Homework Assignment No. 02:

HW No. 02: Bayesian Decision Theory

submitted to:

Professor Joseph Picone
ECE 8527: Introduction to Pattern Recognition and Machine Learning
Temple University
College of Engineering
1947 North 12th Street
Philadelphia, Pennsylvania 19122

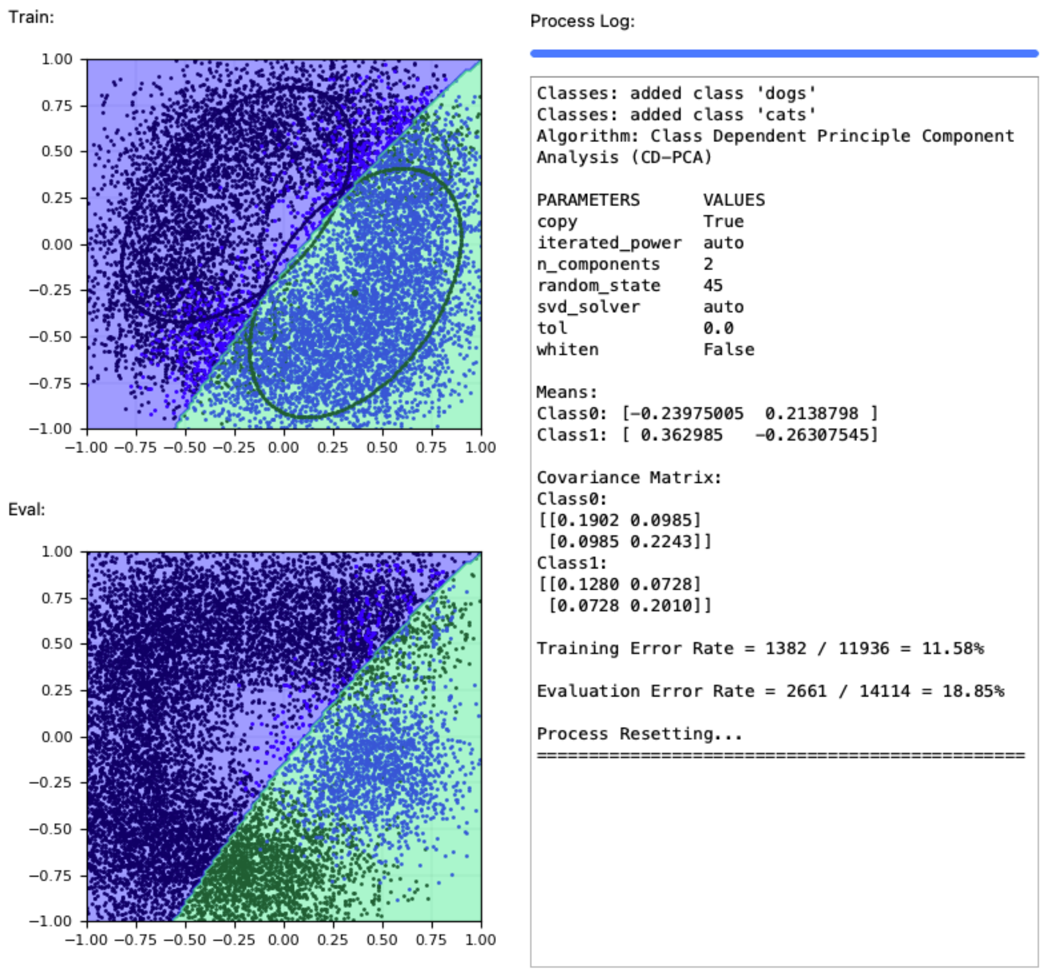
January 30th, 2022

prepared by:

Gavin Koma
Email: gavintkoma@temple.edu

A. IMLD.PY

imld.py is a custom GUI that was designed by Dr. Picone’s lab and offers a simple and effective way to implement various machine learning algorithms. The first task of this assignment is to utilize this executable and perform “class-dependent principal components analysis”. Doing so, provides a window that looks like this:

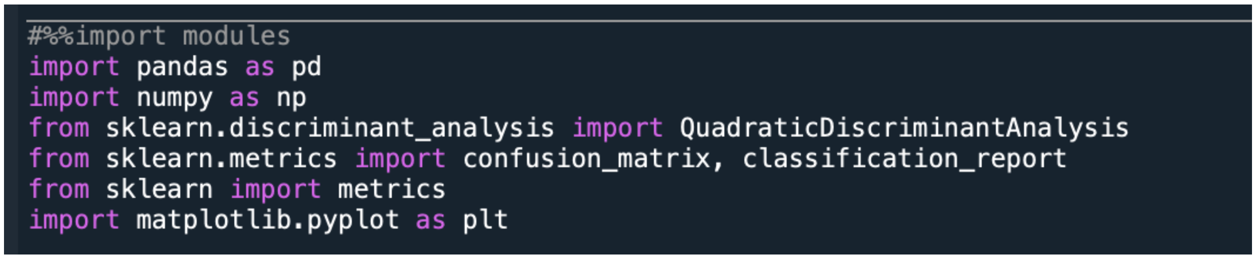


```
Classes: added class 'dogs'
Classes: added class 'cats'
Algorithm: Class Dependent Principle Component
Analysis (CD-PCA)

PARAMETERS      VALUES
copy            True
iterated_power  auto
n_components    2
random_state    45
svd_solver      auto
tol             0.0
whiten          False

Means:
Class0: [-0.23975005  0.2138798 ]
Class1: [ 0.362985   -0.26307545]

Covariance Matrix:
Class0:
[[0.1902 0.0985]
 [0.0985 0.2243]]
Class1:
[[0.1280 0.0728]
 [0.0728 0.2010]]

Training Error Rate = 1382 / 11936 = 11.58%

Evaluation Error Rate = 2661 / 14114 = 18.85%

Process Resetting...
==============================================
```

One can see the training error rate and the evaluation error rate which are 11.58% and 18.85%, respectively. In order to maintain similarity between imld.py and my own python scripts, I have altered the report and instead of reporting the miscalculation rate, I have included the percent of properly classified data. This was done by subtracting the error rate from 100. Thus, my accuracy rate was 88.42% and 81.15% for the training data and evaluation data.

