INFORMATION RETRIEVAL(ASSIGNMENT-2)

GROUP MEMBERS :

Ankit Talreja Sahitya (MT22012)

Ashutosh Choubey (MT22020)

Arohi Shrivastava (MT22017)

LIBRARIES USED :

- ✓ os : os is used for importing dataset folders from directory.
- ✓ NLTK : The Natural Language Toolkit (nltk) is used for performing tasks such as: Tokenization, Part-of-speech tagging, Sentiment Analysis,Stemming and Lemmatization etc.
- ✓ re : Provides operations of detecting patterns with the help of regular expressions.
- ✓ json : Using json we can easily convert between Python objects and JSON data, making it easier to exchange data between applications written in different programming languages. The json module provides a method for encoding Python objects into JSON format: json.dumps() used to return a string representation of a JSON-encoded object.
- ✓ string : For working with text data present in dataset files.
- ✓ numpy : It offers functions for mathematical operations like linear algebra, Fourier transforms, and random number generation, as well as efficient numerical operations on multidimensional arrays and matrices.
- ✓ joblib : provides a channel for lightening.
- ✓ pandas : It offers capabilities for cleaning, combining, and reshaping data as well as data structures for effectively storing and handling huge datasets.
- ✓ BeautifulSoup : It is used for web scraping and parsing HTML and XML documents, providing a convenient and flexible way to extract and manipulate data from web pages.
- ✓ operator : It is used for providing mathematical operations.

PREPROCESSING:

DATA UPLOADING : Import files from the device.

DATA UNDERSTANDING:

1. Read data using os.listdir("file location").
2. Checked the total count of files in the folder. The number of files in the folder is 1400.

EXTRACTION OF TEXT FROM THE FIRST 5 FILES:

We retrieved the data between the title and text and concatenated together to make a new file and saved it.

PRE-PROCESSING STEPS ON THE DATASET:

1. Converted the complete text in all the files to lowercase using the .lower() function.
2. Then performed tokenization using the .word_tokenize() method.
3. Removing stopwords from the tokenized text file using stopwords module.
4. Used re.sub() to remove the punctuation marks from the text files.
5. Then used re.sub() to remove the blank spaces from all the files in the folder

Ans 1.1 : TF-IDF Matrix :

METHODOLOGY :

The TF-IDF matrix is a tool that helps quantify the importance of words in a set of texts by assigning them numerical values. To create this matrix, several steps are taken.

First, the texts are preprocessed and tokenized, which means they are broken down into individual words and cleaned up.

Next, the Term Frequency (TF) is calculated for each word in each document. This means that for each word, the number of times it appears in a particular document is counted.

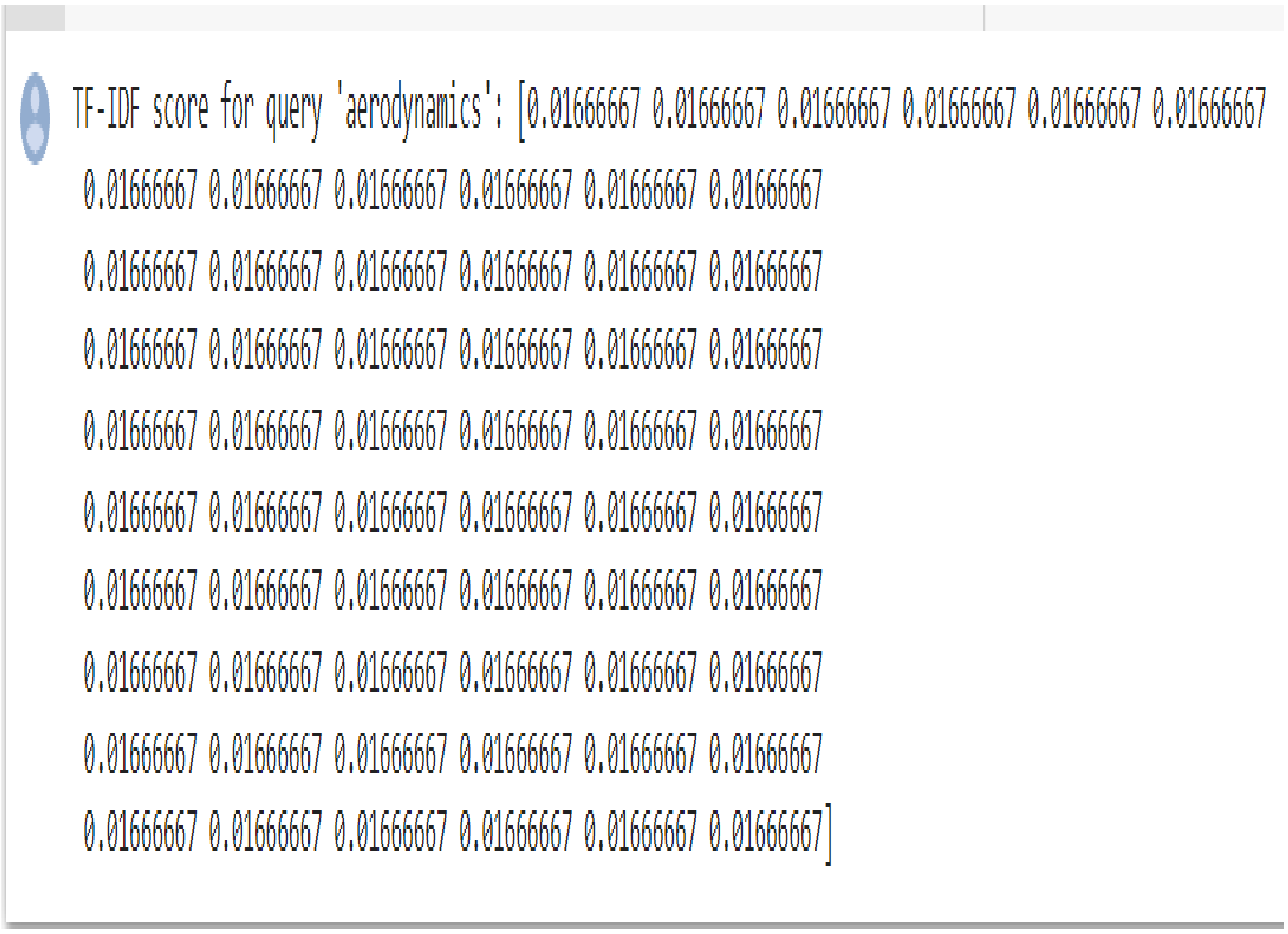
Then, a list of unique words is created using the findUniqueWords function, and these words are stored in a dictionary called uniqueWordsDict.

