CSE508 Information Retrieval Assignment-2 Report

Q1.TF IDF and Jaccard Coefficient

1.TF IDF calculation for different weighting scheme

Methodology :

First we need to do preprocessing for both the vocabulary and the query

```
import string

def lowercase(my_list):
  lowercase_list = [string.lower() for string in my_list]
  return my_list

#input is text query and output is a list of strings
def tokenization(my_string):
  li=[]
  li=my_string.split(" ")
  return li

#stopward removal from list of string
def stopword_removal(my_list):
  stop_words = set(stopwords.words('english'))
  processed_list = [word for word in my_list if word not in stop_words]
  return processed_list


def remove_punctuations(my_list):
    processed_list = [word.translate(str.maketrans('', '', string.punctuation)) for word in my_list]
    return processed_list

def remove_blankspace(my_list):
  processed_list = [word.strip() for word in my_list if word.strip()]
  return processed_list
```

After preprocessing the vocabulary we need to build the tf idf matrix for each of the weighting schemes

```
def double_tfidf_calculator(matrix):
  for tf_column in range(1400):
    for tf_row in range(len(matrix)):
      if doub_term_freq[tf_row][tf_column] != 0.5:
        matrix[tf_row][tf_column]=doub_term_freq[tf_row][tf_column]*fp[tf_row]
    return matrix



dp_double_normal_tfidf=double_tfidf_calculator(copy.deepcopy(doub_term_freq))
#printing(dp_double_normal_tfidf)
print(len(dp_double_normal_tfidf),len(dp_double_normal_tfidf[0]))
```

```
7599 1400
```

```
raw_f=np.array(raw_f).T.tolist()
print(len(raw_f),len(raw_f[0]))
```

```
1400 7599
```

```
##traversing the corpus and making the frequency nested dictionary
 (variable) doc_no: Literal[0]
doc_no=0
#iterating through whole corpus which is list of list
for i in corpus:
  small_di={}
  #print(i)
  for j in i:
    #print(j)
    if j not in small_di:
      small_di[j]=1
    else:
      small_di[j]=small_di[j]+1
  large_di[doc_no]=small_di
  doc_no=doc_no+1

# large_di
```

.

Taking input query and preprocessing an then finding the files that are relevant to the below query

```
def preprocessing(query_list):
  query_list=tokenization(query_list)
  my_list=remove_blankspace(query_list)
  my_list=remove_punctuations(my_list)
  my_list=lowercase(my_list)
  my_list=stopword_removal(my_list)
  return my_list

#testing my preprocessing code
print("enter a Query that you want to check")
query_list=input()
input_query=preprocessing(query_list)
preprocessing(query_list)
input_query
```

Query:-in the study of high-speed viscous flow past a two-dimensional body it is usually necessary to consider a curved shock wave emitting from the nose or leading edge of the body . consequently, there exists an inviscid rotational flow region between the shock wave and the boundary layer

Assumption :

We are assuming that the words that are coming in the query will be part of vocabulary otherwise we are deleting the word from the list. Tfidf follows the bag of word concept, and assumes that the order of the words does not matter in defining the relevance. • Tfidf assumes that each term is independent of any other term in the document. We are assuming that the vocabulary is static and that it is not updated at a particular interval.

Result:

The result for different weighting schemes are as follows

```
#for binary tfidf
#first transforming the query_vector in terms of binary_tfidf
query_vector_binary=query_vector.copy()#this is the termfrequency vector of query
# print(query_vector_binary[2571])
# print(fp[2571])

#this is the term frequency vector for binary
for i in range(len(query_vector_binary)):
  if(query_vector_binary[i]>0):
    query_vector_binary[i]=1

  #converting to tfidf for the binary term frequency of the query
query_vector_binary=tf_to_tfidf(query_vector_binary.copy())

#now finding the relevant documents in binary term frequency term
cos_vector=[]
for i in range(len(binary_tfidf)):
  li=[]
  for j in range(len(binary_tfidf[i])):
    li.append(binary_tfidf[i][j])
  cos_vector.append(cosine_similarity(li,query_vector_binary))
  #print(cos_vector)
m=top_5_indexes(cos_vector)
print("the files that are relevant are")
for i in m:
  number_to_file_name(i+1)
```

```
the files that are relevant are
/content/CSE508_Winter2023_Dataset/cranfield0002
/content/CSE508_Winter2023_Dataset/cranfield1267
/content/CSE508_Winter2023_Dataset/cranfield0308
/content/CSE508_Winter2023_Dataset/cranfield0334
/content/CSE508_Winter2023_Dataset/cranfield1309
```

