IR-A2 REPORT - Group 22

Github link: https://github.com/swaib22078/CSE508_Winter2023_A2_Group_22

Q1.

Install the required packages.
Import required packages and download required files
Define the path of the directory containing the files to be processed.
Get a list of all the files in the directory and filter out the files with "cranfield" in their names.
Preprocessing
Extracting the required text and title in the docs and overwriting them
Converting the text to lowercase
Removing stopwords
Removing punctuations
Removing blank spaces

Importing packages & Preprocessing

```
[ ]  pip install Bs4

[ ] pip install nltk

[ ] import nltk
    nltk.download('punkt')
    import string
    from bs4 import BeautifulSoup
    import os
    nltk.download('stopwords')
    from nltk.corpus import stopwords
    stopword1=stopwords.words('english')

[ ] pip install requests

[ ] import requests
    path="C:/Users/Devanshu/Downloads/dummy_data original-20230319T121407Z-001/dummy_data original/"
    dir_list=os.listdir(path)
    print(path)

[ ] print(dir_list)
```

```
[ ] #Doing lower case
    i=0
    for i in range(len(dir_list)):
        file1 = open(dir_list[i],"r+")
        contents = file1.read()
        contents = contents.lower()
        file1.close()
        file1 = open(dir_list[i], "w")
        file1.seek(0)
        file1.write(contents)
        file1.truncate()
        file1.close()

    #Stopwords
    i=0
    j=0
    for i in range(len(dir_list)):
        file_content = open(dir_list[i],"r+")
        contents=file_content.read()
        tokens=nltk.word_tokenize(contents)
        contents = [w for w in tokens if not w in stopword1]
        str_wt_s=""
        for t in contents:
            str_wt_s+= t+ " "
```

```
[ ] #For removing punctuations
    i=0
    for i in range(len(dir_list)):
        file1= open(dir_list[i],"r+")
        contents = file1.read()
        contents =contents.translate(str.maketrans('', '', string.punctuation))
        file1.close()
        file1 = open(dir_list[i], "w")
        file1.seek(0)
        file1.write(contents)
        file1.truncate()
        file1.close()
```

```
[ ] #Removing blankspace
    i=0
    j=0
    for i in range(len(dir_list)):
        file1 = open(dir_list[i],"r+")
        without_blank = ""
        for j in file1:
            if not j.isspace():
                without_blank+=j
        file1.close()
```

Creating an inverted index dictionary with the keys as unique words and the values as a set of document ids where the word appears.Sorting the document ids for each word in the inverted index.Counting the frequency of each word in the inverted index.Saving the inverted index in a file using the pickle module.Loading the inverted index from the saved file using the pickle module.

Unigram index

```
[ ] import pickle

    inverted_index = {}
    i=0
    for i in range(len(dir_list)):
        file1 = open(dir_list[i],"r+")
        contents = file1.read()
        words = contents.split()
        for word in words:
            if word not in inverted_index:
                    inverted_index[word]={i+1}
            inverted_index[word].add(i+1)

    #sorting the doc_ids(value) for each word(key)
    inverted_index={key: sorted(value) for key, value in inverted_index.items()}
    inverted_index=dict(inverted_index)
    print(inverted_index)


    #counting the frequency of each word in inverted index
    for key,value in inverted_index.items():
        freq=len([item for item in value if item])
        inverted_index[key].append(freq)
    print("new line")
    print(inverted_index['frequency'])

    #saving inverted index in pickle
    with open('unigram_inverted_index.pickle', 'wb') as file:
        pickle.dump(inverted_index, file)

    #loading inverted index
    print("Unpickling the inverted index")
    with open('unigram_inverted_index.pickle', 'rb') as file:
        inverted_index_load = pickle.load(file)
        file.close()
    print(inverted_index_load)
```

Creating an inverted index and then calculating IDF for each word in the index. It then computes TF for each document using five different methods: simple TF, raw TF, binary TF,log normalization, double normalization. After that, the code calculates TF-IDF for each document using the IDF and the different TF measures.The final output is a set of TF-IDF scores for each document, calculated using the different methods.

Calculating the IDF

Calculating the IDF

```
[ ] #correct IDF
    unigram={}
    id1={}

    i=0
    for i in range(len(dir_list)):
       file1 = open(dir_list[i],"r+")
    #     contents = file1.read()
    for key,value in inverted_index_load.items():
     unigram[key]=len(set(value))
    id1[dir_list[i]]=unigram

    #print(id1)
```

```
[ ] root={}
    for i in range(len(dir_list)):
        file1 = open(dir_list[i],"r+")
        contents = file1.read()
        word_list=contents.split()
        temp={}
        for j in word_list:
            try:
                temp[j]=unigram[j]
            except:
                print("error",dir_list[i])
                pass
        root[dir_list[i]]=temp

    print(root)
```

Calculating all TF by different method

1).Calculating the TF-simple

Calculating the TF

```
#correct TF
from collections import Counter

i=0
d={}
for i in range(len(dir_list)):
    file1 = open(dir_list[i],"r+")
    contents = file1.read()
    content1=contents.split()
    #content1=list(set(content2))
    tf={}
    # Term frequency
    for doc in content1:
        tf_count=content1.count(doc)
        total_words = len(content1)
        tf[doc]=tf_count/total_words
    #print(tf)
    d[dir_list[i]]=tf
```

