CS 6200 HW2

Parth Shah

February 24, 2023

Task 1 - Tokenizer:

The code reads a collection of text documents from a directory, preprocesses them by tokenizing and removing stopwords, and then creates a dictionary of document IDs and their corresponding tokenized text. Additionally, it creates a dictionary mapping each document's ID to its document number and the number of tokens in the document. The document IDs and their corresponding document numbers and token counts are written to a file, and the dictionary of tokenized documents is serialized to a pickle file.

Task 2 - Indexer:

The code creates an inverted index of tokens from a given set of documents. The indexer function indexes the given documents' tokens and creates a dictionary of them. Then, sort_tokens sorts the token dictionary in ascending order and returns it as an OrderedDict. write_index takes this sorted token dictionary and writes two files: a catalog file and an inverted index file. The catalog file contains information about the token, including its offset and length, while the inverted index file contains information about the tokens, their positions in the documents, and the documents they are found in. The code splits the documents into chunks of 1000 and creates separate index files for each chunk. The code uses stemming on the tokens if STEMMING is set to true.

Task 3 - Merger:

The code merges multiple catalogs and their respective indices into a single catalog and index file. The program takes in a path to the directory containing the catalogs, indices, and the directory where the merged files will be stored. The function load_catalog() loads a catalog file into an ordered dictionary. The merge_indices() function takes in the two catalog files to be merged, their respective index files, and creates two new merged files, one for the merged catalog and another for the merged index. The list_files() function returns a list of all the files in the given path. The main program first loads the full catalog file and its corresponding index file, and then iterates through all the other catalog and index files, merging them in sequence, and updating the full catalog and index files until all the catalogs have been merged.

Merging Algorithm:

The merge algorithm takes two sorted lists of tuples, representing catalog entries for two sets of documents, and merges them into a single sorted list. The resulting merged list is then used to create a merged catalog file and index file. The algorithm iterates over the two lists of catalog entries and compares the terms in each entry. If the terms match, the algorithm combines the corresponding index information for each entry into a new combined entry and writes it to the merged index file, and writes the combined catalog entry to the merged catalog file. If the terms do not match, the algorithm determines which entry has the smaller term and writes the corresponding catalog entry and index information to the merged files.

This process is repeated until all entries in both lists have been processed. If there are any remaining entries in either list after the initial comparison and merging process, the algorithm iterates over the remaining entries in each list and writes them to the merged catalog and index files. The resulting merged catalog file consists of lines of the form "term:offset, length" where the term is a term in the document collection, the offset is the byte offset of the corresponding index information in the merged index file, and the length is the length of the corresponding index information. The merged index file consists of concatenated strings of index information for each document in the collection.

Task 4 - Compressor:

The code above compresses a given file using the zlib library's compress function. The input file's document information is first decompressed if it's in binary form. Then, the document information is combined into a new string, which is compressed using the zlib library's compress function with a default compression level of 6. The compressed data is then written to the output file.

Task 5 - Models:

The codes for Okapi TF, Okapi BM25, and Unigram Laplace retrieval models have been reused from homework 1. The new implementation has been made for Okapi TF for the compressed version of the indices. The only change in the approach is that the document line is decompressed as and when necessary and then parsed for the required details. Another new model added is the Proximity Search model.

Proximity Search:

The general algorithm in the code performs a proximity search on a given set of queries. It uses the Okapi BM25 scoring algorithm to rank the documents based on the query terms' frequency in the documents. The proximity search is done by calculating a proximity score for each document and query pair using the accumulator and proximity functions. The accumulator function computes the accumulated score of a given term with all other terms in the query. The proximity function calculates the proximity score of the term based on its frequency and the accumulated score. The proximity search is performed for each query, and the scores are returned as a dictionary. The code uses the catalogs to find the inverted index file's location for each term in the query and reads the index file to extract the relevant document information. The code obtains the document's length information and uses the average document length to calculate the proximity score. The code uses the Okapi BM25 algorithm's parameters to calculate the scores.

Homework 2 Results

Inverted Index Type	File Size (in MB)
Raw Non-Stemmed Inverted Indices	153.1
Raw Stemmed Inverted Indices	148.6
Compressed Stemmed Inverted Indices	71.7
Stemmed	
Model	**Uninterpolated MAP**
Okapi TF	0.21
Okapi BM25	0.28
Laplace	0.20
Proximity Search	0.23
Non-stemmed	
Model	**Uninterpolated MAP**
Okapi TF	0.15
Okapi BM25	0.17
Laplace	0.12
Proximity Search	0.13
Compressed	
Model	**Uninterpolated MAP**
Okapi TF	0.21

Homework 1 Results

Baseline Retrieval Models			
Model	**Uninterpolated Mean Average Precision**	**Precision at 10 documents**	**Precision at 30 documents**
ES Inbuilt	0.2978	0.4480	0.3653
Okapi TF	0.2460	0.4360	0.3307
TF-IDF	0.2879	0.4560	0.3613
Okapi BM25	0.2948	0.4440	0.3640
Laplace	0.2271	0.4280	0.3280
Jelinek-Mercer	0.2505	0.3520	0.3200

Custom Pseudo-relevance Feedback			
Model	**Uninterpolated Mean Average Precision**	**Precision at 10 documents**	**Precision at 30 documents**
ES Inbuilt	0.2843	0.4360	0.3427
Okapi TF	0.2548	0.4080	0.3440
TF-IDF	0.2780	0.4200	0.3413
Okapi BM25	0.2852	0.4240	0.3413
Laplace	0.2498	0.4360	0.3387
Jelinek-Mercer	0.2495	0.3600	0.3227

ES "significant terms" Pseudo-relevance Feedback			
Model	**Uninterpolated Mean Average Precision**	**Precision at 10 documents**	**Precision at 30 documents**
ES Inbuilt	0.2209	0.3560	0.2760
Okapi TF	0.1831	0.3320	0.2560
TF-IDF	0.2215	0.3160	0.2827
Okapi BM25	0.2243	0.3600	0.2787
Laplace	0.1682	0.3320	0.2480
Jelinek-Mercer	0.1972	0.2840	0.2333