|  |  |
| --- | --- |
| Information Retrieval: An Introduction For SEOs  **FINAL PROJECT REPORT**  **CSC 575 : Intelligent Information Retrieval** | **Search Retrieval System**  AYESHA ALI , SWATHI BABU |

**CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Topic** | **Page No.** |
| 1. | [About the project](#About) | 2 |
| 2. | [Description of the System](#Description) | 2 |
| 3. | [Test runs of different possibilities](#Test) | 5 |
| 4. | [Evaluation of the System](#Evaluation) | 15 |
| 5. | [References](#References) | 18 |

**ABOUT THE PROJECT:**

The project is a SEARCH RETRIEVAL SYSTEM that runs on the local directory. The system has a crawler, indexer, and query processing components. The system recommends the top 10 documents based on the given query using the inverted index and cosine ranking. It also uses Rocchio relevance feedback to improve the system by receiving more relevant documents for the given query.

**DESCRIPTION OF THE SYSTEM:**

1. **Crawler**

We have a local based crawler that crawls through, local directory containing documents. This function implements some of the os module functions like the walk function. Generally this crawler will go through the local directory, each file as document is preprocessed to remove stop words, punctuations and converting to lowercase. This preprocessing is done by a separate function **preprocess\_words(line)**. We have imported nltk modules such as PorterStemmer to stem the tokens and remove stopwords to preprocess our tokens. After preprocessing, the system creates an inverted index for each document.

1. **Inverted Index**

Primarily, our inverted index is a dictionary, where key is token and its value is a nested dictionary containing the docId as key and count of the token in that document as value.

After the preprocessing step, the inverted index construction is done by **create\_inverted\_index(doc, docid, inverted\_index):** .

The docid is the name of the file identified when the crawler crawls through the directory.

Simultaneously , the doc lengths dictionary is also constructed using the function **doc\_lengths(inverted\_index, docLengths, NDoc)** . This function utilizes the **tf\_idf(dict, N) ,** such the for each token in inverted index ,we computed the IDF weight for each token , and incremented the length of D by. The document length in Doc Lengths dictionary is set to the square root of the current document length. (referred from Implementation Notes).

The entire process of creating Inverted Index and Doc Lengths is done for one time initially by calling the function **first\_time(filepath = '/Users/swathib/Downloads/med/MED.ALL').** (filepath given initially for test purposes.). It saves the inverted index, doc\_lengths dictionaries as json files in our local storage which are retrieved each time the system starts.

1. **Query Processing**

The main process of the system starts now. The **main()** function is called where the inverted index file (inverted\_index.json) and the doc lengths file (doc\_lengths.json) that were previously saved are retrieved to start the system. Then the total number of documents **NDoc** is calculated. This happens everytime the system starts. Then the system asks the user for a query using the input() function in python and is stored in the **query** variable. Once the user enters a query, the query undergoes the same preprocessing that the documents did using the **query\_process(query)** function i.e., the leading spaces are removed, punctuation and stop words are removed from the query, and it is converted to lowercase and the tokens in the query are stemmed (using Porter stemmer algorithm). This function returns the preprocessed tokens of the query **p\_query** and the vector form of the query required for further steps in the retrieval system in the form of a python dictionary **QtermDict** where key is the token and the value is the count of the token in the query (raw weight of the token for now).

1. **Retrieval Scores**

Using the query terms, query vector (in dictionary form), total number of documents and the inverted index, the scores of each document for the given query is calculated using the function **retrieval\_scores(p\_query, QtermDict, NDoc, inverted\_index).** This function goes through each token in the query and calculates the idf value for that term using the **tf\_idf(inverted\_index[token], NDoc)** function. The idf value is used to calculate the query length **Qlength** by incrementing the score by the square of the idf value. The scores of the documents are stored in a dictionary **rankDict** with key as document id and value as the scores. Then for each document that contains the term, the score of that document is incremented if the document already had a previous parsed token (the docid is in the rankDict dictionary) by **count of the token in that document \* idf \* count of the token in the query \*idf**. If the document is not in the rankDict, it is initialized with the same value rather than being incremented.The function then applies square root on the query length. The function returns the calculated query length and the dictionary with the scores for the documents retrieved for the given query. Then the control flow comes back to the main function where the **cosine\_Ranking(rankDict, Qlength,docLengths)** function is called to calculate the cosine ranking for the retrieved documents using the previously calculated scores, query length and the doc lengths that was retrieved from local storage. This cosine ranking function returns the dictionary **rankedDocs** sorted by the cosine ranking with key as the retrieved document id and the value as cosine ranking. The main function then calls the **printDocs(rankedDocs)** function to print the results in a user-friendly way. The printDocs function goes through the top 10 retrieved documents and prints the document id and using the function **print\_title(docid)** prints the title of the document retrieved. The function print\_title is especially for the dataset used in the evaluation of this project but it can very easily be customized to work on other documents. Followed by this, relevance feedback is applied based on user input.

