Assignment 2 part A

March 9, 2020

1 Assignment 2 - Part (A) : Naive Bayes

- In this part I have implemented Naive bayes from scratch for sentiment analysis on twitter dataset.
- In this part I have worked on the this Dataset.

[0]:
```
import pandas as pd
from tqdm import tqdm
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import os
import math
import random
import re
from time import time
import pickle

from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.tokenize import TweetTokenizer
import nltk
import re

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import GaussianNB

from joblib import Parallel, delayed
import multiprocessing

from sklearn.metrics import roc_curve, auc
from sklearn.metrics import roc_auc_score

from scipy.stats import boxcox
from sklearn.preprocessing import  StandardScaler
```

1.1 Part (a) : Implementing Naive Bayes and getting accuracy on Train and Test Datasets.

- In this part I have implemented Naive Bayes class that will be used in all the consecutive parts of the question.
- In this part I have computerd thetas for all the words in the vocabulary for both output classes 0 and 4.
- I have also incorporated various flags like addBigrams, addTrigrams and addPOS that will add features as mentioned in part e.
- I have also created one metric class that will compute various performence metrics for our model like accuracy score, confusion matrix and ROC curve.

[0]:
```
class NaiveBayes:

    dict0, dict4 = dict(), dict()
    phi0, phi4, = 0, 0
    n0, n4, = 0, 0
    vocab=list()
    v=0
    splitString = ',|\.| '
    tt=TweetTokenizer(preserve_case=False, strip_handles=True)
    simpleTokenizer = lambda self,s : [word for word in re.split(self.
↪splitString, s) if word!='' ]
    tokenizer = simpleTokenizer

    tweetTok = lambda self,s : self.tt.tokenize(s)

    def __init__(self, useTweetTokenizer=False, addBigrams=False,␣
↪addTrigrams=False, addPOS=False):
        dict0, dict4 = dict(), dict()
        if useTweetTokenizer:
            self.tokenizer = self.tweetTok
        self.addBigrams = addBigrams
        self.addTrigrams = addTrigrams
        self.addPOS = addPOS

    def fit(self, X_train, Y_train):
        #----Creating vocabulary-----
        st = set()

        def getPOS(doc):
            words = self.tokenizer(doc)
            if words!=[]:
                return list((list(zip(*nltk.pos_tag(words))))[0])
            else:
                return []
```

```
        if self.addPOS:
            pos_lst = Parallel(n_jobs=6)(delayed(getPOS)(doc) for doc in␣
↪tqdm(X_train, desc='Generating POS List'))

        for (doc,j) in tqdm(list(zip(X_train, range(len(X_train)))),␣
↪desc='Generating Vocabulary'):
            words = self.tokenizer(doc)
            for word in words:
                st.add(word)
            if self.addBigrams:
                for i in range(len(words)-1):
                    st.add(words[i]+' '+words[i+1])
            if self.addTrigrams:
                for i in range(len(words)-2):
                    st.add(words[i]+' '+words[i+1]+' '+words[i+2])
            if self.addPOS:
                for pos in pos_lst[j]:
                    st.add(pos)
        self.vocab = list(st) # List of vocabulary
        self.v=len(self.vocab) #Vocabulary size

        #Splitting Dataset According to label
        X_train_0 = np.array(X_train)[np.where(Y_train == 0)]
        X_train_4 = np.array(X_train)[np.where(Y_train == 4)]

        #Creating empty dictionaries for every class
        self.dict0 = dict(zip(self.vocab, np.ones(len(self.vocab), dtype=int)))
        self.dict4 = dict(zip(self.vocab, np.ones(len(self.vocab), dtype=int)))

        #Counting occurance of each word in all three classes and then updating
        #corrsponding dictionary
        for stat, j in tqdm(list(zip(X_train_0, range(len(X_train_0)))),␣
↪desc='Generating Theta List for Label=0'):
            words = self.tokenizer(stat)
            for word in words:
                self.dict0[word] = self.dict0[word]+1
            if self.addBigrams:
                for i in range(len(words)-1):
                    bGram = words[i]+' '+words[i+1]
                    self.dict0[bGram] += 1
            if self.addTrigrams:
                for i in range(len(words)-2):
                    tGram = words[i]+' '+words[i+1]+' '+words[i+2]
                    self.dict0[tGram] += 1
            if self.addPOS:
                for pos in pos_lst[j]:
```

```
                self.dict0[pos] += 1
        for stat, j in tqdm(list(zip(X_train_4, range(len(X_train_4)))),␣
↪desc='Generating Theta List for Label=4'):
            words = self.tokenizer(stat)
            for word in words:
                self.dict4[word] = self.dict4[word]+1
            if self.addBigrams:
                for i in range(len(words)-1):
                    bGram = words[i]+' '+words[i+1]
                    self.dict4[bGram] += 1
            if self.addTrigrams:
                for i in range(len(words)-2):
                    tGram = words[i]+' '+words[i+1]+' '+words[i+2]
                    self.dict4[tGram] += 1
            if self.addPOS:
                for pos in pos_lst[j]:
                    self.dict4[pos] += 1

        #----Finding number of words in all these three classes
        self.n0 = sum(list(self.dict0.values()))
        self.n4 = sum(list(self.dict4.values()))

        self.dict0 = dict(zip(self.dict0.keys(), np.log(np.array(list(self.
↪dict0.values()), dtype=int)/(self.n0+self.v))))
        self.dict4 = dict(zip(self.dict4.keys(), np.log(np.array(list(self.
↪dict4.values()), dtype=int)/(self.n4+self.v))))


        #----Finding class priors
        self.phi0 = (X_train_0.shape[0]+1) / (X_train.shape[0]+2)
        self.phi4 = (X_train_4.shape[0]+1) / (X_train.shape[0]+2)

    def predict(self, X):
        pred_lst=list()

        for x, j in tqdm(list(zip(X,range(len(X)))), desc='Generating␣
↪Predictions'):
            prob0 = math.log(self.phi0)
            prob4 = math.log(self.phi4)
            words = self.tokenizer(x)
            for word in words:
                prob0 += self.dict0.get(word) if self.dict0.get(word)!=None␣
↪else math.log(1/(self.n0+self.v))
                prob4 += self.dict4.get(word) if self.dict4.get(word)!=None␣
↪else math.log(1/(self.n4+self.v))
            if self.addBigrams:
```

```
                for i in range(len(words)-1):
                    bGram = words[i]+' '+words[i+1]
                    prob0 += self.dict0.get(bGram) if self.dict0.get(bGram)!=None else math.log(1/(self.n0+self.v))
                    prob4 += self.dict4.get(bGram) if self.dict4.get(bGram)!=None else math.log(1/(self.n4+self.v))
            if self.addTrigrams:
                for i in range(len(words)-2):
                    tGram = words[i]+' '+words[i+1]+' '+words[i+2]
                    prob0 += self.dict0.get(tGram) if self.dict0.get(tGram)!=None else math.log(1/(self.n0+self.v))
                    prob4 += self.dict4.get(tGram) if self.dict4.get(tGram)!=None else math.log(1/(self.n4+self.v))
            if self.addPOS:
                for pos in (list(zip(*nltk.pos_tag(words)))[1]):
                    prob0 += self.dict0.get(pos) if self.dict0.get(pos)!=None else math.log(1/(self.n0+self.v))
                    prob4 += self.dict4.get(pos) if self.dict4.get(pos)!=None else math.log(1/(self.n4+self.v))
            pred_lst.append(4*(np.array([prob0, prob4]).argmax()))
        return pred_lst

    def predict_log_proba(self, X):
        log_proba0=list()
        log_proba4=list()

        for x, j in tqdm(list(zip(X,range(len(X)))), desc='Generating Log probabilities'):
            prob0 = math.log(self.phi0) #if self.phi0!=0 else 0
            prob4 = math.log(self.phi4) #if self.phi0!=0 else 0
            words = self.tokenizer(x)
            for word in words:
                prob0 += self.dict0.get(word) if self.dict0.get(word)!=None else np.log(1/(self.n0+self.v))
                prob4 += self.dict4.get(word) if self.dict4.get(word)!=None else np.log(1/(self.n4+self.v))
            if self.addBigrams:
                for i in range(len(words)-1):
                    bGram = words[i]+' '+words[i+1]
                    prob0 += self.dict0.get(bGram) if self.dict0.get(bGram)!=None else math.log(1/(self.n0+self.v))
                    prob4 += self.dict4.get(bGram) if self.dict4.get(bGram)!=None else math.log(1/(self.n4+self.v))
            if self.addTrigrams:
                for i in range(len(words)-2):
                    tGram = words[i]+' '+words[i+1]+' '+words[i+2]
```

```
                    prob0 += self.dict0.get(tGram) if self.dict0.get(tGram)!
↪=None else math.log(1/(self.n0+self.v))
                    prob4 += self.dict4.get(tGram) if self.dict4.get(tGram)!
↪=None else math.log(1/(self.n4+self.v))
            if self.addPOS:
                for pos in (list(zip(*nltk.pos_tag(words)))[1]):
                    prob0 += self.dict0.get(pos) if self.dict0.get(pos)!=None␣
↪else math.log(1/(self.n0+self.v))
                    prob4 += self.dict4.get(pos) if self.dict4.get(pos)!=None␣
↪else math.log(1/(self.n4+self.v))
            log_proba0.append(prob0)
            log_proba4.append(prob4)
        return [log_proba0, log_proba4]

    def randomPred(self, size):
        choices = [0,4]
        pred = list()
        for x in range(size):
            pred.append(random.choice(choices))
        return pred

    def majorityPred(self, size, out=4):
        return [out]*size



class metrics:

    def accuracy_score(y_true, y_pred):
        true = list(y_true == y_pred).count(True)
        false = list(y_true == y_pred).count(False)
        return true*100/(true+false)

    def confusion_matrix(y_true, y_pred, title = 'Confusion Matrix'):
        cm=np.zeros((2,2), dtype=int)
        for (y_hat, y) in zip(y_pred, y_true):
            cm[y_hat//4][y//4] += 1

        sns.heatmap(cm, annot=True, fmt="d",linecolor='black',linewidth='0', \
        cmap='Blues',xticklabels=[0,4], yticklabels=[0,4], cbar=False)
        plt.xlabel('Actual')
        plt.ylabel('Predicted')
        plt.title(title)
        plt.show()
```

- **Loading the Dataset**

[0]:
```
train_data = pd.read_csv('/content/drive/My Drive/ML/training.1600000.processed.
 ↪noemoticon.csv', sep=',', encoding='latin-1', names=['polarity', 'id',␣
 ↪'date', 'query', 'user', 'text'])

X_train = train_data['text']
Y_train = train_data['polarity']

test_data = pd.read_csv('/content/drive/My Drive/ML/testdata.manual.2009.06.14.
 ↪csv', sep=',', encoding='latin-1', names=['polarity', 'id', 'date', 'query',␣
 ↪'user', 'text'])

X_test = test_data['text']
Y_test = test_data['polarity']

X_test_new_0 = np.array(X_test)[np.where(Y_test==0)]# or Y_test==4)]
X_test_new_4 = np.array(X_test)[np.where(Y_test==4)]

X_test = np.concatenate((X_test_new_0, X_test_new_4))
Y_test = pd.Series(np.concatenate((np.zeros(X_test_new_0.shape[0]), np.
 ↪zeros(X_test_new_4.shape[0])+4)), dtype=int)
```

- **Intererting finding :** Data contains duplicate Entries

[4]:
```
vc = train_data[train_data.duplicated(subset=['id','text'], keep=False)]
print('Number of duplicate entries : ', len(vc))
vc.sort_values('id').head(10)
```

```
Number of duplicate entries :  3370
```

[4]:
```
        polarity  …                                               text
213            0  …  Awwh babs… you look so sad underneith that s…
800261         4  …  Awwh babs… you look so sad underneith that s…
275            0  …  Haven't tweeted nearly all day  Posted my webs…
800300         4  …  Haven't tweeted nearly all day  Posted my webs…
989            0  …  @hellobebe I also send some updates in plurk b…
801280         4  …  @hellobebe I also send some updates in plurk b…
1177           0  …  good night swetdreamss to everyonee   and jare…
801573         4  …  good night swetdreamss to everyonee   and jare…
1254           0  …  @ientje89 aw i'm fine too thanks! yeah i miss …
801650         4  …  @ientje89 aw i'm fine too thanks! yeah i miss …

[10 rows x 6 columns]
```

1.1.1 Getting Accuracy on train and test data

[5]:
```
nb_original=NaiveBayes()

nb_original.fit(X_train, Y_train)

train_pred = nb_original.predict(X_train)
print('Accuracy on Train Data : %.4f'%(metrics.accuracy_score(Y_train,␣
 ↪train_pred)))

test_pred = nb_original.predict(X_test)
print('Accuracy on Test Data : %.4f'%(metrics.accuracy_score(Y_test,␣
 ↪test_pred)))
```

```
Generating Vocabulary: 100%|      | 1600000/1600000 [00:11<00:00,
139901.30it/s]
Generating Theta List for Label=0: 100%|      | 800000/800000 [00:10<00:00,
74279.12it/s]
Generating Theta List for Label=4: 100%|      | 800000/800000 [00:10<00:00,
76293.41it/s]
Generating Predictions: 100%|      | 1600000/1600000 [00:46<00:00,
34253.83it/s]
Generating Predictions: 100%|      | 359/359 [00:00<00:00, 25570.25it/s]
```

```
Accuracy on Train Data : 84.8601
Accuracy on Test Data : 81.3370
```

1.2 Part (b) : Getting base model accuracy by finding accuracy of test data using random prediction and majority vote prediction

[6]:
```
nb=NaiveBayes()
random_pred = nb.randomPred(Y_test.shape[0])
print('Accuracy of Random predictor on Test Data (Attempt 1): ',metrics.
 ↪accuracy_score(Y_test, random_pred))
random_pred = nb.randomPred(Y_test.shape[0])
print('Accuracy of Random predictor on Test Data (Attempt 2): ',metrics.
 ↪accuracy_score(Y_test, random_pred))
random_pred = nb.randomPred(Y_test.shape[0])
print('Accuracy of Random predictor on Test Data (Attempt 3): ',metrics.
 ↪accuracy_score(Y_test, random_pred))

majority_pred = nb.majorityPred(Y_test.shape[0], out=4)
print('Accuracy of Majority Vote(4) predictor on Test Data : ',metrics.
 ↪accuracy_score(Y_test, majority_pred))
majority_pred = nb.majorityPred(Y_test.shape[0], out=0)
```

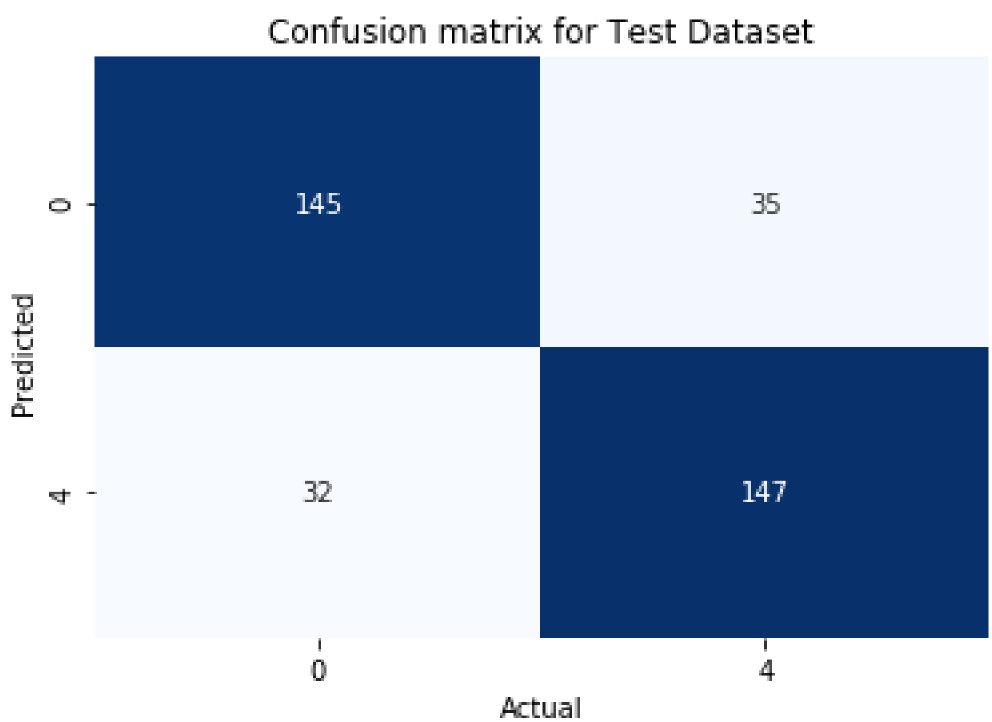
```
print('Accuracy of Majority Vote(0) predictor on Test Data : ',metrics.
    ↪accuracy_score(Y_test, majority_pred))
```

```
Accuracy of Random predictor on Test Data (Attempt 1):  51.25348189415042
Accuracy of Random predictor on Test Data (Attempt 2):  47.910863509749305
Accuracy of Random predictor on Test Data (Attempt 3):  50.41782729805014
Accuracy of Majority Vote(4) predictor on Test Data :  50.69637883008357
Accuracy of Majority Vote(0) predictor on Test Data :  49.30362116991643
```

- Here we can see that our model gave test accuracy of 81.3370 and random and Majority vode predictor gave accuracies of around 50% hence our model is performing significantely better.

