Data-Driven Classifiers

Neural Networks: ECE 5930

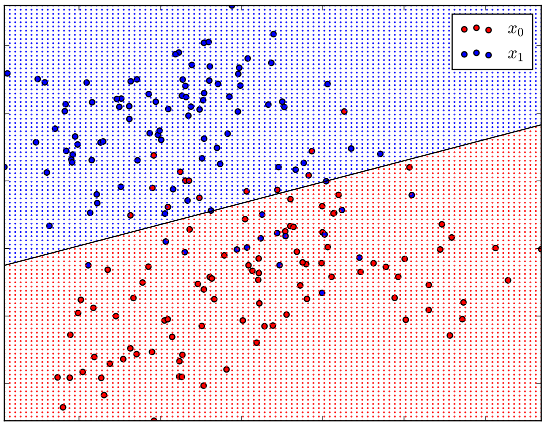

Figure: Linear Data Classifier

Clint Ferrin
Utah State University
Fri Sep 25, 2017

Contents

1 Overview **1**

2 Linear Regression **1**

Problem 1: 1

Problem 2: 1

Problem 3: 2

3 Quadratic Regression **4**

Problem 4: 4

Problem 5: 6

4 Linear and Quadratic Discriminant Analysis **7**

Problem 6: 7

Problem 7: 7

5 Linear Logistic Regression **12**

Problem 8: 12

Problem 9: 13

Problem 10: 14

6 k-nearest Neighbor Classifier **14**

Problem 13: 14

7 Naive Bayes Classifier **18**

Problem 14: 18

Problem 15: 19

8 Optimal Bayes Classifier **22**

Problem 16: 22

9 Discussion **25**

Problem 17: 25

List of Figures

1 Linear Classifier 2
2 Quadratic Regression Graph 4
3 Linear Discriminant Analysis 7
4 Quadratic Discriminant Function Classifier 8
5 k=1 Nearest Neighbor 15
6 k=5 Nearest Neighbor 15
7 k=15 Nearest Neighbor 16
8 Calculating Parzen Distribution 18
9 Bayes 1D plot of X and Y 20
10 Naive Bayes Parzen Graph 20
11 Optimal Bayes Classifier 23

List of Tables

1 Nearest Neighbor Performance Comparison 15
2 Binary Classifier Performance Comparison 26

1 Overview

This Document answers 17 questions that walk through pattern recognition on binary static data. To view the problem set and description of several of these classifiers, visit this link or navigate to the following website: `https://drive.google.com/open?id=0B5NW7S3txe5UTE0xSHJHNWxJbEE`

This document is not intended to be a comprehensive teaching document to describe each binary classifier, but rather aims to analyze the differences between them as discussed in Section 9.

2 Linear Regression

Problem 1: Show that the β that minimizes $RSS(\beta)$ is $\beta = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$.

To prove the previous statement, we will multiply the polynomial out, and find where the derivative equals zero in order to minimize β.

$$\begin{aligned}(\boldsymbol{y}-\boldsymbol{X}\beta)^T(\boldsymbol{y}-\boldsymbol{X}\beta) =& \boldsymbol{y}^T\boldsymbol{y}-\boldsymbol{y}^T\boldsymbol{X}\beta-\boldsymbol{X}^T\beta^T\boldsymbol{y}+\boldsymbol{X}^T\beta^T\boldsymbol{X}\beta \\ =& \boldsymbol{X}^T\beta^T\boldsymbol{X}\beta-2\boldsymbol{X}^T\beta^T\boldsymbol{y}+\boldsymbol{y}^T\boldsymbol{y}\end{aligned}$$

To find the minimized β we will now take the derivative and solve for β at zero.

$$\begin{aligned}\frac{d}{d\beta}\boldsymbol{X}^T\beta^T\boldsymbol{X}\beta-2\boldsymbol{X}^T\beta^T\boldsymbol{y}+\boldsymbol{y}^T\boldsymbol{y}&=0 \\ 2\boldsymbol{X}^T\boldsymbol{X}\beta-2\boldsymbol{X}^T\boldsymbol{y}&=0 \\ \boldsymbol{X}^T\boldsymbol{X}\beta&=\boldsymbol{X}^T\boldsymbol{y} \\ \beta=(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}\end{aligned} \tag{1}$$

Problem 2: Show that if the norm of $\left\|\boldsymbol{Y}-\boldsymbol{X}\hat{\boldsymbol{B}}\right\|^2$ is the Frobenius norm, then that the $\hat{\boldsymbol{B}}$ minimizing the same is determined by $\hat{\boldsymbol{B}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y}$

Given that the Frobenius Norm for a matrix with real numbers is:

$$\sqrt{Tr(\boldsymbol{A}\boldsymbol{A}^T)}$$

Then the Frobenius Norm of the problem statement is:

$$\sqrt{Tr(\boldsymbol{Y}-\boldsymbol{X}\hat{\boldsymbol{B}})(\boldsymbol{Y}-\boldsymbol{X}\hat{\boldsymbol{B}})^T} = \sqrt{Tr(\boldsymbol{X}^T\hat{\boldsymbol{B}}^T\boldsymbol{X}\hat{\boldsymbol{B}}-2\boldsymbol{X}^T\hat{\boldsymbol{B}}^T\boldsymbol{Y}+\boldsymbol{Y}^T\boldsymbol{Y})}$$

To find the $\hat{\boldsymbol{B}}$ minimizing the problem statement, we will take the deriving with respect to $\hat{\boldsymbol{B}}$

$$\begin{aligned}\frac{d}{d\hat{\boldsymbol{B}}}\left\|\sqrt{Tr(\boldsymbol{X}^T\hat{\boldsymbol{B}}^T\boldsymbol{X}\hat{\boldsymbol{B}}-2\boldsymbol{X}^T\hat{\boldsymbol{B}}^T\boldsymbol{Y}+\boldsymbol{Y}^T\boldsymbol{Y})}\right\|^2&=0 \\ 2\boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{B}}-2\boldsymbol{X}^T\boldsymbol{Y}&=0 \\ \boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{B}}&=\boldsymbol{X}^T\boldsymbol{Y}\end{aligned}$$

$$\hat{\boldsymbol{B}}=(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y} \tag{2}$$

Note that the trace could be removed from the equation because the result of the trace was zero, meaning that the sum of the trace was filled with all zeros.

Problem 3: Re-write the function `gendat2.m` into Python. Using the 100 points of training data in `classasgntrain1.dat`, write `PYTHON` code to train the coefficient matrix $\hat{\beta}$.

The program produced the desired results and the outcome can be seen in Figure 1. Note that the data is not completely linearly separable, and there were 29 errors on the wrong side of the line after it was drawn. The results of the outcome can be seen in Table 2.

