INFORMATION RETRIEVAL ASSIGNMENT-2

Group No. 78

Nellore Bhavana (MT21131)
Prachi Gupta (MT21135)

Importing necessary libraries

In [1]:
```
#Installing necessary libraries
import os
import glob
import nltk
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.stem.porter import PorterStemmer
import re
import string
import json
import pandas as pd
import numpy as np
import math
from textblob import TextBlob, Word
import joblib
import matplotlib.pyplot as plt
from nltk.tokenize import word_tokenize
from pandas.core.dtypes.cast import dict_compat

nltk.download('punkt')
nltk.download('wordnet')
nltk.download('stopwords')

lema = WordNetLemmatizer()
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\bhava\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\bhava\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

QUESTION 1

Methodology:

Read files

- To read files present in “`Humor,Hist,Media,Food`” folder
  `path = "/content/drive/MyDrive/Datasets/Humor,Hist,Media,Food"`
- To open a particular file
  `rf = open(path,'r',errors = 'ignore')`
- To read content of file
  `rf = rf.read()`

a) Preprocessing steps for the given data :

(i) Converting the text to lower case
(ii) Word Tokenization
(iii) Removing the stop words from tokens
(iv) Removing the punctuation marks from tokens
(v) Removing the blank spaces tokens

```
[ ]  ln = len(string.punctuation)
     #Cleaning the data
     def pre_process(content):
       #Convert the text to lower case
       content = content.lower()
       #Remove punctuation marks from tokens
       content = content.translate(str.maketrans(string.punctuation, " "*ln,''))
       #Perform word tokenization
       ctokens = word_tokenize(content)
       #Remove stopwords from tokens and do lemmatization
       #Checking length, if length = 1
       ctokens = [lema.lemmatize(s) for s in ctokens if s not in stopwords.words('english') and s.isalpha and len(s)>1]
       return ctokens
```

b)Performing intersection and union between document and query

```
In [7]: function to make set of the document token and query token and perform intersection and union between the query and each document
        ef jaccard_coeff(tdoc,tquery):
           set1 = set(tdoc)
           set2 = set(tquery)
           un = set1.union(set2)
           it = set1.intersection(set2)
           #Jaccard Coefficient = Intersection of (doc,query) / Union of (doc,query)
           jc = len(it)/len(un)
           return jc
```

c) Implementation of Jaccard co-efficient and reporting top 5 relevant documents

```
In [8]: #Function to Report the top 5 relevant documents based on the value of the Jaccard coefficient
        def top_doc(pquery):
            js = []
            for i in dc:
                js.append([jaccard_coeff(dc[i],pquery),i])
             #Sorting based on the coefficient value
            js.sort(key=lambda x:x[0])
            return js
```

<u>Output :</u>

```
In [9]: #Take input query from user
        print("Enter Query:")
        query = input()
        #Pre process the input query
        pquery = pre_process(query)
        print(pquery)
        js = top_doc(pquery)
        #reporting top 5
        for i in range(5):
            print(document[js[i][1]])

        Enter Query:
        100 west by 50 north
        ['100', 'west', '50', 'north']
        1st_aid.txt
        abbott.txt
        acetab1.txt
        aclamt.txt
        acronym.lis
```

<u>To calculate Jaccard coefficient:</u>

- Create tokens for queries and documents after pre-processing
- Converting them to sets
- Jaccard coefficient = Intersection of (doc, query)/Union of (doc, query)
- Sort values in descending order
- Reporting files having top 5 jaccard coefficients

Ranked-Information Retrieval and Evaluation

]:
```
from pandas.core.dtypes.cast import dict_compat
#Finding frequency
def freq():
    f = {}
    for i in dc:
        f[i] = {}
        for t in dc[i]:
            if t in f[i]:
                f[i][t] += 1
            else:
                f[i][t] = 1
    return f
```

]:
```
f = freq()
print(len(f))
```

To find the unique words from the documents

```
unq = []
#finding unique words present
for lt in dc:
    list1 = list(set(dc[lt]))
    unq.extend(list1)
    unq = list(set(unq))
print(len(unq))
```

