IR

Assignment 2

Akhil Majahan (MT20107)

Anjali (MT20082)

Prabal Jain (MT20115)

Shradha Sabhlok(MT20069)

Question 1 : Positional Index

Dataset : Stories.zip contains 3 Folders Stories, Farcon, SRE in which there is total of 467 Documents

1. Preprocessing :

   i. Convert the text to lowercase : Text is converted to lowercase with the help of **`convert_lower_case()`**.

   ii. Perform word tokenization : Tokenization with the help of **`word_tokenize()`**.

   iii. Remove stopwords from tokens : **`remove_stop_words()`** is used to remove Stopwords.

   iv. Remove punctuation marks from tokens : **`remove_punctuation()`** is used to remove punctuation.

   v. Remove blank space tokens : Blank space tokens are removed inside remove_punctuation() method.

Output :

```
[34] test_before = "Oh, those SCREAMS!! Etano jerked straight up. Sweat poured off his body."

[35] print('Test Before : {}'.format(test_before))
     print('Test After : {}'.format(preprocess(test_before, False)))

     Test Before : Oh, those SCREAMS!! Etano jerked straight up. Sweat poured off his body.
     Test After :  oh screams etano jerked straight sweat poured body
```

2. Implement the positional index data structure

   Methodology : Since Inverted index is not Sufficient for Phrase queries like (“Stanford university”, “Where is my Guitar” etc.) so we are implementing positional Index Data Structure in which Firstly we will fetch the document then performing the appropriate preprocessing such as removing punctuation, stopwords, blank tokens and converting text to lowercase etc. then we will perform tokenization if the word is already present in our dictionary then we have to add the document and appropriate positions it appears in. Also we will update the Frequency of word in each Document as well as the total Frequency it appears in.

```
[ ] pos_index

{'shareware': [6, {0: [0], 42: [0], 76: [6], 78: [9730], 126: [15, 19]}],
 'trial': [50,
  {0: [1],
   26: [13200, 13203],
   41: [1564],
   42: [1],
```

For an example Position index for a Term “day” will be

```
[ ] print(pos_index['day'])

[1799, {0: [492, 498, 539, 609, 745, 785, 875, 876, 945, 1263, 1289, 1305, 1359, 1360, 1722], 1: [128, 133], 2: [303], 3:
```

Which shows the term “day” appeared 1799 times and the dictionary shows as key : docID and value as a list of indexes where the term appears in it.

3. Searching of Phrase Query :

   Methodology : A test_phrase Query is given such that firstly we will preprocess the query with the same preprocessing technique we applied to the set of Documents, then after preprocessing we will Tokenize our phrase query and for each word as a key we will retrieve their postings which will contain total frequency and a dictionary that contains in which document where it appears. Here we will initialize a variable count which will show us the distance between the tokens so each time we account into a new token from our phrase query we will perform “AND” of the postings list from positional index data structure and if there exists appropriate difference between the indexes of token then we will add that document to our dictionary “result” which will contains key as document ID and value as a list of indexes where the phrase appeared.

4. Evaluation :

   Without Using Lemmatization in Preprocessing :

```
[76] test_phrase = "Good day"
     preprocessed_text = preprocess(test_phrase, False)
     print(preprocessed_text)
     # test_phrase_lemmat = [lemmatizer.lemmatize(each_word) for each_word in word_tokenize(str(preprocessed_text))]
     test_phrase_lemat =  word_tokenize(str(preprocessed_text))

     good day
```

```
[19] # Length of documents retrived
     print("Number of Documents retrived are :" ,len(result))

     Number of Documents retrived are : 19

[20] for id in list(result.keys()):
       print(dataset[id][0].split('/')[6:])

     ['stories', '13chil.txt']
     ['stories', 'aesopa10.txt']
     ['stories', 'brain.damage']
     ['stories', 'breaks2.asc']
     ['stories', 'enchdup.hum']
     ['stories', 'fantasy.hum']
     ['stories', 'fantasy.txt']
     ['stories', 'fic5']
     ['stories', 'forgotte']
     ['stories', 'history5.txt']
     ['stories', 'horswolf.txt']
     ['stories', 'hound-b.txt']
     ['stories', 'mazarin.txt']
     ['stories', 'melissa.txt']
     ['stories', 'outcast.dos']
     ['stories', 'sick-kid.txt']
     ['stories', 'startrek.txt']
     ['stories', 'superg1']
     ['stories', 'SRE', 'srex.txt']
```

