**CS498 AMO Homework 1**

Team :

Minyuan Gu ([minyuan3@illinois.edu](mailto:minyuan3@illinois.edu))

Yanislav Shterev ([shterev2@illinois.edu](mailto:shterev2@illinois.edu))

**Problem 1: Diabetes Classification**

We have been using the Pima Indians dataset to train a Naïve Bayes classifier to predict whether given patient has diabetes or not. The dataset contains 8 feature columns containing only continuous values and 1 label column having as values 1 (having diabetes) and 0 (negative diabetes results). There are total of 767 data points. We have used Normal (Gaussian) distribution for estimating the parameters. The classifier’s final accuracy has been gathered by the average of 10 random splits the data having 80% for training and 20% for testing.

As a second part to the problem we examined what the effect of clearing the missing data (omitting feature values having 0) will be. For 4 features from vectors (attribute 3 (Diastolic blood pressure), attribute 4 (Triceps skinfold thickness), attribute 6 (Body mass index), and attribute 8 (Age)) we replaced the zeros with NaN and in the process of training, parameter prediction and predicting (we only skipped these features/column values, but not the entire row, to retain largest possible training set size).

**Part 1 Accuracies**

The accuracies rounded up to the second decimal digit are as follow:

| **Setup** | **Cross-validation Accuracy** |
| --- | --- |
| Unprocessed data | 75.68% |
| 0-value elements ignored | 74.57% |

**Part 1 Code Snippets**

1. Calculation of distribution parameters
2. Calculation of naive Bayes predictions  
   Please find the above information from the below codes (for NOT 0 ignoring, refer to the end):  
   def train\_and\_predict\_ignore\_o(train\_data, test\_data):  
    # train/build the model based on training set data.  
    # Noted: the last column of both train\_data and test\_data are the label!  
    # Calculate p(Positive) & p(negative)  
    positive = train\_data[train\_data[:, 8] == 1]  
    negative = train\_data[train\_data[:, 8] == 0]  
     
    p\_positive = 1.0\*positive.shape[0]/train\_data.shape[0]  
    p\_negative = 1 - p\_positive  
     
    # adjust attribute 3,4,6,8 to ignore the zero values  
    positive[(positive[:, 2] == 0), 2] = np.nan # filter out attribute 3  
    positive[(positive[:, 3] == 0), 3] = np.nan # filter out attribute 4  
    positive[(positive[:, 5] == 0), 5] = np.nan # filter out attribute 6  
    positive[(positive[:, 7] == 0), 7] = np.nan # filter out attribute 8  
     
    # Calculate mean and variance for all features for positive samples  
    positive\_mean = np.nanmean(positive[:,:-1], axis=0)  
    positive\_var = np.nanvar(positive[:,:-1], axis=0)  
     
    # adjust attribute 3,4,6,8 to ignore the zero values  
    negative[(negative[:, 2] == 0), 2] = np.nan # filter out attribute 3  
    negative[(negative[:, 3] == 0), 3] = np.nan # filter out attribute 4  
    negative[(negative[:, 5] == 0), 5] = np.nan # filter out attribute 6  
    negative[(negative[:, 7] == 0), 7] = np.nan # filter out attribute 8  
     
    # Calculate mean and variance for all features for negative samples  
    negative\_mean = np.nanmean(negative[:,:-1], axis=0)  
    negative\_var = np.nanvar(negative[:,:-1], axis=0)  
     
    # to predict on the test data set.  
    test\_X = np.array(test\_data)[:, :-1]  
    test\_Y = np.array(test\_data)[:, -1]  
     
    p\_x\_positive = np.log(norm.pdf(test\_X, positive\_mean, np.sqrt(positive\_var)))  
    p\_x\_negative = np.log(norm.pdf(test\_X, negative\_mean, np.sqrt(negative\_var)))  
     