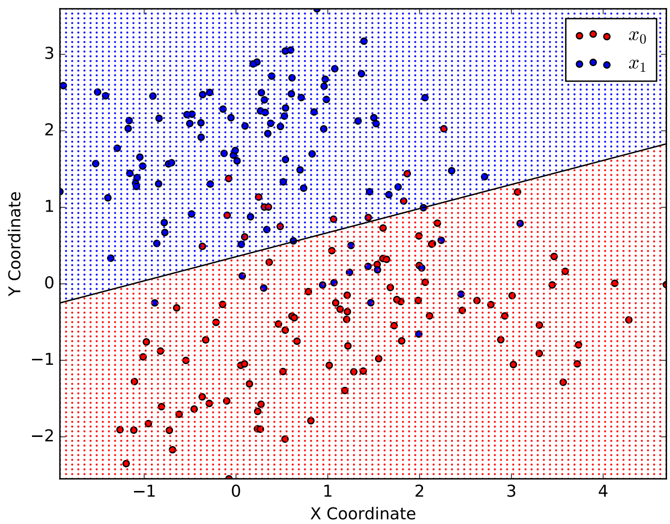

Figure 1: Linear Classifier

```python
# Clint Ferrin
# Mon Sep 25, 2017
# Linear Classifier

import sys
import numpy as np
import matplotlib.pyplot as plt

def gendata2(class_type,N):
    m0 = np.array(
         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])

    m1 = np.array(
          [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])

    x = np.array([[],[]])
    for i in range(N):
        idx = np.random.randint(10)
        if class_type == 0:
            m = m0[:,idx]
        elif class_type == 1:
            m = m1[:,idx]
        else:
            print("not a proper classifier")
            return 0
        x = np.c_[x, [[m[0]],[m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
    return x

def plotData(x0,x1):
```

```
    fig = plt.figure() # make handle to save plot
    plt.scatter(x0[0,:],x0[1,:],c='red',label='$x_0$')
    plt.scatter(x1[0,:],x1[1,:],c='blue',label='$x_1$')
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.legend()


data = np.loadtxt("../data/classasgntrain1.dat",dtype=float)
x0 = data[:,0:2].T
x1 = data[:,2:4].T
data_tot = np.c_[x0,x1]

N0 = x0.shape[1]
N1 = x1.shape[1];
N = N0 + N1

# linear regression classifier
X = np.r_[np.c_[np.ones((N0,1)),x0.T],
          np.c_[np.ones((N1,1)),x1.T]]

Y = np.r_[np.c_[np.ones((N0,1)),np.zeros((N0,1))],
          np.c_[np.zeros((N1,1)),np.ones((N1,1))]]

# find parameter matrix
Bhat = np.dot(np.linalg.inv(np.dot(X.T,X)),np.dot(X.T,Y))

# find approximate response
Yhat = np.dot(X,Bhat)
Yhathard = Yhat > 0.5

num_err = sum(sum(abs(Yhathard - Y)))/2
print("Number of errors: %d"%(num_err))

Ntest0 = 10000;
Ntest1 = 10000;

err_rate_linregress_train = float(num_err) / N
print("Percent of errors: %.4f"%(err_rate_linregress_train))

# generate the test data for class 0
xtest0 = gendata2(0,Ntest0)
xtest1 = gendata2(1,Ntest1)
num_err = 0;

for i in range(Ntest0):
    yhat = np.dot(np.r_[1,xtest0[:,i]],Bhat)
    if yhat[1] > yhat[0]:
        num_err = num_err + 1;

for i in range(Ntest1):
    yhat = np.dot(np.r_[1,xtest1[:,i]],Bhat)
    if yhat[1] < yhat[0]:
        num_err = num_err + 1;

print("Number of errors: %d"%(num_err))
err_rate_linregress_test = float(num_err) / (Ntest0 + Ntest1);
print("Percent of errors: %.4f"%(err_rate_linregress_test))


# find max and min of sets
x_tot = np.r_[x0[0,:],x1[0,:]]
y_tot = np.r_[x0[1,:],x1[1,:]]
xlim = [np.min(x_tot),np.max(x_tot)]
ylim = [np.min(y_tot),np.max(y_tot)]

# find x,y coordinate of separating line
x_cor_lin = [xlim[0],xlim[1]]
y_cor_lin = [
    (Bhat[0,0]-Bhat[0,1]+(Bhat[1,0]-Bhat[1,1])*xlim[0])
             /(Bhat[2,1]-Bhat[2,0]),

    (Bhat[0,0]-Bhat[0,1]+(Bhat[1,0]-Bhat[1,1])*xlim[1])
```

```
            /(Bhat[2,1]-Bhat[2,0])
]

# create colored graph above/below line
xp1 = np.linspace(xlim[0],xlim[1], num=100)
yp1 = np.linspace(ylim[0],ylim[1], num=100)

red_pts = np.array([[],[]])
green_pts= np.array([[],[]])

for x in xp1:
    for y in yp1:
        yhat = np.dot(np.r_[1,x,y],Bhat)
        if yhat[1] > yhat[0]:
            green_pts = np.c_[green_pts,[x,y]]
        else:
            red_pts = np.c_[red_pts,[x,y]]

plotData(x0,x1)
plt.plot(x_cor_lin,y_cor_lin,color='black')
plt.scatter(green_pts[0,:],green_pts[1,:],color='blue',s=0.25)
plt.scatter(red_pts[0,:],red_pts[1,:],color='red',s=0.25)
plt.xlim(xlim)
plt.ylim(ylim)
plt.show()
```

3 Quadratic Regression

Problem 4: For the data described in Problem 3, train the regression coefficient matrix $\hat{B}$. Determine the classification error rate on the training data and 10,000 points of test data (as before) and fill in the corresponding row of the results table. Plot the classification regions as before.

The program performed as expected, and the outcome graph can be seen in Figure 2. Note that due to the data that appears mostly linearly separable, the line does not curve much. The results of the program can be seen in Table 2.

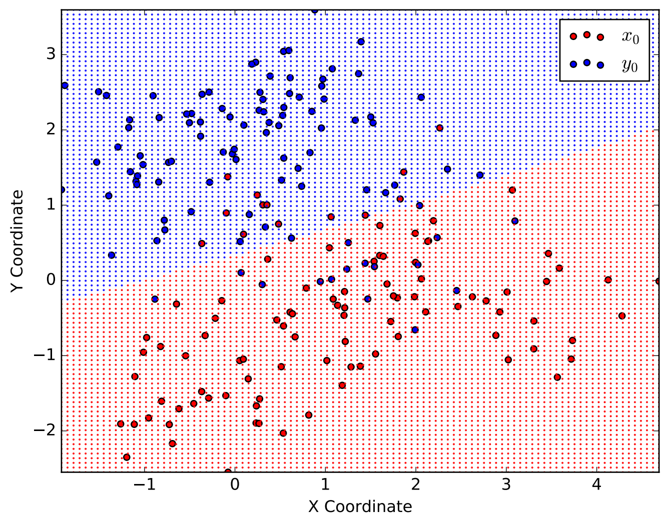

Figure 2: Quadratic Regression Graph

```
# Clint Ferrin
# Mon Sep 25, 2017
```

```
# Quadratic Classifier

import sys
import numpy as np
import matplotlib.pyplot as plt

def gendata2(class_type,N):
    m0 = np.array(
         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])

    m1 = np.array(
          [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])

    x = np.array([[],[]])
    for i in range(N):
        idx = np.random.randint(10)
        if class_type == 0:
            m = m0[:,idx]
        elif class_type == 1:
            m = m1[:,idx]
        else:
            print("not a proper classifier")
            return 0
        x = np.c_[x, [[m[0]],[m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
    return x

data = np.loadtxt("../data/classasgntrain1.dat",dtype=float)
x0 = data[:,0:2].T
x1 = data[:,2:4].T
data_tot = np.c_[x0,x1]

fig = plt.figure() # make handle to save plot
plt.scatter(x0[0,:],x0[1,:],c='red',label='$x_0$')
plt.scatter(x1[0,:],x1[1,:],c='blue',label='$y_0$')
plt.xlabel('X Coordinate')
plt.ylabel('Y Coordinate')
plt.legend()

N0 = x0.shape[1]
N1 = x1.shape[1];
N = N0 + N1

# quadratic
X = np.c_[np.ones((N,1)),data_tot.T,data_tot[0].T*data_tot[0].T, data_tot[0]*data_tot[1],
    data_tot[1]*data_tot[1]]

Y = np.r_[np.c_[np.ones((N0,1)),np.zeros((N0,1))],
          np.c_[np.zeros((N1,1)),np.ones((N1,1))]]

# find parameter matrix
Bhat = np.linalg.lstsq(np.dot(X.T,X),np.dot(X.T,Y))[0]

# find approximate response
Yhat = np.dot(X,Bhat)
Yhathard = Yhat > 0.5

num_err = sum(sum(abs(Yhathard - Y)))/2

Ntest0 = 10000;
Ntest1 = 10000;

err_rate_linregress_train = float(num_err) / N

print(err_rate_linregress_train)
# generate the test data for class 0
xtest0 = gendata2(0,Ntest0)
xtest1 = gendata2(1,Ntest1)
num_err = 0;

for i in range(Ntest0):
    yhat = np.dot(np.r_[1,xtest0[:,i],xtest0[0,i]*xtest0[0,i], xtest0[0,i]*xtest0[1,i],xtest0
```

```
        [1,i]*xtest0[1,i]],Bhat)
    if yhat[1] > yhat[0]:
        num_err = num_err + 1;

for i in range(Ntest1):
    yhat = np.dot(np.r_[1,xtest1[:,i],xtest1[0,i]*xtest1[0,i], xtest1[0,i]*xtest1[1,i],xtest1
        [1,i]*xtest1[1,i]],Bhat)
    if yhat[1] < yhat[0]:
        num_err = num_err + 1;

print("Number of errors: %d"%(num_err))

err_rate_linregress_test = float(num_err) / (Ntest0 + Ntest1);
print(err_rate_linregress_test)


# find max and min of sets
x_tot = np.r_[x0[0,:],x1[0,:]]
y_tot = np.r_[x0[1,:],x1[1,:]]
xlim = [np.min(x_tot),np.max(x_tot)]
ylim = [np.min(y_tot),np.max(y_tot)]

# create colored graph above/below line
xp1 = np.linspace(xlim[0],xlim[1], num=100)
yp1 = np.linspace(ylim[0],ylim[1], num=100)

red_pts = np.array([[],[]])
green_pts= np.array([[],[]])

for x in xp1:
    for y in yp1:
        yhat = np.dot(np.r_[1,x,y,x*x,x*y,y*y],Bhat)
        if yhat[1] > yhat[0]:
            green_pts = np.c_[green_pts,[x,y]]
        else:
            red_pts = np.c_[red_pts,[x,y]]

plt.scatter(green_pts[0,:],green_pts[1,:],color='blue',s=0.25)
plt.scatter(red_pts[0,:],red_pts[1,:],color='red',s=0.25)
plt.xlim(xlim)
plt.ylim(ylim)
plt.show()
```