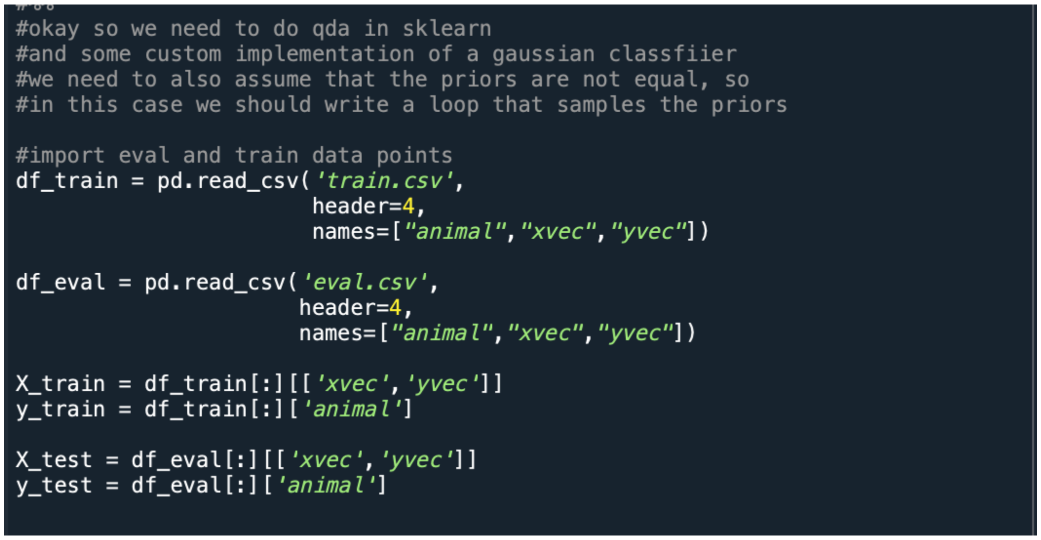
B. QDA PERFORMED IN SKLEARN, JMP PRO 16 AND A HARDCODED QDA CLASSIFIER

The first step of any coding project requires the implementation of the proper python modules. To complete this assignment, I utilized pandas, numpy, various portions of SKLearn, and matplotlib for basic plotting.



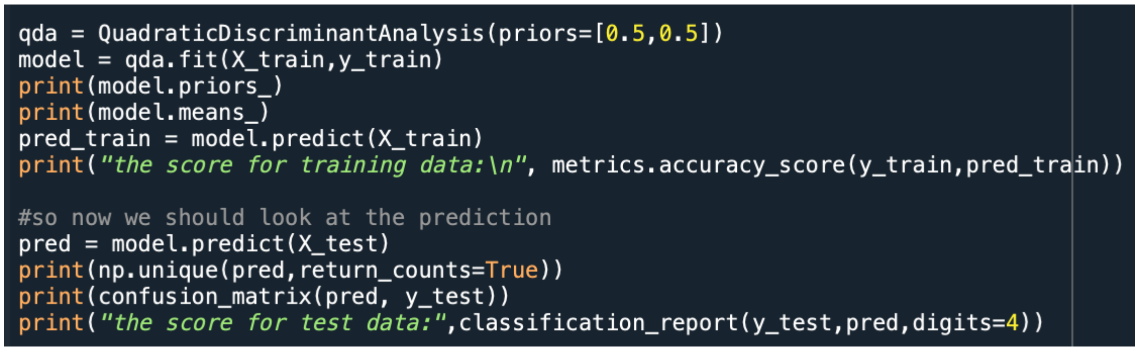
```
#%%import modules
import pandas as pd
import numpy as np
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.metrics import confusion_matrix, classification_report
from sklearn import metrics
import matplotlib.pyplot as plt
```

In addition to proper modules, the data must first be imported into one's workspace. SKLearn offers great methods for splitting data into test and evaluation sets. Luckily, we did not need to use this tool as our data was provided in two separate data files: train.csv and eval.csv. I did, however, decide to read in my csv files into a manner that allowed me to easily differentiate between train and test data. This is not a necessary step, but it makes the data easier for me to work.



```
#okay so we need to do qda in sklearn
#and some custom implementation of a gaussian classfiier
#we need to also assume that the priors are not equal, so
#in this case we should write a loop that samples the priors

#import eval and train data points
df_train = pd.read_csv('train.csv',
                       header=4,
                       names=["animal","xvec","yvec"])

df_eval = pd.read_csv('eval.csv',
                      header=4,
                      names=["animal","xvec","yvec"])

X_train = df_train[:][['xvec','yvec']]
y_train = df_train[:]['animal']

X_test = df_eval[:][['xvec','yvec']]
y_test = df_eval[:]['animal']
```

The assignment specifies that we should assume that the priors are equal in this case and therefore require us to assign the priors of dogs and cats to be [0.5,0.5]. I also have a variety of print statements that I use to check my own work as I progress, this print statements can be ignored if desired. The assignment also dictates that the error rate be reported for *both* evaluation data and training data.



