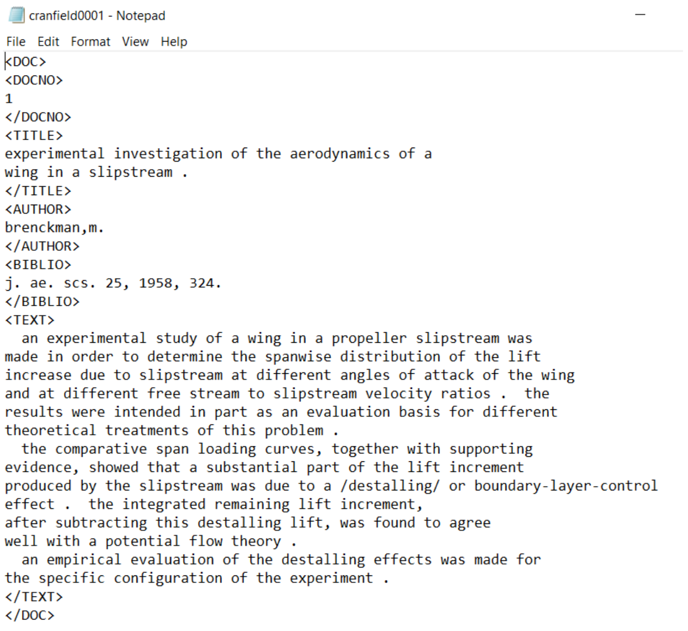
Group Number: 104
Team Members:
1) Akshita Gupta - 2020491
2) Shivam Agrawal - 2020124
3) Yash Kumar - 2020350

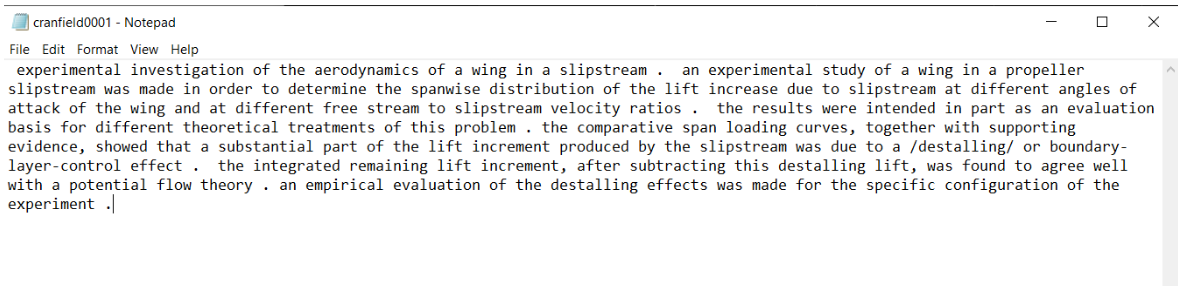
IR Assignment 2

Q1)
Ans1)
For this assignment, we were given a dataset of 1400 documents containing basic HTML format with different tags.



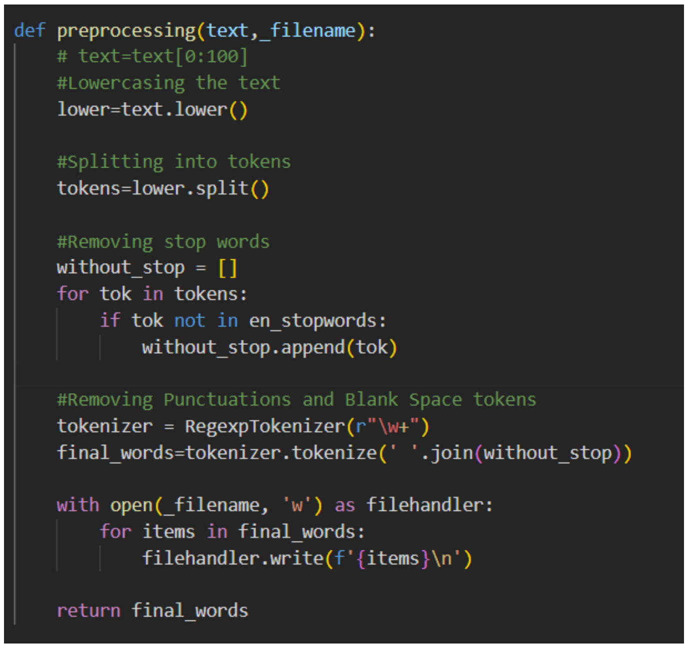
```
<DOC>
<DOCNO>
1
</DOCNO>
<TITLE>
experimental investigation of the aerodynamics of a
wing in a slipstream .
</TITLE>
<AUTHOR>
brenckman,m.
</AUTHOR>
<BIBLIO>
j. ae. scs. 25, 1958, 324.
</BIBLIO>
<TEXT>
  an experimental study of a wing in a propeller slipstream was
made in order to determine the spanwise distribution of the lift
increase due to slipstream at different angles of attack of the wing
and at different free stream to slipstream velocity ratios .  the
results were intended in part as an evaluation basis for different
theoretical treatments of this problem .
  the comparative span loading curves, together with supporting
evidence, showed that a substantial part of the lift increment
produced by the slipstream was due to a /destalling/ or boundary-layer-control
effect .  the integrated remaining lift increment,
after subtracting this destalling lift, was found to agree
well with a potential flow theory .
  an empirical evaluation of the destalling effects was made for
the specific configuration of the experiment .
</TEXT>
</DOC>
```

From this file we extracted the content under the “TITLE” and “TEXT” tag. For this we used simple python code which is mentioned in the “_text_extraction.py” file. First we find the file names and open them in a loop and find the required data, after which we overwrite the file with the extracted data in a single line.



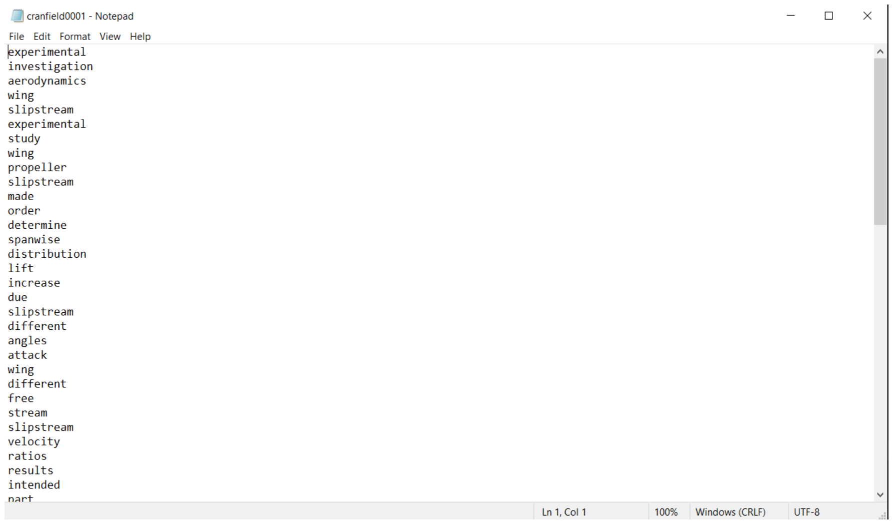
```
 experimental investigation of the aerodynamics of a wing in a slipstream .  an experimental study of a wing in a propeller
slipstream was made in order to determine the spanwise distribution of the lift increase due to slipstream at different angles of
attack of the wing and at different free stream to slipstream velocity ratios .  the results were intended in part as an evaluation
basis for different theoretical treatments of this problem . the comparative span loading curves, together with supporting
evidence, showed that a substantial part of the lift increment produced by the slipstream was due to a /destalling/ or boundary-
layer-control effect .  the integrated remaining lift increment, after subtracting this destalling lift, was found to agree well
with a potential flow theory . an empirical evaluation of the destalling effects was made for the specific configuration of the
experiment .
```

This is done for all the given 1400 files. This data is then preprocessed to remove the unnecessary words and blanks and punctuations. This is done in the file “preprocessing.ipynb”.



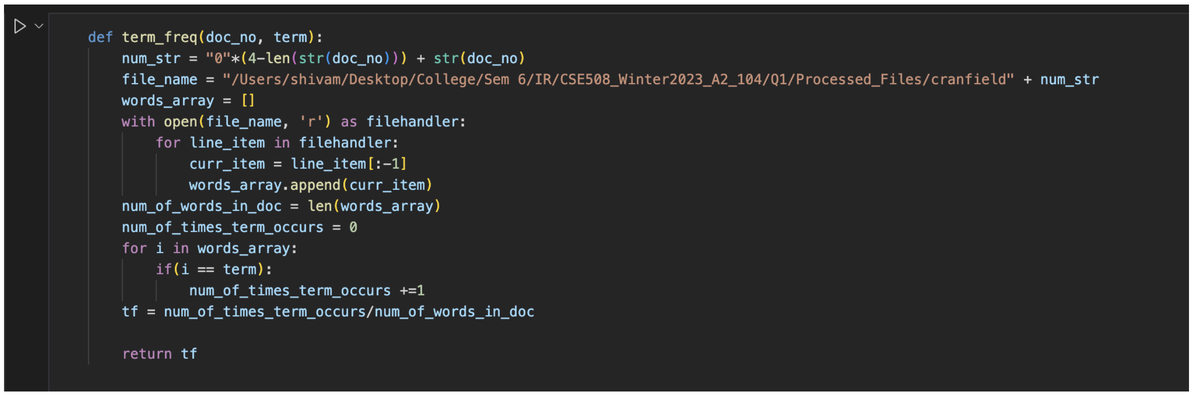
```
def preprocessing(text,_filename):
    # text=text[0:100]
    #Lowercasing the text
    lower=text.lower()

    #Splitting into tokens
    tokens=lower.split()

    #Removing stop words
    without_stop = []
    for tok in tokens:
        if tok not in en_stopwords:
            without_stop.append(tok)

    #Removing Punctuations and Blank Space tokens
    tokenizer = RegexpTokenizer(r"\w+")
    final_words=tokenizer.tokenize(' '.join(without_stop))

    with open(_filename, 'w') as filehandler:
        for items in final_words:
            filehandler.write(f'{items}\n')

    return final_words
```