Calculating the TF-IDF Simple

Calculating the TF-IDF in simple

```python
#Doing TF-IDF simple

tf1={}
for j in df:
        # print(j)
    tf_idf={}
    for k in d[j]:
        tf_idf[k]=df[j][k]*d[j][k]
    tf1[j]=tf_idf
```

Printing the TF-IDF

```python
#printing TF-IDF
for i in tf1:
    print(i,tf1[i])
```

```
cranfield0001.txt {'experimental': 0.016719504303735207, 'investigation': 0.010515172751035176, 'aerodynamics': 0.022703740610470136, 'wing': 0.035576568517
cranfield0002.txt {'simple': 0.01583705058153557, 'shear': 0.021303427177165456, 'flow': 0.016711962087496632, 'past': 0.04384537333203407, 'flat': 0.024886
cranfield0003.txt {'boundary': 0.03380143684607054, 'layer': 0.03873679739302462, 'simple': 0.054934769204701514, 'shear': 0.07389626302079268, 'flow': 0.03
cranfield0004.txt {'approximate': 0.02100209272074287, 'solutions': 0.03701767522530582, 'incompressible': 0.048164573273441355, 'laminar': 0.01811097941051
cranfield0005.txt {'onedimensional': 0.05045476554190377, 'transient': 0.09016721780635674, 'heat': 0.09029739182687033, 'conduction': 0.08761966736905728,
cranfield0006.txt {'onedimensional': 0.032702162851233924, 'transient': 0.029220857622430423, 'heat': 0.043894565471395294, 'flow': 0.005725394418864585, 'm
cranfield0007.txt {'effect': 0.015538353728679363, 'controlled': 0.014725975966459429, 'threedimensional': 0.030560116221391175, 'roughness': 0.08698079638
cranfield0008.txt {'measurements': 0.011357124239396832, 'effect': 0.02614991237265551, 'twodimensional': 0.03368849099825763, 'threedimensional': 0.0342869
cranfield0009.txt {'transition': 0.030685590488033977, 'studies': 0.006906602493459776, 'skin': 0.013156976159566996, 'friction': 0.01360599077062943, 'meas
```

2).Calculating the TF by RAW

TF by RAW

```python
#Calculating TF By RAw
from collections import Counter

i=0
tf_raw={}
for i in range(len(dir_list)):
    file1 = open(dir_list[i],"r+")
    contents = file1.read()
    content1=contents.split()
    #content1=list(set(content2))
    tf={}
    # Term frequency
    for doc in content1:
        tf_count=content1.count(doc)
        #total_words = len(content1)
        tf[doc]=tf_count
    #print(tf)
    tf_raw[dir_list[i]]=tf
```

Calculating the TF-IDF by RAW

TF_IDF BY RAW

```python
#Doing TF-IDF byRAW

tf2_raw={}
for j in df:
        # print(j)
    tf_idf={}
    for k in tf_raw[j]:
        tf_idf[k]=df[j][k]*tf_raw[j][k]
    tf2_raw[j]=tf_idf
```

Printing the TF_IDF RAW

```python
#printing TF-IDF by RAw
for i in tf2_raw:
    print(i,tf2_raw[i])
```

```
cranfield0001.txt {'experimental': 1.2874018313876108, 'investigation': 0.8096683018297085, 'aerodynamics': 1.7481880270062005, 'wing': 2.739395775858253, ':
cranfield0002.txt {'simple': 1.7579126145504484, 'shear': 2.3646804166653657, 'flow': 1.8550277917121258, 'past': 4.866836439855781, 'flat': 2.762456261857
cranfield0003.txt {'boundary': 0.5408229895371286, 'layer': 0.619788758288394, 'simple': 0.8789563072752242, 'shear': 1.1823402083326828, 'flow': 0.61834259
cranfield0004.txt {'approximate': 0.9030899869919435, 'solutions': 1.5917600346881504, 'incompressible': 2.0710766507579783, 'laminar': 0.778772114652219l,
cranfield0005.txt {'onedimensional': 1.765916793966632, 'transient': 3.155852623222486, 'heat': 3.1604087139404613, 'conduction': 3.066688357917005, 'double
cranfield0006.txt {'onedimensional': 1.765916793966632, 'transient': 1.577926311611243, 'heat': 2.370306535455346, 'flow': 0.30917129861868764, 'multilayer'
cranfield0007.txt {'effect': 2.144292814557752, 'controlled': 2.0321846833714012, 'threedimensional': 4.217296038551982, 'roughness': 12.003349900634756, 'b
cranfield0008.txt {'measurements': 0.9312841876305402, 'effect': 2.144292814557752, 'twodimensional': 2.7624562618571256, 'threedimensional': 2.81153069236
cranfield0009.txt {'transition': 5.983690145166626, 'studies': 1.3467874862246563, 'skin': 2.565610351115564, 'friction': 2.653168200272739, 'measurements':
```

3).Calculating the TF by binary

Finding the TF_by Binary

```python
#Finding TF by binary

i=0
tf_binary={}

for i in range(len(dir_list)):
    file1 = open(dir_list[i],"r+")
    contents = file1.read()
    word_list=contents.split()
    temp={}
    for word in unique_words:
        temp[word]=0
        for k in word_list:
            if k==word:
                temp[word]=1
    tf_binary[dir_list[i]]=temp
```