With Lemmatization :

```
[36] # Length of documents retrived
     print("Number of Documents retrived are :" ,len(result))

     Number of Documents retrived are : 21

     for id in list(result.keys()):
       print(dataset[id][0].split('/')[6:])

     ['stories', '13chil.txt']
     ['stories', 'aesop11.txt']
     ['stories', 'aesopa10.txt']
     ['stories', 'brain.damage']
     ['stories', 'breaks2.asc']
     ['stories', 'bruce-p.txt']
     ['stories', 'enchdup.hum']
     ['stories', 'fantasy.hum']
     ['stories', 'fantasy.txt']
     ['stories', 'fic5']
     ['stories', 'forgotte']
     ['stories', 'history5.txt']
     ['stories', 'horswolf.txt']
     ['stories', 'hound-b.txt']
     ['stories', 'mazarin.txt']
     ['stories', 'melissa.txt']
     ['stories', 'outcast.dos']
     ['stories', 'sick-kid.txt']
     ['stories', 'startrek.txt']
     ['stories', 'superg1']
     ['stories', 'SRE', 'srex.txt']
```

Question 2 : Scoring and Term Weighting

JACCARD COEFFICIENT : It is one of the similarity measures that is used to get similarity between any two documents. The value lies between 0 and 1. 0 show that documents are dissimilar and 1 show those documents are identical with each other. Values between 0 and 1 show the probability of similarity between the documents. Here we are using Jaccard Coefficient to get the similarity values between input query and the documents in the stories dataset. It is given as below where A and B can be considered as two sets.

$$\text{JACCARD}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In our case A and B refer to the tokens list in documents and query respectively.

- **Preprocessing** of all the documents is done similar to Question 1. Snippet can be referred below:

Preprocessing Functions

[75]

```
def l_case(data):
    return np.char.lower(data)

def stop_words(data):
    stop_words = stopwords.words('english')
    words = word_tokenize(str(data))
    new_word = ""
    for w in words:
        if w not in stop_words and len(w) > 1:
            new_word = new_word + " " + w
    return np.char.strip(new_word)

def punctuation(data):
    symbols = "!\"#$%&()*+-./:;<=>?@[\]^_`{|}~\n"
    for i in range(len(symbols)):
        data = np.char.replace(data, symbols[i], ' ')
        data = np.char.replace(data, "  ", " ")
    return data

def apostrophe(data):
    return np.char.replace(data, "'", "")
```

- Word Tokenization : Tokenization is done with the help of **`word_tokenize()`** to get the words as tokens in each document which will help to compare with the query later in the code.

```
doc = 0
postings = pd.DataFrame()
for i in dataset:
    file = open(i[0], 'r', encoding='utf-8', errors='ignore')
    text = file.read().strip()
    file.close()
    preprocessed_text = preprocess(text)
    tokens = word_tokenize(str(preprocessed_text))
```

- User input for the query is taken and similar preprocessing is applied on the query to get a list of tokens in the query.

```
flag1=0
query = preprocess(input('Enter Input Query')).tolist().split()
print(query)
```

```
['telephone', 'paved', 'roads']
```