Once the data is processed we store this in a new Folder by the name “Processed_Files”. This data looks like:

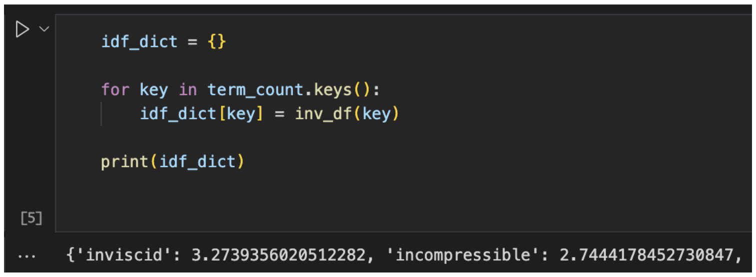


```
experimental
investigation
aerodynamics
wing
slipstream
experimental
study
wing
propeller
slipstream
made
order
determine
spanwise
distribution
lift
increase
due
slipstream
different
angles
attack
wing
different
free
stream
slipstream
velocity
ratios
results
intended
part
```

For calculating the TF-IDF matrix, we used the given 5 weighting schemes, according to each weighting scheme we created a different function for calculating the Term Frequency, and used a common IDF list.



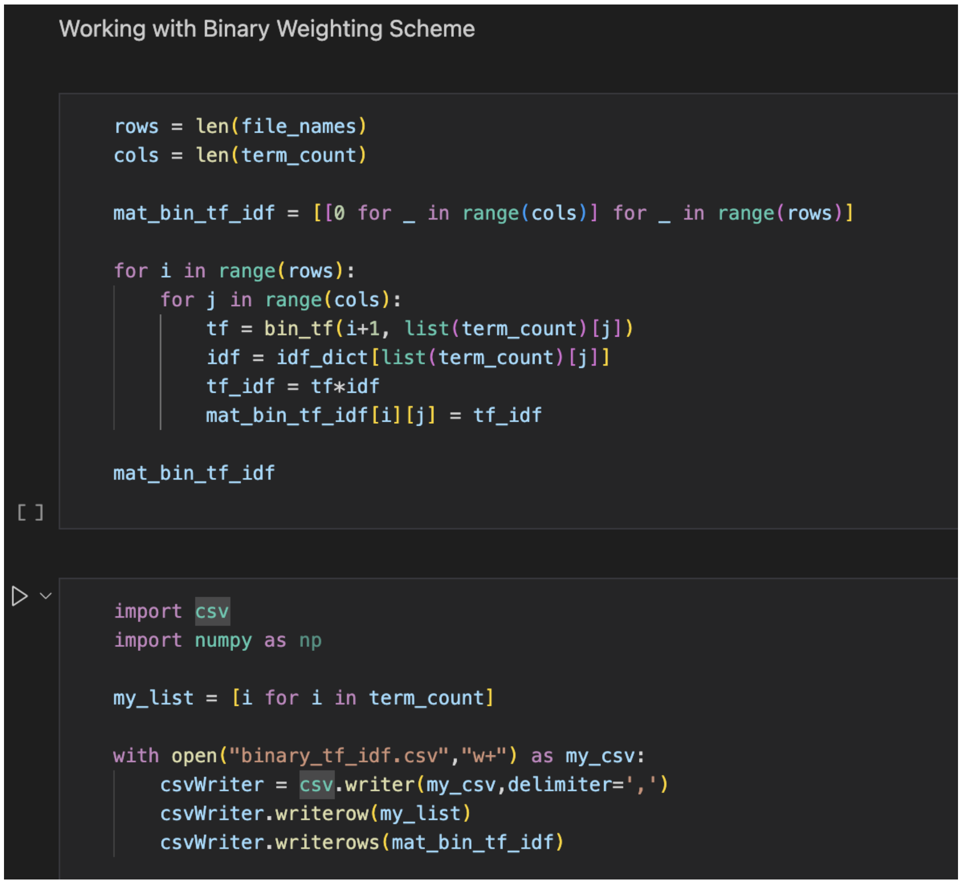
```
def term_freq(doc_no, term):
    num_str = "0"*(4-len(str(doc_no))) + str(doc_no)
    file_name = "/Users/shivam/Desktop/College/Sem 6/IR/CSE508_Winter2023_A2_104/Q1/Processed_Files/cranfield" + num_str
    words_array = []
    with open(file_name, 'r') as filehandler:
        for line_item in filehandler:
            curr_item = line_item[:-1]
            words_array.append(curr_item)
    num_of_words_in_doc = len(words_array)
    num_of_times_term_occurs = 0
    for i in words_array:
        if(i == term):
            num_of_times_term_occurs +=1
    tf = num_of_times_term_occurs/num_of_words_in_doc

    return tf
```



```
idf_dict = {}

for key in term_count.keys():
    idf_dict[key] = inv_df(key)

print(idf_dict)
```

```
{'inviscid': 3.2739356020512282, 'incompressible': 2.7444178452730847,
```

For each weighting scheme, we created a different matrix and stored them as CSV files to use later. The computation time for creating the matrix was around 15 mins for each matrix and thus we preferred saving it rather than computing again and again.

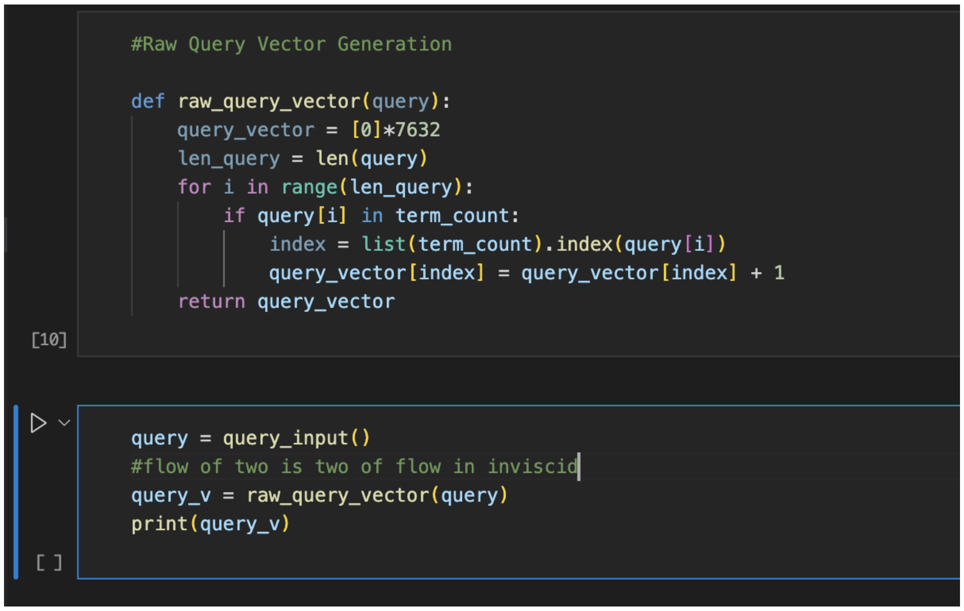


Working with Binary Weighting Scheme

```
rows = len(file_names)
cols = len(term_count)

mat_bin_tf_idf = [[0 for _ in range(cols)] for _ in range(rows)]

for i in range(rows):
    for j in range(cols):
        tf = bin_tf(i+1, list(term_count)[j])
        idf = idf_dict[list(term_count)[j]]
        tf_idf = tf*idf
        mat_bin_tf_idf[i][j] = tf_idf

mat_bin_tf_idf
```

```
import csv
import numpy as np

my_list = [i for i in term_count]

with open("binary_tf_idf.csv","w+") as my_csv:
    csvWriter = csv.writer(my_csv,delimiter=',')
    csvWriter.writerow(my_list)
    csvWriter.writerows(mat_bin_tf_idf)
```

Next we constructed the query vector for a given query according to the weighting scheme. For each weighting scheme a different query vector was generated. First the query is processed such that all stop words are removed and all are lower cases. Next we compare this with the dictionary of the terms that we used and create a query vector.



