Q1.TFIDF and Jaccard Coefficient

TFIDF calculation for different weighting scheme

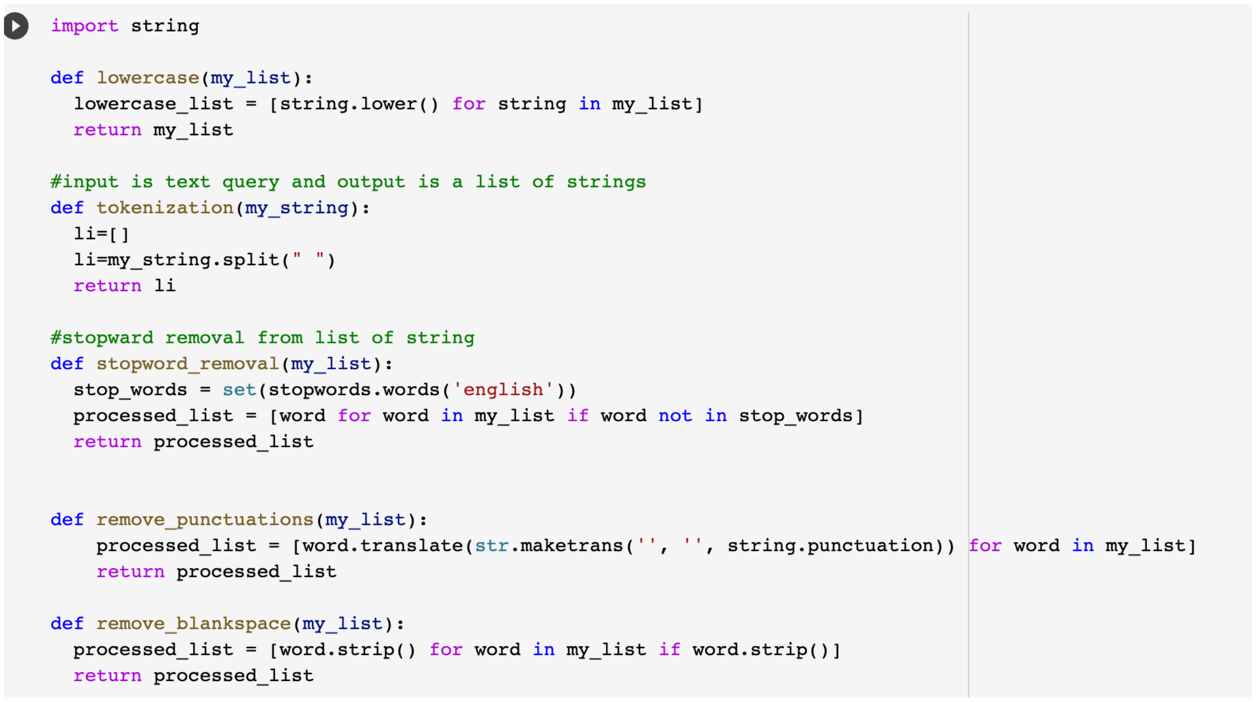
Assumption

- We are assuming that the words that are coming in query will be part of vocabulary otherwise we are deleting the word from the list.
- Tfidf follows the bag of word concept, and assumes that the order of the words does not matter in defining the relevance.
- Tfidf assumes that the each term is independent of any other term in the document.

- We are assuming that the vocabulary is static that it is not updated at particular interval.