- Then Jaccard Coefficient value is calculated for the input query with each document in the stories dataset. These values are stored in a list of size 467 as there are total 467 number of documents in the dataset.

```
def get_jaccard_values(query):
    jaccard_coefficient = []
    for i in dataset:
        file = open(i[0], 'r', encoding='utf-8', errors='ignore')
        text = file.read().strip()
        file.close()
        preprocessed_text = preprocess(text)
        tokens = word_tokenize(str(preprocessed_text))
        # print(type(tokens))
        intersection_size = len(set(query).intersection(set(tokens)))
        union_size = len(set(query).union(set(tokens)))
        jaccard_coefficient_val = intersection_size/float(union_size)
        print(jaccard_coefficient_val)
        jaccard_coefficient.append(jaccard_coefficient_val)
    print(len(jaccard_coefficient))
    return jaccard_coefficient
```

```
jaccard_coefficient = get_jaccard_values(query)
```

```
0.0
0.0
0.0005737234652897303
0.002544529262086514
0.0
0.0
0.0
0.0
0.0
0.0
0.006134969325153374
```

- These values are saved into a list from which top 5 values are retrieved and their corresponding documents are named in the code along with their Jaccard Coefficients wrt the input query.

```
[0.004878048780487805, 0.004962779156327543, 0.004962779156327543, 0.005181347150259067, 0.006134969325153374]
0.006134969325153374  and index is  28
0.005181347150259067  and index is  211
0.004962779156327543  and index is  386
0.004962779156327543  and index is  387
0.004878048780487805  and index is  235

List of Top Documents wrt Jaccard Coefficient for i/p query is:
aircon.txt  with Jaccard Similarity Value: 0.006134969325153374
 graymare.txt  with Jaccard Similarity Value: 0.005181347150259067
 social.vikings  with Jaccard Similarity Value: 0.004962779156327543
 socialvikings.txt  with Jaccard Similarity Value: 0.004962779156327543
 hotline4.txt  with Jaccard Similarity Value: 0.004878048780487805
```

As can be seen in the above snippet, for the input query “telephone,paved, roads”, **the top 5 documents are aircon.txt, graymare.txt, social.vikings, socialvikings.txt, hotline4.txt** and the respective Jaccard Coefficient values is also mentioned in the output.

TD - IDF MATRIX :

1. The dataset is the same as the first question and the same preprocessing steps are carried out as mentioned before.
2. Then term frequency (TF) is calculated which involves computing the raw count of the word in each document and stored as a nested list for each document.
3. Document frequency (DF) is calculated by iterating through the tokens after the text is preprocessed and the count is stored in a dictionary.
4. The Inverse Document Frequency (IDF) value of each word is calculated using the formula:

   Using smoothing:-

   IDF(word)=log(total no. of documents +1/document frequency(word)+1)

5. The Term Frequency is calculated using 5 different variants as asked in the question:

Weighting Scheme	TF Weight
Binary	0,1
Raw count	f(t,d)
Term Frequency	f(t,d)/∑f(t`,d)
Log Normalization	log(1+f(t,d))
Double Normalization	0.5+0.5*(f(t,d)/ max(f(t`,d))

Binary - It is a simple method to calculate term frequency wherein vocab size vector is taken for each document and all the entries for the terms present in the document are labelled 1 and the rest of the term entries are labelled zero.

Raw Count - This is the method to sum up the number of terms existing in documents.

Term frequency - This is what's called tf, which is a ratio calculated by raw count and number of words in documents.

Log Normalization - In this method, log is taken over the raw count.

Double Normalisation - In this variation of tf, where raw count is divided by the frequency of the term occurring maximum times in the document. Also a constant is added.

6. Then after calculating tf and idf, tf-idf matrix is built of size #documents X vocab size and tf -idf values are filled for each word of the vocab.
7. Query vector is made into the size of vocab.
8. Then, TF-IDF score is computed for the query using the TF-IDF matrix. And the top 5 relevant documents based on the score are reported.
9. This is done for all the five tf weighting schemes.

   Below are the snapshots of the process and results obtained.