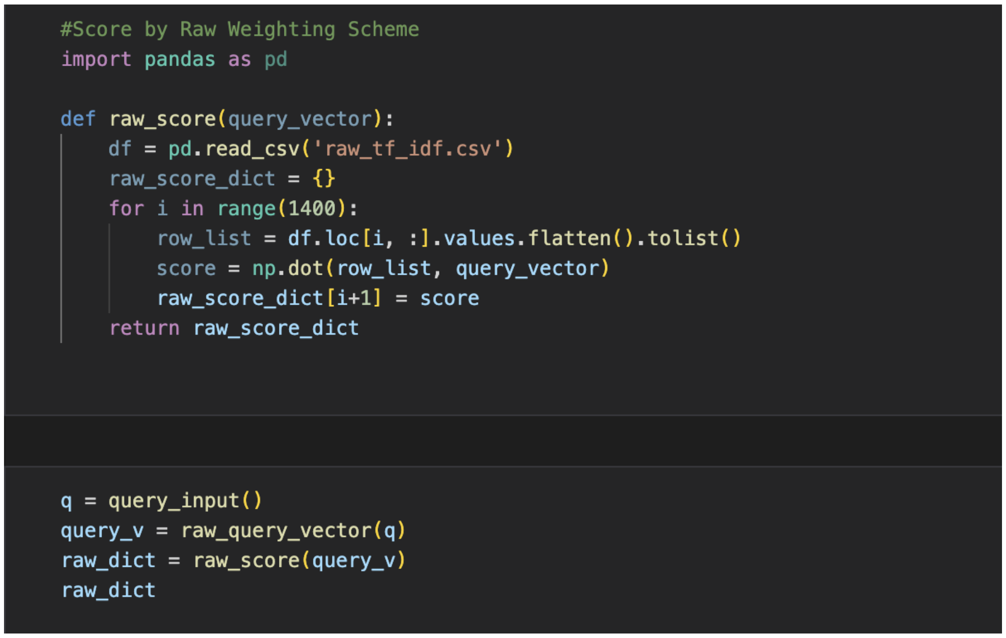
```
#Raw Query Vector Generation

def raw_query_vector(query):
    query_vector = [0]*7632
    len_query = len(query)
    for i in range(len_query):
        if query[i] in term_count:
            index = list(term_count).index(query[i])
            query_vector[index] = query_vector[index] + 1
    return query_vector
```

[10]

```
query = query_input()
#flow of two is two of flow in inviscid
query_v = raw_query_vector(query)
print(query_v)
```

[ ]

Finally we calculate the TF-IDF score of the query according to all 5 weighting schemes. This is done by taking a doc product of the query vector with each document and then storing the score in a separate dictionary.



```
#Score by Raw Weighting Scheme
import pandas as pd

def raw_score(query_vector):
    df = pd.read_csv('raw_tf_idf.csv')
    raw_score_dict = {}
    for i in range(1400):
        row_list = df.loc[i, :].values.flatten().tolist()
        score = np.dot(row_list, query_vector)
        raw_score_dict[i+1] = score
    return raw_score_dict
```

```
q = query_input()
query_v = raw_query_vector(q)
raw_dict = raw_score(query_v)
raw_dict
```

<u>Pros and Cons of each weighting scheme:</u>

<u>Binary Weighting Scheme:</u>
Pro:

1. Simple and efficient: The binary weighting scheme is easy to implement and requires minimal computation, making it a fast and efficient method.

Con:

1. Loss of information: By setting the term frequency to 1 or 0, the binary weighting scheme completely ignores the frequency of the term in the document, resulting in the loss of potentially useful information.

Raw Count Weighting Scheme:

Pro:

1. Preserves information: Unlike binary weighting scheme, the raw count weighting scheme preserves the information about the frequency of a term in a document, which can be important in many applications.

Con:

1. Sensitive to document length: Raw count weighting scheme can be sensitive to the length of the document, as longer documents are likely to have more occurrences of a term than shorter documents.

Term Frequency Weighting Scheme:

Pro:

1. Captures term importance: Term frequency weighting scheme captures the importance of a term in a document relative to the other terms in the same document.

Con:

1. Not suitable for all applications: Term frequency weighting scheme may not be suitable for some applications, such as text clustering, where terms that occur frequently across multiple documents need to be emphasized.

Log Normalization Weighting Scheme:

Pro:

1. Reduces the impact of high-frequency terms: Log normalization weighting scheme reduces the impact of very high-frequency terms, which can skew the results and reduce the accuracy of the analysis.

Con:

1. Requires more computation: Log normalization weighting scheme requires more computation than some other weighting schemes, which can be a drawback in large-scale text mining applications.

Double Normalization Weighting Scheme:

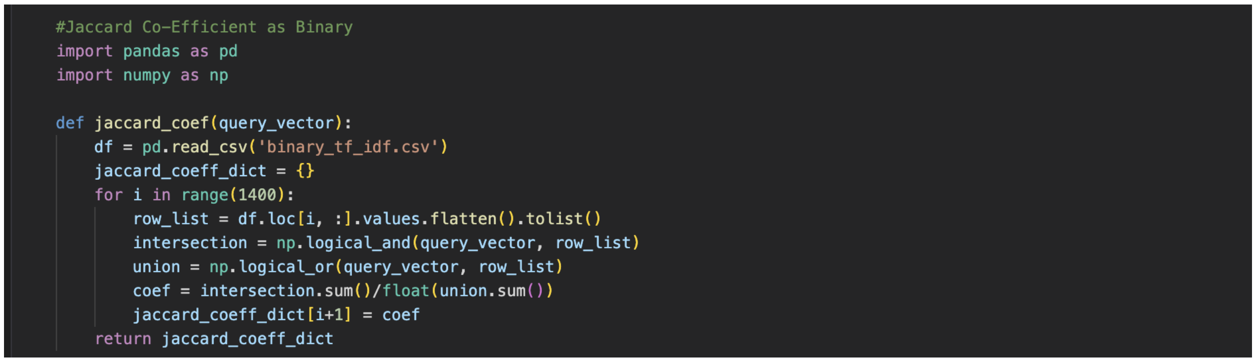
Pro:

1. Captures term importance: Double normalization weighting scheme captures the importance of a term in a document relative to the other terms in the same document, while accounting for the potential impact of high-frequency terms and differences in document length.

Con:

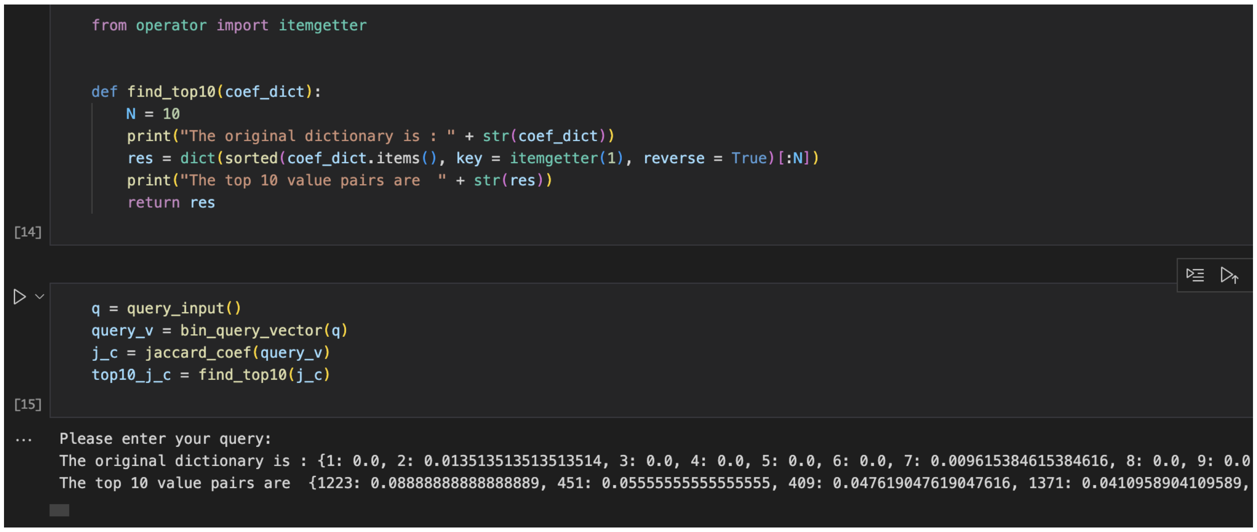
1. Not suitable for all applications: Double normalization weighting scheme may not be suitable for some applications, such as text clustering or topic modeling, where terms that occur frequently across multiple documents need to be emphasized.

Calculating Jaccard Coefficient:
For calculating Jaccard coefficient we divide the intersection of the query vector and document vector with the union of the query vector and document vector.



```python
#Jaccard Co-Efficient as Binary
import pandas as pd
import numpy as np

def jaccard_coef(query_vector):
    df = pd.read_csv('binary_tf_idf.csv')
    jaccard_coeff_dict = {}
    for i in range(1400):
        row_list = df.loc[i, :].values.flatten().tolist()
        intersection = np.logical_and(query_vector, row_list)
        union = np.logical_or(query_vector, row_list)
        coef = intersection.sum()/float(union.sum())
        jaccard_coeff_dict[i+1] = coef
    return jaccard_coeff_dict
```

The Jaccard Coefficient for all documents is stored in a dictionary and using that dictionary we find the top 10 relevant documents.



```python
from operator import itemgetter


def find_top10(coef_dict):
    N = 10
    print("The original dictionary is : " + str(coef_dict))
    res = dict(sorted(coef_dict.items(), key = itemgetter(1), reverse = True)[:N])
    print("The top 10 value pairs are  " + str(res))
    return res
```

[14]

```python
q = query_input()
query_v = bin_query_vector(q)
j_c = jaccard_coef(query_v)
top10_j_c = find_top10(j_c)
```

