CS54701: Project 1 - Latent Semantic Indexing

Brian Olsen

1 Introduction

For this project I decided to follow most of the advice given by the professor and the TA since I am not familiar with any commerical IR packages and have no strong preference for any languages dealing with matrices. For this reason I decided to use Indri to build my index, and matlab to perform the necessary computations on the term-document matrix. To generate this matrix I used the available Indri libraries and wrote my code under the specified app/ directory under the indri-5.6/ directory.

2 Indexing

2.1 Methods

1. I linked professor Clifton's directory and copied the indri directory to my local directory and built the Indri libraries using "configure" and "make".

2. I copied the parameter.xml file and added a list of stopwords and pointed to a location in my scratch directory to store the index. I kept the porter stemmer as I believed it didn't wasn't of interest which stemmer we used when comparing different retrieval methods as long as we stay consistent between methods.

3. I ran the IndriBuildIndex Command using the parameter.xml file to generate my index.

4. I used the indri-5.6/dumpindex/dumpindex commands to learn more about the index and the available libraries of Indri.

3 Compute Term Document Matrix with TF-IDF weighting

3.1 Methods

1. In the app folder I updated the TFIDFMatrix.cpp file to iterate through the inverted list generated by Indri to output a Matlab compatible inverted index file I labeled "TFIDFMatrix.dat". It output the values as such:

   ```
   <rowid> "\t" <colid> "\t" <tfidf_val> "\n"
   ```

   The TFIDF values I use qqq.ddd = ntc.ntc representation.

4 Singular Value Decomposition

4.1 Methods

1. I created a matlab file called "GenerateLSI.m" that takes in the previously generated TFIDFMatrix and a given$k$ value and computes the SVD producing three different matrices U_k, S_k and V_k.

2. From these matrices I computed a matrix I call L by doing the following, $L_k = U_k * S_k^{-1}$ and I wrote this matrix to an asciii file called "LSIMatrix.dat".

3. I also applied the transpose operation on $V_k = V_k^T$ and wrote this matrix to an ascii filie called "VMatrix.dat".

4. One issue is to choose a k value that will optimize the performance of the LSI method. If k is too small then there will be few but frequent set of concepts that produce a lot of false positives since so many terms are bunched into the same set of concepts. Conversely, if our k is too large we approach the idea that there are as many concepts as their ar documents and lose the advantage that LSI provides in that it groups similar concepts that often occur between the documents.

   - There are many methods that seek to optimize based on the results output by the LSI for a given k. This approach has the issue to optimize for certain queries or concept while avoiding generality for the entire corpus.
   - My approach: I will use a different method that tries using the eigenvalues from the SVD decomposition to assist in choosing the k. If you look at Figure 1. I show the plot used to visualize eigenvectors vs. the k-concepts. In some sense the eigenvalues represent the frequency of how often the concepts show up in the matrix. Each eigenvalue is relative the the sum of all eigenvalues for all k and the higher our k the more concepts are being covered in the corpus. The goal to minimize the k value to simplify the computation while maximizing the cumulative percentage of the corpus that is being represented by that k number of concepts. I first tested between 70% 75% and 80% to see if I could see any trend towards better performance. I ran 3 queries (see queries.txt to see corresponding queries) on the different k and compared the performance by comparing the related-retrieved counts. In fact I will do this for all comparisons since the numerator in both precision and recall is the related retrieved counts and so both denominators for precision and recall cancel out. For this comparison I am comparing LSI of different k but I will apply the same rational when comparing LSI and TFIDF. The following shows the relevant count from the top 20 results:

Cumulative Percentage	k	q1	q17	q21	Total
70%	397	12	8	7	27
75%	465	12	10	9	31
80%	543	11	11	8	30

- One last approach that I decided not to use was to choose my k based on a different data set or similar data set with the idea in mind to avoid choosing a k that will only work well on the current corpus. In the same way we can argue that technically LSI won't work well for queries that contain all words outside of the corpus so why should we bother concerning ourselves with choosing a k that generalizes outside of our corpus. I believe the method I have outlined above does a good job at staying general enough for queries containing words in the corpus and if we receive new documents we can recompute the k when we recompute the LSI matrices.

