Information Retrieval Project Two
By Ben Sikora
Complete
45 Hours

There are a couple things I wish I had known at the start of the assignment. First, I wish there were concrete and written up examples of the retrieval models. Despite watching the lectures multiple times, I was still confused about many of the symbols and the summations (when to loop through the query, document, etc). It took multiple office hour visits to clear my confusion, and even then, I was unsure of the results I was getting. Second, I wish that the elastic search tutorial was slightly more hands on. It was hard to translate what I had learned in the PowerPoint to code, even after trying my best to read through the documentation.

Design Document

General Structure

When going about my design for project 2, I had two goals in mind: The first was to never read through the posting list more than once. And the second was to minimize the number of duplicate calculations. The structure I then came to, was a series of dictionaries that stored "evaluation" nodes for both documents and terms. Upon reading the posting list, I would create and manage these nodes so that they contain all the relevant information for a given term or document. The goal of making "evaluation" nodes was so that when I needed a statistic about a term or document, I would only have to find the appropriate node and type the function. The evaluation nodes also made it intuitive for me to store document and term weights, which solve my initial set out of removing duplicate calculations. For simplicity, I also applied the node model to the queries. Although I believe there are other structures that minimize storage space better, my end-design made it very intuitive to calculate Cosine, BM25, and Dirichlet Smoothing relevancy scores. The entire structure is created, stored, and run through the **Evaluator** class.

Project 2 Key Class Guide

Evaluator	
self.queryNodeList	List of **queryNodes**
self.termDict	Dictionary of terms and their **evalTermNode**
self.docDict	Dictionary of docs and their **evalDocNode**

queryNode	
self.tag	The tagged number of this query
sefl.orginalQ	The original query, unaltered
self.procQ	Dictionary of processed query terms and their df in the query
self.totalWeightVSM	For Cosine: The total weight once its calculated
self.querySumSqrdVSM	For Cosine: The denominator of the weight calculation.

evalTermNode	
self.termID	The term this node represents.
self.idf	For Cosine: the idf of a term once its calculated.
self.termDict	Dictionary of all docs with this term and their associated term frequencies
Methods	
getDocFrequency()	
getCollectionTermFrequency()	
getTermFrequency(docID)	
getDocList()	

evalTermDoc	
self.docID	The doc this node represents.
self.docLength	The length of this document.
self.bvWeight	For Cosine: The weight of all terms in this doc
self.docDict()	Dictionary of all terms in this Doc and their term frequencys
Methods:	
getDocLength()	
getTermFrequency(docID)	
getDocList()	

Figure 1: Outline of my Evaluation Node Structure for Project 2.

Program Flow of Static Searching

The entire process for static searching is contained with the "staticSearching" method of the Evaluator Class.

1) Reading Files in and Query Processing

The lexicon, documents, and posting list are all read using their associated "read" methods. The read methods are the ones that build and maintain the evaluation nodes listed in Figure 1, and thus are one of the most crucial components of my project. For processing the query list, I used the "buildQuery" class, which utilizes a skeleton version of my project 1 to process the queries into terms and storing them in "queryNodes." All the query nodes are then handed back to the evaluator class and stored under "queryNodeList."

2) Calculations, Calculations, Calculations!

For each of the retrieval methods—Cosine, BM25, and Dirichlet Smoothing—there are three methods. The first, "allCalculations*type*", is a simple for loop to calculate scores for all the queries. The second, "calculating*type*Query", calculates all the doc scores for a given query. The final method, "calculating*type*Doc" calculates the score of a document given a certain query. The methods work in a tree structure, using the method preceding it, to handle all the calculations. There are also various helper functions to help with finding term weights. The scores of documents in a query are stored in dictionaries, which are then stored in the "queryScores" list.

3) Making the Results File

When all the calculations have been completed, the results are printed in "makeResults." For each query, the document scores are extracted from the dictionary, sorted, and then ranked. Then they are printed following the Trec Eval format, and given the appropriate tag of the retrieval method—Cosine, BM25, or Dirichlet smoothing

Program Flow of Dynamic Searching

The entire process for dynamic searching is contained with the "dynamicSearching" method of the Evaluator Class.

1) Phrase Searching

Since my project 1 implementation split phrases into two parts, I did the same thing for project 2. First my program begins with three phrases. It will read and process the associated queries, posting list, lexicon, and docs. Then for each query, if the any phrase term is equal to or above 3 (*Note Table 9 For This Decision),* the scores will be calculated using BM25. I chose BM25 for dynamic searching, because it was the fastest retrieval method (*Table 1*) and, since proximity searching requires a large amount of processing, I didn't want to extend the search time any more than I had too. Once all queries have been checked, the entire Evaluator is cleared, and the process is repeated for two phrases for all queries that were not given scores. A list of query IDs, "querysLeft," tracks all the queries which remain to be calculated following each stage.