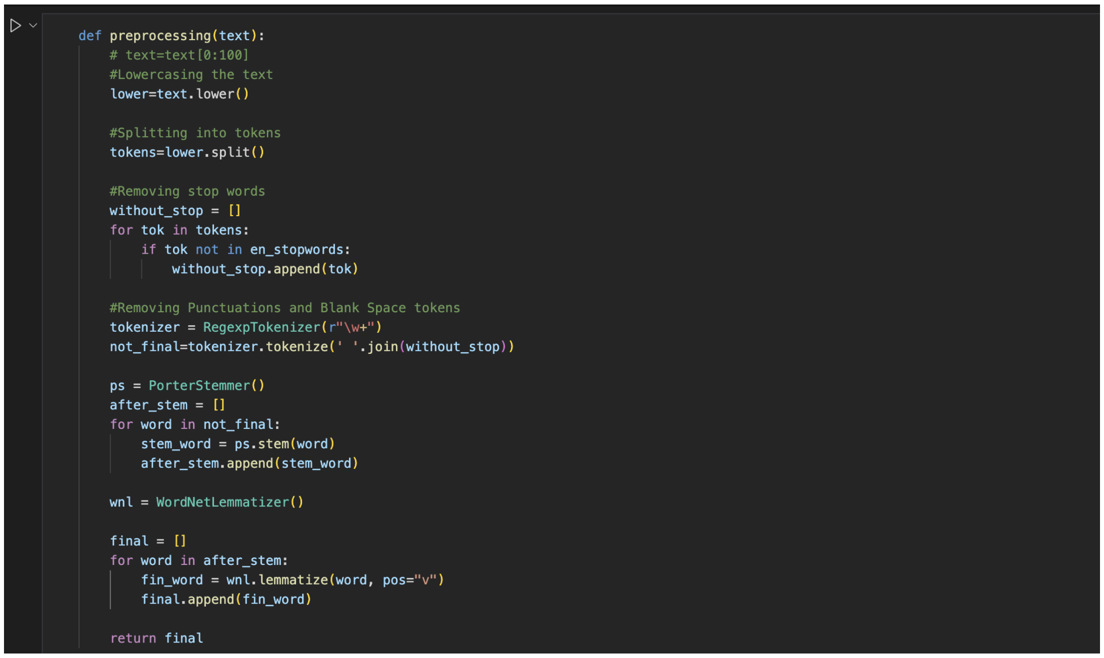
[15]

```
Please enter your query:
The original dictionary is : {1: 0.0, 2: 0.013513513513513514, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.009615384615384616, 8: 0.0, 9: 0.0
The top 10 value pairs are  {1223: 0.08888888888888889, 451: 0.05555555555555555, 409: 0.047619047619047616, 1371: 0.0410958904109589,
```

Q2)

Ans2)

For this question, we first cleaned the data by preprocessing it to remove the stop words, punctuations, and perform tokenizing, stemming and lemmatization on the text.

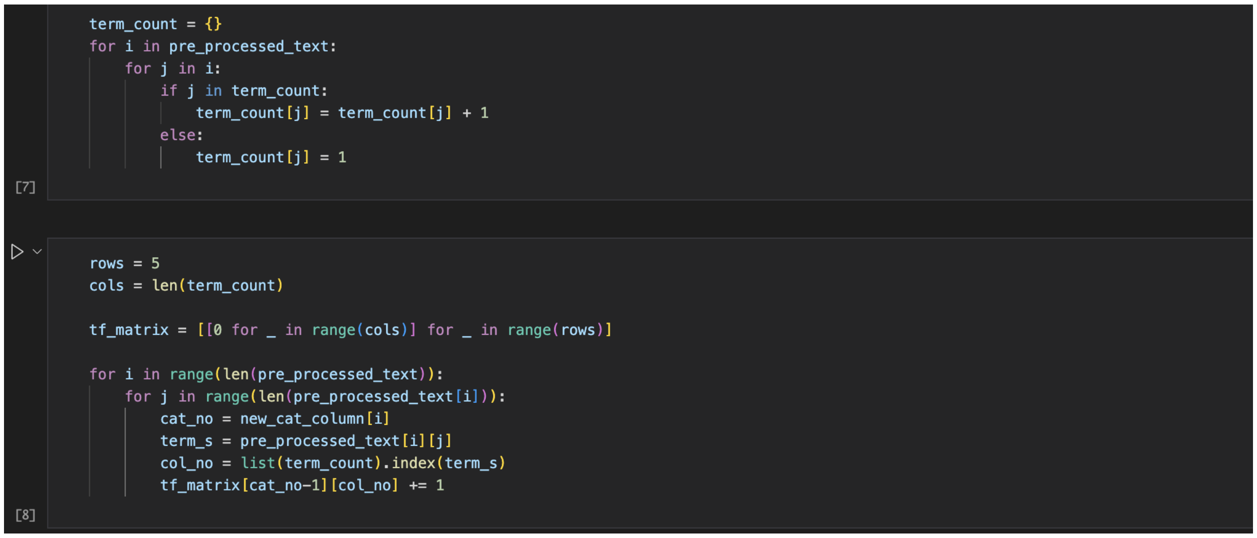


```
def preprocessing(text):
    # text=text[0:100]
    #Lowercasing the text
    lower=text.lower()

    #Splitting into tokens
    tokens=lower.split()

    #Removing stop words
    without_stop = []
    for tok in tokens:
        if tok not in en_stopwords:
            without_stop.append(tok)

    #Removing Punctuations and Blank Space tokens
    tokenizer = RegexpTokenizer(r"\w+")
    not_final=tokenizer.tokenize(' '.join(without_stop))

    ps = PorterStemmer()
    after_stem = []
    for word in not_final:
        stem_word = ps.stem(word)
        after_stem.append(stem_word)

    wnl = WordNetLemmatizer()

    final = []
    for word in after_stem:
        fin_word = wnl.lemmatize(word, pos="v")
        final.append(fin_word)

    return final
```

Once the text is pre-processed we begin to create the tf-icf matrix. For this we first find the classes from the text and allot them an integer.

```
# business = 1
# tech = 2
# politics = 3
# sport = 4
# entertainment = 5
```

After classifying the categories into integers, we find the unique terms and their occurrences, these are stored in the term_count dictionary. Now we create a term frequency matrix, where the number of rows = 5, as the number of categories are 5. The number of columns is the number of unique terms. Thus this matrix tells the term frequency for each term in each category.



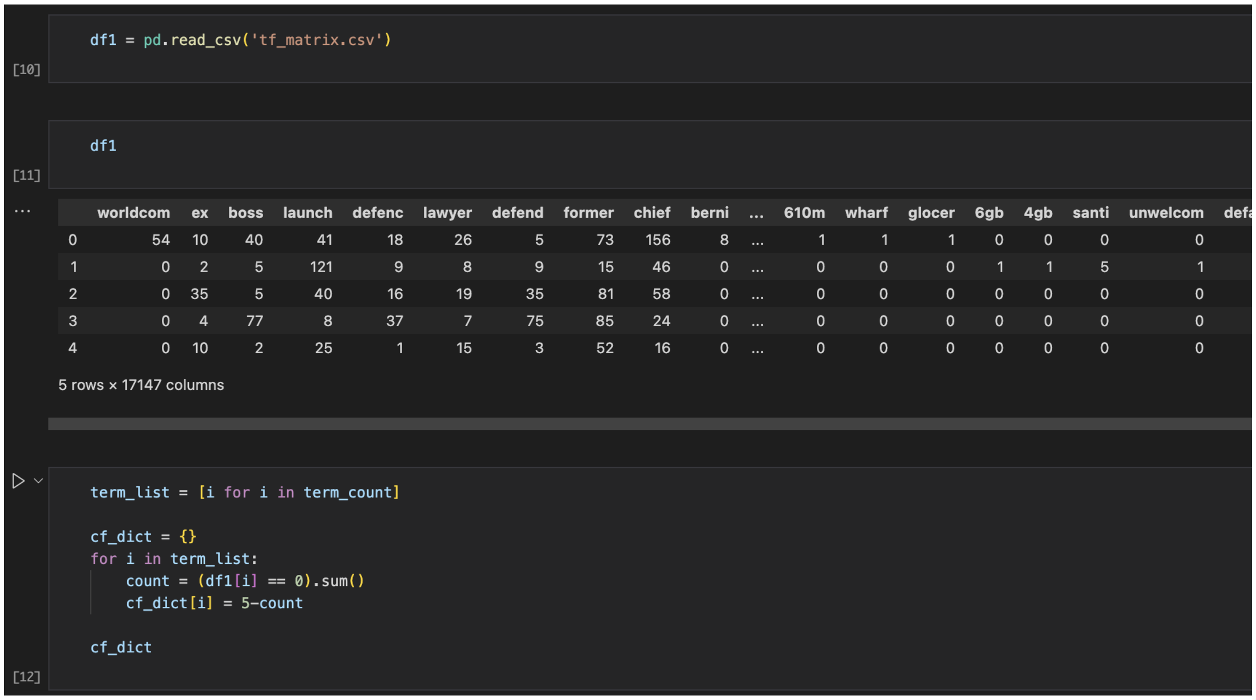
```python
term_count = {}
for i in pre_processed_text:
    for j in i:
        if j in term_count:
            term_count[j] = term_count[j] + 1
        else:
            term_count[j] = 1
```

[7]

```python
rows = 5
cols = len(term_count)

tf_matrix = [[0 for _ in range(cols)] for _ in range(rows)]

for i in range(len(pre_processed_text)):
    for j in range(len(pre_processed_text[i])):
        cat_no = new_cat_column[i]
        term_s = pre_processed_text[i][j]
        col_no = list(term_count).index(term_s)
        tf_matrix[cat_no-1][col_no] += 1
```

[8]

Using this term frequency matrix, we find the Class Frequency. And store them in a dictionary.



```python
df1 = pd.read_csv('tf_matrix.csv')
```