Problem 5: Show that log $P(\text{class} = k|X = x) = \log\hat{\pi}_k - \frac{1}{2}\log|\hat{R}_k| - \frac{1}{2}(\boldsymbol{x} - \hat{\boldsymbol{\mu}}_k)^T\hat{R}_k^{-1}(\boldsymbol{x} - \hat{\boldsymbol{\mu}}_k)$ is true. In particular, make sure you understand what is meant by "up to a constant which does not depend on the class"

$$f_k(\boldsymbol{x}) = \frac{1}{(2\pi)^{d/2}|\hat{R}^{1/2}|}\exp[-\frac{1}{2}(\boldsymbol{x} - \hat{\boldsymbol{\mu}}_k)^T\hat{R}_k^{-1}(\boldsymbol{x} - \hat{\boldsymbol{\mu}}_k)]$$

Using Bayes rule, we can produce the following form. Note: When using Bayes Rule, constants exuding the random variable can be eliminated without affecting the results:

$$\hat{\pi}_k|\hat{R}_k|^{-1/2}\exp[-\frac{1}{2}(\boldsymbol{x} - \hat{\boldsymbol{\mu}}_k)^T\hat{R}_k^{-1}(\boldsymbol{x} - \hat{\boldsymbol{\mu}}_k)]$$

Now taking the log of the equation gives us:

$$\log\hat{\pi}_k - \frac{1}{2}\log|\hat{R}_k| - \frac{1}{2}(\boldsymbol{x} - \hat{\boldsymbol{\mu}}_k)^T\hat{R}_k^{-1}(\boldsymbol{x} - \hat{\boldsymbol{\mu}}_k) \tag{3}$$

4 Linear and Quadratic Discriminant Analysis

Problem 6: For the data set described in problem 3, build a LDA classifier. That is, train sample means for each class and population co-variance, and classify based on the linear discriminant functions in $\delta_k^l = \boldsymbol{x}^T \hat{R}^{-1} \hat{\boldsymbol{\mu}}_k - \frac{1}{2} \hat{\boldsymbol{\mu}}_k^T \hat{R}^{-1} \hat{\boldsymbol{\mu}}_k + \log \pi_k$. Characterize the error rate on the training data and on 10,000 points of test data. Plot the classification regions as before.

The graph seen in Figure 3 shows the effectiveness of the linear discriminant analysis. The error rates and results for the LDA can be seen in Table 2, and the code for the classifier is seen following Figure 4.

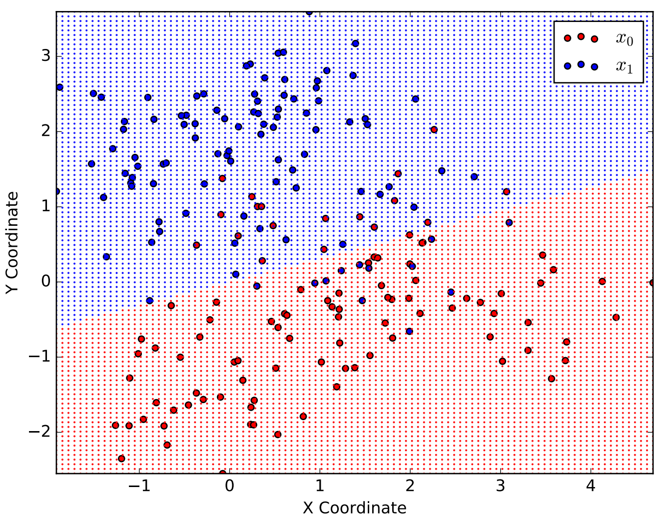

Figure 3: Linear Discriminant Analysis

Problem 7: For the data set described in problem 3, build a QDA classifier. In this case, you will also need to build the class co-variance matrices. Classify based on the quadratic discriminant functions in the equation $\log \hat{\pi}_k - \frac{1}{2} \log |\hat{R}_k| - \frac{1}{2} (\boldsymbol{x} - \hat{\boldsymbol{\mu}}_k)^T \hat{R}_k^{-1} (\boldsymbol{x} - \hat{\boldsymbol{\mu}}_k)$. Characterize the error rate on the training data and on 10,000 points of test data. Plot the classification regions as before. Compare the decision boundaries between QDA and quadratic regression.

The plotted data for Problem 7 can be seen in Figure 4 . The co-variance matrix can be seen in the second code block below, and it is referenced to as `Rhat` in the code. The decision boundaries between the LDA and QDA are significantly different; the LDA has a linear shape almost exactly the same as the Linear Regression, whereas the QDA has a steep curve towards the class1 data as seen in Figure 4. This curve can allow the classifier to be more sensitive to nonlinearities.

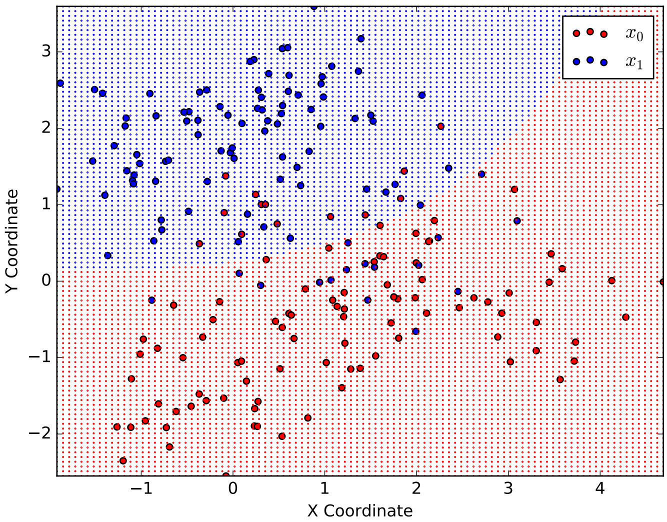

Figure 4: Quadratic Discriminant Function Classifier

```
# Clint Ferrin
# Mon Sep 25, 2017
# Linear Discriminant Analysis

import sys
import numpy as np
import matplotlib.pyplot as plt

def gendata2(class_type,N):
    m0 = np.array(
         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])

    m1 = np.array(
          [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])

    x = np.array([[],[]])
    for i in range(N):
        idx = np.random.randint(10)
        if class_type == 0:
            m = m0[:,idx]
        elif class_type == 1:
            m = m1[:,idx]
        else:
            print("not a proper classifier")
            return 0
        x = np.c_[x, [[m[0]],[m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
    return x

def getRhat(x0,x1):
    N0 = x0.shape[1]
    N1 = x1.shape[1]
    N = N0 + N1

    mu = np.array([[np.mean(x0[0,:]),np.mean(x0[1,:])],
                   [np.mean(x1[0,:]),np.mean(x1[1,:])]])

    Rhat = np.empty([2,2])
    for i in range(N0):
        Rhat = Rhat + np.outer(x0[:,i]-mu[0],x0[:,i]+mu[0])

    for i in range(N1):
        Rhat = Rhat + np.outer(x1[:,i]-mu[1],x1[:,i]+mu[1])