Calculating the TF-IDF by Binary

Finding TF-IDF by Binary

```
#Doing TF-IDF for Binary

tf3_binary={}
for j in df:
        # print(j)
    tf_idf={}
    for k in tf_binary[j]:
        try:
          tf_idf[k]=df[j][k]*tf_binary[j][k]

        except:
            tf_idf[k]=0
    tf3_binary[j]=tf_idf
```

4).Calculating the TF by log Normalization

TF by log Normalization

```
#TF for log normalization

from collections import Counter
import math

i=0
tf_log_normalization={}
for i in range(len(dir_list)):
    file1 = open(dir_list[i],"r+")
    contents = file1.read()
    content1=contents.split()
    #content1=list(set(content2))
    tf={}
    # Term frequency
    for doc in content1:
        tf_count=math.log10(1+content1.count(doc))
        #total_words = len(content1)
        tf[doc]=tf_count
    #print(tf)
    tf_log_normalization[dir_list[i]]=tf
```

Calculating TF-IDF by log normalization

TF_IDF by Log Normalization

```
#Doing TF-IDF for log normalization
tf4_log_normalization={}
for j in df:
        # print(j)
    tf_idf={}
    for k in tf_log_normalization[j]:
        tf_idf[k]=df[j][k]*tf_log_normalization[j][k]

    tf4_log_normalization[j]=tf_idf
```

Printing by TF_IDF_by log Normalization

```
tf4_log_normalization
```

```
{'cranfield0001.txt': {'experimental': 0.30712338856600241,
  'investigation': 0.24373444538906017,
  'aerodynamics': 0.5262570341895003,
  'wing': 0.5497601990190255,
  'slipstream': 1.5563025007672873,
  'study': 0.29917555796128087,
  'propeller': 0.5490512403779657,
  'made': 0.2849009212292515,
  'order': 0.297347057301817,
```

5).Calculating the TF by double normalization

TF_Double Normalization

```
#TF for double normalization

from collections import Counter
m=0
tf_double_normalization={}

for i in range(len(dir_list)):
    file1 = open(dir_list[i],"r+")
    contents = file1.read()
    word_list=contents.split()
    for k in word_list:
        tf_count=word_list.count(k)
        m=max(m,tf_count)

    tf={}
    # Term frequency
    for doc in word_list:
        tf_count=word_list.count(doc)
        tf[doc]=0.5+0.5*(tf_count/m)
    tf_double_normalization[dir_list[i]]=tf
```

Calculating the TF-IDF by double normalization

TF_IDF for Double Normalization

```
#Doing TF-IDF for double normalization

tf5_double_normalization={}
for j in df:
        # print(j)
    tf_idf={}
    for k in tf_double_normalization[j]:
        tf_idf[k]=df[j][k]*tf_double_normalization[j][k]

    tf5_double_normalization[j]=tf_idf
```

The query "experimental investigation" is preprocessed to convert all characters to lowercase and tokenized into words. Stopwords are removed from the query tokens and only alphanumeric tokens are kept in a list called 'query'. .Two functions 'union' and 'intersection' are defined to find the union and intersection of two lists respectively. The Jacard coefficient is calculated between the 'query' and each file in the 'dir_list'.

Now we will do the preprocessing of the query

Query

```
stop_words = set(stopwords.words('english'))
```

```
#For query preprocessing
d="experimental investigation"
d_lower=d.lower()
nltk_tokens = nltk.word_tokenize(d_lower)

stop_words_removed = []
for w in nltk_tokens:
    if w not in stop_words:
        stop_words_removed.append(w)

query = []
for x in stop_words_removed:
    if(x.isalnum() and x!=" "):
        query.append(x)
```

```
query
```

Now we will print the first five files by all TF

1).**Printing first 5 file TF_IDF_by Term Frequency**

Printing first 5 file TF_IDF_by Term Frequency

```
tf_idf = {}
for doc in tf1:
    s = 0
    for q in query:
        try:
            s += tf1[doc][q]
        except:
            s=0
    tf_idf[doc] = s

sorted_items = sorted(tf_idf.items(), key=lambda item: item[1],reverse=True)
sorted_dict = {}

for key, value in sorted_items:
    sorted_dict[key] = value

first_five_pairs = list(sorted_dict.items())[:5]

for key, value in first_five_pairs:
    print(key, value)
```

```
cranfield0372.txt 0.06092470060453666
cranfield0549.txt 0.05802660306033 9035
cranfield0932.txt 0.05114805202969 0715
cranfield0339.txt 0.04844564058411713
cranfield0836.txt 0.04178360450129543
```

2).**Printing first 5 file TF_IDF_by RAW**

Printing first 5 file TF_IDF_by RAW

```
tf_idf_raw = {}
for doc in tf2_raw:
    s = 0
    for q in query:
        try:
            s += tf2_raw[doc][q]
        except:
            s=0
    tf_idf_raw[doc] = s

sorted_items = sorted(tf_idf_raw.items(), key=lambda item: item[1],reverse=True)
sorted_dict = {}

for key, value in sorted_items:
    sorted_dict[key] = value

first_five_pairs = list(sorted_dict.items())[:5]

for key, value in first_five_pairs:
    print(key, value)
```

```
cranfield0712.txt 3.8823741230126396
cranfield0372.txt 3.716406736876736
cranfield0442.txt 3.716406736876736
cranfield0522.txt 3.5504393507408 33
cranfield1225.txt 3.550439350740833
```