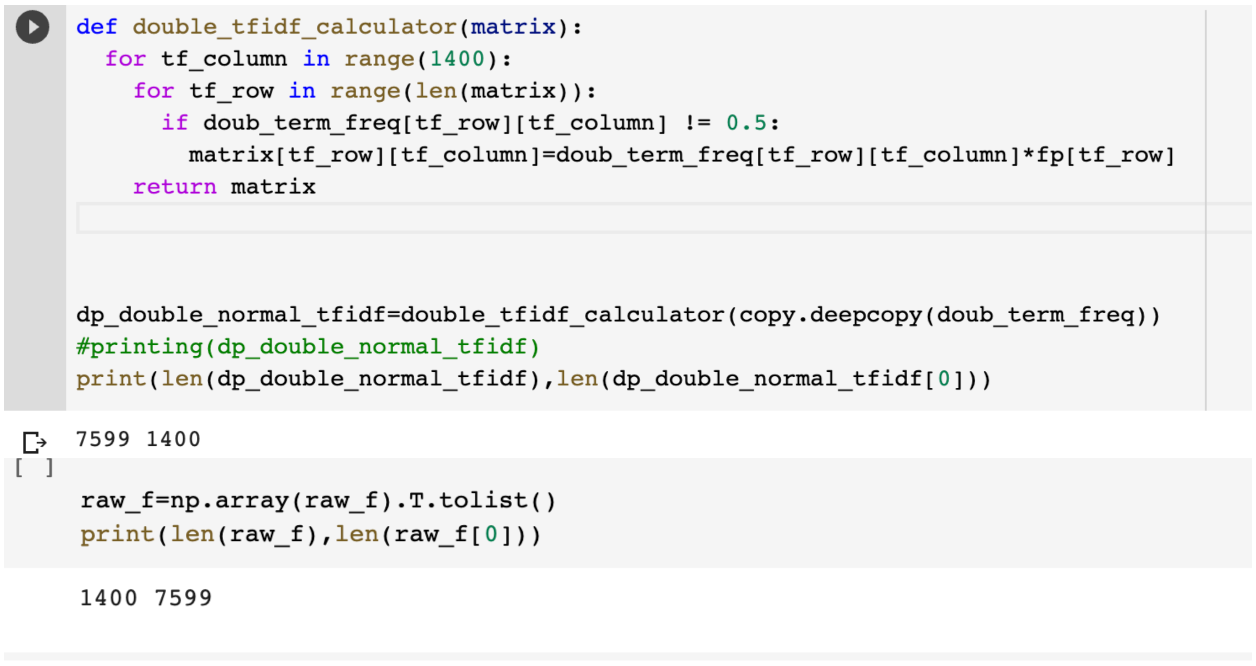
Methodology

1)First we need to do preprocessing for both the vocabulary and the query



```python
import string

def lowercase(my_list):
  lowercase_list = [string.lower() for string in my_list]
  return my_list

#input is text query and output is a list of strings
def tokenization(my_string):
  li=[]
  li=my_string.split(" ")
  return li

#stopward removal from list of string
def stopword_removal(my_list):
  stop_words = set(stopwords.words('english'))
  processed_list = [word for word in my_list if word not in stop_words]
  return processed_list


def remove_punctuations(my_list):
    processed_list = [word.translate(str.maketrans('', '', string.punctuation)) for word in my_list]
    return processed_list

def remove_blankspace(my_list):
  processed_list = [word.strip() for word in my_list if word.strip()]
  return processed_list
```

2) after pre-processing the vocabulary we need to build the tfidf matrix for each of the weighting scheme.