1. **Rocchio Algorithm**

The system asks the user whether they want to provide feedback on the relevance of the documents. On confirmation, the users would be prompted again to enter the list of the relevant document IDs. Once the list is given, it is passed as argument to the function **feedback\_user(QtermDict, relevant, 10,rankedDocs,inverted\_index,docLengths,NDoc)** . This function returns the ranked Documents .

The **feedback\_user(QtermDict, relevant, 10,rankedDocs,inverted\_index,docLengths,NDoc)** implements all the process in **Step 4** but after the function of Rocchio Algorithm.

We implemented the standard IR approach Rocchio Method.

The **rocchio\_process(rankedDocs, rel, QtermDict,k,inverted\_index):** We create a dataframe of the relevant and the non relevant documents from the top 10 ranked Documents and counts from the inverted index (not a full document -term matrix). It's only the dataframe document term matrix of 10 documents.

We also considered a scenario where a query token may not exist in the inverted index, in that case the query token is added to the dataframe with count 0 for each of the documents.

We create numpy arrays for each of the Q vector, Relevant and Non Relevant documents

And call the **modify\_query\_rocchio(Q, R, NR, alpha, beta).** This function implements the following formula:

Rocchio formula : Q1= Q+(alpha\*mean\_R)-(beta\*mean\_NR)

We considered alpha= 0.5 and beta =0.25

We get the new query with modified term weights from the existing query.

Using the modified query, we perform the same processes in **Step 4 : Retrieval Scores**

to get retrieval scores and cosine ranked top 10 documents.

Upon getting the result of 10 documents, we observed that we got better results in terms of more relevant documents than the previous result without relevance feedback . We have cross checked from our benchmark test collections that contain the relevance documents for each query.

The user also has the option to skip providing feedback on the relevance of the documents, otherwise the user can enter the relevant documents and get the ranked documents . If the user would like to ask another query, they would choose the ‘Click for new Search’ button which would let them repeat this process in a new window.

**TEST RUNS ON MULTIPLE QUERY INPUTS AND THE INTERMEDIATE OUTPUTS:**

**TEST 1 → Running a Query with Relevance feedback**

Asking the user for the query

Graphical user interface, text, application, Teams

Description automatically generated

The user then gets the top 10 retrieved documents for the above query which can be seen by scrolling

Graphical user interface, text, application, email

Description automatically generated

Following this, the user is asked if they would like to give feedback

Graphical user interface, application

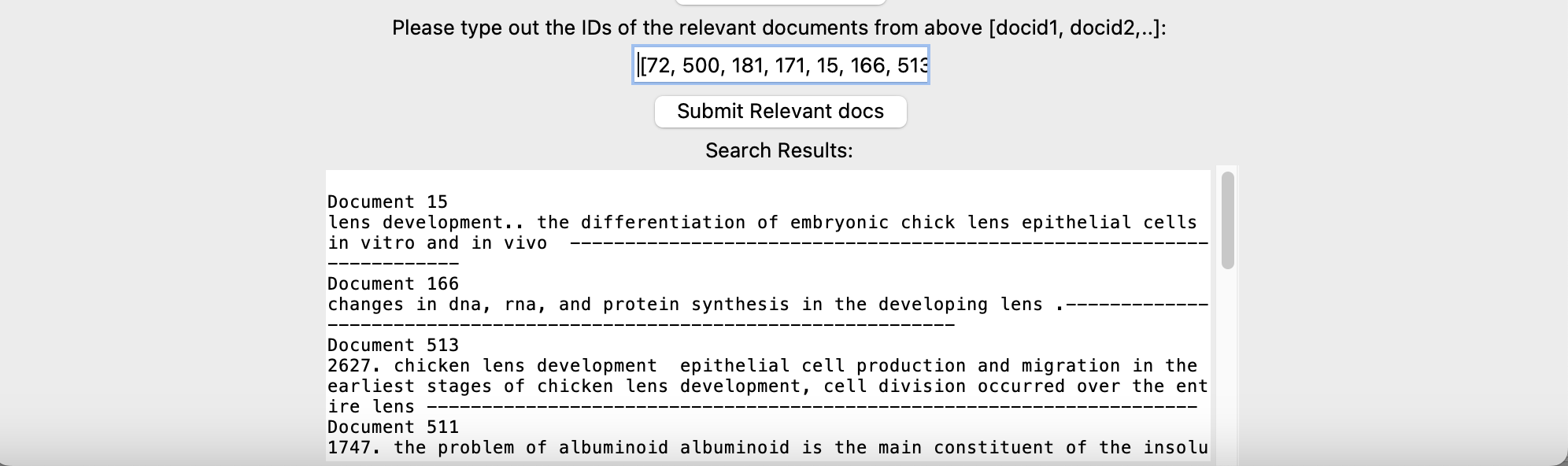
Description automatically generated

If the user says yes, then the user is asked to give a list of relevant documents