3).**Printing first 5 file TF_IDF_by Binary**

Printing first 5 file TF_IDF_by Binary

```
tf_idf_binary = {}
for doc in tf3_binary:
    s = 0
    for q in query:
        try:
            s += tf3_binary[doc][q]
        except:
            s=0
    tf_idf_binary[doc]= s

sorted_items = sorted(tf_idf_binary.items(), key=lambda item: item[1],reverse=True)
sorted_dict = {}

for key, value in sorted_items:
    sorted_dict[key] = value

first_five_pairs = list(sorted_dict.items())[:5]

for key, value in first_five_pairs:
    print(key, value)
```

```
cranfield0001.txt 1.4533692175235138
cranfield0019.txt 1.4533692175235138
cranfield0029.txt 1.4533692175235138
cranfield0030.txt 1.4533692175235138
cranfield0074.txt 1.4533692175235138
```

4).Printing first 5 file TF_IDF_by Log normalization

Printing first 5 file TF_IDF_by Log Normalization

```
tf_idf_log = {}
for doc in tf4_log_normalization:
    s = 0
    for q in query:
        try:
            s += tf4_log_normalization[doc][q]
        except:
            s=0
    tf_idf_log[doc]= s

sorted_items = sorted(tf_idf_log.items(), key=lambda item: item[1],reverse=True)
sorted_dict = {}

for key, value in sorted_items:
    sorted_dict[key] = value

first_five_pairs = list(sorted_dict.items())[:5]

for key, value in first_five_pairs:
    print(key, value)
```

```
cranfield0372.txt 0.7945922793381444
cranfield0442.txt 0.7945922793381444
cranfield0522.txt 0.7738565237961428
cranfield1225.txt 0.7738565237961428
cranfield0712.txt 0.7597071403008554
```

5).Printing first 5 file TF_IDF_by Double normalization

```
tf_idf_double = {}
for doc in tf5_double_normalization:
    s = 0
    for q in query:
        try:
            s += tf5_double_normalization[doc][q]
        except:
            s=0
    tf_idf_double[doc]= s

sorted_items = sorted(tf_idf_double.items(), key=lambda item: item[1],reverse=True)

sorted_dict = {}
for key, value in sorted_items:
    sorted_dict[key] = value

first_five_pairs = list(sorted_dict.items())[:5]

for key, value in first_five_pairs:
    print(key, value)
```

```
cranfield0001.txt 0.9363916220834889
cranfield0084.txt 0.8384822408789503
cranfield0179.txt 0.8384822408789503
cranfield0019.txt 0.8304966957277222
cranfield0712.txt 0.8288523488410369
```

JACARD COEFFICIENT

Jacard coefficient is a measure of similarity between two sets, which is calculated as the size of the intersection divided by the size of the union of the sets. A dictionary called 'jacard' is created to store the Jacard coefficient of each file in 'dir_list'. The keys are the filenames and the values are the Jacard coefficients.

Doing preprocessing for Query in Jaccard coefficient

Jacard Coefficient

```
stop_words = set(stopwords.words('english'))
```

```
#For query preprocessing
d="experimental investigation"
d_lower=d.lower()
nltk_tokens = nltk.word_tokenize(d_lower)

stop_words_removed = []
for w in nltk_tokens:
    if w not in stop_words:
        stop_words_removed.append(w)

query = []
for x in stop_words_removed:
    if(x.isalnum() and x!=" "):
        query.append(x)
```

Now doing the union and intersection to find the Jaccard coefficient for all files.

```
def union(lst1, lst2):
    final_list = lst1 + lst2
    return len(final_list)
```

```
def intersection(lst1, lst2):
    final_list=list(set(lst1) & set(lst2))
    return len(final_list)
```

