Hand Written Alphabet Classification-Naive Bayes

By:Sachin Yadav
(2213105)
Vishawjeet Jyoti
(2213103)

Supervisor: Dr. Satyanath Bhat

Indian Institute of Technology, Goa

Contents

1 ABSTRACT

Aim: To build a classifier for predicting Hand Written Alphabets from pixel data of images using Naive Bayes approach. Each alphabet is represented by a 28x28 grid of pixels and each pixel can take grey-scale values ranging 0-255. Each image has 784 pixels, which are then considered as independent random variables. And given a set of 784 pixels, we have to predict which letter is represented by the set. Since the pixels are independent random variables, hence Naive Bayes Multinomial class is used to predict the results.

2 INTRODUCTION

The image classification problem is one of the important research topics in the field of Artificial Intelligence. The rapid rise in the size of data in different fields has been observed lately. Automatic processing of these contents requires highly effective pattern classification or recognition techniques. Basically, automatic data classification tasks, including image retrieval require two critical processes one of them is extracting appropriate features, and another one is designing the correct classifier for this process. For image classification work, a feature extraction process can be thought as basis of content-based image extraction. The features can be further classified into two groups, domain related features and General features. The General features are like colors, textures etc., whereas the domain-related features are basically faces, fingerprints, and human irises. Out of these available features, we never know which one of the combinations of features is good for characterizing an image perfectly from the other images. The best approach is to utilize the available features. By using different methods, different features are extracted for other reasons. However, all the features can help in describing the objects more precisely. In most of cases, more features aid a more accurate classification.
In this text, in order to achieve high accuracy in classification we use Naive Bayes approach. Since Handwritten images are given in form of pixels we Assume these pixels as Independent features which makes Naive Bayes algorithm as best fit. The rest of the text is organized into multiple sections. We start discussing how Naive Bayes can be implemented and in later section we discuss the problem solving approach and finally we propose the results that we obtain.

3 NAIVE BAYES CLASSIFIER

The Naive Bayes classifier [1] is based on Bayes rule of probability. Bayes rule is used to calculate the conditional probability of a random variable. In Bayes Rule the conditional probability of an event A given that another event B has occurred is mathematically denoted as

$$P(A \mid B) = P(A) * P(B \mid A)/P(B) \tag{1}$$

We use the same idea in our problem of classifying handwritten alphabet images. Since the handwritten alphabets are given as images which in turn has 784 pixels, and each image has corresponding label Y, therefore we say Y is a random variable that takes values from 0 to 25 where 0 means A, 1 means B, and so on till 25, where 25 means Z. Similarly, we assume each of the pixels as a independent random variable as

$$X_1, X_2, X_3.....X_{784}, \tag{2}$$

Essentially we need to predict the class(k) given that we have some image as $X_1, X_2, X_3.....X_{784}$

This can be denoted mathematically as:

$$P(Y = k \mid X_1, X_2, X_3.....X_{784}) \tag{3}$$

Which reads as: "Probability of Y being k, where k = [0,25], given the features $X_1, X_2, X_3.....X_{784}$ ".
Using the Bayes rule, the above equation becomes:

$$\frac{P(Y = k, X_1, X_2, X_3.....X_{784})}{P(X_1, X_2, X_3.....X_{784})}$$

Which is same as:

$$P(Y = k) * \frac{P(X_1, X_2, X_3.....X_{784} \mid Y = k)}{P(X_1, X_2, X_3.....X_{784})}$$

Since the random variables X_i, where $i \in [1, 784]$ are independent, we can write the above equation as:

$$P(Y = k) * \frac{\prod_{i=1}^{\infty} P(X_i \mid Y = k)}{P(X_1, X_2, X_3.....X_{784})}$$

Here Naive Bayes classifier assumes that each feature or pixel in our case is independent and since $P(X_1, X_2, X_3.....X_{784})$ is same for each class hence we need to determine.

$$Y = argmax_k(P(Y = k) * \prod_{i=1}^{\infty} P(X_i \mid Y = k)) \tag{4}$$

Y will contain the predicted output of the given data point based on the derived probabilities for each class.
Let's suppose we have data set X_1=x_1, $X_2 = x_2$... and so on. And the probability of this data representing the letter A (i.e., Probability of this data representing class 0) = 0.03712653, probability of this data representing letter B = 0.02333796 ... and so on.

