Q-1).

- First of all, we have imported some required libraries in order to do the pre-processing of data. Also, we have mounted the dataset files in drive.
- We have calculated the |A or B| and |A and B| as a part of pre-processing and also performed tokenizer for generate the tokens and converted it into the lower case and remove the stop words.
- Then we have implemented the function for Jaccard coefficient as you can see on the below snippet and see the value of 'department agriculture'.
- Also, we have reported Top 5 relevant document based on the value we got for the Jaccard Coefficient.

```
[ ]  # Calculation of Jaccard coefficient

     def jaccardCoefficient(query):
       tokens = preprocessing(query)
       print(tokens)
       jc = {}
       for i in textdf.index:
         intlen = intersection(tokens,textdf.loc[i,'Text'])
         unilen = union(tokens,textdf.loc[i,'Text'])
         jc["d"+ str(i)] = intlen/unilen

       jc = pd.DataFrame.from_dict(jc,orient = 'index')
       jc.rename(columns={0:'Jaccard Coefficient'},inplace=True)
       jc.sort_values('Jaccard Coefficient',inplace=True,ascending=False)
       return jc

     print(jaccardCoefficient("department agriculture").head())

     ['department', 'agriculture']
          Jaccard Coefficient
     d3              0.007782
     d456            0.002564
     d1              0.000000
     d754            0.000000
     d760            0.000000
```

- So here we have computed the term frequency involves the binary as weighting scheme of the word in each document and see the same in the below snippet.

```
binarytf.head()
```

id	rinaldo	laws	leaving	washington	area	early	may	thought	appropriate	share	...	tspecimen	tego	tsucceed	apolloway	unetix	calistogas	commencing	mo
d1	1	1	1	1	1	1	1	1	1	1	...	0	0	0	0	0	0	0	
d2	0	0	0	0	0	0	1	0	0	0	...	0	0	0	0	0	0	0	
d3	0	0	0	0	1	0	0	0	0	0	...	0	0	0	0	0	0	0	
d4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
d5	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	

5 rows × 70548 columns

- So here we have computed the term frequency involves the raw count as weighting scheme of the word in each document and see the same in the below snippet.

```
rowcounttf.head()
```

id	rinaldo	laws	leaving	washington	area	early	may	thought	appropriate	share	...	tspecimen	tego	tsucceed	apolloway	unetix	calistogas	commencing	mo
d1	1	2	2	1	1	1	3	1	1	1	...	0	0	0	0	0	0	0	
d2	0	0	0	0	0	0	2	0	0	0	...	0	0	0	0	0	0	0	
d3	0	0	0	0	1	0	0	0	0	0	...	0	0	0	0	0	0	0	
d4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
d5	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	

5 rows × 70548 columns

- So here we have computed the term frequency involves the term frequency as weighting scheme of the word in each document and see the same in the below snippet.

```
termfrequency.head()
```

id	rinaldo	laws	leaving	washington	area	early	may	thought	appropriate	share	...	tspecimen	tego	tsucceed	apolloway	unetix	calist
d1	0.25	0.012658	0.010417	0.00641	0.002558	0.002865	0.001715	0.001183	0.010101	0.007812	...	0.0	0.0	0.0	0.0	0.0	
d2	0.00	0.000000	0.000000	0.00000	0.000000	0.000000	0.001144	0.000000	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0	
d3	0.00	0.000000	0.000000	0.00000	0.002558	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0	
d4	0.00	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0	
d5	0.00	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.0	0.0	0.0	0.0	0.0	

5 rows × 70548 columns

- So here we have computed the term frequency involves the Log normalization as weighting scheme of the word in each document and see the same in the below snippet.

```
lognormalization.head()
```