    # ignore the features with zero values for both classes  
    for index, item in enumerate(test\_X):  
    if item[2] == 0:  
    p\_x\_positive[index][2] = 0  
    p\_x\_negative[index][2] = 0  
    if item[3] == 0:  
    p\_x\_positive[index][3] = 0  
    p\_x\_negative[index][3] = 0  
    if item[5] == 0:  
    p\_x\_positive[index][5] = 0  
    p\_x\_negative[index][5] = 0  
    if item[7] == 0:  
    p\_x\_positive[index][7] = 0  
    p\_x\_negative[index][7] = 0  
     
    positive\_class = np.sum(p\_x\_positive, axis=1) + np.log(p\_positive)  
    negative\_class = np.sum(p\_x\_negative, axis=1) + np.log(p\_negative)  
    predicted = (positive\_class > negative\_class)\*1  
    accuracy = sum(predicted == test\_Y)/test\_Y.shape[0]  
    return accuracy\*100
3. Test-train split code

split\_idx = int(data.shape[0]\*0.2)  
total\_accuracy = 0.0  
total\_iteration = 10  
for i in range(total\_iteration):  
 np.random.shuffle(data)  
 test\_set = data[:split\_idx, :]  
 train\_set = data[split\_idx:, :]  
 total\_accuracy = total\_accuracy + train\_and\_predict\_ignore\_o(train\_set, test\_set)  
  
average\_accuracy = total\_accuracy/total\_iteration  
print(**"Averaged accuracy of cross validation is : "**, average\_accuracy)

**Part 2 MNIST Accuracies**

We used the MNIST dataset located at <https://github.com/amplab/datascience-sp14/raw/master/lab7/mldata/mnist-original.mat> to train Naïve Bayes classifier with Normal and Bernoulli distributions for estimating the parameters for the posterior probability. As comparison to this we also used the Random Forest classifier from the sklearn library to train and predict over the same dataset using different number of trees 10, 30 and different depth 4, 16 as additional parameters.

For each of the classifiers above we used untouched images (no cropping or resizing over the original image pixels) and bounded box stretched images (bounded and resized to 20x20 dimensions) as input. The results show that models using cleaned images have better accuracy.

Results are as follows:

| **x** | **Method** | **Training Set Accuracy** | **Test Set Accuracy** |
| --- | --- | --- | --- |
| 1 | Gaussian + untouched | 79.47% | 80.62% |
| 2 | Gaussian + stretched | 83.16% | 84.04% |
| 3 | Bernoulli + untouched | 83.32% | 84.27% |
| 4 | Bernoulli + stretched | 81.55% | 82.94% |
| 5 | 10 trees + 4 depth + untouched | 70.86% | 69.7% |
| 6 | 10 trees + 4 depth + stretched | 71.06% | 73.73% |
| 7 | 10 trees + 16 depth + untouched | 98.99% | 93.7% |
| 8 | 10 trees + 16 depth + stretched | 99.52% | 94.41% |
| 9 | 30 trees + 4 depth + untouched | 75.05% | 72.72% |
| 10 | 30 trees + 4 depth + stretched | 74.38% | 75.55% |
| 11 | 30 trees + 16 depth + untouched | 99.46% | 95.52% |
| 12 | 30 trees + 16 depth + stretched | 99.74% | 96.31% |

**Part 2A Digit Images**

| **Digit** | **Mean Image** |
| --- | --- |
| 0 |  |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 |  |

**Part 2 Code**

* **Calculation of the Normal distribution parameters**  
  The below function belongs to class of NaiveBayesNormalDistr. epsilon is like var\_smoothing from the sklearn library GaussianNB. Please refer to the full code attached at the end.

def train(self, train\_data, train\_labels):  
 self.digits = []  
 self.p = []  
 self.digits\_mean = []  
 self.digits\_var = []  
 for i in range(10):  
 self.digits.append(train\_data[train\_labels[:] == i])  
 self.p.append(1.0 \* self.digits[i].shape[0] / train\_data.shape[0])   
 self.digits\_mean.append(np.mean(self.digits[i], axis=0))   
 self.digits\_var.append(np.var(self.digits[i], axis=0))   
 self.digits\_var = np.array(self.digits\_var)  
 self.digits\_var += self.epsilon \* self.digits\_var.max()

* **Calculation of the Bernoulli distribution parameters**  
  The below function belongs to class of NaiveBayesBernoulli. We applied addictive smoothing (LAPLACE smoothing) with α = 1 (plus one). We mark ink pixels’ value = 1, instead of 255 (refer to full code for threshold and assigning paper pixel =0 and ink pixel=1), so we can use sum to calculate p(ink|digit).