Based on the results I've gathered I chose to use a $k = 465$ and applied steps 1-3 using this value.

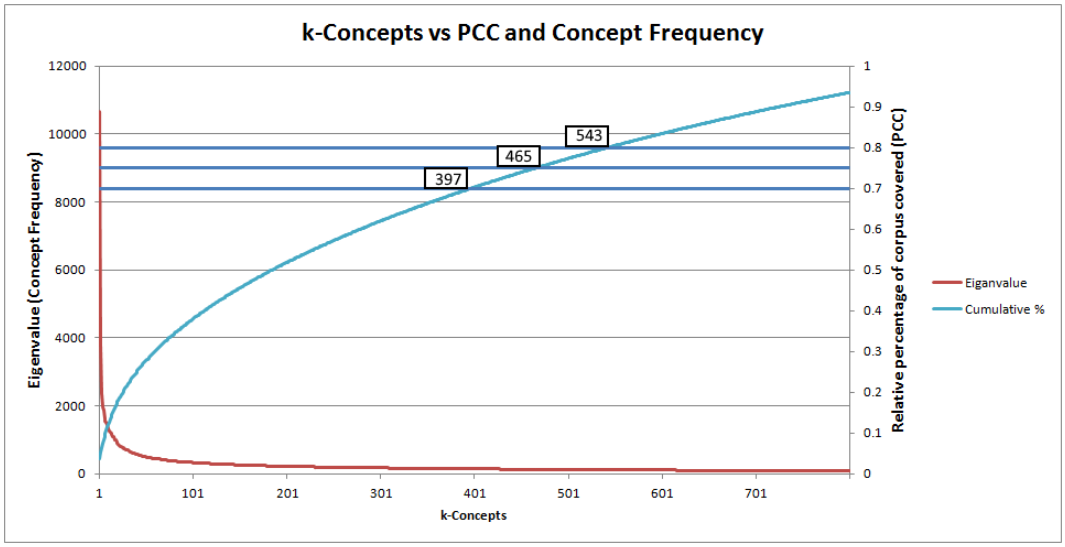

Figure 1: Eigenvalues of the S matrix (computed with the svd with a k equal to the number of documents) against the different values of k. Also plotted is the cumulative percentage of the area beneath the curve.

5 Evaluation

1. To analyze the difference between my results I wrote "QueryEvaluation.cpp" that reads in either the TFIDFMatrix.dat to compute standard TFIDF rankings as a baseline or the LSIMatrix.dat and VMatrix.dat to apply the LSI model to generate the document rankings. To quickly evaluate queries I also read in a changeable query document where each line contains a query id followed by the query.

   `<queryid>" "<query>"\n"`

   . The queryid will can be used to identify the specific query being ran, although the query itself is attached to the results. Once QueryEvaluation is invoked it will do the

proper computations for either LSI or TFIDF based on the inputs. As output for the top 20 documents it will copy the contents of the related document, and output the ranking, a link , the cosine similarity score, and the indri document id. See Figure 2.

