CS47300 Fall 2019 Project 1 Solution Walk-Through

Part 2: Part 2 is really just running the commands given to you

1. Total Number of documents in corpus: run ‘galago dump-index manifest /path/to/index/corpus’
   
   Answer: 3204

2. Number of documents containing ‘retrieval’: run ‘galago doccount --x+retrieval --index=project1-index’

3. There are a couple ways to get the documents containing the word ‘Rice’. You could use batch search:
   ‘galago batch-search --index=project1-index --query="Rice"’

   Or you could use batch search with a json file.
   Or you could use the dump-key-value command on the postings folder in your index, which will
   return doc id’s that contain the word ‘rice’ (only lowercase because of some preprocessing),
   and then you’d want to translate those doc id’s into document names to look at the actual
   original document (although not entirely necessary). You would use the doc-name command
   for this.

   Whatever method you used, you should get the same files as listed below.

   unk-0 Q0 /homes/cs473/project1/cacm/CACM-1677.html 1 -6.73131542 galago
   unk-0 Q0 /homes/cs473/project1/cacm/CACM-0302.html 2 -7.31091066 galago
   unk-0 Q0 /homes/cs473/project1/cacm/CACM-1312.html 3 -7.3263718 galago
   unk-0 Q0 /homes/cs473/project1/cacm/CACM-2312.html 4 -7.33037915 galago

4. You use batch search for this section

   Query = ‘information retrieval’

   unk-0 Q0 /homes/cs473/project1/cacm/CACM-3134.html 1 -5.27224741 galago
   unk-0 Q0 /homes/cs473/project1/cacm/CACM-1699.html 2 -5.32447710 galago
   unk-0 Q0 /homes/cs473/project1/cacm/CACM-2288.html 3 -5.45138068 galago
   unk-0 Q0 /homes/cs473/project1/cacm/CACM-1681.html 4 -5.51583323 galago
   unk-0 Q0 /homes/cs473/project1/cacm/CACM-0891.html 5 -5.60211040 galago

   Query = ‘machine learning’

   unk-0 Q0 /homes/cs473/project1/cacm/CACM-2979.html 1 -5.82114865 galago
   unk-0 Q0 /homes/cs473/project1/cacm/CACM-2217.html 2 -6.06939866 galago
   unk-0 Q0 /homes/cs473/project1/cacm/CACM-2471.html 3 -6.09832118 galago
   unk-0 Q0 /homes/cs473/project1/cacm/CACM-1699.html 4 -6.34474011 galago
   unk-0 Q0 /homes/cs473/project1/cacm/CACM-1170.html 5 -6.50645165 galago
Part 3: This part involves running your evaluation between the queries and their relevant data SEPARATELY or using a particular flag to output separate results.

This part was tricky! There were no relevant documents for query 50. If you examine cacm_fullpath.rel and look for relevant documents to query 50, you won’t find any. That would result in all the statistics reported being 0.

For this part, first you’d want to run a batch search the query in question (19 then 50) and the index. Then you’d want to run eval with the relevance data (.rel file) as judgements and the results of the batch search as the baseline file.

The results should be as follows

Query 19:
Number retrieved: 1000
Number of relevant docs: 11
Number of relevant retrieved: 9
p@20: 0.3

Query 50:
Number retrieved: 0
Number of relevant docs: 0
Number of relevant retrieved: 0
p@20: 0
Part 4:

There is no clear correct answer here as some liberties with implementation could result in fairly significant differences in ordering of relevant documents. Specifically, depending on how you decided to deal with situations involving $N, n, S, \text{ or } s = 0$, or situations where you encounter divide-by-zero errors. It’s important to intuit what exactly these divide-by-zero errors mean.

We’re going to solve a sample problem using a small corpus, relevant data, and a query. This is intended to show you what the different statistics mean, how they’re collected, and how to use them in the two models in the project.

Corpus = {
  Doc1 = “dogs are cool”
  Doc2 = “cats are cool fun”
  Doc3 = “dogs are fun”
  Doc4 = “dogs are fun cool”
}

Rel = {
  Query 1: Doc1, Doc3, Doc4
}

Query1 = “cool dogs”

Now, we compute the necessary statistics. The approach used here of collecting them all up front was done for convenience sake and is definitely not efficient. It would be better to collect them as you go and reuse them as you can, because you’ll notice NONE OF THE STATISTICS COMPUTED FOR EACH TERM IN THE QUERY CHANGE FROM DOCUMENT TO DOCUMENT YOU SCORE, ONLY BETWEEN QUERIES. This means, for estimate.out, I can collect statistics and compute the value of a term, and reuse it in the next document given the word from the query is in the document. You’ll notice this when I work out estimate.out. There’s an even stronger observation for base.out: none of the statistics computed for a term changes between queries. Once you compute a term’s value, you can reuse it between queries as well as documents.

Statistics needed for computation:
We score the model for a given query and document pair. The project requests two separate scores: base.out and estimate.out. base.out is simply idf, estimate.out is the binary independence model. The slides have the equation for the BIM.

In both of these next sections, we’ll go through and score each document in the corpus with the query.

Base.out section:

Base.out = IDF!

