IRE Assignment 1

Tasks

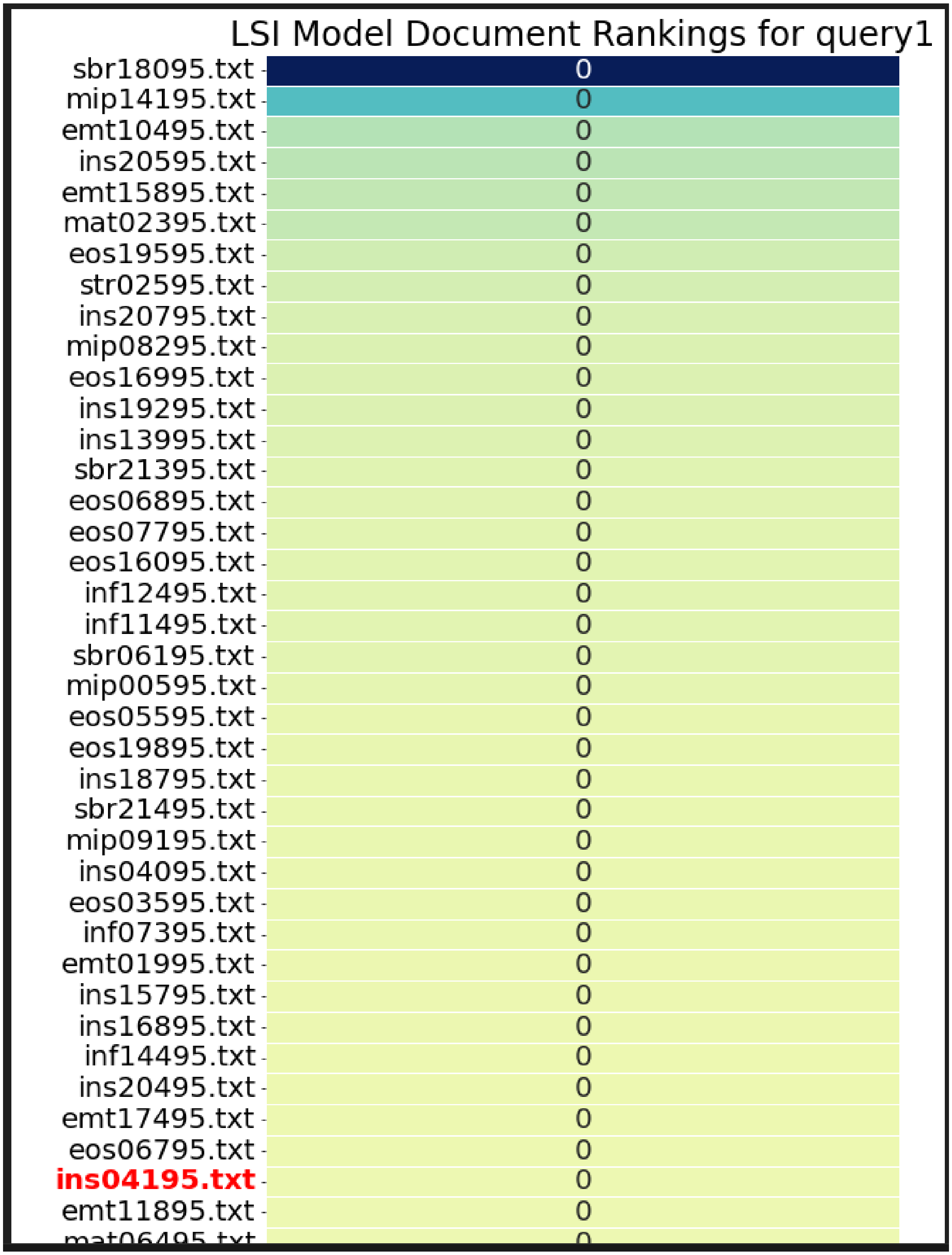
1. The code for task 1 uses 4 functions to read the raw data, clean it by removing non-alphabetic characters and applying **tokenization** (for 15 documents). This is the preprocessing step.
2. The stemmer function in task 2 generates the stems for each document (stemmed_text) and the overall frequency of each stem across all documents (stemmed_dict). This helps in the later steps, like generating the word cloud or for tf-idf. The **get_top_stems** function returns the most popular stems across all documents. This is also used in the word cloud. Here, only 15 documents are used, which can be randomly selected or sequentially selected.
3. In task 3, the preprocessing step is performed again, but for all files. Then, the term frequency dictionary is generated and converted to a dataframe (tf_matrix). The code for doing this is similar to the stemmer function, except there is the option to not stem the words as well. The idf dictionary is calculated separately, since idf is document-independent and remains fixed for a word/term. The idf dict and tf_matrix are used to get the **tf-idf matrix**. The same procedure is followed for calculating the tf-idf matrix and idf dictionary without stopwords, by removing the stopwords after the tokenization step.
4. Next, the **get_top_stems_per_doc** function extracts the columns (documents) of the tf-idf matrix and returns the top p stems for each document (column) as a list of series and as a dataframe. The **vector_model** function completes the vocabulary of the dataframe returned from the get_top_stems_per_doc function. This results in the **tfidf_vector_matrix** dataframe, which is the tf-idf model based on top p stems. Next, the **probabilistic model** and **LSI model** functions are used to structure the model and perform some computations beforehand, so that it is ready to rank based on the query.
   LSI_model is a single function that performs SVD on the tf-idf vector matrix using num_eigen number of eigenvalues to generate the document correlation matrix in a lower dimensional space. For ranking, the query is taken as a document or set of documents and the average score from the correlation matrix across the query relevant documents is computed as the score to rank each document.
   ProbabilisticInformationRetrievalModel is a class that handles the computation of scores and ranking based on probabilities (weights) that get updated. For now, for less computation time, I have done just 2 iterations - one for the initial ranking based on initial probabilities and the second iteration for the final re-ranking based on top 20 retrieved documents. More iterations can be run for a better ranking, however the results appear to be satisfactory even for 2