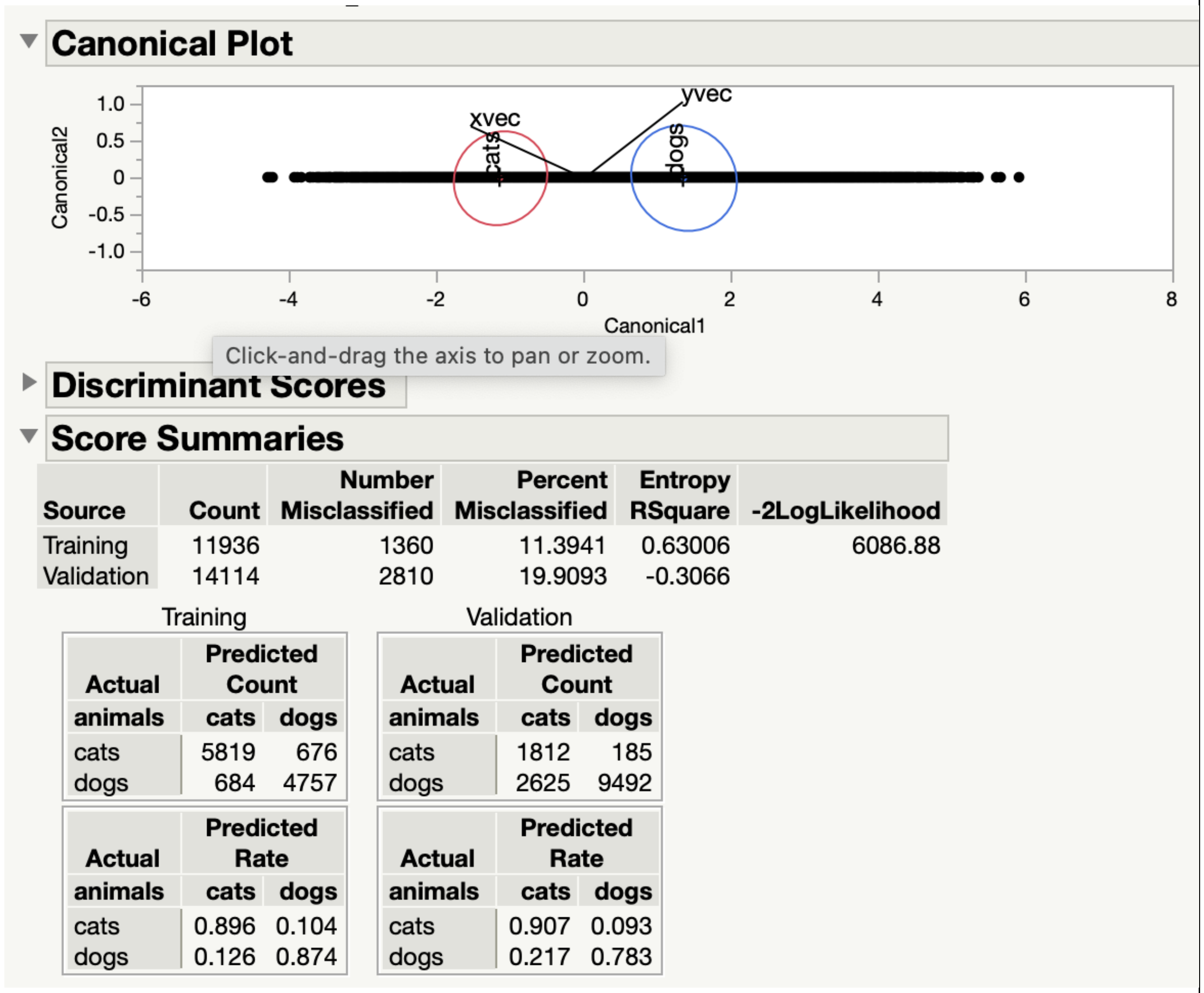
```
qda = QuadraticDiscriminantAnalysis(priors=[0.5,0.5])
model = qda.fit(X_train,y_train)
print(model.priors_)
print(model.means_)
pred_train = model.predict(X_train)
print("the score for training data:\n", metrics.accuracy_score(y_train,pred_train))

#so now we should look at the prediction
pred = model.predict(X_test)
print(np.unique(pred,return_counts=True))
print(confusion_matrix(pred, y_test))
print("the score for test data:",classification_report(y_test,pred,digits=4))
```

One can see that using QDA in SKLearn only requires three lines of code. The first is the calling of the quadratic discriminant analysis with predefined priors. We then use the x vector and y vector from our training data to fit the model. Our prediction for training is the third line of code where we have our model predict for the training data. To predict for the test data, we provide out model with the X_test vector.

To obtain the scores, we include the print functions seen above. I specified a string within each print statement to better determine which number corresponded to the desired output in my terminal. SKLearn outputted a score of 88.61% and 80.09% for training data and evaluation data, respectively.

