COSC 488: Introduction to Information Retrieval Spring 2018
Project Part 2: Query Processing

Sabrina Ma slm243@georgetown.edu

Cover page

a. Status of this assignment: Complete

b. Time spent on this assignment: 35 hours

c. Things you wish you had been told prior to being given the assignment

I wish I had known that storing each term's df, idf, and cf and each document's length in the inverted index would be important while completing P1 so I wouldn't have to go back and edit my P1 code.

Design Document

Shell Interface Instructions

Building The Index

```
python3 build.py [trec-files-directory-path] [index-type] [output-dir]
Example: python3 build.py data/ positional output/
```

- `[trec-files-directory-path]` is the directory containing the raw documents
- `[index-type]` can be one of the following: "*single*", "*stem*", "*phrase*", "*positional*"
- `[output-dir]` is the directory where index and lexicon files will be written

Query Processing (Report 1, Static)

```
python3 query.py [index-dir-path][query-file-path][retrieval-model][index-type][results-file]
Example: python3 query.py ./indexes/ ./data/queryfile.txt cosine single ./results/results.txt
```

- `[index-dir-path]` takes the path to the directory where you store your index files (the [output] of the "build index" step).
- `[query-file-path]` path to the query file
- `[retrieval-model]` can one of the following: "cosine", "bm25", "lm"
- `[index-type]` one of the following: "single", "stem"
- `[results-file]` is the path to the results file, this file will be run with trec_eval to get the performance of your system.

Query Processing (Report 2, Dynamic)

```
python3 query_dynamic.py [index-directory-path] [query-file-path] [results-file]
Ex: python3 query_dynamic.py ./indexes/ ./data/queryfile.txt ./results/results-dynamic.txt
```

- `[index-dir-path]` takes the path to the directory where you store your index files (the [output] of the "build index" step).
- `[query-file-path]` path to the query file
- `[results-file]` is the path to the results file, this file will be run with trec_eval to get the performance of your system.

Note: Please run using Python 3!

Inverted Index

- To optimize runtime efficiency, I set the memory constraint to unlimited.
- To make querying faster, I computed and stored each term's idf and cf in the inverted index. The format of the index is now:

  term df idf cf [[doc_i, tf_d], ...]
- I also created a `docLength` dictionary (key: docID, value: object that contains the sum of the tf and the sum of the squared tf * idf values) for each index. The dictionaries are found in `data/[index]-docLength.txt` and loaded prior to query processing for Report 1 and dynamically for Report 2.

Query Pre-processing

- Queries are tagged and pre-processed the same way the documents were in Project 1
    - Special terms, phrase, and stems are also identified the same way

Query Processing (Report 1, Static) - main ()

- Creates result output directory if necessary
- Loads inverted index and document length dictionary into memory
- Computes number of documents `N`, numbers of terms in collection `C`, and the average document length `avgn`
- Preprocesses queries using `parse()` in `preprocess.py`
- Sends queries to retrieval model to calculate similarity scores
- Sorts scores and selects top 100 documents
    - Note: some queries returned < 100 documents
- Writes top scores to `[results-file]`

Relevance Ranking - Vector Space Model using Cosine measure

- For loop in main() function iterates through each query and first calculates the similarity score numerator (i.e., inner product) for each query, document SC(Q,Di) pair.
- After the loop, the similarity scores are normalized for document length by dividing by the l2-norm of the query and the document.

$$SC(Q,D_i)=\frac{\sum_{j=1}^{t} w_{qj}\ d_{ij}}{\sqrt{\sum_{j=1}^{t}(d_{ij})^2 \sum_{j=1}^{t}(w_{qj})^2}}$$

Relevance Ranking - Probabilistic model using BM25

- I used natural logarithm to calculate `w` (i.e., idf).
- Because k2 could have a value ranging from 0-1000, I tested different k2 values and their effect on MAP. There was no noticeable difference.

$$SC(Q,D_i)=\sum_{j=1}^{t} w\left(\frac{(k_1+1)\,tf_{ij}}{\underbrace{tf_{ij}+k_1\left(1-b+b\frac{|D|}{avgdl}\right)}_{K}}\right)\left(\frac{(k_2+1)qtf_j}{k_2+qtf_j}\right)$$

$$w=idf=\log\left(\frac{N-n+0.5}{n+0.5}\right) \quad \leftarrow \textbf{\textit{IDF is used and normally defined as this!}}$$

k_1, k_2 and b are parameters to be empirically determined.
k_1:1.2 ; k_2:0-1000; b=0.75 (in many cases)

Relevance Ranking - Language model: Query Likelihood with Dirichlet smoothing

- Set the smoothing factor μ to the average document length in the collection. Because μ is multiplied with expected frequency of a term in the collection, it represents the expected number of times the term appears in any document.
- In order to score documents that did not contain the query terms (tfqi, D) = 0, I stored all the documents in which (tfqi, D) > 0 in a set `docsWithTf` and found the set difference between the set of all documents and `docsWithTf.` Each of the documents in the set difference were then passed to `LM()` with doc_tf=0 set as one of the parameters and incorporated in the document's similarity score.