Let's represent these probabilities in a list of size 26 as:

Prob=[0.03712653 0.02333796 0.06266016 0.02720723 0.03071553 0.00314733 0.01539357 0.01956713 0.00302501 0.02271744 0.01513104 0.03101982 0.03309915 0.05109709 0.15533181 0.05186975 0.01552781 0.03110037 0.1299026 0.06042273 0.07813129 0.01115735 0.02888978 0.01682254 0.02931341 0.01628556]

Then, the maximum value in this list represents that the probability of the provided data belonging to that class is highest.
For the above list, highest probability value is 0.15533181 at the index number 13, hence the provided set of pixels would represent the letter corresponding to 13, which is M.

4 DEVELOPING THE MODEL

4.1 Setting up the Environment

Python is one of the general-purpose language, code written in python is easy to interpret and at the same time Python as a programming language is capable of writing very complex computer programs with ease. Due to these advantages, Python is heavily used in machine learning to build machine learning models and solve real-life problems which can be solved using statistical models.
To start working on this project we needed Python installed in the system. After the successful installation of python we installed jupyter notebook which is a web based integrated development environment. We begin by importing the necessary classes in our code.

```
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
```

The train_test_split is a function in sklearn library which is used to split a dataset into two separate data sets, one of which can then be used for training the model, and the other for testing the trained model.

The train_test_split function takes 5 parameters but not all of them are compulsory. One of the parameters is the features set, another parameter is the expected output set. One can tell the function the ratio in which the data set has to be divided by specifying the test_size or the train_size. In our model, the test size is taken as 10% and the train set size is taken as 90% of the original data set.

pandas and numpy are libraries which are used to work with the files, dataframes and arrays.
When we write "import pandas as pd", it basically means that throughout the code, we can use the pandas functionalities by calling them upon 'pd', same goes for the "import numpy as np" term.

4.2 Preparing the dataset

We start by importing the dataset into pandas dataframe by using below python code.

```
data = pd.read_csv("A_Z Handwritten Data.csv")
```

Every machine learning model require data, but before we start fitting the data into desired machine learning model, we need to take every precaution because fitting erroneous data may not give us the expected results; therefore its very important to analyse the data and then clean the data or removing errors from the data which could cause unexpected results at later stage. We can also say that to achieve highest level of accuracy it's extremely important to do data cleaning which improves the quality and accuracy of the machine learning model.
Depending upon the type of the data and state of the data we take different steps to clean the data set. Some of the common errors that we can find in data are incorrect values, corrupted values, incorrectly formatted and missing values etc.
We begin by dividing the entire data set to convert into training and test data set.

```
# Splitting dataset
X_train, X_test, Y_train, Y_test
= train_test_split(data,letterVal,test_size=0.1)
```

Here "*data*" is basically "$dataframe$" of features or pixels in our case. Similarly, labels corresponds to respective alphabets for each set of 784 pixels. We pass both "*data*" and "$letterVal$" to "$train_test_split$" method to split the entire data set into train and test data sets where train and test both having features being "*X_train*" and "*X_test*" and label being "*Y_train*" and "*Y_test*".

