IR-Assignment2

Hritika Shah(MT22101)

Gnanesh Patel(MT22030)

Nikita Kapoor(2020531)

Q1)

❖ Tf-idf matrix:

- At first we stored the contents of all the documents in the corpus, tokenized it and found all the unique words present in the corpus which were nearly 9k.
- Through unique words we found the document frequency of the terms and calculated the idf values for each term.

```
import math
import os
from collections import Counter
import pandas as pd

path = "D://ir//assignment2//CSE508_Winter2023_Dataset"
dir_list = os.listdir(path)

# Create a corpus containing the contents of all 1400 documents
corpus = []
for file in dir_list:
    x = 'D://ir//assignment2//CSE508_Winter2023_Dataset//'+file
    with open(x, 'r') as file:
        corpus.append(file.read())


# Split all the words to create a tokenized corpus
tokenized_corpus = [x.split() for x in corpus]

# Find the set containing all the unique words from tokenized corpus
unique_words = set(word for doc in tokenized_corpus for word in doc)

# Calculate documnet frequency of all the unique terms
doc_freq = {word: sum(1 for doc in tokenized_corpus if word in doc)
            for word in unique_words}

# Calculating idf values of all the unique words
idf = {word: math.log10(
    len(corpus) / (1 + doc_freq[word])) for word in unique_words}
```

➔ Binary scheme:

- To find the tf values of binary scheme, 0 if word is not present and 1 if present.
- To calculated the tfidf matrix multiplied the tf values with the idf values for each particular term.
- Finally we converted the output list to dataframe to form the perfect output matrix with rows containing each unique word and columns containing the 1400 documents.

```
# ---------------------------------------------------------BINARY---------------------------------------------------#
def binary():
    words=[]
    for x in unique_words:
        tfidf = []
        for y1 in tokenized_corpus:
            if x not in y1:
                # append 0 if word is not present
                tfidf.append(0)
            else:
                # append 1 * idf[word] to get its idf value
                tfidf.append(idf[x])
        words.append(x)
        output.append(tfidf)

    df = pd.DataFrame(output)
    df.index=words
    df.columns = range(1,len(df.columns)+1)
    return df
```

➔ Raw count:

- For raw count, to find the tf values we appended 0 if word is not present and count of the word in each particular document.
- To calculated the tfidf matrix multiplied the tf values with the idf values for each particular term.

```
# ---------------------------------------------------------RAW COUNT---------------------------------------------------#
def rawcount():
    words=[]
    for x in unique_words:
        tfidf = []
        for y1 in tokenized_corpus:
            if x not in y1:
                # append 0 if word is not present
                tfidf.append(0)
            else:
                # append word count in particular document * idf[word]
                tfidf.append(y1.count(x)*idf[x])
        words.append(x)
        output.append(tfidf)

    df = pd.DataFrame(output)
    df.index=words
    df.columns = range(1,len(df.columns)+1)
    return df
```

➔ Term Frequency

- To find the tf values using term frequency, append 0 if word is not present and append count of the particular term/count of all other terms for a particular document but here if one only word is present in the document than it would result into error so even including the frequency of that terms also in that case.
- To calculated the tfidf matrix multiplied the tf values with the idf values for each particular term.

```
# -----------------------------------------------------------TERM FREQUENCY-----------------------------------------------------#
def term_freq():
    words=[]
    for x in unique_words:
        tfidf = []
        for y1 in tokenized_corpus:
            l = Counter(y1)
            temp=0
            for j in l:
                temp = temp + l[j]
            if x not in y1:
                # append 0 if word is not present
                tfidf.append(0)
            else:
                # append word count in particular document/total number of terms * idf[word]
                tfidf.append((y1.count(x)/temp)*idf[x])
        words.append(x)
        output.append(tfidf)

    df = pd.DataFrame(output)
    df.index=words
    df.columns = range(1,len(df.columns)+1)
    return df
```

➔ Log Normalization:

- To calculate the tf value for log normalization, append 0 if word is not present, and append (log of word count in particular document + 1) if word is present.
- To calculated the tfidf matrix multiplied the tf values with the idf values for each particular term.

```
# -------------------------------------------------------LOG NORMALIZATION----------------------------------------------------#
def log_normalize():
    words=[]
    for x in unique_words:
        tfidf = []
        count1 = 0
        for y1 in tokenized_corpus:
            if x not in y1:
                # append 0 if word is not present
                tfidf.append(0)
            else:
                # append (log of word count in particular document + 1) * idf[word]
                tfidf.append((math.log10(y1.count(x)+1))*idf[x])
        words.append(x)
        output.append(tfidf)

    df = pd.DataFrame(output)
    df.index=words
    df.columns = range(1,len(df.columns)+1)
    return df
```

