Explain how merging was done without processing everything into the memory (Important)*
    a. *The merging process works without processing everything into memory which involves a sequential approach where data is processed in chunks and written directly to files without storing the entire dataset in memory simultaneously.*
        - *The function iterates through the catalogs and index files in parallel. It reads data in chunks, processes them, and writes the merged data directly to new files.*
        - *It maintains two pointers, one for each catalog being merged. It compares terms from the catalogs and writes the merged catalog and index entries accordingly.*
        - ***Rather than loading entire catalogs** and index files into memory, the function reads data in chunks. It reads catalog entries and corresponding index entries one by one using the offset provided in the catalog file, processes them, and writes the merged data to new files.*
        - *Merged catalog entries are written to a new catalog file, and merged index entries are written to a new index file. This ensures that the merged data is written incrementally without the need to store the entire dataset in memory.*
        - *Once all chunks have been processed and merged, the function completes the merging process by closing the output files and copying them to the destination directory.*
4. *How did you do Index Compression (For CS6200)*

   a. *During the merging process, zlib compression is applied to the document information before it is written into the merged index file. I specifically opted for level 6 compression, which offers a good balance between compression ratio and speed.*
   b. *This compression occurs at the point where the document information is combined from multiple index files into a single merged index file. Specifically, after retrieving the document information from each index file, zlib compression is performed on this data before it is written into the merged index file.*
   c. *This ensures that the document information is compressed before being stored, rather than compressing the file as a whole as encoding will vary when we access part of information later.*
   d. *Additionally, during retrieval or querying, the compressed data can be decompressed on-the-fly to obtain the original document information.*

5. *Brief explanation on the Results obtained*
   a. *I was able to reduce the storage size to the required ones for all versions. The order for me is as follows Stemmed Compressed < Unstemmed Compressed < Stemmed Uncompressed < Unstemmed Compressed which is the expected behavior.*
   b. *All the results of precision that I got are above the 90% expected threshold except for the TFIDF where the difference is very less 89% with an absolute precision difference of 0.004.*
6. *How did you obtain terms from inverted index*
   a. *The term's unique token is first found from the map we have. Then if that unique token exists in the catalog, it retrieves the corresponding entry from the full merged catalog, which contains the start position and length of the postings list for that term in the index file.*
   b. *Using the start position and length obtained from the catalog, I seek to that position in the index file and use the length to get the corresponding segment. This segment contains the compressed or uncompressed postings list for the term.*

PROXIMITY SEARCH (*For CS6200*)

Index	**Score**
Unstemmed	0.1698
Stemmed	0.2673

<u>Inference on the proximity search results</u> *(Make sure to address below points in your inference)*

1. *Which matching technique you have implemented*
   a. *For each query term, the code calculates an accumulator value(idf) that represents the proximity of the term to other terms within the document. This is achieved by iterating through the positions of each term and computing a score based on their proximity within a specified threshold of 2.*
   b. *Using the accumulator value and other document-specific factors (such as term frequency, document length, and average document length), the code computes a proximity score for each document which is a BM25 . This score reflects the relevance of the document to the*

query based on the proximity of query terms within the document.

c. *The proximity scores obtained for each document are aggregated with scores from other matching techniques, such as BM25, to produce a comprehensive ranking of documents based on their relevance to the query.*

2. *Pseudo algorithm of your Implementation*

```
a. Function calculate_accumulator(total_no_of_docs, proximity_threshold, word,
   word_positions_dict):
     i.   Initialize accumulator_value to 0
     ii.  Retrieve positions of word from word_positions_dict
     iii. For each term, positions in word_positions_dict:
            1. If term is equal to word, continue
            2. Iterate through positions of word and other terms
            3. Calculate window between positions
            4. If window is within proximity_threshold:
                 a. Calculate temp_accumulator which is idf
            5. Add temp_accumulator to accumulator_value
     iv.  Return accumulator_value

b. Function calculate_proximity_score(no_of_docs, term_frequency, accumulator_term,
   document_length, average_document_length, k1, b):
     i.   Calculate inverse_document_frequency
     ii.  Calculate calculation_2 which is the other part in bm25 calculation
     iii. Calculate proximity_score
     iv.  Return proximity_score

c. Function proximity_search(query_dictionary, stem=True, compression=True):
     i.   Open index file for reading
     ii.  Initialize scores using bm25 as the base
     iii. Iterate through queries for each query token
            1. Read document details from index file
            2. Parse document details and store in query_info that has details about the
               tokens in the query along with their positions.
     iv.  For each document and terms_positions in query_info:
            1. For each term and positions in terms_positions:
                 a. Calculate term_frequency
                 b. Calculate current_accumulator using calculate_accumulator
                 c. Calculate proximity_score using calculate_proximity_score
                 d. Update scores for that document for this query
     v.   Close the file
     vi.  Return scores
```

Extra Credits

- *If any EC done, please explain using subheadings*
- *Include any precision increment/ results you have got using EC implementation*
- *If query Optimisaton done, then include time in seconds taken for one query*
- *If you have added HEAD field, make a table depicting precision with and without Head Info*