$$P(Q\mid\theta D)=\prod_{i=1}^{n}P(q_i\mid D)$$

$$P(q_i\mid D)=\frac{tf_{q_i,D}+\mu\frac{tf_{q_i,C}}{|C|}}{|D|+\mu}$$

$$\log P(Q\mid D)=\sum_{i=1}^{n}\log\frac{tf_{q_i,D}+\mu\frac{tf_{q_i,C}}{|C|}}{|D|+\mu}$$

Query Processing (Report 2, Dynamic) - main ()

- Creates result output directory if necessary
- Loads filtered phrase index document length dictionary into memory
- Finds phrases in queries using `parsePhrase()` in `preprocess.py`
- Checks if phrase is in the filtered phrase index (phrases with df > 1). If it is, processes query using the filtered phrase index, otherwise sends it to the proximity index
- If not enough documents found, then query is processed using the single term index
    - The union of the scores generated using the phrase/positional and single index is found. For documents that exists in both, the average of the scores is found
- Sends queries to retrieval model to calculate similarity scores
- Sorts scores in descending order and selects top 100 documents
    - Some queries returned < 100 documents
- Writes top scores to `[results-file]`

Phrase Detection using Positional Index: isPhrase()

- If the positional index contains every term in the query, each term's posting list is loaded.
- The intersection of the posting lists is found in `intersect()`, which returns the documents in all of the posting lists.
- The heuristic I used to determine phrase validity is that "all the query terms must appear within 30 terms of each other. Specifically, the algorithm was:
    - Initialize pointers to point at each positional lists first element
    - Find max position among all positions that are being pointed at
    - Make this the middle of the range and set upper (upper = maxPos + 15 and lower (lower = maxPos - 15) bounds
    - Using the skip pointer concept, move other pointers to a position than is greater than the lower bound
        - If the pointer reaches the end of the positional list, return False
        - If all positions are within the range, then a phrase is found

Evaluation

Report 1

Retrieval Model	**MAP** **single term index**		**Query Processing Time (sec)**		**MAP** **stem index**		**Query Processing Time (sec)**	
	My engine	**Elastic Search**	**My engine**	**Elastic Search**	**My engine**	**Elastic Search**	**My engine**	**Elastic Search**
Cosine	0.2809	0.3326	4.172 s	0.428 s	0.3313	0.3225	3.726 s	0.468 s
BM25	0.4234	0.4300*	4.284 s	0.377 s	0.4425	0.4795	4.648 s	0.521 s
LM	0.4176	0.4310**	4.700 s	0.375 s	0.4377	0.4701	3.855 s	0.476 s

*Parameters: k1 = 1.2, k2 = 700, b = 0.75
**Parameter: mu = 431

Notes:
- Query Processing runtimes include query tokenization and scoring phase for both my engine and ElasticSearch.
- Used Luca's ElasticSearch-Demo code to perform Elasticsearch MAP and query processing time calculations

Report 2

Retrieval model	**MAP**	**Query Processing Time (sec)**
BM25	0.4535	386.474 s

Analysis

According to the Language Model powerpoint, Query likelihood with Dirichlet smoothing offers similar performance to TF-IDF & BM25 retrieval functions. My findings correspond to this statement. The MAP for all models was around 30-40%. The MAP of the single and stem index was better for BM25 and LM than Cosine by around 10%. This may be because a problem with the Cosine similarity measure is that longer documents are somewhat penalized although they indeed they might have more components that are actually relevant. Between the stem and single term index, the stem index had a slightly higher MAP. This is evidence that stemming improves effectiveness by providing a better match between query and a relevant document.

My engine's query processing time was shortest for Cosine and longest for LM. This may be because in addition to calculating similarity scores with documents with tf > 0, Query likelihood with Dirichlet smoothing also assigns probabilities to unseen terms in document. This is important because it aims to fix the estimation (score = 0 if a term is missing in document) and data sparsity (document may be relevant to query but the query term is absent from document) problems. This required more computation, thereby increasing the run time. The query processing time for the stem index relative to the single term index was slightly faster for my engine for but slightly slower for ElasticSearch's. Mine was faster most likely because my stem index is smaller than my single term index.

Across the board, ElasticSearch's query processing times were significantly faster than mine. This is because of ElasticSearch operates in a distributed environment. It partitions its index into different shards stored on a set of clusters. This allows queries to be processed in parallel, which is much more efficient.

My dynamic query processing program performed slightly better than my static query processing program did. The reason why it didn't have a significant impact could be because a relatively small number of phrases were sent to the phrase and positional indexes, and because a small number of documents were retrieved by these indexes. Specifically, phrases were sent to the phrase index only 11 times and to the positional index 6 times. Additionally, different indexes have different values for df, tf, cf, idf, and C, which could affect the similarity score calculation.