**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 of the feature 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 skipped these features which values is NaN.

**Part 1 Accuracies**

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

| **Setup** | **Cross-validation Accuracy** |
| --- | --- |
| Unprocessed data | 75.45 |
| 0-value elements ignored | 73.03 |

**Part 1 Code Snippets**

1. Calculation of distribution parameters

def mean(records):  
 return sum(records) / float(len(records))  
  
def standard\_deviation(average, records):  
 variance = sum([pow(x - average, 2) for x in records]) / float(len(records) - 1)  
 return math.sqrt(variance)

1. Calculation of naive Bayes predictions

# Source: https://stattrek.com/probability-distributions/normal.aspx  
def estimate\_posterior(x, mean, std):  
 exponent = math.exp(-(math.pow(float(x) - mean, 2) / (2 \* math.pow(std, 2))))  
 return (1 / (math.sqrt(2 \* math.pi) \* std)) \* exponent  
  
def estimate\_probabilities\_per\_class(distribution\_parameters, test\_vector):  
 probabilities = {}  
 for label, dist\_values in distribution\_parameters.items():  
 probabilities[label] = 1 #default it to 1  
 for i in range(len(dist\_values)):  
 mean, std = dist\_values[i]  
 X = test\_vector[i]  
 if math.isnan(X):  
 continue  
 probabilities[label] = probabilities[label] \* estimate\_posterior(X, mean, std)  
 return probabilities

def predict(distribution\_parameters, test\_vector):  
 probabilities = estimate\_probabilities\_per\_class(distribution\_parameters, test\_vector)  
 predicted\_label = None  
 max\_probability = -1  
  
 for label, probability in probabilities.items():  
 if probability > max\_probability:  
 max\_probability = probability  
 predicted\_label = label  
 return predicted\_label

1. Test-train split code

def split\_dataframe(dataframe, train\_ratio):  
 dataframe = dataframe.iloc[np.random.permutation(len(dataframe))]  
  
 trainset\_len = int(len(dataframe)\*train\_ratio)  
 trainset = dataframe.iloc[0:trainset\_len , : ]  
 testset = dataframe.iloc[trainset\_len:len(dataframe) , : ]  
   
 return trainset,testset

**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 RandomForest 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 modification or data cleaning over the original image pixels) and bounded box stretched images(cleaned images in 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 | 80.25% | 79.62% |
| 2 | Gaussian + stretched | 83.95% | 83.82% |
| 3 | Bernoulli + untouched | 83.85% | 83.23% |
| 4 | Bernoulli + stretched | 84.83% | 83.15% |
| 5 | 10 trees + 4 depth + untouched | 71.02% | 66.31% |
| 6 | 10 trees + 4 depth + stretched | 75.32% | 72.01% |
| 7 | 10 trees + 16 depth + untouched | 93.88% | 93.56% |
| 8 | 10 trees + 16 depth + stretched | 95.17% | 94.50% |
| 9 | 30 trees + 4 depth + untouched | 77.84% | 74.32% |
| 10 | 30 trees + 4 depth + stretched | 79.56% | 74.17% |
| 11 | 30 trees + 16 depth + untouched | 95.36% | 95.42% |
| 12 | 30 trees + 16 depth + stretched | 96.18% | 96.04% |

**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

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

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: Please see the full code in the end of the pdf.
* Calculation of a decision tree predictions: Please see the full code in the end of the pdf.

**Full Code of Problem 1**

# -\*- coding: utf-8 -\*-  
  
  
import numpy as np  
import math  
import pandas as pd  
  
  
#Generic enough to work for multiclass datasets  
def NaiveBayes(dataset, n):  
 accuracies = []  
 for i in range(n):  
   
 X\_train, X\_test = split\_dataframe(dataset, 0.8)  
  
 class\_division = {}  
  
 for index, feature\_vector in X\_train.iterrows():  
 feature\_vector=list(feature\_vector)  
 if feature\_vector[8] not in class\_division:  
 class\_division[feature\_vector[8]] = []   
 class\_division[feature\_vector[8]].append(feature\_vector)  
   
 mean\_std\_tuples = {}  
 for label, instances in class\_division.items():  
 mean\_std\_tuples[label] = group\_mean\_std\_tuples(instances)  
   
 predictions = bulk\_predict(mean\_std\_tuples, X\_test)  
 #print(predictions)  
 accuracy = estimate\_accuracy(X\_test, predictions)  
 accuracies.append(accuracy)  
 return accuracies  
  