4.3 Selecting Appropriate Algorithm

After studying about the problem statement and understanding data, the next important phase is to choose the right algorithm to work on the problem, thus choosing right algorithm is essential and we cannot get away from this because choosing right model will have direct affect on the accuracy, interoperability, complexity, scalability and how long does the training and testing will take. As mentioned earlier every Image/Alphabet is represented in the form of pixels $X_1, X_2, X_3.....X_{784}$ where each of $X_1, X_2, X_3.....X_{784}$ are assumed to be independent random variable, which makes this classification problem multinomial. Hence we propose to build Naive Bayes Classifier.We know that we can use naive bayes classifier to solve multi-class prediction problems as it's quite useful with them. Naive Bayes classifier performs better than other models with less training data if the assumption of independence of features holds

4.4 Training the Model

Before using the model to predict the output for unlabelled features data, we first have to train our model using a large data set so that it can make use of the known data to predict the output. For this, we divide the original data set into two parts: The Train_set and the Test_set. Train_set contains 90% of the original data set and the remaining goes to the Test_set. The idea behind this is that the model should be trained on a large data set so that it recognises all the possible conditions. Hence it is ideal to divide the data set in (75+x):(25+x) ratio.

After splitting the data, we get 4 data sets as:

- X_train: This data set contains 335205 rows (90% of original data set), each row representing the features of an image. and there are 784 columns, each representing the corresponding pixel value. This is used for training the model.

- X_test: This data set contains the remaining rows of the original data set which are not in the X_train. This is used for testing the model.

- Y_train: This data set contains the same number of rows as that in X_train and a single column. It stores the letter values of the corresponding rows of X_train data set. This is used during the training time for establishing a relationship between the features and labels.

- Y_test: This data set contains the same number of rows as that in Y_train and a single column and stores the corresponding letter values. The predicted output from our model is matched with this data set to find the accuracy of our model.

Since we don't want to modify our original data sets to avoid any miscalculation during training, we make a copy of our data sets as:

```
forPrior = Y_train.copy()
#Creating a separate dataframe to work,
#so that the original value is not modified in the process

arr = X_train.to_numpy(None,True)
#True means that we are making
#a copy of X_train, so that any updation on arr
#doesn't change the values in X_train
```

Now forPrior is another data set which is just the copy of our Y_train data set, and arr is a 2-D array which stores the values of X_train.

What is needed for training the Model?
For finding a relationship between the features and labels, we need to find the following data:

- Prior probability: We need to find the prior probability of each Letter. Prior probability of a letter is the probability of occurrence of a letter in the data set without applying any conditions, i.e., Total number of times the letter is appearing in the X_train data set divided by the total number of letters in the X_train data set.
  Mathematically:
  $$P(Y = k) = \frac{n(k)}{n(X\_train)}$$
  $y \in [0, 25]$

  Where, n(k) is the number of times the letter 'k' has appeared the X_train data set and n(X_train) is the total number of data/features (basically number of rows) in X_train. We stored prior probability in an array of size 26. Where each index stores the prior probability of corresponding letter.

```
prior=np.ones(26)
n=0
while(n<26):
    prior[n]=numCount[n]/len2
n=n+1
```

- Conditional Probability: We need to find the conditional probability of every pixel of each letter, i.e., the probability of a pixel being 0 or 1 given that it is letter 'k'.

  Before finding the conditional probabilities of each pixel, we scaled down the pixel values from the range [0,255] to [0,1] so that it becomes a problem of binomial distribution.
  We went through every pixel of the 'arr' data set and if the pixel value was lesser than 128, we updated the value with 0, and otherwise with 1. Also, this is done on the copy of the X_train data set, so no values in the original training data set was modified.

```
while(r<rows):
    c=0
    while(c<columns):
        if(arr[r][c]<128):
            arr[r][c]=0
        else:
            arr[r][c]=1
        c=c+1
    r=r+1
```

count0 and count1 arrays:

- **count0:** count0 is an array of size 26x784 which stores the number of times 0 appeared for a pixel for a particular image. i.e., count0[0][x] would store the number of times the x'th pixel of letter 'A' was 0, out of the total number of times that 'A' appeared in the whole training data set.

- **count1:** Just like count0, this array stores the number of appearance of 1 in a pixel for a particular image.

Smoothing: Since there are some images for which some pixels are always 0 or always 1. and hence the count0 and count1 array will contain some cells which is equal to the total number of occurrences of that letter. This may cause divide by 0 error when we try to find the log of the conditional probability of each pixels, as the conditional probability of such a pixel being 1 (or 0) will turn out to be 0. So, to deal with such situations, we deliberately decrease the count of such cells by 1, so that the conditional probability doesn't turn out to be 0.