```

```
    return Rhat/(N-2)

def calcDel(data,Rhat,mu,N0,N):
     return np.dot(np.dot(data,np.linalg.inv(Rhat)),mu)
     - 0.5*np.dot(np.dot(mu,np.linalg.inv(Rhat)),mu)
     + np.log(N0)-np.log(N)

def getDel(x0,x1,Rhat):
    N0 = x0.shape[1]
    N1 = x1.shape[1]
    N = N0 + N1
    data_tot = np.c_[x0,x1]
    num_errors = 0
    mu = [[np.mean(x0[0,:]),np.mean(x0[1,:])],
          [np.mean(x1[0,:]),np.mean(x1[1,:])]]

    del_l = np.array([[],[]])

    for i in range(N):
        del_l = np.c_[del_l,np.array(
                [calcDel(data_tot[:,i],Rhat,mu[0],N0,N) ,
                 calcDel(data_tot[:,i],Rhat,mu[1],N0,N)]).T ]

    for i in range(N0):
        if del_l[0,i]<del_l[1,i]:
            num_errors=num_errors+1

    for i in range(N1):
        if del_l[0,N0+i]>del_l[1,N0+i]:
            num_errors=num_errors+1

    # return an array of 2 values for every point. Larger=class
    return del_l, num_errors


data = np.loadtxt("../data/classasgntrain1.dat",dtype=float)
x0 = data[:,0:2].T
x1 = data[:,2:4].T

N0 = x0.shape[1]
N1 = x1.shape[1]
N = N0 + N1

mu = np.array([[np.mean(x0[0,:]),np.mean(x0[1,:])],
               [np.mean(x1[0,:]),np.mean(x1[1,:])]])

fig = plt.figure() # make handle to save plot
plt.scatter(x0[0,:],x0[1,:],c='red',label='$x_0$')
plt.scatter(x1[0,:],x1[1,:],c='blue',label='$x_1$')
plt.xlabel('X Coordinate')
plt.ylabel('Y Coordinate')
plt.legend()

# find parameter matrix
Rhat = getRhat(x0,x1)

del_l,num_err = getDel(x0,x1,Rhat)
print("Number of Errors: %d"%(num_err))
print("Percent errors: %.4f"%(float(num_err)/N))
Ntest0 = 10000;
Ntest1 = 10000;

# generate the test data for class 0
xtest0 = gendata2(0,Ntest0)
xtest1 = gendata2(1,Ntest1)

del_l,num_err = getDel(xtest0,xtest1,Rhat)

np.savetxt('output.out', del_l)

print("Number of Errors: %d"%(num_err))
print("Percent errors: %.4f"%(float(num_err)/(Ntest0 + Ntest1)))
```

```
# find max and min of sets
x_tot = np.r_[x0[0,:],x1[0,:]]
y_tot = np.r_[x0[1,:],x1[1,:]]
xlim = [np.min(x_tot),np.max(x_tot)]
ylim = [np.min(y_tot),np.max(y_tot)]

# create colored graph above/below line
xp1 = np.linspace(xlim[0],xlim[1], num=100)
yp1 = np.linspace(ylim[0],ylim[1], num=100)

red_pts = np.array([[],[]])
green_pts= np.array([[],[]])

for x in xp1:
    for y in yp1:
        del_l =  np.array(
                [calcDel([x,y],Rhat,mu[0],N0,N),
                 calcDel([x,y],Rhat,mu[1],N1,N)])

        if del_l[0]<del_l[1]:
            green_pts = np.c_[green_pts,[x,y]]
        else:
            red_pts = np.c_[red_pts,[x,y]]

plt.scatter(green_pts[0,:],green_pts[1,:],color='blue',s=0.25)
plt.scatter(red_pts[0,:],red_pts[1,:],color='red',s=0.25)
plt.xlim(xlim)
plt.ylim(ylim)
plt.show()
```

```
# Clint Ferrin
# Mon Sep 25, 2017
# Quadratic Discriminant Analysis
import sys
import numpy as np
import matplotlib.pyplot as plt

def gendata2(class_type,N):
    m0 = np.array(
         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])

    m1 = np.array(
          [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])

    x = np.array([[],[]])
    for i in range(N):
        idx = np.random.randint(10)
        if class_type == 0:
            m = m0[:,idx]
        elif class_type == 1:
            m = m1[:,idx]
        else:
            print("not a proper classifier")
            return 0
        x = np.c_[x, [[m[0]],[m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
    return x

def getRhat(x0,x1):
    N0 = x0.shape[1]
    N1 = x1.shape[1]
    N = N0 + N1

    mu = np.array([[np.mean(x0[0,:]),np.mean(x0[1,:])],
                   [np.mean(x1[0,:]),np.mean(x1[1,:])]])

    Rhat = [np.empty([2,2]),np.empty([2,2])]
    for i in range(N0):
        Rhat[0] = Rhat[0] + np.outer(x0[:,i]-mu[0],x0[:,i]+mu[0])

    for i in range(N1):
        Rhat[1] = Rhat[1] + np.outer(x1[:,i]-mu[1],x1[:,i]+mu[1])
```

```

    return Rhat[0]/(N0-1),Rhat[1]/(N1-1)

def calcDelQDA(data,Rhat,mu,N0,N):
    return (np.log(N0)-np.log(N))-0.5*np.log(np.linalg.norm(Rhat))-0.5*np.dot(np.dot((data-mu)
        .T,np.linalg.inv(Rhat)),data-mu)

def getDel(x0,x1,Rhat):
    N0 = x0.shape[1]
    N1 = x1.shape[1]
    N = N0 + N1

    num_errors = 0

    mu = [[np.mean(x0[0,:]),np.mean(x0[1,:])],
          [np.mean(x1[0,:]),np.mean(x1[1,:])]]

    del_l = np.array([[],[]])
    for i in range(N0):
        del_l = np.c_[del_l,np.array([
            calcDelQDA(x0[:,i],Rhat[0],mu[0],N0,N),
            calcDelQDA(x0[:,i],Rhat[1],mu[1],N0,N)]).T ]

    for i in range(N1):
        del_l = np.c_[del_l,np.array([
            calcDelQDA(x1[:,i],Rhat[0],mu[0],N0,N),
            calcDelQDA(x1[:,i],Rhat[1],mu[1],N0,N)]).T ]

    for i in range(N0):
        if del_l[0,i]<del_l[1,i]:
            num_errors=num_errors+1

    for i in range(N1):
        if del_l[0,N0+i]>del_l[1,N0+i]:
            num_errors=num_errors+1

    # return an array of 2 values for every point. Larger=class
    return del_l, num_errors

# def returnBound(x,Rhat,mu):


data = np.loadtxt("../data/classasgntrain1.dat",dtype=float)
x0 = data[:,0:2].T
x1 = data[:,2:4].T

N0 = x0.shape[1]
N1 = x1.shape[1]
N = N0 + N1

mu = np.array([[np.mean(x0[0,:]),np.mean(x0[1,:])],
               [np.mean(x1[0,:]),np.mean(x1[1,:])]])


fig = plt.figure() # make handle to save plot
plt.scatter(x0[0,:],x0[1,:],c='red',label='$x_0$')
plt.scatter(x1[0,:],x1[1,:],c='blue',label='$x_1$')
plt.xlabel('X Coordinate')
plt.ylabel('Y Coordinate')
plt.legend()

# find parameter matrix
Rhat = getRhat(x0,x1)
print(Rhat)
del_l,num_err = getDel(x0,x1,Rhat)

Ntest0 = 10000;
Ntest1 = 10000;


print(float(num_err)/N)
# generate the test data for class 0
xtest0 = gendata2(0,Ntest0)
```

```
xtest1 = gendata2(1,Ntest1)

del_l,num_err = getDel(xtest0,xtest1,Rhat)
print(num_err)

print("Percent errors: %.4f"%(float(num_err)/(Ntest0 + Ntest1)))

# find max and min of sets
x_tot = np.r_[x0[0,:],x1[0,:]]
y_tot = np.r_[x0[1,:],x1[1,:]]
xlim = [np.min(x_tot),np.max(x_tot)]
ylim = [np.min(y_tot),np.max(y_tot)]

# create colored graph above/below line
xp1 = np.linspace(xlim[0],xlim[1], num=100)
yp1 = np.linspace(ylim[0],ylim[1], num=100)

red_pts = np.array([[],[]])
green_pts= np.array([[],[]])

for x in xp1:
    for y in yp1:
        del_l =  np.array(
                [calcDelQDA([x,y],Rhat[0],mu[0],N0,N),
                 calcDelQDA([x,y],Rhat[1],mu[1],N1,N)])

        if del_l[0]<del_l[1]:
            green_pts = np.c_[green_pts,[x,y]]
        else:
            red_pts = np.c_[red_pts,[x,y]]

plt.scatter(green_pts[0,:],green_pts[1,:],color='blue',s=0.25)
plt.scatter(red_pts[0,:],red_pts[1,:],color='red',s=0.25)
plt.xlim(xlim)
plt.ylim(ylim)
plt.show()
```

5 Linear Logistic Regression

Problem 8: Using the probability model $P(Y = 0|X = \boldsymbol{x}) = \frac{1}{1+\exp[-\beta^T \boldsymbol{x}]}$, show that $l(\beta)$ can be written as

$$l(\beta) = \sum_{i=1}^{N} y_i \beta^T \boldsymbol{x}_i - log(1 + e^{\beta^T \boldsymbol{x}_i})$$

The following note was necessary to remove a negative sign from the leading term of the result.