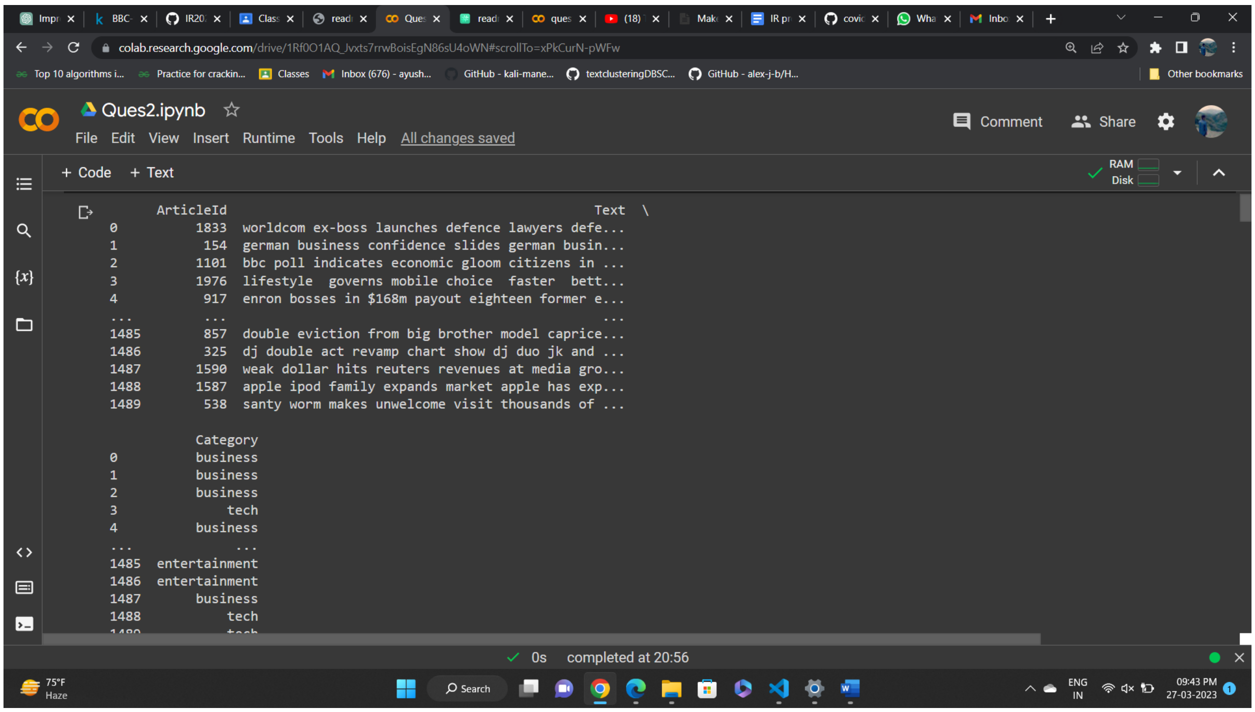
```
#Jacard coefficient

jacard={}
i=0
for i in range(len(dir_list)):
    file1 = open(dir_list[i],"r+")
    contents = file1.read()
    content1=contents.split()
    #content1=list(set(content2))
    union1=union(content1,query)
    intersection1=intersection(content1,query)
    p=intersection1/union1
    jacard[dir_list[i]]=p
```

```
print(jacard)
```

```
{'cranfield0001.txt': 0.02531645569620253, 'cranfield0002.txt': 0.0, 'cranfield0003.txt': 0.0,
```

Question 2:

1. *The aim of the question is to apply Naïve Bayes Classifier with TF-ICF weighting scheme.*
2. *First preprocessing is done like lowercase, tokenization, stopwords, punctuations,lemmatization etc.*



1. *Then tf is calculated for each category*
2. *Then cf is calculated for each category*
3. *Then implemented the TF-ICF weighting scheme.*
4. *Matrix for the dataset.*
5. *Train-test split is done before applying Naïve Bayes Classifier.*

After applying various variations in the Classifier we got following conclusions:

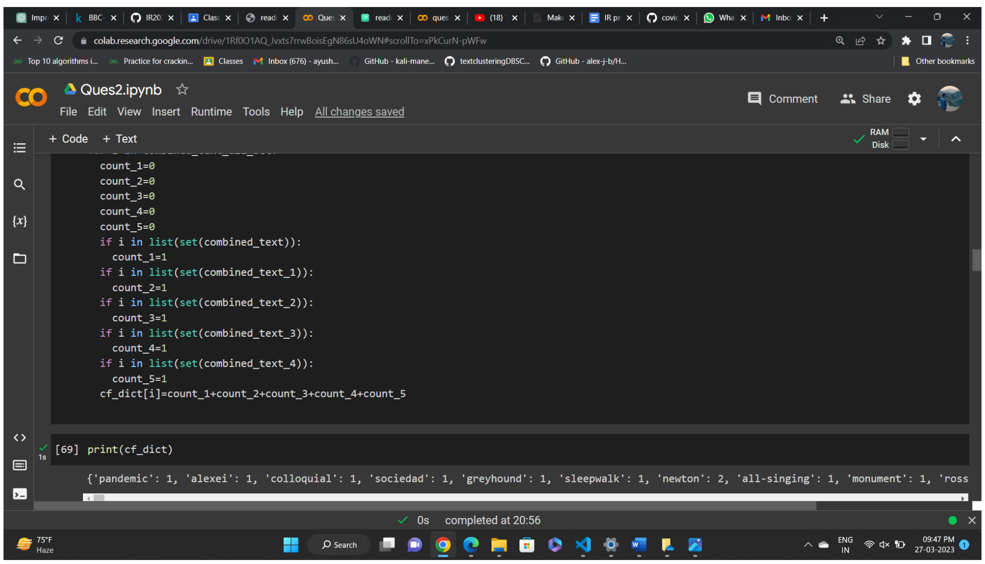
1.Maximum accurcacy came to be maximum at 70-30 split when random state was varied but alpha was constant.