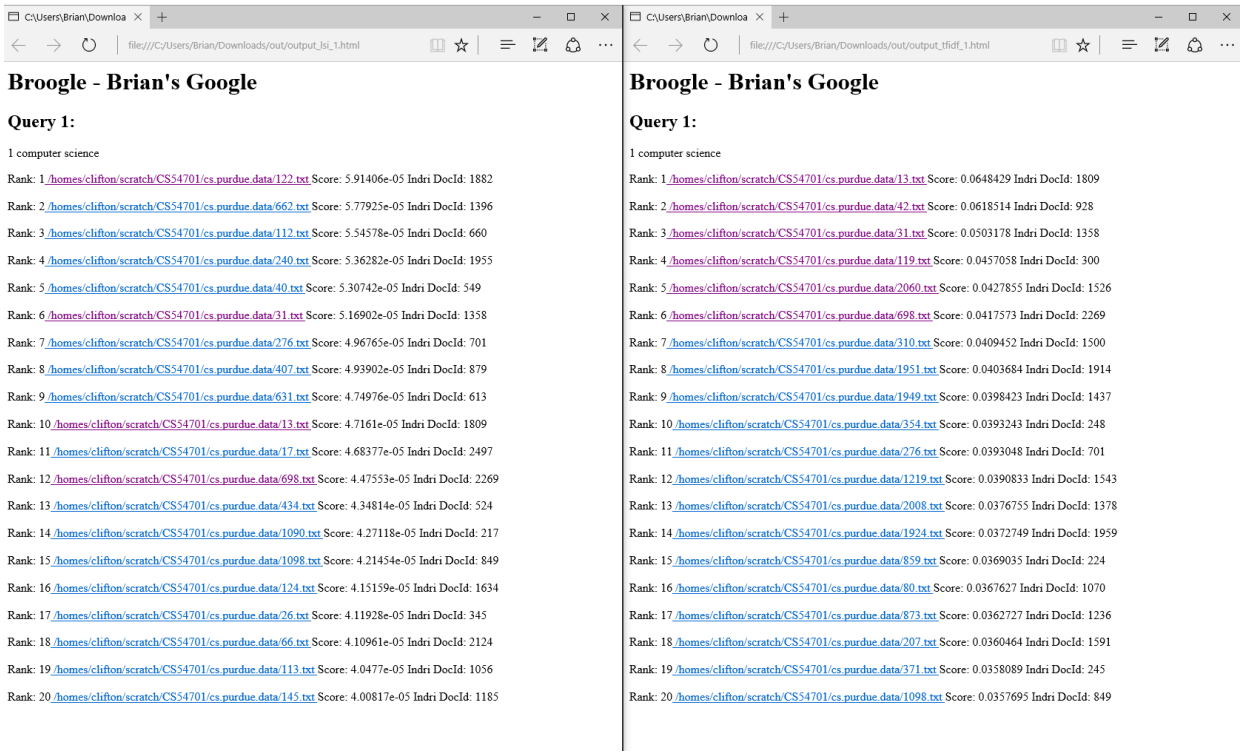

Figure 2: Output for query 1. On the left is the LSI ranking and on the right is the TFIDF ranking.

2. To judge the effectiveness of my retrieval engine I tried making queries of a few different categories that totaled up to 30 queries. The categories are, Generic queries that contain common words, Specific queries that contain less common words (more focused query), Homonym queries that contain homonyms that are shared with another query with different meanings, Queries with only one word in the corpus, and Queries with mixed topics. To see the specific queries please check the queries.txt added to this file.

Below are the comparison of the counts of the relevant docs retrieved between TFIDF method and LSI method from the top 20 results of each. The counts do not include documents that were retrieved by both systems.

	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14	q15	q16
TFIDF	3	2	2	6	10	0	5	0	0	0	2	4	3	3	2	1
LSI	10	4	7	5	10	11	5	0	0	0	0	1	3	2	1	2

	q17	q18	q19	q20	q21	q22	q23	q24	q25	q26	q27	q28	q29	q30	Total
TFIDF	3	4	2	4	1	3	2	3	4	2	0	1	2	1	75
LSI	3	5	5	6	2	3	2	1	5	3	0	0	1	0	97

Overall LSI is ahead but not by a large margin. In the specific queries the TFIDF model was able to find the most popular items but as certain terms (e.g. names of professors) became less common the lower part of the list was less populated with relevant information regarding those terms. Queries 8 - 10 and 27 - 30 caused both queries to do poorly either from the small number of words in the query existing in the corpus or random topics suprisingly didn't just affect LSI negatively. Perhaps this is because it is difficult to judge relevancy when the topic isn't focused. The LSI model did fairly well with the Homonym topics but not as well as I thought. I believe it would have done better if the topics in general weren't focused primarily on computer science topics.