[10]

```python
df1
```

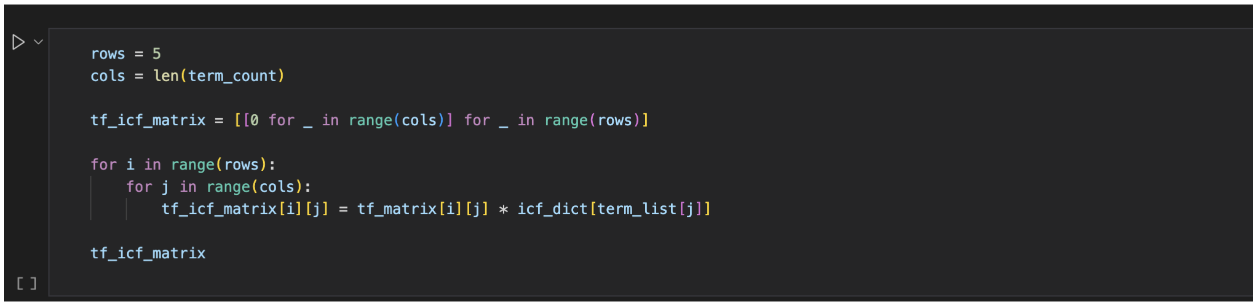
[11]

```
   worldcom  ex  boss  launch  defenc  lawyer  defend  former  chief  berni  ...  610m  wharf  glocer  6gb  4gb  santi  unwelcom  defa
0        54  10    40      41      18      26       5      73    156      8  ...     1      1       1    0    0      0         0
1         0   2     5     121       9       8       9      15     46      0  ...     0      0       0    1    1      5         1
2         0  35     5      40      16      19      35      81     58      0  ...     0      0       0    0    0      0         0
3         0   4    77       8      37       7      75      85     24      0  ...     0      0       0    0    0      0         0
4         0  10     2      25       1      15       3      52     16      0  ...     0      0       0    0    0      0         0

5 rows × 17147 columns
```

```python
term_list = [i for i in term_count]

cf_dict = {}
for i in term_list:
    count = (df1[i] == 0).sum()
    cf_dict[i] = 5-count

cf_dict
```

[12]

Using the CF, we calculate the ICF from the given formula and finally find the TF-ICF.

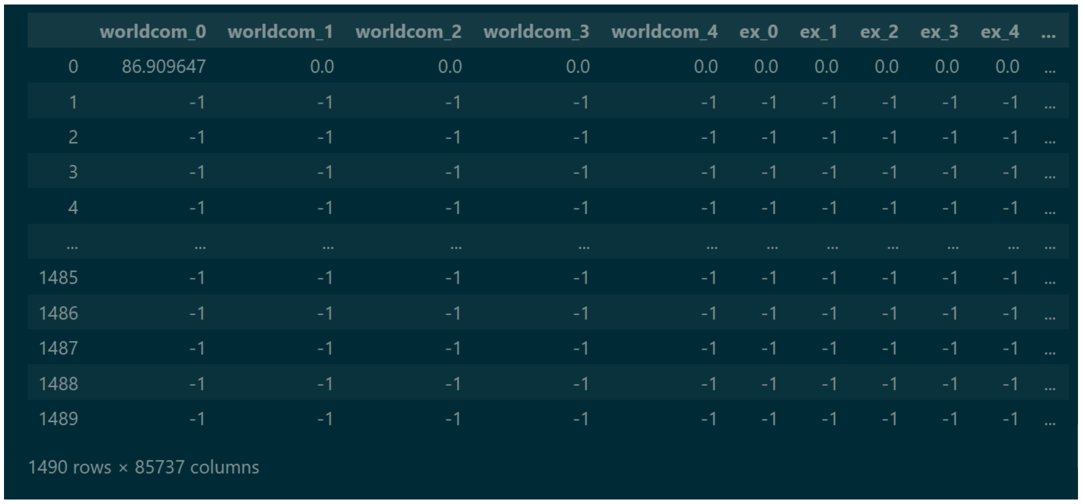


```python
rows = 5
cols = len(term_count)

tf_icf_matrix = [[0 for _ in range(cols)] for _ in range(rows)]

for i in range(rows):
    for j in range(cols):
        tf_icf_matrix[i][j] = tf_matrix[i][j] * icf_dict[term_list[j]]

tf_icf_matrix
```

[ ]

Training the Naive Bayes classifier with TF-ICF:

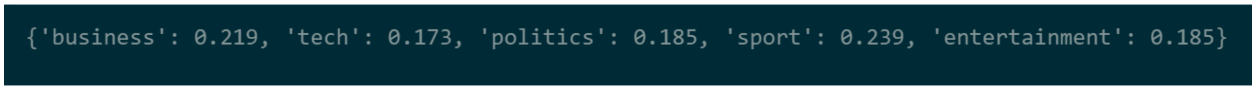
Naive Bayes Classifier is implemented with TC-ICF scores of each word and and each category for every doc i.e. every doc is represented as a vector of all words for each category, where the value for the words present in the doc is the tf_icf scores of that word and for the words not present in the doc, it is -1. We have chosen it to be -1, as absence of certain words also governs the category of the doc.
Here, each row represents a doc, and columns represent words_category:



	worldcom_0	worldcom_1	worldcom_2	worldcom_3	worldcom_4	ex_0	ex_1	ex_2	ex_3	ex_4	...
0	86.909647	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...
2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...
3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...
...	...	...	...	...	...	...	...	...	...	...	...
1485	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...
1486	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...
1487	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...
1488	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...
1489	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...

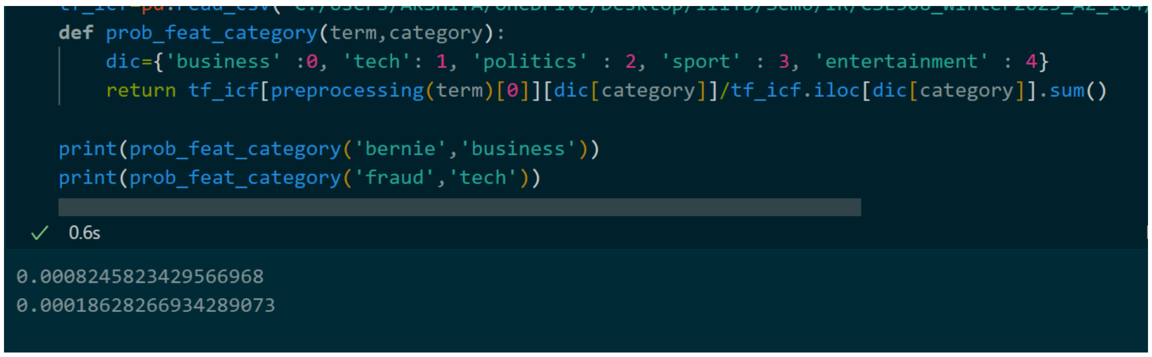
1490 rows × 85737 columns

Probability of each category based on the frequency of documents in the training set that belong to that category:



```
{'business': 0.219, 'tech': 0.173, 'politics': 0.185, 'sport': 0.239, 'entertainment': 0.185}
```

Probability of each feature given each category based on the TF-ICF values of that feature in documents belonging to that category:
We have created a function which takes the category and the term for which we want to calculate the probability. The function takes the original term present in the doc, and not the preprocessed one and then processes it, as it is very natural and intuitive to search for relevance of original terms.



```
def prob_feat_category(term,category):
    dic={'business' :0, 'tech': 1, 'politics' : 2, 'sport' : 3, 'entertainment' : 4}
    return tf_icf[preprocessing(term)[0]][dic[category]]/tf_icf.iloc[dic[category]].sum()

print(prob_feat_category('bernie','business'))
print(prob_feat_category('fraud','tech'))
```

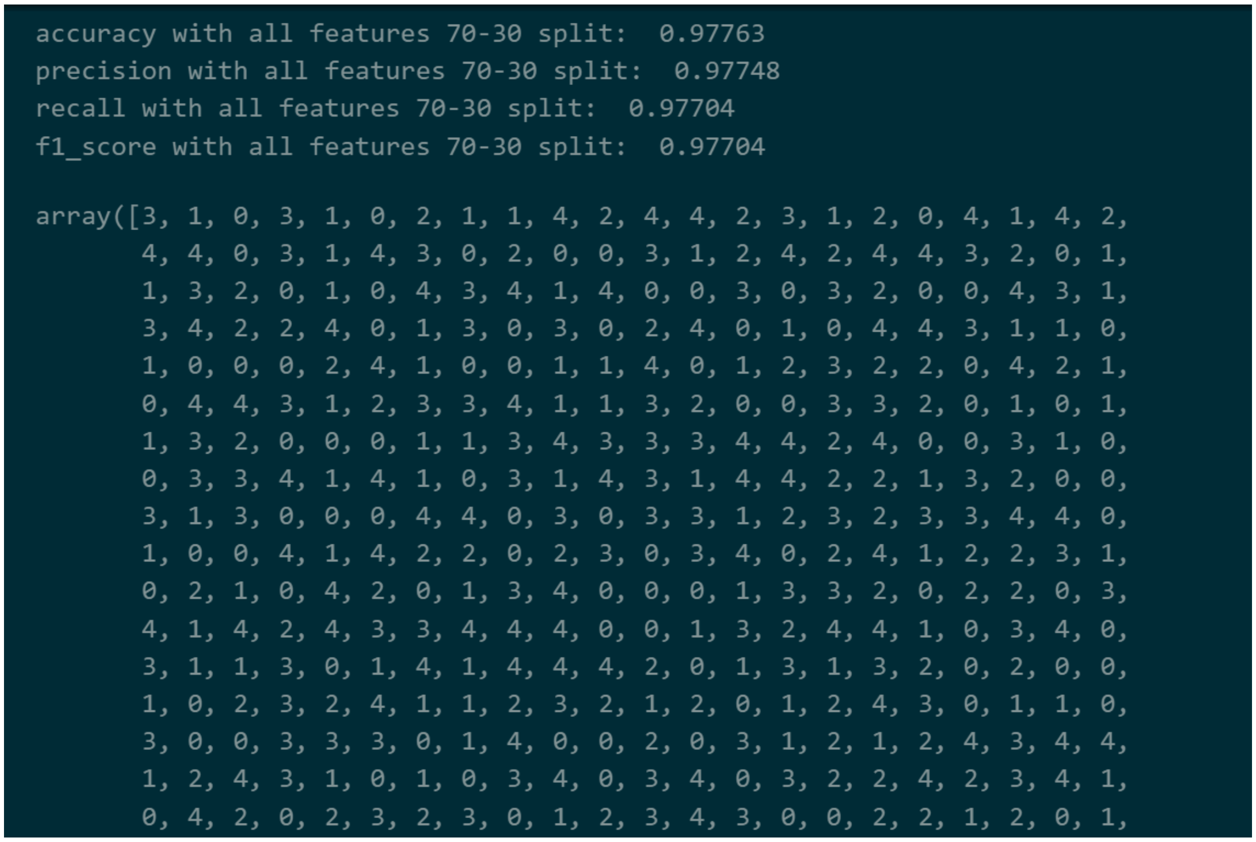
✓ 0.6s

```
0.0008245823429566968
0.00018628266934289073
```