Note:

$$\frac{e^{\beta^T \boldsymbol{x}_i}}{1 + e^{\beta^T \boldsymbol{x}_i}} = \frac{1}{1 + e^{-\beta^T \boldsymbol{x}_i}}$$

We begin with the equation:

$$
\begin{aligned}
l(\beta) &= \sum_{i=1}^{N} y_i \, \log p(\boldsymbol{x}_i; \beta) + (1 - y_i) \, \log(1 - p(\boldsymbol{x}_i; \beta)) \\
&= \sum_{i=1}^{N} y_i \, \log(\frac{1}{1 + e^{-\beta^T \boldsymbol{x}_i}}) + (1 - y_i) \, \log(1 - \frac{1}{1 + e^{-\beta^T \boldsymbol{x}_i}}) \\
&= \sum_{i=1}^{N} -y_i \, \log(1 + e^{-\beta^T \boldsymbol{x}_i}) + (1 - y_i) \, \log(\frac{1 + e^{-\beta^T \boldsymbol{x}_i}}{1 + e^{-\beta^T \boldsymbol{x}_i}} - \frac{1}{1 + e^{-\beta^T \boldsymbol{x}_i}}) \\
&= \sum_{i=1}^{N} -y_i \, \log(1 + e^{-\beta^T \boldsymbol{x}_i}) + (1 - y_i) \, \log(\frac{e^{-\beta^T \boldsymbol{x}_i}}{1 + e^{-\beta^T \boldsymbol{x}_i}}) \\
&= \sum_{i=1}^{N} -y_i \, \log(1 + e^{-\beta^T \boldsymbol{x}_i}) + (1 - y_i)(\log(e^{-\beta^T \boldsymbol{x}_i}) - \log(1 + e^{-\beta^T \boldsymbol{x}_i})) \\
&= \sum_{i=1}^{N} -y_i \, \log(1 + e^{-\beta^T \boldsymbol{x}_i}) + (1 - y_i)(-\beta^T \boldsymbol{x}_i - \log(1 + e^{-\beta^T \boldsymbol{x}_i})) \\
&= \sum_{i=1}^{N} -y_i \, \log(1 + e^{-\beta^T \boldsymbol{x}_i}) - \beta^T \boldsymbol{x}_i - \log(1 + e^{-\beta^T \boldsymbol{x}_i}) + y_i \beta^T \boldsymbol{x}_i + y_i \, \log(1 + e^{-\beta^T \boldsymbol{x}_i}) \\
&= \sum_{i=1}^{N} y_i \beta^T \boldsymbol{x}_i - \beta^T \boldsymbol{x}_i - \log(1 + e^{-\beta^T \boldsymbol{x}_i}) \\
&= \sum_{i=1}^{N} y_i \beta^T \boldsymbol{x}_i - (\beta^T \boldsymbol{x}_i + \log(1 + e^{-\beta^T \boldsymbol{x}_i})) \\
&= \sum_{i=1}^{N} y_i \beta^T \boldsymbol{x}_i - (\log(e^{\beta^T \boldsymbol{x}_i}) + \log(1 + e^{-\beta^T \boldsymbol{x}_i}))
\end{aligned}
$$

$$
l(\beta) = \sum_{i=1}^{N} y_i \beta^T \boldsymbol{x}_i - \log(1 + e^{\beta^T \boldsymbol{x}_i})) \tag{4}
$$

Problem 9: Show that

$$
\frac{\partial l(\beta)}{\partial \beta} = \sum_{i=1}^{N} \boldsymbol{x}_i (y_i - p(\boldsymbol{x}_i; \beta))
$$

Starting with the equation from the previous problem.

$$
\begin{aligned}
&\frac{\partial}{\partial \beta} \sum_{i=1}^{N} y_i \beta^T \boldsymbol{x}_i - \log(1 + e^{\beta^T \boldsymbol{x}_i}) \\
&\sum_{i=1}^{N} y_i \boldsymbol{x}_i - \frac{\partial}{\partial \beta} \log(1 + e^{\beta^T \boldsymbol{x}_i}) \\
&\sum_{i=1}^{N} y_i \boldsymbol{x}_i - \frac{\boldsymbol{x}_i e^{\beta^T \boldsymbol{x}_i}}{1 + e^{\beta^T \boldsymbol{x}_i}} \\
&\sum_{i=1}^{N} y_i \boldsymbol{x}_i - \frac{\boldsymbol{x}_i e^{\beta^T \boldsymbol{x}_i}}{1 + e^{\beta^T \boldsymbol{x}_i}} \\
&\sum_{i=1}^{N} y_i \boldsymbol{x}_i - \boldsymbol{x}_i p(\boldsymbol{x}_i; \beta)
\end{aligned}
$$

$$\sum_{i=1}^{N} \boldsymbol{x}_i(y_i - p(\boldsymbol{x}_i;\beta)) \tag{5}$$

Problem 10: Show that

$$\frac{\partial^2 l(\beta)}{\partial\beta\partial\beta^T} = \sum_{i=1}^{N} \boldsymbol{x}_i\boldsymbol{x}_i^T p(\boldsymbol{x}_i;\beta)(1 - p(\boldsymbol{x}_i;\beta))$$

Beginning with the first derivative from the previous problem:

$$\begin{gathered}
\frac{\partial}{\partial\beta}\sum_{i=1}^{N} \boldsymbol{x}_i(y_i - p(\boldsymbol{x}_i;\beta)) \\
\sum_{i=1}^{N} -\frac{\partial}{\partial\beta}\boldsymbol{x}_i p(\boldsymbol{x}_i;\beta)) \\
\sum_{i=1}^{N} -\boldsymbol{x}_i\frac{\partial}{\partial\beta}\frac{1}{1+e^{\beta^T\boldsymbol{x}_i}} \\
\sum_{i=1}^{N} \frac{-\boldsymbol{x}_i e^{\beta^T\boldsymbol{x}_i} x^T}{(1+e^{\beta^T\boldsymbol{x}_i})^2} \\
\sum_{i=1}^{N} \frac{-\boldsymbol{x}_i e^{\beta^T\boldsymbol{x}_i} x^T}{(1+e^{\beta^T\boldsymbol{x}_i})^2}
\end{gathered}$$

Note:

$$\begin{aligned}
1 - p(\boldsymbol{x}_i;\beta) &= \frac{1+e^{\beta^T\boldsymbol{x}_i}}{1+e^{\beta^T\boldsymbol{x}_i}} - \frac{1}{1+e^{\beta^T\boldsymbol{x}_i}} \\
&= \frac{e^{\beta^T\boldsymbol{x}_i}}{1+e^{\beta^T\boldsymbol{x}_i}}
\end{aligned}$$

So,

$$\begin{aligned}
\sum_{i=1}^{N} \frac{-\boldsymbol{x}_i e^{\beta^T\boldsymbol{x}_i} x^T}{(1+e^{\beta^T\boldsymbol{x}_i})^2} &= \frac{-\boldsymbol{x}_i^T\boldsymbol{x}(1 - p(\boldsymbol{x}_i;\beta))}{1+e^{\beta^T\boldsymbol{x}_i}} \\
&= -\boldsymbol{x}_i^T\boldsymbol{x}\, p(\boldsymbol{x}_i;\beta)(1 - p(\boldsymbol{x}_i;\beta))
\end{aligned} \tag{6}$$

6 k-nearest Neighbor Classifier

Problem 13: For the data set described in problem 3, program a k-nearest neighbor function. Make it so that you can change the value of k. Use your k-nearest neighbor function for classification of the training data and 10,000 points of test data for k = 1, k = 5, and k = 15. Comment on the probability of error on the training data when k = 1. Plot the classification regions. Record the probability of classification error for test and training data on the table.

As seen in the following code, the k value can be easily manipulated to give the corresponding results seen in Figure 5 through Figure 7.

Table 1 shows how the nearest neighbor program reacts when k=1. The program overfits to the training data and reports “0 Errors,” but later has the highest error rate on the test data.

The classification regions can be seen in Figure 5 through Figure 7.