Graphical user interface, text, application, email

Description automatically generated

Then the list of the top 10 retrieved documents for the Rocchio altered query is printed



Then the system can be closed by shutting down the window or they can click on new search to continue

Graphical user interface, text, application

Description automatically generated

Since the user clicked on it, the system asks for the next query in a new window.

**TEST 2 → Running another query without Relevance feedback**

The system asks the user for the query

Graphical user interface, text, application, Teams

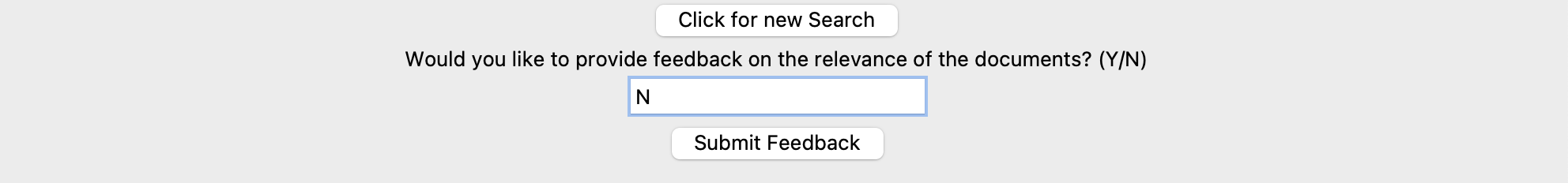
Description automatically generated

The system prints the top 10 documents

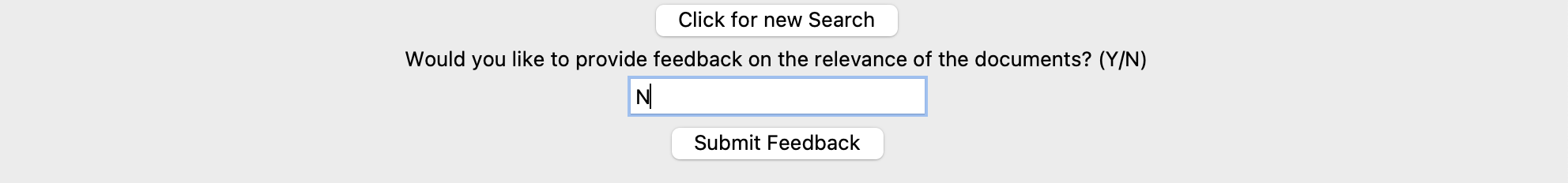
Graphical user interface, text, application, email

Description automatically generated

The system asks if the user would like to give feedback.



If the user says no, the system stops. Then the system can be closed by shutting down the window or they can click on new search to continue



The system proceeds as usual since we the user clicked on new search. We can run two more queries to test the efficiency of the system