Testing the Naive Bayes classifier with TF-ICF:

Used the testing set to evaluate performance of the model. This is done for the 70-30 split data.

Accuracy, precision, recall, and F1 score of the classifier:



```
accuracy with all features 70-30 split:  0.97763
precision with all features 70-30 split:  0.97748
recall with all features 70-30 split:  0.97704
f1_score with all features 70-30 split:  0.97704

array([3, 1, 0, 3, 1, 0, 2, 1, 1, 4, 2, 4, 4, 2, 3, 1, 2, 0, 4, 1, 4, 2,
       4, 4, 0, 3, 1, 4, 3, 0, 2, 0, 0, 3, 1, 2, 4, 2, 4, 4, 3, 2, 0, 1,
       1, 3, 2, 0, 1, 0, 4, 3, 4, 1, 4, 0, 0, 3, 0, 3, 2, 0, 0, 4, 3, 1,
       3, 4, 2, 2, 4, 0, 1, 3, 0, 3, 0, 2, 4, 0, 1, 0, 4, 4, 3, 1, 1, 0,
       1, 0, 0, 0, 2, 4, 1, 0, 0, 1, 1, 4, 0, 1, 2, 3, 2, 2, 0, 4, 2, 1,
       0, 4, 4, 3, 1, 2, 3, 3, 4, 1, 1, 3, 2, 0, 0, 3, 3, 2, 0, 1, 0, 1,
       1, 3, 2, 0, 0, 0, 1, 1, 3, 4, 3, 3, 3, 4, 4, 2, 4, 0, 0, 3, 1, 0,
       0, 3, 3, 4, 1, 4, 1, 0, 3, 1, 4, 3, 1, 4, 4, 2, 2, 1, 3, 2, 0, 0,
       3, 1, 3, 0, 0, 0, 4, 4, 0, 3, 0, 3, 3, 1, 2, 3, 2, 3, 3, 4, 4, 0,
       1, 0, 0, 4, 1, 4, 2, 2, 0, 2, 3, 0, 3, 4, 0, 2, 4, 1, 2, 2, 3, 1,
       0, 2, 1, 0, 4, 2, 0, 1, 3, 4, 0, 0, 0, 1, 3, 3, 2, 0, 2, 2, 0, 3,
       4, 1, 4, 2, 4, 3, 3, 4, 4, 4, 0, 0, 1, 3, 2, 4, 4, 1, 0, 3, 4, 0,
       3, 1, 1, 3, 0, 1, 4, 1, 4, 4, 4, 2, 0, 1, 3, 1, 3, 2, 0, 2, 0, 0,
       1, 0, 2, 3, 2, 4, 1, 1, 2, 3, 2, 1, 2, 0, 1, 2, 4, 3, 0, 1, 1, 0,
       3, 0, 0, 3, 3, 3, 0, 1, 4, 0, 0, 2, 0, 3, 1, 2, 1, 2, 4, 3, 4, 4,
       1, 2, 4, 3, 1, 0, 1, 0, 3, 4, 0, 3, 4, 0, 3, 2, 2, 4, 2, 3, 4, 1,
       0, 4, 2, 0, 2, 3, 2, 3, 0, 1, 2, 3, 4, 3, 0, 0, 2, 2, 1, 2, 0, 1,
```

Improving the classifier:

For improving the classifier performance, we have tried the following things:

1. 50-50 split with tf-icf features
2. 60-40 split with tf-icf features
3. 80-20 split with tf-icf features

As, 80-20 split gave best performance of all 4 splits, we tried further things on that only:

4. Selecting 20 best tf-icf features, 80-20 split
5. Selecting 200 best tf-icf features, 80-20 split

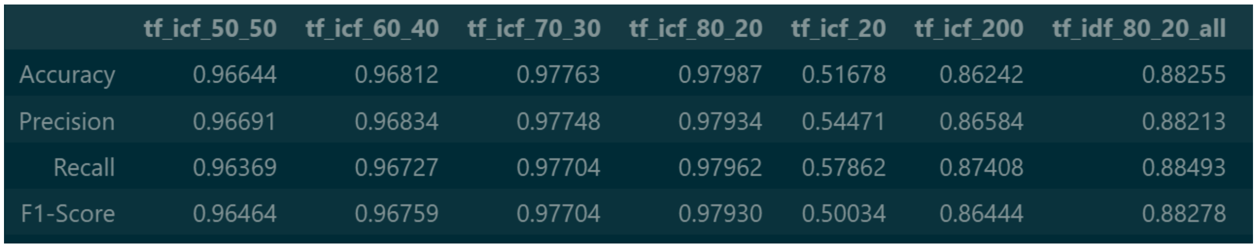
Tried taking the tf-idf features instead of tf–cf features:

6. 80-20 split with all tf-idf features

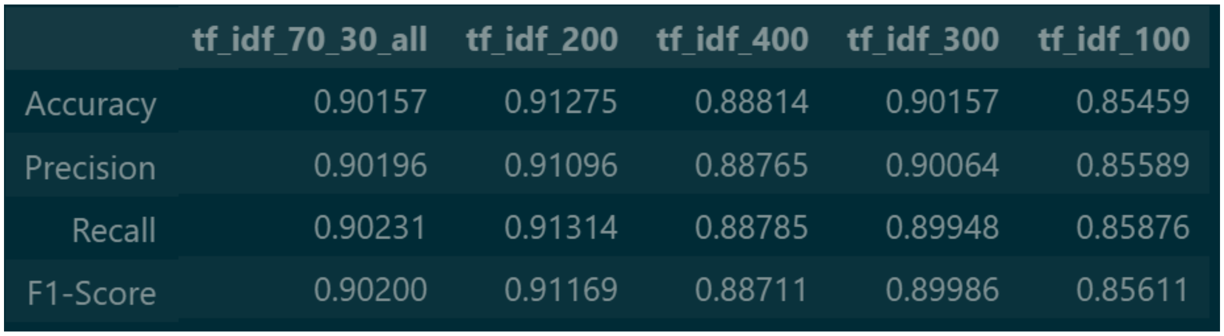
7. 70-30 split with all tf-idf features

Out of these two, 70-30 split gave better performance, gave further tried by selecting k best features with it:

8. 80-20 split, 200 best features
9. 80-20 split, 400 best features
10. 80-20 split, 300 best features
11. 80-20 split, 100 best features



	tf_icf_50_50	tf_icf_60_40	tf_icf_70_30	tf_icf_80_20	tf_icf_20	tf_icf_200	tf_idf_80_20_all
Accuracy	0.96644	0.96812	0.97763	0.97987	0.51678	0.86242	0.88255
Precision	0.96691	0.96834	0.97748	0.97934	0.54471	0.86584	0.88213
Recall	0.96369	0.96727	0.97704	0.97962	0.57862	0.87408	0.88493
F1-Score	0.96464	0.96759	0.97704	0.97930	0.50034	0.86444	0.88278



