Ques 1)

(i), (ii) Relevant Text Extraction and Data Preprocessing :
Data preprocessing from the extracted text from all the files is the same as the previous assignment and uses the same methods.

TF-IDF Matrix

In order to create the matrix of size No. of documents X No. of unique words in the preprocessed data corpus we generated prior, firstly we got the set of all the unique words in “vocab” .
Further we created a dictionary that maps each word to its index in the vocab called vocab_dict.
Then the matrix of size : no. of documents X vocab size is created whos values are initialized by zeroes.

As per the computation requirement, the Term Frequency, the frequency count of each term in every document is computed and stored as a nested dictionary for each document.
We are given 5 different term frequency weighting schemes. Accordingly, we created 5 different functions to determine the term frequencies of each of the word in vocab for all the documents.
The values are stored in the form of nested dictionaries.

1. def binary_tf(preproData, vocab):

This functions checks if the term is present in the current document, and if present, it sets the value as 1 and otherwise 0.
Returns a nested dictionary containing the term frequency having binary values. (either 0 or 1)

2. raw_count(preproData, vocab):

This function gives the term frequency values as raw_count expression. Which is the basic form of calculating the tf values, i.e. by dividing the no. of times a term occurs in a document divided by the total number of words in that document. This gives a normalized value for term frequency.

3. term_freq(preproData, vocab):

In this function, the term frequency calculated by the precious function is divided by the summation of all the term frequencies in the document minus the tf of the term.

4. log_normal(preproData, vocab):

Uses log(1+f(t,d)) function to get normalized scoring for very large documents.

5. double_normal(preproData, vocab):

Uses 0.5+0.5*(f(t,d)/ max(f(t‘,d)) function. More efficient for scoring for very large databases.

Now we needed the Inverse Document Frequency (IDF). For that we made an inverted index for the documents generating the postings list for each terms occurring in it, to determine the document frequencies for a term in the inverted index.
From the document frequencies, we calculated the IDF using the given formula :

IDF(term) = log(total number of documents/document frequency(term)+1)

Then a function is created to calculate the Tf-IDf scores which multiplies the term frequencies derived from above different functions by the idf of a given term for all the terms in vocab through each document.

TF-IDF value dictionaries are created for the different weighting schemes.
Then these TF-IDF values are filled in the Document X Vocabulary Matrix which was created initially. 5 TF-IDF matrices are generated.
For the Query, it is tokenized and the TF-IDF score is determined for each of the query term.
TF-IDF scoring is done using cosine-similarity indexing. 5 different TF-IDF matrices are used.

The Binary weighting scheme gives the least relevant scoring as the term frequency is only reflective of whether a word is present in the document or not. It does not take into account the frequency of a term in a document, which may lead to loss of information about the finer details of document content. Binary weighting is beneficial for for producing stable and consistent results as it is simple and easy to interpret for very large documents/datasets.

Raw count weighting is the basic method of calculating term frequencies. It accounts for the frequently occurring terms and will assign a higher tf score to the terms that reflect the importance of the word in a given document. The cons of raw count method is that it may lead to bias towards very large document. Even if the document might not be relevant, due to the no. of times a term occurs, it gets a higher tf-idf scoring. Important Rare terms are not noted well.

In Log Normalization method, log(1+f(t,d)) function helps to reduce the impact of high frequency terms, and does not let the terms in very large data/document overpower the more relevant terms. However, it may also lead to a loss of information for very low frequency terms, which might be relevant because they are rare and hold importance for a technical/specific query.

The double normalization with 0.5+0.5*(f(t,d)/ max(f(t‘,d)) function helps to reduce the impact of document length by giving higher weight to terms that occur frequently in a document but not in other documents, and encourages higher precision by giving higher weights to terms that are more discriminative. It disadvantages are that it only takes into account the frequency of a term in a document and its inverse frequency in the corpus, ignoring other important factors. It may not work well for very short documents, and may require tuning for different datasets or applications.

