IR Assignment-2

Vishnumolakala Nikhila (MT21103)
Madadi Vineeth Reddy (MT21046)

Question 1 - Scoring and Term-Weighting

- Jaccard Coefficient

➢ Firstly, import all the necessary libraries such as nltk, pandas, numpy, etc…
➢ Import tokenizer, stop words, stemmer and lemmatizer from the nltk library.

```
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

from nltk.stem import PorterStemmer
ps = PorterStemmer()

from nltk.stem import WordNetLemmatizer
lem = WordNetLemmatizer()
```

Pre-processing steps:

```
def pre_process(s):
    s = s.lower()
    l=len(string.punctuation)
    s = s.translate(s.maketrans(string.punctuation,' '*l,''))
    #s = re.sub('[^A-Za-z\s\n ]+', ' ',s)

    t = word_tokenize(s)
    t = [lem.lemmatize(w) for w in t if w not in stopwords.words('english') and w.isalpha() and len(w)>1]
    return t
```

➢ Converted all the words to lowercase.
➢ Removed the punctuations and replaced it with space and also removed extra white spaces.
➢ Stop words are being removed.
➢ Word is tokenized with the help of tokenizer obtained from nltk library.

- ➢ Lemmatization is done with the help of lemmatizer obtained from the nltk library.
- ➢ Here we are mapping the document names with document numbers.

Calculating Jaccard Coefficient:

```
def jaccard(t1,t2):
    s1=set(t1)
    s2=set(t2)
    u=s1.union(s2)
    i=s1.intersection(s2)
    j=len(i)/len(u)
    return j
```

- ➢ In the above code, the function takes as it's parameters a document and input query.
- ➢ It then applies the Jaccard coefficient formula which is intersection of those by it's union.
- ➢ Then we finally return the answer.

```
def find_js(qt):
    js=[]
    for i in d:
        js.append([jaccard(d[i],qt),i])
    js.sort(key=lambda x:x[0])
    return js
```

- ➢ The find_js function takes as its parameters the query and then gets each document and sends both the document and query as parameters while calling the jaccard function.
- ➢ It finally gets the result from the jaccard function and then sorts them according to Jaccard score.

Sample Execution:

```
print("Enter Input Query:")
q=input()
qt=pre_process(q)
print(qt)
js=find_js(qt)
for i in range(5):
    print(doc[js[i][1]])
```

```
Enter Input Query:
once upon a time.
['upon', 'time']
acetab1.txt
aclamt.txt
adameve.hum
addrmeri.txt
alcatax.txt
```

- ➢ Initially, we prompt the user to give input query and then pre-process the given input query.
- ➢ Then we print the pre-processed query.
- ➢ Then we pass this query as a parameter and call the find_js function which inturn calls the jaccard function to finally get the result.
- ➢ We then print the top five documents matching the input query based on the Jaccard coefficient calculated.

- TF-IDF Matrix

Finding all the unique words:

```
uw=[]
for l in d:
    l1=list(set(d[l]))
    uw.extend(l1)
    uw=list(set(uw))
print(len(uw))
```

Calculating the document frequency:

```
def doc_freq():
    df={}
    for i in d:
        tl=list(set(d[i]))
        for t in tl:
            if t in df:
                df[t]+=1
            else:
                df[t]=1
    return df
```

- ➢ The function doc_freq computes the document frequency in the loop and finally returns the document frequency.

Calculating the inverse document frequency:

```
import math
def inv_df():
    idf={}
    total_doc=len(d)
    for t in df:
        idf[t]=math.log(total_doc/df[t]+1)
    return idf
```

- ➢ The function inv_df is used to calculate the inverse document frequency.
- ➢ It applies the formula of inverse document frequency which is the log of total number of documents upon documents that contain the word.
- ➢ Finally, the function returns the inverse document frequency value.

Calculating the frequency:

```
def freq():
    f={}
    for i in d:
        f[i]={}
        for t in d[i]:
            if t in f[i]:
                f[i][t]+=1
            else:
                f[i][t]=1
    return f
```

Binary Scheme:

```
def bin_tf():
    btf={}
    for di in f:
        btf[di]={}
        for w in f[di]:
            #if w in f[di]:
            btf[di][w]=1
            #else:
                #btf[di][w]=0
    return btf
```

- ➢ One of the scheme for TF-IDF is the binary scheme which is implemented in the above bin_tf function.
- ➢ It gives 1 if present and 0 if not present and nothing between 0 and 1.
- ➢ This function finally returns the term frequency using the binary scheme method.