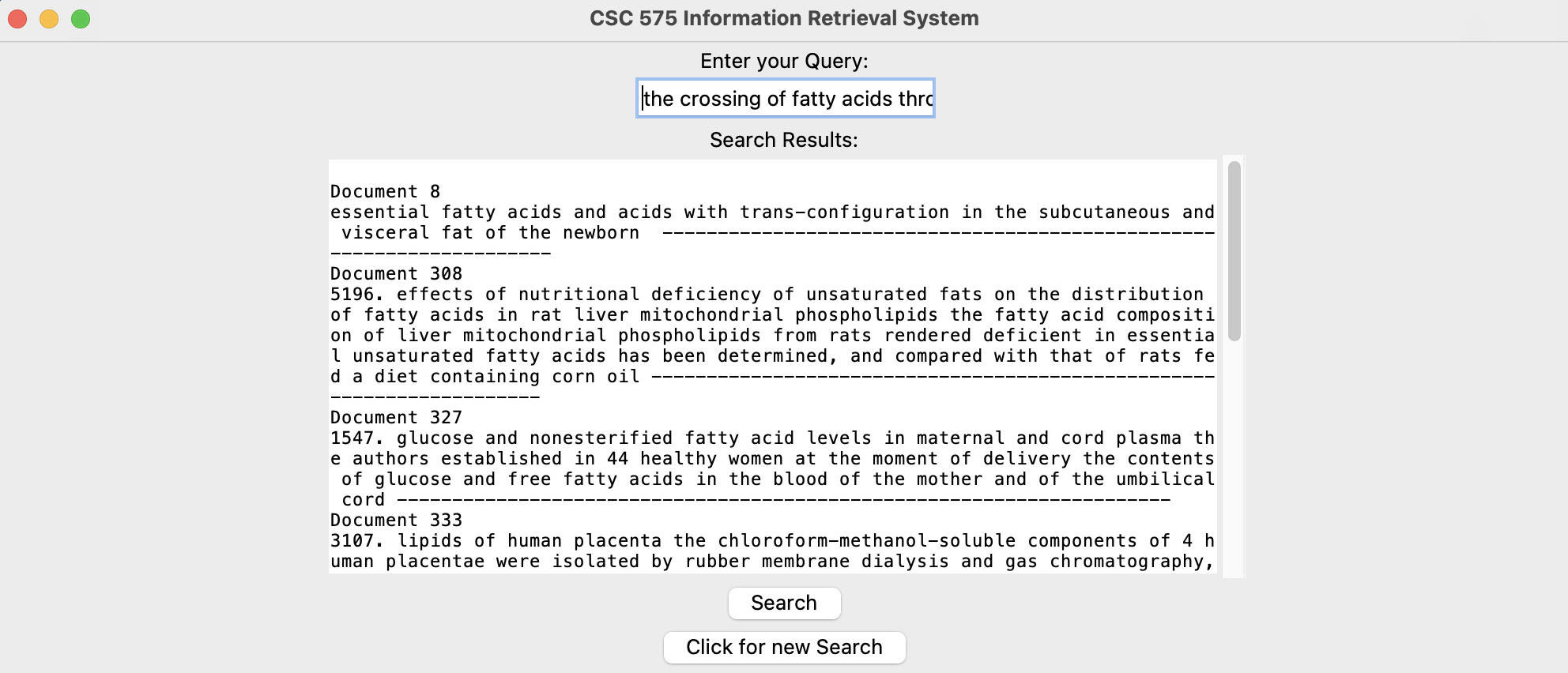
**TEST 3 → Testing another query with Relevance feedback**

Asking the user for the query

Graphical user interface, text, application

Description automatically generated

The user then gets the top 10 retrieved documents for the above query



Following this, the user is asked if they would like to give feedback

Graphical user interface, text, application

Description automatically generated

If the user says yes, then the user is asked to give a list of relevant documents

Graphical user interface, text, application, email

Description automatically generated

Then the list of the top 10 retrieved documents for the Rocchio altered query is printed

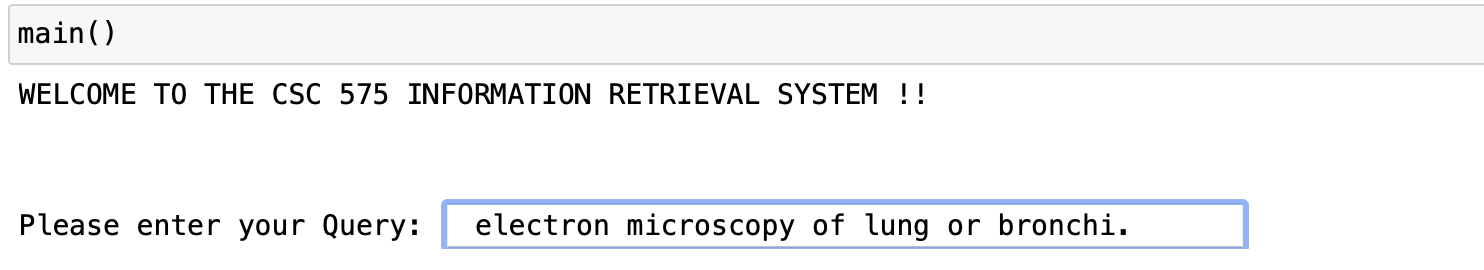
Graphical user interface, text, application, email