Finding Conditional Probabilities: After we have the data regarding the number of times a pixel is 0 or 1 and the prior probabilities, now we can find the conditional probability of the pixels of letters A-Z being 0 or 1. For this, we create two arrays of size 26x784:

- **conditional0:** This array will store the conditional probability of a pixel to be 0, given that a particular letter has appeared. So, conditional0[0][459] would store the conditional probability of the 460'th pixel being 0 given that letter 'A' has occurred.
  This is done as:

```
while(n<26):
    c=0
    while(c<784):
        conditional0[n][c]=count0[n][c]/numCount[n]
        c=c+1
    n=n+1
```

- **conditional1:** Just like conditional0, conditional1 array if also of size 26x784 and stores the conditional probability of a pixel being 1.
  This is done as:

```
n=0
while(n<26):
    c=0
    while(c<784):
        conditional1[n][c]=count1[n][c]/numCount[n]
        c=c+1
    n=n+1
```

Inference issues: Since all the pixels are independent random variables, hence their probabilities gets multiplied in the numerator of the Bayes' rule. Now, since all the values of conditional probabilities are small, and there are 784 features for each image, the very small values gets multiplied 784 times with each other, generating smaller values each time. There may be a time after which the python precision is not able to store the values without using the scientific notation. Hence, to get rid of such situation, we take the log

values of all the probabilities and then add them.
As:

$$log(a_1 * a_2 * ...) = log(a_1) + log(a_2) + ...$$

We store the corresponding log values in separate arrays of same size.

- logprior:

```
logprior=np.log(prior)
#creating an array from prior,
#storing the log values of correponding values
```

- logconditional0:

```
logconditional0=np.log(conditional0)
#creating an array from conditional,
#storing the log values
```

- logconditional1:

```
logconditional1=np.log(conditional1)
#creating an array from conditional1,
#storing the log values of corresponding values
```

4.5 Testing and Model Accuracy

Now that we have all the required data that relates the Features with their corresponding labels, we test our model by feeding the data from the X_test data set to the algorithm and then storing the predicted output in an array. which will then be compared with the actual output in order to verify the correctness of our prediction and finding the accuracy.

Since we don't want to modify the data of the original X_test data set, hence we make a copy of the test data into an array, we name it as 'testarr'.

```
 testarr=X_test.to_numpy(None,copy=True)
#testarr is a 2-D array of size same as our X_test
```

Now, since our model is built by converting the multinomial values into binomial values, we need to scale down the pixel values of 'testarr' as well in order to make predictions. This is done as:

```
numRow=testarr.shape[0]
numCol=testarr.shape[1]
n=0
c=0
while(n<numRow):
    c=0
    while(c<numCol):
        if(testarr[n][c]<128):
            testarr[n][c]=0
        else:
            testarr[n][c]=1
        c=c+1
    n=n+1
```

Now, to store the predicted output, we create another array of same size as the size of our 'testarr', and store the predicted values in corresponding indexes.

```
n=0
Predicted_OP=np.zeros(37245)
vec=np.zeros(784)
#vec will store the 784 features of
#the 'n'th data point and we will
#pass this to our function forall
while(n<37245):
#copying the features of our n'th data point in vec
    vec=testarr[n].copy()
    Predicted_OP[n]=forall(vec)
    n=n+1
```

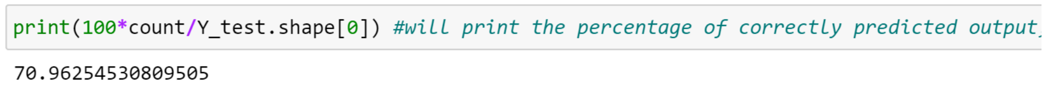
Once our model has completed predicting the values, we can compare the values in our Predicted_OP set and the Y_test set. We can maintain a count variable and increase the value of count by 1 each time a correct prediction has been made.

```
#creating a confusion_matrix
confusion_matrix=np.zeros([26,26])
count=0
n=0
while(n<Y_test.shape[0]):
```

```
x=int(Y_test.iloc[n,0])
y=int(Predicted_OP[n])
confusion_matrix[x,y]+=1
if(Predicted_OP[n]==Y_test.iloc[n,0]):
    count=count+1
    #if our predicted output==actual,
    #increase the count value by 1
n=n+1
```