Raw count Scheme:

```
def rawcnt_tf():
    rctf={}
    for di in f:
        rctf[di]={}
        for w in f[di]:
            rctf[di][w]=f[di][w]
        """for w in uw:
            if w in f[di]:
                rctf[di][w]=f[di][w]
            else:
                rctf[di][w]=0"""
    return rctf
```

- ➢ This is another scheme for calculating the term frequency which just gives the raw count of how many times it is repeated.
- ➢ Finally, returns the term frequency value.

Term frequency Scheme:

```
def term_freq():
    tf={}
    for di in f:
        tf[di]={}
        s=sum(f[di].values())
        for w in f[di]:
            tf[di][w]=f[di][w]/s
        """for w in uw:
            if w in f[di]:
                tf[di][w]=f[di][w]/s
            else:
                tf[di][w]=0"""
    return tf
```

- ➢ The other variant of term frequency is the term frequency scheme in which we compute the frequency by taking raw count upon summation of raw counts.
- ➢ Finally, we return the value of term frequency using this scheme.

Log normalization:

```
def log_tf():
    ltf={}
    for di in f:
        ltf[di]={}
        for w in f[di]:
            ltf[di][w]=math.log(1+f[di][w])
        """for w in uw:
            if w in f[di]:
                ltf[di][w]=math.log(1+f[di][w])
            else:
                ltf[di][w]=0"""
    return ltf
```

- In the above function log_tf we calculate the term frequency by applying the formula log(1+f(t,d)).
- We return the value of term frequency at the end.

Double normalization:

```
def dn_tf():
    dntf={}
    for di in f:
        dntf[di]={}
        m=max(f[di].values())
        for w in f[di]:
            dntf[di][w]=0.5+0.5*(f[di][w]/m)
        """for w in uw:
            if w in f[di]:
                dntf[di][w]=0.5+0.5*(f[di][w]/m)
            else:
                dntf[di][w]=0"""
    return dntf
```

- In the above function dn_tf we calculate the term frequency by applying the formula 0.5+0.5*(f(t,d)/ max(f(t‘,d))
- We return the value of term frequency at the end.

Sample Execution:

```
print("Enter Input Query:")
q=input()
qt=pre_process(q)
print(qt)

#binary
print("\nbinary ")
bd=query(qt,btfidf)
print(*bd)

#raw count
print("\nraw count ")
rcd=query(qt,rctfidf)
print(*rcd)

#term frequency
print("\nterm frequency ")
tfd=query(qt,tfidf)
print(*tfd)

#log norm
print("\nlog norm ")
ld=query(qt,ltfidf)
print(*ld)

#double norm
print("\ndouble norm ")
dnd=query(qt,dntfidf)
print(*dnd)
```

```
Enter Input Query:
once upon a time
['upon', 'time']

binary
adt_miam.txt allusion all_grai amazing.epi ambrose.bie ayurved.txt

raw count
mlverb.hum practica.txt barney.txt humor9.txt manners.txt xibovac.txt

term frequency
timetr.hum ookpik.hum sysman.txt yuppies.hum trukdeth.txt corporat.txt

log norm
barney.txt mindvox practica.txt quack26.txt humor9.txt jokes1.txt

double norm
ookpik.hum jokes1.txt trukdeth.txt ambrose.bie mindvox flux_fix.txt
```

- ➢ Initially, we prompt the user to input a query and then we pre-process that input query.
- ➢ Then all the five schemes are being called one after the other and based on values obtained, the documents are retrieved for each scheme separately.

Advantages (Pros):

- ➢ Binary scheme: It is the simplest scheme of all as it only says about presence and absence of words.
- ➢ Raw count scheme: This scheme solves the issues with binary scheme since it takes frequency of words, it is more relevant.
- ➢ Term frequency scheme: The above scheme could be biased, this issue is solved by the term frequency scheme since it normalizes frequency of words using length of the documents.
- ➢ Log normalization scheme: This scheme reduces the required computational resources since it normalizes raw count and takes log of it and thus very less number is obtained.
- ➢ Double normalization: The schema does normalization by length of the document and also log normalization. This also reduces the usage of computational resources.