```
64206
```

To find the document frequency

```
#Computing the document frequency
def doc_freq():
    doc_f = {}
    for i in dc:
        tl = list(set(dc[i]))
        for t in tl:
            if t in doc_f:
                doc_f[t] = doc_f[t] + 1
            else:
                doc_f[t] = 1
    return doc_f
```

```
doc_f = doc_freq()
print(len(doc_f))
```

```
64206
```

Finding the inverse document frequency

```python
#Function to compute inverse document frequency
def inverse_df():
    idf = {}
    total_doc = len(dc)
    for t in doc_f:
        idf[t] = math.log(total_doc/doc_f[t]+1)
    return idf
```

```python
from pandas.core.dtypes.cast import dict_compat
#Finding frequency
def freq():
    f = {}
    for i in dc:
        f[i] = {}
        for t in dc[i]:
            if t in f[i]:
                f[i][t] += 1
            else:
                f[i][t] = 1
    return f
```

```python
f = freq()
print(len(f))
```

Finding the Frequency and its length

```python
from pandas.core.dtypes.cast import dict_compat
#Finding frequency
def freq():
    f = {}
    for i in dc:
        f[i] = {}
        for t in dc[i]:
            if t in f[i]:
                f[i][t] += 1
            else:
                f[i][t] = 1
    return f
```

```python
f = freq()
print(len(f))
```

```
1133
```

e) Implementation of TF-IDF matrix and finding top 5 relevant documents

```
#
def iquery(qt,tfidf):
    score = {}
    for d1 in dc:
        score[d1] = 0
        for t in qt:
            score[d1]+= tfidf[d1][t]
    dl = []
    dl1 = sorted(range(1,len(score)+1), key=lambda i: score[i], reverse=True)[:6]
    for i in dl1:
        dl.append(document[i])
    return dl
```

f) Implementation of top 5 Weighting Schemas

The five Weighting Schemas are:

- Binary weighting scheme
- Raw count weighting scheme
- Term frequency weighting scheme
- Log normalization weighting scheme
- Double normalization weighting scheme

Binary weighting scheme

```
]: #Function to compute Binary weighting scheme
   def binary_tf():
       btf = {}
       for di in f:
           btf[di]={}
           for w in f[di]:
               btf[di][w] = 1
       return btf
```

```
]: btf = binary_tf()
   print(btf[1])
```

```
{'herbalherb1st': 1, 'aidcalendulacomfreyremediessickmedicine
'ointment': 1, 'use': 1, 'minor': 1, 'cut': 1, 'graz': 1, 're
e': 1, 'damage': 1, 'external': 1, 'blood': 1, 'vessel': 1,
```

Raw count weighting scheme

```
#Function to compute Raw count weighting scheme
def rawcount_tf():
    rctf = {}
    for di in f:
        rctf[di]={}
        for w in f[di]:
            rctf[di][w]=f[di][w]
    return rctf
```

```
rctf = rawcount_tf()
print(len(rctf[1]))
```

148

Term frequency weighting scheme

```
#Function to compute term frequency weighting 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
    return tf
```

```
tf = term_freq()
print(len(tf[1]))
```

148

Log normalization weighting scheme

```
#Function to compute Log normalization weighting scheme
def log_tf():
    ltf={}
    for di in f:
        ltf[di]={}
        for w in f[di]:
            ltf[di][w]=math.log(1+f[di][w])
    return ltf
```

```
ltf = log_tf()
print(len(ltf[1]))
```

148

Double normalization weighting scheme

```
#Function to compute Double normalization weighting scheme
def double_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)
    return dntf
```

```
dntf = double_tf()
print(len(dntf[1]))
```

```
148
```

Output

```
print("Enter Input Query:")
query = input()
pquery = pre_process(query)
print(pquery)

#binary weighting scheme
print("\nbinary ")
bd = iquery(pquery,btfidf)
print(*bd)

#raw count weighting scheme
print("\nraw count ")
rcd = iquery(pquery,rctfidf)
print(*rcd)

#term frequency weighting scheme
print("\nterm frequency ")
tfd = iquery(pquery,tfidf)
print(*tfd)

