September 1, 2023

1 File Structure

1.1 Construction of the Indices

- invidx.sh (arguments)

Based on Compression Flag (0/1) and BPE Tokeniser flag (0/1) calls one of 4 functions housed in

- invidx_cons.py

and creates a Dictionary and a Postings list

1.2 Searching the Queries

- tf_idf_search.sh <queryfile> <resultfile> <indexfile> <dictionary>

Calls

- top.py

This file reads some metadata from the dictionary and postings file, which tells it about the compression mode and tokenization technique. Based in this, calls one of these 4 files

- q_00.py
- q_01.py
- q_10.py
- q_11.py

where in q_ij , i is the compression on/off, and j is BPE on/off. These 4 files parse the query document, extract keywords for searching, and call the functions for evaluating these files.

Next, there are supplementary files to help read the dictionary and postings files in these respective modes. They read information like pointers -> (docNum) and tf data, and token -> pointer data, docNum -> docID data and other metadata.

- readers_00.py
- readers_01.py
- readers_10.py
- readers_11.py

Further, Query processing is done in one of these 4 files. These files read find out the appropriate document(s) per query term, accumulate score for these documents term-at-a-time (TAAT). Then, the top100 of these are returned and printed into the the result file.

- process_query_00.py
- process_query_01.py
- process_query_10.py
- process_query_11.py

2 Search Result Evaluation Parameters

- Index Construction Time - 982.55 seconds = 16:22 Minutes
- Average Query Retrieval Time - 1.276 seconds
- Average F1@100 score - 0.229
- Average Recall - 0.281
- Average Precision - 0.611

3 Operations

No Compression, Standard Tokenisation

3.1 Construction

a) Traverse through all the files, using lxml parser to parse and split content on whitespaces and lowercase it. Then, for each document, maintain the set of tokens that occur in it. Keep a global dictionary tokens that maps token -> number of documents it occurs in. This is later used in pointer allocation and idf score calculation. Also count the number of documents in the collection.

b) Based on token count and the max lengths of document frequency and document number, set the lengths representing the number and the df. Now, traversing the documents, count the tokens and their respective tf's, write them into the postings file at the proper pointer location, and evaluate the normalising score for each document.

c) Having written the above, append the docNum -> docID information and the docNum -> docScore information at the end of it. In the dictionary file, write out the token -> pointer mappings.

3.2 Searching

a) From the metadata in Dictionary and Posting file, figure out the compression and encoding modes.

b) From the Dictionary, read off and store the token -> pointer information, and from the Postings file, read the docNum -> (docID , docScore) information

c) Parse the query file. For each query, split the title and description into tokens and for each token, find the documents and their tfs associated with it, evaluate and add to it's score the idf*tf score.

d) After all the terms have been evaluated and scores accumulated (using TAAT), I normalise all documents scores using their respective normalising scores precomputed at construction time and saved in the dictionary file.

e) Enlist the top 100 results based on their score in the output file.

4 Variable Byte Encoding

I realised that in the earlier scheme, the majority of the tokens are relatively less frequent, and we waste a good amount of storage in padding 0's to tf scores for each token and document. Further, we waste some space in padding document numbers with 0's at the front to conform to the byte allocation standard.

In order to not let go of this space, I did not pad the term frequencies and document numbers. Instead, I put in two special characters (the space and the "?") that alternate the docNum and tf sequences. This average addition of 2 characters per entry instead of padding reduces storage requirements of the postings list from 1.22Gb to 1.07Gb. However, I now needed to store the number of bytes per token in order to know the point upto which to read. This was stored in the dictionary file along with the pointers.

5 Byte Pair Encoding

5.1 Learning

Going through a random 25% of the files, I accumulate the vocabulary of the white-space separated tokens in these files. This vocabulary maps a tuple of tokens to the frequency. For example, at the initialisation, (l, o, w, e, s, t, _) -> 5 .

Now, for each pair of neighbouring tokens across the vocabulary tokens, I count the frequency of occurrence. Then, the highest frequency is the one I need to merge. This merger is stored for later as well. Then, all vocabulary tokens that have this pair, are modified. For example, if e and s have to be merged, the above vocabulary key becomes (l, o, w, es, t, _) -> 5

I repeat this 5000 number of times, which I believe for our data set is a good tradeoff between overfitting and underfitting. These merges are stored for later.

5.2 Tokenisation

For each document, the merges learnt earlier are performed greedily and in-order of learning on the vocabulary of tokens in this document. The tokens obtained after performing all mergers are the final set that need to be indexed and stored for this document.

Note that the query file also needs to be tokenized similarly before it's tokens are used to search the constructed index file.