Performing the same procedure in JMP Pro is notably easier; however, I am unsure of its effectiveness when working with a dataset that has not already been organized to facilitate such a procedure. In order to perform this, one must load their data and then go to Analyze > Multivariate Methods > Discriminant. When the discriminant pop-up shows, it is important to change the Discriminant method which is by default, Linear, Common Covariance to Quadratic, Different Covariances; as per our assignment instructions. At this point, your continuous vectors will be placed in the Covariates box while the animal category will be placed in Categories. Your outputs for training data and evaluation data will be as follows:

Training & Evaluation Data:



Score Summaries

Source	Count	Number Misclassified	Percent Misclassified	Entropy RSquare	-2LogLikelihood
Training	11936	1360	11.3941	0.63006	6086.88
Validation	14114	2810	19.9093	-0.3066	

Training

Actual	Predicted Count	
animals	cats	dogs
cats	5819	676
dogs	684	4757

Actual	Predicted Rate	
animals	cats	dogs
cats	0.896	0.104
dogs	0.126	0.874

Validation

Actual	Predicted Count	
animals	cats	dogs
cats	1812	185
dogs	2625	9492

Actual	Predicted Rate	
animals	cats	dogs
cats	0.907	0.093
dogs	0.217	0.783

The final purpose of this assignment was to apply Bayes rule directly through Gaussian models and full covariance matrices. This is essentially a task to ensure that we understand the workings of a Quadratic Discriminant Analysis (QDA) classification algorithm and to gain a better understanding of the mathematics required to perform said algorithm.

Linear Discriminant Analysis (LDA) and Quadratic Discriminant Analysis are quite similar to each other, but the primary difference is that QDA assumes that each class has its own full covariance matrix while LDA assumes that there is some *common* matrix that is common to all classes of the data. It is very important to recall that QDA does assume that each class we are working with follows a Gaussian distribution. In order to perform QDA by hand, we are required to obtain three values: the class-specific prior, the class-specific mean vector, and the class-specific covariance matrix.

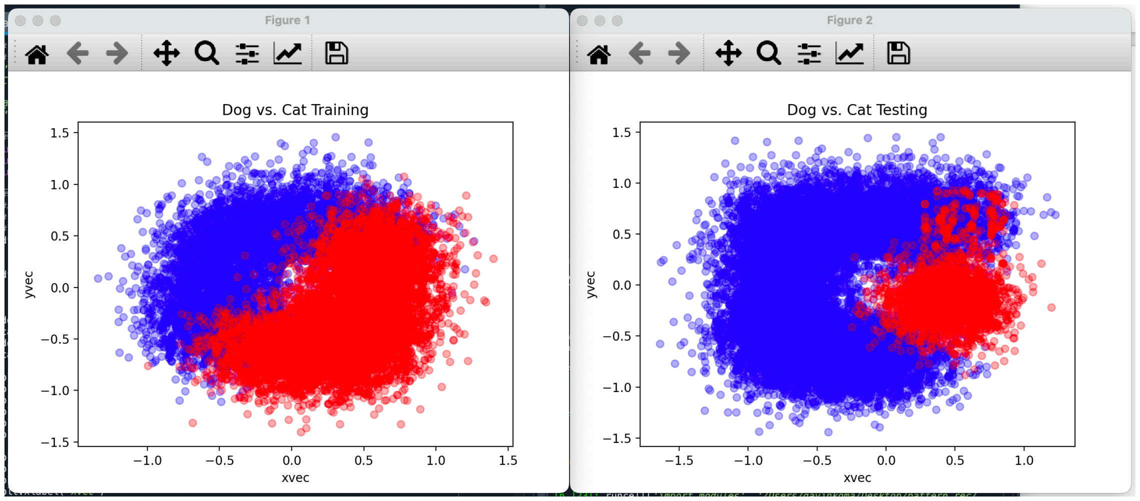
Let's take a moment to define these for future reference and ease in studying for exams at a later date:

Class-specific prior – The proportion of data points that belong to the class.

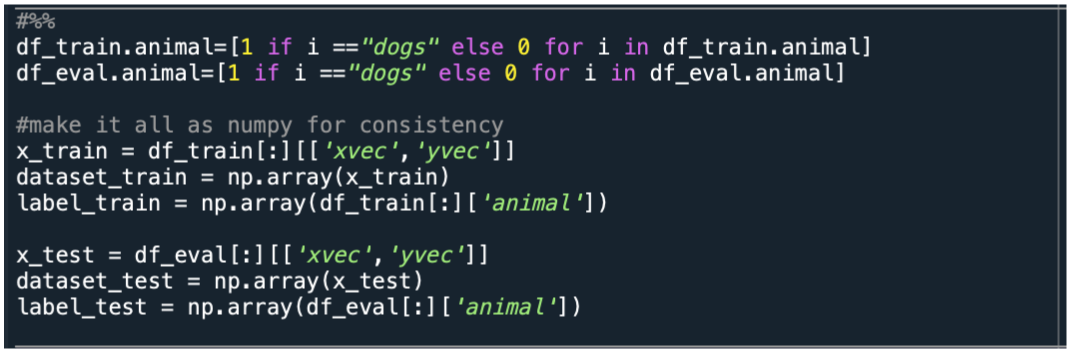
Class-specific mean vector – The average of the input variables belonging to the class.

Class-specific covariance matrix – The covariance of the vectors that belong to a class.

To begin this portion of the assignment, I imported only three, very common, python libraries: numpy, pandas, and matplotlib. Thus, allowing me to perform basic data manipulation, maths, and simple plots for data visualization. I first elected to visualize the data using simple scatterplots but did chose not to include this python code because it is relatively simple. The purpose of this was to ensure that there was no accidental alterations in my data and that it looked the same as what was visualized in imld.py.