TERM FREQUENCY RELEVANCE//wrong have to change the term frequency calculation

```
#for term frequency  tfidf
#first transforming the query_vector in terms of binary_tfidf
query_vector_term=query_vector.copy()
sum=0
for i in range(len(query_vector_term)):
  sum=sum+query_vector_term[i]
for i in range(len(query_vector_term)):
  query_vector_term[i]=query_vector_term[i]/sum

query_vector_term=tf_to_tfidf(query_vector_term.copy())
#now finding the relevant documents in binary term frequency term
cos_vector=[]
for i in range(len(dp_term_tfidf)):
  li=[]
  for j in range(len(dp_term_tfidf[i])):
    li.append(dp_term_tfidf[i][j])
  cos_vector.append(cosine_similarity(li,query_vector_term))
  #print(cos_vector)
m=top_5_indexes(cos_vector)
print("the files that are relevant are")
for i in m:
  #print(i+1,end=" ")
  number_to_file_name(i+1)
```

```
the files that are relevant are
/content/CSE508_Winter2023_Dataset/cranfield0002
/content/CSE508_Winter2023_Dataset/cranfield0334
/content/CSE508_Winter2023_Dataset/cranfield0439
/content/CSE508_Winter2023_Dataset/cranfield0025
/content/CSE508_Winter2023_Dataset/cranfield0064
```

LOG NORMALIZATION

```
#dp_log_tfidf
#for term frequency  tfidf
#first transforming the query_vector in terms of binary_tfidf
query_vector_log=query_vector.copy()
for i in range(len(query_vector_log)):
  query_vector_log[i]=(math.log(1+query_vector[i],10))

query_vector_log=tf_to_tfidf(query_vector_log.copy())
#now finding the relevant documents in binary term frequency term
cos_vector=[]
for i in range(len(dp_log_tfidf)):
  li=[]
  for j in range(len(dp_log_tfidf[i])):
    li.append(dp_log_tfidf[i][j])
  cos_vector.append(cosine_similarity(li,query_vector_log))
  #print(cos_vector)
m=top_5_indexes(cos_vector)
print("the files that are relevant are")
for i in m:
  #print(i+1,end=" ")
  number_to_file_name(i+1)
```

```
the files that are relevant are
/content/CSE508_Winter2023_Dataset/cranfield0002
/content/CSE508_Winter2023_Dataset/cranfield0334
/content/CSE508_Winter2023_Dataset/cranfield0439
/content/CSE508_Winter2023_Dataset/cranfield1267
/content/CSE508_Winter2023_Dataset/cranfield0192
```

```
#double_normal
import numpy as np
#for term frequency  tfidf
#first transforming the query_vector in terms of binary_tfidf
query_vector_double_normal=query_vector.copy()
#i need to find the maximum term frequency of terms of that query

# max=-1
# for i in range(len(query_vector_double_normal)):
#   if query_vector_double_normal[i]>max:
#     max=query_vector_double_normal[i]

max = np.max(query_vector_double_normal)

for i in range(len(query_vector_double_normal)):
  query_vector_double_normal[i]=0.5+0.5*(query_vector[i]/max)


query_vector_double_normal=tf_to_tfidf2(query_vector_double_normal.copy())
#now finding the relevant documents in binary term frequency term
cos_vector=[]
for i in range(len(dp_double_normal_tfidf)):
  li=[]
  for j in range(len(dp_double_normal_tfidf[i])):
    li.append(dp_double_normal_tfidf[i][j])
  cos_vector.append(cosine_similarity(li,query_vector_double_normal))
  #print(cos_vector)
m=top_5_indexes(cos_vector)
print("the files that are relevant are")
for i in m:
  #print(i+1,end=" ")
  number_to_file_name(i+1)
```