Disadvantages (Cons):

- ➢ Binary scheme: As it does not include frequency of the term, it is very unreliable as term frequency is one of the most important components.
- ➢ Raw count scheme: This prefers large documents over smaller ones which is not a good criterion in many cases.
- ➢ Term frequency: Generally a lot of storage space is needed because it gives huge numbers.
- ➢ Log normalization: As the log normalization is similar to raw count, it prefers large documents over small ones which is not always correct because large always doesn't mean more relevant.
- ➢ Double normalization: In the denominator of double normalization, we have a frequency of term that is maximum in the document. It is a disadvantage because there can occur a case in which that term is not relevant at all which gives wrong interpretation.

Question 2 - Ranked-Information Retrieval and Evaluation

- ➢ Firstly, import all the necessary libraries such as pandas, numpy, matplotlib and math.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math
```

(1)Queries with qid:4

```
qdf=df[df[1]=="qid:4"]
#qdf=qdf.reset_index(drop=True)
qdf
```

(2)Rearranging the query-url pairs in order of max DCG & number of files:

```
# Q) Make a file rearranging the query-url pairs in order of max DCG
np.savetxt(r'maxDCG.txt', mdf.values, fmt='%s')
```

```
# Q) State how many such files could be made.
l=list(mdf[0].unique())
cnt=1
for i in l:
    c=len(mdf[mdf[0]==i])
    print(f"{i} = {c}")
    cnt=cnt*(math.factorial(c))
print(f"no. of files with max DCG {cnt}")
```

```
3 = 1
2 = 17
1 = 26
0 = 59
no. of files with max DCG 19893497375938370599826047614905329896936840170566570588205180312704857992695193482412686565431050240
000000000000000000000
```

- ➢ In the above code, first the query-url pairs are being re-arranged in the order of maximum dcg value.
- ➢ Then the number of such files are computed whose value turned out to be 19893497375938370599826047614905329896936840170566570588205180312704857992695193482412686565431050240000000000000000000000000.

(3)nDCG value at 50 and whole dataset:

```
def Compute_nDCG(n,d):
    d = np.asfarray(d)[:n]
    dcg = d[0] + np.sum(d[1:] / np.log2(np.arange(2, d.size + 1)))

    """dcg=0
    for i in range(1,n+1):
        dcg += (math.pow(2,(d[i-1]-1))/math.log2(i+1))"""
    return dcg
```

- ➢ The Compute_nDCG method is used to compute the dcg value based on parameters passed to it.

At 50:

```
dcg50=Compute_nDCG(50,qdf[0])
idcg50=Compute_nDCG(50,mdf[0])
ndcg50=dcg50/idcg50
print(ndcg50)
```

```
0.35210427403248856
```

➢ From the above, it can be noted that the ndcg value is 0.35210427403248856.

For the whole dataset:

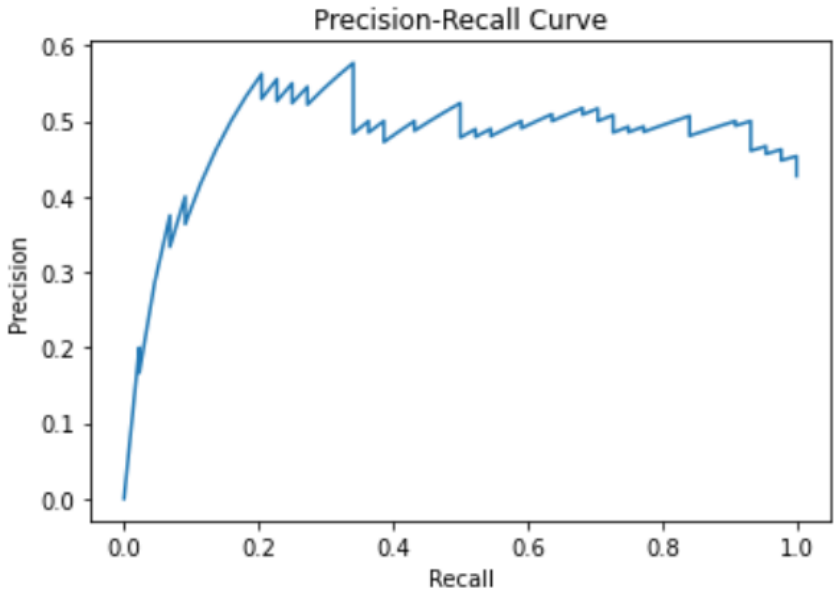
```
# (b) For the whole dataset
dcgwhole=Compute_nDCG(len(qdf),qdf[0])
idcgwhole=Compute_nDCG(len(mdf),mdf[0])
ndcgwhole=dcgwhole/idcgwhole
print(ndcgwhole)
```