So, for each term in the query, we take the log of its IDF and add them together if the term in the query is also present in the document.

Query1, doc1 = “cool dogs”, “dogs are cool”

Both words from the query are present in the document, so:

Score(query1, doc1) = log(N/n_{cool}) + log(N/n_{dogs}) = .249877
Query1, doc2 = “cool dogs”, “cats are cool”
One word from the query is present in the document, the word “cool”, so we only use the previously computed idf for the word “cool”:
Score(query1, doc1) = log(N/n_{cool}) = .12494

Query1, doc3 = “cool dogs”, “dogs are fun”
One word from the query is present in the document, the word “dogs”, so we only use the previously computed idf for the word “dogs”:
Score(query1, doc1) = log(N/n_{dogs}) = .12494

Query1, doc4 = “cool dogs”, “dogs are fun cool”
Both words from the query are present in the document, so:
Score(query1, doc1) = log(N/n_{cool}) + log(N/n_{dogs}) = .249877

And when we order these documents based on their score, we get (asc to break ties):
Doc1, doc4, doc2, doc3

Estimate.out:
BIM: value per term = log((s / (S – s)) / ((n – s) / (N – n – S + s)))
Sum each value per term across all terms present in both query and document

Query1, doc1: “cool dogs”, “dogs are cool”
So, for each term in the query that’s present in the document, we utilize the statistics we collected to compute the score using the above equation. In this case, both words are present.
S_{cool} = num of relevant docs containing term = 2
n_{cool} = num of docs containing term = 3
S_{dog} = num of relevant docs containing term = 3
n_{dogs} = num of docs containing term = 3

Term “cool”: log((2 / (3-2)) / ((3 – 2) / (4 – 3 – 3 + 2))) = divide by 0 error! = log(0.01) = -2
How should we handle this particular case? Well, if we look at the equation, we’re actually making the denominator of the term in the log infinitely large as we approach 0. Which means, the whole term in the log should approach 0. But \( \log(0) \) is negative infinity. So we’ll just use \( \log(0.01) = -2 \) as a sufficiently small number to “approximate” 0 in the log statement.

Term Dogs: \( \log((3 / (3 – 3)) / ((3 – 3) / (4 – 3 – 3 + 3))) = \log((3 / 0) / (0 / 1)) \) divide by 0 error again! = 1

This time, we have two 0’s: the denominator of the top fraction and the numerator of the bottom fraction. Both of these serve to drive the resulting log towards infinity. So, for the sake of convenience, we’ll use 1 for this value.

Now, we have the calculations done for both terms in the query and document, now we sum them:

\[-2 + 1 = -1, \text{ so our score for the query1, document1 pair is -1.}\]

And, a valuable observation for this section of calculations as mentioned before, the actual values of each of these terms will not change from document to document. The value for the term “dogs” will always be -2, and the value for the term “cool” will always be 1 because you’ll notice the statistics collected won’t change between documents we score. The only thing that changes between documents is whether the term is present in both the query and the document. So, lets move on to document 2, as document 2 will highlight this well.

Query1, doc2: “cool dogs”, “cats are cool fun”

In this document, the word “dogs” isn’t present, so the only term that we use is “cool”

The statistics collected already can be used here again, as the stats collected don’t change between docs. So the score contributed from the term “cool” is still -2

So the score for this document = -2

Query1, doc3: “cool dogs”, “dogs are fun”

This time, the word “cool” isn’t present, the only word present in both query and doc is dogs.

The value for the term dogs computed earlier is 1, so the score for this doc = 1
Query1, doc4: “cool dogs”, “dogs are fun cool”

Each term in the query is present, just like doc 1, so the score will be the same as doc 1, score = -1

And now we order the documents based on score:

Doc3, doc1, doc4, doc2

As you can see, there’s ambiguity to the scoring and ordering in the sense that there are many different intuitions that could lead one to deal with divide-by-zero errors in a variety of ways. If I’d chosen a different way to deal with divide by zero errors, I could have ended up with an entirely different ordering as every document depended on solving this divide by zero dilemma

And the next step is to eval the results of base.out and estimate.out with the .rel data given, which requires running the galago eval command with cacm_fullpath.rel as the judgments file and the base.out as baseline, and then doing the same again except with estimate.out as baseline

Part 5

1. Well, word of mouth said the galago calls were quite expensive. Multi-threaded-ness and the map-reduce framework probably resulting in singular calls being quite expensive, especially on smaller datasets. But on bigger datasets, where the calls could be quite computationally expensive, we would probably see better performance relative to a non-multi-threaded, and non-map-reduce implementation.

2. The space consumed by the index is much smaller, which is expected as the documents are no longer stored as full text documents but as indexes. This means words and their presence in documents are represented by a 1 in a table as opposed to having the whole word there in the document. This means the same full text word isn’t repeated multiple times, only once in the index, with 1’s repeated multiple times to indicate the word’s presence in a document. Interestingly enough, this shrinkage in space also reflects some loss of information, namely order information. Indexes don’t preserve ordering, they’re somewhat “bag of words” esque. So some information is lost in indexes, and it’s somewhat reflected by the reduction in size.