Description automatically generated

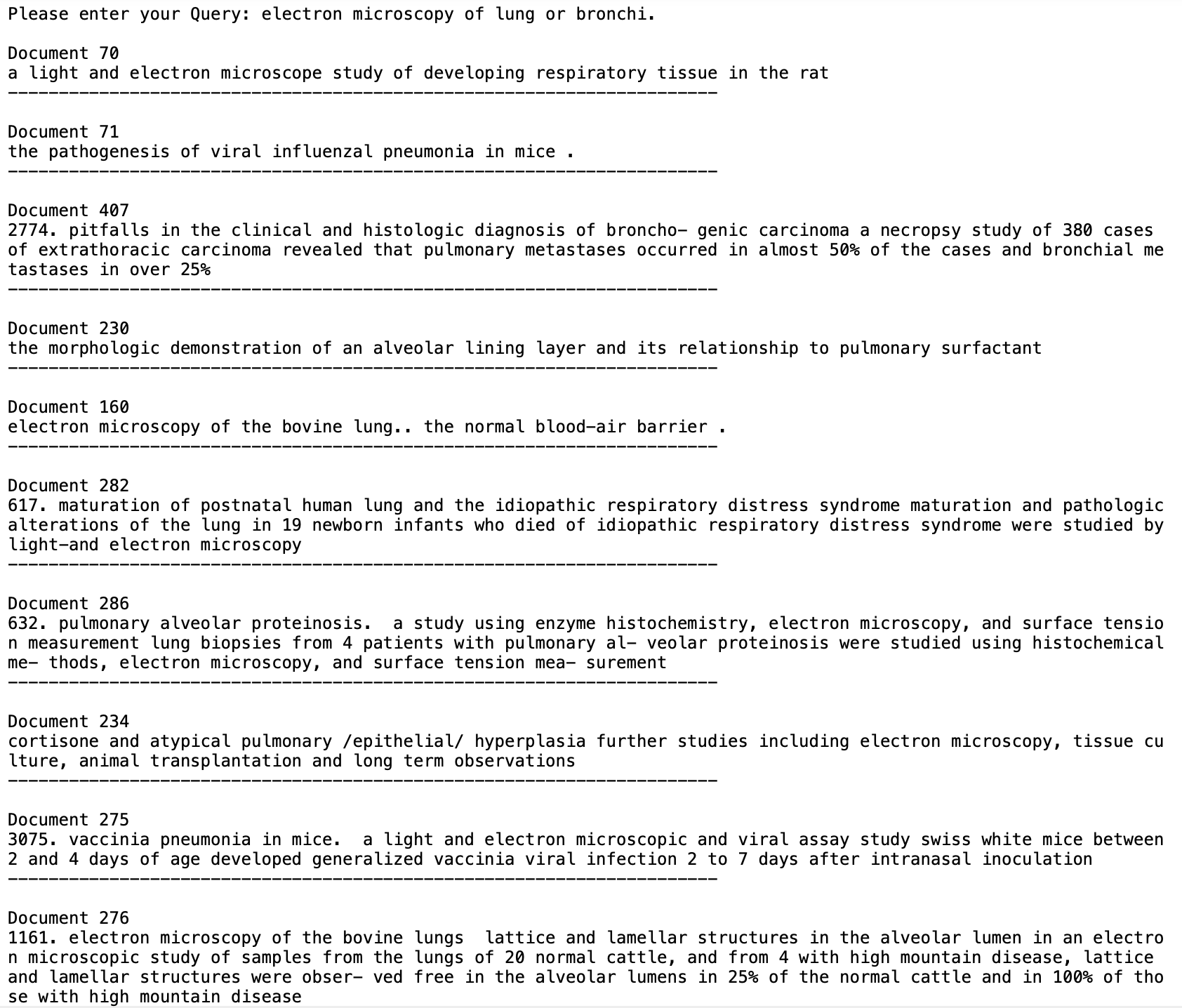
Then the user can continue or stop the system.

**TEST 4 → Testing another query with Relevance feedback with all the intermediate outputs printed (using Jupyter Notebook to see the entire output instead of our GUI)**