1.3 Part (c) : Plotting Confusion matrix for the Train as well as Test Dataset for our Model

[7]:
```
metrics.confusion_matrix(Y_test, test_pred, title='Confusion matrix for Test␣
    ↪Dataset')
```



- **Observations from confusion matrix:**
  - Here we can see that diagonal entries are large as compared to non diagonal ones, hence our model is performing nice.
  - Also note that we have predicted 145 datapoints with y=0 out of total 177 datapoints, whereas we have predicted 147 datapoints with y=4 correctly from total of 182 data-

points, hence although we have less number for class=0 but ratio with total datapoints with particular class label is same, therefore TNR and TPR are 0.8192 and 0.8077(Considering 0 as negative label and 1 as positive label). Hence our model is performing well for both the labels.

1.4 Part (d) : Stemming, Removing Stopwords and cleaning Data

In this part I have preprocessed the original data and for that I have followed follwing steps.

1. Remove usernames
2. Remove stopwords
3. Stemming

This method uses 6 cores of my machine to clean data in parallel. and you can set flags as mentioned in the docstring of the method.

[0]:
```
def cleanData(X, useTweetTokenizer=False, removeNot=False,␣
↪usePorterStemmer=False):
    '''
        This method returns list of preprocessed text.

        **********
        Arguments:
            X : list of strings to clean
            useTweetTokenizer=False : Flag that mentiones which tokenizer to␣
↪use.
                                        True : use nltk's TweetTokenizer
                                        False : re.split('\,|,| ',s)
            removeNot=False : Remove not and words like aren't isn't from␣
↪stopwords.
            useProterStemmer=False : use Porter or snowball stemmer from nltk
        **********
    '''
    stop = set(stopwords.words('english')) #set of stopwords
    porter = nltk.stem.PorterStemmer()
    sno = nltk.stem.SnowballStemmer('english') #initialising the snowball␣
↪stemmer
    tt = TweetTokenizer(preserve_case=False, strip_handles=True)

    stemmer=sno
    if usePorterStemmer:
        stemmer=porter

    if removeNot:
        stop.remove('not')
        stop.remove('no')
        new_stop = set()
        for word in stop:
```

```
            if word.endswith('n\'t'):
                continue
            else:
                new_stop.add(word)
        stop = new_stop

    splitString = ',|\.| '

    def cleanUserName(sentence): #function to clean the word of any html-tags
        cleanr = re.compile('@\S*')
        cleantext = re.sub(cleanr, '', sentence)
        return cleantext

    simpleTok = lambda s : [x for x in re.split(splitString, cleanUserName(s))␣
↪if x!='']
    tweetTok = lambda s : tt.tokenize(s)

    tokenize = simpleTok
    if useTweetTokenizer:
        tokenize = tweetTok

    def cleanSingleDoc(sent):
        filtered_sentence=[]
        words = tokenize(sent)

        for word in words:
            if((len(word)>1 or word.isalpha()) & (word not in stop)):
                s=(stemmer.stem(word)).encode('utf-8')
                filtered_sentence.append(s)
            else:
                continue
        str1 = b" ".join(filtered_sentence) #final string of cleaned words
        return (str1.decode('utf-8'))

    final_string = Parallel(n_jobs=10)(delayed(cleanSingleDoc)(sent) for sent␣
↪in tqdm(X, desc = 'Cleaning'))

    return np.array(final_string)
```

[9]:
```
#Cleaning our dataset

X_train_cleaned = cleanData(X_train, useTweetTokenizer=True, removeNot=True,␣
↪usePorterStemmer=True)
X_test_cleaned = cleanData(X_test, useTweetTokenizer=True, removeNot=True,␣
↪usePorterStemmer=True)
```

```
Cleaning: 100%|      | 1600000/1600000 [00:52<00:00, 30637.64it/s]
```

```
Cleaning: 100%|      | 359/359 [00:00<00:00, 2237.25it/s]
```

[10]:
```
#Getting Test accuracy on cleaned Data

nb_cleaned = NaiveBayes()

nb_cleaned.fit(X_train_cleaned, Y_train)

# train_pred_cleaned = nb_cleaned.predict(X_train_cleaned)
# print('Accuracy on Cleaned Train Data : %.4f %c'%(metrics.
 ↪accuracy_score(Y_train, train_pred_cleaned),'%'))

test_pred_cleaned = nb_cleaned.predict(X_test_cleaned)
print('Accuracy on Cleaned Test Data : %.4f %c'%(metrics.accuracy_score(Y_test,␣
 ↪test_pred_cleaned),'%'))
```

```
Generating Vocabulary: 100%|      | 1600000/1600000 [00:06<00:00,
232238.60it/s]
Generating Theta List for Label=0: 100%|      | 800000/800000 [00:06<00:00,
131717.72it/s]
Generating Theta List for Label=4: 100%|      | 800000/800000 [00:05<00:00,
134640.01it/s]
Generating Predictions: 100%|      | 359/359 [00:00<00:00, 36283.26it/s]

Accuracy on Cleaned Test Data : 82.7298 %
```

- **Observations :**

Here we can see that after cleaning our test accuracy increases from 81.337 to 82.7298 which is significant.

1.5 Part (e) : Feature Engineering

- In this part I have implemented 3 features that are bi-grams, tri-grams and part of speech taggings. They are implemented as flags in main class that I implemented earlier in part (a). We can just pass argument to that class' constructor and accordingly model will be trained.
- Now testing model by adding various features.

[11]:
```
nb_bi = NaiveBayes(addBigrams=True)

nb_bi.fit(X_train_cleaned, Y_train)

test_pred_cleaned = nb_bi.predict(X_test_cleaned)
print('Accuracy on Cleaned Test Data : %.4f %c'%(metrics.accuracy_score(Y_test,␣
 ↪test_pred_cleaned),'%'))
```

```
Generating Vocabulary: 100%|          | 1600000/1600000 [00:13<00:00,
117590.02it/s]
Generating Theta List for Label=0: 100%|          | 800000/800000 [00:13<00:00,
59814.75it/s]
Generating Theta List for Label=4: 100%|          | 800000/800000 [00:13<00:00,
60814.51it/s]
Generating Predictions: 100%|          | 359/359 [00:00<00:00, 18135.95it/s]

Accuracy on Cleaned Test Data : 82.4513 %
```

[12]:
```python
nb_tri = NaiveBayes(addTrigrams=True)

nb_tri.fit(X_train_cleaned, Y_train)

test_pred_cleaned = nb_tri.predict(X_test_cleaned)
print('Accuracy on Cleaned Test Data : %.4f %c'%(metrics.accuracy_score(Y_test,␣
↪test_pred_cleaned),'%'))
```

```
Generating Vocabulary: 100%|          | 1600000/1600000 [00:14<00:00,
110188.15it/s]
Generating Theta List for Label=0: 100%|          | 800000/800000 [00:13<00:00,
58454.80it/s]
Generating Theta List for Label=4: 100%|          | 800000/800000 [00:13<00:00,
60073.27it/s]
Generating Predictions: 100%|          | 359/359 [00:00<00:00, 18361.30it/s]

Accuracy on Cleaned Test Data : 83.8440 %
```

[13]:
```python
nb_BiTri = NaiveBayes(addBigrams=True, addTrigrams=True)

nb_BiTri.fit(X_train_cleaned, Y_train)

test_pred_BiTri = nb_BiTri.predict(X_test_cleaned)
print('Accuracy on Cleaned Test Data : %.4f %c'%(metrics.accuracy_score(Y_test,␣
↪test_pred_BiTri),'%'))
```

```
Generating Vocabulary: 100%|          | 1600000/1600000 [00:20<00:00,
77910.86it/s]
Generating Theta List for Label=0: 100%|          | 800000/800000 [00:20<00:00,
38443.96it/s]
Generating Theta List for Label=4: 100%|          | 800000/800000 [00:19<00:00,
40176.52it/s]
Generating Predictions: 100%|          | 359/359 [00:00<00:00, 12420.65it/s]

Accuracy on Cleaned Test Data : 82.4513 %
```

```
[14]: nb_POS = NaiveBayes(addPOS=True)

nb_POS.fit(X_train_cleaned, Y_train)

test_pred_POS = nb_POS.predict(X_test_cleaned)
print('Accuracy on Cleaned Test Data : %.4f %c'%(metrics.accuracy_score(Y_test,␣
 ↪test_pred_POS),'%'))
```

```
Generating POS List: 100%|      | 1600000/1600000 [04:20<00:00, 6135.13it/s]
Generating Vocabulary: 100%|      | 1600000/1600000 [00:09<00:00,
174632.52it/s]
Generating Theta List for Label=0: 100%|      | 800000/800000 [00:08<00:00,
89014.69it/s]
Generating Theta List for Label=4: 100%|      | 800000/800000 [00:09<00:00,
85560.08it/s]
Generating Predictions: 100%|      | 359/359 [00:00<00:00, 1100.48it/s]

Accuracy on Cleaned Test Data : 76.8802 %
```

```
[15]: nb_POS_Tri = NaiveBayes(addPOS=True, addTrigrams=True)

nb_POS_Tri.fit(X_train_cleaned, Y_train)

test_pred_POS_Tri = nb_POS_Tri.predict(X_test_cleaned)
print('Accuracy on Cleaned Test Data : %.4f %c'%(metrics.accuracy_score(Y_test,␣
 ↪test_pred_POS_Tri),'%'))
```

```
Generating POS List: 100%|      | 1600000/1600000 [04:16<00:00, 6226.81it/s]
Generating Vocabulary: 100%|      | 1600000/1600000 [00:17<00:00,
91743.44it/s]
Generating Theta List for Label=0: 100%|      | 800000/800000 [00:17<00:00,
47039.76it/s]
Generating Theta List for Label=4: 100%|      | 800000/800000 [00:17<00:00,
44733.98it/s]
Generating Predictions: 100%|      | 359/359 [00:00<00:00, 1366.89it/s]

Accuracy on Cleaned Test Data : 77.7159 %
```

- **Observations :**
- As we can see that adding trigrams gave us best accuracy among all the models.
- All the other model's accuracies are summerized in the table below.

Features	Test Accuracy
Bigrams	82.4513
TriGrams	83.8440

Features	Test Accuracy
BiGrams+TriGrams	82.4513
POS	76.8802
POS+Trigrams	77.7159

2 Part (f) : tf-idf + GaussianNB

- In this part I have got tf-idf vectors of the trained data and then applied GaussianNB model on top of that and found accuracies for test dataset.
- In this part I was doing it by partial_fitting data on a single model but taht was taking more than 1 hour and 10 minutes to train as it was using only one core of cpu. So I vertically splitted the train Data and trained multiple models in parallel and then predicted on test dataset using predict_log_proba and taking sum of all teh log probabilities as naive bayes assumes that features are independent and subtracting class priors accordingy as in each model we would have multiplied it. This implementation was so much faster and trained model within 10 minutes(Around 8 minutes).

[16]:
```
tfidf = TfidfVectorizer(dtype=np.float32)

X_train_tfidf = tfidf.fit_transform(X_train_cleaned)
X_test_tfidf = tfidf.transform(X_test_cleaned)

print('Shape of X_train_tfidf : ',X_train_tfidf.shape)
```

```
Shape of X_train_tfidf :  (1600000, 307449)
```

[0]:
```
class MyGaussianNB:
    '''
    This class is implementation to train gaussian naive bayes model on sparse␣
 ↪data set on multiple cores.

    *************
    Parameters:

    n_jobs=1 : Number of cores
    v_split_size=200 : Max number of features in each base models
    *************
    '''
    def __init__(self, n_jobs=1, v_split_size=200):
        self.n_jobs = n_jobs
        self.v_split_size = v_split_size

    def fit(self, X_tr, Y_tr):
        start_time = time()
```

```
        def fit_parallel(i):
            return GaussianNB().fit(X_tr[:,i:(i+self.v_split_size)].toarray(),␣
↪Y_tr)
        self.models = Parallel(n_jobs=self.n_jobs)(delayed(fit_parallel)(i) for␣
↪i in tqdm(range(0, X_tr.shape[1], self.v_split_size), desc='Fitting Multiple␣
↪models based on verticle splits'))

        end_time = time()
        print('Completed training in %.2f minutes'%((end_time-start_time)/60))

    def predict(self, X):

        log_proba=np.zeros((X.shape[0],2))

        for model,i in zip(self.models, range(0, X.shape[1], self.
↪v_split_size)):
            log_proba += model.predict_log_proba(X[:,i:(i+self.v_split_size)].
↪todense())

        log_proba -= (len(self.models)-1)*self.models[0].class_prior_
        return log_proba.argmax(axis=1)*4

    def predict_log_proba(self, X):

        log_proba=np.zeros((X.shape[0],2))

        for model,i in zip(self.models, range(0, X.shape[1], self.
↪v_split_size)):
            log_proba += model.predict_log_proba(X[:,i:(i+self.v_split_size)].
↪todense())

        log_proba -= (len(self.models)-1)*self.models[0].class_prior_
        return log_proba
```

[18]:
```
myGNB_full = MyGaussianNB(n_jobs=10, v_split_size=200)
myGNB_full.fit(X_train_tfidf, Y_train)

pred_gnb = myGNB_full.predict(X_test_tfidf)

print('Accuracy on Test Dataset using whole dataset without using min_df : %.2f␣
↪%c'%(sum(pred_gnb == Y_test) * 100 / len(pred_gnb), '%'))
```

```
Fitting Multiple models based on verticle splits:   2%|          | 30/1538
[00:15<10:25,  2.41it/s]/usr/local/lib/python3.6/dist-
packages/joblib/externals/loky/process_executor.py:706: UserWarning: A worker
stopped while some jobs were given to the executor. This can be caused by a too
short worker timeout or by a memory leak.
```

```
  "timeout or by a memory leak.", UserWarning
Fitting Multiple models based on verticle splits: 100%|      | 1538/1538
[10:18<00:00,  2.49it/s]
```

```
Completed training in 10.48 minutes
Accuracy on Test Dataset using whole dataset without using min_df : 49.58 %
```

- **Doing Select Percentile to select 2% features**

[19]:
```
from sklearn.feature_selection import SelectPercentile
from sklearn.feature_selection import f_classif

sel = SelectPercentile(f_classif, percentile=2).fit(X_train_tfidf, Y_train)

X_train_2 = sel.transform(X_train_tfidf)
X_test_2 = sel.transform(X_test_tfidf)

print('Dimnsions of train dataset after selecting 2 percentile features : ',␣
 ↪X_train_2.shape)
```

```
Dimnsions of train dataset after selecting 2 percentile features :  (1600000,
6149)
```

[27]:
```
myGNB_full_2 = MyGaussianNB(n_jobs=6, v_split_size=150)
myGNB_full_2.fit(X_train_2, Y_train)

pred_gnb_2 = myGNB_full_2.predict(X_test_2)
print('Accuracy on Test Dataset having 10 percentile features without using␣
 ↪min_df : %.2f %c'%(sum(pred_gnb_2 == Y_test) * 100 / len(pred_gnb_2), '%'))
```

```
Fitting Multiple models based on verticle splits: 100%|      | 41/41
[00:11<00:00,  3.50it/s]
```

```
Completed training in 0.27 minutes
Accuracy on Test Dataset having 10 percentile features without using min_df :
70.19 %
```

- **Doing Select Percentile to select 1% features**

[30]:
```
sel = SelectPercentile(f_classif, percentile=1).fit(X_train_tfidf, Y_train)

X_train_1 = sel.transform(X_train_tfidf)
X_test_1 = sel.transform(X_test_tfidf)

print('Dimnsions of train dataset after selecting 1 percentile features : ',␣
 ↪X_train_1.shape)
```

```
Dimnsions of train dataset after selecting 1 percentile features :  (1600000,
3075)
```

[29]:
```
myGNB_full_1 = MyGaussianNB(n_jobs=6, v_split_size=150)
myGNB_full_1.fit(X_train_1, Y_train)
pred_gnb_1 = myGNB_full_1.predict(X_test_1)
print('Accuracy on Test Dataset having 10 percentile features without using␣
↪min_df : %.2f %c'%(sum(pred_gnb_1 == Y_test) * 100 / len(pred_gnb_1), '%'))
```

```
Fitting Multiple models based on verticle splits: 100%|      | 21/21
[00:04<00:00,  4.50it/s]

Completed training in 0.15 minutes
Accuracy on Test Dataset having 10 percentile features without using min_df :
73.54 %
```

