Assignment – 2

Report

Solution 1) : In question 1

Part 1):

Assumption:

- All documents are text based , hence equally important.
- Stopwords are not useful in prediction hence we have excluded stopwords.

Preprocessing:

- Punctuations are removed
- Converted the text to lower case
- Remove stopwords
- Tokenise the text
- Eliminate white spaces

Methodology:

- We have made a dictionary consist of document name as key and list of tokens present in that document as value.
- Take input from the user , then convert the input string to list of tokens.
- Then for each document using the dictionary we calculate the jaccard coefficient value between the document's token list and the list of input tokens.
- Then we have made a dictionary ,where key is document name and value is jaccard coefficient between the query list and document list.
- The we have sorted the dictionary and the top 5 documents which has higher value will be given as answer.

Part 2):

Assumptions and preprocessing is same for this part.

Methodology:

- First we have made 3 dictionaries 1st to store token as key and value as a nested dictonaries where keys are documents name and values are count of that token in that document. 2nd to store document name as key and list of token frequencies for token present in that document as value. 3rd to store document name as key and sum of the frequencies of all the token present in that document as value.
- Then we have created a dataframe to where 1st column contains unique tokens and rest of the columns are name of the documents and each document column contains the tfidf value of that token.

- Then we have taken input query , process it and make a vector out of it where the values of the vector represents the tfidf values of the terms present in the query.
- Then using loop we have calculated the similarity between the document vector present in the dataframe and the query vector and top top 5 documents with highest cosine similarity will be our answer.
- We have repeated this procedure for all the 5 different TF values which are mentioned in the question.

Pros and cons for different weighting schemes:

Jaccard coefficient:

- It is less complex.
- It does not use frequency of the term in account.because it works on sets.
- Easy to implement.
- Does not give good results compare to tfidf becuase word frequency is not taken in account.

TF-IDF:

Binary TFIDF:

- Easy to understand and implement.
- Does not take word frequency in account hence perform similar to jaccard coefficient.

Raw Count:

- Easy and efficient because it takes the frequency of the term into account.
- But does not a good approch when the documents are of different sizes.

Term frequency:

- Efficient and length free approch. Because it uses fraction of the frequency from the total frequency.
- Approach is good but the term frequency scheme is not normalised it is naive.

Log normalisation:

- Performs better than others because it uses the log normalised scores of raw count.
- But single normalisation sometimes not good .it can be better.

Double normalisation:

- It is very sophasticated approach as it uses Normalised values and its log.
- Complex to implement but efficient.

Solution 3):

Assumptions are same that we have used in the above question.

Preprocessing:

- Punctuation removal
- Stopword removal
- Convertion of numbers to text
- Uppercase to lower case conversion
- Lemmatization
- Stemming
- Remove the tokens having length 1.

Methodology:

- Created a dataframe where each row represent a document, one column consist of list of all the tokens present in that document and the 2nd column consists of the label(the folder name in which that document present).
- Then we have divided the data into 80:20 ratio for training and testing.
- Then we have created the dictionary in which keys are the token and values are dictionaries in which keys are class name or label name and values are the frequency of that token present in that class. This dictionary will help us to get TF value for the term and also in calculating ICF value for the term.
- Then we have make a dataframe where rows represent different tokens and columns are the class names.
- Then we fill this dataframe with with the TF-ICF value of each token for that class.
- Then we calculate the top k terms from each class using the values present in the column class of the dataframe and sorting it.
- After that we have merged all the top k features of each class and make a set.
- Now we have calculated the probalility of each of this token present in the top fature set for each class.
- Where probability is no.of documents that belongs to that class in which that token present divided by total no. Of documents present in that class.
- We have also calculated the probabilty of the token not present in that class.
- Now while testing we take each row one by one and calculate probabililty of that document for each class. And the class which has highest probability will be declared as the label for that document.
- For calculating the probability we have multiplied the probaility of presence of that top k features tokens in the answer and if that toekn in not present in the testing document then we have multiplued its probility of not presenting in the document,
- We have done the same process for 70:30 and 50:50 split and for different values of k (no of features to be selected).
- And the results of accuracy for different k values are shown as follows:

For 80:20 split

K value	Accuracy in presentage
8	89.5
9	89.5
10	89.7
11	90.2
12	90.8
13	91.1

14	91.0
15	90.9

For 70:30 split

K value	Accuracy in presentage
8	89.6
9	90.8
10	90.8
11	91.13
12	91.66
13	91.80
14	92.26
15	92.26

For 50:50 split

K value	Accuracy in presentage
8	88.12
9	88.36
10	89.6
11	91.4
12	91.44
13	91.64
14	91.48
15	91.96

and the confusion matrix corresponsing to the best accuracies are as follows:

for 80:20 split

```
[[191.   1.   8.   0.   0.]
 [  1. 193.   1.   5.   0.]
 [  0.  68. 131.   1.   0.]
 [  0.   3.   1. 196.   0.]
 [  0.   0.   0.   0. 200.]]
```

For 70:30 split:

```
[[295   0   7   0   0]
 [  0 299   0   0   0]
 [  4   1 289  96   3]
 [  1   0   2 198   1]
 [  0   0   2   6 296]]
```

For 50:50 split:

```
[[493   0   9   0   0]
 [  0 490   0   0   0]
 [  6  10 480 134  13]
 [  1   0   9 350   1]
 [  0   0   2  16 486]]
```

Analysis:

From the above accuracies we can say that 70:30 split is giving good accuracy out of all. And when we have increased the features upto a limit the accuracy increase and then it stops increasing.

