Assignment No. 2
Report*(Group 97)*

MT21074	2020020	MT22067
Samiksha Garg	Akshat Tilak	Shambhavi Pathak

1. TF_IDF Matrix and Matching scores

Dataset provided:
Same as Assignment 1 (1400 documents).
Preprocessing was the same as Assignment 1.

Creating the TF_IDF matrix:
Query set ⇒ contains all the unique words that occur in all the documents.
Firstly we created functions to calculate the TF values based on weights provided:

```
def tfval(doc , word):
    n = len(doc)
    freq = len([token for token in doc if token == word])
    return freq/n

def tfraw(doc, word):
    freq = len([token for token in doc if token == word])
    return freq

def tflognorm(doc, word):
    freq = len([token for token in doc if token == word])
    return np.log(1 + freq)

def tfbin(doc, word):
    freq = 0
    for token in doc:
        if token == word:
            freq = 1
            break
    return freq

def tfdoublenorm(doc , word):
    freq = len([token for token in doc if token == word])
    maxfreq = 0
    res = max(set(doc), key = doc.count)
    for x in doc:
        if x == res:
            maxfreq += 1
    return 0.5 + (0.5 * (freq/maxfreq))
```

Each returns a singular term value based on the word provided.
Then we calculate the idf value as well using another function.

```
def idfval(word):
    try:
        freq = word_count[word] + 1
    except:
        freq = 1
    return np.log(total_documents/freq)
```

Then we calculate the final tf_idf values for our matrix.

```
def tf_idf(sentence):
    tf_idf_vec_Bin = np.zeros((len(word_set),))
    tf_idf_vec_Raw = np.zeros((len(word_set),))
    tf_idf_vec_TF = np.zeros((len(word_set),))
    tf_idf_vec_Log = np.zeros((len(word_set),))
    tf_idf_vec_Doub = np.zeros((len(word_set),))
    for word in sentence:

        #calculating all tfs
        tfVal = tfval(sentence,word)
        tfRaw = tfraw(sentence, word)
        tfBin = tfbin(sentence, word)
        tfLog = tflognorm(sentence, word)
        tfDoub = tfdoublenorm(sentence, word)

        # calculating idf
        idf = idfval(word)

        #final values
        valuetf = tfVal*idf
        valueRaw = tfRaw*idf
        valueBin = tfBin*idf
        valueLog = tfLog*idf
        valueDoub = tfDoub*idf

        #vectors
        tf_idf_vec_Bin[index_dict[word]] = valueBin
        tf_idf_vec_Raw[index_dict[word]] = valueRaw
        tf_idf_vec_TF[index_dict[word]] = valuetf
        tf_idf_vec_Log[index_dict[word]] = valueLog
        tf_idf_vec_Doub[index_dict[word]] = valueDoub
    return tf_idf_vec_Bin, tf_idf_vec_Raw, tf_idf_vec_TF, tf_idf_vec_Log, tf_idf_vec_Doub
```

Then we calculate the matching score by based on the sum of the tf_idf values found in the matrix wrt. The query.
Finally based on these scores we find the top 5 documents and show them.

```
top 5 docs for Binary weights:
243 : 820.0399351371057
1312 : 747.3307025595849
343 : 733.893762426369
791 : 731.2613659033082
797 : 725.6171926847227

_____________________________
top 5 docs for Raw weights:
1312 : 1210.3641710911859
243 : 1121.1338430351122
328 : 1092.8616694789423
797 : 1044.1977916289654
720 : 988.94212321471

_____________________________
top 5 docs for TF weights:
470 : 6.145615226935241
994 : 6.145615226935241
717 : 4.958400315875565
1167 : 4.694335078809936
82 : 4.585627064778936

_____________________________
top 5 docs for Log weights:
243 : 652.5214598318863
1312 : 651.2275901756898
797 : 603.5084661623157
791 : 568.8810713220656
343 : 562.9779114116719

_____________________________
top 5 docs for Double weights:
243 : 445.0554001634001
343 : 441.99054536789384
791 : 440.8109389867688
797 : 415.0184859238093
1312 : 403.9244555570726
```