2) Proximity Searching

Once all the queries with a high enough document frequency are scored, the remaining are passed to "runAllPositional" to be scored using proximity searching. "runAllPositional" uses a

similar tree structure of the retrieval methods, to find documents with the given query phrases. Since BM25 can already determine the presence and absence of query terms, it was important that my proximity search took the order of terms as a high priority. For a document to be scored by proximity search with BM25, it must have contained all the query terms in the exact order of the query and within three words of the term before it (*Note Table 10 For This Design Decision).*

3) Single BM25

For every query, if 100 docs have yet to be scored either by phrasing or proximity, then the query is passed to be scored by BM25 using statics single indxing.

4) Making the Results File

Since for every query there can be two sets of scores, one for phrasing or proximity and another for single, special handling had to be taken for printing the results file. In "makeResultsDynamic," the first set of scores, either phrase or proximity, are ranked and printed. Then if 100 doc threshold is still not meant, the second set of scores that were calculated using single indexing, are ranked, and printed under the initial scores.

Design Decisions for Retrieval Methods

All

1. If a query term is not in the collection at all, that term is skipped. This was chosen because although BM25 can handle having no terms in the collection, cosine and Dirichlet Smoothing become undefined (Dividing by 0 for idf of cosine, and log of 0 in Dirichlet Smoothing).

Cosine:

1. As specified in the project, I used this normalized weight function.
   a. $$w_{ij} = \frac{(\log tf_{ij} + 1.0) idf_j}{\sum_{j=1}^{t} [(\log tf_{ij} + 1.0) idf_j]^2}$$

BM25

1. The following constants for BM25 were chosen for both Elastic Search and My Engine (Note K2 is not an option with Elastic Search). Please see Tables 4-6 for the justification of this decision.
   a. K1= 1.2
   b. B= 0.75
   c. K2= 500

LM

1. For my language model, I chose a Dirichlet Smoothing query likelihood model, specifically the log version of Dirichlet Smoothing.
   a. $$\log P(Q \mid D) = \sum_{i=1}^{n} \log \frac{tf_{q_i,D} + \mu \frac{tf_{q_i,C}}{|C|}}{|D| + \mu}$$
   b. For u, I used 429 (the average document length in the collection) for both my own engine and Elastic Search.

A Note About The Elastic Search Engine:
Much of the code from my Elastic Search was from Eugen Yang's Elastic Search Demo on GitHub. My final version of Elastic search was modified for the specific retrieval methods and as described in ***Tweaking Elastic Search.***

https://github.com/eugene-yang/elasticsearch-demo

Results And Analysis

Report 1:

Table 1: Results using Static Searching across all retrieval methods with Single and Stem indexing. Comparisons to Elastic Search.

Retrieval Mode	MAP		Query Processing Time (sec)		MAP		Query Processing Time (sec)	
	Single Term Index				Stem Index			
	My Engine	Elastic Search	My Engine	Elastic Search	My Engine	Elastic Search	My Engine	Elastic Search
Cosine	0.3861	NA	5.516	NA	0.3864	NA	5.356	NA
BM25	0.4191	0.4277	1.729	2.650	0.4431	0.4414	1.365	2.107
Dirichlet Smoothing	0.4423	0.4216	3.015	2.867	0.4361	0.447	2.868	2.438

Table 2: Additional Results using Static Searching on My Engine across all Retrieval Methods with Single and Stem Indexing.

Index	Retrieval Method	Number Retrieved	Relative Retrieved	MAP	Rprec
Single	Cosine	1776	83	0.3861	0.3601
	BM25	1776	87	0.4191	0.4055
	Dirichlet	2000	75	0.4423	0.4215
Stem	Cosine	1884	85	0.3864	0.358
	BM25	1884	87	0.4431	0.3874
	Dirichlet	2000	80	0.4361	0.3915

Table 3: Additional Results using Static Searching on Elastic Search Across All Retrieval Methods with Single and Stem Indexing.