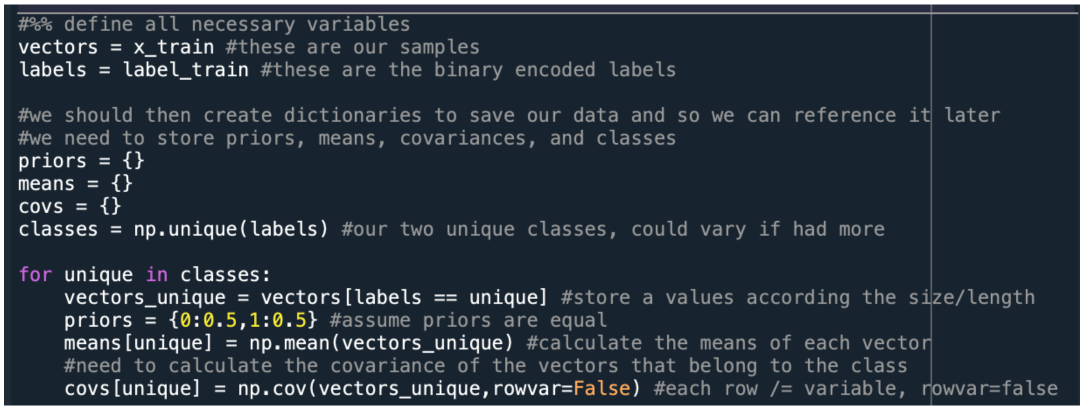


I then worked to binary encode the classes of 'cat' and 'dog' as 0, and 1, respectively. These values can be chosen at random but must stay consistent because they are numeric versions of categorical classes. This was performed with simple *if else* statements implemented via list comprehension:



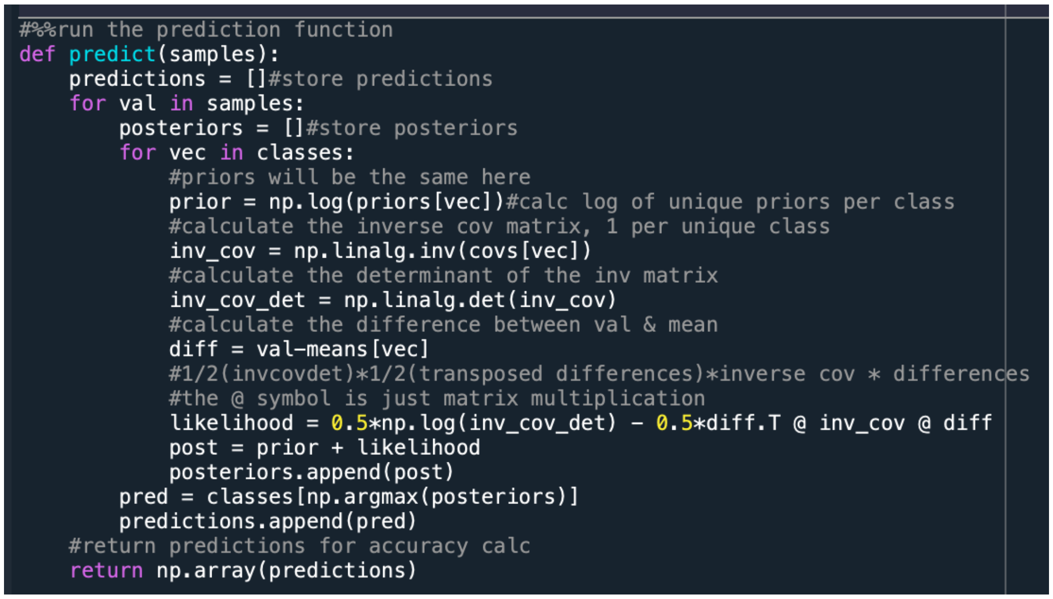
```
#%%
df_train.animal=[1 if i =="dogs" else 0 for i in df_train.animal]
df_eval.animal=[1 if i =="dogs" else 0 for i in df_eval.animal]

#make it all as numpy for consistency
x_train = df_train[:][['xvec','yvec']]
dataset_train = np.array(x_train)
label_train = np.array(df_train[:]['animal'])

x_test = df_eval[:][['xvec','yvec']]
dataset_test = np.array(x_test)
label_test = np.array(df_eval[:]['animal'])
```

Now that the data has been properly prepared, I was able to begin designing the model and preparing the matrices. I defined my samples and my labels as separate variables. I then initiated three separate dictionaries that will act to store our necessary variables. My for-loop first ensures that the number of labels (2) is equal to the number of unique classes (2). This step is not necessary to have, but I like to have multiple checks while coding to ensure proper results. I then defined priors to be equal to each other ([0.5,0.5]), calculated the means of the vectors and calculated covariances as well.



```
#%% define all necessary variables
vectors = x_train #these are our samples
labels = label_train #these are the binary encoded labels

#we should then create dictionaries to save our data and so we can reference it later
#we need to store priors, means, covariances, and classes
priors = {}
means = {}
covs = {}
classes = np.unique(labels) #our two unique classes, could vary if had more

for unique in classes:
    vectors_unique = vectors[labels == unique] #store a values according the size/length
    priors = {0:0.5,1:0.5} #assume priors are equal
    means[unique] = np.mean(vectors_unique) #calculate the means of each vector
    #need to calculate the covariance of the vectors that belong to the class
    covs[unique] = np.cov(vectors_unique,rowvar=False) #each row /= variable, rowvar=false
```

Once all of the primary variables were properly calculated, I was able to begin writing the prediction function that I would use on both the training data and the testing data. The code is attached below and is commented out in my text editor to offer explanation. I elected to include the commented code and to not also offer more written explanation for brevity's sake. The same code was run for both training data and test data and resulted in 88.51% and 80.09% accuracy, respectively.