Jaccard Coefficient Methodologies:

First pre-processed the data as per question and then pre processed the query.

```
#testing my preprocessing code
print("enter a Query that you want to check")
query_list=input()

def preprocessing(query_list):
  query_list=tokenization(query_list)
  my_list=remove_blankspace(query_list)
  my_list=remove_punctuations(my_list)
  my_list=lowercase(my_list)
  my_list=stopword_removal(my_list)
  return my_list


my_query_list=preprocessing(query_list)
preprocessing(query_list)
```

We are finding the jaccard coefficient one by one for each document and then sorting in descending order

```
# i have the preprocessed word  in query which is in temp
#i have the vocabulary list in result
#we have a list of list for the vocabulary document by document
jaccard_vector=[]
for i in range(len(corpus)):
  s1=set(corpus[i].copy())
  s2=set(temp.copy())
  resU=s1.union(s2)
  resI=s1.intersection(s2)
  jaccard_vector.append(len(resI)/len(resU))


jaccard_vector
```

Given the below query this is the output that we are getting:-in the study of high-speed viscous flow past a two-dimensional body it is usually necessary to consider a curved shock wave emitting from the nose or leading edge of the body . Consequently, there exists an inviscid rotational flow region between the shock wave and the boundary layer.

```
#this willl give the file name that are relevant with th
m=top_10_indexes(jaccard_vector)
for i in m:
  #print(i+1,end=" ")
  number_to_file_name(i+1)
```

```
/content/CSE508_Winter2023_Dataset/cranfield0002
/content/CSE508_Winter2023_Dataset/cranfield1267
/content/CSE508_Winter2023_Dataset/cranfield0308
/content/CSE508_Winter2023_Dataset/cranfield0666
/content/CSE508_Winter2023_Dataset/cranfield0327
/content/CSE508_Winter2023_Dataset/cranfield0537
/content/CSE508_Winter2023_Dataset/cranfield0192
/content/CSE508_Winter2023_Dataset/cranfield0003
/content/CSE508_Winter2023_Dataset/cranfield0310
/content/CSE508_Winter2023_Dataset/cranfield1228
```

Binary weighting:

Pros:
It is easy and simple compared to other normalizations.
Is used for plagiarism detection, since the are present or absent is represented binary.

Cons:
The frequency of a particular term does not matter in the result.

Raw count weighting:

Pros:
It is also simple to compute
It is also efficient.

Cons:
Can get biased towards more frequently occurring terms
It may deviate from giving real relevant documents since it is the simplest of all.

Term frequency weighting:

Pros:
It takes into account the frequency of the rems in each document

Cons:
It gives more weightage to longer document which is not true always.

Log normalisation weighting:

Pros:
It compresses the value inside a certain interval so it is easy to calculate with accuracy
It is more robust to document length and term frequency.

Cons:
It seems to me that it is computation wise expensive than the before few techniques but easier than double normalization

Double normalisation weighting:

Pros:
The impact of common repetitive terms is eliminated to most extent.
It is a perfect example of balancing the high frequency and low frequency term.

Cons:
It may not work well for very short documents.
Computationally more expensive than other weighting schemes.