Finally, the Term Frequency is weighted using one of five different schemes: Binary, Raw Count, Term Frequency, Log Normalization, and Double Normalization. Each of these schemes gives a different weight to each word, depending on how frequently it appears in the document and how common it is across all documents.

The Term Frequency is calculated using 5 different weighting schemes,

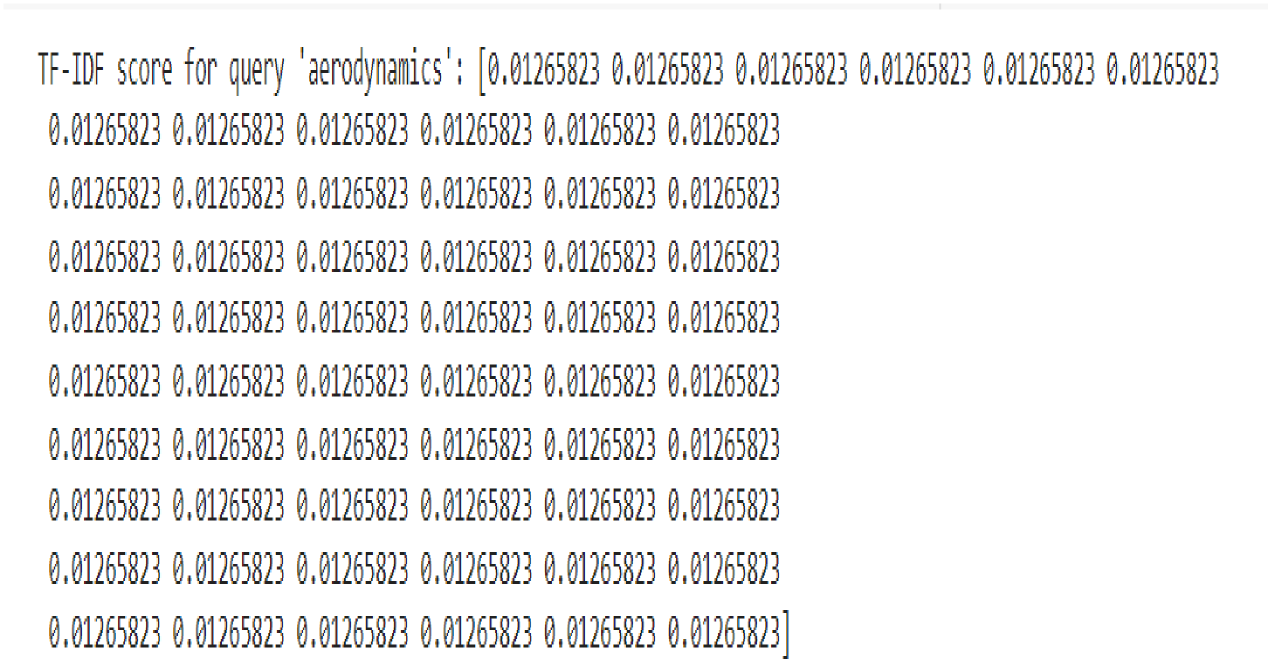
Weighting Scheme	TF Weight
Binary	0,1
Raw count	f(t,d)
Term frequency	f(t,d)/Pf(t‘, d)
Log normalization	log (1+f(t,d))
Double normalization	0.5+0.5*(f(t,d)/ max(f(t‘,d)