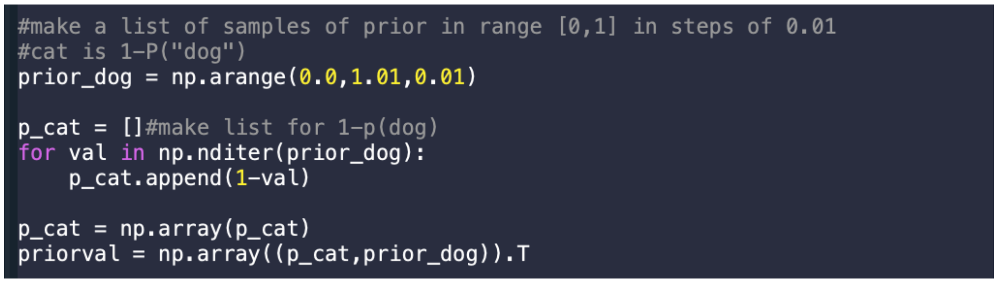


```
#%%run the prediction function
def predict(samples):
    predictions = []#store predictions
    for val in samples:
        posteriors = []#store posteriors
        for vec in classes:
            #priors will be the same here
            prior = np.log(priors[vec])#calc log of unique priors per class
            #calculate the inverse cov matrix, 1 per unique class
            inv_cov = np.linalg.inv(covs[vec])
            #calculate the determinant of the inv matrix
            inv_cov_det = np.linalg.det(inv_cov)
            #calculate the difference between val & mean
            diff = val-means[vec]
            #1/2(invcovdet)*1/2(transposed differences)*inverse cov * differences
            #the @ symbol is just matrix multiplication
            likelihood = 0.5*np.log(inv_cov_det) - 0.5*diff.T @ inv_cov @ diff
            post = prior + likelihood
            posteriors.append(post)
        pred = classes[np.argmax(posteriors)]
        predictions.append(pred)
    #return predictions for accuracy calc
    return np.array(predictions)
```

These accuracy values have continued to be the same for all of the implementations of QDA when compared to a JMP Pro implementation, an SKLearn implementation, and a hard-coded implementation. We have only seen variation between CD-PCA and QDA. The three different methods of implementing QDA make sense because the maths between implementation methods are not different.