References

GeeksforGeeks
Stackoverflow

Q2. Naive Bayes Classifier with TF-ICF

1. Preprocessing the dataset.

Methodology:

First we are dropping the "ArticleId" column as it is not necessary. Then we are pre-processing the text by converting it to lowercase,removing the punctuations and stop words , tokenizing it and then performing lemmatization. After all the pre-processing steps are done we implement the TF-ICF weighting scheme. For TF-ICF calculation we are first calculating the term frequency and the class frequency then using the provided formula calculating the TF-ICF for each word. Then we are creating an empty dataframe having columns as the unique words and rows as the number of sentences in the original dataframe and then adding the value of respective row,column as per the TF-ICF calculated.

Assumptions:

We are assuming that the words in the dataset are all correct as per English dictionary.

Results:

After dropping "ArticleId" Column

	Text	Category
0	worldcom ex-boss launches defence lawyers defe...	business
1	german business confidence slides german busin...	business
2	bbc poll indicates economic gloom citizens in ...	business
3	lifestyle governs mobile choice faster bett...	tech
4	enron bosses in $168m payout eighteen former e...	business

After removing punctuations

	Text	Category
0	worldcom exboss launches defence lawyers defen...	business
1	german business confidence slides german busin...	business
2	bbc poll indicates economic gloom citizens in ...	business
3	lifestyle governs mobile choice faster bett...	tech
4	enron bosses in 168m payout eighteen former en...	business

After removing stopwords

	Text	Category
0	worldcom exboss launches defence lawyers defen...	business
1	german business confidence slides german busin...	business
2	bbc poll indicates economic gloom citizens maj...	business
3	lifestyle governs mobile choice faster better ...	tech
4	enron bosses 168m payout eighteen former enron...	business

After performing lowercase

	Text	Category
0	worldcom exboss launches defence lawyers defen...	business
1	german business confidence slides german busin...	business
2	bbc poll indicates economic gloom citizens maj...	business
3	lifestyle governs mobile choice faster better ...	tech
4	enron bosses 168m payout eighteen former enron...	business

After performing tokenization

	Text	Category
0	[worldcom, exboss, launches, defence, lawyers,...	business
1	[german, business, confidence, slides, german,...	business
2	[bbc, poll, indicates, economic, gloom, citize...	business
3	[lifestyle, governs, mobile, choice, faster, b...	tech
4	[enron, bosses, 168m, payout, eighteen, former...	business

After performing lemmatization

	Text	Category
0	[worldcom, exboss, launch, defence, lawyer, de...	business
1	[german, business, confidence, slide, german, ...	business
2	[bbc, poll, indicate, economic, gloom, citizen...	business
3	[lifestyle, governs, mobile, choice, faster, w...	tech
4	[enron, boss, 168m, payout, eighteen, former, ...	business

New DataFrame based on TF-ICF values

worldcom	exboss	launch	defence	lawyer	defend	former	chief	bernie	ebbers	...
37.74438	1.39794	0.0	0.0	0.0	0.0	0.0	0.0	5.59176	31.45365	...
0.00000	0.00000	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.00000	...
0.00000	0.00000	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.00000	...
0.00000	0.00000	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.00000	...
0.00000	0.00000	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.00000	...

2. The dataset:

Methodology:

We are splitting the dataset using list slicing.

Assumptions:

We are assuming that all the preprocessing has been correctly above.

3.Training the Naive Bayes classifier with TF-ICF:

Methodology:

After splitting the dataset we are training on Multinomial Naive Bayes classifiers and getting the Y-predicted. To calculate the probability of each category we are simply iterating over the Y-train.

Assumptions:

We are assuming that all the preprocessing has been correctly above and that we can use the Multinomial NaiveBayes directly from sklearn library.

Results:

Probability of each category

```
Probability of business :  0.22722914669223393
Probability of tech :  0.17833173537871524
Probability of politics :  0.18024928092042186
Probability of sport :  0.23873441994247363
Probability of entertainment :  0.1754541706615533
```

4. Testing the Naive Bayes classifier with TF-ICF:

Methodology:

We use different metrics to evaluate the performance of the classifier.