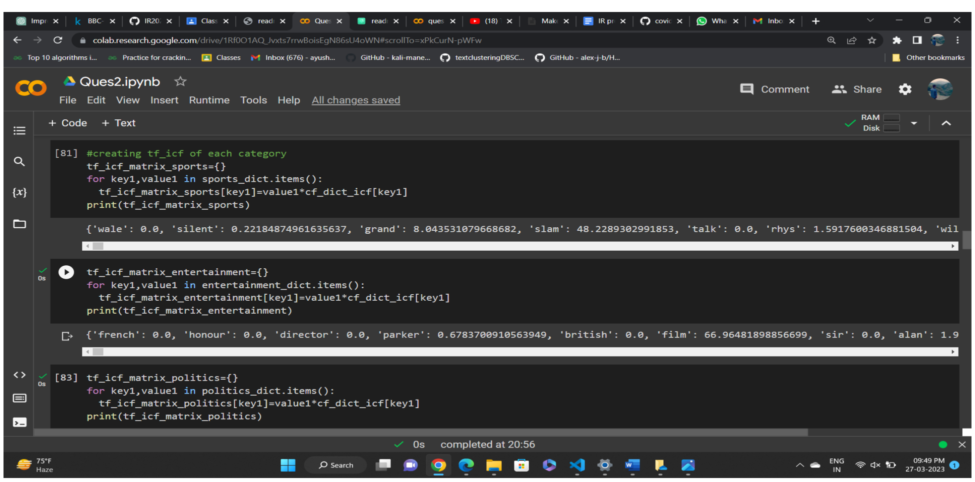
2.Also when varying alpha accuracy came out be close to the first one.

3.When the splits were 50-50,80-20 and 60-40 the accuracy were reduced in each case but precision,F1-score and recall were almost same in all cases.

4. After Applying various variations in the Naive Bayes Classifier we found that splitting in 70-30 dataset and applying random state and alpha in parameter tuning we get the maximum accuracy but the precision,recall,F-1 score is almost same in all cases.

`





Ques2.ipynb

```
new_df_2dmatrix_new
```

	0	1	2	3	4	5	6	7	8	9	...	1684	1685	1686	1687	1688
0	37.744380	1.397940	0.000000	0.000000	0.000000	0.193820	0.000000	37.744380	0.000000	5.591760	...	0.0	0.0	0.0	0.0	0.0
1	0.000000	0.000000	2.713480	3.327731	0.000000	0.000000	2.713480	0.000000	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0
2	0.000000	0.000000	2.096910	19.575823	0.795880	0.000000	0.000000	0.000000	0.484550	0.000000	...	0.0	0.0	0.0	0.0	0.0
3	7.560860	0.795880	40.702205	0.000000	4.880672	0.000000	1.397940	10.346440	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0
4	7.688670	1.453650	2.096910	4.193820	2.795880	0.000000	7.688670	0.000000	0.000000	2.096910	...	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1485	0.000000	5.571160	0.000000	0.000000	0.000000	2.795880	1.397940	0.000000	19.769643	0.000000	...	0.0	0.0	0.0	0.0	0.0
1486	6.877311	0.000000	0.000000	0.221849	52.528081	0.000000	6.877311	0.775280	3.494850	1.331092	...	0.0	0.0	0.0	0.0	0.0
1487	0.000000	14.730322	0.000000	7.764706	0.000000	0.000000	0.000000	7.764706	1.774790	0.000000	...	0.0	0.0	0.0	0.0	0.0
1488	11.435382	23.876401	0.000000	0.397940	14.827232	11.435382	0.096910	23.876401	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0
1489	3.494850	11.540260	0.000000	0.698970	0.000000	0.000000	0.000000	0.665546	0.000000	1.397940	...	0.0	0.0	0.0	0.0	0.0