   Binary tf-

```
# Binary tf-idf
import pandas as pd

TF_idf0 =  np.zeros((N, len(total_vocab)))
tf_idf0 = []
Bin_tf = []
bin_tfidf_df = pd.DataFrame()
for p in range(N):

    sent_vec = []
    tf_vec0 =[]

    for token in total_vocab:
        if token in processed_text[p]:
            sent_vec.append(1)
        else:
            sent_vec.append(0)


    for j in range(len(total_vocab)):

        df0 = doc_freq(total_vocab[j])
        idf0 = np.log((N+1)/(df0+1))

        temp = sent_vec[j]*idf0
        tf_vec0.append(temp)

        TF_idf0[p][j]= sent_vec[j]*idf0
```

Calculation of tf-idf score for query -

```
for j in range(N):
    tfscore = 0
    for k in range(len(total_vocab)):
        if(q_vec[k] == 1):
            tfscore += TF_idf0[j][k]
        else:
            tfscore = tfscore
    alltfscore.append(tfscore)
```

Result for binary tf-

```
Matching Score

Query: THE CRAB AND THE HERON

['crab', 'heron']

[122, 26, 418, 270, 169]
9.406564833939129
9.406564833939129
4.356708826689592
4.356708826689592
4.356708826689592
['crabhern.txt', 'aesop11.txt', 'timem.hac', 'long1-3.txt', 'fgoose.txt']
```

Raw Count-

```
tokens = processed_text[i]

counter = Counter(tokens + processed_title[i])
words_count = len(tokens + processed_title[i])



for token in total_vocab:
    if token in processed_text[i]:

        tf = counter[token]
```

Result for raw count -

```
Matching Score

Query: THE CRAB AND THE HERON

['crab', 'heron']

[122, 26, 418, 270, 169]
83.96593632489221
31.19010896738709
13.070126480068776
8.713417653379183
4.356708826689592
['crabhern.txt', 'aesop11.txt', 'timem.hac', 'long1-3.txt', 'fgoose.txt']
```

Term Frequency -

```
for i in range(N):
    tf_vec = []
    tf_vec1 =[]
    tokens = processed_text[i]

    counter = Counter(tokens + processed_title[i])
    words_count = len(tokens + processed_title[i])


    for token in total_vocab:
        if token in processed_text[i]:

            tf = counter[token]/words_count
            tf_vec.append(tf)
```

Result for Tf-

```
Matching Score

Query: THE CRAB AND THE HERON

['crab', 'heron']

[122, 26, 270, 418, 169]
0.41362530209306514
0.0016366746585185015
0.001309303929884175
0.0008390118423461789
0.000632598929387192
['crabhern.txt', 'aesop11.txt', 'long1-3.txt', 'timem.hac', 'fgoose.txt']
```

Log Normalisation-

```
for token in total_vocab:
    if token in processed_text[i]:
        tf = np.log(1 + counter[token])
        tf_vec.append(tf)
```

Result for Log Normalisation -

```
Matching Score

Query: THE CRAB AND THE HERON

['crab', 'heron']

[122, 26, 418, 270, 169]
21.54259923161554
11.978057375992861
6.039680879481035
4.786333855150008
3.0198404397405176
['crabhern.txt', 'aesop11.txt', 'timem.hac', 'long1-3.txt', 'fgoose.txt']
```

Double Normalisation -

```
counter = Counter(tokens + processed_title[i])
# words_count = len(tokens + processed_title[i])

count_max_freq = counter.most_common(1)[0][1]
# print(count_max_freq)

for token in total_vocab:
    if token in processed_text[i]:
        tf = 0.5+0.5*(counter[token]/count_max_freq)
        # 0.5+0.5*(f(t,d)/ max(f(t`,d))

        tf_vec.append(tf)
```

Result for Double Normalisation -

```
Matching Score

Query: THE CRAB AND THE HERON

['crab', 'heron']

[122, 26, 270, 418, 169]
8.901579233214175
4.776156503341964
2.2296098113058496
2.210867165782778
2.2099247671613873
['crabhern.txt', 'aesop11.txt', 'long1-3.txt', 'timem.hac', 'fgoose.txt']
```