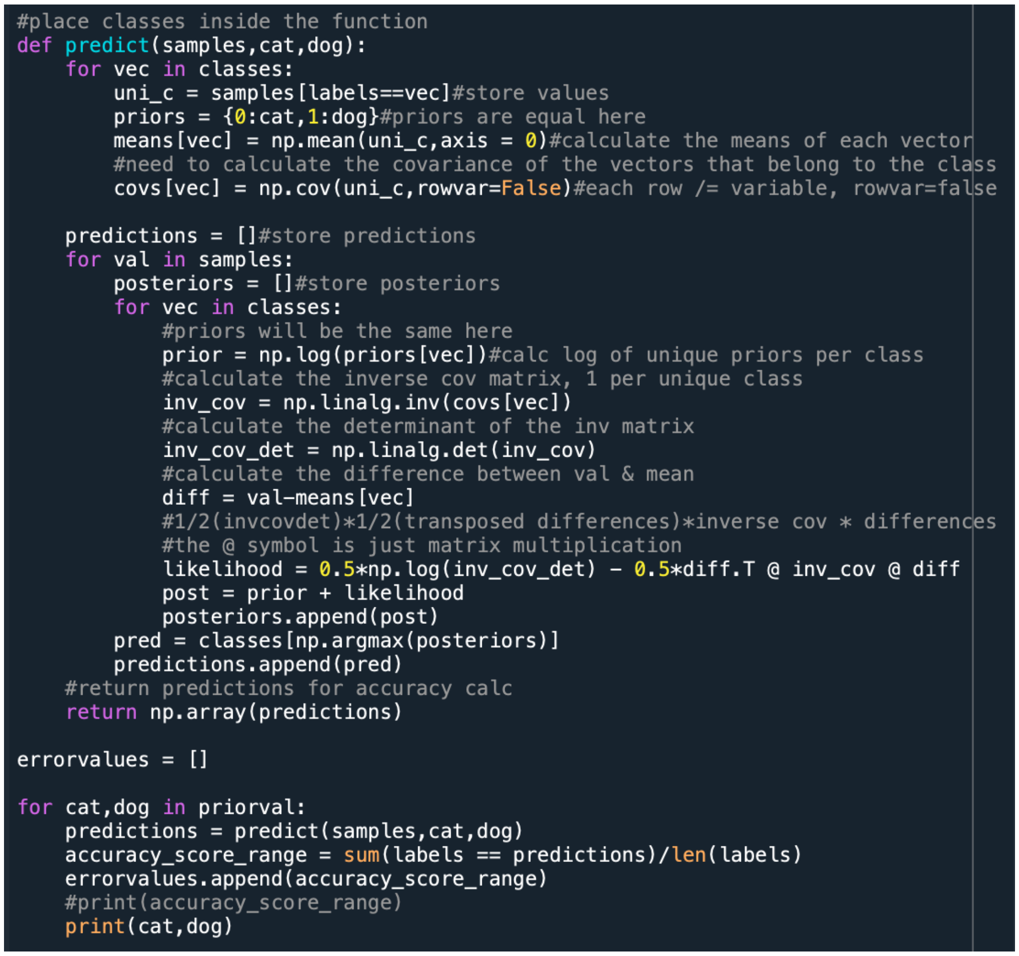
C. GRAPH COMPARISON WITH VARYING PRIORS

This task requires us to assume that the priors are not equal. To do so, I implemented a loop that populates a list of 101 values over the range of [0,1] with a step-size increment of 0.01. It was necessary to make the range command go from 0.0 to 1.01 to ensure that 1.0 was included in the final output. I then created another list for the priors of the cat (p_cat) by writing loop that will subtract each value in the dog priors list and append the given value to p_cat. Finally, I concatenated the two numpy arrays into one array and transposed the matrix to have two columns.



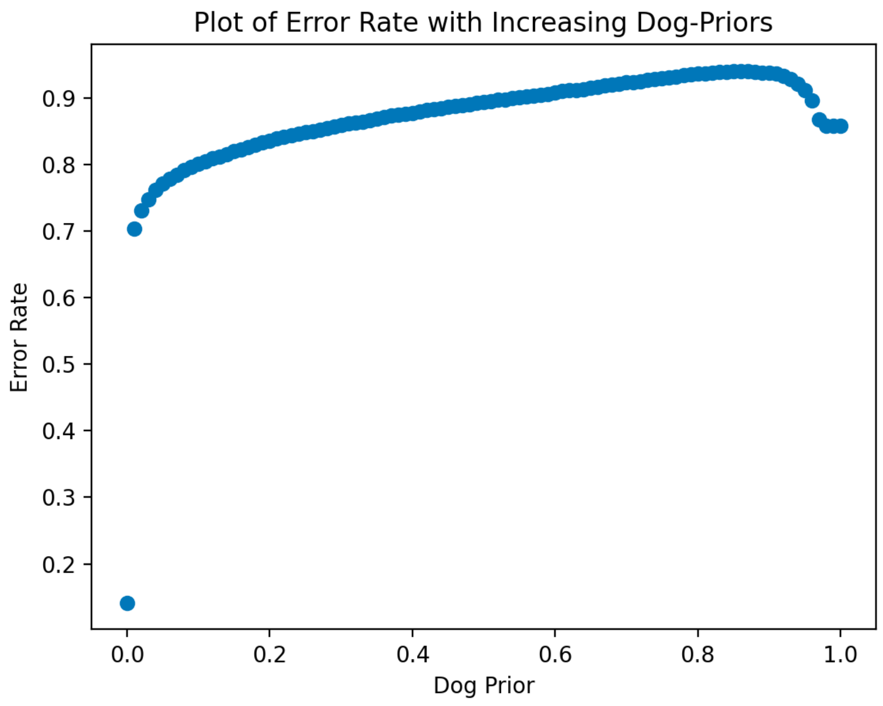
```
#make a list of samples of prior in range [0,1] in steps of 0.01
#cat is 1-P("dog")
prior_dog = np.arange(0.0,1.01,0.01)

p_cat = []#make list for 1-p(dog)
for val in np.nditer(prior_dog):
    p_cat.append(1-val)

p_cat = np.array(p_cat)
priorval = np.array((p_cat,prior_dog)).T
```