- **Making tf-idf vectors with min_df=100 and max_features=5000**

[42]:
```
tfidf = TfidfVectorizer(dtype=np.float32, min_df=100, max_features=5000)

X_train_tfidf_mindf = tfidf.fit_transform(X_train_cleaned)
X_test_tfidf_mindf = tfidf.transform(X_test_cleaned)

print('Shape of X_train_tfidf : ',X_train_tfidf_mindf.shape)
```

```
Shape of X_train_tfidf :  (1600000, 5000)
```

[43]:
```
myGNB_mindf = MyGaussianNB(n_jobs=6, v_split_size=150)
myGNB_mindf.fit(X_train_tfidf_mindf, Y_train)
pred_mindf = myGNB_mindf.predict(X_test_tfidf_mindf)

print('Accuracy on Test Dataset using whole dataset with using min_df : %.2f␣
↪%c'%(sum(pred_mindf == Y_test) * 100 / len(pred_mindf), '%'))
```

```
Fitting Multiple models based on verticle splits: 100%|      | 34/34
[00:09<00:00,  3.61it/s]

Completed training in 0.23 minutes
Accuracy on Test Dataset using whole dataset with using min_df : 72.42 %
```

- **Doing Select Percentile to select 10% features**

[44]:
```
sel = SelectPercentile(f_classif, percentile=10).fit(X_train_tfidf_mindf,␣
↪Y_train)

X_train_10_mindf = sel.transform(X_train_tfidf_mindf)
X_test_10_mindf = sel.transform(X_test_tfidf_mindf)
print('Dimnsions of train dataset after selecting 10 percentile features : ',␣
↪X_train_10_mindf.shape)
```

```
Dimnsions of train dataset after selecting 10 percentile features :  (1600000,
500)
```

```
[45]: myGNB_mindf_10 = MyGaussianNB(n_jobs=6, v_split_size=150)
myGNB_mindf_10.fit(X_train_10_mindf, Y_train)
pred_10_mindf = myGNB_mindf_10.predict(X_test_10_mindf)

print('Accuracy on Test Dataset having 10 percentile features with using min_df␣
↪: %.2f %c'%(sum(pred_10_mindf == Y_test) * 100 / len(pred_10_mindf), '%'))
```

```
Fitting Multiple models based on verticle splits: 100%|      | 4/4
[00:00<00:00, 4911.36it/s]

Completed training in 0.04 minutes
Accuracy on Test Dataset having 10 percentile features with using min_df : 78.83
%
```

- **Doing Select Percentile to select 15% features**

```
[46]: sel = SelectPercentile(f_classif, percentile=15).fit(X_train_tfidf_mindf,␣
↪Y_train)

X_train_15_mindf = sel.transform(X_train_tfidf_mindf)
X_test_15_mindf = sel.transform(X_test_tfidf_mindf)
print('Dimnsions of train dataset after selecting 10 percentile features : ',␣
↪X_train_15_mindf.shape)
```

```
Dimnsions of train dataset after selecting 10 percentile features :  (1600000,
750)
```

```
[47]: myGNB_mindf_15 = MyGaussianNB(n_jobs=6, v_split_size=150)
myGNB_mindf_15.fit(X_train_15_mindf, Y_train)
pred_15_mindf = myGNB_mindf_15.predict(X_test_15_mindf)

print('Accuracy on Test Dataset having 10 percentile features with using min_df␣
↪: %.2f %c'%(sum(pred_15_mindf == Y_test) * 100 / len(pred_15_mindf), '%'))
```

```
Fitting Multiple models based on verticle splits: 100%|      | 5/5
[00:00<00:00, 4974.27it/s]

Completed training in 0.04 minutes
Accuracy on Test Dataset having 10 percentile features with using min_df : 75.49
%
```

- **Doing Select Percentile to select 5% features**

```
[48]: sel = SelectPercentile(f_classif, percentile=5).fit(X_train_tfidf_mindf,␣
↪Y_train)

X_train_5_mindf = sel.transform(X_train_tfidf_mindf)
X_test_5_mindf = sel.transform(X_test_tfidf_mindf)
print('Dimnsions of train dataset after selecting 10 percentile features : ',␣
↪X_train_5_mindf.shape)
```

```
Dimnsions of train dataset after selecting 10 percentile features :  (1600000,
250)
```

[49]:
```
myGNB_mindf_5 = MyGaussianNB(n_jobs=6, v_split_size=150)
myGNB_mindf_5.fit(X_train_5_mindf, Y_train)
pred_5_mindf = myGNB_mindf_5.predict(X_test_5_mindf)

print('Accuracy on Test Dataset having 10 percentile features with using min_df␣
↪: %.2f %c'%(sum(pred_5_mindf == Y_test) * 100 / len(pred_5_mindf), '%'))
```

```
Fitting Multiple models based on verticle splits: 100%|      | 2/2
[00:00<00:00, 2159.79it/s]
```

```
Completed training in 0.04 minutes
Accuracy on Test Dataset having 10 percentile features with using min_df : 76.04
%
```

- **Observations:**
  - Here we can see that this implementation of GaussianNB took so less time due to parallelization.
  - Accuracies on the whole cleaned dataset is much poor.
  - But when we reduced dimensions using select_percentile then we got better accuracy.
  - Also when we initially reduced dimensions using min_df=100 and max_features=5000 then our model got trained very faster and we also got better accuracy numbers.
  - When we select the smaller set of features time required to train model decreases drastically as full tfidf dataset took 8 minutes to train on 10 cores whereas all the reduced dimensions datasets got trained in from a second to 10s of seconds.
  - All the accuracy figures are summerized in the figure below.

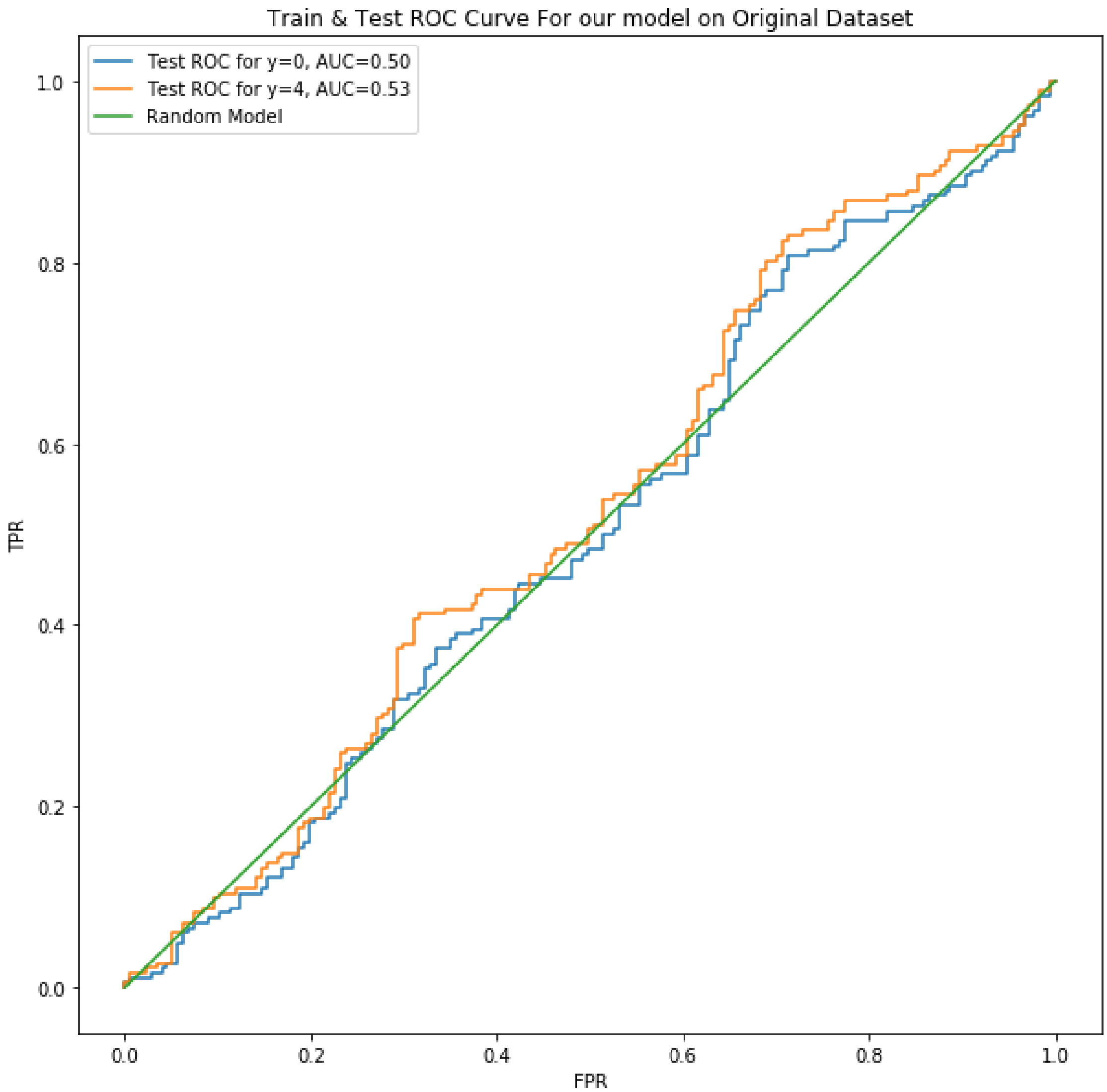
Model	Test Accuracy	Dimensions
Whole Dataset	49.58	307449
Whole Dataset + percentile=1	73.54	3075
Whole Dataset + percentile=2	70.19	6149
min_df Dataset	72.42	5000
min_df Dataset + percentile=10	78.83	500
min_df Dataset + percentile=15	75.49	750
min_df Dataset + percentile=5	76.04	250

3 Part (g): Plotting ROC

[50]:
```
test_proba = nb_original.predict_log_proba(X_test)
test_proba0 = np.array(test_proba)[0,:]
test_proba4 = np.array(test_proba)[1,:]

plt.figure(figsize=(10,10))
```

```
fpr, tpr, thresholds = roc_curve(Y_test, test_proba0, pos_label=4)
auc=roc_auc_score(Y_test==4, test_proba0)
plt.plot(fpr, tpr, label='Test ROC for y=0, AUC=%.2f'%auc)

fpr, tpr, thresholds = roc_curve(Y_test, test_proba4, pos_label=4)
auc=roc_auc_score(Y_test==4, test_proba4)
plt.plot(fpr, tpr, label='Test ROC for y=4, AUC=%.2f'%auc)

plt.plot([0,1],[0,1], label='Random Model')
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('Train & Test ROC Curve For our model on Original Dataset')
plt.legend()
plt.show()
```

```
Generating Log probabilities: 100%|     | 359/359 [00:00<00:00,
12945.72it/s]
```

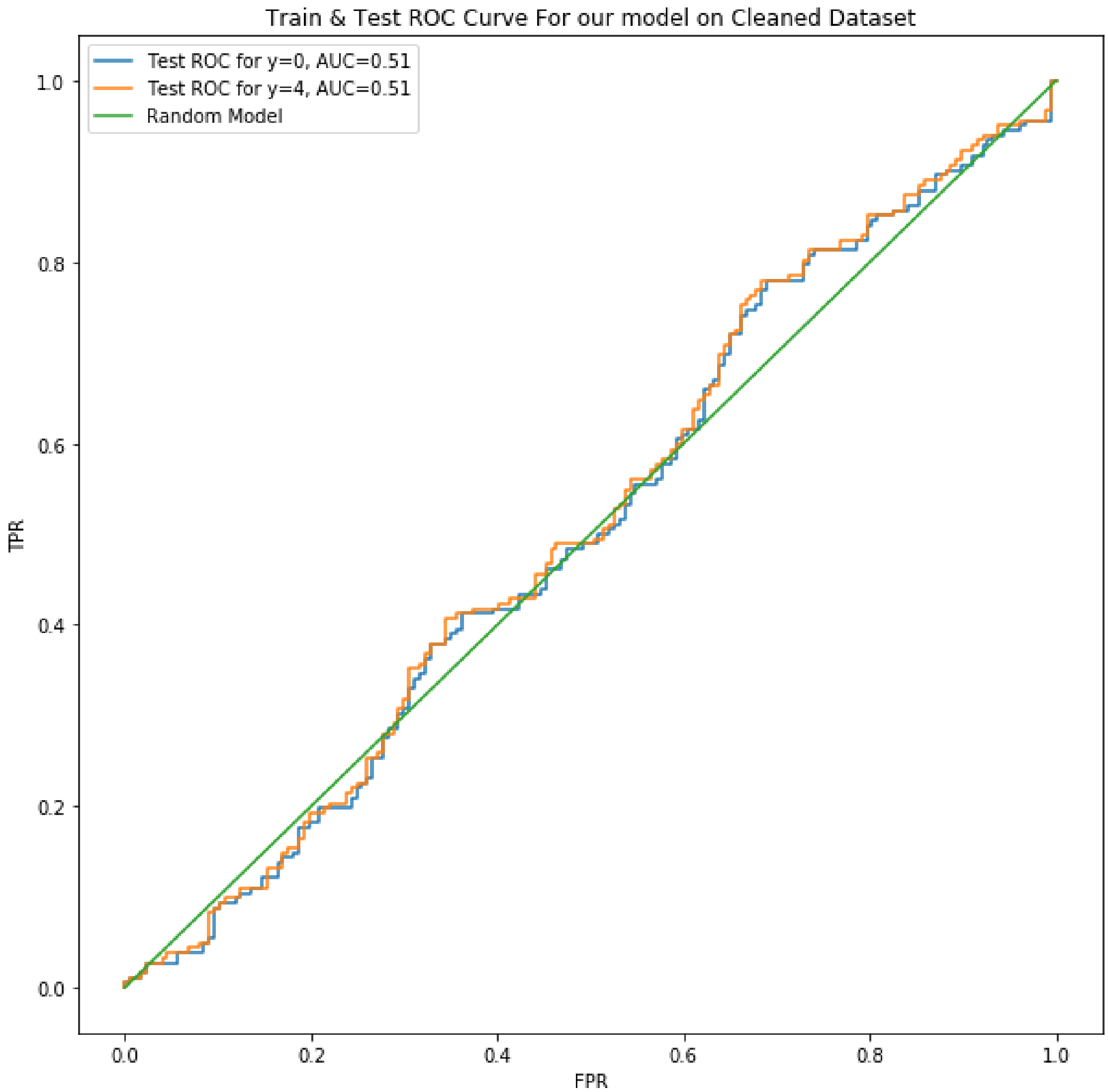


[51]:

```
test_proba = nb_cleaned.predict_log_proba(X_test)
test_proba0 = np.array(test_proba)[0,:]
test_proba4 = np.array(test_proba)[1,:]

plt.figure(figsize=(10,10))
fpr, tpr, thresholds = roc_curve(Y_test, test_proba0, pos_label=4)
auc=roc_auc_score(Y_test==4, test_proba0)
plt.plot(fpr, tpr, label='Test ROC for y=0, AUC=%.2f'%auc)

fpr, tpr, thresholds = roc_curve(Y_test, test_proba4, pos_label=4)
auc=roc_auc_score(Y_test==4, test_proba4)
plt.plot(fpr, tpr, label='Test ROC for y=4, AUC=%.2f'%auc)
```

```
plt.plot([0,1],[0,1], label='Random Model')
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('Train & Test ROC Curve For our model on Cleaned Dataset')
plt.legend()
plt.show()
```