COSINE SIMILARITY : Cosine similarity is another measure to get the similarity between two vectors of an inner product space. It is measured by the cosine of the angle between two vectors and determines whether two vectors are pointing in roughly the same direction or not. It is applied to get document similarity matrices as well in text analysis mainly for information retrieval purposes. All documents are represented as term-frequency vectors as in above step using five different Weighting Schemes. Cosine similarity measures will give us a ranking of the documents in stories dataset with respect to an input query. It is calculated using:

$$similarity(A,B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

We have used the **cosine_similarity function from sklearn library** and applied it to the preprocessed query and the documents.

```
# Document Vectorization
cosine_score_term_freq = []
from sklearn.metrics.pairwise import cosine_similarity
D = np.zeros((N, total_vocab_size))
q_vec_1 = np.asarray(q_vec, dtype=np.float32)
for i in TF_idf:
    doc = i.reshape(1,total_vocab_size)
    doc = np.array(doc)
    # print(type(doc))
    query = q_vec_1.reshape(1,total_vocab_size)
    query = np.array(query)
    cosSim = cosine_similarity(doc, query)
    # list1 = cosSim[0].tolist()
    list1 = cosSim[0].item()
    cosine_score_term_freq.append(list1)
print(cosine_score_term_freq)
```

We have done this for all the applied Weighting Schemes for the tf-idf and obtained results are logged in the code.

For Binary Weighting Scheme:

```
List of Top Documents wrt Cosine Similarity for i/p query(Binary TF Weighting) is:
crabhern.txt  with Cosine Similarity Value: 0.2401234519815826
 aesop11.txt  with Cosine Similarity Value: 0.025439293354229037
 long1-3.txt  with Cosine Similarity Value: 0.0211669121083841
 fgoose.txt  with Cosine Similarity Value: 0.01770831819721052
 timem.hac  with Cosine Similarity Value: 0.013841932742447044
```

For Raw Count Weighting Scheme:

```
List of Top Documents wrt Cosine Similarity for i/p query(Raw count) is:
crabhern.txt  with Cosine Similarity Value: 0.7973139054443423
 aesop11.txt  with Cosine Similarity Value: 0.018439642087588422
 timem.hac  with Cosine Similarity Value: 0.011905221345725029
 long1-3.txt  with Cosine Similarity Value: 0.010776157800016615
 fgoose.txt  with Cosine Similarity Value: 0.006676890533805383
```

For Term Frequency Scheme:

```
List of Top Documents wrt Cosine Similarity for i/p query is:
crabhern.txt  with Cosine Similarity Value: 0.7973139054443426
 aesop11.txt  with Cosine Similarity Value: 0.0184396420875884 56
 timem.hac  with Cosine Similarity Value: 0.011905221345725024
 long1-3.txt  with Cosine Similarity Value: 0.010776157800016617
 fgoose.txt  with Cosine Similarity Value: 0.006676890533805379
```

For Log Normalization Scheme:

```
 List of Top Documents wrt Cosine Similarity for i/p query(Log Normalization) is:
 crabhern.txt  with Cosine Similarity Value: 0.5607367318281735
  aesop11.txt  with Cosine Similarity Value: 0.03016404874411095
  long1-3.txt  with Cosine Similarity Value: 0.023900241380333767
  timem.hac  with Cosine Similarity Value: 0.019228825736402343
  fgoose.txt  with Cosine Similarity Value: 0.012887410111712457
```

For Double Normalization Scheme:

```
List of Top Documents wrt Cosine Similarity for i/p query(Double Normalization) is:
crabhern.txt  with Cosine Similarity Value: 0.3808980579800617
 aesop11.txt  with Cosine Similarity Value: 0.02557373919250654
 long1-3.txt  with Cosine Similarity Value: 0.02119659412756276
 fgoose.txt  with Cosine Similarity Value: 0.01753623169044474
 timem.hac  with Cosine Similarity Value: 0.013919207191540366
```

Pros and cons of each approach-

Jaccard Similarity

Pros

- Used in applications where binary or binarized data are used
- Used to compare set of patterns
- It is good for cases where duplication does not matter

Cons

- Jaccard index is highly influenced by the size of the data. Large datasets can have a big impact on the index as it could significantly increase the union whilst keeping the intersection similar.

TF-IDF Scheme

Pros

- Easy to compute
- Easy to compute similarity between two documents
- We can extract most descriptive terms using this scheme
- TF is good for text similarity in general, but TF-IDF is good for search query relevance.