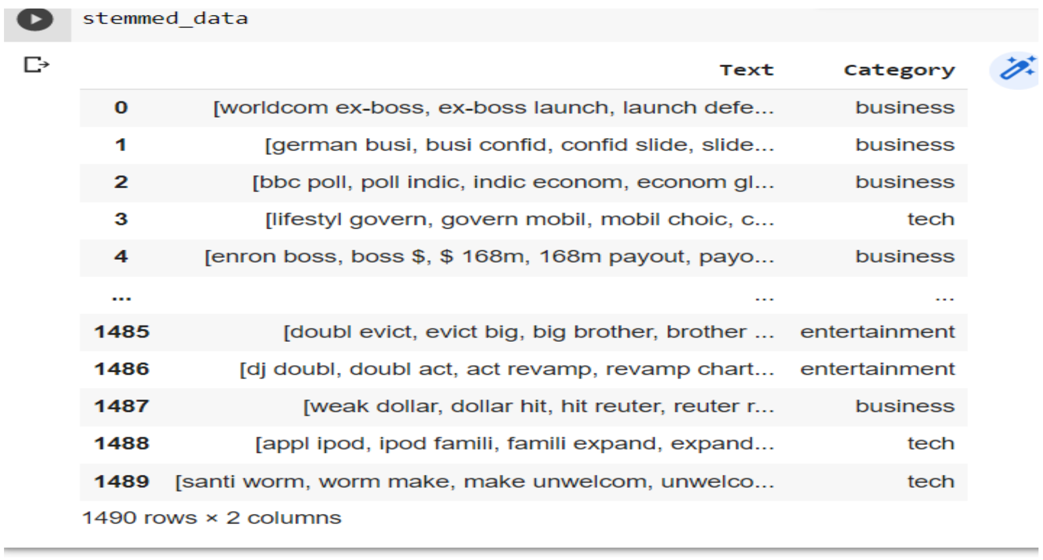
Jaccard Coefficient

The jaccard coefficient finds the most relevant to least relevant documents based on the scoring method where it takes the union of the query terms and the terms occurring in the preprocessed data. All the terms in the documents are collected in a set. Then we find the union between each query term and the document set.

Jaccard coefficient is found using the given formula. Then the jaccard coefficients of each document are added in a list. Sorted in descending order based on the scoring. Top ten documents correspond to the best to least scores are retrieved.

Ques 2)

Preprocessing the dataset:
The dataset provided is in the form of raw Text, which contains strings of terms.
To perform preprocessing, nltk library is imported.
Firstly, the article id is not relevant, so it is removed from the dataset. Next, punctuation, stop words etc are removed and all text is converted into lower text, using the function clean(text), perform cleaning of text, then the cleaned text is tokenized and the tokens are stemmed.



```
stemmed_data
```

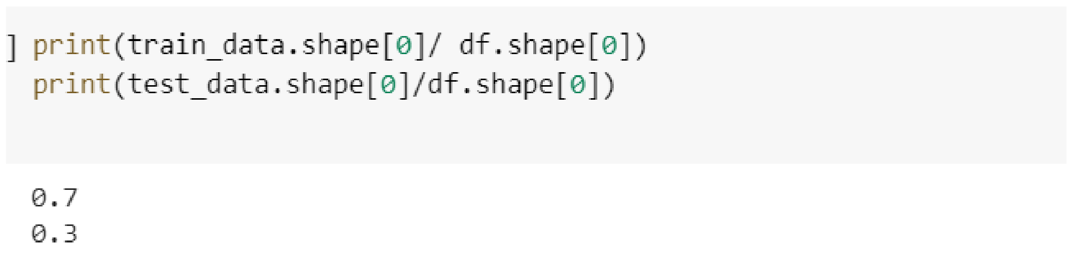
```
                                                   Text       Category
0            [worldcom ex-boss, ex-boss launch, launch defe...      business
1                [german busi, busi confid, confid slide, slide...      business
2               [bbc poll, poll indic, indic econom, econom gl...      business
3                [lifestyl govern, govern mobil, mobil choic, c...          tech
4            [enron boss, boss $, $ 168m, 168m payout, payo...      business
...                                                        ...           ...
1485            [doubl evict, evict big, big brother, brother ...  entertainment
1486          [dj doubl, doubl act, act revamp, revamp chart...  entertainment
1487               [weak dollar, dollar hit, hit reuter, reuter r...      business
1488           [appl ipod, ipod famili, famili expand, expand...          tech
1489   [santi worm, worm make, make unwelcom, unwelco...          tech
1490 rows × 2 columns
```

To implement the tf-icf weighting scheme, the tf-icf values are calculated for the documents.
The tf values for all the terms as per the category is calculated by taking a dictionary containing all the categories as keys, where each key corresponds to another dictionary. Then iterated through the dataframe and counted the number of occurrences of terms which are inserted into the sub dict of the corresponding category.

The icf values a class_freq={} is a dict where in the dataset, for each distinct term, an array is initialized and if the doc containing the term belongs to then that category is appended to the array of the term.
icf is calculated as log(N / CF), where N represents the number of classes, CF is n_t denotes the Number of classes in which that term occurs in a particular category class freq is Number of classes in which that

term occurs. To calculate the tf_icf values for a pair (term, category), a function tf_icf() is implemented which takes the tuple as the argument.
To calculate icf, math.log(N/n_t) is calc and the product term_freq*icf is returned.