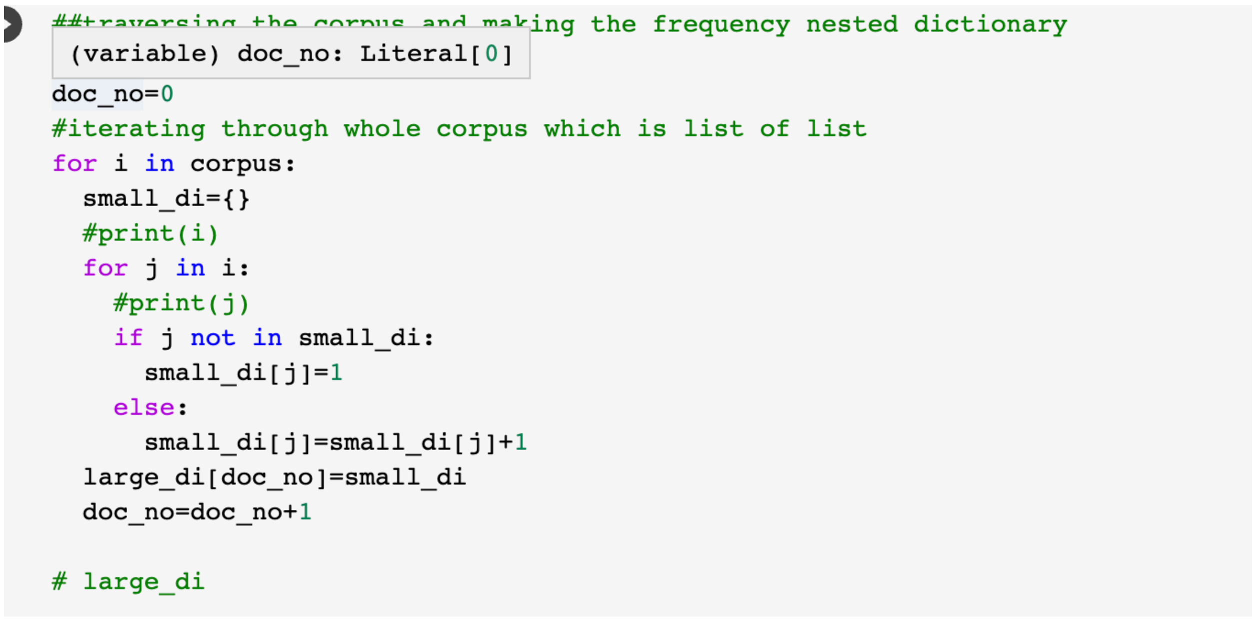
```python
def double_tfidf_calculator(matrix):
  for tf_column in range(1400):
    for tf_row in range(len(matrix)):
      if doub_term_freq[tf_row][tf_column] != 0.5:
        matrix[tf_row][tf_column]=doub_term_freq[tf_row][tf_column]*fp[tf_row]
    return matrix



dp_double_normal_tfidf=double_tfidf_calculator(copy.deepcopy(doub_term_freq))
#printing(dp_double_normal_tfidf)
print(len(dp_double_normal_tfidf),len(dp_double_normal_tfidf[0]))
```

```
7599 1400
```

```
[ ]
```

```python
raw_f=np.array(raw_f).T.tolist()
print(len(raw_f),len(raw_f[0]))
```

```
1400 7599
```



```python
##traversing the corpus and making the frequency nested dictionary
 (variable) doc_no: Literal[0]
doc_no=0
#iterating through whole corpus which is list of list
for i in corpus:
  small_di={}
  #print(i)
  for j in i:
    #print(j)
    if j not in small_di:
      small_di[j]=1
    else:
      small_di[j]=small_di[j]+1
  large_di[doc_no]=small_di
  doc_no=doc_no+1

# large_di
```

3) taking input query and preprocessing an then finding the files that are relevant to the below query

```python
def preprocessing(query_list):
  query_list=tokenization(query_list)
  my_list=remove_blankspace(query_list)
  my_list=remove_punctuations(my_list)
  my_list=lowercase(my_list)
  my_list=stopword_removal(my_list)
  return my_list

#testing my preprocessing code
print("enter a Query that you want to check")
query_list=input()
input_query=preprocessing(query_list)
preprocessing(query_list)
input_query
```

Query:-in the study of high-speed viscous flow past a two-dimensional body it is usually necessary to consider a curved shock wave emitting from the nose or leading edge of the body . consequently, there exists an inviscid rotational flow region between the shock wave and the boundary layer

Result:-

The result for different weighting schemes are as follows

```python
#for binary tfidf
#first transforming the query_vector in terms of binary_tfidf
query_vector_binary=query_vector.copy()#this is the termfrequency vector of query
# print(query_vector_binary[2571])                                                        in a list
# print(fp[2571])

#this is the term frequency vector for binary
for i in range(len(query_vector_binary)):
  if(query_vector_binary[i]>0):
    query_vector_binary[i]=1

  #converting to tfidf for the binary term frequency of the query
query_vector_binary=tf_to_tfidf(query_vector_binary.copy())

#now finding the relevant documents in binary term frequency term
cos_vector=[]
for i in range(len(binary_tfidf)):
  li=[]
  for j in range(len(binary_tfidf[i])):
    li.append(binary_tfidf[i][j])
  cos_vector.append(cosine_similarity(li,query_vector_binary))
  #print(cos_vector)
m=top_5_indexes(cos_vector)
print("the files that are relevant are")
for i in m:
  number_to_file_name(i+1)
```

```
the files that are relevant are
/content/CSE508_Winter2023_Dataset/cranfield0002
/content/CSE508_Winter2023_Dataset/cranfield1267
/content/CSE508_Winter2023_Dataset/cranfield0308
/content/CSE508_Winter2023_Dataset/cranfield0334
/content/CSE508_Winter2023_Dataset/cranfield1309
```