At last we find the accuracy of our model by:

```
print(100*count/Y_test.shape[0])
#will print the percentage of
#correctly predicted output,
#i.e. the accuracy of our model
```



```
print(100*count/Y_test.shape[0]) #will print the percentage of correctly predicted output
70.96254530809505
```

Figure 2: Accuracy of Model

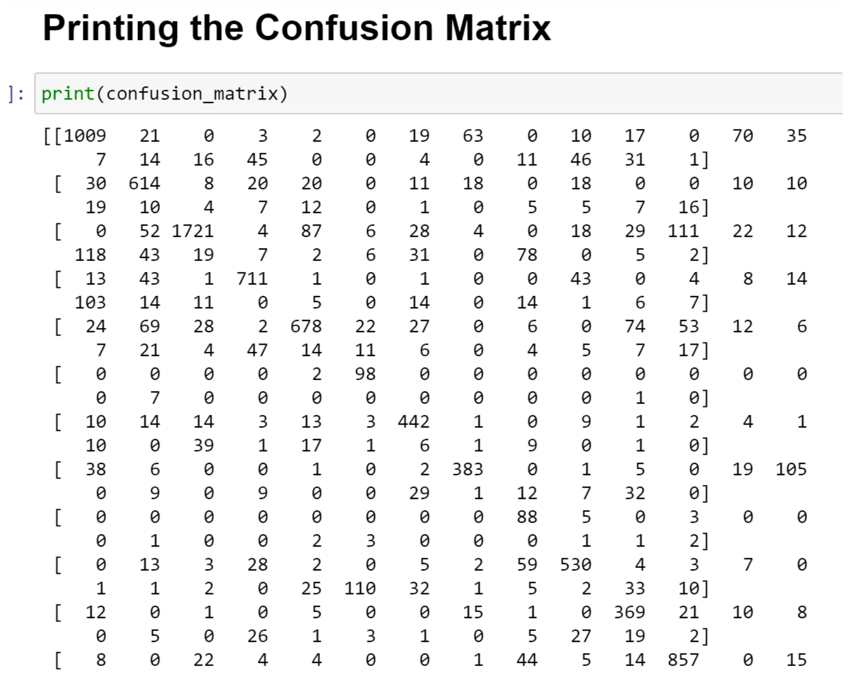
4.6 Evaluation

In order to evaluate the machine learning model we consider various metrics. Some of them are listed below.
1: Confusion Matrix
2: Accuracy
3: Precision
4: Sensitivity

Confusion matrix is a matrix representation of predicted results versus the actual results. On the row-side we represent the actual output values and on the column-side we represent the predicted output values. So, higher diagonal values implies that our predictions are strong. We defined a confusion matrix of 26x26 size as we have 26 letters in total.

```
confusion_matrix=np.zeros([26,26])
```

So, **confusion_matrix**[**x**][**y**] would store the value such that the actual output was 'x' and predicted output was 'y'. Using the confusion matrix we can also visualize which letters are likely to be confused.