Results:

```
Accuracy :  0.9977628635346756
Precision :  0.998
Recall :  0.9976744186046511
F1 score :  0.997825384231097
```

5. Improving the classifier:

Methodology:

We are experimenting with a different pre-processing technique (stemming) and different split sizes. We have also tried out using TF-IDF vectorizer.

When using stemming and TF-ICF for split size 0.6

```
df=pd.read_csv("BBC News Train.csv",encoding = "ISO-8859-1")
df=df.drop(['ArticleId'], axis=1)
df['Text'] = df['Text'].apply(removePunctuation)
df['Text'] = df['Text'].apply(removeStopwords)
df['Text'] = df['Text'].apply(tolower)
df['Text'] = df['Text'].apply(tokenization)
#df['Text'] = df['Text'].apply(lemmatize)
df['Text'] = df['Text'].apply(stemming)
diff(df,0.6)
```

```
Accuracy :  0.9983221476510067
Precision :  0.9984848484848484
Recall :  0.9983606557377049
F1 score :  0.998416498458746
```

When using stemming and TF-ICF for split size 0.8

```
df=pd.read_csv("BBC News Train.csv",encoding = "ISO-8859-1")
df=df.drop(['ArticleId'], axis=1)
df['Text'] = df['Text'].apply(removePunctuation)
df['Text'] = df['Text'].apply(removeStopwords)
df['Text'] = df['Text'].apply(tolower)
df['Text'] = df['Text'].apply(tokenization)
#df['Text'] = df['Text'].apply(lemmatize)
df['Text'] = df['Text'].apply(stemming)
diff(df,0.8)
```

```
Accuracy :  0.9966442953020134
Precision :  0.9971014492753623
Recall :  0.9966101694915255
F1 score :  0.9968307442759997
```

When using lemmatization and TF-ICF for split size 0.5

```
df=pd.read_csv("BBC News Train.csv",encoding = "ISO-8859-1")
df=df.drop(['ArticleId'], axis=1)
df['Text'] = df['Text'].apply(removePunctuation)
df['Text'] = df['Text'].apply(removeStopwords)
df['Text'] = df['Text'].apply(tolower)
df['Text'] = df['Text'].apply(tokenization)
df['Text'] = df['Text'].apply(lemmatize)
#df['Text'] = df['Text'].apply(stemming)
diff(df,0.5)
```

```
Accuracy :  0.9986577181208054
Precision :  0.9987341772151899
Recall :  0.9986842105263157
F1 score :  0.9987050133584787
```

When using stemming and TF-IDF for split size 0.7

```
Accuracy :  0.970917225950783
Precision :  0.9713831281500453
Recall :  0.9699074465586094
F1 score :  0.9703292402737617
```

6. Conclusion

As we can see from the above results TF-IDF gives the lowest accuracy , thus tf-icf is a better weighing scheme. For TF-ICF for different split sizes and different pre-processing techniques the accuracy is almost the same.

References

Stackoverflow
GeeksforGeeks
Medium
Analytics Vidhya

Q3.Ranked-Information Retrieval and Evaluation

i) The first objective is to create a file that rearranges the query-url pairs in order of the maximum DCG (discounted cumulative gain). The number of such files that could be made should also be stated.

Methodology:

We are selecting the queries with qid:4 and using the relevance judgment labels as the relevance score.Then,maintaining a list that consists of a tuple with index and relevance score.
Sort the list using the lambda function on the basis of the second element as a key.Create a file with the query-url pairs in order of the maximum DCG (discounted cumulative gain).
Next,we have counted the number of files with (max) DCG.

Assumptions:

The dataset contains various and their associated URLs with relevance judgment labels as relevance scores but here we have considered the queries with qid:4 using the relevance judgment label as the relevance score.