Split the BBC train dataset into training and testing sets.
To perform the split, ‘random’ library is imported, and test_size =0.3 for training: testing ratio to be 0.7:0.3 Then sample function is applied on the df, after which train_data and test_data is extracted out of the dataframe.



```
] print(train_data.shape[0]/ df.shape[0])
  print(test_data.shape[0]/df.shape[0])
```

```
0.7
0.3
```

Training the Naive Bayes classifier with TF-ICF:

Implement the Naive Bayes classifier with the TF-ICF weighting scheme.
For this, a class NB is created, which has the following class variables, ‘tf_icf_terms’, ‘ prob_of_each_category’, ‘prob_of_each_term_for_category’.
The constructor of the class takes the argument dataset, and calculates the values of the class vars using the dataset.

1. Tf_icf_terms is calculated using the tf_icf weighting scheme, as explained above.
2. Prob_of_each_category is the dict that stores the probability of each category.
   Here the probability of each category is found by calculating the number of docs categorized as the category, decided by the number of docs in the dataset.
3. Prob_of_each_term_for_category takes (term, category) as additional arguments along with the train_data dataset. The probability of each term given a category is found by the foll formula:

   p(term, category)= tf_icf value of (term, category) / sum of tf_icf values of all the terms belonging to a category

   The numerator is found using the tf_icf matrix, and denominator is found by doing the summation of a column corresponding to that category in the matrix

   The classifier is trained in the trained data, so the variables are only calculated for the vocab in the train_data.
   The probability for the terms that are not present here but are in test_data, have to be 0.

Testing the Naive Bayes classifier with TF-ICF:

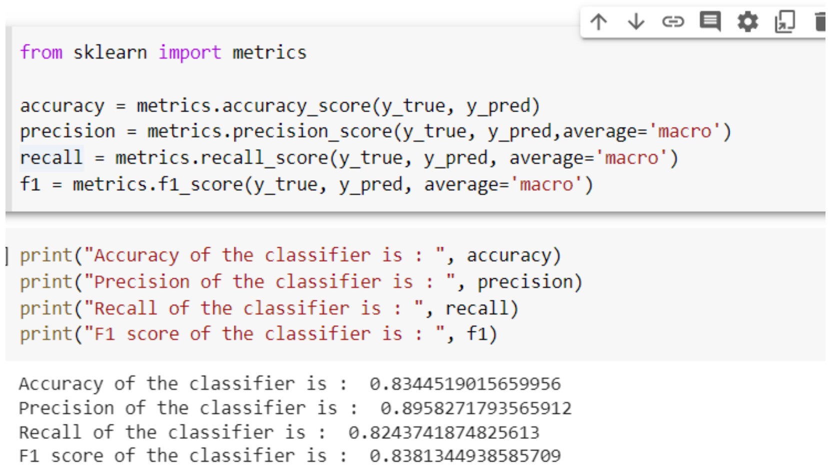
To test the classifier that is trained, with the train_data dataset, predict() function is used. Each doc in the test_data is passed as a vector of terms.
Here, the prediction is done by considering the num of naive bayes formula ,(since denom is constant for all the categories)

p(cat|x)= p(x1|cat_i).p(x2|cat_i)..p(xn|cat_i)p(cat_i)

Only the numerator is found, since comparative analysis is done, to perform mle.
The category which gets maximum p(cat|x) is considered as the final class of the given input vector of terms. Here the probabilities are already very small in magnitude, so the product is actually very small and converges to 0. Hence log probability is considered here
log(p1.p2…pn)=log(p1)+log(p2)..log(pn).
Also 1 is added to all the log terms to apply smoothing as there might be cases where probabilities are 0, but log(0) is not defined.

The predicted classes are stored in y_pred[] and the actual categ of docs are stored in y_true[]

Calculate the accuracy, precision, recall, and F1 score of the classifier.