Index	Retrieval Method	Number Retrieved	Relative Retrieved	MAP	Rprec
Single	Cosine	NA	NA	NA	NA
	BM25	5082	101	0.4277	0.394

	Dirichlet	5082	101	0.4216	0.3628
Stem	Cosine	NA	NA	NA	NA
	BM25	7012	105	0.4414	0.3984
	Dirichlet	7012	105	0.447	0.3942

Analysis of Report 1

Comparison To Elastic Search

Overall, my engine's performance was relatively similar to Elastic Search. Looking at MAP scores alone, my engine performed better about half the time (Table 1). Granted the differences, were less than a single percentage point, except for single Dirichlet smoothing, but the similarity is reassuring for the integrity and accuracy of my own engine. At first, I was taken aback by Elastic Search's recall, consistently returning 20 or 30 more relative more documents than my own (Table 2 and 3), but when I saw the total number retrieved, 5000-7000, this explained why the precision scores were more similar. One thing I would be more curious to investigate would be how the MAP scores differ with the presence of special token queries. Many of the supplied queries in queryfile.txt were simple, none required special case handling, and I believe the differences in both engines will be exemplified on special tokens. In project 1, we only were to handle a few, but because Elastic Search is a much more comprehensive library, I predict its performance could really stand out on special token queries.

Comparison Across Retrieval Methods

Cosine performed slightly worse than BM25 and Dirichlet, having the lowest MAP scores and the highest processing time. The high processing time could be contributed to the high number of calculations needed per document (calculating the term weight for every single document for normalization), but it's much harder to contribute the lower MAP scores to a single reason. One thing that was interesting was cosine had a higher recall than Dirichlet, retrieving more relevant documents (Table 2), but its lower MAP score most likely means that the relative document rankings were not as high. Considering Cosine still has some flaws with document length, having been shown to penalize longer documents, I would be interested to see how Pivoted Cosine would compare.

While Cosine was slightly lacking in performance, both BM25 and Dirichlet performed better about equally well. BM25 stem had the highest overall MAP score, and Dirichlet Smoothing stem and single had the second and third highest. The processing times were also considerably less than Cosine with BM25 having the lowest across the retrieval methods. Again, it's hard to say what exactly is contributing to the better performance, but there are some theories.

The lower processing times with BM25 does align with what I was expecting. The BM25 equation only requires statistics with constant retrieval time under my design structure, and scores only have to be calculated for documents with the query terms. And although BM25 had only a slightly higher relative documents retrieved to Cosine, it does not have as such a severe penalty on higher document lengths, hence the higher MAP scores.

Although the processing times of Dirichlet Smoothing are slightly higher than BM25, considering every document is receiving a score for a query, it was still less far less than Cosine. What was interesting about Dirichlet Smoothing was it always had the highest number of documents retrieved and the lowest number of relative documents retrieved but was still able to achieve high MAP scores. Dirichlet Smoothing does not have such a large punishment of longer documents as Cosine, but it also has a large advantage in being able to score and distinguish documents that are lacking some or all query terms. Considering ranking came into play so much, Dirichlet Smoothing unique scoring likely made a significant impact on its MAP score.

Comparison Across Stem and Single

Although it is not the focus of this project, I thought it was also interesting to compare the performance of Single and Stem indexing. Overall, I was slightly surprised to see that stem only slightly increased MAP scores on just some of the retrieval methods across my own engine and Elastic Search. In addition, the R-precision scores were all consistently worse for stemming in my own implementation (Table 2). There are significant tradeoffs to using stem indexing— the fact that it is lossy compression and significantly increases the indexing time— so these underwhelming results do not justify stem's utility.

Changing BM25 Constants

Table 4: MAP Results Across different K1 Constants for BM25 For Both My Engine and Elastic Search.

	My Engine			Elastic Search		
K1 Values	0.6	1.2	1.4	0.6	1.2	1.4
Single MAP	0.4339	0.4191	0.4164	0.4521	0.4277	0.4249
Stem MAP	0.4343	0.4431	0.4374	0.441	0.4414	0.4488

Table 5: MAP Results Across different K2 Constants for BM25 For My Engine. Elastic Search does not support modifying K2.

	My Engine			Elastic Search		
K2 Values	100	500	1000	100	500	1000
Single MAP	0.4191	0.4191	0.4191	NA	NA	NA
Stem MAP	0.4431	0.4431	0.4431	NA	NA	NA

Table 6: MAP Results Across different b Constants for BM25 For Both My Engine and Elastic Search.

	My Engine			Elastic Search		
b Values	0.5	0.75	1	0.5	0.75	1
Single MAP	0.4239	0.4191	0.4424	0.4247	0.4277	0.4572
Stem MAP	0.4352	0.4431	0.4337	0.4352	0.4414	0.4504

I was curious if by modifying the constants in BM25, I would be able to improve the overall MAP performance for both my own engine and the Elastic Search. The results, however, were never clear enough to justify changing the constants from their empirically determined values. While there were some cases that I could significantly improve the MAP scores, specifically for a b value 1, it always seemed more random rather than a specific trend. Initially I was surprised that changing K2 was did not have any effect on MAP scores but given that K2 controls the weight of the query term frequency, and there were no queries with duplicate terms, it made sense it wouldn't have any effect.