		Errors in %	
Method	Run-time	Training	Test
1-Nearest Neighbor	35.02s	00.0	21.83
5-Nearest Neighbor	37.92s	12.0	20.29
15-Nearest Neighbor	36.47s	16.0	19.25

Table 1: Nearest Neighbor Performance Comparison

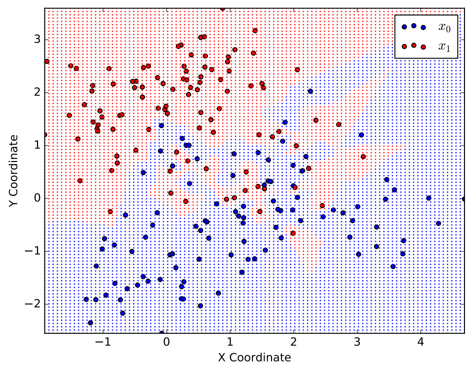

Figure 5: k=1 Nearest Neighbor

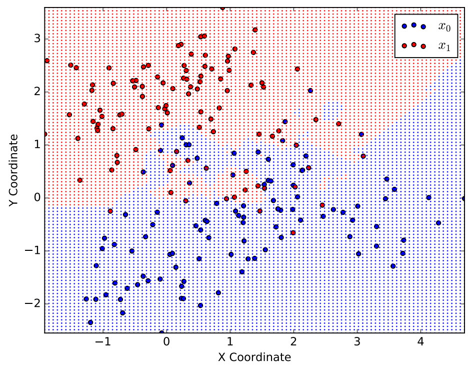

Figure 6: k=5 Nearest Neighbor

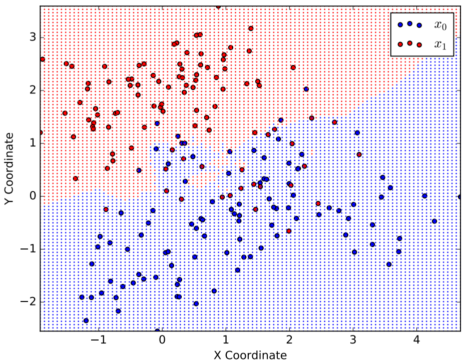

Figure 7: k=15 Nearest Neighbor

```
# Clint Ferrin
# Mon Sep 25, 2017
# K Nearest Neighbor

import sys
import numpy as np
import matplotlib.pyplot as plt

def gendata2(class_type,N):
    m0 = np.array(
         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])

    m1 = np.array(
          [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])

    x = np.array([[],[]])
    for i in range(N):
        idx = np.random.randint(10)
        if class_type == 0:
            m = m0[:,idx]
        elif class_type == 1:
            m = m1[:,idx]
        else:
            print("not a proper classifier")
            return 0
        x = np.c_[x, [[m[0]],[m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
    return x

class data_frame:
    def __init__(self, data):
        self.x0 = data[:,0:2]
        self.x1 = data[:,2:4]
        self.xtot = np.r_[self.x0,self.x1]
        self.N0 = self.x0.shape[0]
        self.N1 = self.x1.shape[0]
        self.N = self.N0 + self.N1
        self.xlim = [np.min(self.xtot[:,0]),np.max(self.xtot[:,0])]
        self.ylim = [np.min(self.xtot[:,1]),np.max(self.xtot[:,1])]

def plot_data(data):
    fig = plt.figure() # make handle to save plot
    plt.scatter(data.x0[:,0],data.x0[:,1],c='blue',label='$x_0$')
    plt.scatter(data.x1[:,0],data.x1[:,1],c='red',label='$x_1$')
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.legend()

def get_distance_matrix(X,point):
    dist_mat = np.empty([X.shape[0],2])
    for i in range(X.shape[0]):
```

```
        dist_mat[i] = [np.linalg.norm(point - X[i,0:2]),X[i,2]]
    return dist_mat

def find_class(dist_mat,k):
    neighbors = dist_mat[0:k]
    for i in range(k,X.shape[0]):
        if np.max(neighbors[:,0]) > dist_mat[i,0]:
            neighbors[np.argmax(neighbors[:,0])] = dist_mat[i]
    prob = sum(neighbors[:,1])/k
    if prob > 0.5:
        class_type = 1
    else:
        class_type = 0
    return class_type

data = np.loadtxt("../data/classasgntrain1.dat",dtype=float)
data = data_frame(data)
k = 5
y = np.r_[np.zeros([data.N0,1]),np.ones([data.N1,1])]
X = np.c_[data.xtot,y]
yhat = np.empty([data.N,1])

for i in range(data.N):
    dist_mat = get_distance_matrix(X,data.xtot[i])
    yhat[i] = find_class(dist_mat,k)

num_err = (sum(abs(yhat - y)))
print("Percent of errors: %.4f"%(float(num_err)/data.N))

Ntest0 = 10000;
Ntest1 = 10000;

xtest0 = gendata2(0,Ntest0)
xtest1 = gendata2(1,Ntest1)
num_err = 0

for i in range(Ntest0):
    dist_mat = get_distance_matrix(X,xtest0[:,i])
    class_type = find_class(dist_mat,k)
    if class_type == 1:
        num_err = num_err + 1

for i in range(Ntest1):
    dist_mat = get_distance_matrix(X,xtest1[:,i])
    class_type = find_class(dist_mat,k)
    if class_type == 0:
        num_err = num_err + 1

print("Number of errors: %d"%(num_err))
err_rate_linregress_test = float(num_err) / (Ntest0 + Ntest1);
print("Percent of errors: %.4f"%(err_rate_linregress_test))

# create colored graph above/below line
xp1 = np.linspace(data.xlim[0],data.xlim[1], num=100)
yp1 = np.linspace(data.ylim[0],data.ylim[1], num=100)

red_pts = np.array([[],[]])
blue_pts= np.array([[],[]])

for x in xp1:
    for y in yp1:
        dist_mat = get_distance_matrix(X,[x,y])
        class_type = find_class(dist_mat,k)
        if class_type == 0:
            blue_pts = np.c_[blue_pts,[x,y]]
        else:
            red_pts = np.c_[red_pts,[x,y]]

plot_data(data)
plt.scatter(blue_pts[0,:],blue_pts[1,:],color='blue',s=0.25)
plt.scatter(red_pts[0,:],red_pts[1,:],color='red',s=0.25)
plt.xlim(data.xlim)
plt.ylim(data.ylim)
```

```
plt.show()
```

7 Naive Bayes Classifier

Problem 14: The true density is a Gaussian mixture, $f_X(x) = p_0 N(x, \mu_0, \sigma_0^2) + p_1 N(x, \mu_1, \sigma_1^2)$. Using this equation, with given values of μ and $sigma$, make a plot of the true density, the sample empirical distribution, the kernel functions, and the estimated density. Try different values of λ.

As seen in Figure 8, the Parzen distribution was calculated using an empirical distribution and adding the corresponding Gaussian distributions. The λ of $\lambda = 0.8$ was chosen for the best results. If λ is too small, the data begins to look corrupt, and if it was much larger than $\lambda = 0.8$, it became too peaked in the middle regions.

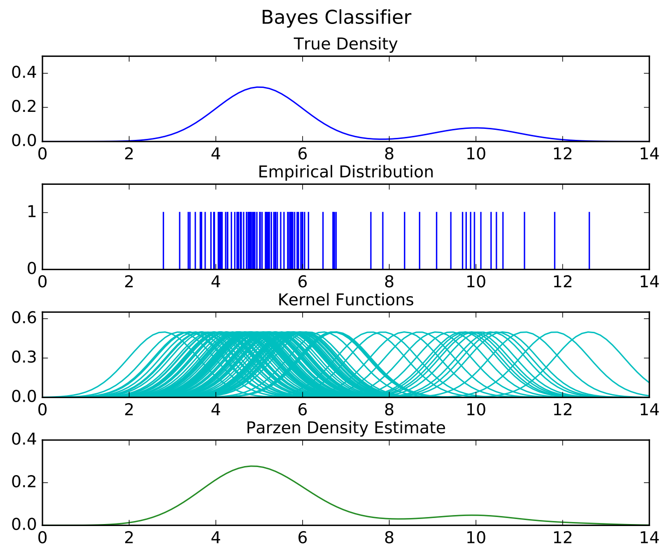

Figure 8: Calculating Parzen Distribution

```
# Clint Ferrin
# Mon Sep 25, 2017
# Parzen Density Estimate

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

p_0 = 0.8
p_1 = 0.2
mu_0 = 5
mu_1 = 10
sig_0 = 1
sig_1 = 1

lam = 0.8

f, ax = plt.subplots(4,1)
f.subplots_adjust(hspace=.5)
f.suptitle("Bayes Classifier",fontsize=14)

ax[0].set_title("True Density",fontsize=12)
ax[0].set_ylim(0, 0.5)
ax[0].yaxis.set_ticks(np.arange(0,0.6,0.2))
```

```
ax[1].set_title("Empirical Distribution",fontsize=12)
ax[1].set_ylim(0, 1.5)
ax[1].yaxis.set_ticks(np.arange(0,2,1))
ax[2].set_title("Kernel Functions",fontsize=12)
ax[2].set_ylim(0, .65)
ax[2].yaxis.set_ticks(np.arange(0,.64,.3))
ax[3].set_title("Parzen Density Estimate",fontsize=12)
ax[3].set_ylim(0, 0.35)
ax[3].yaxis.set_ticks(np.arange(0,0.6,0.2))
for i in range(4):
    ax[i].set_xlim(0, 14)

pts = 100
x = np.linspace(0,14, pts)
pdf = p_0*mlab.normpdf(x, mu_0, sig_0)+p_1*mlab.normpdf(x, mu_1, sig_1)
ax[0].plot(x,pdf)
cdf = np.cumsum(pdf)/sum(pdf)



emperical = np.interp(np.random.rand(pts,1),cdf,x)
emperical = np.sort(emperical,axis=None)

ax[1].stem(emperical,np.ones(pts),'b',markerfmt=' ')
ax[1].set_xlim(0,14)

kernel = np.empty([emperical.shape[0],pts])
for i in range(emperical.shape[0]):
    kernel[i] = mlab.normpdf(x, emperical[i], lam)
    ax[2].plot(x,kernel[i],color='c')

parzen = np.empty(100)
for i in range(pts):
    parzen[i] = sum(kernel[:,i])/pts

ax[3].plot(x,parzen,color='forestgreen')

plt.show()
```

Problem 15: Using the training data in `classassgntrain1.dat` to estimate the densities, apply the Naive Bayes estimator to our data set. Plot the classification regions. Record the probability of classification error for test and training data on the table.