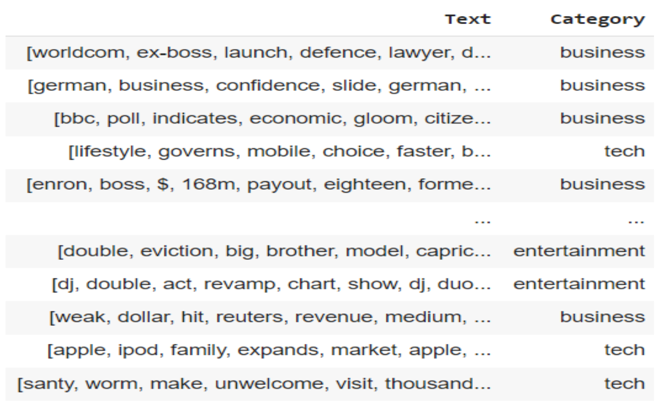
```
from sklearn import metrics

accuracy = metrics.accuracy_score(y_true, y_pred)
precision = metrics.precision_score(y_true, y_pred,average='macro')
recall = metrics.recall_score(y_true, y_pred, average='macro')
f1 = metrics.f1_score(y_true, y_pred, average='macro')
```

```
print("Accuracy of the classifier is : ", accuracy)
print("Precision of the classifier is : ", precision)
print("Recall of the classifier is : ", recall)
print("F1 score of the classifier is : ", f1)
```

```
Accuracy of the classifier is :  0.8344519015659956
Precision of the classifier is :  0.8958271793565912
Recall of the classifier is :  0.8243741874825613
F1 score of the classifier is :  0.8381344938585709
```

Improving the classifier

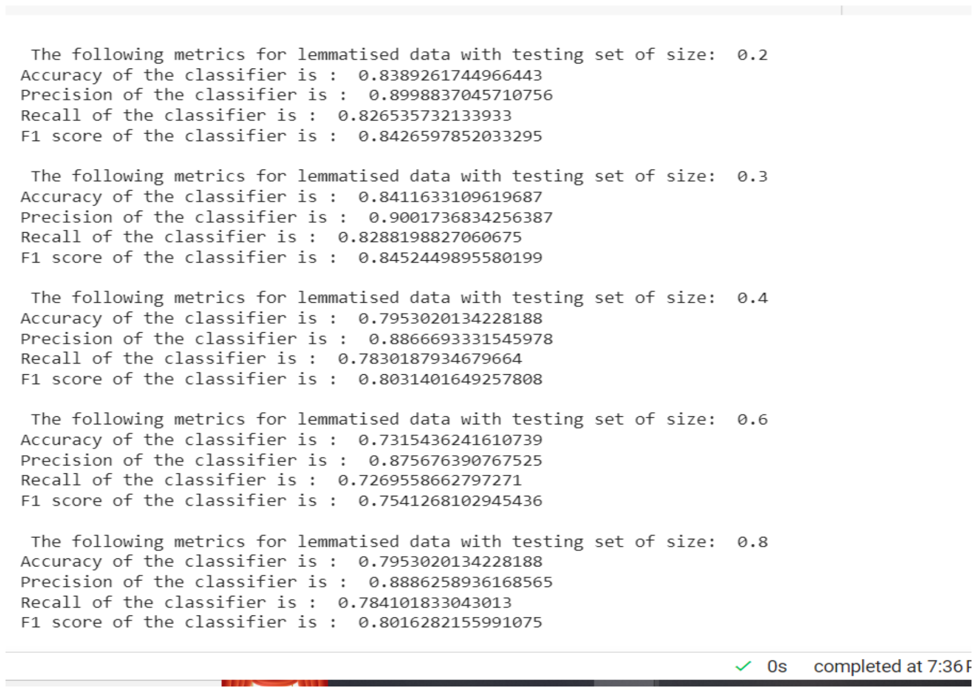
1. **try applying lemmatization instead of stemming:**



	Text	Category
	[worldcom, ex-boss, launch, defence, lawyer, d...	business
	[german, business, confidence, slide, german, ...	business
	[bbc, poll, indicates, economic, gloom, citize...	business
	[lifestyle, governs, mobile, choice, faster, b...	tech
	[enron, boss, $, 168m, payout, eighteen, forme...	business
	...	...
	[double, eviction, big, brother, model, capric...	entertainment
	[dj, double, act, revamp, chart, show, dj, duo...	entertainment
	[weak, dollar, hit, reuters, revenue, medium, ...	business
	[apple, ipod, family, expands, market, apple, ...	tech
	[santy, worm, make, unwelcome, visit, thousand...	tech