Jaccard Coefficient

We create functions to calculate the union and intersection for the tokens and documents

```python
def union(tokens , doc):
    unionlist = list(tokens) + list(doc)
    unionset = set(unionlist)
    return len(unionset)

def intersect(tokens , doc):
    intersect = 0
    for x in tokens:
        if x in doc:
            intersect += 1
    return intersect
```

Then we calculate the jaccard coefficient with the jaccard coefficients

```
def jaccard(tokens , doc):
    jaccardval = intersect(tokens,doc) / union(tokens, doc)
    return jaccardval
```

Now we sort the final documents based on the jaccard coefficients and print them.

```
top 10 docs using Jaccard coefficent :
cranfield0798 : 0.022853957636566332
cranfield1313 : 0.022408026755852843
cranfield0792 : 0.021181716833890748
cranfield1040 : 0.020958751393534
cranfield0244 : 0.02073578595317726
cranfield0329 : 0.02040133779264214
cranfield0014 : 0.01950947603121516
cranfield0576 : 0.01950947603121516
cranfield0344 : 0.0189520624303233
cranfield0721 : 0.018283166109253065
```

2.Naive Bayes Classifier with TF-ICF

Dataset provided:

BBC_NEWS_train csv file which contains 3 columns: ArticleId, Text, Category
ArticleId: [1,1490] article Ids
Text: contains the text of the Article document
Category: [business, tech, politics, sport, entertainment]
Dataset size is 1490 X 3

Task 1: Preprocessing the dataset

(i) As part of preprocessing following steps were performed, in the same order removing punctuation, stop words, and converting all text to lowercase. Tokenize the text by splitting it into words, performing lemmatization to reduce words to their root form.

```
stop = stopwords.words('english')
df['Text_woStopwords'] = df['Text'].apply(lambda x: ' '.join([word for
word in x.split() if word not in (stop)]))
df['Text1'] = df['Text_woStopwords'].str.replace('[^\w\s]','')
df['Text1'] = df['Text1'].str.lower()
df['Text1'] = df.apply(lambda row: nltk.word_tokenize(row['Text1']),
axis=1)
df = df.drop(['Text_woStopwords'], axis=1)
cnt = 0
```

```
for i in df['Text1']:
  lst = []
  for j in i:
    lst.append(lemmatizer.lemmatize(j))
  df.at[cnt,'Text'] = lst
  cnt = cnt+1
df = df.drop(['Text1'], axis=1)
```

(ii) Implement the TF-ICF weighting scheme - below operations were performed

1. Convert the df dataframe to a dictionary di, where the **keys are the categories from the dataset** and the **values are a list of terms present across all the documents under that category**. *If a term 'xyz' appears in the two articles under a category c1, once in article1 and twice in article2 then the term 'xyz' will be present thrice in the list.*

```
arr = df.to_numpy() #converting the dataframe to numpy arrays
di = {} #final dictionary of list, which
        #has the keys as the news cateogories and values as the list
of dictionary(key=Article Id, Values= list of tokens)
for i in arr:
  li =i[1]
  di_t = {} # creating a temporary dictionary
  di_t[i[0]] = i[1]
  try:
    di[i[2]].append(li)
  except KeyError:
      di[i[2]] = li
#checking the keys of the dictionary
print(di.keys())
print(di['tech'])
```