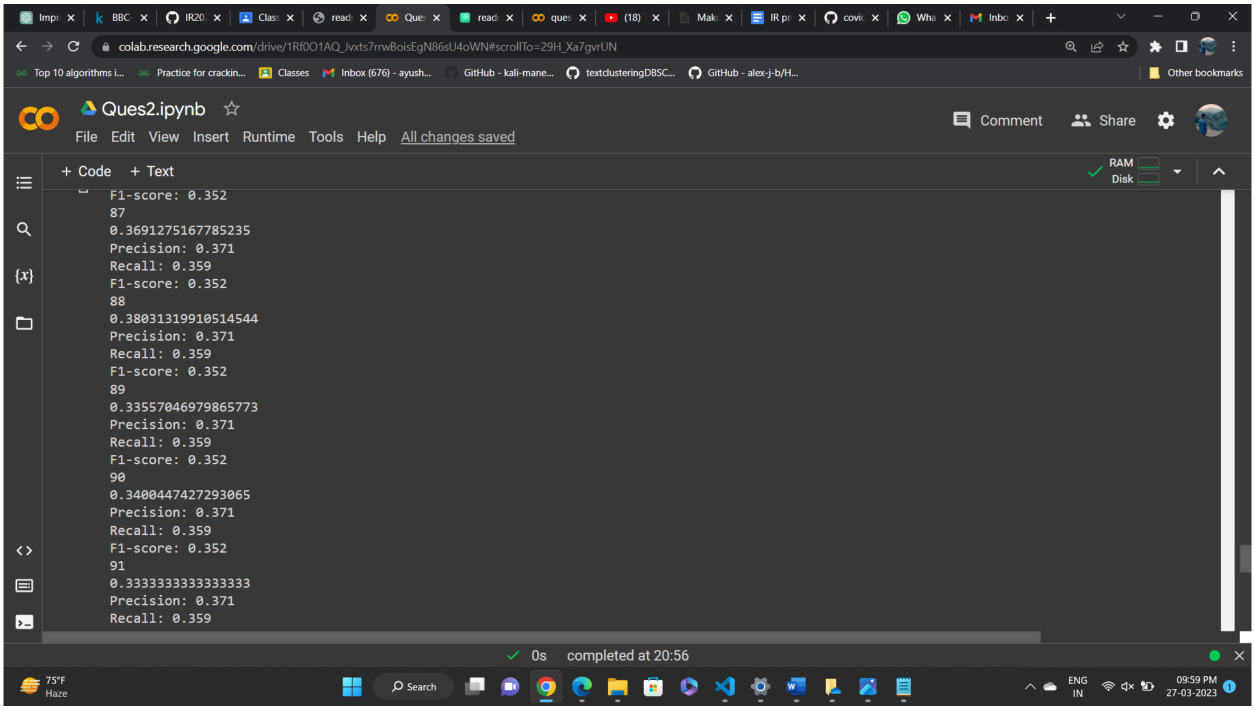
0s completed at 20:56

Ques2.ipynb

```
for i in range(0,100):
  X_train_loop, X_test_loop, y_train_loop, y_test_loop = train_test_split(X_1, Y_1, train_size = 0.7, random_state = i)
  nb = MultinomialNB()
  nb.fit(X_train_loop, y_train_loop)
  y_pred_class_loop = nb.predict(X_test_loop)
  # predict probabilities
  y_pred_proba_loop= nb.predict_proba(X_test_loop)
  accuracy_1=metrics.accuracy_score(y_test_loop, y_pred_class_loop)
  print(accuracy_1)
  print('Precision: %.3f' % precision_score(y_test_new, y_pred_class,average="macro"))
  print('Recall: %.3f' % recall_score(y_test_new, y_pred_class,average="macro"))
  print('F1-score: %.3f' % f1_score(y_test_new, y_pred_class,average="macro"))
  print(i)
```

```
Recall: 0.359
F1-score: 0.352
82
0.33557046979865773
Precision: 0.371
Recall: 0.359
F1-score: 0.352
83
0.2953020134228188
Precision: 0.371
```

0s completed at 20:56



Q3.

1. *We first import necessary libraries like pandas, collections, numpy, math, and seaborn.*
2. *Then we read a dataset and store it in a pandas dataframe.*
3. *We select only the rows where the value in the second column is 'qid:4'.*
4. *The data frame is sorted by relevance score and a new csv file is created with the sorted data.*
5. *The relevance score column is renamed as 'Relevance score'.*
6. *The unique relevance scores are printed.*
7. *Define a function to calculate the Ideal Discounted Cumulative Gain (IDCG) and then print the IDCG for 50 and all documents.*
8. *The number of files that can be made is also calculated.*
9. *Another function is defined to calculate the Discounted Cumulative Gain (DCG) for a given number of documents.*
10. *The DCG is then calculated for 50 and all documents.*
11. *The Normalized Discounted Cumulative Gain (NDCG) is calculated for 50 and all documents and printed.*
12. *We extract the TF-IDF values from column 76 of the dataframe and store them in a list, which is then converted to float and added to the dataframe.*
13. *The data frame is sorted by the TF-IDF value in descending order.*

14. Lastly we plot the precision vs recall graph using matplotlib.

Importing all the packages

Imporing all Libraries

```
[ ] import pandas as pd
    from collections import Counter
    import numpy as np
    import math
    import seaborn as sns
    import pandas as pd
```

Reading the dataset

```
[ ] #Reading the dataset

    df = pd.read_csv('IR-assignment-2-data (2).txt',sep = " " , header = None)
```

```
[ ] #Printing the df
    df
```

	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
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
239088	0	qid:29989	1:2	2:0	3:1	4:1	5:2	6:1	7:0	8:0.50000	...	128:9754	129:29	130:2889	131:63571	132:1	133:1	134:0	135:0	136:0	NaN
239089	0	qid:29989	1:2	2:0	3:1	4:0	5:2	6:1	7:0	8:0.50000	...	128:84	129:1	130:9450	131:19599	132:4	133:4	134:0	135:0	136:0	NaN
239090	1	qid:29989	1:2	2:0	3:2	4:2	5:2	6:1	7:0	8:1	...	128:1	129:0	130:144	131:6701	132:5	133:2	134:0	135:0	136:0	NaN

Taking rows with qid:4

Taking only rows with qid:4

```
[ ] # selecting only qid:4
    df = (df.loc[df[1] == 'qid:4'])
    df
```

	0	1	2	3	4	5	6	7	8	9	...	128	129	130	131	132	133	134	135	136	137
0	0	qid: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	qid: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	qid: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	qid: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	qid: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	qid: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	qid: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	qid: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	qid: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	qid: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

Sorting the df by it's relevence score and creating another file

```
[ ] #Sorting the df by it's relevence score and creating another file

    df_change=df.copy()
    df_change=df_change.sort_values(by=0,ascending=False)
    df_change.to_csv('max_dcg.csv')
```

Sorting all the df by the relevance score and creating another file

Sorting the df by it's relevence score and creating another file

```
[ ] #Sorting the df by it's relevence score and creating another file

    df_change=df.copy()
    df_change=df_change.sort_values(by=0,ascending=False)
    df_change.to_csv('max_dcg.csv')
```