def train(self, train\_data, train\_labels):  
 self.digits = []  
 self.p = []  
 self.digits\_p\_ink = []  
 for i in range(10):  
 self.digits.append(train\_data[train\_labels[:] == i])  
 self.p.append(1.0 \* self.digits[i].shape[0] / train\_data.shape[0])   
 p\_ink = (np.sum(self.digits[i], axis=0) + 1) / (self.digits[i].shape[0] + train\_data.shape[1])  
 self.digits\_p\_ink.append(p\_ink)  
 self.digits\_p\_ink = np.array(self.digits\_p\_ink)

* **Calculation of the Naive Bayes predictions**

def predict(self, test\_data):  
 self.p\_digit\_class = []  
 self.predicted = []  
 if self.digits\_mean == [] or self.digits\_var ==[] or self.p == []:  
 print("Fit your model to training data first")  
 return []  
  
 for i in range(10):  
 normpdf = norm.pdf(test\_data, self.digits\_mean[i], np.sqrt(self.digits\_var[i]))  
 p\_post = np.sum(np.log(normpdf), axis=1) + np.log(self.p[i])  
 self.p\_digit\_class.append(p\_post)  
 self.p\_digit\_class = np.array(self.p\_digit\_class)  
 self.predicted = np.argmax(self.p\_digit\_class, axis=0)  
 return self.predicted

* **Training of a decision tree & Calculation of a decision tree predictions:**

def train\_and\_validate\_randomforest(train\_data, train\_labels, test\_data, test\_label, n\_trees, depth):  
 clf = RandomForestClassifier(n\_estimators=n\_trees, max\_depth=depth)  
 clf.fit(train\_data, train\_labels)  
 predicted = clf.predict(test\_data)  
 accuracy = sum(predicted == test\_label)/test\_label.shape[0]  
 return accuracy

Please refer to the full code for details how to apply the above function on the training data and predict.

**Full Code of Problem 1**

# -\*- coding: utf-8 -\*-  
"""Apply Naive Bayes on Diabetes Classification  
  
"""  
  
import numpy as np  
from scipy.stats import norm  
import csv  
  
  
# The following function will train & predict using Naive Bayes of Normal Distribution Model.  
# This function will ignore the zero values on column 3,4,6,8 as missing data.  
def train\_and\_predict\_ignore\_o(train\_data, test\_data):  
 # train/build the model based on training set data.  
 # Noted: the last column of both train\_data and test\_data are the label!  
  
 # Calculate p(Positive) & p(negative)  
 positive = train\_data[train\_data[:, 8] == 1]  
 negative = train\_data[train\_data[:, 8] == 0]  
  
 p\_positive = 1.0\*positive.shape[0]/train\_data.shape[0]  
 p\_negative = 1 - p\_positive  
  
 # adjust attribute 3,4,6,8 to ignore the zero values  
 positive[(positive[:, 2] == 0), 2] = np.nan # filter out attribute 3  
 positive[(positive[:, 3] == 0), 3] = np.nan # filter out attribute 4  
 positive[(positive[:, 5] == 0), 5] = np.nan # filter out attribute 6  
 positive[(positive[:, 7] == 0), 7] = np.nan # filter out attribute 8  
  
 # Calculate mean and variance for all features for positive samples  
 positive\_mean = np.nanmean(positive[:,:-1], axis=0)  
 positive\_var = np.nanvar(positive[:,:-1], axis=0)  
  
 # adjust attribute 3,4,6,8 to ignore the zero values  
 negative[(negative[:, 2] == 0), 2] = np.nan # filter out attribute 3  
 negative[(negative[:, 3] == 0), 3] = np.nan # filter out attribute 4  
 negative[(negative[:, 5] == 0), 5] = np.nan # filter out attribute 6  
 negative[(negative[:, 7] == 0), 7] = np.nan # filter out attribute 8  
  
 # Calculate mean and variance for all features for negative samples  
 negative\_mean = np.nanmean(negative[:,:-1], axis=0)  
 negative\_var = np.nanvar(negative[:,:-1], axis=0)  
  