2. Calculating tf or term frequency, it is calculated as the number of occurrences of a term in all documents of a particular class/category. A dictionary tf_idf is created where **keys are the distinct terms across the dataset** and **value is a list of size 5** where a value at **position 0 of list** is the tf of the key in class **'business'**, value at **position 1** of list is the tf of the key in class **'tech'**, value at **position 2** of list is the tf of the key in class **'politics'**, value at **position 3** of list is the tf of the key in class **'sport'** and value at **position 4** of list is the tf of the key in class **"entertainment"**.

```
#calculating the term frequency of each of the terms classwise,
```

```
tf_dict = {} #the following dictionary has the keys as the vocab of
the dataset (Text column) and the values as a list of size 5
            #where value at the 0th position of the list denotes the
number of times the term occured in the class business
            #where value at the 1sh position of the list denotes the
number of times the term occured in the class tech
            #where value at the 2nd position of the list denotes the
number of times the term occured in the class politics
            #where value at the 3rd position of the list denotes the
number of times the term occured in the class sport
            #where value at the 4th position of the list denotes the
number of times the term occured in the class entertainement
for key in di.keys():
  for items in di[key]:
    for i in items:
      if tf_dict.get(i) is None:
        tf_dict[i] = [0, 0, 0, 0, 0]
      if key == 'business':
        tf_dict[i][0] = tf_dict[i][0] + 1
      elif key == 'tech':
        tf_dict[i][1] = tf_dict[i][1] + 1
      elif key == 'politics':
        tf_dict[i][2] = tf_dict[i][2] + 1
      elif key == 'sport':
        tf_dict[i][3] = tf_dict[i][3] + 1
      elif key == 'entertainment':
        tf_dict[i][4] = tf_dict[i][4] + 1
print(tf_dict['ageing']) #printing a sample term frequency class
wise
print(len(tf_dict.keys())) #printing the number of distinct terms
```

3. Calculating class frequency (CF): Number of classes in which that term occurs. Dictionary cf_dict contains the keys as distinct terms across the dataset and values are the CF value of the corresponding key. Below is the code:

```
#calculating class frequency
cf_dict ={}
for key in tf_dict.keys():
  cf =0
```

```
  for v in tf_dict[key]:
    if v > 0:
      cf = cf + 1
  cf_dict[key] = cf
print(cf_dict['ageing'])
```

4. Calculating Inverse Class Frequency (ICF): log(N / CF), where N represents the number of classes (log 10). Dictionary cf_dict contains the keys as distinct terms across the dataset and values are the ICF value of the corresponding key Below is the code for the same

```
#calculating the inverse class frequency
for key in cf_dict.keys():
  cf_dict[key] = math.log(5/cf_dict[key],10)
print(cf_dict['ageing'])
```

5. Calculating TF_ICF: TF*ICF for each term per category. The dictionary tf_dict list value contains the tf-icf values.

```
#calculating the tf-icf by multipling the ICF values of the term to
each class's frequency
for key in tf_dict.keys():
  tf_dict[key][0] = cf_dict[key] * tf_dict[key][0]
  tf_dict[key][1] = cf_dict[key] * tf_dict[key][1]
  tf_dict[key][2] = cf_dict[key] * tf_dict[key][2]
  tf_dict[key][3] = cf_dict[key] * tf_dict[key][3]
  tf_dict[key][4] = cf_dict[key] * tf_dict[key][4]
print(tf_dict['ageing'])

#now the tf_dict has the final tf-icf values which now have to be
converted to a dataframe
```

6. The tf-icf values for each term per category is stored in the dictionary tf_dict

Task 2: Splitting the Dataset

1. The dataset which will be trained using MultinomialNB has the columns as all the terms of the preprocessed version of the original dataset and its values as tf-icf score for the the term corresponding to the category (if the

respective row's article contains the term otherwise 0). Below is the code for the same

```
#creating a dataframe wrt to the terms' existence in an article
corresponding to a category
#creating a dictionary based on the terms and dictionary
tf_icf_dict = {}
for key in sorted_tf_dict.keys():
  li = []
  for i in range(1490):
    if key in df['Text'][i]:
      if df['Category'][i] == 'business':
        li.append(sorted_tf_dict[key][0])
      elif df['Category'][i] == 'tech':
        li.append(sorted_tf_dict[key][1])
      elif df['Category'][i] == 'politics':
        li.append(sorted_tf_dict[key][2])
      elif df['Category'][i] == 'sport':
        li.append(sorted_tf_dict[key][3])
      elif df['Category'][i] == 'entertainment':
        li.append(sorted_tf_dict[key][4])
    else:
      li.append('0')
  tf_icf_dict[key] = li
```