1.) For Binary :



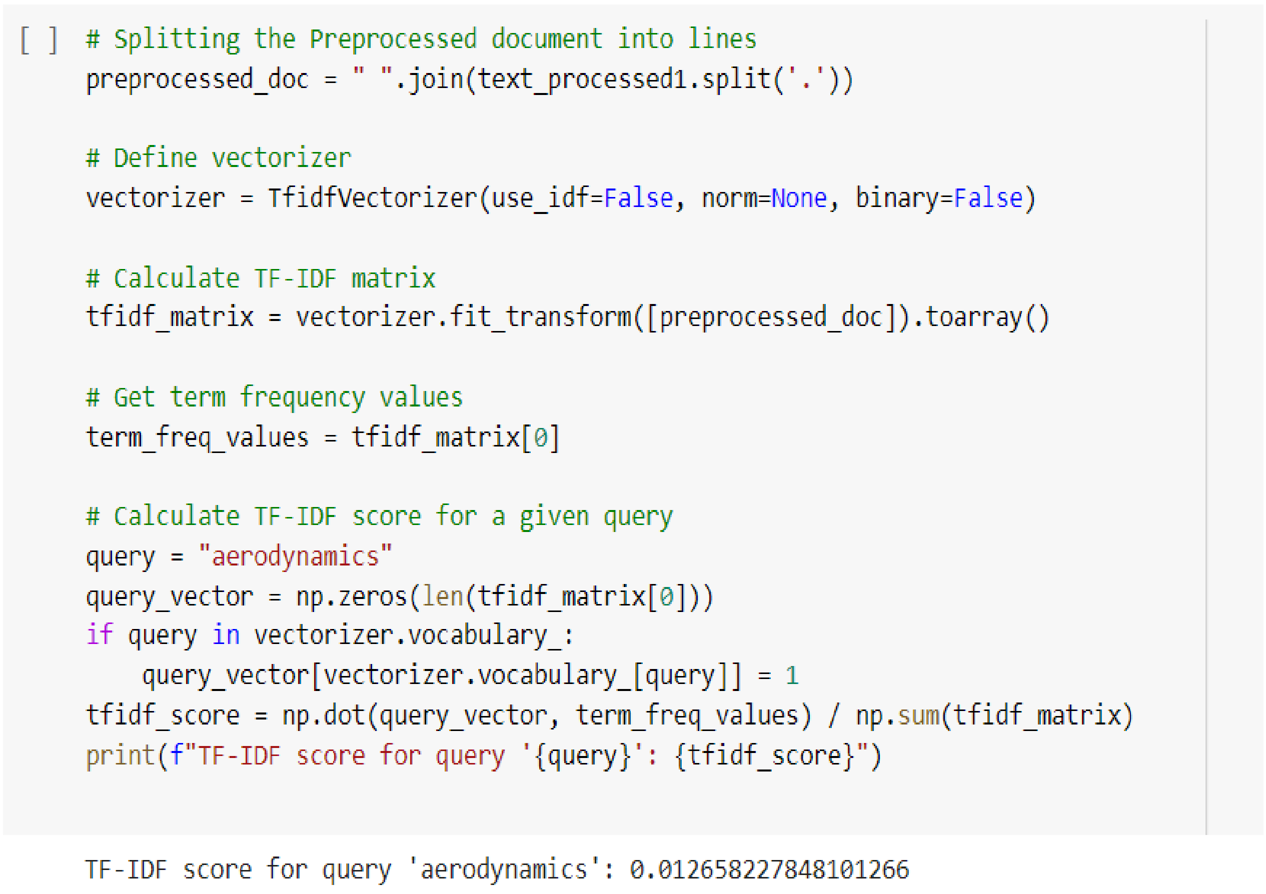
```
TF-IDF score for query 'aerodynamics': [0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667
 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667 0.01666667]
```

2.) raw count :



