CSCE 4013 - Information Retrieval
Howework-3 Report
Kanat Bektemirov

Objective

The objective of this project is to build an index for a collection of documents using the normalized term weights for the tokens that occur in each document. I implemented the memory-based indexing algorithm (in Python).

Requirements

Upon typing:

```
$ tok_index.py [-h] -i INPUT_DIR -o OUTPUT_DIR [-n NUMBER_FILES]
```

the program generates (in the output directory):

1. *dict*: a file containing term num_docs start_loc.
2. *post*: a file containing doc_id, wt that go with *dict*.
3. *map*: a file containing the doc_id => actual filename mapping. (doci_id == line number)

Flow

The structure of the program is as following using the memory-based indexing algorithm:

1. `tok_index.py` accepts input directory, output directory, and a number of files.
2. For each HTML file, I build a frequency counter for each token. (“local hash table”)
3. After processing each file, I update the global hash table.
4. Write a fixed-length record maps, dict, and post files.

Improvements

I extended my project-2 by removing stoplist words; removing low frequency words; removing high frequency words; removing words of length 1; writing normalized term weights instead of raw frequencies in the post file; creating fixed-length output files; and adding a config file to the project.

Stopwords

In the `configs.cfg` file, the file path to a stop words is specified. In my tokenizer (`tok.py`), my program ignores tokens that are in the stoplist before updating the local hashtable.
I have also add the following words to the list: am, anyway, anyways, comes, evermore

Words of Length 1

In the `configs.cfg` file, the `min_len` variable defines the minimum token length to be tokenized. I defined it as 1. In my tokenizer, my program ignores tokens that have the length of of before updating the local hashtable.

Low Frequency Words

In the `config.cfg` file, the `min_frequency` variable defines the threshold for low frequency documents. I defined it as 5. When writing to the dict file, the program ignores tokens that appeared in less than 5 documents.

High Frequency Words

In the `config.cfg` file, the `max_frequency` variable defines the ratio threshold for high frequency documents. I defined it as 0.8. When writing to the dict file, the program ignores tokens that appeared in more than 80% of the documents in the collection. For the document collection we are working with, the following words appeared in 80% or more of the documents: privacy, advertise, policy, site, terms

Normalized Term Weight

I performed a dirty length normalization. I normalized the term weight in the document by the number of unique words in the document. I also tried normalizing by total number of tokens in the document. The relative orders of the documents are very similar when sorted by term weights for a given token. Here's are the results for the term "**airport**". The highlighted portion is slightly out of order because of a very similar term weights between the three documents.

Normalized by total tokens	Normalized by unique tokens
[1208, 516.8011258464705] [863, 170.89110710897668] [471, 141.50507017224788] [965, 91.90535485414037] **[1465, 72.18477263847463]** **[1068, 70.19542851064266]** **[945, 61.83689771688983]** [1149, 55.486427930611725]	[1208, 775.2016887697058] [863, 297.1606473617206] [471, 232.56050663091173] [965, 133.72229131277425] **[1068, 118.60070182951154]** **[945, 114.78308267190923]** **[1465, 107.40746290182669]** [1149, 89.59617508393585]

I used the follow formula for normalization:

```
N = number of documents in the collection
idf = log2(N / num_docs)
wt = tf / num_unique_tokens * idf * 10,000
```

And here is the pseudo-code that implements that:

```
for each document, doc_id:
    build a local_ht with token->frequency counter
    for each token in local_ht:
        global_ht[token].nodes.add(doc_id=doc_id,
                                   weight=local_ht[token] / local_ht.size)
for each token in global_ht:
    write token, gnum_docs, start_pos to dict file
    idf = log2(N / global_ht[token].nodes.size)
    for each node in global_ht[token].nodes:
        node.weight *= idf * 10,000
        write doc_id, node.weight to post file
```

Fixed-Length Output Files

map file

map record: filename

Since the longest filename in the collection is 11 characters (medium.html and simple.html), every record in my map file is 11 characters (not including the newline). Spaces are used to fill up the filename if less than 11 characters. This is defined in `configs.cfg` as `map_rec`.

post file

post record: doc_id term_weight

Since there are ~1600 documents in the collection, doc_id is 4 characters long. Since the highest term weight is about 540,000, term_weight is 6 characters long. Spaces are used to fill up doc_id and term_weights are less than the mentioned character lengths. Each record is 4+1+6=11 characters (not including the newline). These are defined in `configs.cfg` as `post_rec[0]..[1]`.

dict file

dict record: token num_docs start_pos

Since there are ~1600 documents in the collection, num_docs is 4 characters. Since the post file has >500,000 lines start_pos is 6 characters long.

Since my program indexes URLs as well, I needed a somewhat longer token length. I specified token length as 40 characters. I take the first 20 and last 20 characters for each token when writing to the dict file. Each dict record is 40+1+4+1+6=52 characters (not including the newline). These are defined in `configs.cfg` as `dict_rec[0]...[2]`.

Statistics

1683 HTML files, the program yielded the following statistics:

Without filtering: `unique_tokens: 63322, total_tokens: 902947`

With filtering: `unique_tokens: 14307, total_tokens: 661632`

Records and file sizes:

```
$ wc < out/map
  1682    1683   20195
$ wc < out/dict
  188334   42921 9981702
$ wc < out/post
  661631 1323264 7939583
$ head -3 out/map
  e1.html....
  e10.html...
  e100.html..
$ head -3 out/post
  8    2501..
  126  1764..
  182  1230..
$ sed -n 102,105p out/dict
cutty                                       5    273...
.................................................
lopez                                       170  278...
```

Performance

My program tokenizes the files first, and then does the indexing for each file. Here, I report the both the total time and the indexing time alone.

On a machine with 8GB RAM, 2.6 GHz processor, and a solid-state drive, the program performed as following:

Files	Total Time (s)	Indexing Time (s)
1	0.21	0.008
10	0.90	0.02
50	4.57	0.06
100	9.2	0.17
500	41.6	1.06
1000	82.1	2.17
1683	158.2	4.82

Number of files vs. total runtime and indexing time.