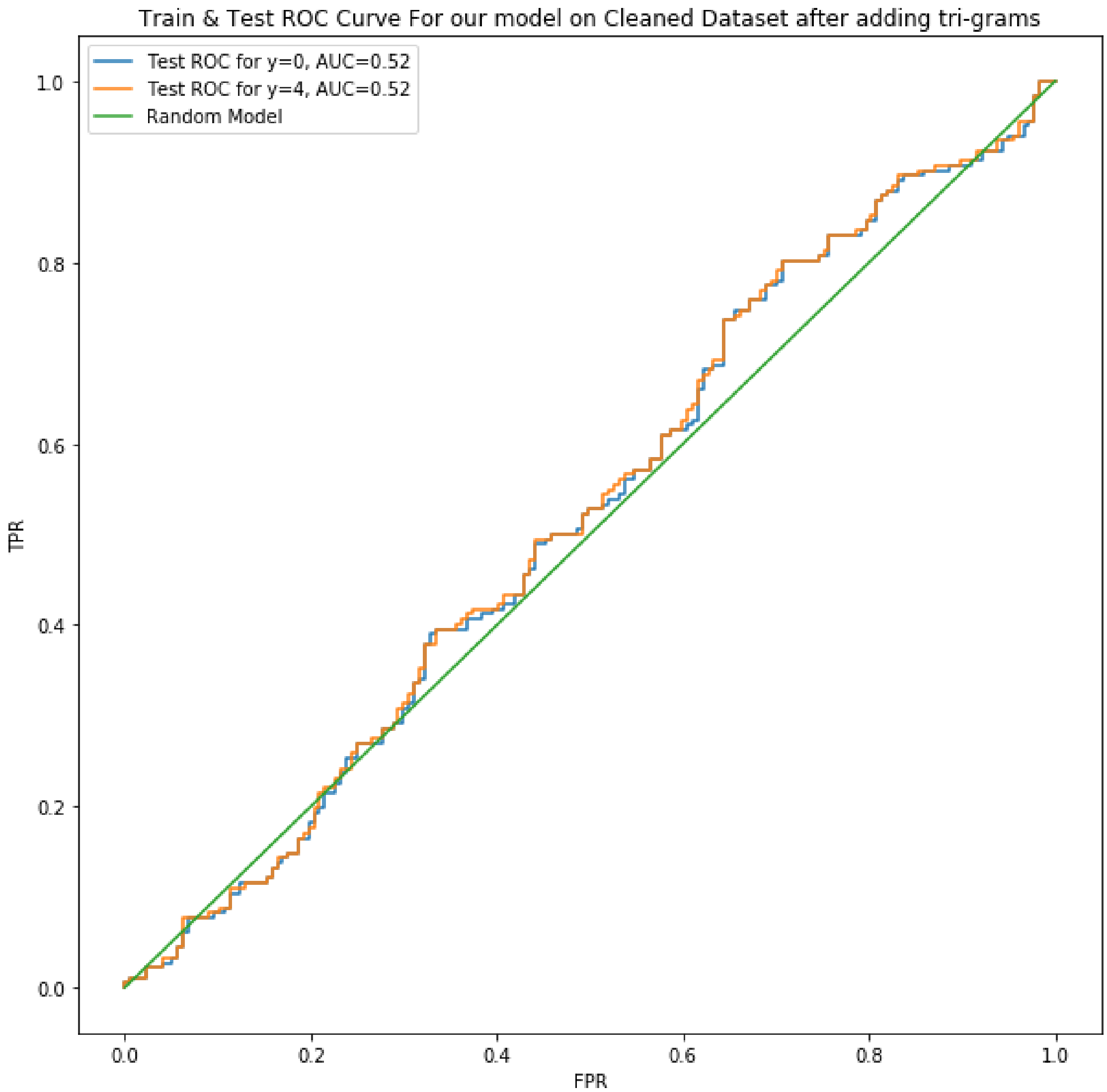
```
Generating Log probabilities: 100%|      | 359/359 [00:00<00:00,
12605.74it/s]
```



[70]:
```
test_proba = nb_tri.predict_log_proba(X_test)
test_proba0 = np.array(test_proba)[0,:]
test_proba4 = np.array(test_proba)[1,:]
```

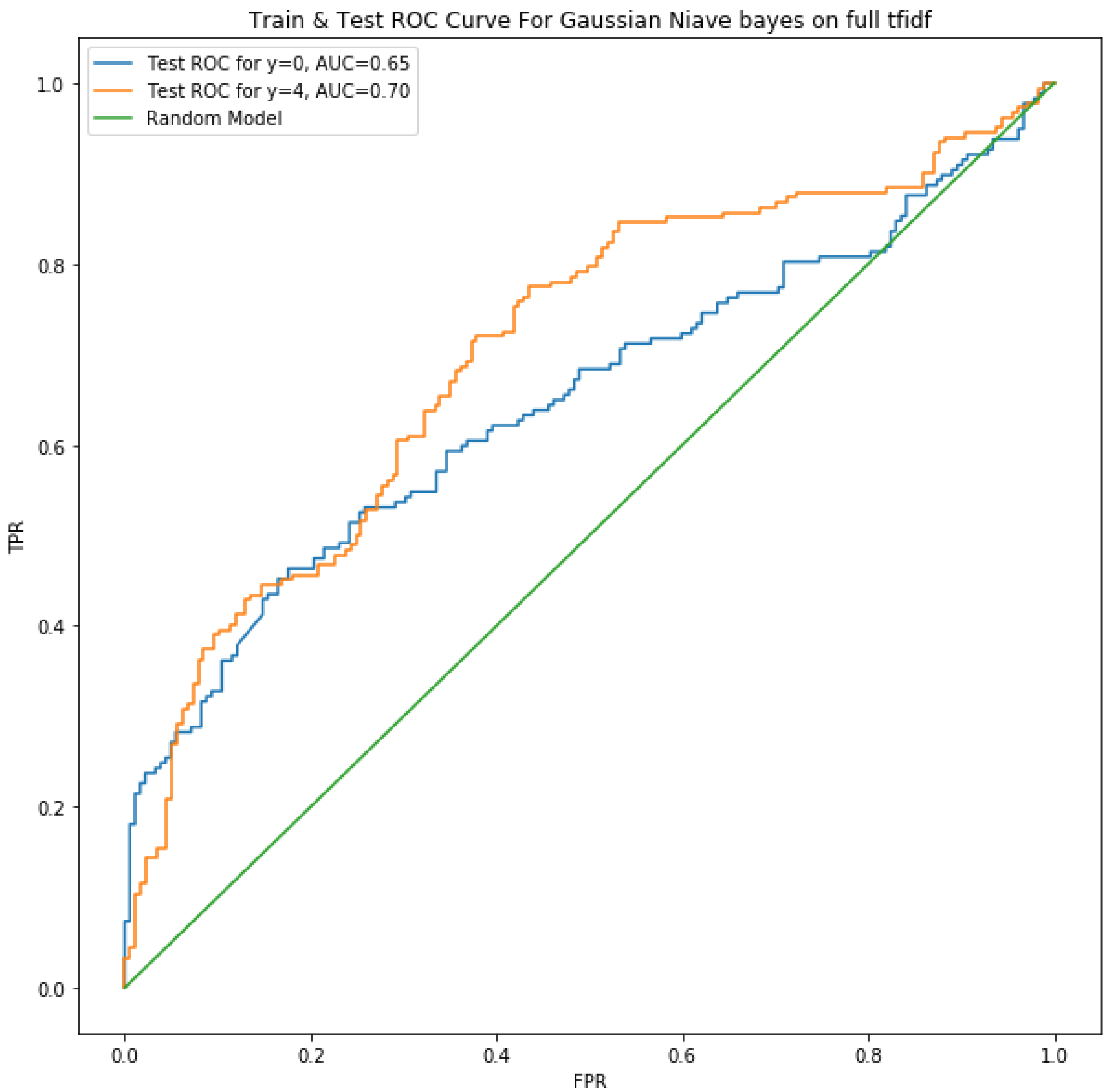
```
plt.figure(figsize=(10,10))
fpr, tpr, thresholds = roc_curve(Y_test, test_proba0, pos_label=4)
auc=roc_auc_score(Y_test==4, test_proba0)
plt.plot(fpr, tpr, label='Test ROC for y=0, AUC=%.2f'%auc)

fpr, tpr, thresholds = roc_curve(Y_test, test_proba4, pos_label=4)
auc=roc_auc_score(Y_test==4, test_proba4)
plt.plot(fpr, tpr, label='Test ROC for y=4, AUC=%.2f'%auc)

plt.plot([0,1],[0,1], label='Random Model')
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('Train & Test ROC Curve For our model on Cleaned Dataset after adding␣
↪tri-grams')
plt.legend()
plt.show()
```

```
Generating Log probabilities: 100%|      | 359/359 [00:00<00:00,
3992.66it/s]
```



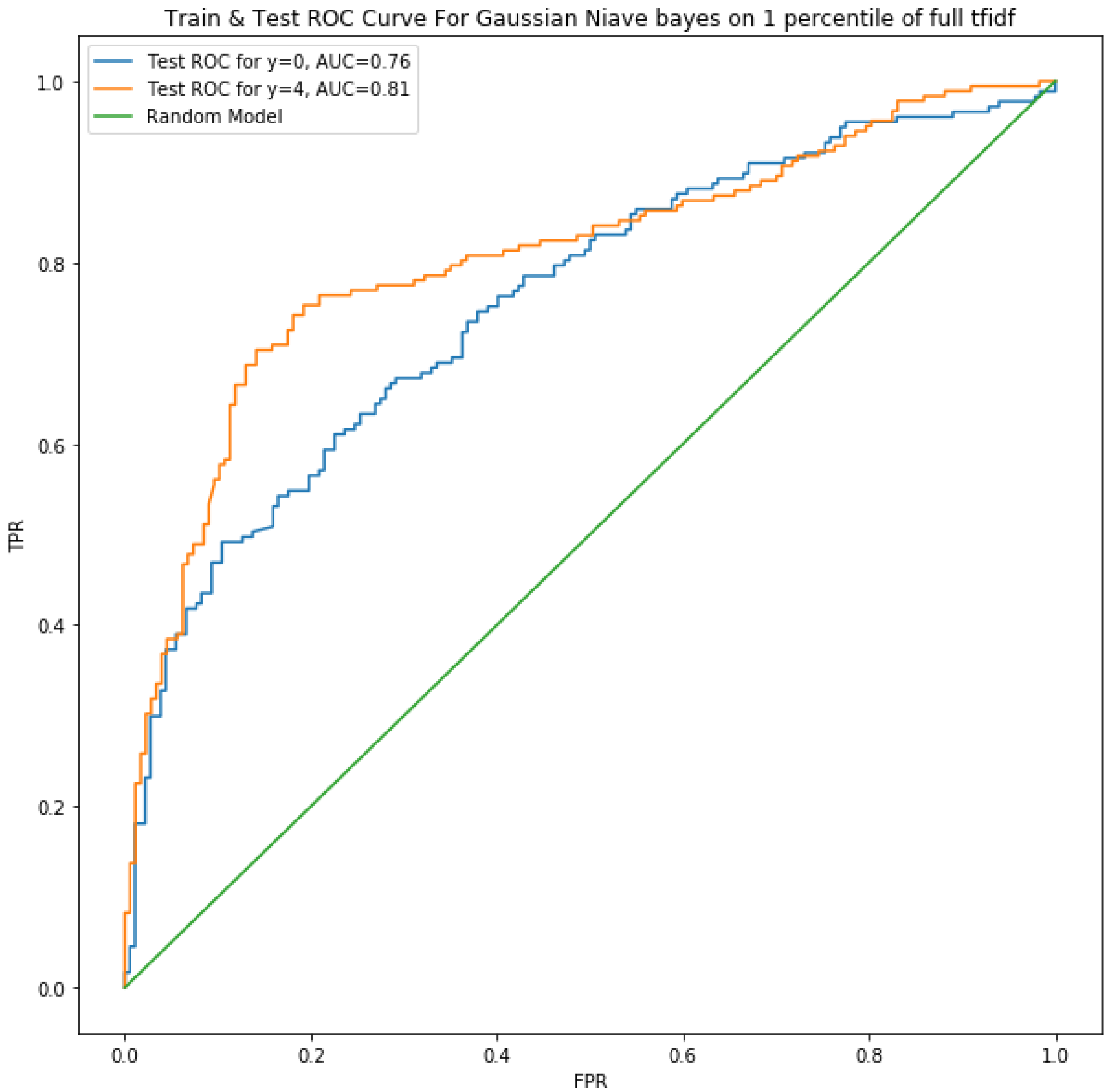
[71]:
```
test_proba = myGNB_full.predict_log_proba(X_test_tfidf)
test_proba0 = np.array(test_proba)[:,0]
test_proba4 = np.array(test_proba)[:,1]

plt.figure(figsize=(10,10))
fpr, tpr, thresholds = roc_curve(Y_test, test_proba0, pos_label=0)
auc=roc_auc_score(Y_test==0, test_proba0)
plt.plot(fpr, tpr, label='Test ROC for y=0, AUC=%.2f'%auc)

fpr, tpr, thresholds = roc_curve(Y_test, test_proba4, pos_label=4)
auc=roc_auc_score(Y_test==4, test_proba4)
plt.plot(fpr, tpr, label='Test ROC for y=4, AUC=%.2f'%auc)
```

```
plt.plot([0,1],[0,1], label='Random Model')
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('Train & Test ROC Curve For Gaussian Niave bayes on full tfidf')
plt.legend()
plt.show()
```



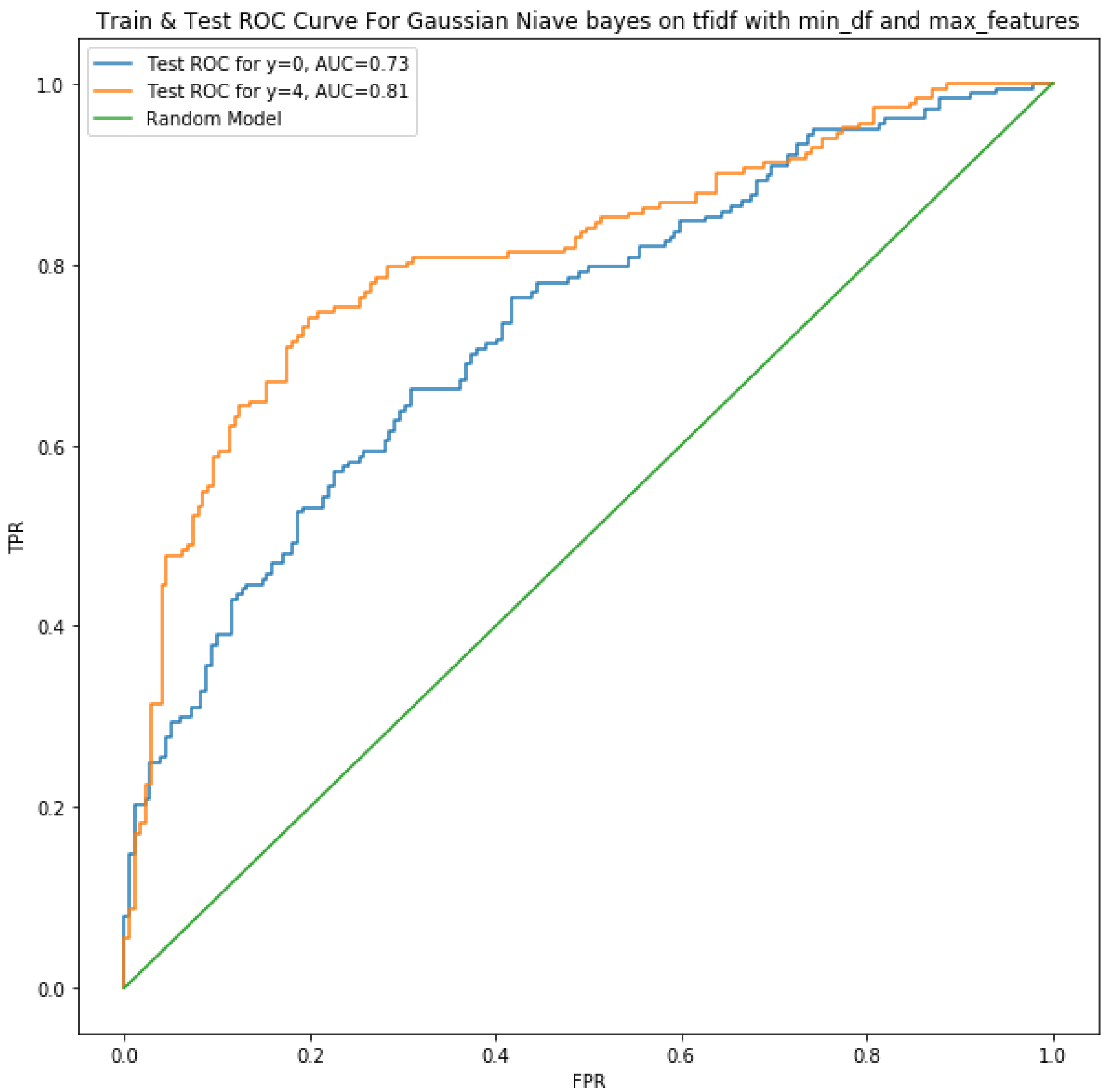
[72]:
```
per = SelectPercentile(f_classif, percentile=1).fit(X_train_tfidf, Y_train)

test_proba = myGNB_full_1.predict_log_proba(per.transform(X_test_tfidf))
test_proba0 = np.array(test_proba)[:,0]
test_proba4 = np.array(test_proba)[:,1]
```

```
plt.figure(figsize=(10,10))
fpr, tpr, thresholds = roc_curve(Y_test, test_proba0, pos_label=0)
auc=roc_auc_score(Y_test==0, test_proba0)
plt.plot(fpr, tpr, label='Test ROC for y=0, AUC=%.2f'%auc)

fpr, tpr, thresholds = roc_curve(Y_test, test_proba4, pos_label=4)
auc=roc_auc_score(Y_test==4, test_proba4)
plt.plot(fpr, tpr, label='Test ROC for y=4, AUC=%.2f'%auc)

plt.plot([0,1],[0,1], label='Random Model')
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('Train & Test ROC Curve For Gaussian Niave bayes on 1 percentile of␣
 ↪full tfidf')
plt.legend()
plt.show()
```