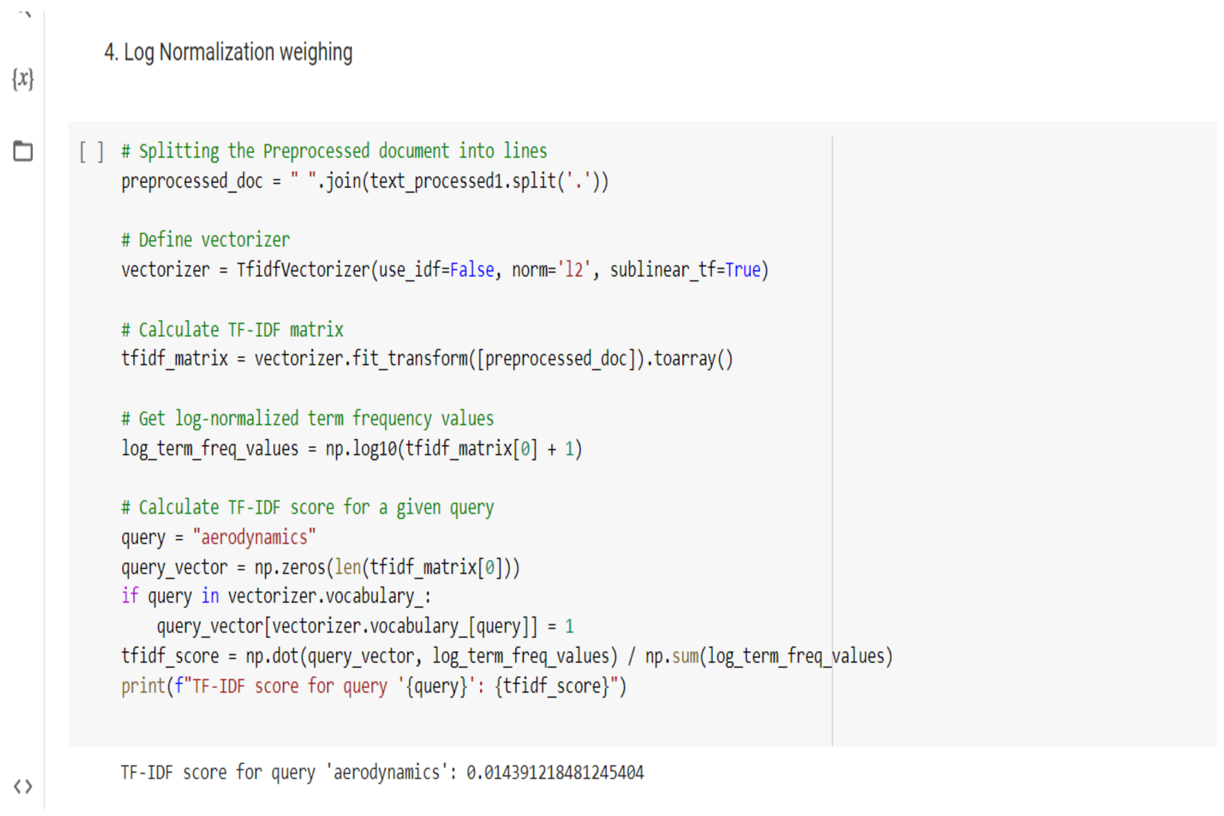
```
TF-IDF score for query 'aerodynamics': [0.01265823 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823
 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823
 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823
 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823
 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823
 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823
 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823
 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823
 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823
 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823 0.01265823]
```

3.) Term frequency :



```
[ ]  # Splitting the Preprocessed document into lines
     preprocessed_doc = " ".join(text_processed1.split('.'))

     # Define vectorizer
     vectorizer = TfidfVectorizer(use_idf=False, norm=None, binary=False)

     # Calculate TF-IDF matrix
     tfidf_matrix = vectorizer.fit_transform([preprocessed_doc]).toarray()

     # Get term frequency values
     term_freq_values = tfidf_matrix[0]

     # Calculate TF-IDF score for a given query
     query = "aerodynamics"
     query_vector = np.zeros(len(tfidf_matrix[0]))
     if query in vectorizer.vocabulary_:
         query_vector[vectorizer.vocabulary_[query]] = 1
     tfidf_score = np.dot(query_vector, term_freq_values) / np.sum(tfidf_matrix)
     print(f"TF-IDF score for query '{query}': {tfidf_score}")

     TF-IDF score for query 'aerodynamics': 0.012658227848101266
```

4.) Log normalization log:

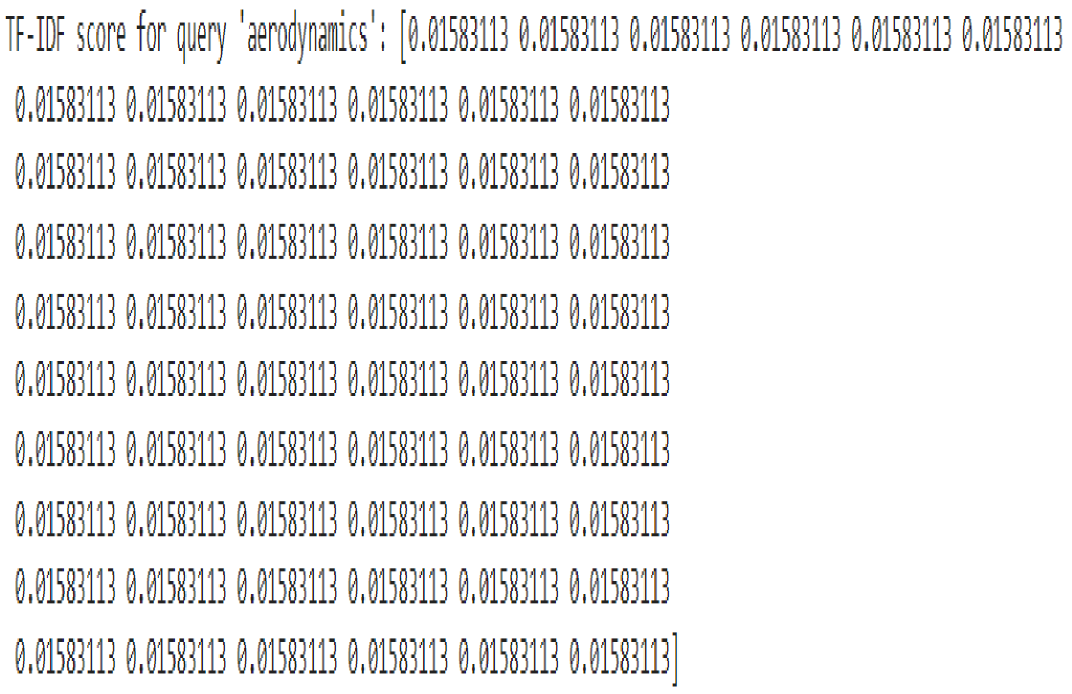


4. Log Normalization weighing

```
# Splitting the Preprocessed document into lines
preprocessed_doc = " ".join(text_processed1.split('.'))

# Define vectorizer
vectorizer = TfidfVectorizer(use_idf=False, norm='l2', sublinear_tf=True)

# Calculate TF-IDF matrix
tfidf_matrix = vectorizer.fit_transform([preprocessed_doc]).toarray()

# Get log-normalized term frequency values
log_term_freq_values = np.log10(tfidf_matrix[0] + 1)

# Calculate TF-IDF score for a given query
query = "aerodynamics"
query_vector = np.zeros(len(tfidf_matrix[0]))
if query in vectorizer.vocabulary_:
    query_vector[vectorizer.vocabulary_[query]] = 1
tfidf_score = np.dot(query_vector, log_term_freq_values) / np.sum(log_term_freq_values)
print(f"TF-IDF score for query '{query}': {tfidf_score}")
```

```
TF-IDF score for query 'aerodynamics': 0.014391218481245404
```

5.) Double Normalization :



```
TF-IDF score for query 'aerodynamics': [0.01583113 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113
 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113
 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113
 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113
 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113
 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113
 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113
 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113
 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113
 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113 0.01583113]
```

pros and cons of using each weighting scheme:

1. Binary Weighting Scheme :
Pros :

- ✓ ● It is simple and easy to implement.
- ✓ ● It is useful for that type of applications where only presence or absence of terms matters, for example text classification.

Cons :

- ✓ It does not take into account how often the term appears in the document, which can lead to the loss of important data.
- ✓ It ignores the term's frequency in the corpus, which could result in an excess of rare terms.

2. Raw Count Weighting Scheme :
Pros :

- ✓ It is simple / easy to implement and understand.
- ✓ It is more informative than the binary weighting method since it takes the frequency of the term in the document into account.

Cons :

- ✓ The length of the paper, which could influence its ranking, is not taken into consideration.
- ✓ It ignores the term's frequency in the corpus, which could result in an excess of common terms.

3. Term Frequency Weighting Scheme :
Pros :

- ✓ It is more informative than the binary and raw count weighting systems since it takes the frequency of the term in the document into account.
- ✓ It is widely used in information retrieval systems.

Cons :

- ✓ The length of the paper, which could influence its ranking, is not taken into consideration.
- ✓ It ignores the term's frequency in the corpus, which could result in an over-representation of common terms.

4. Log Normalization Weighting Scheme :

Pros :

- ✓ It minimizes the influence of high frequency terms, which may impact the document's ranking.
- ✓ It is widely used in information retrieval systems.