#log normalization weighting scheme
print("\nlog norm ")
ld = iquery(pquery,ltfidf)
print(*ld)

#double normalization weighting scheme
print("\ndouble norm ")
dnd = iquery(pquery,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 corporat.txt trukdeth.txt yuppies.hum

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
```

Pros and Cons

- Binary:
  Advantage**:** This scheme is the simplest one to compute, as for this only presence and absence of words matters
  Disadvantages: does not consider frequency od word
- Raw Count:
  Advantage: raw count of words in a document is determined, and comparatively more relevant documents are retrieved.
  Disadvantage: Large documents are favored more
- Term Frequency
  Advantage: Reduced the bias caused by the length of documents
  Disadvantage: more storage space
- Log normalization
  Advantage: Reduced computational power
  Disadvantage: This is similar to raw count prefers large documents over small ones which is not always the case in reality.
- Double normalization
  Advantage: Reduced computational power and also considers the term frequency that is normalized using document length
  Disadvantage: In the denominator, we have a frequency of terms that is maximum in the document. There can occur a case in which that term is not relevant which gives wrong results

QUESTION 2

Methodology:

Read files

- To read files present in “`Humor,Hist,Media,Food`” folder
  ```
  path = "/content/drive/MyDrive/Datasets/Humor,Hist,Media,Food"
  ```
- To open a particular file
  ```
  rf = open(path,'r',errors = 'ignore')
  ```
- To read content of file
  ```
  rf = rf.read()
  ```
- To read “`assignment-2.txt`” file
  ```
  data =
  pd.read_csv("/content/drive/MyDrive/IR-assignment-2-data.txt", sep='
  ', header=None)
  ```
- To convert the text file to csv file
  ```
  data.to_csv('Q2.csv', index = None)
  ```

```
data = pd.read_csv("/content/drive/MyDrive/IR-assignment-2-data.txt", sep=' ', header=None)
data.to_csv('Q2.csv', index = None)
```

a) Preprocessing steps for the given data :

(i) Converting the text to lower case
(ii) Word Tokenization
(iii) Removing the stop words from tokens
(iv) Removing the punctuation marks from tokens
(v) Removing the blank spaces tokens

```
[ ]  ln = len(string.punctuation)
     #Cleaning the data
     def pre_process(content):
       #Convert the text to lower case
       content = content.lower()
       #Remove punctuation marks from tokens
       content = content.translate(str.maketrans(string.punctuation, " "*ln,''))
       #Perform word tokenization
       ctokens = word_tokenize(content)
       #Remove stopwords from tokens and do lemmatization
       #Checking length, if length = 1
       ctokens = [lema.lemmatize(s) for s in ctokens if s not in stopwords.words('english') and s.isalpha and len(s)>1]
       return ctokens
```

b) qid:4 queries consideration

```
In [47]: #Select rows with qid:4
         fdata = df[df[1] == 'qid:4']

         total_files = 1
         temp = []
         r = fdata[0].unique()
         for i in r:
             temp.append(len(fdata[fdata[0] == i]))
             total_files = total_files * math.factorial(len(fdata[fdata[0] == i]))

         #Sort the data on the basis of relevance judgement label
         final_data = fdata.sort_values(by = 0, ascending = False )
         final_data = final_data.reset_index(drop=True)
```

```
In [53]: #Select rows with qid:4
         fdata = df[df[1] == 'qid:4']

         total_files = 1
         temp = []
         r = fdata[0].unique()
         for i in r:
             temp.append(len(fdata[fdata[0] == i]))
             total_files = total_files * math.factorial(len(fdata[fdata[0] == i]))

         #Sort the data on the basis of relevance judgement label
         final_data = fdata.sort_values(by = 0, ascending = False )
         final_data = final_data.reset_index(drop=True)
```

c) File arrangement

```
In [48]: #File rearranging the query-url pairs in order of max DCG.
         final_data.to_csv('query-url-pairs.csv')

         #Computing total number of such files
         print(f"Total number of files: {total_files}")