TERM FREQUENCY RELEVANCE//wrong have to change the term frequency calculation

```
#for term frequency  tfidf
#first transforming the query_vector in terms of binary_tfidf
query_vector_term=query_vector.copy()
sum=0
for i in range(len(query_vector_term)):
  sum=sum+query_vector_term[i]
for i in range(len(query_vector_term)):
  query_vector_term[i]=query_vector_term[i]/sum

query_vector_term=tf_to_tfidf(query_vector_term.copy())
#now finding the relevant documents in binary term frequency term
cos_vector=[]
for i in range(len(dp_term_tfidf)):
  li=[]
  for j in range(len(dp_term_tfidf[i])):
    li.append(dp_term_tfidf[i][j])
  cos_vector.append(cosine_similarity(li,query_vector_term))
  #print(cos_vector)
m=top_5_indexes(cos_vector)
print("the files that are relevant are")
for i in m:
  #print(i+1,end=" ")
  number_to_file_name(i+1)
```

```
the files that are relevant are
/content/CSE508_Winter2023_Dataset/cranfield0002
/content/CSE508_Winter2023_Dataset/cranfield0334
/content/CSE508_Winter2023_Dataset/cranfield0439
/content/CSE508_Winter2023_Dataset/cranfield0025
/content/CSE508_Winter2023_Dataset/cranfield0064
```

LOG NORMALIZATION

```
#dp_log_tfidf
#for term frequency  tfidf
#first transforming the query_vector in terms of binary_tfidf
query_vector_log=query_vector.copy()
for i in range(len(query_vector_log)):
  query_vector_log[i]=(math.log(1+query_vector[i],10))

query_vector_log=tf_to_tfidf(query_vector_log.copy())
#now finding the relevant documents in binary term frequency term
cos_vector=[]
for i in range(len(dp_log_tfidf)):
  li=[]
  for j in range(len(dp_log_tfidf[i])):
    li.append(dp_log_tfidf[i][j])
  cos_vector.append(cosine_similarity(li,query_vector_log))
  #print(cos_vector)
m=top_5_indexes(cos_vector)
print("the files that are relevant are")
for i in m:
  #print(i+1,end=" ")
  number_to_file_name(i+1)
```