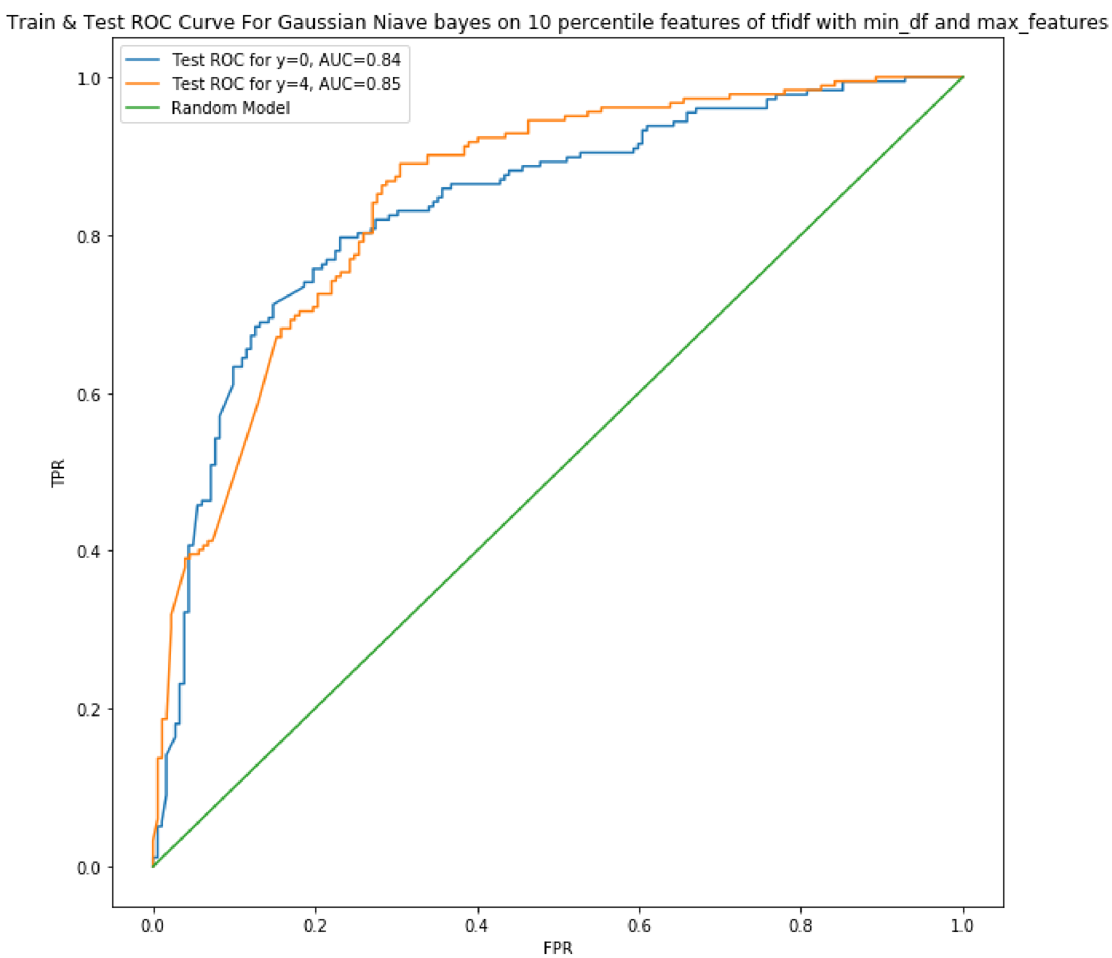
[73]:
```
test_proba = myGNB_mindf.predict_log_proba(X_test_tfidf_mindf)
test_proba0 = np.array(test_proba)[:,0]
test_proba4 = np.array(test_proba)[:,1]

plt.figure(figsize=(10,10))
fpr, tpr, thresholds = roc_curve(Y_test, test_proba0, pos_label=0)
auc=roc_auc_score(Y_test==0, test_proba0)
plt.plot(fpr, tpr, label='Test ROC for y=0, AUC=%.2f'%auc)

fpr, tpr, thresholds = roc_curve(Y_test, test_proba4, pos_label=4)
auc=roc_auc_score(Y_test==4, test_proba4)
plt.plot(fpr, tpr, label='Test ROC for y=4, AUC=%.2f'%auc)
```

```
plt.plot([0,1],[0,1], label='Random Model')
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('Train & Test ROC Curve For Gaussian Niave bayes on tfidf with min_df␣
 ↪and max_features')
plt.legend()
plt.show()
```



[75]:
```
per = SelectPercentile(f_classif, percentile=10).fit(X_train_tfidf_mindf,␣
 ↪Y_train)

test_proba = myGNB_mindf_10.predict_log_proba(per.transform(X_test_tfidf_mindf))
test_proba0 = np.array(test_proba)[:,0]
```

```
test_proba4 = np.array(test_proba)[:,1]

plt.figure(figsize=(10,10))
fpr, tpr, thresholds = roc_curve(Y_test, test_proba0, pos_label=0)
auc=roc_auc_score(Y_test==0, test_proba0)
plt.plot(fpr, tpr, label='Test ROC for y=0, AUC=%.2f'%auc)

fpr, tpr, thresholds = roc_curve(Y_test, test_proba4, pos_label=4)
auc=roc_auc_score(Y_test==4, test_proba4)
plt.plot(fpr, tpr, label='Test ROC for y=4, AUC=%.2f'%auc)

plt.plot([0,1],[0,1], label='Random Model')
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('Train & Test ROC Curve For Gaussian Niave bayes on 10 percentile␣
 ↪features of tfidf with min_df and max_features')
plt.legend()
plt.show()
```



- **Observations :**
  - Here we can see that AUC on our original dataset and cleaned dataset is very close to 0.5 hence we can say that our model did performed well in terms of accuracy but it was not confident in detecting classes.
  - We got better ROC curve for GaussianNB on tfidf data.

Assignment 2 Part B

March 9, 2020

1 Assignment 2 (Part b) : SVMs

1.1 Qeustion 1 : Binary Classification

- In this Part I have implmented SVM using CVXOPT module and used qp solvers of that modules which solves following optimization problem

$$\min_x \frac{1}{2}x^T Px + q^T x \text{ s.t.} Gx \preceq h \text{ and } \quad Ax = b$$

- Here in soft margin SVM our optimization problem is:

$$\max_{\alpha_i} \sum_m \alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} \alpha_i\alpha_j y_i y_j \ K < x_i, x_j > \quad = \min_{\alpha_i} \frac{1}{2}\sum_{i,j=1}^{m} \alpha_i\alpha_j y_i y_j \ K < x_i, x_j > - \sum_m \alpha_i$$

- Here in $K < x_i, x_j >$ is kernel value between x_i and x_j. I have create kernel matrix of size mxm for training dataset in whihc $(i,j)^{th}$ entry will represent kernel value between i^{th} and j^{th} datapoint. Therefore for linear kernel $K = XX^T$ and for gaussain kernel we need to find $K < x_i, x_j >= \exp(-\gamma * ||x_i - x_j||^2)$
- Now finding P,q,G,h,A and b for our optimization problem that we can pass in the qp solver of CVXOPT.

$$x^T = [\alpha_1, \alpha_2, ..., \alpha_m]_m$$

$$P = YY^T K, \text{where} K = XX^T \text{in case of linear kernel}$$

$$q^T = [-1, -1, -1... - 1]_{1\times m}$$

$$G = \begin{bmatrix} -1 & & & & & \\ & -1 & & & & \\ & & \ddots & & & \\ & & & \ddots & & \\ & & & & \ddots & \\ & & & & & -1 \\ 1 & & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & \ddots & & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix}_{2m \times m}$$

$$h = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ C \\ C \\ \vdots \\ C \end{bmatrix}_{2m \times 1}$$

$$A = Y^T$$

$$b = [0]$$

- So now using this optimization problem solvers we have implemnted binary classification model in Question 1 and multiclass classification model using one-vs-one stretergy in next question.
- All the experiments in this assignment were ran in google colab with 35 GB Ram and 40 core machine.

[0]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
from tqdm import tqdm
from time import time
import math
from cvxopt import solvers, matrix

from scipy.spatial.distance import cdist
```

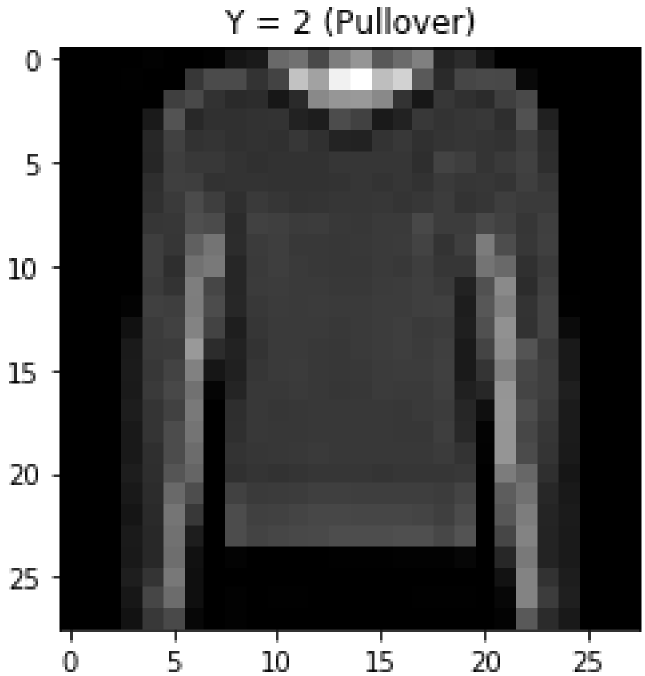
```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

from itertools import combinations

from joblib import Parallel, delayed
import multiprocessing
```

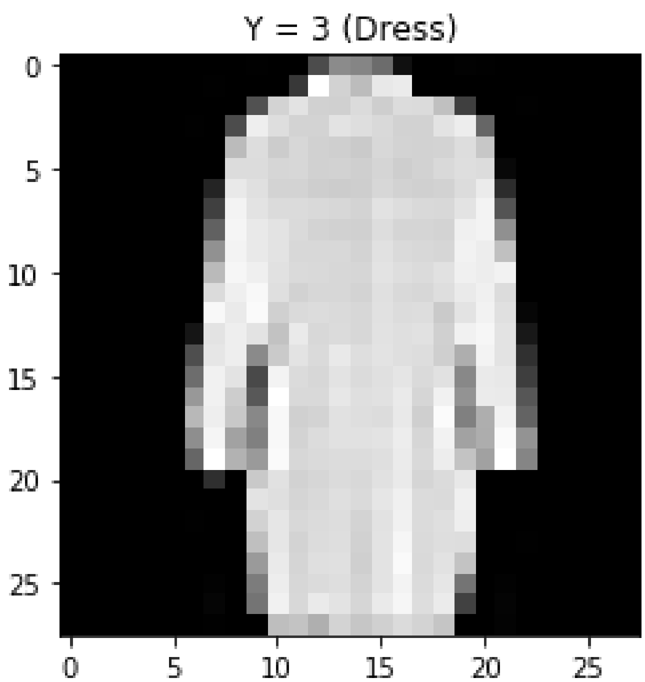
- Loading the data for class 2 and 3 as my entry number is 2019mcs2562

[0]:
```
def getData(train, test, val, class1=2, class2=3):
    '''
    This method returns X and Y of Train, Test and validation dataset having␣
↪output labels= class1 or class2
    '''
    #Slicing data where y=class1 or class2
    train = train[np.logical_or(train[:,-1]==class1, train[:,-1]==class2)]
    val = val[np.logical_or(val[:,-1]==class1, val[:,-1]==class2)]
    test = test[np.logical_or(test[:,-1]==class1, test[:,-1]==class2)]

    #Scaling X and mapping class1 to -1 and class2 to 1 to return Yi
    X_train = train[:,:-1]/255
    Y_train = train[:,-1]
    Y_train = np.array([-1 if i==class1 else 1 for i in Y_train], dtype=int).
↪reshape((Y_train.shape[0],1))

    X_val = val[:,:-1]/255
    Y_val = val[:,-1]
    Y_val = np.array([-1 if i==class1 else 1 for i in Y_val], dtype=int).
↪reshape((Y_val.shape[0],1))

    X_test = test[:,:-1]/255
    Y_test = test[:,-1]
    Y_test = np.array([-1 if i==class1 else 1 for i in Y_test], dtype=int).
↪reshape((Y_test.shape[0],1))

    return (X_train, Y_train, X_val, Y_val, X_test, Y_test)

#Loading Data-sets
# train = pd.read_csv('/content/drive/My Drive/ML/Assignment2Data/train.csv',␣
↪header=None).to_numpy()
# val = pd.read_csv('/content/drive/My Drive/ML/Assignment2Data/val.csv',␣
↪header=None).to_numpy()
# test = pd.read_csv('/content/drive/My Drive/ML/Assignment2Data/test.csv',␣
↪header=None).to_numpy()
train = pd.read_csv('./data/train.csv', header=None).to_numpy()
```

```
val = pd.read_csv('./data/val.csv', header=None).to_numpy()
test = pd.read_csv('./data/test.csv', header=None).to_numpy()



#Getting train, validation and test dataset for class labels 2 and 3
(X_train, Y_train, X_val, Y_val, X_test, Y_test) = getData(train, test, val,␣
↪class1=2, class2=3)

#Getting whole dataset that will be used in question 2(X's are scaled here)
X_train_whole = train[:,:-1]/255
Y_train_whole = train[:,-1].ravel()
Y_train_whole = Y_train_whole.astype(int)

X_val_whole = val[:,:-1]/255
Y_val_whole = val[:,-1].ravel()
Y_val_whole = Y_val_whole.astype(int)

X_test_whole = test[:,:-1]/255
Y_test_whole = test[:,-1].ravel()
Y_test_whole = Y_test_whole.astype(int)
```

[4]:
```
#Showing class = 2(Pullover) and class = 3(Dress) images
plt.imshow(X_train[np.where(Y_train.ravel() == -1)][0].reshape((28,28)),␣
↪cmap='gray')
plt.title('Y = 2 (Pullover)')
plt.show()
plt.imshow(X_train[np.where(Y_train.ravel() == 1)][0].reshape((28,28)),␣
↪cmap='gray')
plt.title('Y = 3 (Dress)')
plt.show()
```


Y = 2 (Pullover)
0
5
10
15
20
25
0
5
10
15
20
25


Y = 3 (Dress)
0
5
10
15
20
25
0
5
10
15
20
25

1.2 Question 1(a) : Implementing Linear SVM

[0]:
```
class SVM:
    '''
    This class provides implementation of SVM.

    **************
    Parameters:
    C=1 : Regularization parameter (Defalts 1.0)
    threshold=1e-5 : Threshold for alphas i.e. alpha values below this will be␣
↪treated as 0 (Defalts 1e-5)
    kernel='kernel' : Type of kernel you want to use ('linear' or 'rbf')
    gamma=0.05 : Hyperparameter for rbf kernel
    showTime=True : Boolean to show time taken to fit
    silentCVXOPT=False : Supress the output that CVXopt generates while solving␣
↪optimization problem
    **************

    **************
    Attributes:

    alphas : array of size m
    nSV : Number of support vectors
    SV_indices : indices of support vectors in alphas and in X_train(size=nSV)
    SV : support vectors (size=(m,n))
    SV_y : output label corrosponding to every support vectors (size=m)


    **************
    '''
    __slots__=['C', 'threshold', 'kernel', 'gamma', 'showTime', 'silentCVXOPT',␣
↪'alphas', 'nSV', 'SV_indices', 'SV_y', 'SV', 'w', 'b']

    def __init__(self, C=1, threshold=1e-5, kernel='linear', gamma=0.05,␣
↪showTime=True, silentCVXOPT = False):
        self.C = C
        self.threshold = threshold
        self.kernel = kernel
        self.gamma = gamma
        self.showTime = showTime
        self.silentCVXOPT = silentCVXOPT

    def _getRBFKernelMat(self, X, gamma = 0.05):
        '''Returns rbf kernel matrix for the given data X'''
        return np.exp(-gamma*(cdist(X,X,'euclidean')**2))

    def fit(self, X_train, Y_train):
```

```
        '''
        This method trains SVM model and stores corrosponding parameters

        Important : Y should be mapped to -1 aand 1 only
        '''
        t0 = time()

        if self.kernel=='linear':
            P = Y_train.dot(Y_train.T) * (X_train.dot(X_train.T))
        elif self.kernel=='rbf':
            K = self._getRBFKernelMat(X_train, self.gamma)
            P = Y_train.dot(Y_train.T) * (K)
        P=matrix(P,tc='d')

        q = -np.ones(X_train.shape[0])
        q=matrix(q,tc='d')

        G = np.vstack((-np.identity(X_train.shape[0]), np.identity(X_train.
↪shape[0])))
        G=matrix(G,tc='d')

        h = np.hstack((np.zeros(X_train.shape[0]), np.zeros(X_train.
↪shape[0])+self.C))
        h=matrix(h,tc='d')