Tweaking Elastic Search

To try to maximize the performance out of Elastic Search, I experimented with various pre-processing and scoring functions. The results were a mixed bagged. In pre-processing, I tried to create a custom indexer that split only on punctation but that dropped my MAP score to 26%. I also experimented with some more pre-processing steps, adding in more html tags and chaning the filters, but that only just hurt my results. For tweaking the scoring function, I had mixed success. As stated above, I did not have any luck in modifying the constants of BM25. But the one area that I did find the most success in was altering the u constant in Dirichlet Smoothing.

Table 7: MAP Results Across Different U Constants for Dirichlet Smoothing of Elastic Search.

U Constant	2000	429
Single MAP	0.4067	0.4216
Stem MAP	0.3975	0.447

By default, Elastic Search sets u to 2000 for Dirichlet Smoothing. I was able to increase the MAP scores significantly (Table 7), however, by changing u to 429, the average document length of the collection. This was the only modification of Elastic Search that I decided to make, when calculating the results in Table 1.

Report 2:

Table 8: Results using Dynamic Searching on My Engine.

Retrieval Method	MAP	Query Processing Time
BM25	0.4328	5.612

Analysis of Report 2

While adding in dynamic searching did improve my MAP score slightly from static single searching, increasing the MAP by 1.37% (Table 1 and 8), the improvement was not as high as I would expect. Dynamic searching is computationally expensive, and three additional seconds of query processing time doesn't really justify such a small bump in performance. As to why the improvement was so low, when looking through the results file, I noticed that very few queries had any phrases or proximal searching if it at all. At first, I thought there was a bug in my implementation, but after examining many of the documents by hand and thorough testing, I am fairly confident it had to do with low document frequency of the query phrases. If there were more query phrases with a higher document frequency or it if the proximal searching rules were

not as strict, I believe the dynamic searching MAP scores could improve on static searching significantly.

Picking a Threshold for Phrases

Table 9: Results using Dynamic Searching on My Engine Across Various Phrase Thresholds.

Phrase Threshold	1	2	3
MAP	0.4328	0.4328	0.4328
Number of Relative Retrieved	86	86	86
R-Precision	0.3975	0.3975	0.3975

The phrase threshold, in my implementation, is the document frequency number that a phrase must hit to prevent the query from going to proximity searching. I was expecting to do significant testing to find an appropriate phrase threshold, but my implementation struggled to find even just a couple query phrases. No matter what threshold I had set, even going to just one, the Trec Eval results were the exact same. And any higher threshold than 2, would prevent any of queries from being scored by phrases at all. The threshold, I came to for my implementation was just three. Although this threshold prevents any query from being evaluated using phrases, I thought the utility of the proximity index made up for that tenfold. Since many of the queries were a couple words, the proximity searching was able to find all the documents with phrases in addition to finding more. Proximity searching is also able to search queries with stop words and punctation. A downside of proximity searching, however, is that it takes significantly longer to parse a document, hence why I chose a phrase cutoff of 3 and not something larger.

Note: Since I have picked a threshold so that no phrases are scored, I know that you no longer have a way to verify that phrases are working properly. I thus have made a back door in my engine, that will change the Phrase threshold to 1 so you can see the Phrases in the result file. I will explain this more in the readme, but when running query_dynamic.py please add the flag -phraseThreshold to change the threshold to 1.

Picking a Threshold for Proximity Searching

Table 10: Results using Dynamic Searching on My Engine Across Various Proximity Thresholds.

Proximity Threshold	0	3	10
MAP	0.4328	0.4328	0.4308
Number of Relative Retrieved	86	86	86
R-Precision	0.3975	0.3975	0.3975

The threshold for proximity searching is the number of words there can be between terms for the proximal phrase to be counted. Again, I was expecting to do significant testing to find an appropriate threshold, but no small change in the counter could create any meaningful changes in

the Trec Eval results. Anything lower than 10 had almost no impact. I kept the threshold of three to allow for some flexibility in my phrase searching, but anything higher started to blend the purpose of even having proximity searching and overall hurt my results.

In Conclusion:

Overall, I was fairly pleased with this project and the performance of my search engine. Many of the retrieval methods were on par or even better than Elastic Search, which was really rewarding to see. I also felt that this project really helped me to understand the nuisances of the retrieval methods. I ended up making many "test" posting lists, where I would calculate the relevancy scores by hand and compare it to my results. Thus, when I was looking at processing times or changing the constants around, I always felt as if I had a solid understanding of what was happening. The only thing I wish I had for this project was more query test files, but I think that was kind of the point. I think I truly understand now how hard it is to test search engines when you don't know what all the relevant documents for a given search term were. Again, overall, very happy with this project and am excited to start on the next.