Cons

- Term ordering is not considered
- Cannot capture position in text or semantics.

Cosine Similarity

Pros

- It is low-complexity, especially for sparse vectors
- The cosine similarity is beneficial because even if the two similar documents are far apart by the Euclidean distance because of the size, they could still have a smaller angle between them
- cosine similarity is good for cases where duplication matters while analyzing text similarity

Cons

- Magnitude is not taken into account, only direction. Thus, this means differences in values are not taken into account.

Question 3- Ranked-Information Retrieval and Evaluation

1. **Consider only the queries with qid:4 and the relevance judgement labels as relevance score:**

   Steps:

   a) The datafile provided with the question is opened in the read mode.

   b) All the lines in the file are read using **readlines()** function.

   c) From these lines, all the lines with **qid:4** are separated and stored into a list.

2. **Make a file rearranging the query-url pairs in order of max DCG. State how many such files could be made.**

   Steps:

   a) To arrange query-url pairs, a function **create_maxDCG_file()** is created.

   b) All lines with **qid:4** are provided as an input to this function.

   c) For max DCG, all pairs must be sorted in **descending order** of their relevance score. This is done using the **sort()** function.

   d) Another file named "**max_DCG.txt**" is opened and all these sorted query-url pairs are written into it.

   e) To calculate how many files could be made, distinct relevance scores and their counts are calculated.

   f) In the given file, there are 59 zero's, 26 one's, 17 two's and 1 three as relevance scores for **qid:4**. So the total number of files that can be made are:
   1! * 17! * 26! * 59!=
   19893497375938370599826047614905329896936840170566570588205180312704857992695193482412686565431050240000000000000000000000000

3. **Compute nDCG:**

    a. At 50

    b. For the whole dataset

Methodology:

Formula to calculate nDCG:

$$\mathrm{nDCG_p} = \frac{DCG_p}{IDCG_p},$$

There are two formulas to calculate DCG:

- **First Formula:**

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

- **Second Formula:**

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

Formula for iDCG:

$$\mathrm{IDCG_p} = \sum_{i=1}^{|REL_p|} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

where $\mathbf{REL_p}$ represents the list of relevant documents (ordered by their relevance) in the corpus **up to position p.**

Steps:

a) To calculate nDCG value, a function **nDCG_calc()** is created.

b) This function takes a list of query-url pairs and a k value (till which the nDCG should be calculated).

c) In this function, DCG and iDCG values are calculated using both the formulas above and the calculated values are returned and printed.

For qid:4 query-url pairs in the given file, following values are obtained:

```
For value =  50 :

Using Formula 1:
DCG =  7.390580969258021
iDCG =  20.989750804831445
nDCG =  0.3521042740324887

Using Formula 2:
DCG =  10.323516383590077
iDCG =  28.98846753873482
nDCG =  0.35612494416255847

For Entire Dataset:

Using Formula 1:
DCG =  12.550247459532576
iDCG =  20.989750804831445
nDCG =  0.5979226516897831

Using Formula 2:
DCG =  16.768935581665193
iDCG =  28.98846753873482
nDCG =  0.5784691984582591
```

4. 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. the higher the value, the more relevant the URL. Assume any non zero relevance judgment value to be relevant. Plot a Precision-Recall curve for query "qid:4".

Steps:

a) To calculate precision and recall values, **calc_precision_recall()** function is created which after calculating these values calls the **plot_precision_recall()** function which plots these values.

b) In the calc_precision_recall() function, values of **feature 75** are stored in a dictionary with key being the entire query-url pair.

c) This dictionary is sorted based on the key values.

d) Total number of relevant documents are calculated by finding the relevance score from the dictionary keys (first index of each key will be the relevance score).

e) A counter named **curr_relevant** keeps track of the total relevant documents seen till now. Precision is calculated by dividing this curr_relevant by the number of documents seen till now.

f) Recall is calculated by dividing the curr_relevant by the total number of relevant documents calculated earlier.

g) This precision and recall values are then plotted using plot function from **matplotlib library** and the plot is shown below.