2. Trying different train-test splits

The train_test split is performed using the sklearn library, instead of doing sampling this time. The following test sizes were considered.
test_sizes=[0.2,0.3,0.4,0.6,0.8]



```
 The following metrics for lemmatised data with testing set of size:  0.2
Accuracy of the classifier is :  0.8389261744966443
Precision of the classifier is :  0.8998837045710756
Recall of the classifier is :  0.826535732133933
F1 score of the classifier is :  0.8426597852033295

 The following metrics for lemmatised data with testing set of size:  0.3
Accuracy of the classifier is :  0.8411633109619687
Precision of the classifier is :  0.9001736834256387
Recall of the classifier is :  0.8288198827060675
F1 score of the classifier is :  0.8452449895580199

 The following metrics for lemmatised data with testing set of size:  0.4
Accuracy of the classifier is :  0.7953020134228188
Precision of the classifier is :  0.8866693331545978
Recall of the classifier is :  0.7830187934679664
F1 score of the classifier is :  0.8031401649257808

 The following metrics for lemmatised data with testing set of size:  0.6
Accuracy of the classifier is :  0.7315436241610739
Precision of the classifier is :  0.875676390767525
Recall of the classifier is :  0.7269558662797271
F1 score of the classifier is :  0.7541268102945436

 The following metrics for lemmatised data with testing set of size:  0.8
Accuracy of the classifier is :  0.7953020134228188
Precision of the classifier is :  0.8886258936168565
Recall of the classifier is :  0.784101833043013
F1 score of the classifier is :  0.8016282155991075
```

✓ 0s completed at 7:36 F

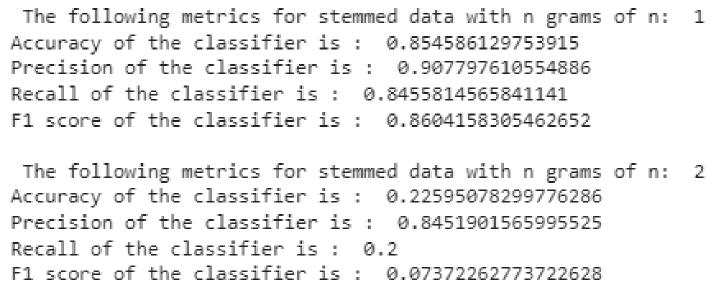
Observation: For lemmatised data, the accuracy value is maximum when the model has trained with training data of size 0.8 and testing data of 0.2 of the whole dataframe and gets reduced overall as the size of training set reduces, this can be explained as the corpus of training for the model is reduced, the probability of encountering a new word in the training set increases drastically, hence the model is not able to categorize the newer terms and overall the accuracy and other metrics reduce.

When comparing the technique stemming and lemmatization, stemming actually performs worse here as for the training test split of 0.7:0.3 the results of accuracy and other metrics for the lemmatised data were much better.

Hence we proceed with the lemmatised dataset, and a train test split of 0.7:0.3, since in the case of 0.8:0.2, there is a high chance that the model is actually overfitting instead of actually being that accurate.

Try using different types of features such as n-grams or TF-IDF weights.

The data's Text is converted into Bigrams and then train and test split is performed. The split ratio is 0.7:0.3



```
 The following metrics for stemmed data with n grams of n:  1
Accuracy of the classifier is :  0.854586129753915
Precision of the classifier is :  0.907797610554886
Recall of the classifier is :  0.8455814565841141
F1 score of the classifier is :  0.8604158305462652

 The following metrics for stemmed data with n grams of n:  2
Accuracy of the classifier is :  0.22595078299776286
Precision of the classifier is :  0.8451901565995525
Recall of the classifier is :  0.2
F1 score of the classifier is :  0.07372262773722628
```

Here, using bi-grams as features increase the dimensionality of the feature space, which can be a problem if the dataset is small, that is the case here. Hence the accuracy of the classifier has reduced drastically.