 # to predict on the test data set.  
 test\_X = np.array(test\_data)[:, :-1]  
 test\_Y = np.array(test\_data)[:, -1]  
  
 p\_x\_positive = np.log(norm.pdf(test\_X, positive\_mean, np.sqrt(positive\_var)))  
 p\_x\_negative = np.log(norm.pdf(test\_X, negative\_mean, np.sqrt(negative\_var)))  
  
 # ignore the features with zero values for both classes  
 for index, item in enumerate(test\_X):  
 if item[2] == 0:  
 p\_x\_positive[index][2] = 0  
 p\_x\_negative[index][2] = 0  
 if item[3] == 0:  
 p\_x\_positive[index][3] = 0  
 p\_x\_negative[index][3] = 0  
 if item[5] == 0:  
 p\_x\_positive[index][5] = 0  
 p\_x\_negative[index][5] = 0  
 if item[7] == 0:  
 p\_x\_positive[index][7] = 0  
 p\_x\_negative[index][7] = 0  
  
 positive\_class = np.sum(p\_x\_positive, axis=1) + np.log(p\_positive)  
 negative\_class = np.sum(p\_x\_negative, axis=1) + np.log(p\_negative)  
 predicted = (positive\_class > negative\_class)\*1  
 accuracy = sum(predicted == test\_Y)/test\_Y.shape[0]  
 return accuracy\*100  
  
  
# The following function will train & predict using Naive Bayes of Normal Distribution Model.  
# This function will NOT ignore the zero values on column 3,4,6,8 as missing data.  
def train\_and\_predict(train\_data, test\_data):  
 # train/build the model based on training set data  
 # Calculate p(Positive) & p(negative)  
 positive = train\_data[train\_data[:, 8] == 1]  
 negative = train\_data[train\_data[:, 8] == 0]  
  
 p\_positive = 1.0\*positive.shape[0]/train\_data.shape[0]  
 p\_negative = 1 - p\_positive  
  
 # Calculate mean and variance for all features for both positive and negative samples  
 positive\_mean = np.mean(positive, axis=0)[:-1]  
 positive\_var = np.var(positive, axis=0)[:-1]  
  
 negative\_mean = np.mean(negative, axis=0)[:-1]  
 negative\_var = np.var(negative, axis=0)[:-1]  
  
 # to predict on the test data set.  
 test\_X = np.array(test\_data)[:, :-1]  
 test\_Y = np.array(test\_data)[:, -1]  
  
 positive\_class = np.sum(np.log(norm.pdf(test\_X, positive\_mean, np.sqrt(positive\_var))), axis=1) + np.log(p\_positive)  
 negative\_class = np.sum(np.log(norm.pdf(test\_X, negative\_mean, np.sqrt(negative\_var))), axis=1) + np.log(p\_negative)  
 predicted = (positive\_class > negative\_class)\*1  
 accuracy = sum(predicted == test\_Y)/test\_Y.shape[0]  
 return accuracy\*100  
  
  
data = []  
X = []  
Y = []  
# import the data from the csv.  
with open(**'pima-indians-diabetes.csv'**, newline=**''**) as f:  
 reader = csv.reader(f, delimiter=**','**, quoting=csv.QUOTE\_NONNUMERIC)  
 for row in reader:  
 data.append(row)  
 # remove the header from the data.  
 data.pop(0)  
  
data = np.array(data)  
split\_idx = int(data.shape[0]\*0.2)  
  
total\_accuracy = 0.0  
total\_iteration = 10  
for i in range(total\_iteration):  
 np.random.shuffle(data)  
 test\_set = data[:split\_idx, :]  
 train\_set = data[split\_idx:, :]  
 total\_accuracy = total\_accuracy + train\_and\_predict\_ignore\_o(train\_set, test\_set)  
  
average\_accuracy = total\_accuracy/total\_iteration  
print(**"Averaged accuracy of cross validation (ignoring zero values) is : "**, average\_accuracy, **"%"**)  
  
total\_accuracy = 0.0  
average\_accuracy = 0.0  
for i in range(total\_iteration):  
 np.random.shuffle(data)  
 test\_set = data[:split\_idx, :]  
 train\_set = data[split\_idx:, :]  
 total\_accuracy = total\_accuracy + train\_and\_predict(train\_set, test\_set)  
  
average\_accuracy = total\_accuracy/total\_iteration  
print(**"Averaged accuracy of cross validation is : "**, average\_accuracy, **"%"**)