Solution 2):

Assumptions and preprocessing steps are same .

Methodology:

Part1

For part 1, we have worked on the queries qid:4 , and their respective relevance score.

Part2

Now our data set is only 103 rows and 139 columns.
For the max DCG, we have to sort the dataset according to the relevance score's descending order.
We have called this data frame newDatta_Part1.
We have saved this data frame by the name of Q2Part2.csv. Now the
question arises: how many such files will be there.
We have taken the relevance values in a list called by dcg_Vales.
Now with the use of collections. On the counter, we have seen the number of occurrences of 3,2,1,0.
Counter({0: 59, 1: 26, 2: 17, 3: 1})
so the, I have made the simple factorial function that is pretty common by givemyFact(). It will give the factorial of any number.
So the number of such files will be calculated by myWay1 =
givemyFact(1) # for 3
myWay2 = givemyFact(17)# for 2
myWay3 = givemyFact(26)# for 1
myWay4 = givemyFact(59) #for 0

And then, we have to multiply these four myWay terms to get our final answers. The Reason is that once 3 can be arranged by myWay1, 2 can be set by myWay2, etc. The multiplication of the 4 ways can give the number of such files.

Part3

For part3, we have to give the value of nDCG at 50. We have
taken the first 50 values,
dcgOne= dcg_Vales1[0:50] idcgOne =
sorted_Vales1[0:50]
In dcgOne, it contains the first 50 relevance values.
In idcgOne, it contains the first 50 sorted values in descending order.

```
def determineNdcg(vales):
  adder = vales[0] + vales[1]
  for idx in range(2,len(vales)-1):
    adder += vales[idx]/math.log2(idx+1) return
  adder
```

The function by the name determineNdcg use the logarithm formula to calculate the dcg value.
SO We apply the formula on both dcgOne and idcgOne,
then we divide the dcOne by idcgOne, we get the nDCG at 50,
Ndcg for the first 50 is 0.5253808413557646

Same thing for the part b, for the whole dataset
dcgWall = determineNdcg(dcg_Vales2) idcgWall =
determineNdcg(sorted_Vales2)
print(dcgWall/idcgWall)
dcg_Vales2 contains all the dcg values.
sorted_Vales contains the dcg values for whole datasets, in descending order.
Then we calculate the nDCG on these two parameters using determineNdcg, we got the value as
0.5979226516897831

Part4

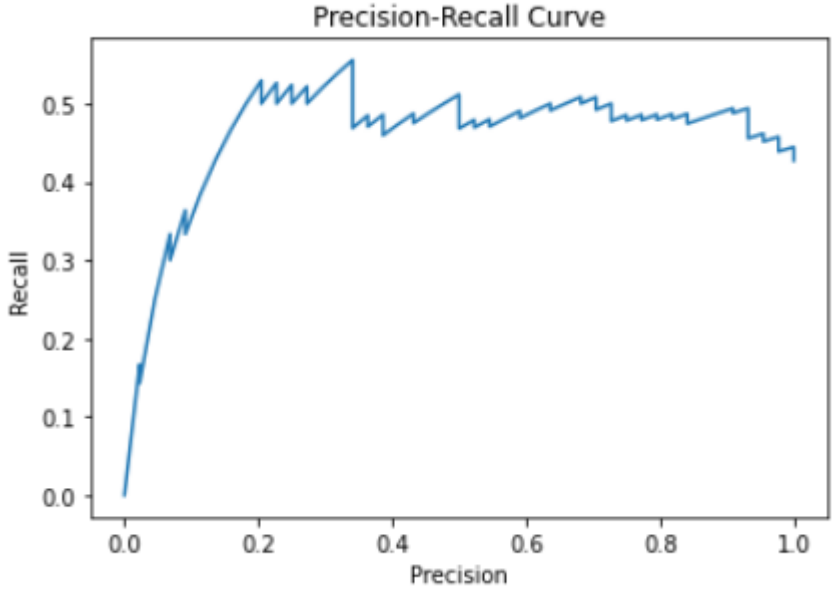
For part4, we have first appended the list relevanceScore with the values of relevances for the feature 75.

```
def changeVals(mydf): arr
  = []

  for idx in range(len(mydf)):
    if (mydf[idx]!=0):
```

graph for :

Recall vs Precision:


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