Konrad Rauscher
Introduction to Information Retrieval
Project 2
11/5/16

Status:
Completed,

except for Elastisearch

Things I wish I had been told prior to being given assignment:
NA

Hours to complete: 65

Instructions:
Code runs according to shell interface instructions specified by Luca and Arman.

Design:

Sources: *nltk.corpus.stopwords, py2casefold, BeautifulSoup, numpy, glob, re, math, pdb, sys, os, time, nltk.stem*

build.py *Builds the index, accepts 3 arguments*

Merely re-packaged Project 1 so that the following arguments can be passed:

1) [trec-files-directory-path] the directory containing the raw documents (e.g. fr940104.0)

2) [index-type] can be one of the following: "single", "stem", "phrase", “positional"

3) [output-dir] the directory where your index and lexicon files will be written (please make your program create it if doesn't exist).

Decided not to incorporate memory management as I was losing terms in my final lexicon due to a bug in my merge functions.

Note: takes a minute or two longer to run the first time, as a file containing the lengths of each document is written if one doesn't exist.

query.py

Built to accept the 5 arguments specified.

Necessary directories are created if not already existing.

Outputs time to create output file, name of output file, Queries are pre-processed in the same manner as was done for index construction, except for steps like stemming (which is performed on the queries at a later point, if necessary)

Based on the index type specified:

a) the queries will be processed in a corresponding manner (porter stemmer, etc.)

b) the proper index type will be passed to an evaluator function to be used with the specified retrieval model

Based on the retrieval model specified:

The relevant evaluation function is called which:

a) constructs an inverted index according to the specified index type

b) obtains the top 100 results (or less) (in the form: docID, score) from the specified retrieval model, and appends to a list of all results.

c) a .trec file is then created from this list of top 100 results

Vector Space - *Cosine*

Design:

go through each term in the query (if the term is in the index) and:

1) obtain the number of documents that contain the current term
2) obtain the posting list for the current term and document
3) obtain the denominator for the weight calculation (which is a normalized *idf* and involves the sum of term frequencies in a posting list)
4) for each entry (docID, tf) in the posting list:

1) obtain the tf of the current term in the particular document
2) use this tf to calculate the numerator for the weight calculation
3) calculate the weight for the current term and document
4) Do the cosine calculation (from book: $wf_{t,d} \times w_{t,q}$ where w is my normalized idf and ft,d is the term frequency of the current term at the current document in the posting list, and wt,q is the number of times a query term occurs in the query, which is incorporated implicitly as each query term is iterated over for this calculation) using the variables obtained in the preceding steps and add the result to the proper docID in the Scores dictionary that has docIDs as keys and scores as values

5) return the top 100 results in Scores as determined by score

Notes:

Implementation is based off of that in textbook (pg. 125)

Tried using different weights such as the normalized idf formula given at the end of the slides on probabilistic evaluation measures and augmenting the weight with a constant, but defining weight by the improvement on idf given on the 4th slide in the vector space presentation gave the best results.

Decided against using a more complex wt,q as the queries in the query file are relatively short and simple.

Probabilistic - *BM25*

Design:

specify: number of documents in collection, average document length.

obtain: k1, k2, b

go through each term in the query (if the term is in the index) and:

1) obtain the number of documents that contain the current term
2) obtain a weight for the term (the normalized idf from the slides about probabilistic models)
3) obtain the frequency of the current term in the query
4) obtain the posting list for the current term and document
5) for each entry (docID, tf) in the posting list:
    1) obtain the length of the document corresponding to the docID
    2) obtain the tf of the current term in the particular document
    3) Do the BM25 calculation using the variables obtained in the preceding steps and add the result to the proper docID in the Scores dictionary that has docIDs as keys and scores as values
6) return the top 100 results in Scores as determined by score

Notes:

Tuning of k1:

starting value: 1.2

increasing by increments of .1 only resulted in lower MAP score

decreasing by increments of .1 resulted in no change in MAP score

Tuning of k2:

starting value: 500

tuning of this parameter did not seem to make a difference; setting k2 to 1 resulted in no change in MAP score, and increasing to 800 did not result in a difference either.

Tuning of b:

b = 0.75

was able to increase MAP score by .006 by tuning b to .55 (other values of b around .55 resulted in slightly worse scores)

Implications of tuning

I think tuning these parameters would have a more noticeable impact if my MAP score was higher.

Language Model - *Dirichlet*

Design:

specify: mew as 1 (tuning will be discussed in results)

obtain: the total number of terms in the collection, C

go through each term in the query (if the term is in the index) and:

1) obtain the posting list for the current term and document
2) obtain the sum of all term frequencies in the posting list (tf, collection)
3) for each entry (docID, tf) in the posting list:
    1) obtain the tf of the current term in the particular document (tf, document)

    2) Do the Dirichlet calculation using the variables obtained in the preceding steps and add the result to the proper docID in the Scores dictionary that has docIDs as keys and scores as values

4) return the top 100 results in Scores as determined by score

Notes: chose Dirichlet because seemed simplest to implement.

Tuning:

Tried setting mew as average document length, but found best value to be 1 (an .023 MAP improvement over my two original mew values, 2000 and avg doc length). Again, I think tuning these parameters would have a more noticeable impact if my MAP score was higher.

query_dynamic.py

Built to accept the 3 arguments specified.