Cons :

- ✓ Log Normalization Weighting Scheme not suitable for short/small documents.
- ✓ It ignores the term's frequency in the corpus, which could result in an over-representation of common terms.

5. Double normalization :

Pros :

- ✓ It is more informative than the previous weighting systems since it takes into account the frequency of the word in the corpus and the document.
- ✓ It is widely used in information retrieval systems.

Cons :

- ✓ More complex than other weighting schemes .
- ✓ Harder to implement.

Answer 1.2 Jaccard Coefficient :

Methodology :

- ❖ Identify the two sets for which you want to calculate the Jaccard coefficient.

- ❖ Define a function called "intersection" that takes in two sets and returns a new set that contains only the elements that are present in both sets.

- ❖ Define a function called "union" that takes in two sets and returns a new set that contains all the unique elements from both sets.

- ❖ Call the "intersection" function on the two sets you identified in step 1, and save the result as a new set.

- ❖ Call the "union" function on the two sets you identified in step 1, and save the result as a new set.

- ❖ Find the size of the intersection set (i.e., the number of elements it contains).

- ❖ Find the size of the union set (i.e., the number of elements it contains).

- ❖ Divide the size of the intersection set by the size of the union set to get the Jaccard coefficient. This value represents the similarity between the two sets.
- ❖ Formula for calculating the jaccard coefficient :

$J(l1, l2) = |l1 \cap l2| / |l1 \cup l2|$ l1 and l2 are (doc and query)

Answer -2)

PRE-PROCESSING STEPS ON THE DATASET:
1. Reading the csv format dataset
2. Removing unnecessary columns
3. Text cleaning by :
   - ❖ Converted the complete text in all the files to lowercase using the .lower() function.
   - ❖ Then performed tokenization using the .word_tokenize() method.
   - ❖ Removing stopwords from the tokenized text file using stopwords module.
4. performing stemming and lemmatization
5. Apply the clean_tokenize_text function to the 'Text' column

TF-ICF weighting scheme is a method of calculating the importance of a term in a corpus of documents. It combines two metrics, term frequency (TF) and inverse corpus frequency (ICF), to assign weights to each term in a document. The process involves:

- ✓ Creating a list of words from the documents
- ✓ Converting the list of words into a string format suitable for the TfidfVectorizer
- ✓ Generating a term-document matrix using TfidfVectorizer and computing the TF-IDF matrix using the formula Tfidf_matrix = **tfidf_transformer.fit_transform(term_doc_matrix).toarray()**

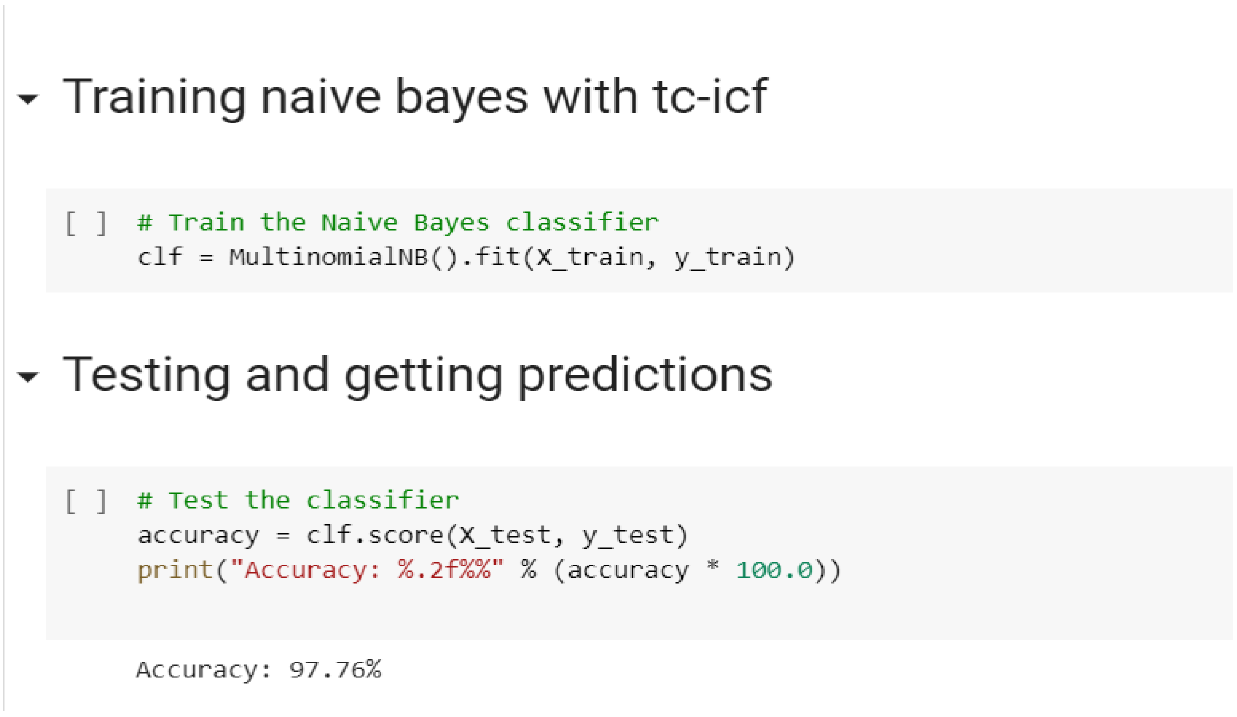
- ✓ Calculating the ICF values for each term in the corpus using icf_values = **np.log(num_docs / np.count_nonzero(tfidf_matrix, axis=0))**