```
0.5979226516897828
```

➢ From the above, it can be noted that the ndcg value is 0.5979226516897828.

(4)Precision-Recall curve for query “qid:4”:

```
plt.title('Precision-Recall Curve')
plt.plot(rl,pl)
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.show()
```



➢ From the above, we can see that precision, recall graph is plotted.

Question 3 - Naive Bayes Classifier with TF-ICF

- ➢ Importing necessary libraries such as numpy, pandas, nltk, etc…
- ➢ Import tokenizer, stop words, stemmer and lemmatizer from the nltk library.

```
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords

from nltk.stem import PorterStemmer
ps = PorterStemmer()

from nltk.stem import WordNetLemmatizer
lem = WordNetLemmatizer()
```

Pre-processing steps:

- ➢ Converted all the words to lowercase.
- ➢ Removed the punctuations and replaced it with space and also removed extra white spaces.
- ➢ Stop words are being removed.
- ➢ Word is tokenized with the help of tokenizer obtained from nltk library.
- ➢ Lemmatization is done with the help of lemmatizer obtained from the nltk library.
- ➢ Here we are mapping the document names with document numbers.

```
def pre_process(s):
    s = s.lower()
    l=len(string.punctuation)
    s = s.translate(s.maketrans(string.punctuation,' '*l,''))
    #s = re.sub('[^A-Za-z\s\n ]+', ' ',s)

    t = word_tokenize(s)
    t = [lem.lemmatize(w) for w in t if w not in stopwords.words('english') and w.isalpha() and len(w)>1]
    return t
```

Initializing:

```
cls={0:"comp.graphics",1:"sci.med",2:"talk.politics.misc",3:"rec.sport.hockey",4:"sci.space"}
labels=['comp.graphics', 'rec.sport.hockey', 'sci.med', 'sci.space', 'talk.politics.misc']
docs={}
data ={'text':[],'label':[]}
```

```
x=[]
y=[]
yc=[]
for i in range(5):
    c=cls[i]
    files = os.listdir("data_a2/"+c)
    print(c,len(files))
    for j in range(len(files)):
        file="data_A2/"+c+"/"+files[j]
        with open(file, 'r', encoding="utf8", errors="ignore") as f:
            text = f.read()
            tl=pre_process(text)
            x.append(tl)
            y.append(i)
            yc.append(c)
    print(len(x))
```

```
def findTFICF(d,wl,cw):
    tficf={}
    c=set(Counter(wl))
    for i in c:
        tf=c[i]
        icf=np.log(len(d)/cw+1)
        tficf[i]=tf*icf
    return tficf
```

```
def accuracy(yt,yp):
    c=0
    for i in range(len(yp)):
        if(yp[i]==yt[i]):
            c+=1
    return c/len(yp)
```

Confusion matrix:

```
def confusion_matrix(yt,yp):
    cm= np.zeros((len(cls), len(cls))).astype(int)
    for i in range(len(yp)):
        cm[cls.index(yp[i])][cls.index(yt[i])]+= 1
    return cm
```

Naive Bayes:

```
class NB():
    def fit(self,x_train,y_train,k):
        w=x_train
        cls=y_train
        n=len(w)
        d={}
        #dictonary of words per class
        for i in range(n):
            if cls[i] in d:
                d[cls[i]]=d[cls[i]]+w[i]
            else:
                d[cls[i]]=w[i]
        #list of words
        wl=[]
        for i in wl:
            wl=wl+w[i]
        #count of word per class
        cw={}
        for i in d:
            l=d[i]
            for j in set(l):
                if j not in cw:
                    cw[j]=1
                else:
                    cw[j]+=1
        #calculating TF-ICF
        tficf=findTFICF(d,wl,cw)
        x=sorted(tficf.items(), key=operator.itemgetter(1), reverse=True)
        x=x[:int(len(x)*k/100)]# considering top k features
        uw=[i[0] for i in x]