2. Split the data into a 70-30 train-test ratio. Below is the code:

```
X_train, X_test, y_train, y_test = train_test_split(tf_icf_dft, target,
test_size=0.3, random_state=42)
```

Task 3: Training the Naive Bayes classifier with TF-ICF:

1. Trained the Multinomial Naive Bayes model over train data which is tf-icf weighted. Below is the code:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
gnb = MultinomialNB()
y_pred = gnb.fit(X_train, y_train).predict(X_test)
```

2. Calculated the probability of each category based on the frequency of documents in the training set that belong to that category. Below is the code:

```
# Calculate the probability of each category based on the frequency
of documents
# in the training set that belong to that category.
count = y_train.value_counts()
prob_b = count['business']/count.sum()
prob_t = count['tech']/count.sum()
prob_p = count['politics']/count.sum()
prob_s = count['sport']/count.sum()
prob_e = count['entertainment']/count.sum()
from tabulate import tabulate
data = [['count',count['business'], count['tech'],
count['politics'], count['sport'], count['entertainment']],
['probability',prob_b,prob_t ,prob_p ,prob_s ,prob_e ]]
print(tabulate(data, headers=["Type of Values:","business", "tech",
"politics", "sport","entertainment"]))
```

Output:

```
Type of Values: business tech  politics  sport  entertainment
----------------  ----------  ----------  ----------  ----------
count            228     188   188         245       194
probability    0.2186  0.180249 0.180249 0.234899   0.186002
```

3. Calculated the probability of each feature given each category based on the TF-ICF values of that feature in documents belonging to that category. Below is the code:

```
#Calculate the probability of each feature given each category based
on the
# TF-ICF values of that feature in documents belonging to that
category.
#formula applied is
df_iii = pd.DataFrame.from_dict(sorted_tf_dict)
df_iii.head() #where row 0= business, row 1 = tech, row2 = politics,
row3 = sport, row4 = entertainment
# considering: the probability of a feature given a category =
df_iii[term][row number as per category]
#.e.g. the probability of zombie given a category
df_iii['zombie'][1], thus the probability of a feature given each
category
# = sum of the values in that feature's column
```

```
prob_0 = df_iii.sum()/df_iii.iloc[0].sum()
prob_1 = df_iii.sum()/df_iii.iloc[1].sum()
prob_2 = df_iii.sum()/df_iii.iloc[2].sum()
prob_3 = df_iii.sum()/df_iii.iloc[3].sum()
prob_4 = df_iii.sum()/df_iii.iloc[4].sum()
prob_0, prob_1, prob_2, prob_3, prob_4
```

Task 4: Testing the Naive Bayes classifier with TF-ICF:

1. Use the testing set to evaluate the performance of the classifier. Below is the code:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
gnb = MultinomialNB()
y_pred = gnb.fit(X_train, y_train).predict(X_test)
acc_score = accuracy_score(y_test, y_pred)
print(acc_score)
```

output : `0.9977628635346756`

2. Classification report code:

```
print(classification_report(y_test, y_pred,target_names=['business',
'tech','politics','sport','entertainment']))
```

Output:

```
              precision    recall  f1-score   support

     business       1.00      0.99      1.00       108
         tech       1.00      1.00      1.00        79
     politics       1.00      1.00      1.00        86
        sport       1.00      1.00      1.00       101
entertainment       0.99      1.00      0.99        73

     accuracy                           1.00       447
    macro avg       1.00      1.00      1.00       447
 weighted avg       1.00      1.00      1.00       447
```

Task 5: Improving the classifier:

The performance of the tf-icf weighted NB has shown an accuracy of 99% at the split of 70-30. We further tried to improve the result of the same by following results:

1. Experimented with different splits like (60-40,80-20,50-50).Below is the code:

```
# Experiment with different preprocessing techniques and parameters
to improve
# the performance of the classifier( including different splits like
60-40,80-20,
# 50-50).

#with different splits:
X_train1, X_test1, y_train1, y_test1 = train_test_split(tf_icf_dft,
target, test_size=0.4, random_state=42) #60-40
X_train2, X_test2, y_train2, y_test2 = train_test_split(tf_icf_dft,
target, test_size=0.2, random_state=42) #80-20
X_train3, X_test3, y_train3, y_test3 = train_test_split(tf_icf_dft,
target, test_size=0.5, random_state=42) #50-50
```

```
#training on each split data:
gnb = MultinomialNB()
y_pred1 = gnb.fit(X_train1, y_train1).predict(X_test1)
y_pred2 = gnb.fit(X_train2, y_train2).predict(X_test2)
y_pred3 = gnb.fit(X_train3, y_train3).predict(X_test3)
acc_score1 = accuracy_score(y_test1, y_pred1)
acc_score2 = accuracy_score(y_test2, y_pred2)
acc_score3 = accuracy_score(y_test3, y_pred3)
print("accuracy with 70-30 split:", acc_score)
print("accuracy with 60-40 split:", acc_score1)
print("accuracy with 80-20 split:", acc_score2)
print("accuracy with 50-50 split:", acc_score3)
```

Output:

```
accuracy with 70-30 split: 0.9977628635346756
accuracy with 60-40 split: 0.9983221476510067
accuracy with 80-20 split: 1.0
accuracy with 50-50 split: 0.9986577181208054
```

2. Experimented with preprocessing techniques: like removed the digits, removed the single length characters and tried it on different splits, the accuracy score was exactly same as the ones previously calculated,

```
X_trainp, X_testp, y_trainp, y_testp = train_test_split(tf_icf_dft,
target, test_size=0.3, random_state=42) #70-30
y_predp = gnb.fit(X_trainp, y_trainp).predict(X_testp)
acc_scorep = accuracy_score(y_testp, y_predp)
print("accuracy with 70-30 split:", acc_score)
print("accuracy with 60-40 split:", acc_score1)
print("accuracy with 80-20 split:", acc_score2)
print("accuracy with 50-50 split:", acc_score3)
print("accuracy with 70-30 split on differently preprocessed data:",
acc_scorep)

#trying it on 50-50 split
X_trainp1, X_testp1, y_trainp1, y_testp1 =
train_test_split(tf_icf_dft, target, test_size=0.5, random_state=42)
#50-50
y_predp1 = gnb.fit(X_trainp1, y_trainp1).predict(X_testp1)
acc_scorep1 = accuracy_score(y_testp1, y_predp1)
print("accuracy with 50-50 split on differently preprocessed data:",
acc_scorep1)
#see no difference
```

Output:

```
accuracy with 70-30 split: 0.9977628635346756
accuracy with 60-40 split: 0.9983221476510067
accuracy with 80-20 split: 1.0
accuracy with 50-50 split: 0.9986577181208054
accuracy with 70-30 split on differently preprocessed data:
0.9977628635346756
accuracy with 50-50 split on differently preprocessed data:
0.9986577181208054
```

3. Tried using different types of features such as TF-IDF weights. And we saw the result to be nearly the same. Below is the code where TF-IDF:

```
tf_vectorizer = CountVectorizer() # or term frequency
X_train_tf = tf_vectorizer.fit_transform(train_X)
X_train_tf1 = tf_vectorizer.fit_transform(train_X1)
X_train_tf2 = tf_vectorizer.fit_transform(train_X2)
X_train_tf3 = tf_vectorizer.fit_transform(train_X3)
```

```
X_test_tf = tf_vectorizer.transform(test_X)
X_test_tf1 = tf_vectorizer.transform(test_X1)
X_test_tf2 = tf_vectorizer.transform(test_X2)
X_test_tf3 = tf_vectorizer.transform(test_X3)
```