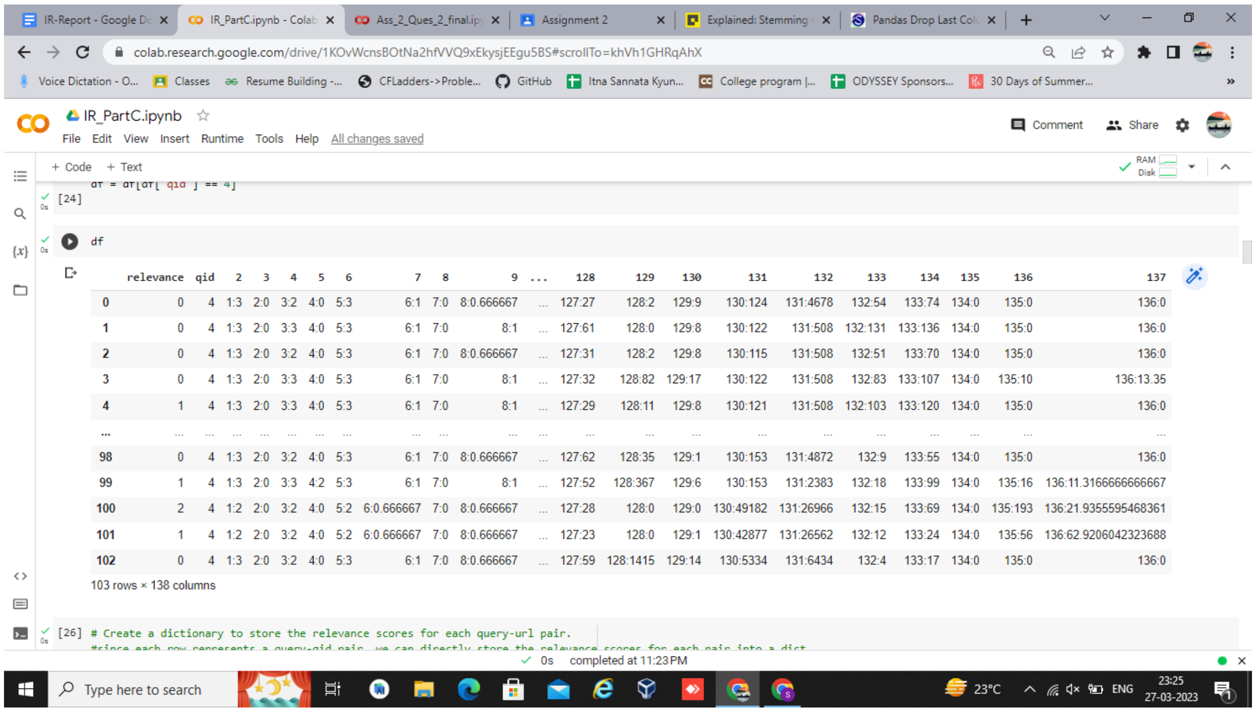
Conclusion

The stemmed data produced actually produced faster results than lemmatization but the accuracy of lemmatized data was actually better, mainly because it produces root words that are linguistically valid, while stemming simply removes the suffixes from words.

The train test split gave the best results for 0.8:0.2 ratio of train:test but this might be a case of overfitting, hence 0.7:0.3 is considered a better ratio. There was a decline in accuracy as the training dataset became smaller and testing became larger, in the case of 0.2:0.8 the accuracy is the minimum of all cases.

Next, the features are passed as bigrams to see if there is any improvement in the classifier performance, but it is actually worse than the previous case, since number of features (words) in the dataset is very high, using bigrams can increase the dimensionality of the feature space and the classifier harder to train.
Hence uni grams are actually a better feature choice for the classifier.

Q3;
MSLR data is cleaned to bring into a format like this:



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

```
df
```

	relevance	qid	2	3	4	5	6	7	8	9	...	128	129	130	131	132	133	134	135	136	137
0	0	4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	...	127:27	128:2	129:9	130:124	131:4678	132:54	133:74	134:0	135:0	136:0
1	0	4	1:3	2:0	3:3	4:0	5:3	6:1	7:0	8:1	...	127:61	128:0	129:8	130:122	131:508	132:131	133:136	134:0	135:0	136:0
2	0	4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	...	127:31	128:2	129:8	130:115	131:508	132:51	133:70	134:0	135:0	136:0
3	0	4	1:3	2:0	3:3	4:0	5:3	6:1	7:0	8:1	...	127:32	128:82	129:17	130:122	131:508	132:83	133:107	134:0	135:10	136:13.35
4	1	4	1:3	2:0	3:3	4:0	5:3	6:1	7:0	8:1	...	127:29	128:11	129:8	130:121	131:508	132:103	133:120	134:0	135:0	136:0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
98	0	4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	...	127:62	128:35	129:1	130:153	131:4872	132:9	133:55	134:0	135:0	136:0
99	1	4	1:3	2:0	3:3	4:2	5:3	6:1	7:0	8:1	...	127:52	128:367	129:6	130:153	131:2383	132:18	133:99	134:0	135:16	136:11.3166666666667
100	2	4	1:2	2:0	3:2	4:0	5:2	6:0.666667	7:0	8:0.666667	...	127:28	128:0	129:0	130:49182	131:26966	132:15	133:69	134:0	135:193	136:21.9355595468361
101	1	4	1:2	2:0	3:2	4:0	5:2	6:0.666667	7:0	8:0.666667	...	127:23	128:0	129:1	130:42877	131:26562	132:12	133:24	134:0	135:56	136:62.9206042323688
102	0	4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	...	127:59	128:1415	129:14	130:5334	131:6434	132:4	133:17	134:0	135:0	136:0

103 rows × 138 columns

```
# Create a dictionary to store the relevance scores for each query-url pair.
```

Each row of the dataset represents a query-url pair, and qid 4 is extracted from the dataset original.
The Assumption here is that the ranking of the urls as per the query is the same as the order in which each row appears in the dataset.

