CSCE 4013 - Information Retrieval
Howework-4 Report
Kanat Bektemirov

Objective

The objective of this project is to build a command-line retrieval engine on top of the inverted files created in previous assignments.

Requirements

Upon typing:

```
$ retrieve.py [-h] -q QUERY -p PATH [-n NUMBER]
```

the program retrieves the top N matching documents for the given query. Arguments:

```
-q or --query: query took look up
-p or --path: path to the directory where inverted files are located
-n or --number: maximum number of results to retrieve (default is 10)
```

Flow

The structure of the retriever is as following using hashtable + heap/array for the accumulator:

1. `retrieve.py` accepts a query, number of results, and the path to inverted files folder.
2. Tokenize the query using the tokenizer from homework-1
3. For each unique token in the query, read the dict entry, then post entries, and update the cumulative weights
4. Print the top N matching results in sorted order.

Changes

- One line (line 46) change in the tokenizer to better handle URLs.
- Decreased the low frequency words threshold from 5 documents to 2 (defined in `configs.cfg` as `min_frequency`). This gave 25,378 unique tokens and 679,993 total tokens. The dict file has 25,378 * 3 lines and is 4,035,102 bytes.
- No other code changes were made to the code from the previous assignment other than adding a new script `retrieve.py`.

Retriever

The retriever performs 5 steps at the high level: (1) tokenizes the query into tokens, (2) reads the dict entry, (3) reads the post entries, (4) updates the weight accumulator, (5) and returns the top N entries with the highest weights.

Tokenization

I used the same tokenizer as I wrote in assignment-1 for processing the query. The tokenizer simply returns a local hashtable.

Dict Entry

The *dict* file is opened once per program execution. It is necessary to find the dict entry for each unique token in the query. The dict entry is located at some line, *L*, in the dict file. This line is the hash value of the token. This is the same hash function that was used when creating the inverted files. The hash function needs to know the total number of lines in the dict file which cannot be obtained in a constant time. However, since the records in the dict file are fixed-length, the total number of lines can be obtained in O(1) as:

```
lines = dict_file_size_in_bytes / dict_rec_size
```

The file size can be obtained in constant time, and `dict_rec_size` is 40+1+4+1+6+1=53. The dict entry is expected to be on line `L = hash(key=token, size=lines)`. However, the token might not exist in the dict file, or there might have been a collision. Therefore, the program starts reading the dict file from line `L` until it sees a blank entry or the entry it is looking for. The specific line can be accessed by first seeking to the correct starting byte `file.seek(dict_rec_size * L)`, then by reading the record starting from there `file.read(dict_rec_size)`. Each dict entry is a tuple with `(string, int, int)` structure.

Post Entries

The post file is opened once per program execution. Once `num_docs` and `start_pos` is obtained from previous step, random bytes in the post file can be accessed in a similar fashion as described earlier. Specifically:

```
for 0 <= i < num_docs:
    file.seek(post_size * (start_pos + i))
    entry = file.read(post_size)
```

Each post entry is a tuple with `(int, int)` structure. Once post entries are loaded for the given token, the accumulator is updated for doc_ids with the weights in the post entries. Another important thing to keep in mind is the repeated tokens in the query where the user wants to weigh query terms differently. With a query such as “dog cat dog cat dog”, the program finds the dict and post entries for “dog” and “cat” only once and multiples those weights of the documents by 3 and 2, respectively in the post entries.

Accumulator

Ideally, three criteria need to be met for a good accumulator representation for weights: (1) fast look-up by doc_id, (2) small space in memory, (3) fast sort by total weight.

- I use a hashtable (a specific version of hashtable called Counter in Python) for storing the total weights. Look up and insert operations are `O(1)`. Criteria-1 check.
- The size of the hashtable is `r`, where `r`=the number of non-zero-weight documents. The size of the array or heap (described in the next bullet point) is also `r`. Total memory the accumulator uses is `O(r+r) = O(r)`. Criteria-2 check.
- The hashtable used (Counter) has a method `most_common(p)` which returns the p-largest entries sorted by values in the hashtable. The implementation of this method is clever.

- If `p` is small relative to the hashtable size, it creates a heap of size `p`, heapifies the entries in the hashtable, and does a max-heapsort. This operation is `O(plogp)`.
- If `p` is large relative to the hashtable size, it creates an array of size `r` with the hashtable entries, sorts the array by entry values (weights), and returns the first `p` items in the array. This operation is `O(rlogr)`.

Criteria-3 double check.

Output

I did not spend much time prettifying the output format. It prints out the document name, document ID, and the total weight for each result. Additionally, the program also outputs the dict entries for each unique token in the query for easier debugging and visualisation.

Testing

Please see the attached testing file.