        #frquency of each word per class
        f={}
        #total number of words in that class
        nw={}
        for i in cls:
```

```
            fl=Counter(d[i])
            for j in uw:
                f[i,j]=fl[j]
                if i not in nw:
                    nw[i]=fl[i]
                else:
                    nw[i]+=fl[j]

        #calculating the frequency of each class
        train={}
        for i in y_train:
            if i not in train:
                train[i]=1
            else:
                train[i]+=1
        self.f=f
        self.uw=uw
        self.nw=nw

    #calculating the probability of each word in test set
    def predict(self,x_test):
        yp=[]
        for i in range(len(x_test)):
            cwp=[]
            for c in cls:
                wp=0
                for w in x_test[i]:
                    try:
                        f=self.f[c,w]
                    except:
                        f=0
                    c1=self.nw[c]
                    p=(f+1)+(c1+len(self.uw))
                    wp+=np.log(p)
                cwp.append(wp)
            yp.append(cls[np.argmax(cwp)])
        return yp
```

Splitting into 50:50

```
#splitting the data of ratio 50:50
train=data.sample(frac=0.5,random_state=41)
test=data.sample(frac=1,random_state=41).drop(train.index)
x_train=train['text'].tolist()
y_train=train['label'].tolist()
x_test=test['text'].tolist()
y_test=test['label'].tolist()
k=500
clf=NB()
clf.fit(x_train,y_train,k)
yp=clf.predict(x_test)
acc=accuracy(y_test,yp)
print(f"Accuracy at 50:50 split = {acc}")
print(confusion_matrix(y_test,yp))
```

➢ In the above code, an 50:50 split is done and after applying the method, accuracy is calculated along with a confusion matrix.

```
At ratio 50:50


Confusion Matrix is given by:
 [[491   4  10  11   1]
 [  1 478   2   1   0]
 [  0   0 465   4   0]
 [  2   0   8 503   4]
 [  0   2   7   3 503]]


Accuracy is 97.60
```

Splitting into 70:30

```python
#splitting the data of ratio 70:30
train=data.sample(frac=0.7,random_state=41)
test=data.sample(frac=1,random_state=41).drop(train.index)
x_train=train['text'].tolist()
y_train=train['label'].tolist()
x_test=test['text'].tolist()
y_test=test['label'].tolist()
k=500
clf=NB()
clf.fit(x_train,y_train,k)
yp=clf.predict(x_test)
acc=accuracy(y_test,yp)
print(f"Accuracy at 70:30 split = {acc}")
print(confusion_matrix(y_test,yp))
```

➢ In the above code, an 70:30 split is done and after applying the method, accuracy is calculated along with a confusion matrix.

```
At ratio 70:30


Confusion Matrix is given by:
 [[299   1   4   5   0]
 [  0 277   0   1   0]
 [  2   0 291   3   0]
 [  1   0   3 302   4]
 [  0   0   3   0 304]]
```

Splitting into 80:20

```
#splitting the data of ratio 80:20
train=data.sample(frac=0.8,random_state=41)
test=data.sample(frac=1,random_state=41).drop(train.index)
x_train=train['text'].tolist()
y_train=train['label'].tolist()
x_test=test['text'].tolist()
y_test=test['label'].tolist()
k=500
clf=NB()
clf.fit(x_train,y_train,k)
yp=clf.predict(x_test)
acc=accuracy(y_test,yp)
print(f"Accuracy at 80:20 split = {acc}")
print(confusion_matrix(y_test,yp))
```

- ➢ In the above code, an 80:20 split is done and after applying the method, accuracy is calculated along with a confusion matrix.

The data is not splitted in not taken in sequential order. This is shuffled using sample method on the dataframe.

```
Confusion Matrix is given by:
 [[189   1   4   2   0]
 [  0 190   0   0   0]
 [  0   0 204   2   0]
 [  0   0   2 200   3]
 [  0   0   3   0 200]]


Accuracy is 98.30
```

Analysis :

Here we observed that with the increase in training data, the accuracy (performance of the model) got increased.