def group\_mean\_std\_tuples(dataset):  
 mean\_std\_tuples = []  
 for vector in zip(\*dataset):  
 vector = clear\_nan\_values(vector)  
 average = mean(vector)  
 std = standard\_deviation(average, vector)  
 mean\_std\_tuples.append((average,std))  
 del mean\_std\_tuples[-1] #remove the last label column summaries  
 return mean\_std\_tuples  
  
def mean(records):  
 return sum(records) / float(len(records))  
  
def standard\_deviation(average, records):  
 variance = sum([pow(x - average, 2) for x in records]) / float(len(records) - 1)  
 return math.sqrt(variance)  
  
# choose y with largest sum(log(p(Xi|y))) + log(p(y))   
# Normal distribution for number  
# Y = { 1/[ σ \* sqrt(2π) ] } \* e-(x - μ)2/2σ2  
# Source: https://stattrek.com/probability-distributions/normal.aspx  
def estimate\_posterior(x, mean, std):  
 exponent = math.exp(-(math.pow(float(x) - mean, 2) / (2 \* math.pow(std, 2))))  
 return (1 / (math.sqrt(2 \* math.pi) \* std)) \* exponent  
  
# for every feature value and the corresponding mean and std use the respective distribution to   
# caculate probabilities of X to belong to any of the chosen classes  
def estimate\_probabilities\_per\_class(distribution\_parameters, test\_vector):  
 probabilities = {}  
 for label, dist\_values in distribution\_parameters.items():  
 probabilities[label] = 1 #default it to 1  
 for i in range(len(dist\_values)):  
 mean, std = dist\_values[i]  
 X = test\_vector[i]  
 if math.isnan(X):  
 continue  
 probabilities[label] = probabilities[label] \* estimate\_posterior(X, mean, std)  
 return probabilities  
  
#shuffle the dataframe and take the ration\*data len for training and the rest for testing  
def split\_dataframe(dataframe, train\_ratio):  
 dataframe = dataframe.iloc[np.random.permutation(len(dataframe))]  
  
 trainset\_len = int(len(dataframe)\*train\_ratio)  
 trainset = dataframe.iloc[0:trainset\_len , : ]  
 testset = dataframe.iloc[trainset\_len:len(dataframe) , : ]  
   
 return trainset,testset  
   
   
def clear\_nan\_values(vector):  
 new\_vector = []  
 for number in vector:  
 if not math.isnan(number):  
 new\_vector.append(number)  
 return new\_vector  
  
# For given test vector and estimated parameters find the class with the highest probability and assign it to this vector  
def predict(distribution\_parameters, test\_vector):  
 probabilities = estimate\_probabilities\_per\_class(distribution\_parameters, test\_vector)  
 predicted\_label = None  
 max\_probability = -1  
  
 for label, probability in probabilities.items():  
 if probability > max\_probability:  
 max\_probability = probability  
 predicted\_label = label  
 return predicted\_label  
  
#Caclucates the prediction per every row in the test set based on trained data  
def bulk\_predict(distribution\_parameters, test\_set):  
 predictions = [predict(distribution\_parameters, list(feature\_test\_vector)) for index, feature\_test\_vector in test\_set.iterrows()]  
 return predictions  
  
# Estimate % of correct results  
def estimate\_accuracy(test\_set, results):  
 correct = 0  
 count = 0  
 for index, feature\_test\_vector in test\_set.iterrows():  
 feature\_test\_vector = clear\_nan\_values(feature\_test\_vector)  
 if feature\_test\_vector[-1] == results[count]:  
 correct += 1  
 count = count+ 1  
 return (correct / float(len(test\_set))) \* 100.0  
  
  
# Part 1  
pima\_indians = pd.read\_csv('pima-indians-diabetes.csv')  
  
print(mean(NaiveBayes(pima\_indians, 10)))  
  
# Part 2  
dataset\_manipulated = pima\_indians.values  
  
for i in [2, 3, 5, 7]:  
 for j in range(len(dataset\_manipulated)):  
 if dataset\_manipulated[j][i] == 0:  
 dataset\_manipulated[j][i] = np.nan  
  
result=NaiveBayes(pd.DataFrame(dataset\_manipulated),10)  
print(mean(result))  
  
 # Making the Confusion Matrix  
#from sklearn.metrics import confusion\_matrix  
#cm = confusion\_matrix(y\_test, y\_pred)

**Full Code of Problem 2**