HomeWork-2 Report Pramatha Bhat

INDEX SIZE

Compressed | Stemmed	71.4MB
Decompressed | Stemmed	181MB
Decompressed | Unstemmed	189MB

Brief Explanation on the process used for Indexing

First, all AP-89 files, with a total of 84,678 are read and parsed through. Preprocessing techniques, including removing stopwords, stemming, removing punctuations, and case folding, are applied. Tokenization is then conducted based on the specified criteria given in the question.

Following preprocessing, document ID mapping is executed and stored in a file. Then, an inverted index is created for each group of 1000 documents, where the term is the key and postings is the value.

Once this process is completed for every 1000 documents, sorting is applied, and catalog and index files are generated. The contents are then written into these files. This procedure is repeated for all 84,678 documents, resulting in the creation of 85 catalog files and 85 index files.

Finally, the individual index files are merged into a single index file using merge sort technique. Simultaneously, a catalog file is generated accordingly. Catalog file consists of word, offset and size and index file consists of mapped doc_id and the array of positions where the term occurs in that doc.

MODEL PERFORMANCE

Index	**Model**	**Old Score**	**New Score**	**Percent (New/Old)**
Decompressed | Stemmed	TF-IDF	0.2955	0.3179	107.5
Decompressed | Stemmed	Okapi BM-25	0.3033	0.3233	106.5
Decompressed | Stemmed	Unigram LM with Laplace smoothing	0.3211	0.3270	101.8
Decompressed | Unstemmed	TF-IDF		0.2413	
Decompressed | Unstemmed	Okapi BM-25		0.2469	
Decompressed | Unstemmed	Unigram LM with Laplace smoothing		0.2621	
Compressed | Stemmed	Okapi BM-25		0.3233	

<u>Inference on above results</u> *(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*

Index creation:

Initially, all AP-89 files, totaling 84,678 in number, are examined and processed. This involves applying various preprocessing steps such as eliminating stopwords, stemming, removing punctuation, and converting text to lowercase. Tokenization follows, adhering to the predefined criteria specified in the question.

After preprocessing, the mapping of document IDs is carried out and stored in a dedicated file. Subsequently, an inverted index is established for each batch of 1000 documents, where terms serve as keys and their corresponding postings as values.

Upon completing this process for every set of 1000 documents, sorting ensues, leading to the creation of catalog and index files. These files are populated with the sorted data and the procedure is repeated for all 84,678 documents, resulting in the generation of 85 catalog files and 85 index files.

Lastly, the individual index files undergo merging using the merge sort technique to form a unified index file. Simultaneously, a catalog file is generated in alignment with this merged index file. The catalog file contains information such as word, offset, and size, while the index file comprises mapped document IDs and arrays of positions where the terms occur within each document.

Pseudo algorithm for how merging was done:

Merge Files Function:

```
1. Define a function to read content from a file based on offset and size.

2. Initialize an offset variable.

3. Open all necessary files for reading and writing: catalog files 1 and 2, index files 1 and 2, merged index file, and
merged terms file.

4. Read the first lines from both catalog files.

5. Begin the merge process.

  6. While there are lines in both catalog files:

    7. Split the lines to extract term, offset, and size information.

    8. Compare terms from both files and merge content accordingly:

      - If term1 is less than term2:

        - Read content from index file 1 and write it to the merged index file.
        - Write term1 and related offset information to the merged terms file.
        - Update the offset.
        - Move to the next line in catalog file 1.
      - If term1 is greater than term2:
        - Read content from index file 2 and write it to the merged index file.
        - Write term2 and related offset information to the merged terms file.
        - Update the offset.
        - Move to the next line in catalog file 2.
      - If term1 is equal to term2:
        - Read content from both index files, merge them, and write to the merged index file.
        - Write term1 and related offset information to the merged terms file.
        - Update the offset.
        - Move to the next line in both catalog files.
  9. Handle remaining lines from either catalog file:
    - If there are remaining lines in catalog file 1, repeat the process for index file 1.
    - If there are remaining lines in catalog file 2, repeat the process for index file 2.
10. Close all opened files.
```

In every iteration, merged file from previous iteration is passed as index_file1 and catalog_file1 for next iteration. The merging process involves sequentially combining one new index file with the existing merged index file, and similarly merging the corresponding catalog files. This operation is repeated for each index file in the provided directory. Since each of the catalog files are alphabetically sorted, merge sort can be efficiently utilized to merge them. After merging, the resulting temporary merged files are written into the final merged files. This iterative process ensures that the merged files are continuously updated with the latest merged data. Once all index files have been processed, the final merged index and catalog files contain the combined data from all input files. This approach effectively employs merge sort methodology to merge multiple sets of indexed data into a unified set.

