Safia Shah

CMSC 476/676: Information Retrieval

C. Pearce, Spring 2022

Phase 4: Query Retrieval

I. Abstract

This report includes a summary and extensions made to the Phase 3 program in order to build a command line retrieval engine. The report will detail the changes made in order to create the query outputs for a given set of words and weights. The output for a query will include the document ID/filename and the cosine similarity score for the query terms in that document. The algorithm used is Term at a Time.The program is a mixed memory and disk based implementation. The command to run the Phase 3 program is as follows:

python3 retrieval.py [Query list in the format '-wt- -term-' for each][1]

II. Changes to Phase 3 for Phase 4

The very first step I took was rerunning the Phase 3 index.py program in order to get an up to date version of the dictionary and posting files for use in Phase 4. This decreases the amount of preprocessing that needs to be done as it takes away the need to read in, tokenize, remove stopwords, compute weights and more. What needed to be added was a dictionary to hold the query terms, function to compute the partial scores, and functions to retrieve the terms needed from the dictionary and postings file to store in an inverted index. Finally, once the doc products were done, I needed to compute the denominator of each document's cosine similarity score in order to rank each document to get the top 10.

III. Command Line Retrieval and Preprocessing

The very first thing to note on the command line retrieval of the query words and weights is that the command line does not allow me to test queries that have non alphanumeric characters as those characters have their own meaning in the command line. Also, because I included all alphanumeric characters in the preprocessing of the html files within the previous phases, if there are numbers within the query token, did will not strip them. I made this decision for consistency but did comment two lines of code that would remove numeric characters, if desired, within the main(). Because of this, there was not much preprocessing to do other than dowcasing the words.

In order to store the user's weights and queries, I read in the command line arguments using python 'sys' library and sys.arg to get the list of arguments. This list of arguments includes the .py filename at index 0. I looped through the arguments array returned by sys.arg to set every even index as the buckets key, and every odd index to the corresponding buckets value. The even indexes held the query terms and the index-1 (odd index) held the corresponding weight for each term. This allowed the program to read in any number of terms the user wants retrieved.

IV. Retrieving tokens and weights [Managing inverted index between memory and disk iA]

[1] Example: python3 retrieval.py 0.4 dog 0.6 cat

In order to compute the cosine similarity score for each document, the first step was to retrieve the information about the term, if it existed, from the dictionary.txt file and store it in a python dictionary with the value being a dictInfo node. The dictInfo node stores the word itself, the number of documents the word occurs in (count), and the line number of the first instance of that word within the postings.txt file. The dictionary.txt retrieval is done in getDictValue(). Using that information I then, in a function called getPostValue(), retrieved all the postings of each word in the query dictionary and added the linked list structure within the H_DICT, the inverted index. This was done by inserting nodes from the data at the start position of a term in postings.txt till the (start position + count). Doing the retrieval of the terms this way saves memory space because I do not load the entire inverted index, created in the previous phase, into H_DICT. For each run of the python program, it will only load in the query terms that the user wants if it exists.

V. Algorithm: Term at a Time and Computing Cosine Similarity

To compute the cosine similarity[2] scores I decided to use Term at a Time. Term at a Time processes each of the query word's posting lists one at a time, updating the scores for the documents. I used a dictionary named cosineSim to hold all the partial weights for the numerator of the cosine similarity. The numerator formula goes as follows:

DOT(weight of query term j * weight of term in currDoc for j)

The dot product for each document the query term is in is added to the corresponding cosineSim[doc_id] (essentially partial scoring that ends up being the summation of this Dot product for each document the terms are present in). At the same time, I also used a denominator dictionary to hold the part of the denominator (sum of squares) for each document. The formula is as follows[3]:

SQRT(wt(doc_id,w1)^2 + wt(doc_id,w2)^2)) (for however many query terms)

Both of these were done by the compNumDen() that is a member function for the written linked list class and traverses each query's postings list and conducts the numerator and partial computation of the denominator. The final step of calculating the cosine similarity is done by a finishCosSim() where the rest of the denominator computation is done and the numerator and denominator values are divided by each other for each document in the corpus. The full formula used goes as follows:

[Sum of Dot(weight of query term j * weight of term in currDoc for j)] / [SQRT((wt(Q,w1)^2 + wt(Q,w2)^2) * SQRT(wt(doc_id,w1)^2 + wt(doc_id,w2)^2))]

(number of words(w) and therefore squares and dot products are mutable)

VI. Complexity[4]

The complexity of the program is pretty good when it comes to the number of queries vs time. The worst case scenario to calculate the cosine similarity score would be O(n) to traverse

---

[2] I chose not to just compute the numerator of the CS score for ease of transition to phase 5

[3] This is only a part of the denominator formula: SQRT((wt(Q,w1)^2 + wt(Q,w2)^2) * SQRT(wt(doc_id,w1)^2 + wt(doc_id,w2)^2))

[4] (not really sure what to talk about exactly nor the accuracy of this)

through the query terms and then O(n) to go through each of the queries postings list and computing the numerator and partial denominator. That would make the complexity about O(n*n) in the worst case (because the structure is kind of like a double nested for loop). To compute the final cosine similarity score the is O(d) with d being the number of documents in the corpus because there is a single traversal through the cosineSim and denominator dictionary. If we talk about time complexity, a number of 3 queries comes back with results in about 0.5 to 0.7 seconds depending on the number of documents each query is present in.