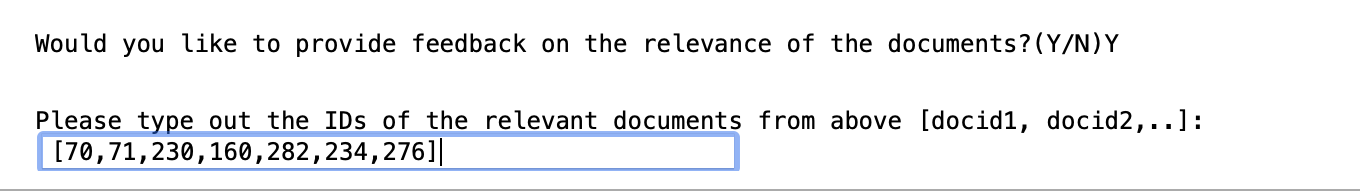
Entering the query

****

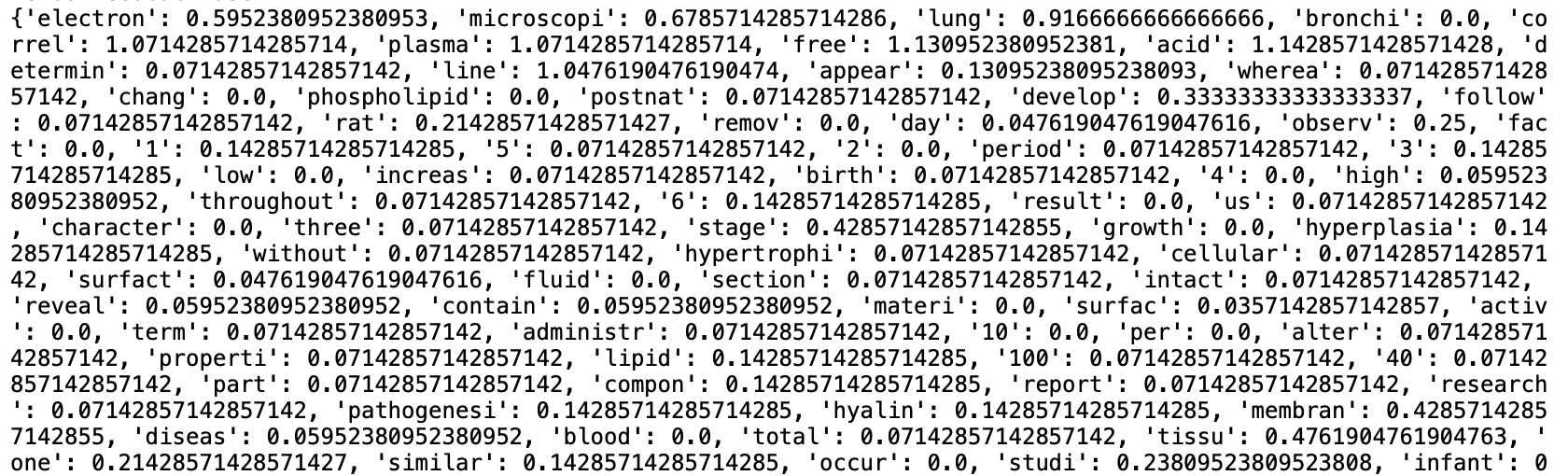
The user then gets the top 10 retrieved documents for the above query

****

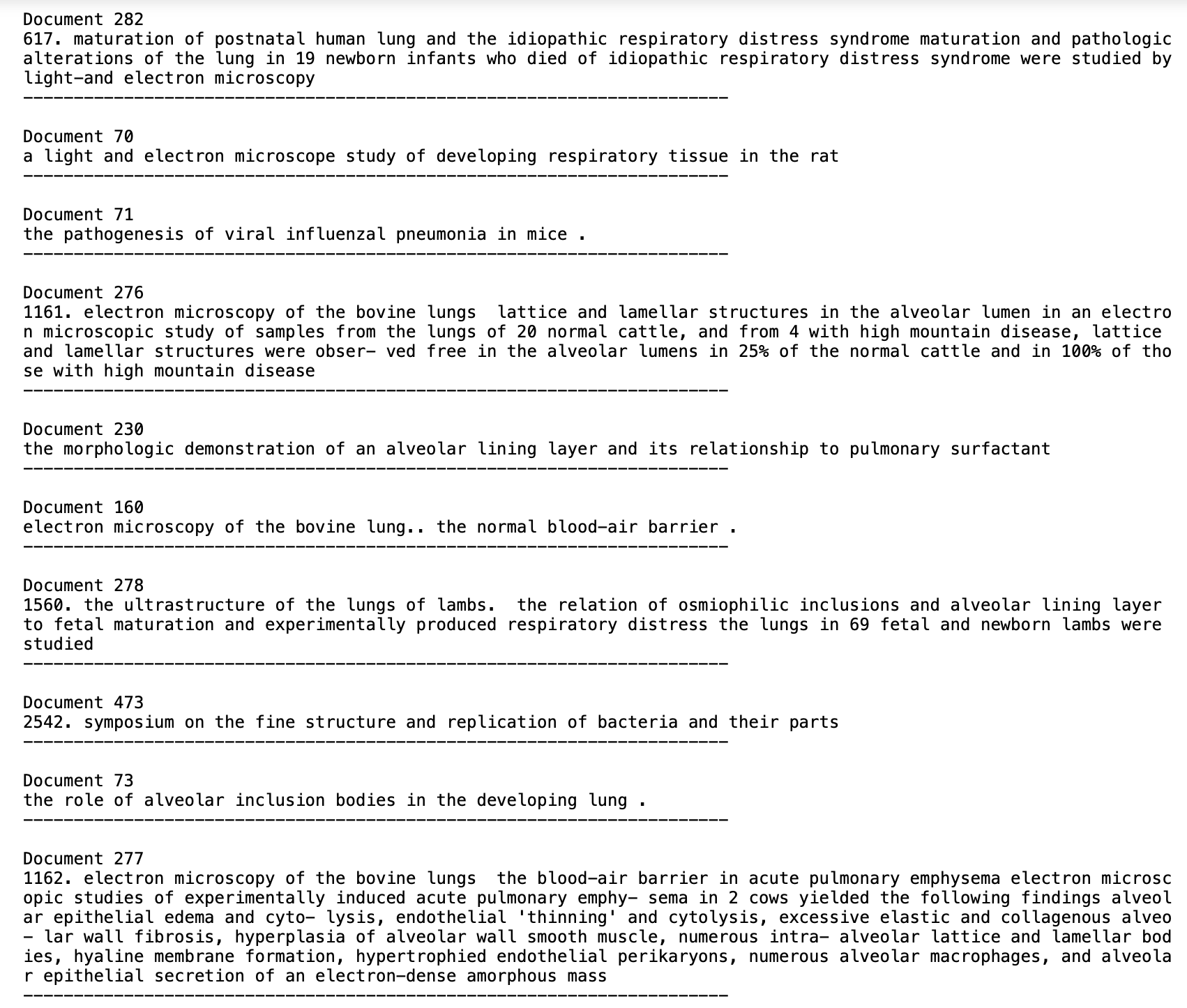
Following this, the user is asked if they would like to give feedback