iterations. The tf-idf vector matrix is passed to this model for computation of scores (weights).

I have passed 3 queries to both models and the probabilistic model appears to perform better than the LSI model, since relevant documents appear to be ranked higher on average. The query for the LSI model is a list of relevant documents (one document by default) and the query for the probabilistic model is a random set of terms from a given set of documents. In this way, the relevant documents for both LSI model and probabilistic model will be the same and we can perform comparisons.

5. Task 5 involves redoing task 4 but after removing stop words from queries and using the tf-idf matrix without stop words computed in task 3. Stop words are removed before stemming. The same 3 queries from task 4 are passed to the new LSI and probabilistic models (without stop words) and it is found that the probabilistic model still performs slightly better that the LSI model.
6. On comparing the model comparison dictionary of results from tasks 4 and 5, we observe that the LSI model in the 2 cases performs about the same, with maybe a minor improvement in task 5. However, the probabilistic model performs worse when stop words are removed. This could be because of the low number of iterations we have taken.

   For each query, for visualization, I have constructed a heatmap to indicate the rankings. The highly ranked documents are closer to the top and the relevant document(s) for that query are highlighted in red.


LSI Model Document Rankings for query1
sbr18095.txt 0
mip14195.txt 0
emt10495.txt 0
ins20595.txt 0
emt15895.txt 0
mat02395.txt 0
eos19595.txt 0
str02595.txt 0
ins20795.txt 0
mip08295.txt 0
eos16995.txt 0
ins19295.txt 0
ins13995.txt 0
sbr21395.txt 0
eos06895.txt 0
eos07795.txt 0
eos16095.txt 0
inf12495.txt 0
inf11495.txt 0
sbr06195.txt 0
mip00595.txt 0
eos05595.txt 0
eos19895.txt 0
ins18795.txt 0
sbr21495.txt 0
mip09195.txt 0
ins04095.txt 0
eos03595.txt 0
inf07395.txt 0
emt01995.txt 0
ins15795.txt 0
ins16895.txt 0
inf14495.txt 0
ins20495.txt 0
emt17495.txt 0
eos06795.txt 0
ins04195.txt 0
emt11895.txt 0
mat06495.txt 0

Practical Requirements

I have included the required functions in the same jupyter notebook. The **TermDocumentMatrixProbModel** function is similar to the boolean model from the previous assignment and combines the preprocessing step and the boolean matrix generation step into a single function. The **TermDocumentMatrixLatentSemanticIndexing** function computes the tf-idf matrix from scratch and uses it to perform SVD and get the final LSI model. The function for query vector representation uses tfidf (tf of term in query * idf of term) to vectorize the query, and the **queryBooleanRepresentationProbModel** function computes the boolean representation of the query. The **RankingLatentSemanticIndexing** function computes the ranking for each document based on the LSI model and a given query. By setting the display parameter to true, we can print the results of LSI ranking. The **RankingProbModel** function is inbuilt into the probabilistic model class as the rank_documents function, so I have not created a separate function for this. In this model, the scores are also available as a class variable.