The data with the classification regions are plotted in Figure 10 and the code is typed out below the graph. The probability of classification error for test and training data are recorded on Table 2.

To estimate the density functions, I simply found the probability that an x value would be chosen in `class0` and a y value in `class0`. Their pdf were calculated using the Parzen technique, and their distribution can be seen in Figure 9.

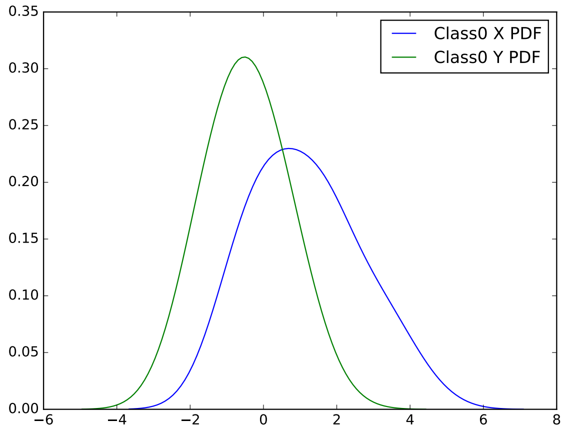

Figure 9: Bayes 1D plot of X and Y

Because the Bayes method assumes the probability is independent, the two can simply be multiplied together to create a probability that a given point is in class 0 or 1. Figure 10 shows how the division line ended up, and the classification error can be seen in Table 2.

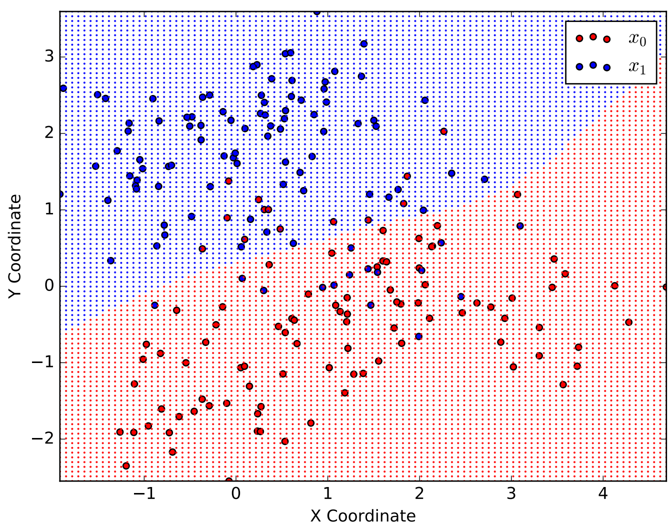

Figure 10: Naive Bayes Parzen Graph

```
# Clint Ferrin
# Mon Sep 25, 2017
# Bayes Naive Classifier

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

class data_frame:
    def __init__(self, data0, data1):
        self.x0 = data0
        self.x1 = data1
        self.xtot = np.c_[self.x0,self.x1]
        self.N0 = self.x0.shape[1]
```

```
        self.N1 = self.x1.shape[1]
        self.N = self.N0 + self.N1
        self.xlim = [np.min(self.xtot[0,:]),np.max(self.xtot[0,:])]
        self.ylim = [np.min(self.xtot[1,:]),np.max(self.xtot[1,:])]

def gendata2(class_type,N):
    m0 = np.array(
         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])

    m1 = np.array(
          [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])

    x = np.array([[],[]])
    for i in range(N):
        idx = np.random.randint(10)
        if class_type == 0:
            m = m0[:,idx]
        elif class_type == 1:
            m = m1[:,idx]
        else:
            print("not a proper classifier")
            return 0
        x = np.c_[x, [[m[0]],[m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
    return x

def gen_test_df(num0,num1):

    return data_frame

def get_parzen(data,pts,lam):
    x = np.linspace(min(data)-3*lam,max(data)+3*lam, pts)
    kernel = np.empty([data.size,pts])
    for i in range(data.size):
        kernel[i] = mlab.normpdf(x, data[i], lam)
        # plt.plot(x,kernel[i],color='b')

    parzen = np.empty(data.size)
    for i in range(data.size):
        parzen[i] = sum(kernel[:,i])/pts

    return x, parzen

def class_parzen(data0,pts,lam):
    x0,parzen_x0 = get_parzen(data0[0,:],pts[0],lam)
    y0,parzen_y0= get_parzen(data0[1,:],pts[1],lam)
    return [x0,y0],[parzen_x0,parzen_y0]

def prob2d(point,linspace0,parzen0):
    prob_x = np.interp(point[0],linspace0[0],parzen0[0])
    prob_y = np.interp(point[1],linspace0[1],parzen0[1])
    # print("prob x: %f, prob y: %f"%(prob_x, prob_y))
    return prob_x*prob_y

def run_bayes_test(data_tot,linspace,parzen):
    y = np.r_[np.zeros([data_tot.shape[1],1]),np.ones([data_tot.shape[1],1])]
    y_hat = np.zeros([data_tot.shape[1],1])

    for i in range(data_tot.shape[1]):
        prob0 = prob2d(data_tot[:,i],linspace[0],parzen[0])
        prob1 = prob2d(data_tot[:,i],linspace[1],parzen[0])
        if prob1 > prob0:
            y_hat[i] = 1

    return y_hat

def plot_data(x0,x1):
    fig = plt.figure() # make handle to save plot
    plt.scatter(x0[0,:],x0[1,:],c='red',label='$x_0$')
    plt.scatter(x1[0,:],x1[1,:],c='blue',label='$x_1$')
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
```

```python
    plt.legend()


data = np.loadtxt("../data/classasgntrain1.dat",dtype=float)
x0 = data[:,0:2].T
x1 = data[:,2:4].T
data = data_frame(x0,x1)

pts = [data.N0,data.N1]
lam = 0.8

linspace0,parzen0 = class_parzen(data.x0,pts,lam)
linspace1,parzen1 = class_parzen(data.x1,pts,lam)

plt.plot(linspace0[0],parzen0[0],label='Class0 X PDF')
plt.plot(linspace0[1],parzen0[1],label='Class0 Y PDF')
plt.legend()

print(np.array(parzen0).shape)

linspace = np.array([linspace0,linspace1])
parzen = np.array([parzen0,parzen1])

y = np.r_[np.zeros([data.N1,1]),np.ones([data.N0,1])]
y_hat = run_bayes_test(data.xtot,linspace,parzen)

num_err = sum(abs(y_hat - y))
print("Percent of errors: %.4f"%(float(num_err)/data.N))


xtest0 = gendata2(0,10000)
xtest1 = gendata2(1,10000)
test_data = data_frame(xtest0,xtest1)
y = np.r_[np.zeros([test_data.N1,1]),np.ones([test_data.N0,1])]

y_hat = run_bayes_test(test_data.xtot,linspace,parzen)

num_err = sum(abs(y_hat - y))
print("Percent of errors: %.4f"%(float(num_err)/test_data.N))

xp1 = np.linspace(data.xlim[0],data.xlim[1], num=100)
yp1 = np.linspace(data.ylim[0],data.ylim[1], num=100)

red_pts = np.array([[],[]])
blue_pts= np.array([[],[]])
for x in xp1:
    for y in yp1:
        prob0 = prob2d([x,y],linspace[0],parzen[0])
        prob1 = prob2d([x,y],linspace[1],parzen[0])
        if prob1 > prob0:
            blue_pts = np.c_[blue_pts,[x,y]]
        else:
            red_pts = np.c_[red_pts,[x,y]]

plot_data(x0,x1)
plt.scatter(blue_pts[0,:],blue_pts[1,:],color='blue',s=0.25)
plt.scatter(red_pts[0,:],red_pts[1,:],color='red',s=0.25)
plt.xlim(data.xlim)
plt.ylim(data.ylim)
plt.show()
```

8 Optimal Bayes Classifier

Problem 16: For the data set described in problem 3, determine the Bayes error rate on the training data and 10,000 points of test data. Record the probability of classification error for test and training data on the table. Plot the classification regions.