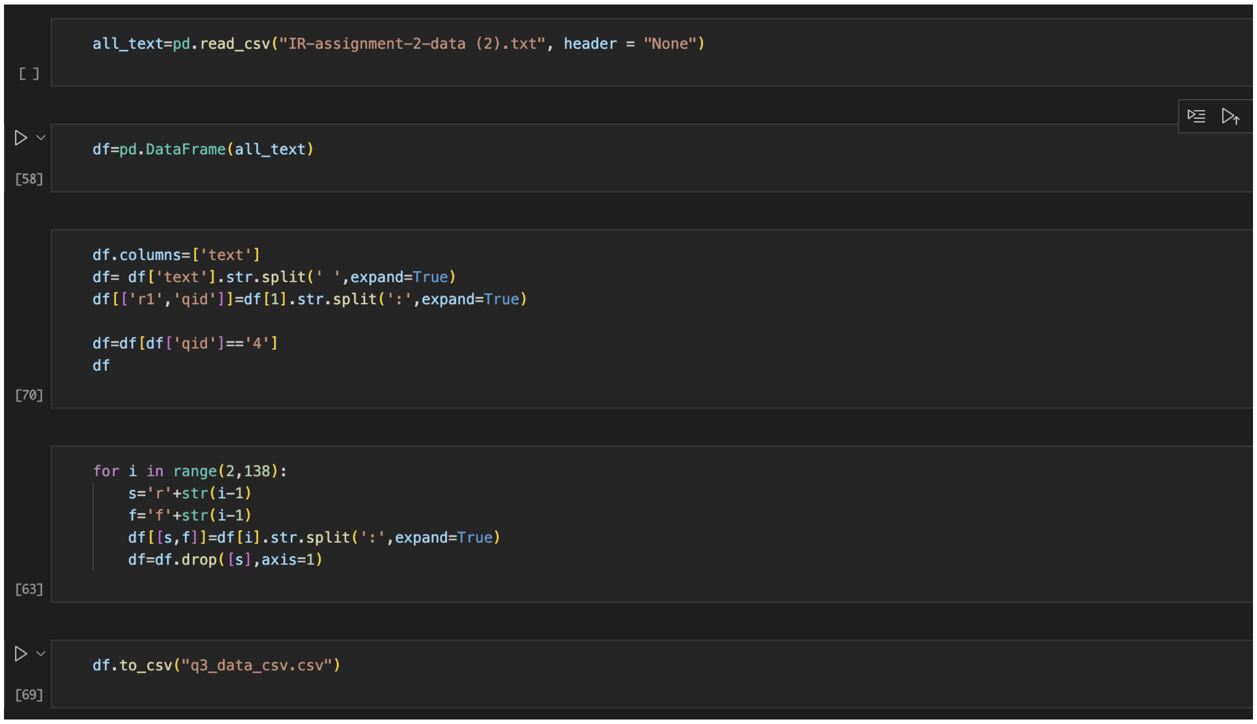
	tf_idf_70_30_all	tf_idf_200	tf_idf_400	tf_idf_300	tf_idf_100
Accuracy	0.90157	0.91275	0.88814	0.90157	0.85459
Precision	0.90196	0.91096	0.88765	0.90064	0.85589
Recall	0.90231	0.91314	0.88785	0.89948	0.85876
F1-Score	0.90200	0.91169	0.88711	0.89986	0.85611

After trying all these changes, the best model we get is the one with:
TF-ICF features, 80-20 split, all features

Conclusion:
TF-ICF values give a better model than TF-IDF. This makes sense as tf-icf preserves the importance of words with respect to each category, i.e. how important a term's role in deciding the category of the doc. The presence/absence of certain words governs categories, and this info is given by tf-icf values. Whereas in tf-idf values, category-wise information is lost.
The impact of different split ratios can be seen from the performance table above.
Selecting best features reduces the performance in case of tf-icf, possibly because it reduces the information, whereas it increases the performance slightly in case of tf-idf and selection of 200 best features.

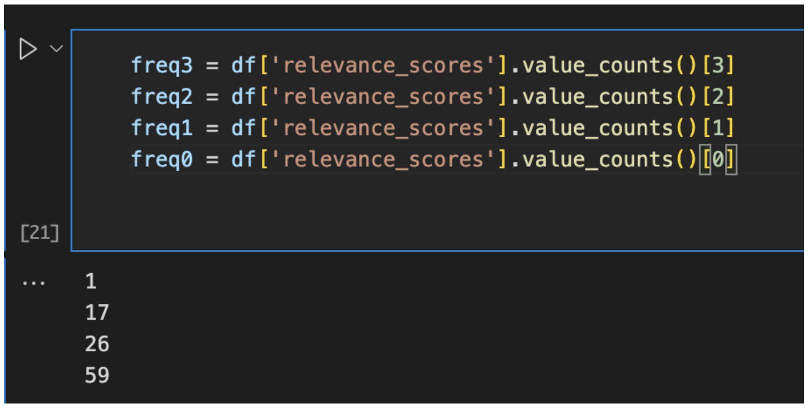
Q3)
Ans3)

For this question, we began by cleaning the given dataset. It contained a lot of data that we did not require. First we selected the data with qid:4, then we spilt the columns of features to drive their value and add it in the dataset. At the end we get a dataset of qid:4 with relevance score of query and the values of 136 features.



```
[ ]
all_text=pd.read_csv("IR-assignment-2-data (2).txt", header = "None")

[58]
df=pd.DataFrame(all_text)

[70]
df.columns=['text']
df= df['text'].str.split(' ',expand=True)
df[['r1','qid']]=df[1].str.split(':',expand=True)

df=df[df['qid']=='4']
df

[63]
for i in range(2,138):
    s='r'+str(i-1)
    f='f'+str(i-1)
    df[[s,f]]=df[i].str.split(':',expand=True)
    df=df.drop([s],axis=1)

[69]
df.to_csv("q3_data_csv.csv")
```

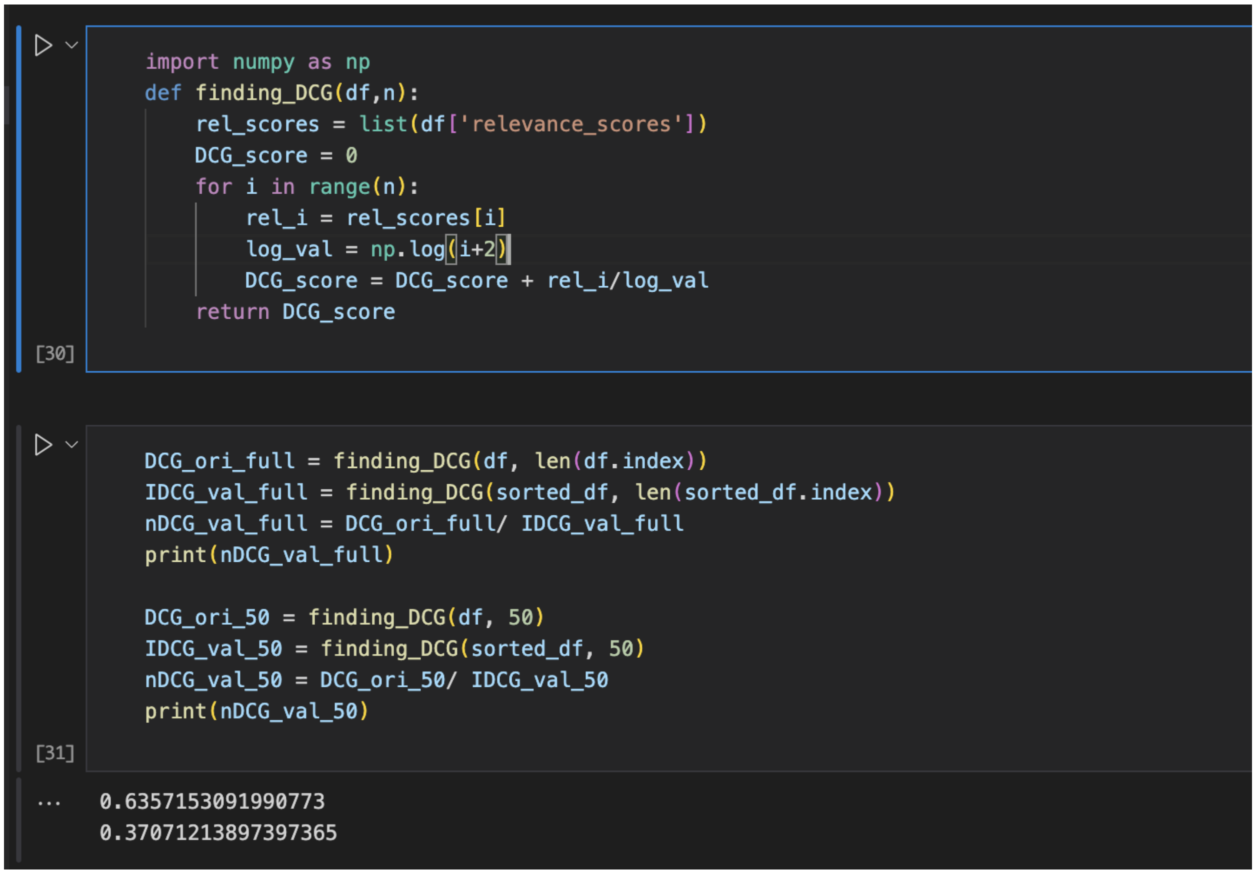
Now we needed create a file that rearranges the query-url pairs in order of the maximum DCG (discounted cumulative gain). For this task the basic logic was to arrange the given dataframe in descending order of the relevance scores. Using that we get a sorted_df which is the ideal query-url pair.
The number of such data frames possible are calculated after knowing the frequency of each relevance score.



```
freq3 = df['relevance_scores'].value_counts()[3]
freq2 = df['relevance_scores'].value_counts()[2]
freq1 = df['relevance_scores'].value_counts()[1]
freq0 = df['relevance_scores'].value_counts()[0]

[21]
...  1
     17
     26
     59
```

Final answer can be calculated as, ans = 1! * 17! * 26! * 59! = 19893497375938370599826047614905329896936840170566570588205180312704857992695193482412686565431050240000000000000000000000

Now in the second part we need to calculate the NDCG, for that we first calculate the DCG of the original processed dataframe and then the DCG of the ideal dataframe. We do this for both till position 50 and also the entire dataframe(103 rows).

