CCSE508: Information Retrieval
Assignment 2

Group- 76
Kshitij Mohan (2019054) - (Github: Kshitij-M)
Shreya Tomar (2019110) - (Github: Shreya0229)

TA assigned- Deepi Garg - (Github: deepigarg)

Q1.

Preprocessing: We have followed the following steps for preprocessing:

- Converted text to lower case
- Removed punctuations
- Removed stop words
- Lemmatized the dataset

First Part:

After preprocessing, We tokenised the all the documents and stored them in a numpy array.

For a given input query, The query is preprocessed and then tokenised.
We then compute Jaccard Similarity with all the input documents and sort them in increasing order. The top 5 similar documents are then shown as result

Second Part:

For this part, We first got the entire corpus and then implemented the 5 variants of Term Frequency.

```python
def tf(word, counter):
    return counter[word] / len(counter)

def tf_binary(word, counter):
    if counter[word]>0:
        return 1
    return 0

def tf_rawcount(word, counter):
    return counter[word]

def tf_lognorm(word, counter):
    return math.log(1+counter[word])

def tf_doublenorm(word, counter, max_count):
    return 0.5+(0.5*counter[word])/max_count

def idf(word, postings):
    return math.log(len(postings) / (1 + postings[word]))
```

Vectors are then generated for every document in the dataset using these 5 schemes. The 5 generated vector matrices are then stored as a numpy array.
For a given input query, We first do the preprocessing and then tokenise the resultant query.

To get the relevant document, We iterate through the tokens and for each tokens, We search our vector matrix and get the Tfidf score for each of these tokens. These are then added for all tokens. The final scores are stored in a dataframe.

	file	binary	raw count	term frequency	log normalization	double normalization
0	1st_aid.txt	27.156528	65.518911	0.431045	31.906132	19.038173
1	a-team	18.082741	18.082741	0.009297	12.534001	9.161922
2	a_fish_c.apo	0.000000	0.000000	0.000000	0.000000	0.000000
3	a_tv_t-p.com	12.000740	12.000740	0.015749	8.318279	6.100376
4	abbott.txt	0.000000	0.000000	0.000000	0.000000	0.000000

Top k documents are shown based on these final scores.

Binary:

	file	binary
501	humor9.txt	31.606085
663	mlverb.hum	31.587736
0	1st_aid.txt	27.156528
349	epikarat.txt	25.633924
9	acronym.lis	25.163920
174	byfb.txt	25.001579
228	coffee.faq	25.001579
252	consp.txt	23.839139
173	bw.txt	23.839139
449	hack7.txt	23.369135

Raw Count:

	file	raw count
663	mlverb.hum	141.212007
11	acronyms.txt	124.772955
817	practica.txt	113.339979
630	manners.txt	113.017486
175	c0dez.txt	104.795726
173	bw.txt	85.101122
483	hop.faq	79.578766
450	hackingcracking.txt	75.459351
39	anime.lif	67.439132
37	anim_lif.txt	67.439132

Term Frequency:

	file	term frequency
0	1st_aid.txt	0.431045
389	flowchrt	0.273309
390	flowchrt.txt	0.149078
413	gas.txt	0.106150
36	aniherb.txt	0.095148
171	bw-phwan.hat	0.081993
723	nukewar.jok	0.077815
1006	temphell.jok	0.075313
591	lifeinfo.hum	0.074990
646	memo.hum	0.074663

Log Normalization

	file	log normalization
663	mlverb.hum	39.010196
817	practica.txt	34.105551
173	bw.txt	33.011395
501	humor9.txt	32.779866
0	1st_aid.txt	31.906132
175	c0dez.txt	30.643494
174	byfb.txt	30.610260
630	manners.txt	29.302768
449	hack7.txt	28.849454
450	hackingcracking.txt	28.048332

Double Normalization

	file	double normalization
0	1st_aid.txt	19.038173
663	mlverb.hum	16.152274
501	humor9.txt	16.027808
9	acronym.lis	13.949391
349	epikarat.txt	12.847406
173	bw.txt	12.806039
174	byfb.txt	12.793575
228	coffee.faq	12.634303
252	consp.txt	12.121596
273	cultmov.faq	11.963350

In Jaccard coefficient, both kinds of variables i.e. continuous and categorical can be used easily, which is a good thing. Moreover, it can be of varied sizes and not necessarily of the equal size. But on the other hand, it gives more emphasis to the rare terms ignoring the fact that the terms which come more often could be more relevant too. It is not meant to work properly on the nominal data too.

If we see TF-IDF, it is based on a metric that is used to extract and get the major descriptive terms. It is easier to compute and hence is a good thing of this technique. On the other hand, TF-IDF has problems in capturing the semantics like positions in text, the occurrence of words in different documents etc. It is based on the bag of words i.e. BoW method which is why this happens, which is a con of this technique.

Q2.