         Total number of files: 1989349737593837059982604761490532989693684017056657058820518031270485799269519348241268656543105024000
         0000000000000000000
```

d) Implementation of DCG

Firstly we will sort the data before the DCG Calculation

```
In [62]: #Sorting data in ascending order
         d1 = fdata.sort_values(by=0, ascending=False)
```

```
In [67]: #Function to compute DCG
         def DCG_Calculation(datadup,n):
             datadup = np.asfarray(datadup)[:n]
             dcg_val = datadup[0] + np.sum(datadup[1:] / np.log2(np.arange(2, datadup.size + 1)))
             return dcg_val
```

e) Max DCG Calculation

```
In [68]: #Computing maximum DCG
         for j in range(1,104):
             l = []
             l.append(DCG_Calculation(d1[0],i))
         print("The Maximium DCG is : ",max(l))

         The Maximium DCG is :  20.989750804831452
```

f) Implementation of nDCG

```
In [69]: #computing nDCG
         def NDCG_Calculation(r, k):
             denominator = DCG_Calculation(d1[0], k)
             numerator = DCG_Calculation(r, k)
             value = numerator/denominator
             return  value
```

g) Implementation of nDCG at 50

```
In [70]: #nDCG value at 50
         nDCG_value_50 = NDCG_Calculation(fdata[0],50)
         print("The value of nDCG at 50 is : ",nDCG_value_50 )

         The value of nDCG at 50 is :  0.35210427403248856
```

h) Implementation of nDCG for whole data set

In [71]:
```
#nDCG value for complete dataset
nDCG_value = NDCG_Calculation(fdata[0],len(fdata))
print("The value of nDCG for all is : ",nDCG_value )
```

```
The value of nDCG for all is :  0.5979226516897828
```

i) Obtaining values from TF-IDF and storing them into a list

In [73]:
```
#To store the values obtained from TF-IDF into a list
p=0
value1 = fdata[76]
value2 = fdata[0]
value1l = value1.tolist()
value2l = value2.tolist()
for dup_list in value1l:
    if dup_list[0] == "7" and dup_list[1] == "5" and dup_list[2] == ":":
        i1 = dup_list[3:]
        value1l[p] = i1
    p = p+1
value1lf = [float(i) for i in value1l]
```

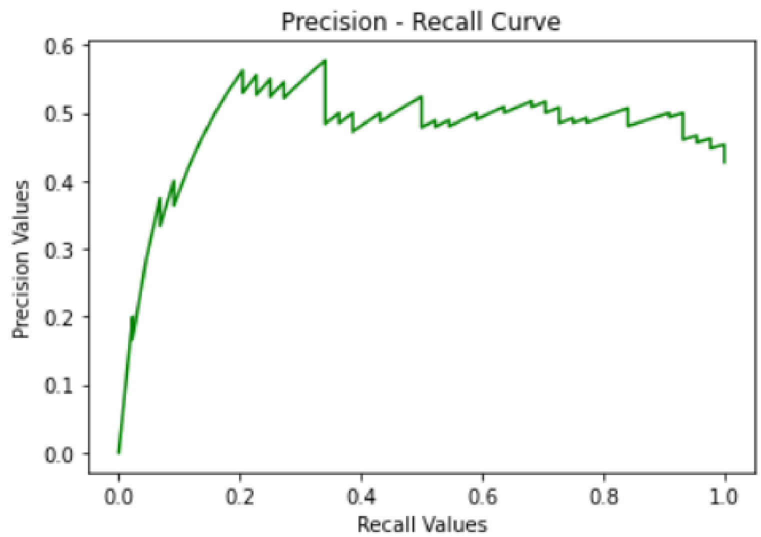
k) Plotting precision and recall curve

In [80]:
```
#Declaring the required variables and the precision and recall lists
total=0
relevant_present=0
relevant_doc=0
list_prevision=[]
list_recall=[]
```

In [79]:
```
#Calculation of precision and recall values
sorted_rel = [m for n,m in sorted(zip(xmlf,xol),reverse=True)]
sorted_rel2 = [(n,m) for n,m in sorted(zip(xmlf,xol),reverse=True)]

for m in sorted_rel:
    if m == 0:
        relevant_doc = relevant_doc
    else:
        relevant_doc = relevant_doc + 1


for m in sorted_rel:
    total= total + 1
    if m == 0:
        relevant_present = relevant_present
    else:
        relevant_present = relevant_present + 1
    list_prevision.append(relevant_present/total)
    list_recall.append(relevant_present/relevant_doc)