➔ Double normalization:

- To calculate the tf value for log normalization, append 0 if word is not present, and append (0.5 + 0.5 * (word count in particular document/ ,ax count among all the terms) if word is present.
- To calculated the tfidf matrix multiplied the tf values with the idf values for each particular term.

```python
# ---------------------------------------------------Double normalization ---------------------------------------------------#
def double_normalize():
    words=[]
    for x in unique_words:
        tfidf = []
        for y1 in tokenized_corpus:
            l = Counter(y1)
            max=0
            for j in l:
                if(max<l[j]):
                    max=l[j]
            if x not in y1:
                # append 0 if word is not present
                tfidf.append(0)
            else:
                # append 0.5 + (0.5 * word count in particular document/max count among all the terms)* idf[word]
                tfidf.append((0.5 + (0.5 * (y1.count(x)/max)))*idf[x])
        words.append(x)
        output.append(tfidf)

    df = pd.DataFrame(output)
    df.index=words
    df.columns = range(1,len(df.columns)+1)
    return df
```

➔ Final Output:

```
D:\ir\assignment2\Q1>python Q1_tfidf.py
1 for binary
2 for raw count
3 for term frequancy
4 for log normalization
5 for double normalization
Enter number here:3
                   1    2    3    4    5    6    7    8    9   10   11   12   13   14   15  ...  1386  1387  1388  1389  1390  1391      1392  1393  1394  1395  1396  1397  1398  1399  1400
panelsupport     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0  0.000000   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
service          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0  0.000000   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
determinantal    0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0  0.034486   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
unyawed          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0  0.000000   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
rameter          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0  0.000000   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
...              ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...   ...   ...   ...   ...   ...   ...       ...   ...   ...   ...   ...   ...   ...   ...   ...
twodimensionality 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0  0.000000   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
investigators    0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0  0.000000   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
wavelength00005  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0  0.000000   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
192              0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0  0.000000   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
lengths          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...   0.0   0.0   0.0   0.0   0.0   0.0  0.000000   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

[9101 rows x 1400 columns]
```

➔ When query of length of size of vocab is given as an input:

- Output using binary scheme:

```
244      356.111287
1313     324.684752
344      320.320448
792      317.786113
798      315.295384
dtype: float64
```

- Output using raw count:

```
1313     525.796354
244      486.885918
329      474.690180
798      453.653180
721      429.503045
dtype: float64
```

- Output using term frequency:

```
995      2.669007
471      2.669007
718      2.153406
1168     2.038724
83       1.997140
dtype: float64
```

- Output using log normalization:

```
244      123.066579
1313     122.868240
798      113.878077
792      107.391047
344      106.672923
dtype: float64
```

- Output using double normalization:

```
244     193.270829
344     192.767115
792     191.575616
798     180.330351
1313    175.487285
dtype: float64
```

➔ Pros and Cons using each scheme:

1) Binary:
   Pros: - It simplifies the computation and hence computes fast, it reduces the impact of the document length and it reduces the impact of the words occurring very frequently.
   Cons: - It may lose some information, it doesn't considers the term frequency of each particular term, and it may reduce the accuracy of the similarity measures.

2) Raw Count:
   Pros: - Preserves the information of each word frequency, Works best for the shorter documents, better than the binary scheme.
   Cons: - Biased towards longer documents, affected by stop words.

3) Term Frequency:
   Pros: - Captures important words, identifies the relevant terms, normalizes the occurrence of words with the total words in particular document.
   Cons: - Sensitive to document length, prone to noise, if a word occurs very frequently it gets affected.

4) Log Normalization:
   Pros: - Reduces the impact of the outlier terms, makes scores more interpretable, solves the problem of term frequency.
   Cons: - May lead to loss of information, may be bias, may not be appropriate for all datasets.

5) Double Normalization:
   Pros: - Smoothing factor 0.5 is added, reduces the impact of the document length.
   Cons: - May not be appropriate for all datasets, may be bias and lead to loss of information.