**Full Code of Problem 2**

# -\*- coding: utf-8 -\*-  
"""Apply Naive Bayes (Gaussian & Bernoulli) and Random Forest on MNIST digits Classification  
  
"""  
  
import numpy as np  
import matplotlib.pyplot as plt  
from scipy.stats import norm  
from scipy.stats import bernoulli  
from sklearn.ensemble import RandomForestClassifier  
from mnist import MNIST  
import cv2  
  
####################################################  
# Stretch images to 400 (20x20) #  
####################################################  
def stretch\_bounding\_box(single\_image\_data):  
 single\_image\_data = single\_image\_data.reshape((28, 28))  
 vertical\_min = np.nonzero(single\_image\_data)[0].min()  
 vertical\_max = np.nonzero(single\_image\_data)[0].max()  
 horizon\_min = np.nonzero(single\_image\_data)[1].min()  
 horizon\_max = np.nonzero(single\_image\_data)[1].max()  
 return cv2.resize(single\_image\_data[vertical\_min: vertical\_max+1, horizon\_min:horizon\_max+1], (20, 20)).reshape(400)  
  
  
####################################################  
# The Gaussian NB Classifier #  
####################################################  
class NaiveBayesNormalDistr:  
 def \_\_init\_\_(self, epsilon=1e-9):  
 self.epsilon = epsilon  
 self.p\_digit\_class = []  
 self.predicted = []  
 self.digits = []  
 self.digits\_mean = []  
 self.digits\_var = []  
 self.p = []  
  
 def train(self, train\_data, train\_labels):  
 self.digits = []  
 self.p = []  
 self.digits\_mean = []  
 self.digits\_var = []  
 # separate training data into different classes (digits)  
 for i in range(10):  
 self.digits.append(train\_data[train\_labels[:] == i])  
 # Calculate p(digits 0,1,2,....9)  
 self.p.append(1.0 \* self.digits[i].shape[0] / train\_data.shape[0]) # len(p) is 10  
 # Calculate mean and variance for all features for all classes(digits)  
 self.digits\_mean.append(np.mean(self.digits[i], axis=0)) # each digits\_mean[i] shape is (784,1)  
 self.digits\_var.append(np.var(self.digits[i], axis=0)) # each digits\_var[i] shape is (784,1)  
 self.digits\_var = np.array(self.digits\_var)  
 self.digits\_var += self.epsilon \* self.digits\_var.max()  
  
 def predict(self, test\_data):  
 self.p\_digit\_class = []  
 self.predicted = []  
 if self.digits\_mean == [] or self.digits\_var ==[] or self.p == []:  
 print(**"Fit your model to training data first"**)  
 return []  
  
 for i in range(10):  
 normpdf = norm.pdf(test\_data, self.digits\_mean[i], np.sqrt(self.digits\_var[i]))  
 p\_post = np.sum(np.log(normpdf), axis=1) + np.log(self.p[i])  
 self.p\_digit\_class.append(p\_post)  
 self.p\_digit\_class = np.array(self.p\_digit\_class)  
 self.predicted = np.argmax(self.p\_digit\_class, axis=0)  
 return self.predicted  
  
 def get\_accuracy(self, test\_label):  
 if len(self.predicted) == 0:  
 print(**"Run predict() on your test data first"**)  
 return 0  
 elif len(self.predicted)!=len(test\_label):  
 print(**"Your test label shape mismatch the shape of your prediction data"**)  
 return 0  
 accuracy = sum(self.predicted == test\_label) / len(test\_label)  
 return accuracy  
  
 def plot\_all\_digits\_mean(self):  
 if len(self.digits\_mean) > 0:  
 # convert each digit mean from [10,784] to [10,28,28] (or [10,400] to [10,20,20])  
 digit\_mean\_to\_plot = np.array(self.digits\_mean)  
 image\_size = int(np.sqrt(digit\_mean\_to\_plot.shape[1]))  
 digit\_mean\_to\_plot = digit\_mean\_to\_plot.reshape((digit\_mean\_to\_plot.shape[0],image\_size,image\_size))  
 digit\_mean\_to\_plot = (digit\_mean\_to\_plot\*255).astype(int)  
  