****

This is what our modified query term weights look like after Rocchio method: (part of the dictionary)

****

Then the list of the top 10 retrieved documents for the Rocchio altered query is printed

****

**Retrieved docs: Before Rochio** 70,71,407,230,160,282,286,234,275,276

**No of Relevant Docs: (7)**

**Retrieved Docs After Rochio :** 282,70,71,276,230,160,278,473,73,277

**No of Relevant Docs: (8)**

**Relevant docs provided from the TestCollection (MED.REL)**

[59,62,67,69,70,71,73,78,81,160,163,230,231,232,233,234,276,277,279,282,283,287]

**After Rochio, the Relevant retrieved docs are increased from 7 to 8.**

**INVERTED INDEX**

****

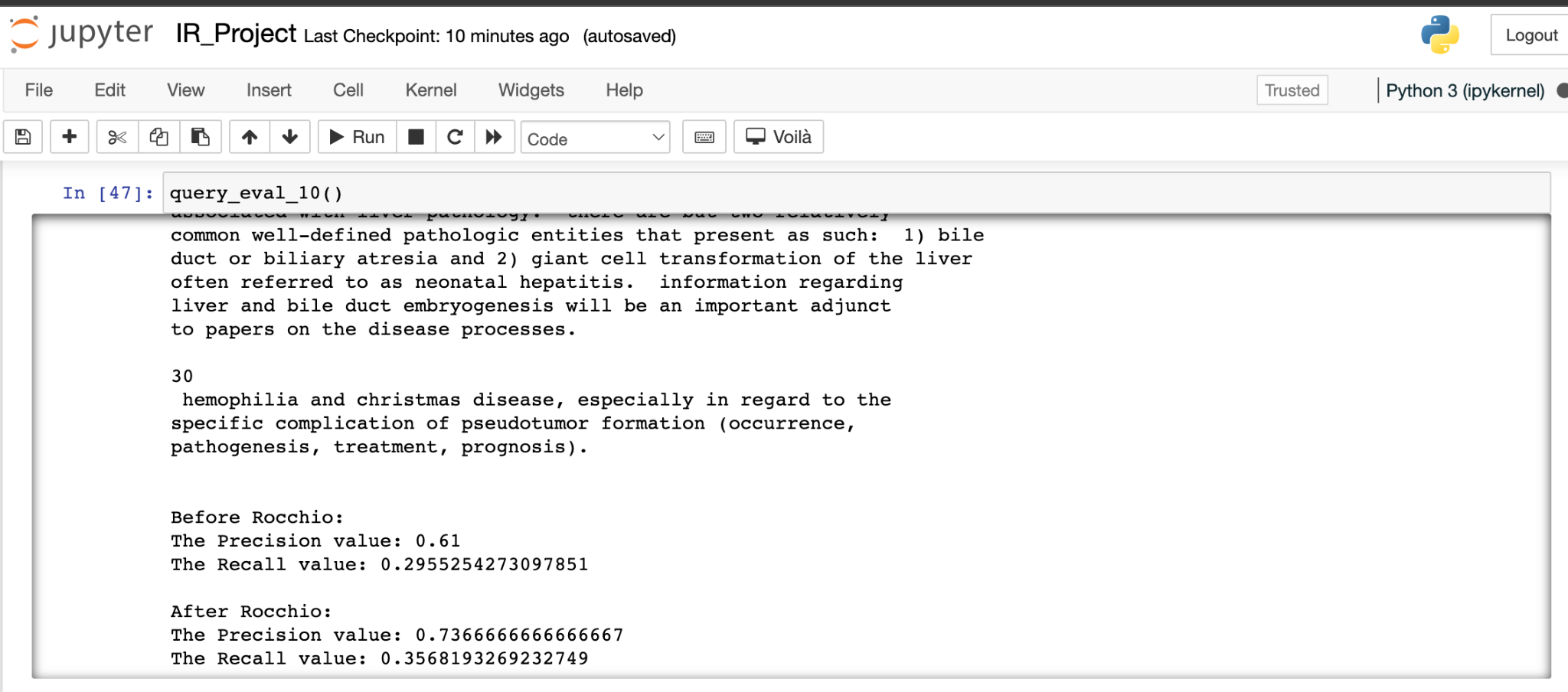
**DOCUMENT LENGTHS**



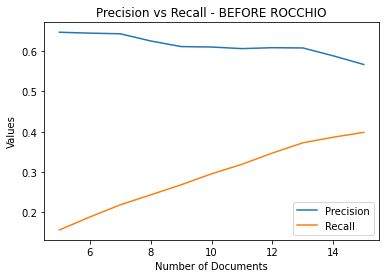
**EVALUATION OF THE SYSTEM:**

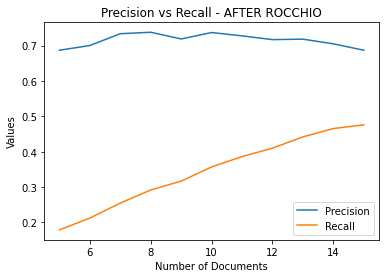
For evaluating our system, we used the MEDLINE which is a collection of articles from a medical journal to test the system (It can be found in the link<https://ir.dcs.gla.ac.uk/resources/test_collections/medl/> ). We used the MED.ALL file which contains all the documents. This dataset has 1033 documents which goes through the crawler and undergoes preprocessing. Then the inverted index and document lengths are calculated for this set of documents and stored in the local directory as explained above. Then these two files are retrieved whenever we would like to run the system now.

We run test queries which are also available as part of the dataset (<https://ir.dcs.gla.ac.uk/resources/test_collections/medl/> ). This has 30 test queries and list of the relevant documents for each query. Using the function **query\_eval\_10()**, the system is tested on its precision and recall by running through all 30 queries and calculating the precision for each query and recall for each query using the function **precision\_recall(retrievedDocs, relevantDocs).