Printing the Confusion Matrix

```
]: print(confusion_matrix)
```

```
[[1009   21    0    3    2    0   19   63    0   10   17    0   70   35
     7   14   16   45    0    0    4    0   11   46   31    1]
 [  30  614    8   20   20    0   11   18    0   18    0    0   10   10
    19   10    4    7   12    0    1    0    5    5    7   16]
 [   0   52 1721    4   87    6   28    4    0   18   29  111   22   12
   118   43   19    7    2    6   31    0   78    0    5    2]
 [  13   43    1  711    1    0    1    0    0   43    0    4    8   14
   103   14   11    0    5    0   14    0   14    1    6    7]
 [  24   69   28    2  678   22   27    0    6    0   74   53   12    6
     7   21    4   47   14   11    6    0    4    5    7   17]
 [   0    0    0    0    2   98    0    0    0    0    0    0    0    0
     0    7    0    0    0    0    0    0    0    0    1    0]
 [  10   14   14    3   13    3  442    1    0    9    1    2    4    1
    10    0   39    1   17    1    6    1    9    0    1    0]
 [  38    6    0    0    1    0    2  383    0    1    5    0   19  105
     0    9    0    9    0    0   29    1   12    7   32    0]
 [   0    0    0    0    0    0    0    0   88    5    0    3    0    0
     0    1    0    0    2    3    0    0    0    1    1    2]
 [   0   13    3   28    2    0    5    2   59  530    4    3    7    0
     1    1    2    0   25  110   32    1    5    2   33   10]
 [  12    0    1    0    5    0    0   15    1    0  369   21   10    8
     0    5    0   26    1    3    1    0    5   27   19    2]
 [   8    0   22    4    4    0    0    1   44    5   14  857    0   15
```

Figure 3: Confusion Matrix

1. True Positives(TP): It represents that we predicted positive and actual result was also positive.
2. True Negative(TN): We predicted negative and actual result was negative.
3. False Negative(FN): We predicted negative and actual result was positive.
4. False Positive(FP): We predicted positive and actual result was negative.

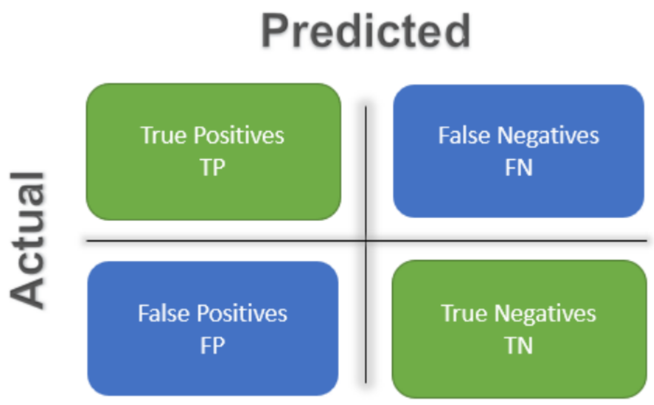
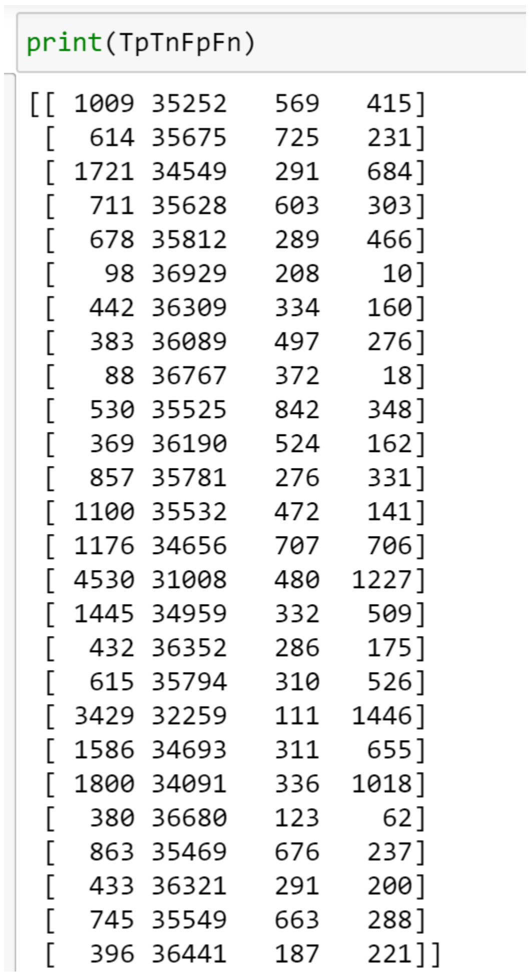


Figure 4: TpTnFpFN

We get the predicted results by passing the test data to our model. We have also constructed the confusion matrix by now. Now from the confusion matrix we can extract the TP, TN, FP and FN values for each letter. For confirming the correctness, we can count all the four values for any class, it should turn out to be equal to the total number of rows in our test data set.