Explain how merging was done without processing everything into the memory

Merging was accomplished without loading all data into memory at once by utilizing a temporary merge file. Here's how the process works:

First, one catalog file is read, and its contents are compared with the temporary merged catalog file. These two files are merged together, taking into account the order of the words.

Then the postings for each word are written directly into the temporary merged index file. The offset and size of each word's postings in the merged index file are then written into the catalog file.

Once the merging of two files is complete, the contents of the temporary merged file are moved to the final merged file. This process is repeated for each pair of input files until all files are merged.

How did you do Index Compression (For CS6200)

I've used gzip for compressing the index file. During decompression, it opens the gzip compressed index file in binary read mode, which helps to efficiently seek the specified offset within the file. It then reads the specified number of bytes(size), decompresses them, and finally decodes the resulting bytes from UTF-8 encoding. This approach is efficient because it doesn't require decompressing the entire file. The file size of the compressed file came down to 71MB from 180MB.

I've improved my approach by dividing the big index file into three smaller ones. Now, instead of searching through the entire large file, I can concentrate on just one of the smaller files whenever I need to find something. This approach works well because the data in these smaller files is already sorted, making it faster and easier to locate what I need. I've stored information in a JSON file which helps me quickly determine the range of words present in each smaller file. The file size of the compressed files is the same as one big compressed file.

Brief explanation on the Results obtained
The custom index that I created significantly improved the performance of the models utilizing stemming. TF-IDF and Okapi BM-25 models experienced notable enhancements in performance, with improvements ranging from 6.5% to 7.5%. The Unigram LM with Laplace Smoothing model also saw a slight improvement in performance. The custom index has performed better than elastic search. Stemmed queries gave better results compared to unstemmed queries throughout. Compressed Stemmed Index gave same results as Uncompressed Stemmed Index.

How did you obtain terms from inverted index
From the inverted index, use the offset and size from the catalog file to get what content to read from index file. Once we get the details, file.seek() is used to go to the starting point from where we need to read and then we read the required size of data and place it in memory.
In case of compressed files, we only read from the file where the token is present. Entire file is not decompressed, only the necessary data is read in binary and converted to text using utf-8 encoding.

PROXIMITY SEARCH (*For CS6200*)

Index	**Score(unmodified queries)**	**Score(modified queries)**
Unstemmed	0.2337	0.2441
Stemmed	0.2915	0.3239

Inference on the proximity search results *(Make sure to address below points in your inference)*
We can see slight increase in the score for Stemmed modified queries after using proximity search on top of Okapi BM-25 model(0.3239), compared to just using Okapi BM-25 model(0.3233). Scores for modified queries are higher compared to that of unmodified queries.

- *Which matching technique you have implemented*
  The matching technique implemented for proximity search is based on the minimum span algorithm. This algorithm efficiently identifies the shortest span of text in a document that contains all of the words in a given query phrase. By utilizing a sliding window technique across the positions of the words in the document, the algorithm determines the smallest window that encompasses all search terms. The proximity search score is then calculated using the formula $\lambda^{(s-k)/k}$, where λ is a constant base (typically around 0.8), s is the minimum span found, and k is the length of the ngram matched. This scoring function ensures that documents containing the searched words close together receive higher relevance scores.

- *Pseudo algorithm of your Implementation*

1. **Calculate Proximity Score Function**:
    - Takes minimum span and ngram length as input.
    - Returns 0 if ngram length is 0.
    - Otherwise, returns 0.8 raised to the power of ((min_span - ngram_length) / ngram_length).

2. **Get Minimum Span Function**:
    - Takes a dictionary of query proximity as input.
    - If the dictionary is empty, returns -1.
    - Otherwise:
        - Creates a list of tuples containing position information for each word in the query.
        - Sorts the list of tuples in ascending order.
        - Initializes a window with the first position from each word's postings.
        - Iterates through the sorted tuples:
            - Updates the window with the current tuple's position.
            - Adjusts the window to maintain the minimum span.
            - Updates the minimum span if the current window span is smaller.
        - Returns the minimum span found.

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*

References

https://www.khoury.northeastern.edu/home/vip/teach/IRcourse/2_indexing_ngrams/lecture_notes/SteveKrenzel-FindingBlurbs.pdf
https://www.khoury.northeastern.edu/home/vip/teach/IRcourse/2_indexing_ngrams/lecture_notes/indexing/indexing.pdf