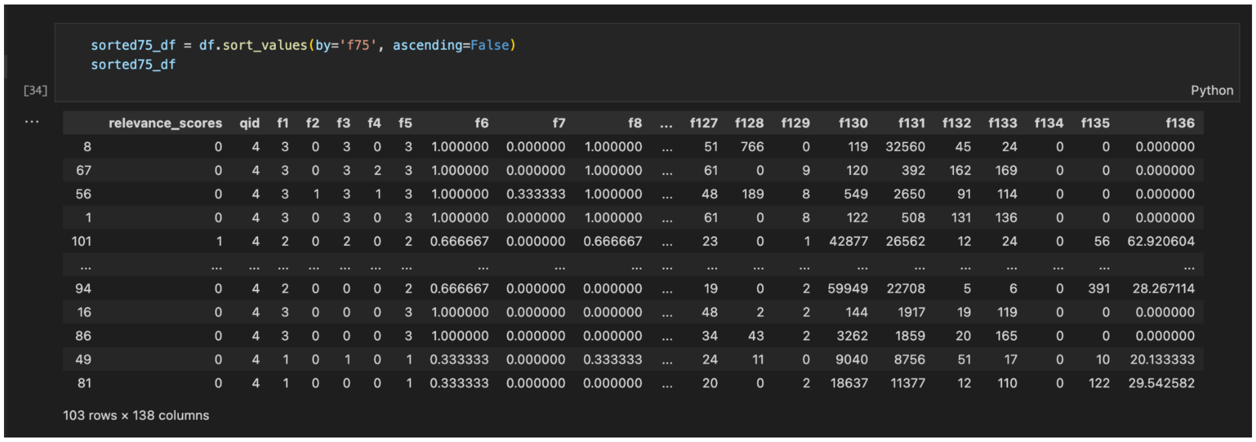


```
import numpy as np
def finding_DCG(df,n):
    rel_scores = list(df['relevance_scores'])
    DCG_score = 0
    for i in range(n):
        rel_i = rel_scores[i]
        log_val = np.log(i+2)
        DCG_score = DCG_score + rel_i/log_val
    return DCG_score
```
[30]

```
DCG_ori_full = finding_DCG(df, len(df.index))
IDCG_val_full = finding_DCG(sorted_df, len(sorted_df.index))
nDCG_val_full = DCG_ori_full/ IDCG_val_full
print(nDCG_val_full)

DCG_ori_50 = finding_DCG(df, 50)
IDCG_val_50 = finding_DCG(sorted_df, 50)
nDCG_val_50 = DCG_ori_50/ IDCG_val_50
print(nDCG_val_50)
```
[31]

```
0.6357153091990773
0.37071213897397365
```

For the third objective we sort the dataset based on the value of feature 75. And then use the relevance scores for further computation.

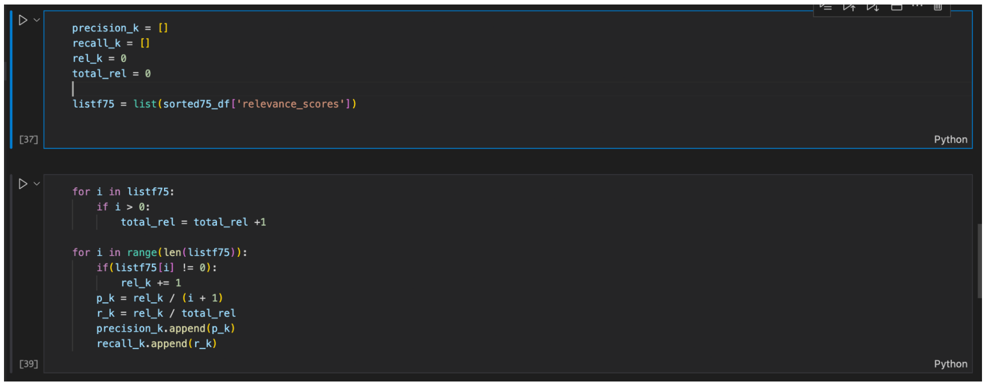


```
sorted75_df = df.sort_values(by='f75', ascending=False)
sorted75_df
```
[34]

	relevance_scores	qid	f1	f2	f3	f4	f5	f6	f7	f8	...	f127	f128	f129	f130	f131	f132	f133	f134	f135	f136
8	0	4	3	0	3	0	3	1.000000	0.000000	1.000000	...	51	766	0	119	32560	45	24	0	0	0.000000
67	0	4	3	0	3	2	3	1.000000	0.000000	1.000000	...	61	0	9	120	392	162	169	0	0	0.000000
56	0	4	3	1	3	1	3	1.000000	0.333333	1.000000	...	48	189	8	549	2650	91	114	0	0	0.000000
1	0	4	3	0	3	0	3	1.000000	0.000000	1.000000	...	61	0	8	122	508	131	136	0	0	0.000000
101	1	4	2	0	2	0	2	0.666667	0.000000	0.666667	...	23	0	1	42877	26562	12	24	0	56	62.920604
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
94	0	4	2	0	0	0	2	0.666667	0.000000	0.000000	...	19	0	2	59949	22708	5	6	0	391	28.267114
16	0	4	3	0	0	0	3	1.000000	0.000000	0.000000	...	48	2	2	144	1917	19	119	0	0	0.000000
86	0	4	3	0	0	0	3	1.000000	0.000000	0.000000	...	34	43	2	3262	1859	20	165	0	0	0.000000
49	0	4	1	0	1	0	1	0.333333	0.000000	0.333333	...	24	11	0	9040	8756	51	17	0	10	20.133333
81	0	4	1	0	0	0	1	0.333333	0.000000	0.000000	...	20	0	2	18637	11377	12	110	0	122	29.542582

103 rows × 138 columns

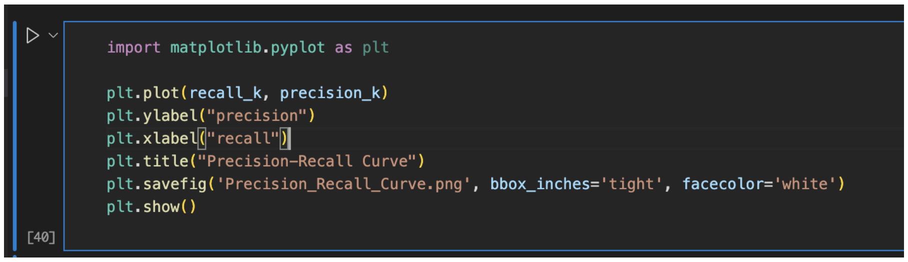
Finally we calculate the precision at each point and recall at each point by using a loop which checks whether the query is relevant or not and thus append the precision and recall at each point.



```
[37]
precision_k = []
recall_k = []
rel_k = 0
total_rel = 0

listf75 = list(sorted75_df['relevance_scores'])
```
Python

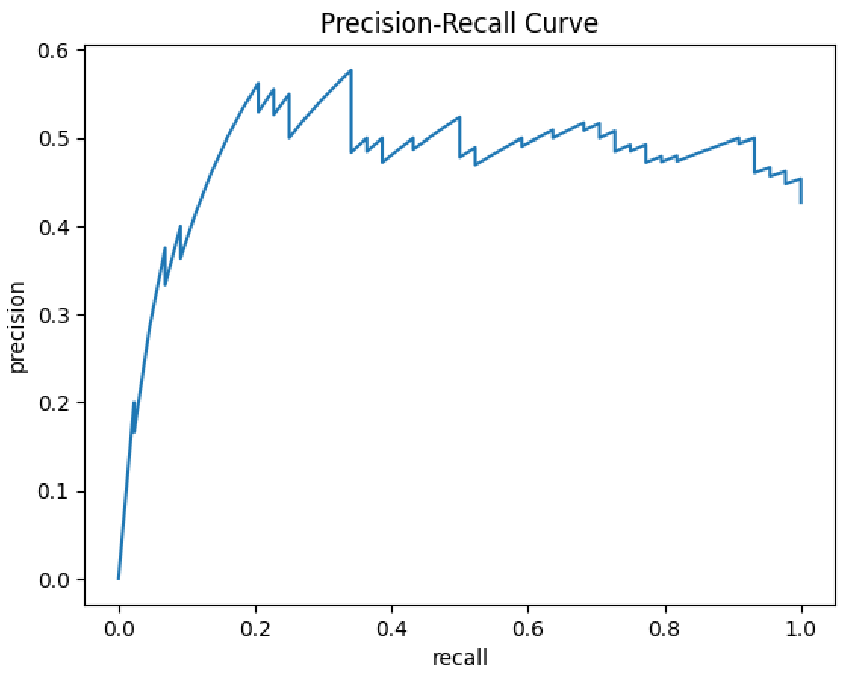
```
[39]
for i in listf75:
    if i > 0:
        total_rel = total_rel +1

for i in range(len(listf75)):
    if(listf75[i] != 0):
        rel_k += 1
    p_k = rel_k / (i + 1)
    r_k = rel_k / total_rel
    precision_k.append(p_k)
    recall_k.append(r_k)
```
Python

Using this we get arrays for precision and recall, and use those to generate the Precision Recall Curve.



```
[40]
import matplotlib.pyplot as plt

plt.plot(recall_k, precision_k)
plt.ylabel("precision")
plt.xlabel("recall")
plt.title("Precision-Recall Curve")
plt.savefig('Precision_Recall_Curve.png', bbox_inches='tight', facecolor='white')
plt.show()
```

Precision-Recall Curve:


Precision-Recall Curve
precision
recall