HomeWork-2 Report Template

INDEX SIZE

Compressed | Stemmed	16.9 MB
Decompressed | Stemmed	176.1 MB
Decompressed | Unstemmed	181.2MB

Brief Explanation on the process used for Indexing

The indexing process begins with parsing documents to extract text, remove stopwords, and optionally apply stemming to reduce words to their base forms. Each document's text is then tokenized into terms, with each term being assigned a unique identifier if it's not already indexed. The index records the frequency of each term in individual documents, along with the positions of terms within documents, facilitating proximity searches and relevance scoring. For efficiency and reduced storage, the index may be compressed using techniques like variable-byte encoding. The process culminates in saving the index and relevant metadata, such as document lengths and term document frequencies, to disk, enabling fast retrieval and scoring of documents in response to queries.

MODEL PERFORMANCE

Index	**Model**	**Old Score**	**New Score**	**Percent (New/Old)**
Decompressed | Stemmed	TF-IDF	0.2792	0.3210	1.1497
Decompressed | Stemmed	Okapi BM-25	0.2762	0.3170	1.1477
Decompressed | Stemmed	Unigram LM with Laplace smoothing	0.2516	0.2658	1.0565
Decompressed | Unstemmed	TF-IDF	0.2792	0.3054	1.0940
Decompressed | Unstemmed	Okapi BM-25	0.2762	0.2793	1.0112
Decompressed | Unstemmed	Unigram LM with Laplace smoothing	0.2516	0.2547	1.0123
Compressed | Stemmed	Okapi BM-25	0.2516	0.2671	1.0671

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

- *Explain how index was created*
- *Pseudo algorithm for how merging was done*
- *Explain how merging was done without processing everything into the emory (Important)*
- *How did you do Index Compression (For CS6200)*
- *Brief explanation on the Results obtained*
- *How did you obtain terms from inverted index*

typically access the df_map attribute of Indexer to get its document frequency (indicating in how many documents the term appears). To retrieve the actual postings list for a term (i.e., the list of documents in which the term appears along with their respective term frequencies), i would use the term's identifier (term_id) to look up the corresponding list in the postings_map.

create the index, the process begins by parsing and processing documents from a collection. Each document is tokenized into terms, optionally applying stemming and removing stopwords. These terms are then mapped to document identifiers (docIDs) along with their positions within the documents, forming an inverted index structure. This structure is initially built in memory and periodically written to disk as partial indexes to manage memory consumption. Merging without loading everything into memory was achieved through a sequential or stream-based approach, where partial indexes were read and merged incrementally. This method involved reading small segments from each partial index simultaneously, comparing term identifiers, and merging their postings lists on-the-fly, writing the merged postings to the final index as the process advanced. This technique minimized memory usage by only keeping the necessary parts of each partial index in memory at any given time, efficiently handling large datasets that exceed memory capacity.

Variable-byte encoding reduces the space needed for storing integers by using fewer bytes for smaller numbers, which is effective for document IDs and term frequencies that often vary in magnitude. Dictionary compression for terms involved creating a shared dictionary where each unique term is mapped to an identifier, reducing redundancy. Additionally, for posting lists, techniques like delta encoding were applied, where only the difference between successive document IDs is stored, exploiting the sorted nature of these lists to further reduce the index size. This combination of methods significantly reduced the storage requirements of the index while maintaining fast access for retrieval operations.

PROXIMITY SEARCH (*For CS6200*)

Index	**Score**
Unstemmed	0.3272
Stemmed	0.3264

Initialize an empty final index.
For each partial index on disk:
Load the partial index.
For each term in the partial index:
If the term exists in the final index, merge the new posting list with the existing one.
If the term does not exist in the final index, add the term and its posting list to the final index.
Apply compression to posting lists if required.
Write the final index to disk.

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

- *Which matching technique you have implemented*
- *Pseudo algorithm of your Implemntation*

skipgrams

This allows for more flexible matching patterns that can capture the proximity of terms without requiring exact adjacency. Skip-grams can be particularly useful for search queries where the order of terms might vary or where the presence of intervening words is acceptable.

Retrieve the postings lists (including positions) for each term in the proximity query.
Intersect these postings lists to find documents where all terms occur.
Within each document, examine the positions of each term to identify instances where the terms occur within the specified proximity constraint.
Score or rank these documents based on the proximity of the terms, often with documents where terms appear closer together receiving higher scores.