Joe Sheppard
CSC 575
Project Report

My system allows users to search for relevant documents in a local document corpus. The system has the capability to both take query input from the user via the command line, but also is pre-loaded with queries in the data set that were used for testing purposes.

The system uses a vector space retrieval model, uses tf-idf weights to weight the query and document terms, and uses a cosine similarity measure determine document relevance. Before any of these calculations are performed, the system also constructs an inverted index of all documents in the corpus that used to effectively perform the search. During both the construction of the index and the processing of the query, the system checks for and removes stopwords, which are words meaningless to the query or document such as "and", and "the. It also uses the Porter Stemming Algorithm to produce positive matches between terms conjugated with different tense and suffixes.

The document corpus that my system works on is a freely-available resource by the University of Glasgow, as suggested by the Professor of this course. The specific corpus I used is known as LISA, or the Library and Information Science Abstracts. This was a useful choice for a number of reasons. First, it provides a large set of documents, along with queries and their relevant documents. This allows for effective and efficient testing of the system. Another item of consideration for this document corpus is that every document relates to topics in a similar field: libraries and information. I wanted to try to use this as the basis of my system, as the similarity between documents would provide increased challenge, and would allow me to learn more about the effectiveness and limits of the Vector Space retrieval model. I would also be able to see the impact different strategies, including idf weights. This means terms such as "library", or more accurately the stemmed version, "librari", would have a very low idf value. This is because it would appear in most documents, and thus have a very low weight.

****

The system is broken down into numerous classes, each of which serve difference functions or steps of the process. Below I will discuss each, how it works, and it importance to the overall results.

TokenInfo:

This is a holder class that will store an idf value and a List of TokenOccurences, and provide access methods for each. It will serve as the class used for the value in the dictionary hashmap.

TokenOccurence:

Each TokenInfo will have a List of these. Similar to the TokenInfo class, it simply holds values containing the a document number, and the amount of times a token occurs in said document. A List of these owned by a TokenInfo instance represents all of the documents that contain a certain token, and the number of times it appears in each.

StopWords:

This is a static class that is used a tool to determine if a token is a stoword. It has a static initialize that reads a list of determined stopwords from a document, and loads them into a List. It also has a static method, isStopword(String token), that returns true if the token is a stopword, and false if it is not.

Stemmer:

This is another static class that has a static method stem(String arg). First, it removes punctuation and possession from the input, and passes it to the OPENNLP PorterStemmer class, and returns the result. OPENNLP, from Apache, is a natural language processing library for java, and I used its implementation of the Porter Stemming Algorithm for this project. Details can be found at : https://opennlp.apache.org/

DocumentLengthTable:

This is a static class that will store and computer document lengths for each document in the corpus, which is needed to normalize similarity scores in the final calculation. It has two static methods used for calculation of the data, addtoLength(Intger docNum, Double toAdd), and doneAdding(). As the documents are crawled, the crawler will call addToLength upon every token, adding the (squared) weight of the token to the proper document number. This will be stored in a static HashMap. Once all of the documents are crawled and indexed, the crawler will crawl the doneAdding method. This class will then take the square root of all the current values to finish the calculation. Finally, the getDocLength(int docNum) method simply fetches the proper data to be returned. There is a flag that will throw an error if the length of a document is asked for before the calculation is finsihed.

Crawler:

This is the main class that crawls all documents in the corpus and builds the inverted index. It reads each file that makes up the corpus, and uses the divider token to determine which tokens belong to each document. It then scans through each token in the document, and first determines if the token is a stopword. If it is not, it stems the token and increments or adds the token to a HashMap reperesenting tokens in the current document. This is known as thr HashMapVector.

Once the end of a document is reached, and the HashMapVector (HMV) is complete, it then must be added to the global dictionary. The program iterates through all entries of the HMV. If there is no entry in the dictionary representing the token, it creates a new TokenInfo to be added to the dictionary, and starts the empty Occurrence List with a TokenOccurence representing the document just crawled and the count that the token appeared. If there already exists an entry in the dictionary for the token, the TokenOccurence is created in the same way and added to the end of the Occurrence List for the proper TokenInfo.