Necessary directories are created if not already existing.

outputs time to create output file, name of output file, as well as distribution of index types that were used to process the queries.

Retrieval model chosen: bm25, because of superior performance in query.py

Backup index chosen: stem, again because of super performance in query.py

Design:

1) phrase index is loaded into memory

2) the list of queries is obtained from the query file

    1) two lists of queries are created:

        1) one is procesesed for use with stem and positional indexes

        2) the other for phrase (so ‘apple’, ‘pie’, becomes ‘apple pie’

3) initialize queryResults list to store all top 100 documents that are produced from each query

4) for each query in the query list

    1) initialize list to store top 100 documents and their corresponding scores.

    2) sum up all of the document frequencies for all phrase terms in the query that has been processed to consists of phrases

    3) If this sum is greater than the determined document frequency threshold then try using phrase index to obtain list of top 100 documents using bm25.

        1) store results to top 100 list

        2) if the number of documents returned is lower than the returned documents threshold, then use stem index instead, and overwrite top 100 list.

    4) else, if sum not greater, use positional index

        1) store results to top 100 list

        2) if the number of documents returned is lower than the returned documents threshold, then use stem index instead, and overwrite top 100 list.

    5) append top 100 list to queryResults

5) write .trec file using queryResults

<u>Stem</u>
No modifications were necessary to retrieval model functions (in this case bm25).

<u>Phrase</u>

No modifications were necessary to retrieval model functions (in this case bm25). All that was different was processing the queries differently (merely using the same phrase creation process that I did when creating my phrase index).

Note: I chose to use a 2-term phrase index as the document frequencies for 2-term query terms was already quite low and 3-term would almost certainly make lower (because more specific).

<u>Positional</u>

BM25 retrieval model had to be modified so as to only use terms that appear next to each other (that is, they form a word n-gram) to calculate the score of a document. This is done by obtaining a list of triples giving docID and the term position in p_1 and p_2. on a per-term basis. The formula for this can be found in the textbook on page 42. Changing the k (the distance allowed between ‘adjacent’ words from 1 up to 3 did not make a meaningful difference in terms of the # of relevant documents returned).

<u>Thresholds:</u>
document frequency (phrase or positional)
number of returned documents (switch to stem)

In order to decide on these thresholds, I first set both to 100, so that the stem index would be used for all queries. This resulted in a MAP of 0.2844, which I used as a baseline from which to improve.

Next, as many possible phrase (About half) (document frequency threshold to 1, number of returned documents 1): 0.1042

Finally, as many possible phrase (About half) (document frequency threshold to 100, number of returned documents 1): 0.1260

Ended up with setting both thresholds to 20, as this tuning not only seemed reasonable in terms of the number of results that where be generated on a per-query basis, but this tuning also resulted in a relatively high MAP score of: 0.1521

Evaluation:

Report 1

Retrieval model (BEFORE tuning)	MAP single term index	Query Processing Time (sec)	MAP stem index	Query Processing Time (sec)
	My Engine	My Engine	My Engine	My Engine
a) Cosine	0.1057	0.484481	0.1780	0.894841
b) BM25	0.1189	0.482348	0.2844	0.860422
c) LM	0.1042	0.498062	0.1502	0.877247

Although I ran out of time working on getting Elasticserach working (ground truth), I was able to obtain a worst case MAP score against which I could compare my results. This 'worst-case' MAP score was obtained by not sorting returned documents on score before writing top 100 to file and was: 0.0749.

The best MAP result I was able to obtain was a 24.63 using a phrase index and bm25. However, very few documents were being returned for each query (generally 3-10), so the precision calculations were conceivably much higher as a result.

There is often a large difference between the number of queries and the number of queries for which my retrieval models return documents for. Upon closer inspect, this was found to be resulting from terms in queries not existing in my lexicon. For example, 'nobel prize winners', 'hydrogen energy', and 'world court' are all queries that are comprised of words not existing in my lexicon. As a result, there are generally 17 queries represented in my .trec file instead of the full 27. I expect this to be drastically decreasing my MAP score across all retrieval models.

In terms of why stem index did relatively better with MAP scores than did the single index, because stemming helps reduce multiple words of the same meaning into one, there must have been a meaningful amount of these similar word in the documents being queried that were able to be incorporated into similarity scoring once stemmed, thus (conceivably) improving precision as well as recall.

In terms of why BM25 did relatively better with MAP scores than did the other retrieval models, I am unsure how to explain the disparity between BM25 with stem and the other two retrieval models with stem, other than that BM25 is merely better tuned for this data and these queries.

Report 2

Retrieval model	MAP	time
b) BM25	0.1521	2.445206

Given the breakdown below for the above result, the above results seems reasonable, according to the scores obtained for all of these indexes separately (see thresholds section of **dynamic_query.py**).

```
phrase index usage:  1
positional index usage:  17
stem index usage:  9
```

It is disappointing, however, that I was not able to achieve a MAP score with this dynamic query that was higher than the MAP of 0.2844 for BM25 with a stem index (report 1). Though I believe I could improve this with more time spent on my positional retrieval model as well as tuning of the thresholds.