So, each of the 26 letters will have these 4 values corresponding to them, hence we have created an array of size 26x4 which will store these values.

```
TpTnFpFn=np.zeros([26,4])
```



```
print(TpTnFpFn)
```

```
[[ 1009 35252   569   415]
 [  614 35675   725   231]
 [ 1721 34549   291   684]
 [  711 35628   603   303]
 [  678 35812   289   466]
 [   98 36929   208    10]
 [  442 36309   334   160]
 [  383 36089   497   276]
 [   88 36767   372    18]
 [  530 35525   842   348]
 [  369 36190   524   162]
 [  857 35781   276   331]
 [ 1100 35532   472   141]
 [ 1176 34656   707   706]
 [ 4530 31008   480  1227]
 [ 1445 34959   332   509]
 [  432 36352   286   175]
 [  615 35794   310   526]
 [ 3429 32259   111  1446]
 [ 1586 34693   311   655]
 [ 1800 34091   336  1018]
 [  380 36680   123    62]
 [  863 35469   676   237]
 [  433 36321   291   200]
 [  745 35549   663   288]
 [  396 36441   187   221]]
```

Figure 5: TP,TN,FP,FN values after prediction

Accuracy: Accuracy is basically the fraction of all the correct prediction made for a class over the total predictions. We measure Accuracy for a class as below.

$$\frac{TN + TP}{TN + TP + FP + FN} \tag{5}$$

From the above equation, the class-wise accuracy of our model is:

```
while(n<26):
    accuracy[n]=(TpTnFpFn[n][0]+TpTnFpFn[n][1])
   x=(TpTnFpFn[n][0]+TpTnFpFn[n][1])
   x=x+TpTnFpFn[n][2]+TpTnFpFn[n][3])
    accuracy[n]=accuracy[n]/x
   n=n+1
```

Sensitivity: Sensitivity which is sometimes referred as recall of a class. Sensitivity for a class is measured as fraction of positive predictions that are actually positive with respect to the sum of positive predictions that are actually positive and Negative prediction that are actually positive.

$$\frac{TP}{TP + FN} \tag{6}$$

From the above equation, the class-wise sensitivity of our model is:

```
while(n<26):
        sensitivity[n]=(TpTnFpFn[n][0])
        x=(TpTnFpFn[n][0]+TpTnFpFn[n][3])
        sensitivity[n]=sensitivity[n]/x
        n=n+1
```