Once all documents are crawled and indexed into the dictionary, the IDF of each token must be calculated. This requires another pass-through of the dictionary, as this cannot be computer until the inverted index is completely built. The IDF of a term is calculated by dividing the total number of documents in the corpus by the number of documents that the term appears in, and then taking the log base 2 of the result. This IDF value is then stored in the TokenInfo Class representing the term in the dictionary. During this pass through, we also scan the occurrence list to add the weight of each TokenOccurence to the document length of the appropriate document. The weight here depends on the IDF value, so it cannot be done until this step.

Finally, the doneAdding method of the DocumentLengthTable class must be called to finalize calculations of the document lengths.

SearchEngine:

This is the class that uses the inverted index to actually give a similarity score to documents and determine which ones could be relevant.

The first step is to parse the query provide by the user and build a HMV for the query in the same way that was done above. We still need to check for stopwords and stem each term to ensure positive matches.

Once we have a query vector, we create a HashMap, R, that maps Integers(Document Numbers) to Doubles(similarity score).

The program then iterates over every token in the query vector, and pulls the occurrence list from the dictionary corresponding to the token. Then, for every TokenOccurence in the occurrence list, we increment the score, in R, of the document of the occurrence by the weight of the term in the query times the weight of the term in the document. The term weight is the idf value of term, times the count of the term in the query/document. This creates the dot product of tf-idf values of the document and the query.

To normalize the score results, the system now does a second pass-through the R map, and divides each result by the document length, and the query length. This results in the cosine similarity score. If the resulting score is above a certain threshold, determined by a constant value, the document is added to an ArrayList to be returned as the results of relevant documents.

****

Results:

When performing a search with typical user-defined queries, the system works reasonably well, as expected. For example, a search with the query "Oregon" effectively returns all documents with the term "Oregon" in it, and no documents that do not. This is a 100% recall and 100% precision ratio. A search for "Oregon Libraries" returns the same results. This tells us that the tf-idf weights are working as intended. Since Oregon is a fairly rare term, and Libraries are in every document, the first term is weighted much heavier than the second, and the addition of the second term into the query makes little impact. We do not get results that talk about libraries, but not Oregon.

That being said, the above is an ideal example, as there are relatively few occurrences of the term "Oregon" in the corpus, and it is unique in that not many other words share similar stems or word structures. More interesting then, is the results on the queries provided in the document corpus. My system does not perform nearly as well here, as the queries are very vague and difficult to process, and often ask for very similar things. In addition, they usually relate to abstract concepts, rather than easily search-able terms. Even so, the system can typically return some relevant documents, even if it also returns some non-relevant ones. Below are the results of a few select queries in the corpus. The following were selected as they had a larger amount of relevant documents.

Query 6:
18 Relevant Documents
Returned Documents:

987	(NR)
2090	(R)
2593	(NR)
2722	(NR)
2866	(NR)
3103	(NR)
5626	(R)
5627	(R)
5830	(NR)

Precision: 1/3
Recall: 1/6

Query 8:
26 Relevant Documents
Returned Documents:

33	(NR)
39	(NR)
40	(R)
49	(R)
1021	(NR)
1038	(NR)
1043	(R)
1044	(R)
1191	(R)
1296	(NR)
1522	(NR)
1534	(R)
2665	(NR)
2806	(NR)
2934	(NR)
3045	(R)
3943	(NR)
4063	(NR)
4469	(NR)
4478	(NR)
4499	(R)
5506	(NR)
5526	(NR)
5551	(NR)
5722	(NR)
5731	(NR)
5789	(NR)

Precision:	8/27
Reacall:	4/13

Query 14:
11 Relevant Documents
Returned Documents:

576	(R)
1165	(NR)
1662	(NR)
2748	(NR)
2752	(NR)
4644	(NR)
4724	(NR)
5848	(NR)
6001	(NR)

Precision:	1/9
Recall:	1/11

Query 18:
53 Relevant Documents
Returned Documents:

440	(R)
651	(NR)
657	(NR)
936	(R)
1901	(NR)
1934	(R)
2167	(NR)
2401	(NR)
2426	(R)
3404	(NR)
3410	(R)
3704	(NR)
3705	(NR)
3708	(NR)
3928	(R)
3976	(R)
4400	(NR)
5687	(NR)

Precision:	7/18
Recall:	7/53

Query 23:
18 Relevant Documents
Returned Documents:

716	(NR)
2179	(NR)
4274	(NR)
4683	(NR)
5136	(R)

Precision:	1/5
Recall:	1/18

Average Results on the LISA Corpus:

Average Precision:	.266
Average Recall:	.151