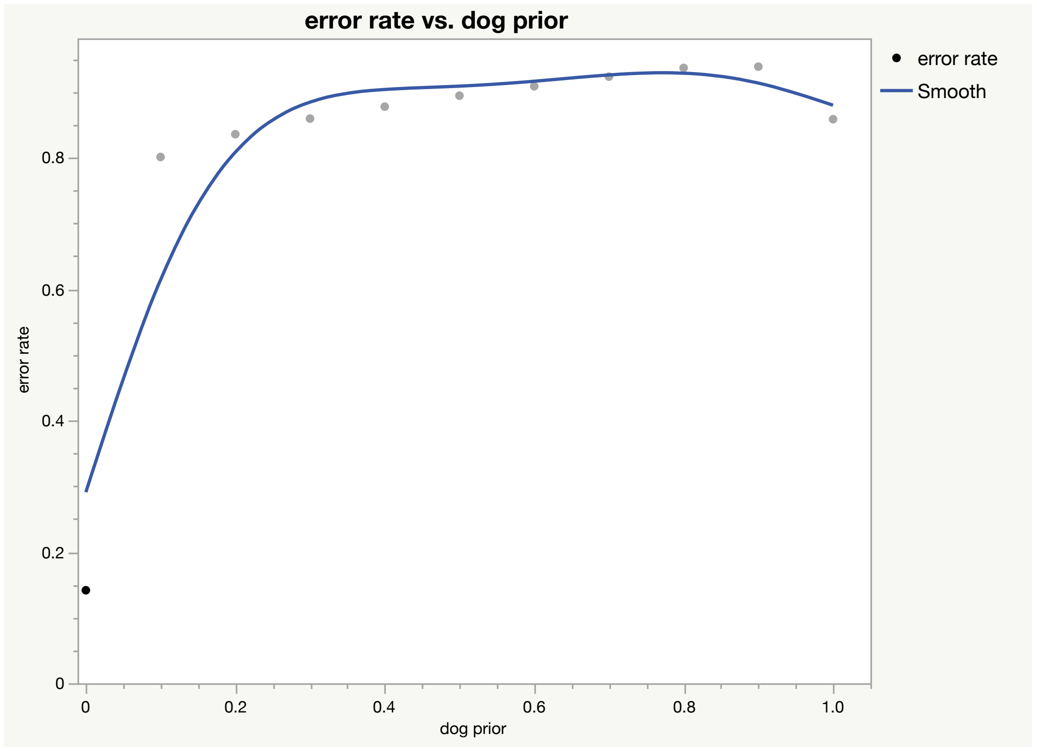
Similar to the last portion of code, the dictionaries that are populated by priors, means, covariances, and classes are placed outside of the prediction function. However, at this point, the code varies and the first for-loop is placed inside the prediction function. By doing so, I am able to iterate through the 100 prior values that are necessary for this task.



```
#place classes inside the function
def predict(samples,cat,dog):
    for vec in classes:
        uni_c = samples[labels==vec]#store values
        priors = {0:cat,1:dog}#priors are equal here
        means[vec] = np.mean(uni_c,axis = 0)#calculate the means of each vector
        #need to calculate the covariance of the vectors that belong to the class
        covs[vec] = np.cov(uni_c,rowvar=False)#each row /= variable, rowvar=false

    predictions = []#store predictions
    for val in samples:
        posteriors = []#store posteriors
        for vec in classes:
            #priors will be the same here
            prior = np.log(priors[vec])#calc log of unique priors per class
            #calculate the inverse cov matrix, 1 per unique class
            inv_cov = np.linalg.inv(covs[vec])
            #calculate the determinant of the inv matrix
            inv_cov_det = np.linalg.det(inv_cov)
            #calculate the difference between val & mean
            diff = val-means[vec]
            #1/2(invcovdet)*1/2(transposed differences)*inverse cov * differences
            #the @ symbol is just matrix multiplication
            likelihood = 0.5*np.log(inv_cov_det) - 0.5*diff.T @ inv_cov @ diff
            post = prior + likelihood
            posteriors.append(post)
        pred = classes[np.argmax(posteriors)]
        predictions.append(pred)
    #return predictions for accuracy calc
    return np.array(predictions)

errorvalues = []

for cat,dog in priorval:
    predictions = predict(samples,cat,dog)
    accuracy_score_range = sum(labels == predictions)/len(labels)
    errorvalues.append(accuracy_score_range)
    #print(accuracy_score_range)
    print(cat,dog)
```

After performing this code, a simple scatter plot is generated that will plot the error rate against the prior values for the dog class. As with all previous plots, this code has been omitted due to its simplicity. The graphs, however, are attached below and are strikingly similar:





These plots are as we would expect them to be, the priors are on the horizontal axis and rate on the vertical axis. Thereby showing the rate of errors over an increase prior value. Priors are almost always paired with likelihood to be understood or relevant for a dataset or problem. Thus, the priors are often chosen with regards to the likelihood and will therefore allow us to choose priors to use based on the rate of error in our classification. These graphs make sense because they allow us to choose priors based on their ability to affect the likelihood of a better classification rate.

D. TABLE SUBMISSION

The following table was filled out during the coding process and was included below. Some data cells were not required to be filled out and instead of leaving them blank, I have elected to fill their spaces with N/A.

Table with Accuracy Rates:

		IMLD		JMP		SKLearn		Python	
Algorithm	**Data**	**Train**	**Eval**	**Train**	**Eval**	**Train**	**Eval**	**Train**	**Eval**
CD-PCA	Set No. 13	88.42%	81.15%	N/A	N/A	N/A	N/A	N/A	N/A
QDA	Set No. 13	N/A	N/A	88.61%	80.09%	88.61%	80.09%	N/A	N/A
QDA Custom	Set No. 13	N/A	N/A	N/A	N/A	N/A	N/A	88.61%	80.09%

Table with Error Rates:

		IMLD		JMP		SKLearn		Python	
Algorithm	**Data**	**Train**	**Eval**	**Train**	**Eval**	**Train**	**Eval**	**Train**	**Eval**
CD-PCA	Set No. 13	11.58%	18.85%	N/A	N/A	N/A	N/A	N/A	N/A
QDA	Set No. 13	N/A	N/A	11.39%	19.91%	11.39%	19.96%	N/A	N/A
QDA Custom	Set No. 13	N/A	N/A	N/A	N/A	N/A	N/A	11.39%	19.96%

E. SUMMARY

This assignment's goal was to further explore Bayesian Decision theory as well as dabble in Gaussian distributions. The assignment helped greatly to apply the math that we have studied in lecture. I regularly find myself struggling to make connections between high-level mathematics and real-world applications. Although I understand that this is not the main goal of this course, it greatly helps with my overall understanding.

I am particularly curious as to whether or not there is a way to perform some of these classifiers in JMP Pro if we do not have them already separated into organized datasets. As I progressed through this assignment, I was unable to see a way to apply a training dataset and a test dataset in JMP Pro, unlike in python where one is easily able to split data as necessary.

One aspect of my approach to solving these homework assignments going forward will be to implement more generalized functions. There were a variety of times where instead of writing a script that I could have defined a function and passed variables to it. I understand that as someone who would not describe themselves as "experienced" in coding that I need to start practicing these techniques to better incorporate and tidy my code.

Overall, this assignment helped to further cement the Naïve-Bayes classifier into my mind and helped to better understand some of the variations that these programs might cause in data analysis. I had not previously considered how using JMP Pro or python may result in small discrepancies that could affect future decisions with a dataset.