** The set of 10 retrieved document ids and the relevant document ids are passed to this function from query\_eval\_10 function. The query is then printed for our reference. The precision and recall for all the queries are calculated and the average precision and recall is returned. For this set of 30 queries, the average precision and recall was **0.61** and **0.295** which is pretty low. We also calculated in the same function with the same process as above, the precision and recall of these 30 queries after Rocchio relevance feedback was applied. Then the average precision and recall was calculated and the values are **0.737** and **0.357** respectively. The system definitely improved on doing relevance feedback.



Apart from this, we also wanted to check what the ideal number of documents to retrieve i.e., the number of documents retrieved for which we get the highest possible precision and recall - the perfect tradeoff. So, using the function **precision\_Recall\_plot()**, we ran through the k values 5 to 15 where k is the number of retrieved documents. Then the average precision and recall for each k value for all queries are calculated and plotted to compare. We decided to stick with 10 as that retrieved the best results with respect to precision but still not too low on recall. There are two plots created here, one comparing precision and recall values before Rocchio and one comparing the values after Rocchio.





**REFERENCES:**

1. Link for image <https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.searchenginejournal.com%2Finformation-retrieval-seo%2F464164%2F&psig=AOvVaw3G8cRgbByrNfODkU84_EHR&ust=1678752790422000&source=images&cd=vfe&ved=0CBAQjhxqFwoTCMj61c_P1_0CFQAAAAAdAAAAABAE>

2. Links useful when creating the system

<https://medium.com/@janujaishree94/searchit-an-information-retrieval-system-33d2af956da4>

Lecture 4 Notes - Implementation notes on Vector Space Retrieval

MEDLINE journals data - <https://ir.dcs.gla.ac.uk/resources/test_collections/medl/>

<https://www.geeksforgeeks.org/python-stemming-words-with-nltk/>

<https://www.geeksforgeeks.org/python-remove-punctuation-from-string/>

<https://www.geeksforgeeks.org/reading-and-writing-json-to-a-file-in-python/>

<https://docs.python.org/3/library/tkinter.html>