Results:

```
[10] sort_list(indexRelList)
     indexRelList
```

```
[(7, 3),
 (18, 2),
 (19, 2),
 (21, 2),
 (22, 2),
 (25, 2),
 (34, 2),
 (36, 2),
 (37, 2),
 (40, 2),
 (52, 2),
 (58, 2),
 (61, 2),
 (62, 2),
 (68, 2),
 (76, 2),
 (90, 2),
 (100, 2),
 (4, 1),
 (6, 1),
 (10, 1),
```

```
13] print("The total number of files with (max) dcg are:  ", total_no_of_files(indexRelList))

    1! * 17! * 26! * 59!
    The total number of files with (max) dcg are:   198934973759383705998260476149053298969368401705665705882051803127048579926951934824126865654310
```

ii) The second task is to compute the nDCG (normalized discounted cumulative gain) for the dataset. This involves calculating nDCG at position 50 and for the entire dataset.

Methodology:
To calculate the nDCG firstly we have calculated the maximum DCG for the ideal case.
Then,compute the nDCG at the position 50 and nDCG for the entire dataset.

Assumption:
We have considered the index and relevance score tuples in the sorted order of the max(DCG) that is already given in the first query to compute the nDCG at position 50 and the entire dataset.

Results:

```
dcg_max = maximum_Dcg(indexRelList)
print("The maximum DCG for the ideal case is: ",dcg_max)
```

```
The maximum DCG for the ideal case is:  20.989750804831445
```

```
entire_ndcg = nDCG(dataframe_id_q4,-100)
ndcg_at_50 = nDCG(dataframe_id_q4,50)
print("The calculated NDCG at postion 50 is:",ndcg_at_50)
print("The calculated NDCG for entire dataset:",entire_ndcg)
```

```
The calculated NDCG at postion 50 is: 0.3521042740324887
The calculated NDCG for entire dataset: 0.5979226516897831
```

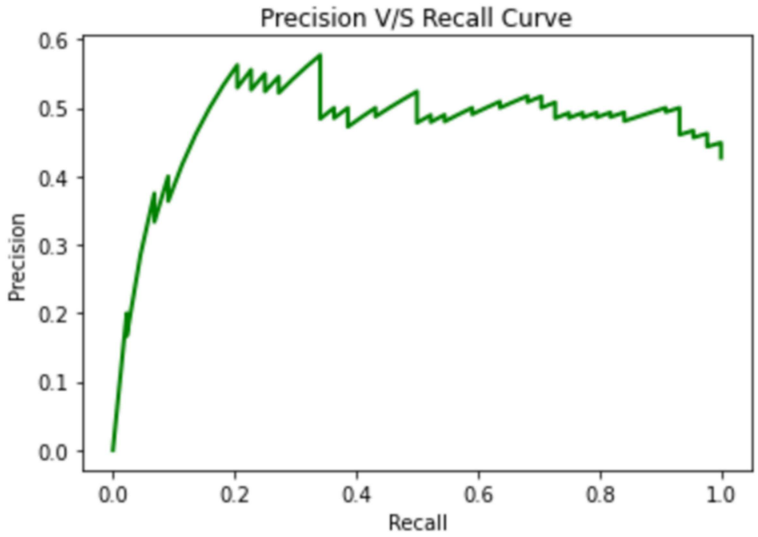
iii) The third objective is to assume a model that simply ranks URLs on the basis of the value of feature 75 (sum of TF-IDF on the whole document) i.e.URL with higher feature values are considered more relevant. Any non zero relevance judgment value to be relevant. Plot a Precision-Recall curve for query "qid:4".

Methodology:
We have extracted the relevance scores and the IDF scores from the input Data Frame and created its tuple and then sorted the tuple list in the descending order by the IDF score.Also,calculated the number of documents to plot the Precision-Recall curve.
First,we have calculated the relevance score and IDF score for each document and then the total number of the relevant documents in the dataset.Then computed the precision and recall for each document in the dataset.Then,plot graph for the same.(Precision-Recall)

Assumption:
We have assumed that the relevance score is in column 0 and the IDF score is in column 76.

Results:

```
calculate_Prec_Rec(dataframe_id_q4)
```



.

References:

GeeksforGeeks
Stackoverflow