        A = Y_train.T
        A = matrix(A,tc='d')
        b=matrix([0],tc='d')
        if self.silentCVXOPT:
            solvers.options['show_progress'] = False
        opt = solvers.qp(P, q, G, h, A, b)

        self.alphas = np.array(opt['x']).ravel()

        self.alphas[self.alphas<self.threshold] = 0

        self.nSV = np.where(self.alphas > 0)[0].shape[0]
        self.SV_indices = np.where(self.alphas > 0)[0]
        self.SV_y = Y_train[self.SV_indices].ravel()
        self.SV = X_train[self.SV_indices]

        if self.kernel == 'linear':
            self.w = (X_train[self.SV_indices]).T.dot((Y_train[self.SV_indices].
↪ravel())*self.alphas[self.SV_indices])
            X_pos = X_train[np.where(Y_train.ravel() == 1)]
            X_neg = X_train[np.where(Y_train.ravel() == -1)]
            self.b = -(1/2)*(min(X_pos.dot(self.w)) + max(X_neg.dot(self.w)))
```

```
        elif self.kernel == 'rbf':
            X_pos_indices = np.where(Y_train.ravel() == 1)[0]
            X_neg_indices = np.where(Y_train.ravel() == -1)[0]
            alphaiXyi = self.alphas*Y_train.ravel()
            Min = math.inf
            for i in X_pos_indices:
                Sum = 0
                for j in self.SV_indices:
                    Sum = Sum + (alphaiXyi[j] * K[i,j])
                if(Sum<Min):
                    Min=Sum
            Max = -math.inf
            for i in X_neg_indices:
                Sum = 0
                for j in self.SV_indices:
                    Sum = Sum + (alphaiXyi[j] * K[i,j])
                if(Sum>Max):
                    Max=Sum
            self.b = -(1/2)*(Max+Min)
        if self.showTime:
            print("Training completed in %.2f seconds"%((time()-t0)))


    def predict(self,X):
        '''
        This method returns predictions generated by our model for the given␣
↪dataset
        '''
        pred = list()
        if self.kernel=='linear':
            for pt in X:
                if (self.SV).T.dot(self.SV_y*self.alphas[self.SV_indices]).
↪dot(pt) + self.b > 0:
                    pred.append(1)
                else:
                    pred.append(-1)
        elif self.kernel=='rbf':
            wTxB = (np.exp(-0.05*cdist(self.SV, X)**2).T)@(self.alphas[self.
↪SV_indices]*self.SV_y).reshape((-1,1)) + self.b
            wTxB[wTxB > 0] = 1
            wTxB[wTxB < 0] = -1
            pred = wTxB
        return np.array(pred, dtype='int32')

    def decision_function(self, X):
        '''
```

```
        returns wTx + b for every points in X
        '''
        score = list()
        if self.kernel=='linear':
            for pt in X:
                score.append((self.SV).T.dot(self.SV_y*self.alphas[self.
 ↪SV_indices]).dot(pt) + self.b)
        elif self.kernel=='rbf':
            score = (np.exp(-0.05*cdist(self.SV, X)**2).T)@(self.alphas[self.
 ↪SV_indices]*self.SV_y).reshape((-1,1)) + self.b
        return np.array(score)

class metrics:
    '''
    This class contains various methods that will be useful in finding various␣
 ↪performance metrics for our model
    '''
    def accuracy_score(y_true, y_pred):
        '''
        Returs accuracy for the given predicted and true values
        '''
        true = sum(y_true.ravel() == y_pred.ravel())
        return true*100/len(y_true)

    def confusion_matrix(y_true, y_pred, title = 'Confusion Matrix'):
        '''
        plots confusion matrix for given predicted and truue values
        '''
        classes = np.unique(y_true)
        n_classes=len(classes)
        cm=np.zeros((n_classes,n_classes), dtype=int)

        for i in range(len(y_true)):
            cm[y_pred[i],y_true[i]]+=1

        plt.figure(figsize=(10,10))
        sns.heatmap(cm, annot=True, fmt="d", cbar=False,␣
 ↪linecolor='black',linewidth='0', \
        cmap='Blues', xticklabels=classes, yticklabels=classes)
        plt.xlabel('Actual')
        plt.ylabel('Predicted')
        plt.title(title)
        plt.show()
```

[6]:
```
#Training linear SVM model with C=1

linearSVM = SVM(C=1.0, kernel='linear')
```

```
linearSVM.fit(X_train, Y_train)
```

```
     pcost       dcost       gap    pres   dres
 0: -5.2396e+02 -9.3983e+03  5e+04  3e+00  3e-12
 1: -3.3568e+02 -5.4793e+03  1e+04  5e-01  2e-12
 2: -2.0714e+02 -1.8273e+03  3e+03  1e-01  2e-12
 3: -1.4999e+02 -9.1602e+02  1e+03  5e-02  1e-12
 4: -1.1131e+02 -4.9405e+02  6e+02  2e-02  9e-13
 5: -9.0544e+01 -3.1115e+02  3e+02  8e-03  9e-13
 6: -8.5666e+01 -1.4689e+02  7e+01  6e-04  9e-13
 7: -9.2854e+01 -1.1845e+02  3e+01  6e-15  1e-12
 8: -9.8193e+01 -1.0801e+02  1e+01  3e-15  9e-13
 9: -1.0058e+02 -1.0337e+02  3e+00  4e-15  1e-12
10: -1.0167e+02 -1.0194e+02  3e-01  7e-16  1e-12
11: -1.0178e+02 -1.0179e+02  5e-03  4e-15  1e-12
12: -1.0179e+02 -1.0179e+02  9e-05  2e-15  1e-12
Optimal solution found.
Training completed in 14.43 seconds
```

[10]:
```
print('b: ', linearSVM.b)
```

```
b:  -1.235384679744142
```

[0]:
```
w = linearSVM.w
```

[12]:
```
print('Number of support vectors :',linearSVM.nSV)
```

```
Number of support vectors : 370
```

[13]:
```
#Generating accuracies figures for train, test and validation datasets

pred_train = linearSVM.predict(X_train)
accuracy_train = metrics.accuracy_score(Y_train, pred_train)

pred_test = linearSVM.predict(X_test)
accuracy_test = metrics.accuracy_score(Y_test, pred_test)

pred_val = linearSVM.predict(X_val)
accuracy_val = metrics.accuracy_score(Y_val, pred_val)

print('Accuracy on Train Dataset using linear kernel : ', accuracy_train)
print('Accuracy on Validation Dataset using linear kernel : ', accuracy_val)
print('Accuracy on Test Dataset using linear kernel : ', accuracy_test)
```

```
Accuracy on Train Dataset using linear kernel :  92.77777777777777
Accuracy on Validation Dataset using linear kernel :  89.6
Accuracy on Test Dataset using linear kernel :  91.3
```

[0]:
```
#Changing b to b that we get using sklearn as we know our formula of b is wrong
#(As sir mentioned in piazza to use formula of b as discussed in hard margin␣
 ↪SVM)
linearSVM.b=0.3995427
```

[15]:
```
#Getting updated accuracies after changing b in our class
#As we will later see that this values are very much close to accuracies␣
 ↪reported by sklearn's model
#Hence formula that we used to calculate b as sir told us is giving wrong b

pred_train = linearSVM.predict(X_train)
accuracy_train = metrics.accuracy_score(Y_train, pred_train)

pred_test = linearSVM.predict(X_test)
accuracy_test = metrics.accuracy_score(Y_test, pred_test)

pred_val = linearSVM.predict(X_val)
accuracy_val = metrics.accuracy_score(Y_val, pred_val)
print('****Accuracies after changing b*****')
print('Accuracy on Train Dataset using linear kernel : ', accuracy_train)
print('Accuracy on Validation Dataset using linear kernel : ', accuracy_val)
print('Accuracy on Test Dataset using linear kernel : ', accuracy_test)
```

```
****Accuracies after changing b*****
Accuracy on Train Dataset using linear kernel :  99.55555555555556
Accuracy on Validation Dataset using linear kernel :  97.0
Accuracy on Test Dataset using linear kernel :  95.7
```

1.3 Question 1(b) : Applying SVM with Gaussian Kernel

[16]:
```
#Training linear SVM model with C=1

rbfSVM = SVM(C=1.0, threshold=1e-5, kernel='rbf', gamma=0.05)

rbfSVM.fit(X_train, Y_train)
```

```
     pcost       dcost       gap    pres   dres
 0: -3.0404e+02 -8.2833e+03  4e+04  2e+00  1e-15
 1: -2.1981e+02 -4.4663e+03  7e+03  3e-01  1e-15
 2: -1.8274e+02 -1.0106e+03  1e+03  3e-02  2e-15
 3: -2.3561e+02 -4.7608e+02  3e+02  6e-03  1e-15
 4: -2.6244e+02 -3.3488e+02  8e+01  1e-03  1e-15
 5: -2.7389e+02 -2.9769e+02  2e+01  2e-04  1e-15
 6: -2.7877e+02 -2.8539e+02  7e+00  4e-05  1e-15
 7: -2.8041e+02 -2.8192e+02  2e+00  3e-06  1e-15
 8: -2.8091e+02 -2.8102e+02  1e-01  7e-08  1e-15
```

```
 9: -2.8095e+02 -2.8095e+02  2e-03  9e-10  1e-15
10: -2.8095e+02 -2.8095e+02  4e-05  1e-11  1e-15
Optimal solution found.
Training completed in 32.37 seconds
```

[17]:
```
print('b :', rbfSVM.b)
```

```
b : -0.534452252549684
```

[18]:
```
print('Number of support vectors in RBF-SVM:',rbfSVM.nSV)
```

```
Number of support vectors in RBF-SVM: 1123
```

[19]:
```
#Generating accuracies figures for train, test and validation datasets

pred_train = rbfSVM.predict(X_train)
accuracy_train = metrics.accuracy_score(Y_train, pred_train)

pred_test = rbfSVM.predict(X_test)
accuracy_test = metrics.accuracy_score(Y_test, pred_test)

pred_val = rbfSVM.predict(X_val)
accuracy_val = metrics.accuracy_score(Y_val, pred_val)

print('Accuracy on Train Dataset using gaussian kernel :', accuracy_train)
print('Accuracy on Validation Dataset using gaussian kernel :', accuracy_val)
print('Accuracy on Test Dataset using gaussian kernel :', accuracy_test)
```

```
Accuracy on Train Dataset using gaussian kernel : 99.4
Accuracy on Validation Dataset using gaussian kernel : 97.2
Accuracy on Test Dataset using gaussian kernel : 96.1
```

[0]:
```
#Changing b to b that we get using sklearn as we know our formula of b is wrong
#(As sir mentioned in piazza to use formula of b as discussed in hard margin␣
 ↪SVM)
rbfSVM.b=-0.19803938
```

[21]:
```
#Getting updated accuracies after changing b in our class
#As we will later see that this values are very much close to accuracies␣
 ↪reported by sklearn's model
#Hence formula that we used to calculate b as sir told us is giving wrong b

pred_train = rbfSVM.predict(X_train)
accuracy_train = metrics.accuracy_score(Y_train, pred_train)

pred_test = rbfSVM.predict(X_test)
accuracy_test = metrics.accuracy_score(Y_test, pred_test)
```

```
pred_val = rbfSVM.predict(X_val)
accuracy_val = metrics.accuracy_score(Y_val, pred_val)
print('****Accuracies after changing b*****')
print('Accuracy on Train Dataset using gaussian kernel :', accuracy_train)
print('Accuracy on Validation Dataset using gaussian kernel :', accuracy_val)
print('Accuracy on Test Dataset using gaussian kernel :', accuracy_test)
```

```
****Accuracies after changing b*****
Accuracy on Train Dataset using gaussian kernel : 99.4
Accuracy on Validation Dataset using gaussian kernel : 98.2
Accuracy on Test Dataset using gaussian kernel : 97.2
```

1.4 Question 1(c) : Comparing our implementation with scikit-learn's implementation

[24]:
```
svm = SVC(C=1, kernel='linear')
t0 = time()
svm.fit(X_train, Y_train.ravel())
print('Linear SVM trined in %.2f sec'%(time()-t0))

w_sklearn = svm.coef_
# print('W : %s'%(svm.coef_))
print('b : %s'%(svm.intercept_))
print('nSV : %s'%(svm.n_support_))
print('Total Support vectors : %s'%(svm.n_support_.sum()))

pred_train = svm.predict(X_train)
pred_val = svm.predict(X_val)
pred_test = svm.predict(X_test)

print('Accuracy on train data %.2f %c'%(accuracy_score(Y_train,␣
 ↪pred_train)*100,'%'))
print('Accuracy on validation data %.2f %c'%(accuracy_score(Y_val,␣
 ↪pred_val)*100,'%'))
print('Accuracy on test data %.2f %c'%(accuracy_score(Y_test,␣
 ↪pred_test)*100,'%'))
```

```
Linear SVM trined in 2.92 sec
b : [0.3995427]
nSV : [167 196]
Total Support vectors : 363
Accuracy on train data 99.56 %
Accuracy on validation data 97.00 %
Accuracy on test data 95.70 %
```

[28]:
```
diff_norm = np.linalg.norm(w-w_sklearn, ord=2)
print('2-norm of difference between w found by our solution vs sklearn\'s␣
↪solution : ', diff_norm)
```

```
2-norm of difference between w found by our solution vs sklearn's solution :
0.005978963328436608
```

[29]:
```
svm = SVC(C=1, kernel='rbf', gamma=0.05)
t0 = time()
svm.fit(X_train, Y_train.ravel())
print('RBF SVM trined in %.2f sec'%(time()-t0))

# print('W : %s'%(svm.coef_))
print('b : %s'%(svm.intercept_))
print('Support vectors : %s'%(svm.n_support_))
print('Total Support vectors : %s'%(svm.n_support_.sum()))

pred_train = svm.predict(X_train)
pred_val = svm.predict(X_val)
pred_test = svm.predict(X_test)

print('Accuracy on train data %.2f %c'%(accuracy_score(Y_train,␣
↪pred_train)*100,'%'))
print('Accuracy on validation data %.2f %c'%(accuracy_score(Y_val,␣
↪pred_val)*100,'%'))
print('Accuracy on test data %.2f %c'%(accuracy_score(Y_test,␣
↪pred_test)*100,'%'))
```

```
RBF SVM trined in 6.59 sec
b : [-0.19803938]
Support vectors : [565 537]
Total Support vectors : 1102
Accuracy on train data 99.40 %
Accuracy on validation data 98.20 %
Accuracy on test data 97.20 %
```

1.4.1 Observations :

Kernel	Implementation	nSV	b	Accuracy (Train)	Accuracy (Validation)	Accuracy(Test)	Time (In seconds)
Linear	My	370	-1.23538	92.78	89.6	91.3	14.43
Linear	Sklearn	363	0.399	99.56	97	95.70	2.92
Linear	My(After b=0.399542)	370	0.399	99.55555556	97	95.70	

Kernel	Implementation	nSV	b	Accuracy (Train)	Accuracy (Validation)	Accuracy(Test)	Time (In seconds)
Gaussian	My	1123	-0.53445	99.4	97.2	96.1	32.37
Gaussian	Sklearn	1102	-0.19803	99.4	98.2	97.2	6.59
Gaussian	My(After b=-0.19803)	1123	-0.19803	99.4	98.2	97.2	

- We can clearly see that **accuracies in rbf kernel SVM increases** as compared to linear SVM.
- Now we can see here that **time taken to train using CVXOPT is more** that sklearn's implemntation as sklearn uses solver that is optimized for svm only whereas CVXOPT is a general optimization problem solver.
- **Number of support vectors** in my and sklaern's implementation **are similar**.
- b and accuracy figures differes in both the implementations, but when we set b in our class to b that sklearn gives then my implementation gives identical accuracies to that we get in sklearn's implementation.
- Now comparing w of our implementation and sklearn's implementation for the case of linear SVM we got **2-norm of difference between two w = 0.00597896**

2 Question 2 : Multi-class Classification

2.1 Question 2(a) : Implemeting Multiclass classification using binary model class (svm) that I created in question 1 using one-vs-one stretergy.

- In this Part I have implemented one-vs-one stretergy for classyfying 10 classes of fashion MNIST dataset

[0]:
```
class multiclassSVM:
    '''
    This class provides implementation of oneVsOne stretergy using rbfSVM with␣
↪gamma=0.05

    **************
    Parameters:
    C=1 : Regularization parameter for base learners (Defalts 1.0)
    gamma=0.05 : Hyperparameter for rbf kernel
    useSklearn=False : Flag that represents that you want to use SVM(My class)␣
↪or SVC
    classes=range(10) : list/iterable of possible output labels
    n_jobs=1 : parallelization parameter. Signifies number of worker threads.
```

```
               (-1: all cores -2: All except one core)
    showTime=True : Boolean to show time taken to fit
    **************

    **************
    Attributes:
    models : (n_classes*(n_classes-1)/2) base models list

    **************
    '''

    def __init__(self,C=1.0, gamma=0.05, useSklearn = False, classes =␣
↪range(10), n_jobs=1, showTime=True):
        self.C = C
        self.gamma = 0.05
        self.useSklearn = useSklearn
        self.classes = classes
        self.comb = list(combinations(classes,2))
        self.n_jobs = n_jobs
        self.showTime = showTime

    def _getData(self, X, Y, class1, class2):
        '''
        Return X and Y's corrosponding to class labesl class1 and class2
        '''
        X = X[np.logical_or(Y==class1, Y==class2)]
        Y = Y[np.logical_or(Y==class1, Y==class2)]