The classification regions for the Optimal Bayes Classifier can be seen in Figure 11. It performed the best on the test data above all other data sets because it had the actual data model embedded into the classifier. See Table 2 for specific details.

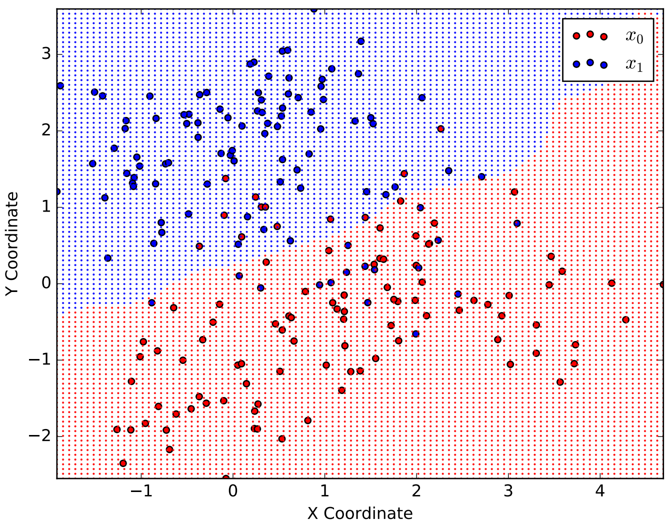

Figure 11: Optimal Bayes Classifier

```
# Clint Ferrin
# Mon Sep 25, 2017
# Bayes Optimal Classifier

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab

class data_frame:
    def __init__(self, data0, data1):
        self.x0 = data0
        self.x1 = data1
        self.xtot = np.c_[self.x0,self.x1]
        self.N0 = self.x0.shape[1]
        self.N1 = self.x1.shape[1]
        self.N = self.N0 + self.N1
        self.xlim = [np.min(self.xtot[0,:]),np.max(self.xtot[0,:])]
        self.ylim = [np.min(self.xtot[1,:]),np.max(self.xtot[1,:])]

def gendata2(class_type,N):
    m0 = np.array(
         [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
          [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]])

    m1 = np.array(
          [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
          [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]])

    x = np.array([[],[]])
    for i in range(N):
        idx = np.random.randint(10)
        if class_type == 0:
            m = m0[:,idx]
        elif class_type == 1:
            m = m1[:,idx]
        else:
            print("not a proper classifier")
            return 0
        x = np.c_[x, [[m[0]],[m[1]]] + np.random.randn(2,1)/np.sqrt(5)]
    return x
```

```

def get_parzen(data,pts,lam):
    x = np.linspace(min(data)-3*lam,max(data)+3*lam, pts)
    kernel = np.empty([data.size,pts])
    for i in range(data.size):
        kernel[i] = mlab.normpdf(x, data[i], lam)
        # plt.plot(x,kernel[i],color='b')

    parzen = np.empty(data.size)
    for i in range(data.size):
        parzen[i] = sum(kernel[:,i])/pts

    return x, parzen

def class_parzen(data0,pts,lam):
    x0,parzen_x0 = get_parzen(data0[0,:],pts[0],lam)
    y0,parzen_y0= get_parzen(data0[1,:],pts[1],lam)
    return [x0,y0],[parzen_x0,parzen_y0]

def prob2d(point,linspace0,parzen0):
    prob_x = np.interp(point[0],linspace0[0],parzen0[0])
    prob_y = np.interp(point[1],linspace0[1],parzen0[1])
    # print("prob x: %f, prob y: %f"%(prob_x, prob_y))
    return prob_x*prob_y

def run_bayes_test(data_tot,linspace,parzen):
    y = np.r_[np.zeros([data_tot.shape[1],1]),np.ones([data_tot.shape[1],1])]
    y_hat = np.zeros([data_tot.shape[1],1])

    for i in range(data_tot.shape[1]):
        prob0 = prob2d(data_tot[:,i],linspace[0],parzen[0])
        prob1 = prob2d(data_tot[:,i],linspace[1],parzen[0])
        if prob1 > prob0:
            y_hat[i] = 1

    return y_hat

def plotData(data):
    fig = plt.figure() # make handle to save plot
    plt.scatter(data.x0[0,:],data.x0[1,:],c='red',label='$x_0$')
    plt.scatter(data.x1[0,:],data.x1[1,:],c='blue',label='$x_1$')
    plt.xlabel('X Coordinate')
    plt.ylabel('Y Coordinate')
    plt.legend()

data = np.loadtxt("../data/classasgntrain1.dat",dtype=float)
x0 = data[:,0:2].T
x1 = data[:,2:4].T
data = data_frame(x0,x1)

m0 = np.array(
     [[-0.132,0.320,1.672,2.230,1.217,-0.819,3.629,0.8210,1.808, 0.1700],
      [-0.711,-1.726,0.139,1.151,-0.373,-1.573,-0.243,-0.5220,-0.511,0.5330]]).T

m1 = np.array(
      [[-1.169,0.813,-0.859,-0.608,-0.832,2.015,0.173,1.432,0.743,1.0328],
      [ 2.065,2.441,0.247,1.806,1.286,0.928,1.923,0.1299,1.847,-0.052]]).T

pts = [m0.shape[0],m1.shape[0]]
lam = 0.8

linspace0,parzen0 = class_parzen(m0.T,pts,lam)
linspace1,parzen1 = class_parzen(m1.T,pts,lam)



linspace = np.array([linspace0,linspace1])
parzen = np.array([parzen0,parzen1])

y = np.r_[np.zeros([data.N0,1]),np.ones([data.N1,1])]
y_hat = run_bayes_test(data.xtot,linspace,parzen)

num_err = sum(abs(y_hat - y))
```

```python
print("Percent of errors: %.4f"%(float(num_err)/data.N))


xtest0 = gendata2(0,10000)
xtest1 = gendata2(1,10000)
test_data = data_frame(xtest0,xtest1)
y = np.r_[np.zeros([test_data.N1,1]),np.ones([test_data.N0,1])]

y_hat = run_bayes_test(test_data.xtot,linspace,parzen)

num_err = sum(abs(y_hat - y))
print("Percent of errors: %.4f"%(float(num_err)/test_data.N))

xp1 = np.linspace(data.xlim[0],data.xlim[1], num=100)
yp1 = np.linspace(data.ylim[0],data.ylim[1], num=100)

red_pts = np.array([[],[]])
blue_pts= np.array([[],[]])
for x in xp1:
    for y in yp1:
        prob0 = prob2d([x,y],linspace[0],parzen[0])
        prob1 = prob2d([x,y],linspace[1],parzen[0])
        if prob1 > prob0:
            blue_pts = np.c_[blue_pts,[x,y]]
        else:
            red_pts = np.c_[red_pts,[x,y]]

plotData(data)
plt.scatter(blue_pts[0,:],blue_pts[1,:],color='blue',s=0.25)
plt.scatter(red_pts[0,:],red_pts[1,:],color='red',s=0.25)
plt.xlim(data.xlim)
plt.ylim(data.ylim)
plt.show()
```

9 Discussion

Problem 17: Discuss the relative merits of the different classification algorithms on this source of data. Comment on differences in performance between training and test data. Also, comment on operating speed of the classification algorithms (after they have been trained). Summarize what you have learned. Turn in with this assignment your table of results, plots of data and classification regions, and listings of your PYTHON code.

The highest performing data classifier was the Bayes Optimal classifier. In this case it was because we had the exact model of the data, but in real world examples, it is not feasible to have such a model.

The Linear regression model was similar in run-time to the optimal classifier, and it performed well, regardless of its seeming simplicity.

The 15-nearest neighbor performed as well as the Bayes Optimal classifier, but it took considerably longer. There exist ways to optimize the code, because I was comparing the distance of every point in the data set to the test point, which required a lot of wasted time. Regardless, one of the major drawbacks of the nearest neighbor, is that it takes a considerable amount of time to run through the algorithm.

		Errors in %	
Method	Run-time	Training	Test
Linear Regression	1.23s	14.5	20.49
Quadratic Regression	1.70s	14.5	20.44
Linear Discriminant Analysis	2.49s	15.0	19.98
Quadratic Discriminant Analysis	3.26s	14.5	20.23
Logistic Regression	2.00s	14.0	20.00
1-Nearest Neighbor	35.02s	00.0	21.83
5-Nearest Neighbor	37.92s	12.0	20.29
15-Nearest Neighbor	36.47s	16.0	19.25
Bayes Naive	1.22s	14.0	20.04
Bayes Optimal Classifier	0.20s	14.0	19.14

Table 2: Binary Classifier Performance Comparison