- ✓ Multiplying the TF and ICF values for each term to get the final TF-ICF weight
- ✓ Converting the TF-IDF matrix and ICF values to sparse matrices for efficient storage and computation

To split the dataset into training and testing sets, we use the train_test_split function from scikit-learn library. This function randomly divides the data into two sets: a training set and a testing set. Here's how it works:

- ✓ We start with a feature matrix, tf_icf_matrix, containing the TF-ICF weights for each term in each document.
- ✓ We also have a target variable, data['Category'], which contains the category labels for each document.
- ✓ We specify the test_size parameter to determine the proportion of data that will be allocated to the testing set. For example, if we set test_size=0.3, then 30% of the data will be used for testing, and 70% will be used for training.
- ✓ We also specify a random state, which ensures that the data is split in the same way each time we run the code.
- ✓ The train_test_split function then returns four objects: X_train, X_test, y_train, and y_test.
- ✓ X_train and X_test are matrices of training and testing features, respectively. They contain the TF-ICF weights for each term in each document in the training and testing sets.
- ✓ y_train and y_test are vectors of training and testing target values, respectively. They contain the category labels for each document in the training and testing sets.

Training the Naive Bayes classifier with TF-ICF :



Training naive bayes with tc-icf

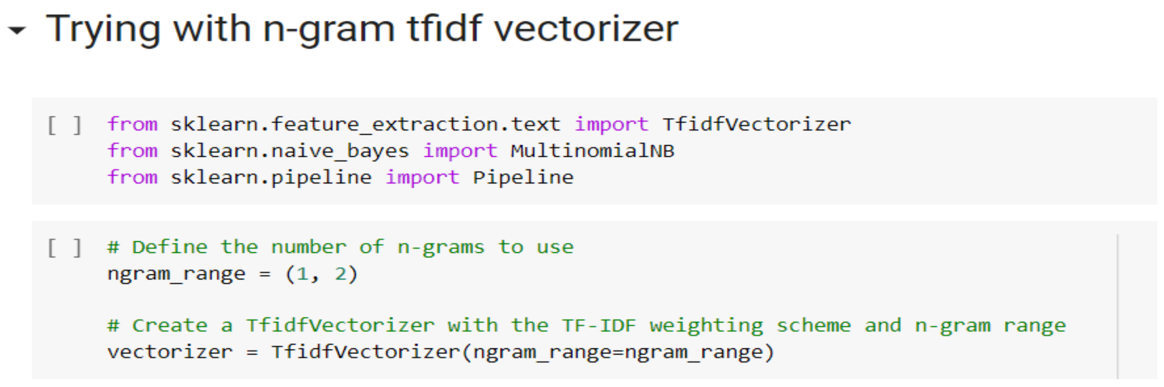
```
[ ]  # Train the Naive Bayes classifier
     clf = MultinomialNB().fit(X_train, y_train)
```

Testing and getting predictions

```
[ ]  # Test the classifier
     accuracy = clf.score(X_test, y_test)
     print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

```
Accuracy: 97.76%
```

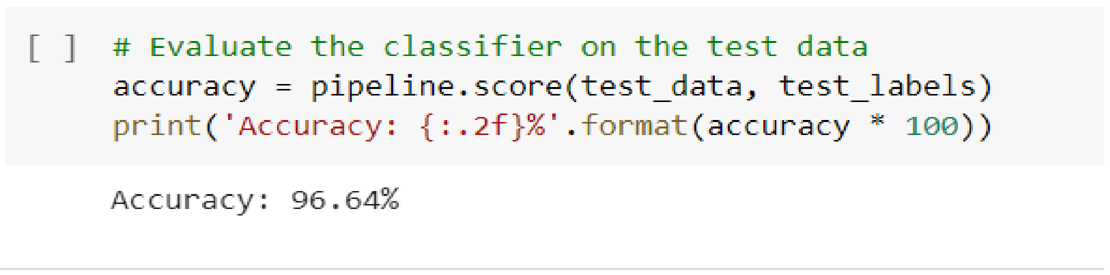
Testing the Naive Bayes classifier with TF-ICF :



Trying with n-gram tfidf vectorizer