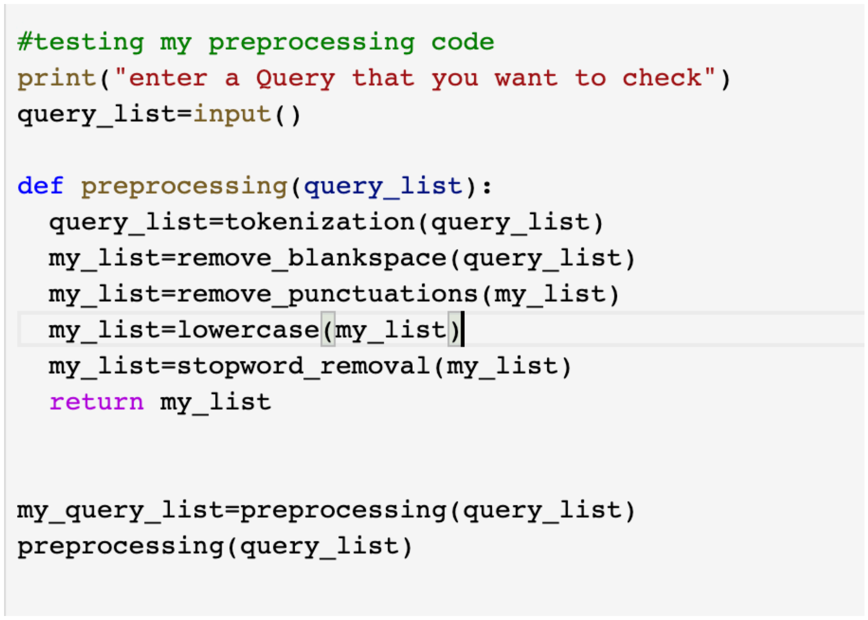
```
the files that are relevant are
/content/CSE508_Winter2023_Dataset/cranfield0002
/content/CSE508_Winter2023_Dataset/cranfield0334
/content/CSE508_Winter2023_Dataset/cranfield0439
/content/CSE508_Winter2023_Dataset/cranfield1267
/content/CSE508_Winter2023_Dataset/cranfield0192
```

```
the files that are relevant are
/content/CSE508_Winter2023_Dataset/cranfield0002
/content/CSE508_Winter2023_Dataset/cranfield0064
/content/CSE508_Winter2023_Dataset/cranfield0334
/content/CSE508_Winter2023_Dataset/cranfield0190
/content/CSE508_Winter2023_Dataset/cranfield1112
```

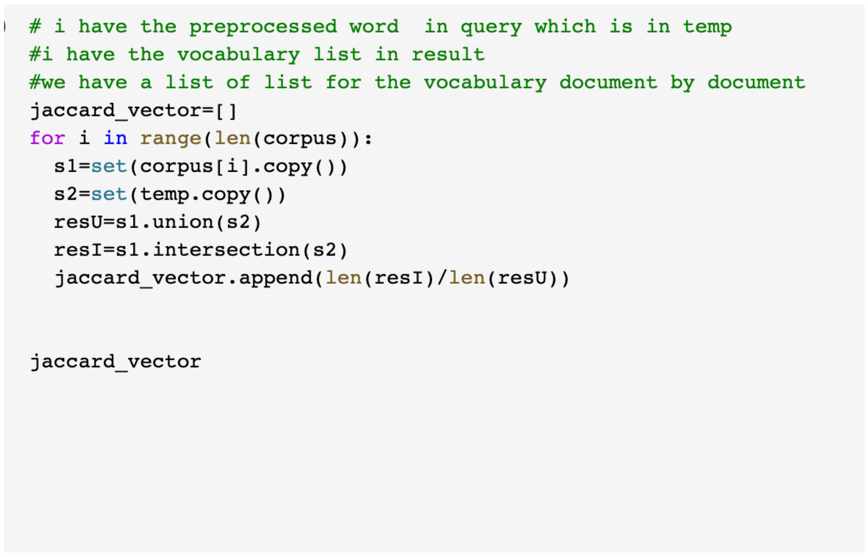
```
#double_normal
import numpy as np
#for term frequency  tfidf
#first transforming the query_vector in terms of binary_tfidf
query_vector_double_normal=query_vector.copy()
#i need to find the maximum term frequency of terms of that query

# max=-1
# for i in range(len(query_vector_double_normal)):
#   if query_vector_double_normal[i]>max:
#     max=query_vector_double_normal[i]

max = np.max(query_vector_double_normal)

for i in range(len(query_vector_double_normal)):
  query_vector_double_normal[i]=0.5+0.5*(query_vector[i]/max)


query_vector_double_normal=tf_to_tfidf2(query_vector_double_normal.copy())
#now finding the relevant documents in binary term frequency term
cos_vector=[]
for i in range(len(dp_double_normal_tfidf)):
  li=[]
  for j in range(len(dp_double_normal_tfidf[i])):
    li.append(dp_double_normal_tfidf[i][j])
  cos_vector.append(cosine_similarity(li,query_vector_double_normal))
  #print(cos_vector)
m=top_5_indexes(cos_vector)
print("the files that are relevant are")
for i in m:
  #print(i+1,end=" ")
  number_to_file_name(i+1)
```