One formula which is used for calculating the discounted cumulative gain is:

$$DCG = \sum_{i=1}^{5} \frac{rel_i}{\log_2(i+1)}$$

and the alternate formula which is used is

```
def alt_DCG(relevance_score):
    res_alt= 0
    for i in range (1, len(relevance_score)):
        vall= float(relevance_score[i-1])
        res_alt += float(pow(2, vall)-1)/math.log2(1+i)
```

These are used to retrieve the most relevant documents.

1. We have considered the queries with qid:4. The relevance judgement labels are considered as relevance scores.

2. The query-url pair file is made using the order of max DCG. It is saved in text form using the name output_assign2 name. The files of qid 4 are total 103 files. The number of such files that can be made by rearranging in different ways are:

```
59 26 17 1 0
Total number of files:  19893497375938370599826047614905329896936840170566570588205180312704857992695193482412686565431050240000000000000000000000
```

   which is the 59! 26! 17! 1! 0! factorial product. These are the number of documents with relevance from 0 to 4 respectively.

3. The nDCG of the documents are calculated:

   a) At 50: The documents are sorted and then top 50 are taken for the dcg using both the formulas mentioned above.

```
relevance_score_50, relevance_score_sort_50= sliced_dict(relevance_score, relevance_score_sort, k)

list_relevance=[]
rel_list_sorted=[]

list_relevance, count= get_list_relevance(relevance_score_50)

for i in relevance_score_sort_50.keys():
    rel_list_sorted.append(relevance_score_sort_50[i])

ndcg= final_ndcg(list_relevance, rel_list_sorted)
print('nDCG for 50 documents: ',ndcg)
```

   The output which we got after this is as shown:

```
nDCG for whole document:  0.5979226516897831
nDCG for 50 documents:  0.3521042740324887


 Using Alternate DCG Formula
 nDCG for whole document:  0.5784691984582591
 nDCG for 50 documents:  0.35612494416255847
```

b) For the whole dataset: Similarly, the nDCG for the whole dataset is calculated using both the formulas of the DCG as written above.

```
# b) for whole document

relevance_score_sort= relevance_score
relevance_score_sort= get_sorted(relevance_score_sort)

list_relevance=[]
rel_list_sorted=[]

list_relevance, count= get_list_relevance(relevance_score)

for i in relevance_score_sort.keys():
    rel_list_sorted.append(relevance_score_sort[i])

ndcg= final_ndcg(list_relevance, rel_list_sorted)
print('nDCG for whole document: ', ndcg)
```

The output which we got after this is as shown:

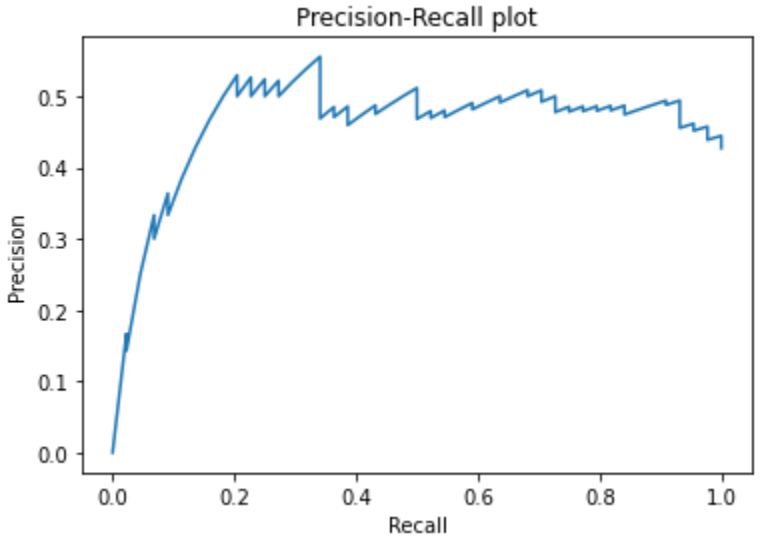
```
nDCG for whole document:  0.5979226516897831
nDCG for 50 documents:  0.3521042740324887


 Using Alternate DCG Formula
 nDCG for whole document:  0.5784691984582591
 nDCG for 50 documents:  0.35612494416255847
```

4. We have taken a model that simply ranks URLs on the basis of the value of feature 75 (sum of TF-IDF on the whole document). So here the higher the value of the feature 75, the more relevant the URL. The precision and recall are calculated as follows:

```
# the precision and recall is calculated using their repsective formulas
for rel in rel_score_list:
    precision.append(cummulative/count)
    count += 1
    recall.append(cummulative/total)
    if rel > '0':
        cummulative += 1
```

The final output plot of the precision-recall for qid:4 is as shown:



Q3

Preprocessing: We have followed the following steps for preprocessing:

- Converted text to lower case
- Removed punctuations
- Removed stop words
- Lemmatized the dataset

The resultant preprocessed documents are grouped by their classes and their texts are combined.
Now TF-ICF values are computed for every class. For each class, We get the top k features based on their TF-ICF values.

```
def tf(word, counter):
    return counter[word] / len(counter)

def icf(word, postings):
    return math.log(len(postings) / (1 + postings[word]))

def tficf(word, blob, postings):
    return tf(word, blob) * icf(word, postings)

def get_corpus(df):
    all_text = TextBlob(' '.join(df['text']))
    counter=collections.Counter(list(all_text.words))

    index = 0
    for key, value in counter.items():
        counter[key] = index
        index+=1

    return counter
```

The new corpus is generated based on these features. The vector matrix is defined and we calculate the TF-ICF score for each tokens in the document. Using these, We generate the vectors corresponding to every document and generate the vector matrix.