#Plotting Precision - Recall Graph
plt.title("Precision - Recall Curve")
plt.plot(list_recall,list_prevision,color="green")
plt.xlabel("Recall Values")
plt.ylabel("Precision Values")
plt.show()
```

Output Graph:

Distinctive Sawtooth shape curve



QUESTION 3

Methodology:

- To read files present in "`Humor,Hist,Media,Food`" folder
  `path = "/content/drive/MyDrive/20_newsgroups.zip"`
- To open a particular file
  `rf = open(path,'r',errors = 'ignore')`
- To read content of file

```
[26] #Load the file
     file = "/content/drive/MyDrive/20_newsgroups.zip"
     #Defining labels
     label = ['comp.graphics', 'rec.sport.hockey', 'sci.med', 'sci.space', 'talk.politics.misc']
     # opening the zip file in read mode
     with ZipFile(file, 'r') as zip:
         zip.printdir()
         zip.extractall()
```

Read files

a) Preprocessing steps for the given data :

(i) Converting the text to lower case

(ii) Word Tokenization

(iii) Removing the stop words from tokens
(iv) Removing the punctuation marks from tokens
(v) Removing the blank spaces tokens

```
[ ] ln = len(string.punctuation)
    #Cleaning the data
    def pre_process(content):
      #Convert the text to lower case
      content = content.lower()
      #Remove punctuation marks from tokens
      content = content.translate(str.maketrans(string.punctuation, " "*ln,''))
      #Perform word tokenization
      ctokens = word_tokenize(content)
      #Remove stopwords from tokens and do lemmatization
      #Checking length, if length = 1
      ctokens = [lema.lemmatize(s) for s in ctokens if s not in stopwords.words('english') and s.isalpha and len(s)>1]
      return ctokens
```

```
[12] #performing preprocessing and saving the data
     docs = []
     # word_list={}
     for path in file_list:
       file = open(path, 'r', encoding='cp1250')
       text = file.read().strip()
       x=pre_process(text)
       file.close()
       docs.append(x)
```

```
'docs_pd = pd.DataFrame([docs,classes]).T\nprint(docs_pd)\ndocs_pd[0] = pre_pro
```

Converting the data frames to CSV :

```
#Converting the dataframes to csv
docs_pd = pd.DataFrame([docs,classes]).T
docs_pd.to_pickle("docs_pd")
docs_pd.to_csv("docs_pd.csv")
docs_pd
```

	0	1
0	[xref, cantaloupe, srv, c, cmu, edu, comp, gra...	comp.graphics
1	[xref, cantaloupe, srv, c, cmu, edu, comp, gra...	comp.graphics
2	[newsgroups, comp, graphic, path, cantaloupe, ...	comp.graphics
3	[newsgroups, comp, graphic, path, cantaloupe, ...	comp.graphics

b) Implementation of TF-ICF

```
#Class for implementing Naive Bayes algorithm with TF-ICF
class NaiveBayes_tf_icf:

  #Function to predict
  def predict(self,X_test):
    predc = []
    for i in range(len(X_test)):
        classes_words_probability = []
        for l in label:
            words_probability = 0
            for word in X_test[i]:
                fr, cn = self._word_freq(word, l)
                pp = (fr+1) /(cn+len(self._unique_words))
                words_probability += np.log(pp)
            classes_words_probability.append(words_probability)
        predc.append(label[np.argmax(classes_words_probability)])
    return predc

  #Function to compute confusion matrix
  def confusion_matrix(self, ypred, ytest):
    matrix= np.zeros((len(label), len(label))).astype(int)
    for i in range(len(ypred)):
        matrix[label.index(ypred[i])][label.index(ytest[i])]+= 1
    return matrix
```