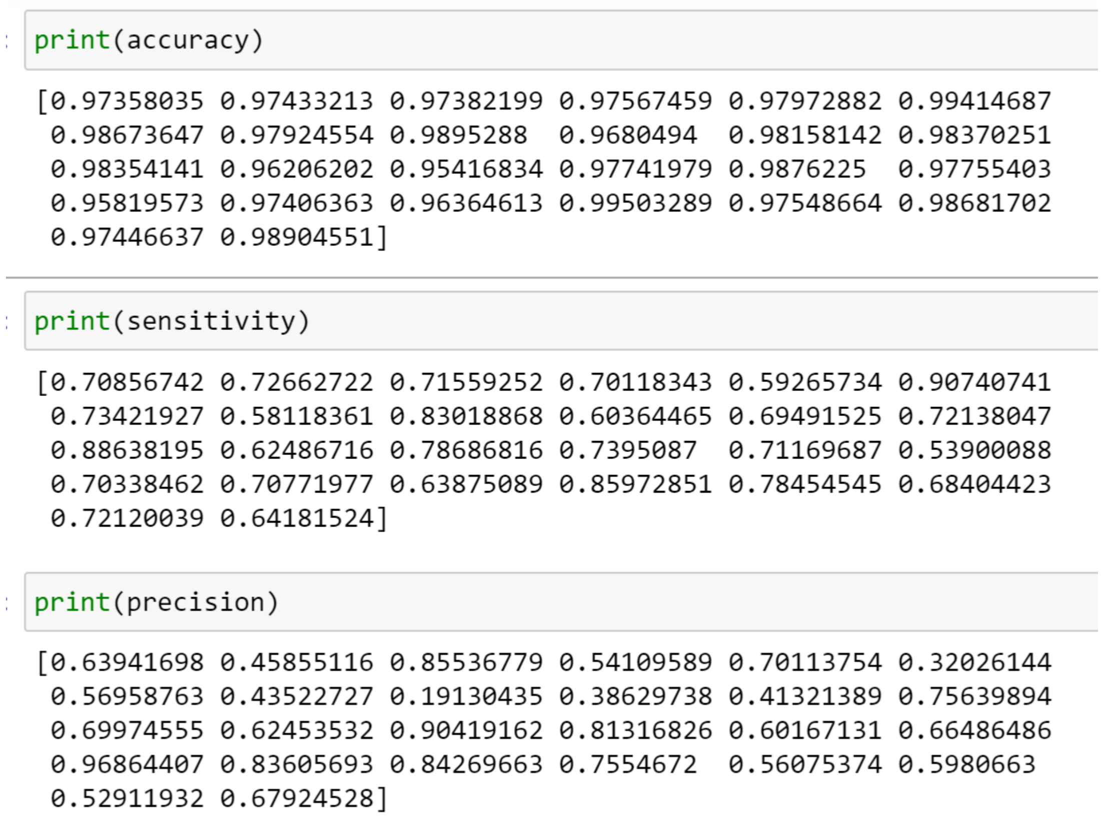
Precision: Precision is basically fraction of positive predictions that are actually positive with respect to the sum of positive predictions that are actually positive and Positive Predictions which are actually negative. We calculate Precision for a class as:

$$\frac{TP}{TP + FP} \tag{7}$$

From the above equation, the class-wise precision of our model is:

```
while(n<26):
    precision[n]=(TpTnFpFn[n][0])
   x=(TpTnFpFn[n][0]+TpTnFpFn[n][2])
```

```
    precision[n]=precision[n]/x
    n=n+1
```



```
print(accuracy)
```

```
[0.97358035 0.97433213 0.97382199 0.97567459 0.97972882 0.99414687
 0.98673647 0.97924554 0.9895288  0.9680494  0.98158142 0.98370251
 0.98354141 0.96206202 0.95416834 0.97741979 0.9876225  0.97755403
 0.95819573 0.97406363 0.96364613 0.99503289 0.97548664 0.98681702
 0.97446637 0.98904551]
```

```
print(sensitivity)
```

```
[0.70856742 0.72662722 0.71559252 0.70118343 0.59265734 0.90740741
 0.73421927 0.58118361 0.83018868 0.60364465 0.69491525 0.72138047
 0.88638195 0.62486716 0.78686816 0.7395087  0.71169687 0.53900088
 0.70338462 0.70771977 0.63875089 0.85972851 0.78454545 0.68404423
 0.72120039 0.64181524]
```

```
print(precision)
```

```
[0.63941698 0.45855116 0.85536779 0.54109589 0.70113754 0.32026144
 0.56958763 0.43522727 0.19130435 0.38629738 0.41321389 0.75639894
 0.69974555 0.62453532 0.90419162 0.81316826 0.60167131 0.66486486
 0.96864407 0.83605693 0.84269663 0.7554672  0.56075374 0.5980663
 0.52911932 0.67924528]
```

Figure 6: Class-wise Accuracy, Sensitivity and Precision

References

[1] Dong-Chul Park. Image classification using naive bayes classifier. *Int. J. Comput. Sci. Electron. Eng*, 4(3):135–139, 2016.