 mainFigure = plt.figure(figsize=(10, 8))  
 columns = 5  
 rows = 2  
 for i in range(1, columns \* rows + 1):  
 mainFigure.add\_subplot(rows, columns, i)  
 plt.imshow(digit\_mean\_to\_plot[i-1], cmap=**'gray'**)  
 plt.show()  
 else:  
 print(**"Train your model with data first."**)  
  
  
####################################################  
# The Bernoulli NB Classifier #  
####################################################  
class NaiveBayesBernoulli:  
 def \_\_init\_\_(self):  
 self.predicted = []  
 self.digits = []  
 self.p = []  
 self.p\_digit\_class = []  
 self.digits\_p\_ink = []  
  
 def train(self, train\_data, train\_labels):  
 self.digits = []  
 self.p = []  
 self.digits\_p\_ink = []  
 # separate training data into different classes (digits)  
 for i in range(10):  
 self.digits.append(train\_data[train\_labels[:] == i])  
 # Calculate p(digits 0,1,2,....9)  
 self.p.append(1.0 \* self.digits[i].shape[0] / train\_data.shape[0]) # len(self.p) is 10  
 # Count each ink pixels to calculate p(ink|C), using plus-one smoothing  
 # Count of ink pixel +1 / Total images of such digit + number of ink pixels (dimension of row sample)  
 p\_ink = (np.sum(self.digits[i], axis=0) + 1) / (self.digits[i].shape[0] + train\_data.shape[1])  
 self.digits\_p\_ink.append(p\_ink)  
 self.digits\_p\_ink = np.array(self.digits\_p\_ink)  
  
 def predict(self, test\_data):  
 self.p\_digit\_class = []  
 self.predicted = []  
 if self.p == [] or self.digits\_p\_ink == []:  
 print(**"Fit your model to training data first"**)  
 return []  
  
 for i in range(10):  
 berpmf = bernoulli.pmf(test\_data, self.digits\_p\_ink[i])  
 p\_post = np.sum(np.log(berpmf), axis=1) + np.log(self.p[i])  
 self.p\_digit\_class.append(p\_post)  
  
 self.p\_digit\_class = np.array(self.p\_digit\_class)  
 self.predicted = np.argmax(self.p\_digit\_class, axis=0)  
 return self.predicted  
  
 def get\_accuracy(self, test\_label):  
 if len(self.predicted) == 0:  
 print(**"Run predict() on your test data first"**)  
 return 0  
 elif len(self.predicted) != len(test\_label):  
 print(**"Your test label shape mismatch the shape of your prediction data"**)  
 return 0  
 accuracy = sum(self.predicted == test\_label) / len(test\_label)  
 return accuracy  
  
  
####################################################  
# The RandomForest Classifier #  
####################################################  
def train\_and\_validate\_randomforest(train\_data, train\_labels, test\_data, test\_label, n\_trees, depth):  
 clf = RandomForestClassifier(n\_estimators=n\_trees, max\_depth=depth)  
 clf.fit(train\_data, train\_labels)  
 predicted = clf.predict(test\_data)  
 accuracy = sum(predicted == test\_label)/test\_label.shape[0]  
 return accuracy  
  
  
mndata = MNIST(**'./MNIST'**)  
mndata.gz = True  
images, labels = mndata.load\_training()  
test\_images, test\_labels = mndata.load\_testing()  
  
# filter out the mid grey pixels and convert it into binary picture  
ink\_threshold = 255\*0.5  
images = np.array(images, dtype=**'uint8'**)  
images[images[:] < ink\_threshold] = 0  
images[images[:] >= ink\_threshold] = 1 # mark it as ink pixel  
labels = np.array(labels, dtype=**'uint8'**)  
  
test\_images = np.array(test\_images, dtype=**'uint8'**)  
test\_images[test\_images[:] < ink\_threshold] = 0  
test\_images[test\_images[:] >= ink\_threshold] = 1 # mark it as ink pixel  
test\_labels = np.array(test\_labels, dtype=**'uint8'**)  
  