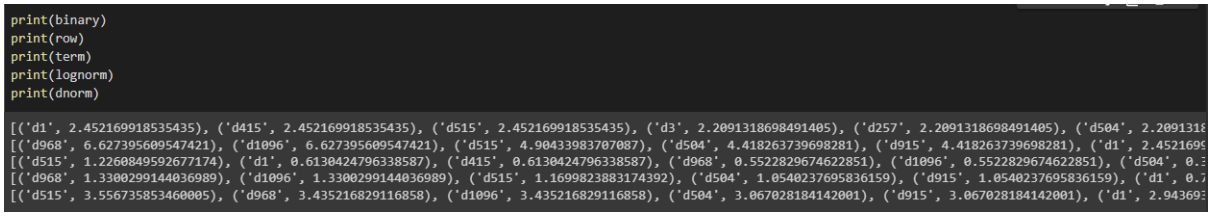
id	rinaldo	laws	leaving	washington	area	early	may	thought	appropriate	share	...	tspecimen	tego	tsucceed	apolloway	unetix	calistogas
d1	0.30103	0.477121	0.477121	0.30103	0.30103	0.30103	0.602060	0.30103	0.30103	0.30103	...	0.0	0.0	0.0	0.0	0.0	0.0
d2	0.00000	0.000000	0.000000	0.00000	0.00000	0.00000	0.477121	0.00000	0.00000	0.00000	...	0.0	0.0	0.0	0.0	0.0	0.0
d3	0.00000	0.000000	0.000000	0.00000	0.30103	0.00000	0.000000	0.00000	0.00000	0.00000	...	0.0	0.0	0.0	0.0	0.0	0.0
d4	0.00000	0.000000	0.000000	0.00000	0.00000	0.00000	0.000000	0.00000	0.00000	0.00000	...	0.0	0.0	0.0	0.0	0.0	0.0
d5	0.00000	0.000000	0.000000	0.00000	0.00000	0.00000	0.000000	0.00000	0.00000	0.00000	...	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 70548 columns

- Now we are using the function to get the TF-IDF score for the query and also used the all 5 weighting scheme techniques and get the result in sorted order.

```
[ ] from pandas.core.internals.array_manager import T
    def tfidfScore(query):
      binary_res = {}
      rowcount_res = {}
      termfreq_res = {}
      lognorm_res = {}
      dnorm_res = {}

      columns = list(binary_tfidf.columns)
      tokens = preprocessing(query)

      for index in binary_tfidf.index:
        sumb,sumr,sumt,suml,sumd = 0,0,0,0,0
        for term in tokens:
          if term in columns:
            sumb += binary_tfidf.loc[index,term]
            sumr += rowcount_tfidf.loc[index,term]
            sumt += termfrequency_tfidf.loc[index,term]
            suml += lognormalization_tfidf.loc[index,term]
            sumd += dnormalization_tfidf.loc[index,term]

        binary_res[index] = sumb
        rowcount_res[index] = sumr
        termfreq_res[index] = sumt
        lognorm_res[index] = suml
        dnorm_res[index] = sumd

      binary_res = sorted(binary_res.items(), key=lambda x: x[1], reverse=True)
      rowcount_res = sorted(rowcount_res.items(), key=lambda x: x[1], reverse=True)
      termfreq_res = sorted(termfreq_res.items(), key=lambda x: x[1], reverse=True)
      lognorm_res = sorted(lognorm_res.items(), key=lambda x: x[1], reverse=True)
      dnorm_res = sorted(dnorm_res.items(), key=lambda x: x[1], reverse=True)


      return binary_res,rowcount_res,termfreq_res,lognorm_res,dnorm_res
```

- So finally, we have reported the TF-IDF score and results for each weighting scheme separately and you can see the score in the below snippet.



Pros and cons of each scoring scheme mentioned here:

- **Jaccard Coefficient**

  i) Pros:
  Here we got the better result where the repetition and duplication of the words not matter.

  ii) Cons:
  Jaccard Coefficient does not consider any rare term in the collection because term frequency not considered so far.

- **TF-IDF Matrix**

  i) Pros:
  TF-IDF is much easy to calculate the similarity between two different documents and also rare terms are much informative than the frequent terms.

  ii) Cons:
  TF-IDF Matrix doesn't capture the position in next semantic because it's based on BoW model (Bag of Word).

Q-2).

DCG (Discounted Cumulative Gain):

- The main aim of DCG is to measure ranking quality and the effectiveness of search algorithms in Information Retrieval.
- It also measures the gain of a document according to its position in the list of results, and it accumulated from the top of the result list to the bottom, with the gain of each result discounted at a lower rank.

- So here we have considered only the queries with qid:4 and performed the further processing steps.

```
#We are consider the queries having only qid=4
df2 = df1[df1[1]=="qid:4"]

df2.head()
```

- In order to print the max pairs DCG query-url we are going to do sorting in descending order.

```
#In order to print the max DCG pairs of query-url we are doing sorting in descending order.
df_sort=df2.sort_values(by=0,ascending=False)
df_sort=df_sort.reset_index(drop=True)
df_sort.to_csv('SortedDCG')
```

- Here we have printed the files after rearranging the query-url pairs in order of max DCG.

```
print("The files after rearranging the query url pairs in order of max DCG..",res)
```

```
The files after rearranging the query url pairs in order of max DCG.. 39575123784646683675471384675010627874852892709737449466978225873433070499508284345269778
```

- Following snippet shows the nDCG score at 50 and for the whole dataset respectively.

```
out1 = calculateDCG(df2,50)/calculateDCG(df_sort,50)
print("At 50 :",out1)
```

```
At 50 : 0.35612494416255847
```

```
out2 = calculateDCG(df2,df2_len)/calculateDCG(df_sort,df2_len)
print("For the Whole dataset :",out2)
```

```
For the Whole dataset : 0.5784691984582591
```

- Here we have simply ranked url on the basis of the value of feature 75 and got the different value of precision and recall. Then we have stored it in a list.