❖ Jaccard Coefficient:

- We have document and query given and we need to find top 10 documents according to jaccard score of document for the given query.
- First we need to preprocess the data od document and query. Perform various steps like remove stop words, stemming, remove punctuation etc.
- After preprocessing we will be getting number of unique words for document as well as for query.
- Count number of unique words in doc and query by using length function of python.
- Perform intersection and union operation on number of unique words of document and query.
- Count the length of the result.
- Then divide number of intersect word with number of union word that will give us jaccard coefficient.
- Sort the documents on reverse order based on that jaccard value and return top 10 document from that.

```python
#store all the document with thier jaccard values in ans
ans=[]
j=0
for i in tokenized_corpus:
    set1=set(data)# convert query in to set
    set2=set(i)# convert document words in to set
    intersection_count= len(set1.intersection(set2)) #perform intersection on document and query
    union_count= len(set1.union(set2)) #perform union on document and query
    jaccard_efficient= intersection_count/union_count #calculate jaccrard coefficent
    j+=1
    ans.append({"name":j,"value":jaccard_efficient}) #append doc with value to the answer list

ans
final_ans=sorted(ans,key=lambda s: s['value'],reverse=True) #reverse sort the ans in order to get top 10 docs
```

```python
final_ans
```

```
Output exceeds the size limit. Open the full output data in a text editor
[{'name': 1045, 'value': 0.07142857142857142},
 {'name': 31, 'value': 0.05555555555555555},
 {'name': 632, 'value': 0.05405405405405406},
 {'name': 137, 'value': 0.045454545454545456},
 {'name': 271, 'value': 0.045454545454545456},
 {'name': 286, 'value': 0.045454545454545456},
 {'name': 1062, 'value': 0.045454545454545456},
 {'name': 250, 'value': 0.043478260869565216},
 {'name': 256, 'value': 0.043478260869565216},
 {'name': 920, 'value': 0.043478260869565216},
 {'name': 1146, 'value': 0.043478260869565216},
 {'name': 670, 'value': 0.041666666666666664},
```

```
#printing top 10 docs for the given query based on jaccard coefficent
for i in range(10):
    print(final_ans[i])
```

```
{'name': 1045, 'value': 0.07142857142857142}
{'name': 31, 'value': 0.05555555555555555}
{'name': 632, 'value': 0.05405405405405406}
{'name': 137, 'value': 0.045454545454545456}
{'name': 271, 'value': 0.045454545454545456}
{'name': 286, 'value': 0.045454545454545456}
{'name': 1062, 'value': 0.045454545454545456}
{'name': 250, 'value': 0.043478260869565216}
{'name': 256, 'value': 0.043478260869565216}
{'name': 920, 'value': 0.043478260869565216}
```

Q2)

- **Tf-Icf:**

- At first, we read the csv file using pandas dataframe.
- Did preprocessing steps: - converting it to lower case, removing punctuations, removing stop words, removing white spaces and performing lemmatization of the text and writing the changes back into the dataframe.
- Dropped the unwanted column i.e. 'Articleid' from the dataframe.
- Created a set of all unique words present in the Text column which were nearly 25k.

```
words12=[]
for i in range(len(df)):
    words12.append(df['Text'][i])
```

```
tokenized_corpus = [x.split() for x in words12]
unique_words = set(word for doc in tokenized_corpus for word in doc)
```

```
print(len(unique_words))
```

```
24834
```

- Than, we found out the unique categories present in the data and accordingly creates 5 corpus and counted the occurrence of each word present in the particular corpus using Counter.

```
c1=[]
c2=[]
c3=[]
c4=[]
c5=[]
for i in range(len(df)):
    if(df['Category'][i]=='business'):
        c1.append(df['Text'][i])
    elif(df['Category'][i]=='tech'):
        c2.append(df['Text'][i])
    elif(df['Category'][i]=='politics'):
        c3.append(df['Text'][i])
    elif(df['Category'][i]=='sport'):
        c4.append(df['Text'][i])
    elif(df['Category'][i]=='entertainment'):
        c5.append(df['Text'][i])
```

```
tokenized_corpus = [x.split() for x in c1]
words1 = (word for doc in tokenized_corpus for word in doc)

tokenized_corpus2 = [x.split() for x in c2]
words2 = (word for doc in tokenized_corpus2 for word in doc)

tokenized_corpus3 = [x.split() for x in c3]
words3 = (word for doc in tokenized_corpus3 for word in doc)

tokenized_corpus4 = [x.split() for x in c4]
words4 = (word for doc in tokenized_corpus4 for word in doc)

tokenized_corpus5 = [x.split() for x in c5]
words5 = (word for doc in tokenized_corpus5 for word in doc)
```

```
from collections import Counter
w1=Counter(words1)
w2=Counter(words2)
w3=Counter(words3)
w4=Counter(words4)
w5=Counter(words5)
```