Jaccard Coefficient

Methodologies:
1)First pre processed the data as per question and then pre processed the query.


```
#testing my preprocessing code
print("enter a Query that you want to check")
query_list=input()

def preprocessing(query_list):
  query_list=tokenization(query_list)
  my_list=remove_blankspace(query_list)
  my_list=remove_punctuations(my_list)
  my_list=lowercase(my_list)
  my_list=stopword_removal(my_list)
  return my_list


my_query_list=preprocessing(query_list)
preprocessing(query_list)
```

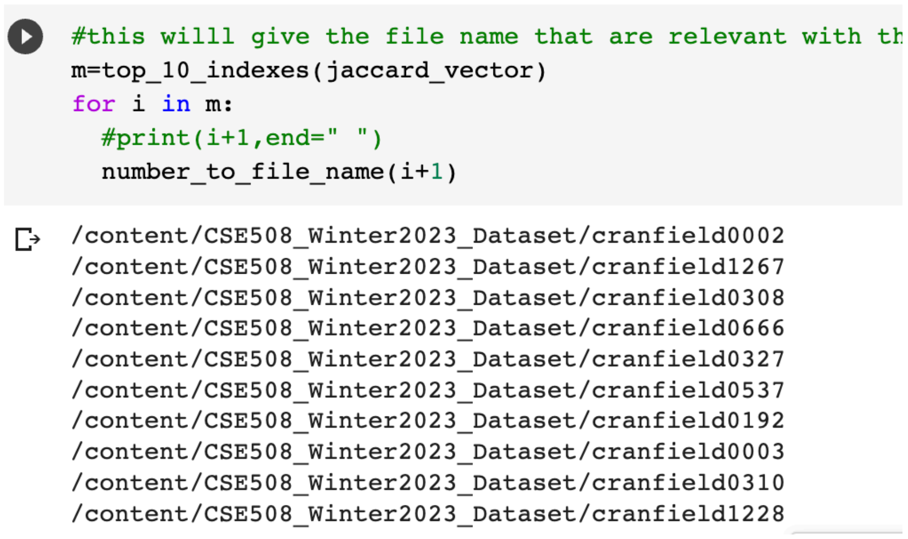
2) We are finding the jaccard coefficient one by one for each document and then sorting in descending order


```
# i have the preprocessed word  in query which is in temp
#i have the vocabulary list in result
#we have a list of list for the vocabulary document by document
jaccard_vector=[]
for i in range(len(corpus)):
  s1=set(corpus[i].copy())
  s2=set(temp.copy())
  resU=s1.union(s2)
  resI=s1.intersection(s2)
  jaccard_vector.append(len(resI)/len(resU))


jaccard_vector
```

3) given the below query this is the output that we are getting

Query:-in the study of high-speed viscous flow past a two-dimensional body it is usually necessary to consider a curved shock wave emitting from the nose or leading edge of the body . consequently, there exists an inviscid rotational flow region between the shock wave and the boundary layer

Result:-



```
#this willl give the file name that are relevant with th
m=top_10_indexes(jaccard_vector)
for i in m:
  #print(i+1,end=" ")
  number_to_file_name(i+1)
```

```
/content/CSE508_Winter2023_Dataset/cranfield0002
/content/CSE508_Winter2023_Dataset/cranfield1267
/content/CSE508_Winter2023_Dataset/cranfield0308
/content/CSE508_Winter2023_Dataset/cranfield0666
/content/CSE508_Winter2023_Dataset/cranfield0327
/content/CSE508_Winter2023_Dataset/cranfield0537
/content/CSE508_Winter2023_Dataset/cranfield0192
/content/CSE508_Winter2023_Dataset/cranfield0003
/content/CSE508_Winter2023_Dataset/cranfield0310
/content/CSE508_Winter2023_Dataset/cranfield1228
```

Q3.Ranked-Information Retrieval and Evaluation

i) The first objective is to create a file that rearranges the query-url pairs in order of the maximum DCG (discounted cumulative gain). The number of such files that could be made should also be stated.

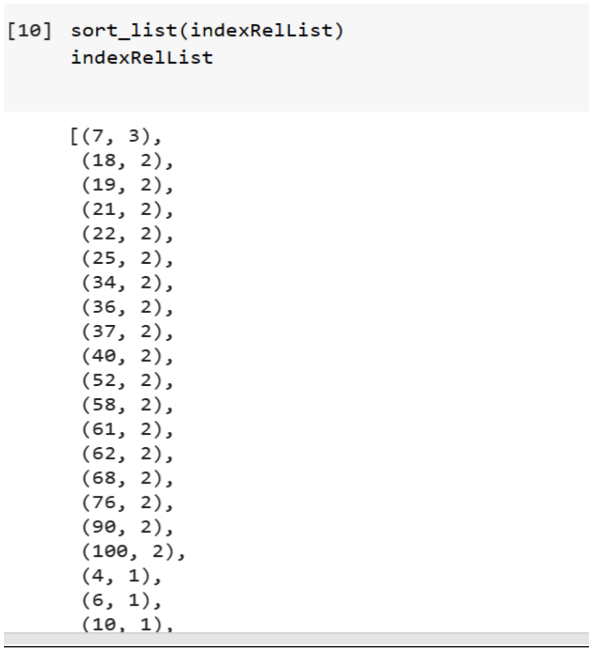
Methodology:

We are selecting the queries with qid:4 and using the relevance judgment labels as the relevance score.Then, maintaining a list that consist of a tuple with index and relevance score. Sort the list using the lambda function on the basis on second element as a key.Create a file with the query-url pairs in order of the maximum DCG (discounted cumulative gain). Next,we have counted the number of files with (max) DCG.

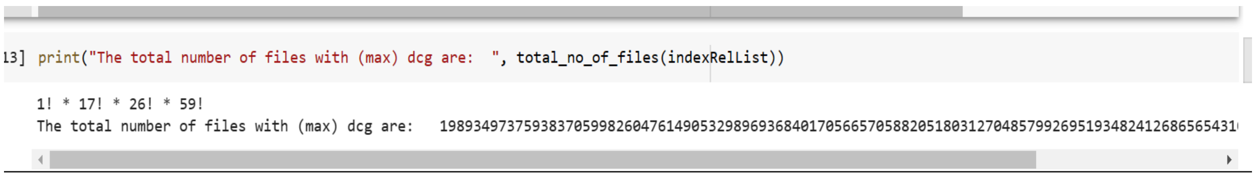
Assumptions:

The dataset contains various and their associated URLs with relevance judgement labels as relevance scores but here we have considered the queries with qid:4 using th relevance judgement label as the relevance score.

Results:



```
[10] sort_list(indexRelList)
     indexRelList

     [(7, 3),
      (18, 2),
      (19, 2),
      (21, 2),
      (22, 2),
      (25, 2),
      (34, 2),
      (36, 2),
      (37, 2),
      (40, 2),
      (52, 2),
      (58, 2),
      (61, 2),
      (62, 2),
      (68, 2),
      (76, 2),
      (90, 2),
      (100, 2),
      (4, 1),
      (6, 1),
      (10, 1),
```



```
13] print("The total number of files with (max) dcg are:  ", total_no_of_files(indexRelList))

    1! * 17! * 26! * 59!
    The total number of files with (max) dcg are:   198934973759383705998260476149053298969368401705665705882051803127048579926951934824126865654310
```

ii) The second task is to compute the nDCG (normalized discounted cumulative gain) for the dataset. This involves calculating nDCG at position 50 and for the entire dataset.

Methodology:

To calculate the nDCG firstly we have calculated the maximum DCG for the ideal case. Then,compute the nDCG at the position 50 and nDCG for the entire dataset.

Assumption:

We have considered the index and relevance score tuples in the sorted order of the max(DCG) that is already given in the first query to compute the nDCG at position 50 and the entire dataset.