        X = X
        Y = np.array([-1 if i==class1 else 1 for i in Y]).reshape((Y.
↪shape[0],1))

        return (X, Y)

    def _getMyModel(self, c1, c2, X, Y):
        '''
        Helper method that will return one my SVM(kernel='rbf') class object␣
↪trined on X and Y

        Will be used to parallelize the training
        '''
        (X_train, Y_train) = self._getData(X, Y, class1=c1, class2=c2)
        model = SVM(C=self.C, kernel='rbf', gamma=self.gamma)
        model.fit(X_train, Y_train)
        return model

    def _getSklearnModel(self, c1, c2, X, Y):
```

```
        '''
        Helper method that will return one sklearn's SVC(kernel='rbf') class␣
↪object trined on X and Y

        Will be used to parallelize the training
        '''
        (X_train, Y_train) = self._getData(X, Y, class1=c1, class2=c2)
        model = SVC(C=self.C, kernel='rbf', gamma=self.gamma)
        model.fit(X_train, Y_train.ravel())
        return model

    def fit(self, X_train, Y_train):
        '''
        This method parallaly trains (n_classes*(n_classes-1)/2) base models
        '''
        self.models = list()
        t0=time()
        if self.useSklearn and self.n_jobs!=1:
            self.models = Parallel(n_jobs=self.n_jobs)(delayed(self.
↪_getSklearnModel)(c1, c2, X_train, Y_train)\
                                                    for (c1,c2) in tqdm(self.
↪comb, desc='OneVsOne Models generation'))
        elif self.useSklearn and self.n_jobs==1:
            for (c1,c2) in tqdm(self.comb):
                self.models.append(self._getSklearnModel(c1,c2, X_train,␣
↪Y_train))
        elif not self.useSklearn and self.n_jobs!=1:
            self.models = Parallel(n_jobs=self.n_jobs)(delayed(self.
↪_getMyModel)(c1, c2, X_train, Y_train)\
                                                    for (c1,c2) in tqdm(self.
↪comb, desc='OneVsOne Models generation'))
        elif not self.useSklearn and self.n_jobs==1:
            for (c1,c2) in tqdm(self.comb):
                self.models.append(self._getMyModel(c1,c2, X_train, Y_train))
        if self.showTime:
            print('Completed training in %.2f minutes.'%((time()-t0)/60))

    def _predict_for_one_model(self, X, model, c0, c1):
        '''
        Helper method that will return prediction on one base model given X and␣
↪a model

        This same method will work for both types of base models
        Will be used to parallelize the prediction
        '''
        pred = model.predict(X)
```

```
        pred_score = model.decision_function(X)
        pred[pred == 1] = c1
        pred[pred == -1] = c0
        return (pred, pred_score)

    def predict_multiclass(self, X):
        '''
        This method parallaly predicts the class labels for the given dataset
        '''
        if self.n_jobs != 1:
            results = Parallel(n_jobs=self.n_jobs)(delayed(self.
↪_predict_for_one_model)(X, model, c1, c2)\
                                                    for (model, (c1,c2)) in␣
↪tqdm(list(zip(self.models, self.comb)), desc='Generating Predictions'))
        else:
            results = [self._predict_for_one_model(X, model, c1, c2)\
                           for (model, (c1,c2)) in tqdm(list(zip(self.models, self.
↪comb)), desc='Generating Predictions')]

        predScoreLst = (list(zip(*results)))
        predictions = np.array(list(predScoreLst[0]))
        predictions = predictions.reshape((predictions.shape[0], predictions.
↪shape[1]))
        scores = np.array(list(predScoreLst[1]))
        scores = scores.reshape((scores.shape[0], scores.shape[1]))

        counts=list()
#       wo_tie = list() #Predctions without tie breaking
        for i in (predictions.T):
            lst = np.zeros(10, dtype='int32')
            for x in i:
                lst[x]+=1
#           wo_tie.append(np.argmax(lst)) #Predctions without tie breaking
            counts.append(lst)
        np.array(counts).shape

        scores_lst = list()
        for j in (range(len(predictions.T))):
            pt_score = list()
            for i in range(10):
                pt_score.append(sum(np.abs(scores.T[j][predictions.T[j] == i])))
            scores_lst.append(pt_score)
        np.array(scores_lst).shape

        pred = list()
        for i in range(len(X)):
            scores_i = scores_lst[i]
```

```
            counts_i = counts[i]
            out = 0
            maxScore = abs(scores_i[0])
            maxCount = counts_i[0]
            for j in range(1,10):
                if counts_i[j] < maxCount:
                    continue
                elif counts_i[j] > maxCount:
                    out = j
                    maxScore = abs(scores_i[j])
                    maxCount = counts_i[j]
                else:
                    if abs(scores_i[j]) > maxScore:
                        out = j
                        maxScore = abs(scores_i[j])
                        maxCount = counts_i[j]
                    else:
                        continue
            pred.append(out)
        return np.array(pred, dtype=int)
```

2.2 Part (a) : Implementing OneVsOne Stretergy for multiclass classification using SVM

[0]:
```
'''
Training Multiclass SVM model with my SVM class as base learner
'''

import os
ovo_my_implementation = multiclassSVM(n_jobs=-2, useSklearn=False)

if not os.path.isfile('./pickle/my_models_multi.pkl'):
    ovo_my_implementation.fit(X_train_whole, Y_train_whole)
    with open('./pickle/my_models_multi.pkl', 'wb') as f:
        pickle.dump(ovo_my_implementation, f)
else:
    with open('./pickle/my_models_multi.pkl', 'rb') as f:
        pickled_class = pickle.load(f)
    ovo_my_implementation.models = pickled_class.models

#output while trained

# OneVsOne Models generation: 100%|      | 45/45 [04:22<00:00,  5.84s/it]
# Completed training in 7.10 minutes.
```

[0]:

```python
'''
Getting accuracies for Multiclass SVM model with my SVM class as base learner
'''

t0 = time()
pred_test_my = ovo_my_implementation.predict_multiclass(X_test_whole)
print('Predicted for test dataset in %.2f minutes. '%((time()-t0)/60))

accuracy_test = metrics.accuracy_score(Y_test_whole.ravel(), pred_test_my)
print('Accuracy on test dataset : %.2f \n\n'%(accuracy_test))

t0 = time()
pred_val_my = ovo_my_implementation.predict_multiclass(X_val_whole)
print('Predicted for validation dataset in in %.2f minutes.'%((time()-t0)/60))

accuracy_val = metrics.accuracy_score(Y_val_whole, pred_val_my)
print('Accuracy on validation dataset : %.2f \n\n'%(accuracy_val))

t0 = time()
pred_train_my = ovo_my_implementation.predict_multiclass(X_train_whole)
print('Predicted for train dataset in %.2f minutes.'%((time()-t0)/60))

accuracy_train = metrics.accuracy_score(Y_train_whole, pred_train_my)
print('Accuracy on train dataset : %.2f \n\n'%(accuracy_train))
```

```
Generating Predictions: 100%|      | 45/45 [06:52<00:00,  9.53s/it]
Generating Predictions:   0%|          | 0/45 [00:00<?, ?it/s]

Predicted for test dataset in 6.89 minutes.
Accuracy on test dataset : 85.08



Generating Predictions: 100%|      | 45/45 [02:55<00:00,  4.09s/it]
Generating Predictions:   0%|          | 0/45 [00:00<?, ?it/s]

Predicted for validation dataset in in 2.94 minutes.
Accuracy on validation dataset : 84.96



Generating Predictions: 100%|      | 45/45 [32:05<00:00, 44.00s/it]

Predicted for train dataset in 32.17 minutes.
Accuracy on train dataset : 96.52
```

2.3 Part (b) : Implementing OneVsOne Stretergy for multiclass classification using sklearn's implementation of SVM i.e. SVC

[0]:
```
'''
Training Multiclass SVM model with sklearn's SVM class as base learner
'''

ovo_sklearn = multiclassSVM(n_jobs=-2, useSklearn=True)

if not os.path.isfile('./pickle/sklearn_models_multi.pkl'):
    ovo_sklearn.fit(X_train_whole, Y_train_whole)
    with open('./pickle/sklearn_models_multi.pkl', 'wb') as f:
        pickle.dump(ovo_sklearn, f)
else:
    with open('./pickle/sklearn_models_multi.pkl', 'rb') as f:
        models = pickle.load(f)
    ovo_sklearn.models = models
```

```
OneVsOne Models generation: 100%|      | 45/45 [00:00<00:00, 64.28it/s]

Completed training in 0.46 minutes.
```

[0]:
```
'''
Getting Acuuracies Multiclass SVM model with sklearn's SVM class as base␣
↪learner
'''

t0 = time()
pred_test_sk = ovo_sklearn.predict_multiclass(X_test_whole)
print('Predicted for test dataset in %.2f minutes. '%((time()-t0)/60))

accuracy_test_sk = metrics.accuracy_score(Y_test_whole.ravel(), pred_test_sk)
print('Accuracy on test dataset : %.2f \n\n'%(accuracy_test_sk))

t0 = time()
pred_val_sk = ovo_sklearn.predict_multiclass(X_val_whole)
print('Predicted for validation dataset in in %.2f minutes.'%((time()-t0)/60))

accuracy_val_sk = metrics.accuracy_score(Y_val_whole, pred_val_sk)
print('Accuracy on validation dataset : %.2f \n\n'%(accuracy_val_sk))

t0 = time()
pred_train_sk = ovo_sklearn.predict_multiclass(X_train_whole)
print('Predicted for train dataset in %.2f minutes.'%((time()-t0)/60))

accuracy_train_sk = metrics.accuracy_score(Y_train_whole, pred_train_sk)
print('Accuracy on train dataset : %.2f \n\n'%(accuracy_train_sk))
```

```
Generating Predictions: 100%|     | 45/45 [00:00<00:00, 13822.31it/s]
Generating Predictions: 100%|     | 45/45 [00:00<00:00, 2283.04it/s]

Predicted for test dataset in 0.88 minutes.
Accuracy on test dataset : 88.08
```

```
Generating Predictions: 100%|     | 45/45 [00:00<00:00, 1713.69it/s]

Predicted for validation dataset in in 0.42 minutes.
Accuracy on validation dataset : 87.88
```