Renaming the column 0 by Relevence score

```
[ ] #Renaming the 0 column by Relevence score
    df_change=df_change.rename({0:'Relevence score'},axis='columns')
    df_change
```

	Relevence score	1	2	3	4	5	6	7	8	9	...	128	129	130	131	132	133	134	135	136	137
7	3	qid:4	1:3	2:0	3:2	4:1	5:3	6:1	7:0	8:0.666667	...	127:32	128:349	129:8	130:123	131:281	132:22	133:6	134:0	135:0	136:0
76	2	qid:4	1:2	2:0	3:1	4:0	5:2	6:0.666667	7:0	8:0.333333	...	127:19	128:0	129:0	130:2417	131:721	132:14	133:113	134:0	135:13	136:47.9
40	2	qid:4	1:3	2:2	3:2	4:0	5:3	6:1	7:0.666667	8:0.666667	...	127:33	128:8	129:3	130:1888	131:9338	132:3	133:11	134:0	135:0	136:0
36	2	qid:4	1:3	2:0	3:2	4:0	5:3	6:1	7:0	8:0.666667	...	127:17	128:0	129:2	130:12028	131:11379	132:26	133:24	134:0	135:77	136:23.9595223404047
90	2	qid:4	1:3	2:0	3:3	4:3	5:3	6:1	7:0	8:1	...	127:67	128:27	129:0	130:814	131:13555	132:108	133:113	134:0	135:0	136:0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
44	0	qid:4	1:2	2:0	3:0	4:0	5:2	6:0.666667	7:0	8:0	...	127:41	128:8	129:0	130:868	131:9260	132:246	133:88	134:0	135:0	136:0
43	0	qid:4	1:2	2:0	3:0	4:0	5:2	6:0.666667	7:0	8:0	...	127:38	128:4	129:0	130:797	131:9260	132:237	133:80	134:0	135:0	136:0
42	0	qid:4	1:3	2:0	3:3	4:1	5:3	6:1	7:0	8:1	...	127:65	128:83	129:5	130:144	131:262	132:157	133:179	134:0	135:0	136:0
41	0	qid:4	1:3	2:1	3:3	4:2	5:3	6:1	7:0.333333	8:1	...	127:65	128:195	129:8	130:124	131:206	132:103	133:121	134:0	135:0	136:0
102	0	qid: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

Printing the unique Relevence score

Printing the unique Relevence score

```
[ ] #Printing the Score

    score=df_change['Relevence score'].value_counts()
    print('The Unique Relevent Score is Given below')
    score
```

```
The Unique Relevent Score is Given below
0    59
1    26
2    17
3     1
Name: Relevence score, dtype: int64
```

Calculating the maximum DCG

Calculating the Maximum DCG

```
[ ] #To calculate the Maximum DCG (IDCG)

    def calculating_ideal_dcg(d):
        documents=list(df[0])[:d]
        documents.sort()
        documents.reverse()

        idcg=0
        for i in range(d):
            idcg+=documents[i]/math.log2(i+2)
        return idcg
```

```
[ ] #Printing the Maximum DCG for 50 documents

    idcg_50=calculating_ideal_dcg(50)
    print(idcg_50)
```

```
12.58382772001186
```

```
[ ] #Printing the Maximum DCG for All documents

    idcg_all=calculating_ideal_dcg(103)
    print(idcg_all)
```

```
19.407247618668023
```

```
[ ] #Number of files that can be made

    score1=list(score)
    count=1
    for i in range(len(score1)):
```

Calculating NDCG at position 50 and for entire dataset

Printing the NDGC for 50 documents

```
[ ] #Calculating and Printing the NDCG for 50 documents

    ndcg_50=dcg_50/idcg_50
    print("Printing NDCG for 50 Documents")
    print(ndcg_50)
```

```
Printing NDCG for 50 Documents
0.5717260627203818
```

Printing the NDGC for All documents

```
[ ] #Calculating and Printing the NDCG for All documents
    ndcg_all=dcg_all/idcg_all
    print("Printing NDCG for All Documents")
    print(ndcg_all)
```

```
Printing NDCG for All Documents
0.6357153091990775
```

Extracting the TF_idf value from 76 column and storing into the list

Extracting the TF_idf value from 76 column and storing into the list

```
[ ] #Extracting the TF_idf value from 76 column and storing into the list
    tf=[]
    for i in df_3.iloc[:,76]:
        tf.append(str(i).split(':')[1])
```

```
#converting into float and storing the tf_idf value along with the relevence score into doc_3

tf_idf=[float(i) for i in tf]
df_3.iloc[:,76]=tf_idf
```

```
C:\Users\Devanshu\AppData\Local\Temp\ipykernel_13648\3296264913.py:4: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attempt to set the values inplace instead of always setting a new array. To retain the old behavi
  df_3.iloc[:,76]=tf_idf
```

```
[ ] #soring the df_3 by it's TF_IDF value in descending order.

    df_4=df_3.loc[:,[0,76]].sort_values(by=76,ascending=False)
```

```
[ ] #Printing the df_4
    df_4
```

	0	76
8	0	972.826451
67	0	612.893205
56	0	571.500533
1	0	538.388954
101	1	528.520116
...	...	...
94	0	15.773388
16	0	14.972391
86	0	14.972391

Precision vs Recall graph