Results:

```
dcg_max = maximum_Dcg(indexRelList)
print("The maximum DCG for the ideal case is: ",dcg_max)
```

```
The maximum DCG for the ideal case is:  20.989750804831445
```

```
entire_ndcg = nDCG(dataframe_id_q4,-100)
ndcg_at_50 = nDCG(dataframe_id_q4,50)
print("The calculated NDCG at postion 50 is:",ndcg_at_50)
print("The calculated NDCG for entire dataset:",entire_ndcg)
```

```
The calculated NDCG at postion 50 is: 0.3521042740324887
The calculated NDCG for entire dataset: 0.5979226516897831
```

iii) The third objective is to assume a model that simply ranks URLs on the basis of the value of feature 75 (sum of TF-IDF on the whole document) i.e.URL with higher feature values are considered more relevant. Any non zero relevance judgment value to be relevant. Plot a Precision-Recall curve for query “qid:4”.

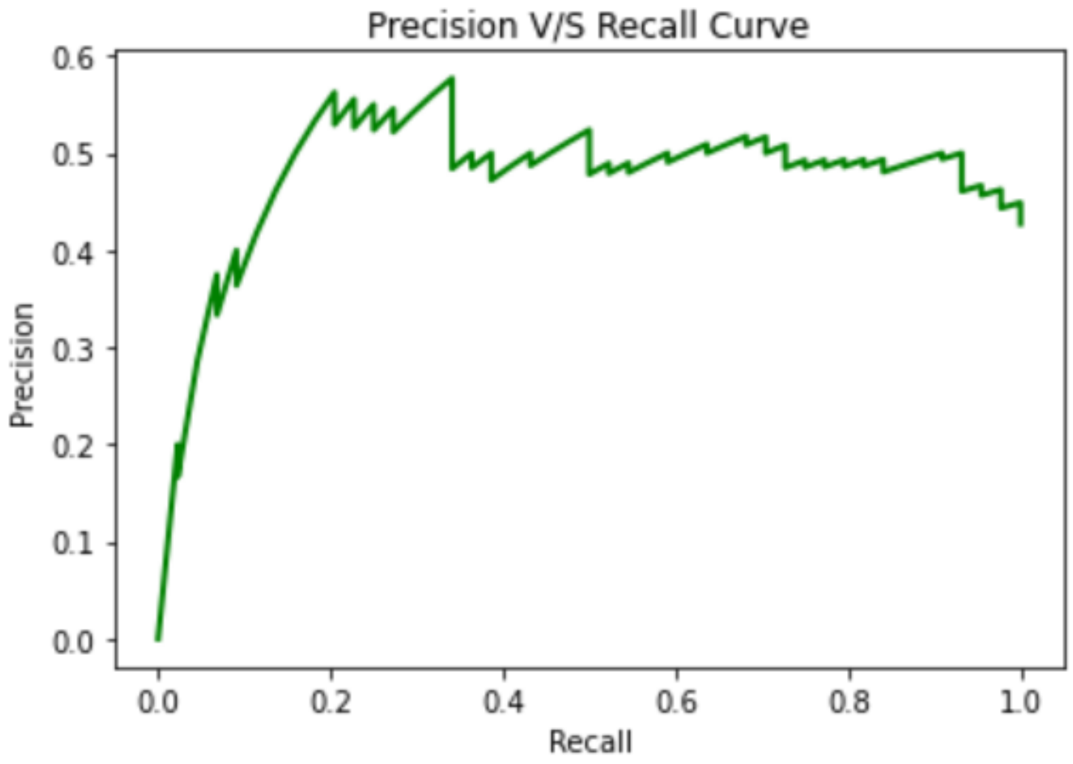
Methodology:

We have extracted the relevance scores and the IDF scores from the input Dataframe and created its tuple and then sort the tuple list in the descending orderby the IDF score.Also,calculated the no of documents to plot Precision-Recall curve.
First,we have calculated the relevance score and IDF score for each document and then the total number of the relevant documents in the dataset.Then computed the precision and recall for each document in the dataset.Then,plot graph for the same.(Precision-Recall)

Assumption:
We have assumed that the relevance score are in column 0 and the IDF score are in column 76.

Results:

```
calculate_Prec_Rec(dataframe_id_q4)
```


Precision V/S Recall Curve
Precision
Recall