Post training the Naive Bayes, calculated the accuracy:

```
y_pred = naive_bayes_classifier.predict(X_test_tf)
# compute the performance measures
score1 = accuracy_score(test_y, y_pred)
score2 = accuracy_score(test_y1, y_pred1)
score3 = accuracy_score(test_y2, y_pred2)
score4 = accuracy_score(test_y3, y_pred3)
print("accuracy with tf-idf on 70-30 split:",score1)
print("accuracy with tf-idf on 60-40 split:",score2)
print("accuracy with tf-idf on 80-20 split:",score3)
print("accuracy with tf-idf on 50-50 split:",score4)
print("accuracy with tf-icf on 70-30 split:", acc_score)
```

Output:

```
accuracy with tf-idf on 70-30 split: 0.9798657718120806
accuracy with tf-idf on 60-40 split: 0.9983221476510067
accuracy with tf-idf on 80-20 split: 1.0
accuracy with tf-idf on 50-50 split: 0.9986577181208054
accuracy with tf-icf on 70-30 split: 0.9977628635346756
```

4. Calculated the average accuracy score across various different tf-idf weighted trained models and tf-icf weighted trained models and the output was as follows:

```
average accuracy of tf-icf over different splits 0.9986856823266219
average accuracy of tf-idf over different splits 0.9942114093959731
```

Conclusion:

tf-icf and tf-idf weighted Naive-Bayes performed almost similar. Each of them provided a 99% average accuracy

Multinomial NB is well suited out of all the NB variations because of its efficiency in handling high dimensional data with the assumption that the features are conditionally independent given the class label. Moreover, MNB can handle feature counts that are zero, which is common in text data when some words do not appear in a particular document.

The reason behind such high accuracy is that the tf-icf values were calculated over the entire dataset before the splitting of it, thus on testing over the test data,

the terms were known and tf-icf values could in a sense provide a mapping between it and the category.
tf-icf performed slight better as the tf-icf formula took the category into consideration

3. Ranked Information Retrieval and Evaluation

Dataset provided:

The datasets are machine learning data in which IDs represent queries and URLs. The datasets consist of feature vectors extracted from query-URL pairs and relevant judgment labels.

Task 1: Focus only on the queries with qid:4 and use the relevance judgment labels as the relevance score

To do the following task, the steps followed as follows:

- Import all the dataset files and append these to store them in a list ‘rel’
- Fetched queries with only qid:4 and kept it in data frame ‘data’

	0	1	2	3	4	5	6	7	8	9	...	129	130	131	132	133	134	135	136	137	138
0	0	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	...	128:2	129:9	130:124	131:4678	132:54	133:74	134:0	135:0	136:0	NaN
1	0	qid:4	1:3	2:0	3:3	4:0	5:3	6:1	7:0	8:1	...	128:0	129:8	130:122	131:508	132:131	133:136	134:0	135:0	136:0	NaN
2	0	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	...	128:2	129:8	130:115	131:508	132:51	133:70	134:0	135:0	136:0	NaN
3	0	qid:4	1:3	2:0	3:3	4:0	5:3	6:1	7:0	8:1	...	128:82	129:17	130:122	131:508	132:83	133:107	134:0	135:10	136:13.35	NaN
4	1	qid:4	1:3	2:0	3:3	4:0	5:3	6:1	7:0	8:1	...	128:11	129:8	130:121	131:508	132:103	133:120	134:0	135:0	136:0	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
98	0	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	...	128:35	129:1	130:153	131:4872	132:9	133:55	134:0	135:0	136:0	NaN
99	1	qid:4	1:3	2:0	3:3	4:2	5:3	6:1	7:0	8:1	...	128:367	129:6	130:153	131:2383	132:18	133:99	134:0	135:16	136:11.3166666666667	NaN
100	2	qid:4	1:2	2:0	3:2	4:0	5:2	6:0.666667	7:0	8:0.666667	...	128:0	129:0	130:49182	131:26966	132:15	133:69	134:0	135:193	136:21.9355595468361	NaN
101	1	qid:4	1:2	2:0	3:2	4:0	5:2	6:0.666667	7:0	8:0.666667	...	128:0	129:1	130:42877	131:26562	132:12	133:24	134:0	135:56	136:62.9206042323688	NaN
102	0	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	...	128:1415	129:14	130:5334	131:6434	132:4	133:17	134:0	135:0	136:0	NaN