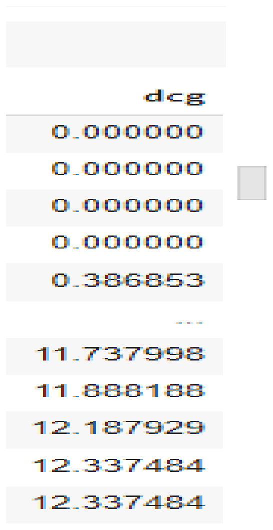
Create a dictionary to store the relevance scores for each query-url pair since each row represents a query-qid pair, we can directly store the relevance scores for each pair into a dict, each key in the dictionary represents a pair id of url and query.

Calculate the DCG scores for each query-url pair, using the relevance scores and the DG formula.

# Formula for calculating DG
a= relevance / np.log2(i+1), i is the ranking of the url.

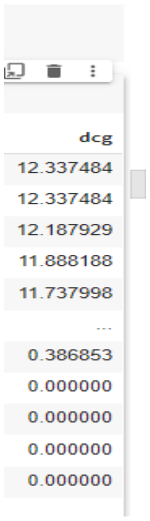
The cumulative DG or the DCG is calculated by taking the sum of the DG values of urls at a particular ranking. These are then appended to an array.

This array is then appended as a column to the dataframe.

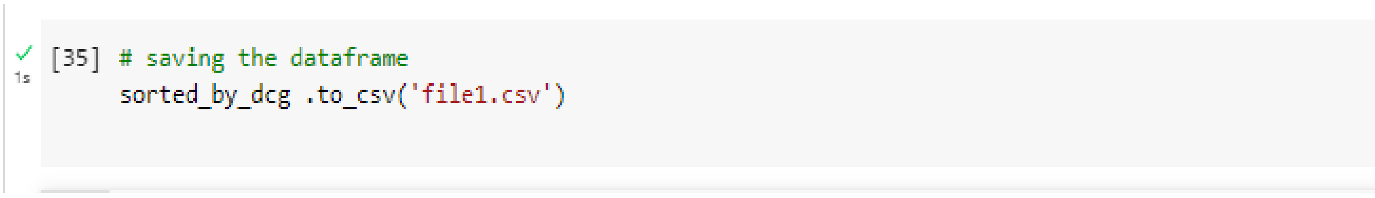


```
      dcg
 0.000000
 0.000000
 0.000000
 0.000000
 0.386853
      ...
11.737998
11.888188
12.187929
12.337484
12.337484
```

Now sorting the data frame based on the column dcg, use sort_values function in dec order.

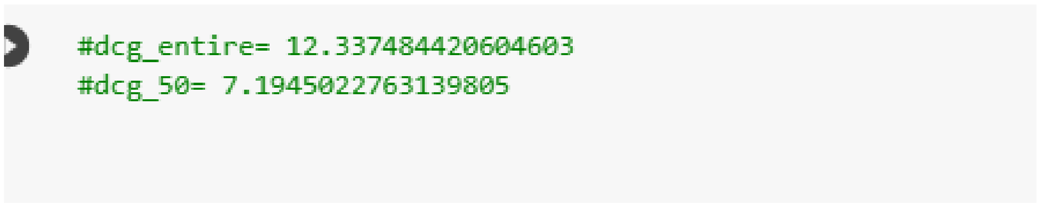


```
         dcg
   12.337484
   12.337484
   12.187929
   11.888188
   11.737998
         ...
    0.386853
    0.000000
    0.000000
    0.000000
    0.000000
```



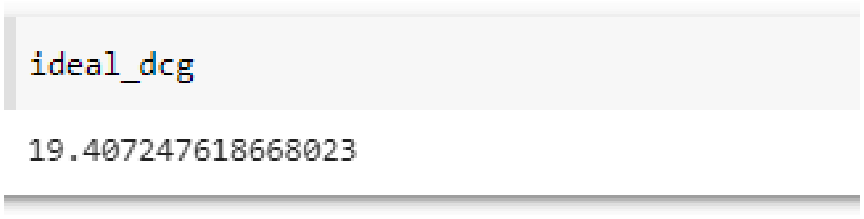
```
[35] # saving the dataframe
     sorted_by_dcg .to_csv('file1.csv')
```

