Assignment 2: Latent Semantic Indexing

The objective of this assignment was to demonstrate the understanding of Latent Semantic Indexing by computing similarities between query and documents and rank the documents accordingly.

Latent Semantic Indexing:

Latent Semantic Indexing (also referred to as Latent Semantic Analysis, abbreviated here as LSI) attempts to address the issue of synonymy and independence in information retrieval systems by matching concepts that a query represents against concepts that the document represents.

In a corpus, more often than not, the documents represent a number of different concepts. Also, terms in a query have different synonyms (synonymy) and at times are also related to each other (dependence). LSI attempts to use this as a basis for creating a concept space such that this concept space is much smaller than the word space (mapping of terms against documents). The initial concept space consists of mapping of all terms against all documents in the form of a matrix. This matrix is then subjected to singular value decomposition. The matrices thus received are then used to represent the relationships between terms and documents. Two documents (of which one can be a query) can then be compared by comparing the concepts that they represent. This can be done by representing the documents in the form of a vector of terms and then finding the cosine similarity between them. The greater the cosine value, the greater the similarity. LSI provides formulae to perform this comparison.

Also, because the concept space is ideally much lesser than the word space, we can limit the number of singular values such that important semantic information is not lost and noise is removed from the system.

This assignment attempts to rank documents based on their similarity with the user query using concepts of LSI. It also attempts to show that varying the number of singular values affects the quality of the results.

Program Structure:

There are two java class files which are being submitted:

- **InvertedFile.java**: Consists of methods and data structures to prepare and retrieve term frequency (TF*IDF) values in the form of a double dimension array to allow construction of the term document matrix. The constructor takes two attributes: path to the directory where the documents are kept and the path to the file consisting of stop list words on separate lines.
- **LSI.java**: This class provides methods to handle the query entered by the user and then compute similarity of the query against documents in the corpus using Latent Semantic Indexing. Three attributes (directory path to corpus files, path to stop list file and number of singular values) need to be provided to the constructor. The first 2 attributes are the ones as required by the InvertedFile class as explained above.
  In order to use the system run this class and provide the required attributes.

Solution Explanation:

The constructors for **LSI** takes two or three arguments: directory path to documents, path to stop list file and the number of singular values (optional). When this constructor is invoked, it creates an object of InvertedFile class with the arguments it gets.

Next, the term document matrix is generated using the **createTermDocumentMatrix** method. This method retrieves TF*IDF values in the form of a double dimension array where rows represent terms and columns represent documents and generates object TD (denoting Term Document) of the Matrix class defined in the JAMA(JAva MAtrix) package.

The system proceeds to perform singular value decomposition by calling the **performSingularValueDecomposition** method. Here, a new object (called **svd)** of the class

SingularValueDecomposition is created by passing the term document matrix TD. The class consists of methods to retrieve the left singular matrix (using the **getU()** method), right singular matrix (using the **getV()** method) and singular value matrix (using the **getS()** method). Once we have these values, the system checks if the user entered some value denoting the number of singular values. If yes, then the system accordingly generates reduced versions of the left singular matrix, right singular matrix and singular value matrix by calling the **prepareMatrices** method. If the value is not there or it does not satisfy the validity condition then it is ignored and no reduction is done.

The system takes user queries. Like the previous assignment the system continues to take in user queries as long as **ZZZ** is not entered. The **handleQuery** function is then called on the query. This function essentially prepares the query vector using the **createQueryVector** and then invokes functions to compare the query against the documents in the system. The createQueryVector returns a object of class Matrix defined in the JAMA package. The package consists of classes and methods which support matrix arithmetic and singular value decomposition which is required in this assignment.

Once we have the query vector in the matrix form (a column matrix representing the query) we go ahead to see if it is a query which has some terms appearing in our term space. If no terms are there or the query is a combination of stop list words and words that are not there, the system returns no results and lets the user know of this particular case.

The system is now ready to perform similarity checks with the various documents in the corpus. The number of documents required per query in descending order of similarity is governed by the variable **relevantDocs** (defined to be 5 in the LSI class). Now, the system creates a TreeMap which maps similarity values to document name by calling the **findSimilarities** method. The method requires the query vector represented in terms of a matrix as was created by the createQueryVector method. For each document, the appropriate column matrix is obtained from the by performing appropriate operations on column representing the document in the right singular matrix using the **getDocumentColumnMatrix** method. The query matrix and the document matrix are now passed to the **getSimilarity** method which computes and returns similarity (cosine value) between the documentMatrix and queryMatrix by using the relationship:

$$\frac{\mathbf{q'd_j^*}}{\mathbf{|T'q||T'd_j^*|}}$$

Here **q'** denotes the transpose of the query matrix, $\mathbf{d_j^*}$ denotes the document matrix in the concept space. **T'** denotes the transpose of the left singular matrix.

The Matrix class in JAMA package provides for methods to perform matrix multiplication (using **times(Matrix)** method) and for finding inverse (using **inverse()** method) and transpose (using **transpose()** method).

The handleQuery method now calls the **displayRelevantDocuments** method. The method lists relevantDocs number of documents in descending order of the similarity value they have with the user query. For each document, its name and relevance score is printed. Also, the document name is passed to the **printWords** of the InvertedFile class which prints the first **contextWords** (defined in InvertedFile class and set to 15) number of words from the document.

Assumptions:

- The query vector represents term frequency i.e. number of times the term appears in the query and not simply term incidence.
- Query 'ZZZ' is used to exit the system.
- The stop list file always consists of terms which are on separate lines.
- Hidden files and sub directories inside the directory path where documents are located are ignored.

Execution Instructions:

Unzip **a2.zip** to get this Report.pdf, SampleRuns.pdf, Binary and Source folders. The Source folder consists of the src (consisting of the invert package which contains InvertedFile.java and the lsi package which contains LSI.java. I have also put in a folder under src consisting of all classes in JAMA so that there is no problem in running the system by using the javac approach) and bin (consisting of the invert package which contains InvertedFile.class and lsi package which contains LSI.class) folders.

Inside the Binary folder there is a file called LSI.jar. Go to the command prompt and type the following command:

> java -classpath **<path to LSI.jar>** lsi.LSI **<directory path on system where documents are kept> <file path on the system where stoplist.txt is kept>**

In order to limit the number of singular values type the following command:

> java -classpath **<path to LSI.jar>** lsi.LSI **<directory path on system where documents are kept> <file path on the system where stoplist.txt is kept> <number of singular values>**

Alternatively, we can run the system by taking the src folder inside the Source folder and change the directory to go inside the src folder. The lsi folder here consists of the the LSI.java file. While inside the src folder type the following commands to run the system:

> 1. javac -classpath . lsi/*
> 2. java lsi.LSI **<directory path on system where documents are kept> <file path on the system where stoplist.txt is kept> <number of singular values [Optional field]>**

Please note that the first command here needs to be run once only.

Note: Please use jdk 6. It is available on machines in the CSUGLAB

Sample Output:

Details of the different test runs are covered in **SampleRuns.pdf**. Please refer to that document.