  
# produce the stretched images for train and test set  
stretched\_image\_map = map(stretch\_bounding\_box, images)  
stretched\_image = np.array(list(stretched\_image\_map))  
  
stretched\_test\_image\_map = map(stretch\_bounding\_box, test\_images)  
stretched\_test\_image = np.array(list(stretched\_test\_image\_map))  
  
  
####################################################  
# The following predict over TEST data #  
####################################################  
# use Naive Bayes Normal D to train and predict on untouched test images:  
nb\_normal = NaiveBayesNormalDistr(1e-1)  
nb\_normal.train(images, labels)  
\_ = nb\_normal.predict(test\_images)  
print(**"Naive Bayes - normal distribution accuracy on untouched test data: "**, nb\_normal.get\_accuracy(test\_labels))  
# to plot the digits mean for all 10 digits.  
nb\_normal.plot\_all\_digits\_mean()  
  
# use Naive Bayes Normal D to train and predict on stretched test images:  
nb\_normal\_stretched = NaiveBayesNormalDistr(1e-1)  
nb\_normal\_stretched.train(stretched\_image, labels)  
\_ = nb\_normal\_stretched.predict(stretched\_test\_image)  
print(**"Naive Bayes - normal distribution accuracy on stretched test data: "**, nb\_normal\_stretched.get\_accuracy(test\_labels))  
  
# use Naive Bayes Bernoulli to train and predict on untouched test images:  
nb\_bernoulli = NaiveBayesBernoulli()  
nb\_bernoulli.train(images, labels)  
\_ = nb\_bernoulli.predict(test\_images)  
print(**"Naive Bayes - bernoulli accuracy on untouched test data: "**, nb\_bernoulli.get\_accuracy(test\_labels))  
# use Naive Bayes Bernoulli to train and predict on stretched test images:  
nb\_bernoulli\_stretched = NaiveBayesBernoulli()  
nb\_bernoulli\_stretched.train(stretched\_image, labels)  
\_ = nb\_bernoulli\_stretched.predict(stretched\_test\_image)  
print(**"Naive Bayes - bernoulli accuracy on stretched test data: "**, nb\_bernoulli\_stretched.get\_accuracy(test\_labels))  
  
# RANDOM FOREST - UNTOUCHED TEST DATA  
# use Random forest with setting of trees = 10 and depth = 4  
print(**"Untouched test data - Random Forest (10 Trees, 4 Depth)"**, train\_and\_validate\_randomforest(images, labels, test\_images, test\_labels, 10, 4))  
# use Random forest with setting of trees = 10 and depth = 16  
print(**"Untouched test data - Random Forest (10 Trees, 16 Depth)"**, train\_and\_validate\_randomforest(images, labels, test\_images, test\_labels, 10, 16))  
# use Random forest with setting of trees = 30 and depth = 4  
print(**"Untouched test data - Random Forest (30 Trees, 4 Depth)"**, train\_and\_validate\_randomforest(images, labels, test\_images, test\_labels, 30, 4))  
# use Random forest with setting of trees = 30 and depth = 16  
print(**"Untouched test data - Random Forest (30 Trees, 16 Depth)"**, train\_and\_validate\_randomforest(images, labels, test\_images, test\_labels, 30, 16))  
  
# RANDOM FOREST - STRETCHED TEST DATA  
# use Random forest with setting of trees = 10 and depth = 4  
print(**"Stretched test data - Random Forest (10 Trees, 4 Depth)"**, train\_and\_validate\_randomforest(stretched\_image, labels, stretched\_test\_image, test\_labels, 10, 4))  
# use Random forest with setting of trees = 10 and depth = 16  
print(**"Stretched test data - Random Forest (10 Trees, 16 Depth)"**, train\_and\_validate\_randomforest(stretched\_image, labels, stretched\_test\_image, test\_labels, 10, 16))  
# use Random forest with setting of trees = 30 and depth = 4  
print(**"Stretched test data - Random Forest (30 Trees, 4 Depth)"**, train\_and\_validate\_randomforest(stretched\_image, labels, stretched\_test\_image, test\_labels, 30, 4))  
# use Random forest with setting of trees = 30 and depth = 16  
print(**"Stretched test data - Random Forest (30 Trees, 16 Depth)"**, train\_and\_validate\_randomforest(stretched\_image, labels, stretched\_test\_image, test\_labels, 30, 16))  
  