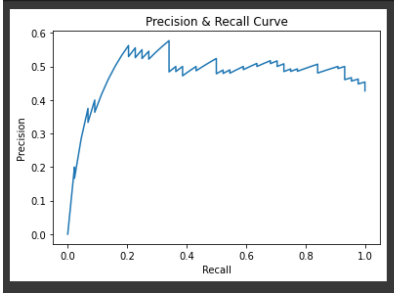
```
for con in fcontent:
  j+=1
  if con[con.find("qid")+4:con.find("1:")-1]=="4":
    f1=con.find("75:")
    f2=con.find("76:")
    values.append(int(con[0]))
    score.append(float(con[f1+3:f2-1]))
    if float(con[f1+3:f2-1])==0.0:
          print(con)
```

Assumption:

- Assume that non zero relevance judgement value to be relevant.
- Plot of the graph goes exponentially straight b/w 0 and 0.2 recall values. Precision remains b/w 0.4 and 0.57.

- Finally, we have plotted the precision-recall curve for the query which has id = qid:4. The same you can see on the below snippet.

```
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision & Recall Curve")
plt.plot(reclist,preclist)
plt.show()
```



Q-3).

- We have performed some pre-processing technique here in order to do further computation. One can easily shows the same in the snippet attached below.
- Here we have performed tokenizer technique and removed stop words from the strings.

```
def preprocessing(s):
  re.sub(r'''(?i)\b((?:https?://|www\d{0,3}[.]|[a-z0-9.\-]+[.][a-z]{2,4}/)(?:[^\s()<>]+|\(([^\s()<>]+|(\([^\s()<>]+\)))*\))+(?:\(([^\s()<>]+|(\([^\s()<>]+\)))*
  tokenizer = nltk.RegexpTokenizer(r"\w+")   # Tokenizer
  s = tokenizer.tokenize(s)
  s = [w.lower() for w in s]
  sw = set(stopwords.words('english'))   #remove stop words from strings
  res = [w for w in s if (not w in sw and len(w)>2 and bool(re.search(r'\d', w))==False)]
  res = [w.replace("_","") for w in res]
  return list(np.unique(res))
```

- Here we have split the data and selecting documents randomly. Also, we have split the train and test dataset into 80:20 criteria.

```
def splitData(frac):
  data = pickle.load(open("/content/drive/MyDrive/IR/dataA2Q3",'rb'))
  length = math.floor(len(data)*frac)
  data = shuffle(data).reset_index(drop=True)
  traindata = data.loc[:length]
  testdata = data.loc[length:]
  return traindata, testdata

traindata,testdata = splitData(0.8)
```

- Then we have implemented the TF-ICF scoring technique for efficient feature selection. The function for the same we can see in the below snippet.

```
def trainTfIcf(i,c,frac):
  tf_icf = []
  icf = []
  termfreq = []
  #traindata,testdata = splitData(frac)
  for w in ut[i-1]:
    tf = 0
    cl = 0
    for l in traindata[traindata['Class']== c].Tokens:
      tf += l.count(w)

    termfreq.append(tf)
    for l in ut:
      if w in l:
        cl += 1
    inversfreq = np.log10(cl/tf)
    icf.append(inversfreq)
    tf_icf.append(tf*inversfreq)
  return tf_icf,termfreq,icf
```

- So now we have tokens with the score of TF-ICF value which can be shown here.

	Token	TF_ICF
0	island	0.79588
1	ahead	0.79588
2	alabama	0.79588
3	temporary	0.79588
4	kids	0.79588

Analysis:

- So here Naïve bayes with TF-ICF has given better performance than Naïve bayes technique.
- Also, Naïve bayes with TF-ICF has slightly better accuracy than the simple naïve bayes algorithm.

Learnings: -

- At the very first, we learned a lot of things from this assignment, like pre-processing the dataset and performing the various operation on it.
- We also have learned how to compute Jaccard Coefficient, TF-IDF matrix, and DCG type operation. It seems to be very hard to calculate, but we take this as an opportunity, and finally, we can make it successful.
- Overall, it has improved my knowledge and understanding of complex problems.