```
  #Function to compute accuracy
  def calculate_accuracy(self, ypred, ytest):
    return len([1 for i in range(len(ypred)) if ypred[i] == ytest[i]])/len(ypred)


  #Function to compute word frequency
  def _word_freq(self, word, label):
    try:
        return self._word_freq_per_class[label, word], self._number_words_perclass[label]
    except:
        return 0, self._number_words_perclass[label]

  #Calculate tf-icf
  def _calculate_tf_icf(self):
    self._tf_icf = {}
    c = Counter(self._word_list)
    for i in set(self._word_list):
      tf = c[i]
      icf = np.log(len(self._m_dict)/self._class_word[i]+1)
      self._tf_icf[i] = tf*icf
```

```
#Function to fit the data
def fit(self,X_train,y_train, k):
  words = X_train
  self._N = len(words)
  classes = y_train
  self._m_dict = {}
  for i in range(self._N):
    if classes[i] in self._m_dict.keys():
        self._m_dict[classes[i]] = self._m_dict[classes[i]] + words[i]
    else:
        self._m_dict[classes[i]] = words[i]

  #Listing words containing multiple occurence of same word
  self._word_list = []
  for i in self._m_dict:
      self._word_list = self._word_list + self._m_dict[i]

  #Count of word per class
  self._class_word = {}
  for i in self._m_dict:
    l=self._m_dict[i]
    for j in set(l):
      if j not in self._class_word.keys():
        self._class_word[j] = 1
      else:
        self._class_word[j] += 1
  self._calculate_tf_icf()
  sorted_x = sorted(self._tf_icf.items(), key = operator.itemgetter(1), reverse=True)
```

```
  #considering top k features
  self._unique_words = [i[0] for i in sorted_x[:int(len(sorted_x)*k)]]
  self._word_freq_per_class = {}
  self._number_words_perclass = {}
  for i in label:
      freq_list= Counter(self._m_dict[i])
      for j in self._unique_words:
          self._word_freq_per_class[i,j] = freq_list[j]
          if i in self._number_words_perclass.keys():
              self._number_words_perclass[i] = self._number_words_perclass[i] +freq_list[j]
          else:
              self._number_words_perclass[i] = freq_list[j]
  self._freq_train = {}
  for i in y_train:
    if i not in self._freq_train.keys():
      self._freq_train[i] = 1
    else:
      self._freq_train[i] += 1
```

C) Training the data with Naive Bayes Classifier

```
[24] #50:50, 70:30, and 80:20 training and testing split ratios
     ratio = [0.5,0.7,0.8]
     naive_tf_dict = []

     for i in range(3):
       train = docs_pd.sample(frac=ratio[i],random_state=42)
       xtrain, ytrain = train[0].tolist(),train[1].tolist()
       test = docs_pd.sample(frac=1,random_state=42).drop(train.index)

       xtest,ytest = test[0].tolist(),test[1].tolist()
       nb = NaiveBayes_tf_icf()
       #Fitting the xtrain and ytrain
       #Taking k as 500
       k = 500
       nb.fit(xtrain, ytrain, k)
       ypred = nb.predict(xtest)

       r = int(ratio[i]*100)
       accuracy = nb.calculate_accuracy(ypred, ytest)*100
       naive_tf_dict.append(accuracy)

       print("At ratio {}:{}".format(r,100-r))
       print("\n")
       print("Confusion Matrix is given by: \n",nb.confusion_matrix(ypred,ytest))
       print("\n")
       print("Accuracy is {:.2f} \n".format(accuracy))
```

Output

Accuracy and Confusion matrix at the ratio of 50 and 50 :

```
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
```

Accuracy and Confusion matrix at the ratio of 70 and 30 :

```
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]]


Accuracy is 98.20
```

Accuracy and Confusion matrix at the ratio of 80 and 20 :

```
At ratio 80:20


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

Naive Bayes using TF-ICF has less noise due to feature selection which only selects the important words whose TF-ICF value is more than other words.
Naive Bayes with TF-ICf performs better than the normal Naive Bayes because of top k feature selection.