- Using that, calculated the tf-icf values for each category individually and stored it in a dictionary.

```python
import math
tficf={}
for i in w1:
    temp=[]
    # print(w1[i])
    tf1 = w1[i]
    cf = 1
    if i in w2:
        cf=cf+1
    if i in w3:
        cf=cf+1
    if i in w4:
        cf=cf+1
    if i in w5:
        cf=cf+1
    icf = math.log10(5/cf)
    tficf[i]=tf1*icf
```

- Now from the unique words present we removed the digits and the words having alphabets + digits and hence total unique words got reduced to 22620.
- Now we created a dataframe of the size number of rows * unique words i.e. 1490 * 22621 containing the tficf values of each term in the document.

	0	1	2	3	4	5	6	7	8	9	...	22611	22612	22613	22614	22615	22616	22617	22618	22619	22620
0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1485	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1486	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1487	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1488	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1489	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

1490 rows × 22621 columns

- Now we have calculated probability of each class and probability of each term for the particular given class.

```
[140] print(prob_c1)
      print(prob_c2)
      print(prob_c3)
      print(prob_c4)
      print(prob_c5)

0.22550335570469798
0.17516778523489934
0.18389261744966443
0.23221476510067113
0.18322147651006712
```

```
[144] prob_feature_c5

        ['fulltime', 0.0],
        ['professor', 2.181256472580275e-05],
        ['performing', 8.7250258903211e-05],
        ['university', 0.0],
        ['cincinnati', 0.0],
        ['critic', 0.0],
        ['dancing', 0.0],
        ['gave', 0.0],
        ['illusion', 0.0],
        ['moving', 0.0],
        ['weight', 0.0],
        ['get', 0.0],
        ['station', 0.0],
        ['refuse', 0.0],
        ['adoption', 0.0],
        ['refused', 9.815654126611237e-05],
        ['adopted', 1.0906282362901375e-05],
        ['child', 0.0],
        ['try', 0.0],
        ['pick', 0.0],
        ['birth', 1.0906282362901375e-05],
        ['win', 0.0],
        ['cash', 0.0],
        ['prize', 0.0],
        ['wraztv', 0.0],
```

- Now, training and testing is done with different split ratios using sklearn Naïve-Bayes classifier .

➔ 70:30 split

```
[[151   0   3   0   0]
 [  0  93   0   0   0]
 [ 12   0  99   0   1]
 [  3   0   0 128   0]
 [ 26   0   4   0  76]]
                    0     1           2           3           4  accuracy  \
precision    0.786458   1.0    0.933962    1.000000    0.987013  0.917785
recall       0.980519   1.0    0.883929    0.977099    0.716981  0.917785
f1-score     0.872832   1.0    0.908257    0.988417    0.830601  0.917785
support    154.000000  93.0  112.000000  131.000000  106.000000  0.917785

            macro avg  weighted avg
precision    0.941487      0.930104
recall       0.911706      0.917785
f1-score     0.920021      0.917227
support    596.000000    596.000000
0.9177852348993288
```

➔ 50:50 split

```
[[105   0   1   0   0]
 [  0  70   0   0   0]
 [ 14   0  71   0   2]
 [  6   0   2  92   0]
 [ 19   0   7   4  54]]
                    0     1          2           3          4  accuracy  \
precision    0.729167   1.0   0.876543    0.958333   0.964286  0.876957
recall       0.990566   1.0   0.816092    0.920000   0.642857  0.876957
f1-score     0.840000   1.0   0.845238    0.938776   0.771429  0.876957
support    106.000000  70.0  87.000000  100.000000  84.000000  0.876957

            macro avg  weighted avg
precision    0.905666      0.895714
recall       0.873903      0.876957
f1-score     0.879088      0.875287
support    447.000000    447.000000
0.8769574944071589
```