####################################################  
# The following predict over TRAIN data #  
####################################################  
# use Naive Bayes Normal D to train and predict on untouched test images:  
nb\_normal = NaiveBayesNormalDistr(1e-1)  
nb\_normal.train(images, labels)  
\_ = nb\_normal.predict(images)  
print(**"Naive Bayes - normal distribution accuracy on untouched training data: "**, nb\_normal.get\_accuracy(labels))  
# to plot the digits mean for all 10 digits.  
nb\_normal.plot\_all\_digits\_mean()  
  
# use Naive Bayes Normal D to train and predict on stretched test images:  
nb\_normal\_stretched = NaiveBayesNormalDistr(1e-1)  
nb\_normal\_stretched.train(stretched\_image, labels)  
\_ = nb\_normal\_stretched.predict(stretched\_image)  
print(**"Naive Bayes - normal distribution accuracy on stretched training data: "**, nb\_normal\_stretched.get\_accuracy(labels))  
  
# use Naive Bayes Bernoulli to train and predict on untouched test images:  
nb\_bernoulli = NaiveBayesBernoulli()  
nb\_bernoulli.train(images, labels)  
\_ = nb\_bernoulli.predict(images)  
print(**"Naive Bayes - bernoulli accuracy on untouched training data: "**, nb\_bernoulli.get\_accuracy(labels))  
# use Naive Bayes Bernoulli to train and predict on stretched test images:  
nb\_bernoulli\_stretched = NaiveBayesBernoulli()  
nb\_bernoulli\_stretched.train(stretched\_image, labels)  
\_ = nb\_bernoulli\_stretched.predict(stretched\_image)  
print(**"Naive Bayes - bernoulli accuracy on stretched training data: "**, nb\_bernoulli\_stretched.get\_accuracy(labels))  
  
# RANDOM FOREST - UNTOUCHED TRAIN DATA  
# use Random forest with setting of trees = 10 and depth = 4  
print(**"Untouched Training data - Random Forest (10 Trees, 4 Depth)"**, train\_and\_validate\_randomforest(images, labels, images, labels, 10, 4))  
# use Random forest with setting of trees = 10 and depth = 16  
print(**"Untouched Training data - Random Forest (10 Trees, 16 Depth)"**, train\_and\_validate\_randomforest(images, labels, images, labels, 10, 16))  
# use Random forest with setting of trees = 30 and depth = 4  
print(**"Untouched Training data - Random Forest (30 Trees, 4 Depth)"**, train\_and\_validate\_randomforest(images, labels, images, labels, 30, 4))  
# use Random forest with setting of trees = 30 and depth = 16  
print(**"Untouched Training data - Random Forest (30 Trees, 16 Depth)"**, train\_and\_validate\_randomforest(images, labels, images, labels, 30, 16))  
  
# RANDOM FOREST - STRETCHED TRAIN DATA  
# use Random forest with setting of trees = 10 and depth = 4  
print(**"Stretched Training data - Random Forest (10 Trees, 4 Depth)"**, train\_and\_validate\_randomforest(stretched\_image, labels, stretched\_image, labels, 10, 4))  
# use Random forest with setting of trees = 10 and depth = 16  
print(**"Stretched Training data - Random Forest (10 Trees, 16 Depth)"**, train\_and\_validate\_randomforest(stretched\_image, labels, stretched\_image, labels, 10, 16))  
# use Random forest with setting of trees = 30 and depth = 4  
print(**"Stretched Training data - Random Forest (30 Trees, 4 Depth)"**, train\_and\_validate\_randomforest(stretched\_image, labels, stretched\_image, labels, 30, 4))  
# use Random forest with setting of trees = 30 and depth = 16  
print(**"Stretched Training data - Random Forest (30 Trees, 16 Depth)"**, train\_and\_validate\_randomforest(stretched\_image, labels, stretched\_image, labels, 30, 16))