Number of ways such files could be made : 12595179495187438268055552000
number of ways is equal to product of factorial of number of like terms in dcg array



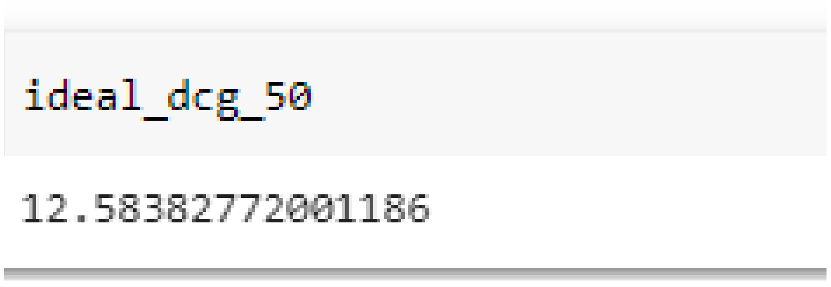
```
#dcg_entire= 12.337484420604603
#dcg_50= 7.1945022763139805
```

We calculate the ideal dcg for the dataset by sorting based on the relevance scores in dec order. Then in the order of ideal ranking, the dcg is calculated, ka the ideal dcg.



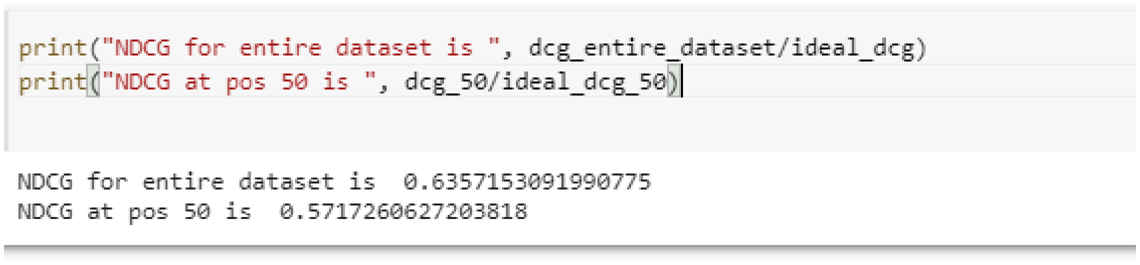
```
ideal_dcg

19.407247618668023
```

In case of dcg_50, only the top 50 rows of the original dataset are considered. We calculate the ideal dcg for the top 50 rows in dataset by sorting based on the relevance scores.



```
ideal_dcg_50
```

```
12.58382772001186
```

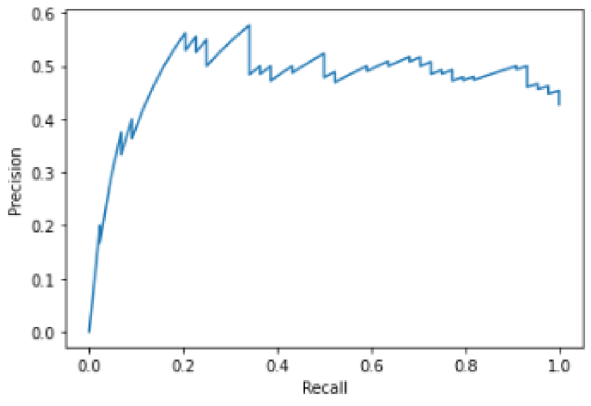


```
print("NDCG for entire dataset is ", dcg_entire_dataset/ideal_dcg)
print("NDCG at pos 50 is ", dcg_50/ideal_dcg_50)
```

```
NDCG for entire dataset is  0.6357153091990775
NDCG at pos 50 is  0.5717260627203818
```

Ndcg is calculated by dividing the dcg of current ranking by the ideal dcg

For the case where feature 75 is given preference, data is sorted by the values of the column representing this feature.
Then precision at k and recall at k are calculated and stored into arrays. The formula is referred from the lecture slides.



The above graph represents the curve bw precision and recall.
Precision is the fraction of retrieved documents that are relevant.
Recall is the fraction of relevant documents that are retrieved.
A high precision value means that most of the retrieved documents are relevant to the query, while a high recall value means that most of the relevant documents are retrieved by the system. It can be observed that Recall ultimately goes till 1 when almost all the relevant docs have been retrieved. To increase recall, we need to retrieve more relevant documents by setting a lower threshold for relevance; this may also retrieve more non-relevant documents, which results in a lower precision value. Thus there is a tradeoff between the two.
There is also a sawtooth pattern to be observed.

Reference: Lecture slides