➔ 80:20 split

```
[[70  0  1  0  0]
 [ 0 56  0  0  0]
 [ 5  0 45  0  0]
 [ 0  0  0 67  0]
 [10  0  2  0 42]]
                   0     1          2     3          4  accuracy   macro avg  \
precision   0.823529   1.0   0.937500   1.0   1.000000  0.939597    0.952206
recall      0.985915   1.0   0.900000   1.0   0.777778  0.939597    0.932739
f1-score    0.897436   1.0   0.918367   1.0   0.875000  0.939597    0.938161
support    71.000000  56.0  50.000000  67.0  54.000000  0.939597  298.000000

           weighted avg
precision      0.947468
recall         0.939597
f1-score       0.939216
support      298.000000
0.9395973154362416
```

➔ 60:40 split

```
[[177   0   2   0   0]
 [  0 123   0   0   0]
 [ 27   0 113   1   0]
 [  2   0   1 161   0]
 [ 33   0  11   5  89]]
                    0      1           2           3           4  accuracy  \
precision    0.740586    1.0    0.889764    0.964072    1.000000  0.889933
recall       0.988827    1.0    0.801418    0.981707    0.644928  0.889933
f1-score     0.846890    1.0    0.843284    0.972810    0.784141  0.889933
support    179.000000  123.0  141.000000  164.000000  138.000000  0.889933

            macro avg  weighted avg
precision    0.918884      0.908898
recall       0.883376      0.889933
f1-score     0.889425      0.887582
support    745.000000    745.000000
0.8899328859060402
```

- After that we computed the accuracy using tfidf weighting scheme.

```
Accuracy: 0.9574944071588367
```

-

➔ Conclusion:

- Preprocessing played a major role, as before when there were 25k unique words at that time the accuracy of the model was comparatively less but further preprocessing the words, it resulted into more accuracy.
- Similarly using lemmatization for large set of corpus was more useful as stemming changed the actual meaning of some words or even changed the actual word.
- Here as per analysis, we get lower accuracy score for 50:50 split ratio, due to less training dataset available for training purposes and get highest accuracy for 70:30 and 80:20 split ratio as we get a much higher dataset and comparatively much accurate result.
- At first we used the tficf weighting scheme which gave comparatively less accuracy as compared to the tfidf weighting scheme though the number of features provided were the same

Q3)

❖ Ranked-Information Retrieval and Evaluation:

- Load the dataset using pandas and separating it by space.
- Extract all the rows having qid:4.
- Now in order to calculate the max dcg we sort the first column containing relevance scores and than calculate the max dcg using the below formula

$$\mathrm{DCG_p} = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)} = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{\log_2(i+1)}$$

```
df1= df.sort_values(by=[0],ascending=False)
```

Calculating Max DCG value

```
maxDcg=0
relevence=0
for i in range(df1.shape[0]):
    relevence=float(df1.iloc[i,0])
    if(i==0):
        maxDcg = relevence
    else:
        maxDcg = maxDcg + (relevence/(math.log2(i+1)))
```

```
maxDcg
```

```
20.989750804831445
```

- Dcg is calculated with the same above formula but without sorting the column of the relevance score and hence dcg is obtained as below

calculating DCG value

```
Dcg=0
for i in range(df.shape[0]):
    relevence=float(df.iloc[i,0])
    if(i==0):
        Dcg=relevence
    else:
        Dcg= Dcg + (relevence/(math.log2(i+1)))
```

```
Dcg
```

```
12.550247459532576
```

- nDCG can be calculated as (Dcg/max Dcg) for the whole dataset calculating NDCG value

```
nDcg = Dcg/maxDcg
```

Normalize DCG of whole dataset

```
nDcg
```

```
0.5979226516897831
```

- Finding the relevance score for each of the unique values of the first column i.e. 0,1,2,3,4 and creating a file that rearranges the query-url pairs in order of the max DCG and stating the number of such files that could be made.

```
Number of rows with 0
: 59
Number of rows with 1
: 26
Number of rows with 3
: 1
Number of rows with 2
: 17
total number of files:  198934973759383705998260476149053298969368401705665705882051803127048579926951934824126865654310502400000000000
```

- Similarly calculating nDCG for top 50 rows:

Normalize DCG of first 50 rows

```
df_50 = df.iloc[:50]
```

DCG for 50 docs

```
Dcg_50=0
for i in range(df_50.shape[0]):
    relevence=float(df_50.iloc[i,0])
    if(i==0):
        Dcg_50=relevence
    else:
        Dcg_50= Dcg_50 + (relevence/(math.log2(i+1)))

Dcg_50
```

```
7.390580969258021
```

```
df1_50 =df1.iloc[:50]
```

Max DCG for 50 docs

```
maxDcg_50=0
relevence=0
for i in range(df1_50.shape[0]):
    relevence=float(df1_50.iloc[i,0])
    if(i==0):
        maxDcg_50 = relevence
    else:
        maxDcg_50 = maxDcg_50 + (relevence/(math.log2(i+1)))

maxDcg_50
```

```
20.989750804831445
```

NDCG for 50 docs

```
Ndcg_50 = Dcg_50/maxDcg_50
```

```
Ndcg_50
```

```
0.3521042740324887
```

- Plotting precision and recall for query qid:4 using the below formula:

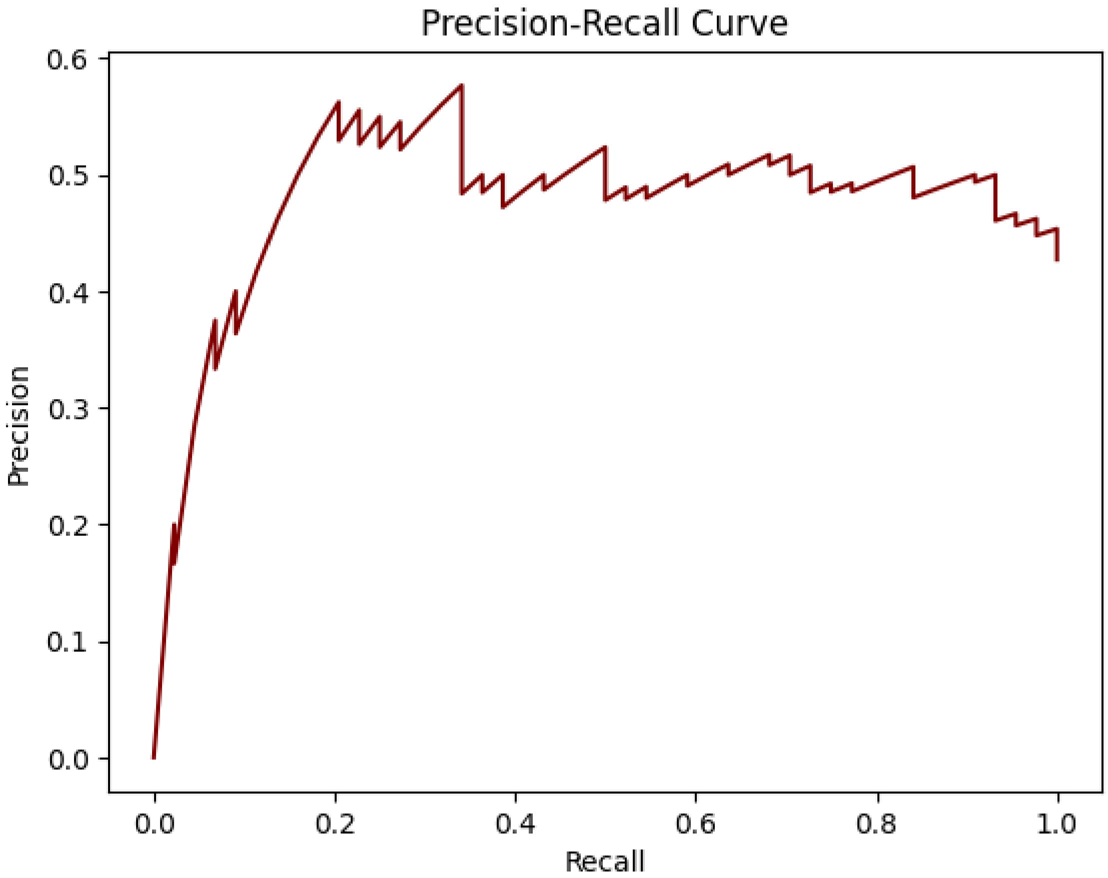
  For Precision:

$$P(relevant \mid retrieved)$$

  For recall:

$$P(retrieved \mid relevant)$$

- The plot obtained is as below:



Precision-Recall Curve
Precision
Recall
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.0
0.2
0.4
0.6
0.8
1.0