```
[ ]  from sklearn.feature_extraction.text import TfidfVectorizer
     from sklearn.naive_bayes import MultinomialNB
     from sklearn.pipeline import Pipeline
```

```
[ ]  # Define the number of n-grams to use
     ngram_range = (1, 2)

     # Create a TfidfVectorizer with the TF-IDF weighting scheme and n-gram range
     vectorizer = TfidfVectorizer(ngram_range=ngram_range)
```



```
[ ]  # Evaluate the classifier on the test data
     accuracy = pipeline.score(test_data, test_labels)
     print('Accuracy: {:.2f}%'.format(accuracy * 100))

     Accuracy: 96.64%
```

Conclusion : we got the following accuracy as 96.64%

Answer 3)

1.) first of all, reading a file called "Dataset_Question3.txt" using the pandas library in Python and assigning it to a variable called "df". Overall loading a text file into a pandas dataframe.

2.) Queries that have qid as 4 are considered using the following code

```
df=pd.read_csv('Dataset_Question3.txt',sep=" ",header=None)

filedatabase={}
for iterator in range(0,len(df.index)):
    check1=df.at[iterator,1]
    if ("qid:4"==check1):
        filedatabase[iterator]=df.at[iterator,0]
print("File database created successfully")
```

3.) storing all unique relevance judgment labels in relevanceJudgementLabel

4. Calculating final sorted dataframe and save to CSV: Totalfiles:

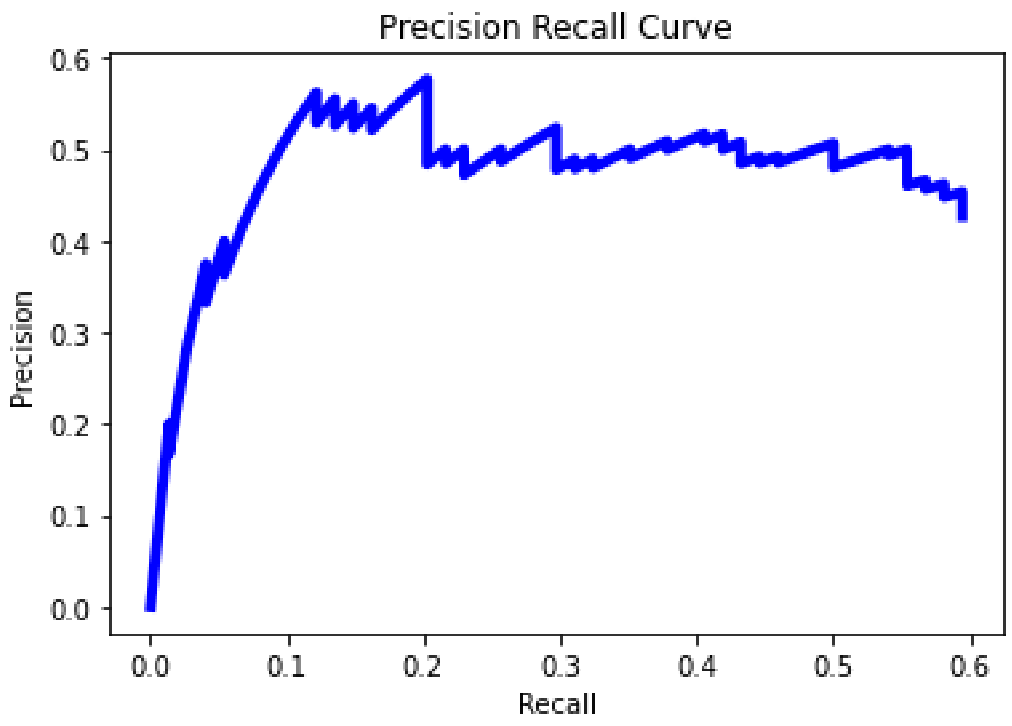
```
19893497375938370599826047614905329896936840170566570588205180
31270485799269519348241268656543105024000000000000000000000
```

5.) Now using function DFG() to calculate nDCG of 50 data and to calculate nDCG of the whole dataset.

```
nDCG at 50:                0.3521042740324887
nDCG for whole Dataset:    0.5979226516897831
```

6.) Sort the relevance labels based on the feature_75 scores in descending order, calculating precision and recall for each document in the sorted list.

7.) Plotting the precision and recall curve.


Precision Recall Curve
Precision
Recall