We then implemented Naive Bayes Algorithm for the classification.

```
def train(X, y, e):
    y_labels, y_value_counts = np.unique(y, return_counts=True)
    x_sep = np.array([np.unique(x) for x in np.transpose(X)])
    dell_y = 1.0*(y_value_counts/y_value_counts.sum())
    u_mean = np.array([X[y==k].mean(axis=0) for k in y_labels])
    prob_x = np.array([X[y==k].var(axis=0) + e for k in y_labels])
    return [prob_x, u_mean, y_labels]

def predict(X, prob_x, u_mean, y_labels):
    return np.apply_along_axis(lambda x: compute_probs(x, prob_x, u_mean, y_labels), 1, X)

def compute_probs(x, prob_x, u_mean, y_labels):
    probs = np.array([get_weight(x, y, prob_x, u_mean) for y in range(len(y_labels))])
    return y_labels[np.argmax(probs)]

def get_weight(x, y, prob_x, u_mean):
    c = 1.0 /np.sqrt(2.0 * np.pi * (prob_x[y]))
    return np.prod(c * np.exp(-1.0 * np.square(x - u_mean[y]) / (2.0 * prob_x[y])))
```

The data was stratified and split which gave us much better results. Smoothing was also applied while calculating the probabilities which resulted in even better accuracy.

Split - 0.2

```
[108] split = 0.2
      X_train, X_test, y_train, y_test = train_test_split(np.array(vecto

      weights = train(X_train, y_train, 1e5)
      preds = predict(X_test, weights[0], weights[1], weights[2])

      print(f'Accuracy: {accuracy_score(y_test, preds)}')
      print(classification_report(y_test, preds))
```

```
Accuracy: 0.986
              precision    recall  f1-score   support

           0       0.99      0.96      0.98       200
           1       1.00      0.98      0.99       200
           2       0.95      1.00      0.98       200
           3       1.00      1.00      1.00       200
           4       0.99      0.98      0.98       200

    accuracy                           0.99      1000
   macro avg       0.99      0.99      0.99      1000
weighted avg       0.99      0.99      0.99      1000
```

Split - 0.3

```
split = 0.3
X_train, X_test, y_train, y_test = train_test_split(np.array(vec

weights = train(X_train, y_train, 1e5)
preds = predict(X_test, weights[0], weights[1], weights[2])

print(f'Accuracy: {accuracy_score(y_test, preds)}')
print(classification_report(y_test, preds))
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:3:
  This is separate from the ipykernel package so we can avoid do
Accuracy: 0.9853333333333333
              precision    recall  f1-score   support

           0       0.99      0.96      0.98       300
           1       1.00      0.99      0.99       300
           2       0.96      1.00      0.98       300
           3       1.00      1.00      1.00       300
           4       0.98      0.98      0.98       300

    accuracy                           0.99      1500
   macro avg       0.99      0.99      0.99      1500
weighted avg       0.99      0.99      0.99      1500
```

Split - 0.5

```
split = 0.5
X_train, X_test, y_train, y_test = train_test_split(np.array(ve

weights = train(X_train, y_train, 1e5)
preds = predict(X_test, weights[0], weights[1], weights[2])

print(f'Accuracy: {accuracy_score(y_test, preds)}')
print(classification_report(y_test, preds))
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:3:
  This is separate from the ipykernel package so we can avoid d
Accuracy: 0.9848
              precision    recall  f1-score   support

           0       1.00      0.97      0.98       500
           1       1.00      0.98      0.99       500
           2       0.96      1.00      0.98       500
           3       1.00      1.00      1.00       500
           4       0.97      0.97      0.97       500

    accuracy                           0.98      2500
   macro avg       0.99      0.98      0.98      2500
weighted avg       0.99      0.98      0.98      2500
```

After analyzing we got to know that as we increase the data i.e. the training data the accuracy increases as less overfitting would be there since diverse data is available to train. Then the test accuracy is increased due to this diverse data. Similarly, less training data is generally not that diverse and hence poor performance on the output. Also as the training and testing data distribution should be similar, Stratifying the dataset based on classes is a good idea to train the model.

References:

https://www.geeksforgeeks.org/normalized-discounted-cumulative-gain-multilabel-ranking-metrics-ml/
https://williamscott701.medium.com/information-retrieval-unigram-postings-and-positional-postings-a28b907c4e8