```
Predicted for train dataset in 3.84 minutes.
Accuracy on train dataset : 96.91
```

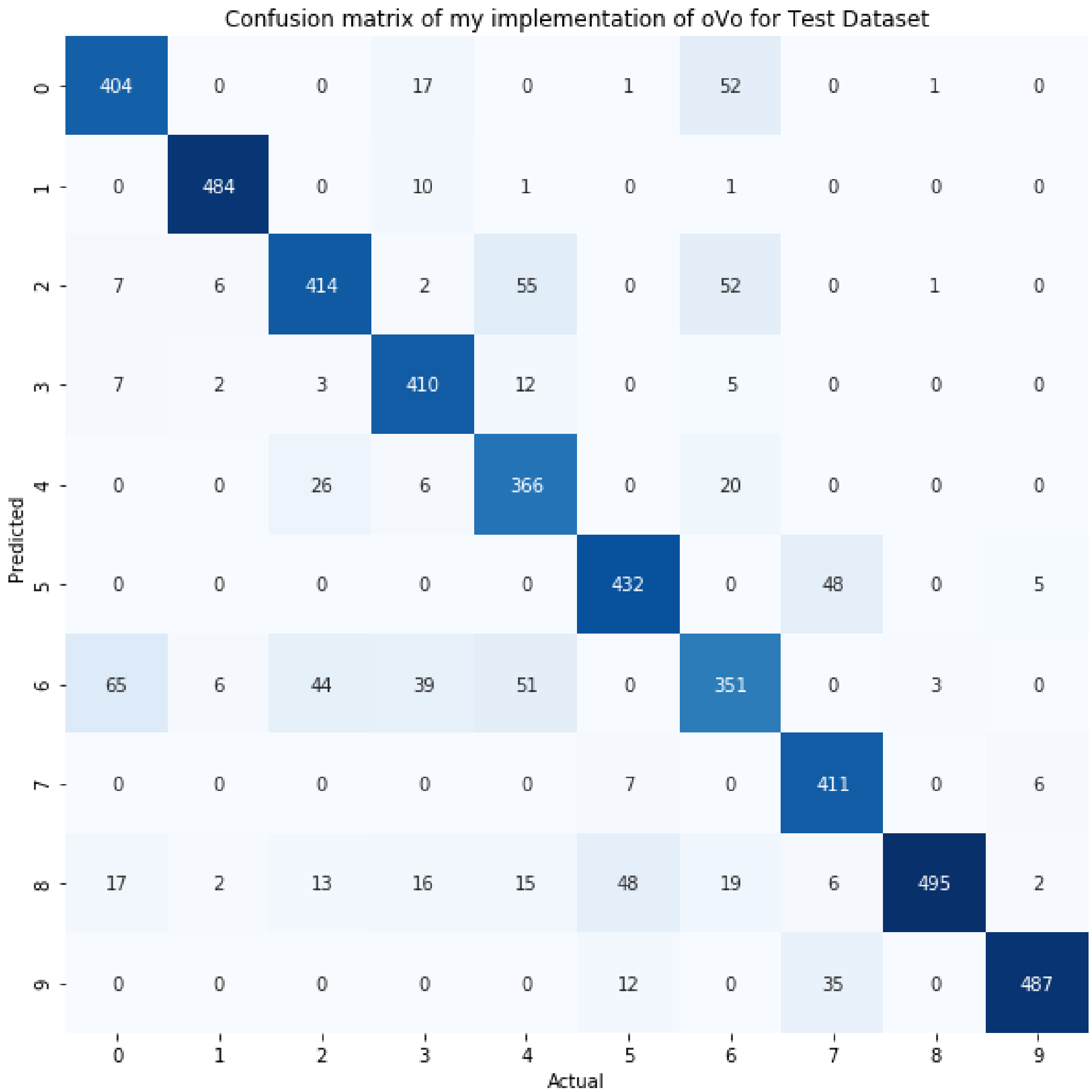
2.3.1 Comparing oVo with base models as SVM and SVC

Implementation	Accuracy On Train dataset	Accuracy On Validation dataset	Accuracy On Test dataseq	Training Time(Minutes)
OneVsOne with SVM as base learner	96.52	84.96	85.08	7.10
OneVsOne with SVC as base learner	96.91	87.88	88.08	0.46

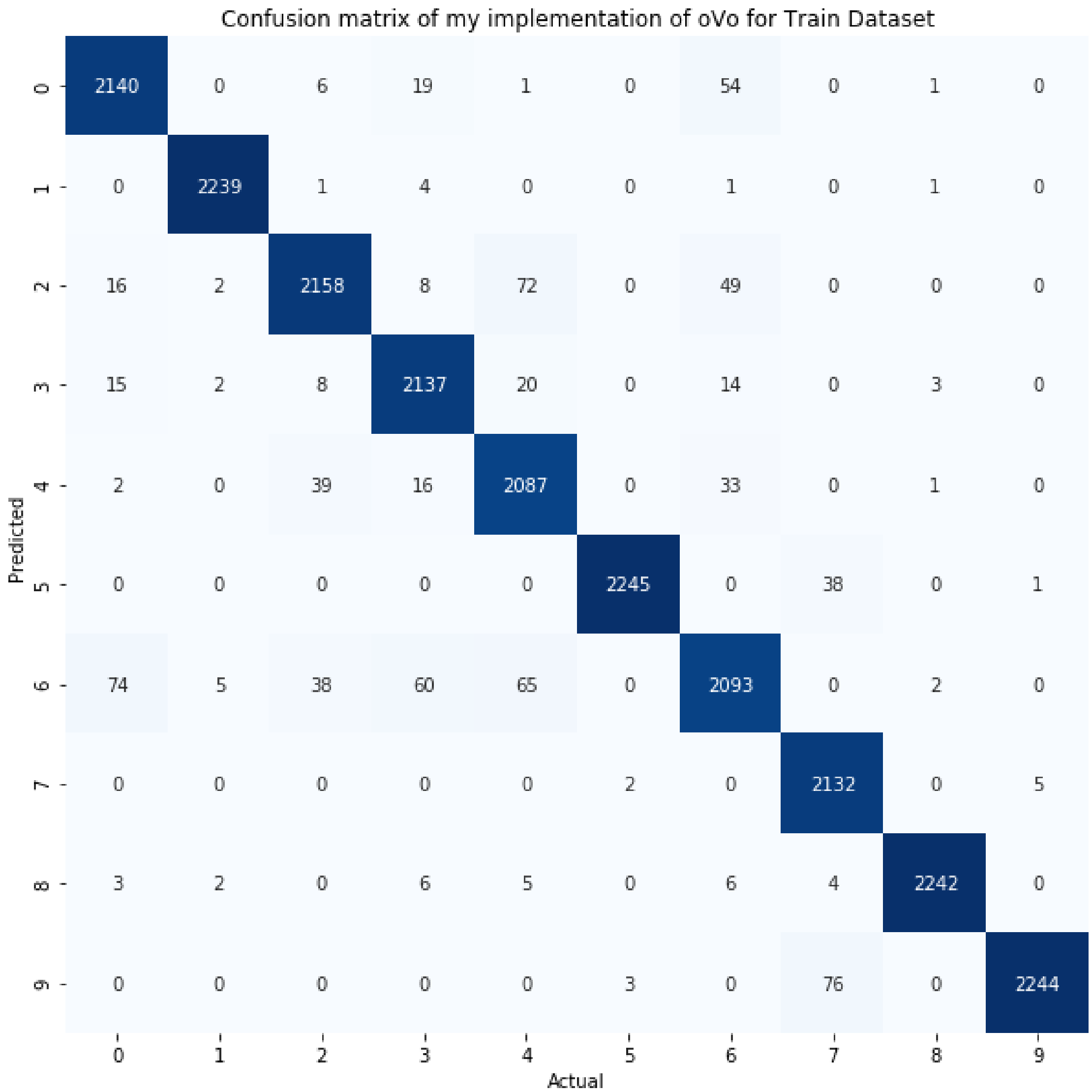
- Here we can see that accuracies when we take SVM as base models are lower. But as we found in last part that formula of b is wrong in our computation. So this was expected.
- This model was trained with parallelization on google colab server and we can see that sklearn's implementation trained faster. **Without parallelization** this code used to run for **more than an hour** but after using 20 cores on colab we gained sigificant speedup.

2.4 Part (c): Confusion Matrix

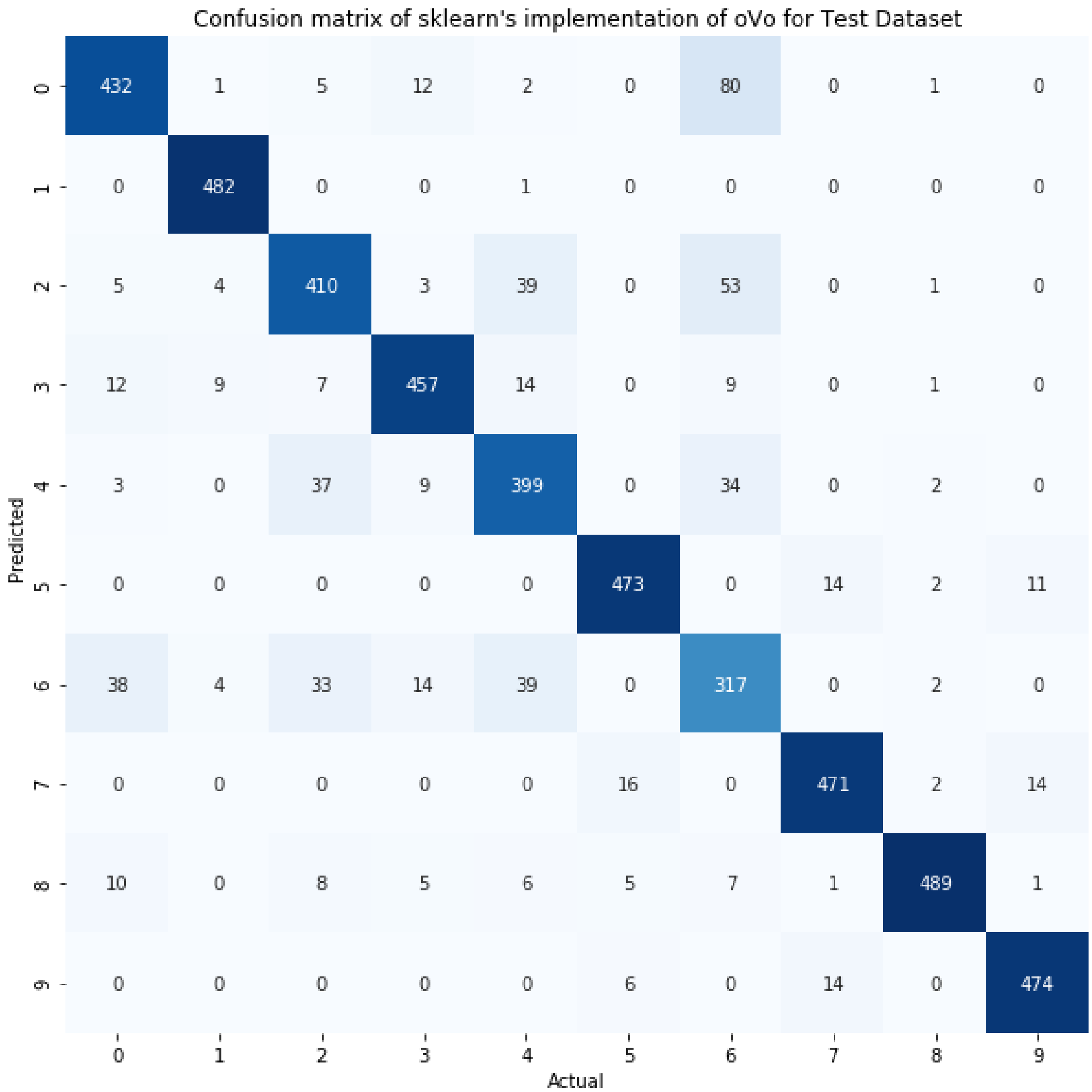
[0]:
```
metrics.confusion_matrix(Y_test_whole, pred_test_my, title='Confusion matrix of␣
↪my implementation of oVo for Test Dataset')
```



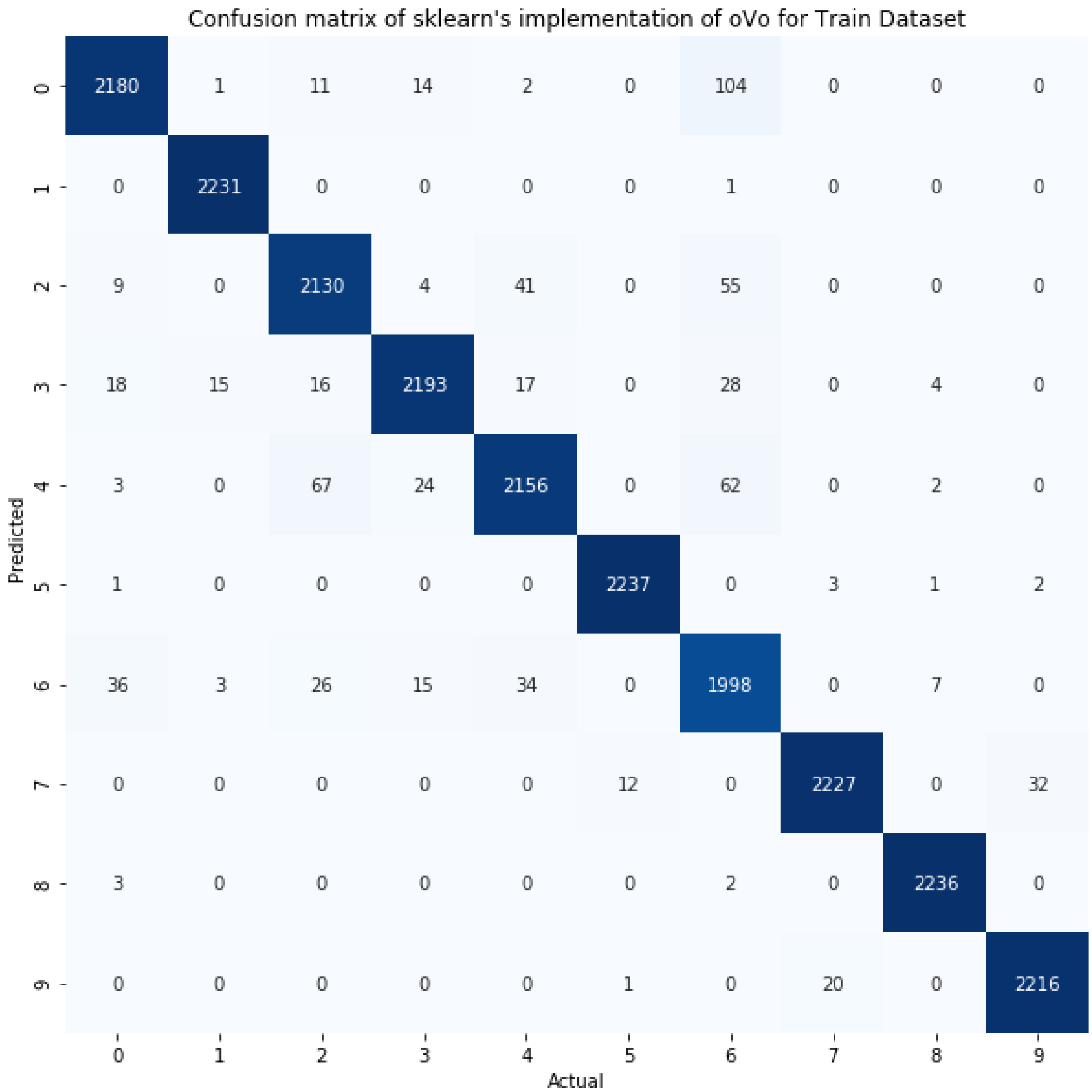
[0]:
```
metrics.confusion_matrix(Y_train_whole, pred_train_my, title='Confusion matrix␣
↪of my implementation of oVo for Train Dataset')
```




```
[0]: metrics.confusion_matrix(Y_test_whole, pred_test_sk, title='Confusion matrix of␣
     ↪sklearn\'s implementation of oVo for Test Dataset')
```



[0]:
```
metrics.confusion_matrix(Y_train_whole, pred_train_sk, title='Confusion matrix␣
↪of sklearn\'s implementation of oVo for Train Dataset')
```



- **Observations :**
  - Here we can see that all diagonal enries are large hence we can say that our model performed well as true poitives are large for all the classes.
  - 1. Here we can see that (6,0) has the large value and that is expected as class 0 is T-shirt/top and 6 is shirt.

2. (6,2) also has large value and 2=pullover and 6=shirt hance justified.
3. (6,4) also has large value and 4=coat and 6=shirt hance justified.
4. (2,4) also has large value and 2=pullover and 4=coat hance justified.
5. (7,8) also has large value and 7=sneaker and 9=Ankle boot justified.

Hence we can see that those classes which are similar are mostly misclassified between each other.

2.5 Part (d): k-fold Cross Validation

- In this part I have performed GridSearchCV to perform 5-fold cross validation on C=[1e-5, 1e-3, 1, 5, 10]. And then plotted average validation and average train accuracies along with test accuracy for each C.

[0]:
```
from sklearn.model_selection import GridSearchCV

#Performing 5-fold Cross validation
t0=time()
C_lst = [10**-5,10**-3,1,5,10]
gscv = GridSearchCV(SVC(kernel='rbf', gamma=0.05,␣
↪decision_function_shape='ovo'), {'C':C_lst}, cv=5, return_train_score=True,␣
↪n_jobs=-1)

gscv.fit(train[:,:-1]/255, train[:,-1].ravel())

print('Cross Validation completed in %d minutes'%((time()-t0)/60))
```

```
Cross Validation completed in 37 minutes
```

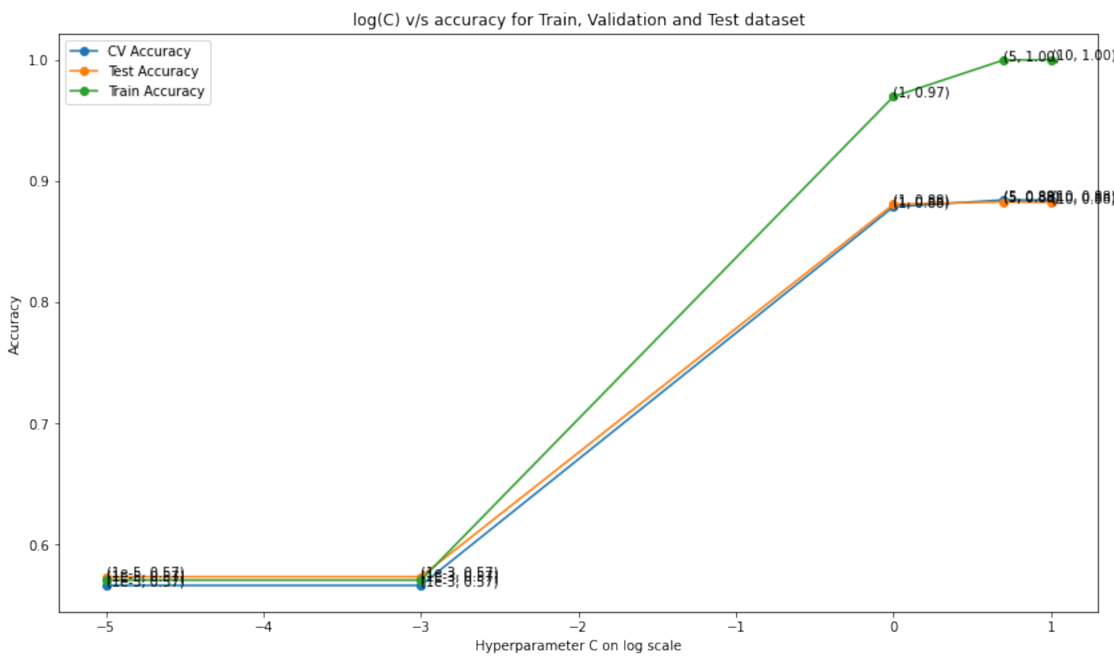
[0]:
```
def get_test_accuracy(c, X_train, Y_train, X_test, Y_test):
    '''
        Helper method that will be used in parallelization to find test␣
↪accuracy on 5 models corrosponding to each C
    '''
    clf = SVC(C=c,kernel='rbf', gamma=0.05, decision_function_shape='ovo')
    clf.fit(X_train, Y_train)
    return clf.score(X_test, Y_test)

#Getting test accuracies
test_score = Parallel(n_jobs=5)(delayed(get_test_accuracy)(c, train[:,:-1]/255,␣
↪train[:,-1].ravel(), test[:,:-1]/255, test[:,-1].ravel()) for c in C_lst)
```

[0]:
```
#Printing the best estimator
print(gscv.best_estimator_)
```

```
SVC(C=5, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovo', degree=3, gamma=0.05, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

[0]:
```
#plotting train, validation and test accuracies
```

```python
plt.figure(figsize=(14,8))
cv_score = gscv.cv_results_['mean_test_score']
train_score = gscv.cv_results_['mean_train_score']
plt.plot(np.log10(C_lst), cv_score, label='CV Accuracy',  marker='o')
plt.plot(np.log10(C_lst), test_score, label='Test Accuracy', marker='o')
plt.plot(np.log10(C_lst), train_score, label='Train Accuracy', marker='o')
for i_x, i_y, c in zip(np.log10(C_lst), cv_score,['1e-5', '1e-3', '1', '5',␣
↪'10']):
    plt.text(i_x, i_y, '(%s, %.2f)'%(c, i_y))
for i_x, i_y, c in zip(np.log10(C_lst), train_score,['1e-5', '1e-3', '1', '5',␣
↪'10']):
    plt.text(i_x, i_y, '(%s, %.2f)'%(c, i_y))
for i_x, i_y, c in zip(np.log10(C_lst), test_score,['1e-5', '1e-3', '1', '5',␣
↪'10']):
    plt.text(i_x, i_y, '(%s, %.2f)'%(c, i_y))
plt.legend()
plt.title('log(C) v/s accuracy for Train, Validation and Test dataset')
plt.xlabel('Hyperparameter C on log scale')
plt.ylabel('Accuracy')
plt.show()
```



[0]:
```python
print('Hyperparameter C : ', C_lst)
print('Train Score : ', train_score.tolist())
print('Test Score : ', test_score)
print('Validation Score : ', cv_score.tolist())
```

```
Hyperparameter C :  [1e-05, 0.001, 1, 5, 10]
Train Score :  [0.5708222222222222, 0.5708222222222222, 0.9696555555555557,
0.9997555555555555, 1.0]
Validation Score :  [0.5664444444444444, 0.5664444444444444, 0.8787111111111111,
0.8844, 0.8842666666666666]
Test Score :  [0.5736, 0.5736, 0.8808, 0.8828, 0.8824]
```

- **Observations :**
  - Here we got C=5 as best C as we got validation accuracy for C=5 of 88.44%
  - In below table I have summarized train, validation and test accuracies for all 5 values of C. And these are some observations from that table
    * Here We can see that we get the best test accuracy for C=5
    * Here we can also see ine interesting trend that as C increases accuracies tend to increase and for C=10 we have train accuracy of 100%. Hence we might be overfitting. This is in line with our intuition that as C increases chances of overfitting increases and as C decreases chances of underfitting increases

	C=1e-5	C=1e-3	C=1	C=5	C=10
Train Accuracy (in %)	57.0822	57.0822	96.9656	99.9756	100
Validation Accuracy (in %)	56.6444	56.6444	87.8711	88.44	88.4267
Test Accuracy (in %)	57.36	57.36	88.08	88.28	88.24