VII. Output with sample queries given

diet (1 query so everything computes to 1, regardless of query weight)
{top information shows the numOfDocs the word is in (7), and the startline in postings (50973)}

```
safiashah@Safias-MBP SafiaShah_phase4 % python3 retrieve.py 1 diet
diet   7   50973
diet            Doc: 9    Weight: 0.17170166
Doc: 18   Weight: 2.63695937
Doc: 50   Weight: 0.10049950
Doc: 152  Weight: 0.04251902
Doc: 252  Weight: 0.15932848
Doc: 263  Weight: 0.20530612
Doc: 353  Weight: 0.02732211

009.html :  1.0
018.html :  1.0
050.html :  1.0
152.html :  1.0
252.html :  1.0
263.html :  1.0
353.html :  1.0
000.html :  0
001.html :  0
002.html :  0
safiashah@Safias-MBP SafiaShah_phase4 %
```

international affairs (these are two terms with the weight of 1 each)

```
safiashah@Safias-MBP SafiaShah_phase4 % python3 retrieve.py 1 international 1 affairs
international   114   1
affairs   38   18691
242.html :  1.0
353.html :  1.0
364.html :  1.0
361.html :  0.9899495749186378
279.html :  0.9899495225205446
336.html :  0.9899494547887429
289.html :  0.980580709949895
340.html :  0.9805806756909201
345.html :  0.9805806756909201
232.html :  0.9805806611599074
```

Zimbabwe (not present in my dictionary)

```
safiashah@Safias-MBP SafiaShah_phase4 % python3 retrieve.py 1 zimbabwe
zimbabwe  does not exist in this dictionary, so will not be used
000.html :  0
001.html :  0
002.html :  0
003.html :  0
004.html :  0
005.html :  0
006.html :  0
007.html :  0
008.html :  0
009.html :  0
```

computer network

```
safiashah@Safias-MBP SafiaShah_phase4 % python3 retrieve.py 1 computer 1 network
computer   42   18731
network   40   44642
047.html :  0.9997026929515919
145.html :  0.9997026905036076
290.html :  0.9997026891362955
388.html :  0.9997026858608346
022.html :  0.9759635850868791
027.html :  0.9547580722662456
064.html :  0.9547580651198097
164.html :  0.9424277374914801
223.html :  0.9218866846379657
156.html :  0.7307354624267798
```

hydrotherapy (with query weight .5, same result if using 1)

```
safiashah@Safias-MBP SafiaShah_phase4 % python3 retrieve.py .5 hydrotherapy
hydrotherapy   1   123659
273.html :  1.0
000.html :  0
001.html :  0
002.html :  0
003.html :  0
004.html :  0
005.html :  0
006.html :  0
007.html :  0
008.html :  0
```

identity theft (according for my postings list for these words, they dont occur in the same documents - which is why the computation looks mostly the same)

```
safiashah@Safias-MBP SafiaShah_phase4 % python3 retrieve.py 1 identity 1 theft
identity   19   67490
theft   3   126688
019.html :  0.7071067811865476
245.html :  0.7071067811865476
298.html :  0.7071067811865476
348.html :  0.7071067811865476
027.html :  0.7071067811865475
043.html :  0.7071067811865475
235.html :  0.7071067811865475
243.html :  0.7071067811865475
272.html :  0.7071067811865475
292.html :  0.7071067811865475
301.html :  0.7071067811865475
303.html :  0.7071067811865475
```

Other queries showing multiple terms, effect of weights (if implemented)

```
safiashah@Safias-MBP SafiaShah_phase4 % python3 retrieve.py 0.1 international 0.2 political
international   114   1
political   158   322
332.html :  0.9998773157035015
276.html :  0.9995235970414726
351.html :  0.9995235848006842
362.html :  0.9995235810497237
331.html :  0.9977657581384421
286.html :  0.9960915420046714
287.html :  0.9915486688258235
298.html :  0.9915485160750672
345.html :  0.9898849542831444
237.html :  0.989884921991987
232.html :  0.989884915750828
184.html :  0.9898849057146663
```

```
safiashah@Safias-MBP SafiaShah_phase4 % python3 retrieve.py 1 international 1 political
international   114   1
political   158   322
242.html :  0.9998127483326059
338.html :  0.9998127378772007
289.html :  0.9997866278719877
010.html :  0.999220245870664
340.html :  0.9992202331420752
348.html :  0.9992202239397409
364.html :  0.9963299650226147
368.html :  0.9871677550295823
361.html :  0.9871674798917787
377.html :  0.9871674784200208
280.html :  0.9871674137048789
025.html :  0.987167393268963
```

You can see the difference in the top 10 documents. The query with a higher weight will have documents with that query score higher (in the first case its documents with political)

```
safiashah@Safias-MBP SafiaShah_phase4 % python3 retrieve.py 0.3 presidents 0.1 solving
presidents   24   67466
solving   11   67510
363.html :  0.9532306598793445
052.html :  0.9486832980505138
255.html :  0.9486832980505138
289.html :  0.9486832980505138
333.html :  0.9486832980505138
338.html :  0.9486832980505138
339.html :  0.9486832980505138
345.html :  0.9486832980505138
351.html :  0.9486832980505138
358.html :  0.9486832980505138
369.html :  0.9486832980505138
376.html :  0.9486832980505138
```

```
safiashah@Safias-MBP SafiaShah_phase4 % python3 retrieve.py 1 presidents 1 solving
presidents   24   67466
solving   11   67510
363.html :  0.9877629538838235
019.html :  0.9374253657896442
331.html :  0.937425258134701
036.html :  0.7071067811865476
256.html :  0.7071067811865476
333.html :  0.7071067811865476
052.html :  0.7071067811865475
082.html :  0.7071067811865475
156.html :  0.7071067811865475
172.html :  0.7071067811865475
233.html :  0.7071067811865475
235.html :  0.7071067811865475
```