103 rows × 139 columns

- Our data frame contains only queries with qid:4 and uses the relevant judgment labels as their relevance scores.

Objective 1: "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".

- Our initial step to obtaining a DCG value involves sorting the complete data frame and saving it to the output.txt file.

```
# Calculating maxDCGdata and sorting the complete dataframe and saving it to the out.txt file.

maxDCGdata = df.sort_values(0,ascending=False)
maxDCGdata.to_csv("out.txt",index = False, sep = " ",  header=False)
```

- The next step would be to find no. of ways such files can be made in the order of maxDCG.

```
print(nWays(maxDCGdata))
```

```
19893497375938370599826047614905329896936840170566570588205180312704857992695193482412686565431050240000000000000000000000
```

Objective 2: "Compute the dataset's nDCG (normalized discounted cumulative gain). This involves calculating nDCG at position 50 and for the entire dataset."

- To calculate the nDCG, we'll first calculate DCG and IDCG and divide them according to the formula.
- DCG can be calculated using the formula given in the slides:

DCG is the total gain accumulated at a particular rank *p*:

$$DCG_p = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{\log_2 i}$$

- **nDCG for entire dataset and at position 50(taking only 50 relevant documents):**

```
#calculating nDCG for the entire dataset
print(DCG(list(data[0].values))/IDCG(list(data[0].values)))
```

```
0.5979226516897831
```

```
#calculating nDCG at position 50
rel_50 = list(df[0].values)[0:50]
print(DCG(rel_50)/IDCG(rel_50))
```

```
0.5253808413557646
```

Objective 3: "Assume a model that ranks URLs based on the value of feature 75, which represents the sum of TF-IDF on the whole document. URLs with higher feature 75 values are considered more relevant. Any non-zero relevance judgment value is considered relevant. Using this model, plot a Precision-Recall curve for the query "qid:4".

- To obtain the value of the 75th feature, we are splitting the dataset and adding the value at the first index to the relevant_features list.(Taking the 75th column of data and arrange it in order based on its importance or relevance.)
- To determine relevance scores, any score of 0 is considered non-relevant, and any non-zero score is considered relevant and given a score of 1.
- We are then sorting the values based on their feature relevance.
- To plot the curve and determine precision and recall, we will use a formula.
  Precision: fraction of retrieved docs that are relevant
  = P(relevant|retrieved)
  Recall: fraction of relevant docs that are retrieved
  = P(retrieved|relevant)

  Precision P = tp/(tp + fp)
  Recall R = tp/(tp + fn)

  Results:

  New URL Dataframe:

new_df

	relevance	feature relevance	Precision	Recall
0	0	90.531710	0.000000	0.000000
1	0	538.388954	0.000000	0.000000
2	0	88.171761	0.000000	0.000000
3	0	144.564444	0.000000	0.000000
4	1	142.589323	0.000000	0.000000
...	...	...	...	...
98	0	70.460443	0.414141	0.931818
99	1	270.132330	0.410000	0.931818
100	1	296.023694	0.415842	0.954545
101	1	528.520116	0.421569	0.977273
102	0	84.625987	0.427184	1.000000

103 rows × 4 columns

Precision-Recall curve: