Problem 1.

Take the following training/test split: 13000 training and 6020 test. Perform all necessary tests, plots, and Monte Carlo simulations to determine your final choice of classifier. You can use the built-in function of your favorite program. Can you beat the 86.6% mean accuracy on test data (based on 100 runs)?

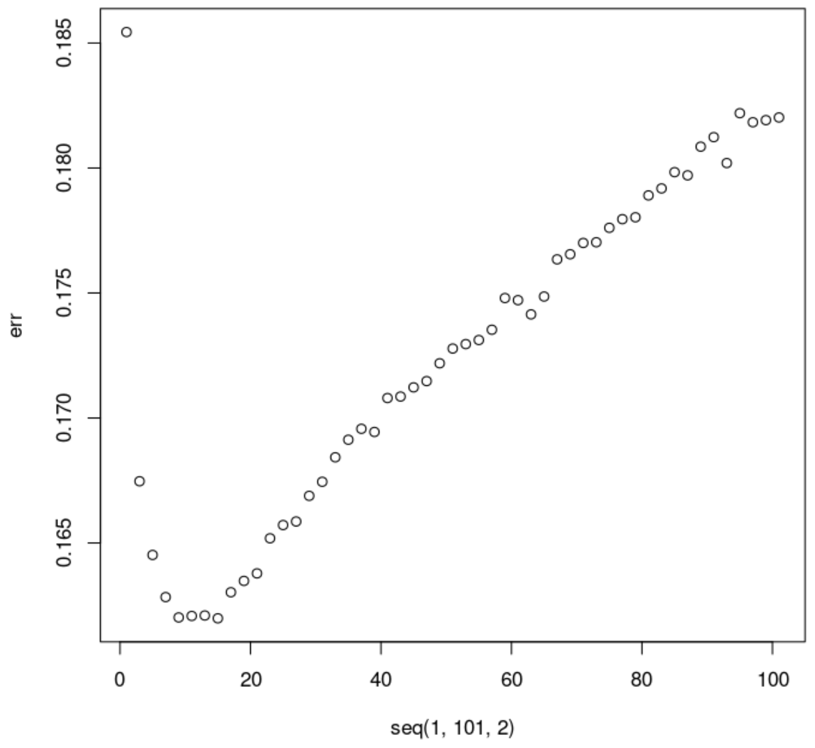
Solution

Comments are provided in all sample R code near print statements detailing the results of runs of the script.

KNN

K-Nearest Neighbors was my best performing method, performing just shy of 81%, due to the magnitude of the data set, finding the best k value time consuming, even with some parallel processing included to speed up the process. Minimum k was found by computing the mean error over 100 runs for all odd k $\in [1, 101]$. I ran some other lighter tests outside of this range with fewer random runs, but the error seemed to continue to increase for lager k values. As such, I ignored these K-values when performing more rigorous tests.

Initially I achieved an accuracy of about 81% with KNN, but by normalizing my features I was able to bump that up to near 84%.

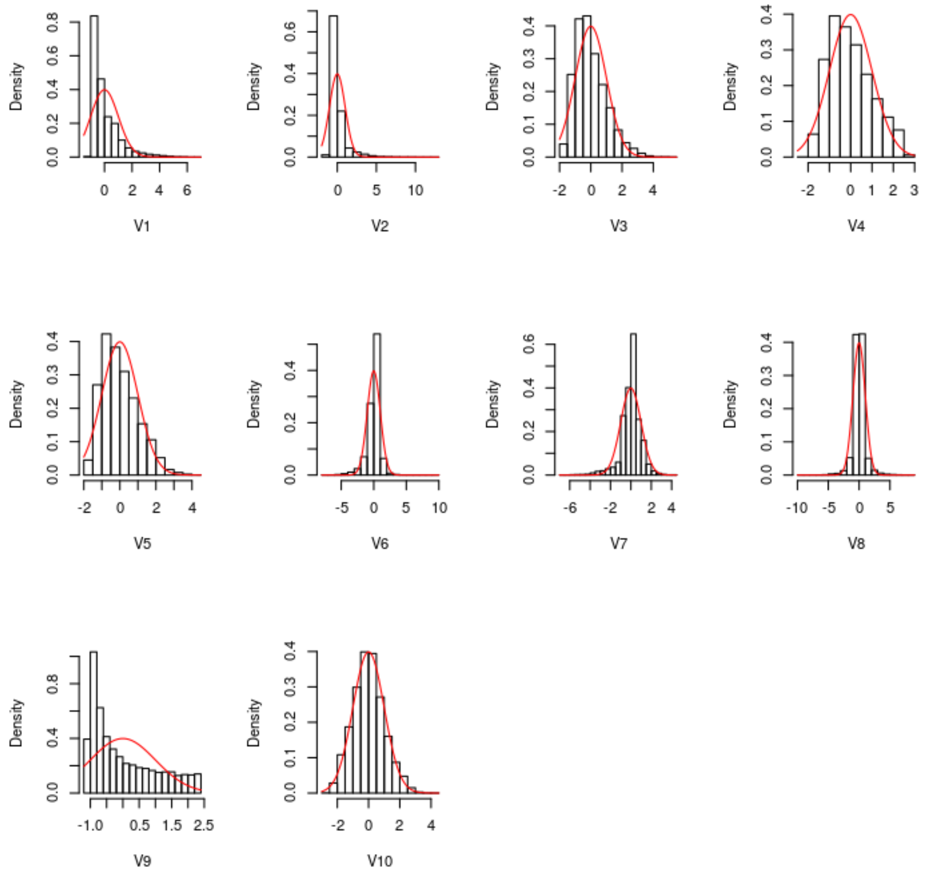


```r
#########
# Setup #
#########

library('foreach')
library('doParallel')
library('ggplot2')

# Read, scale, & center our data.
data <- read.csv("magic04.data", header=F, sep=",")
data[1:nrow(data), 1:10] <- scale(data[1:nrow(data), 1:10], center=T, scale=T)
train.size <- 13000 # Given by homework specification
start.time <- proc.time()

#######
# KNN #
#######

# Try a bunch of different K-values,

err <- foreach (K=seq(1,101,2), .combine = c) %do% {

    # Need to load the library for knn on each thread.
    registerDoParallel(cores=4)

    # Run KNN 100 times for each K value.
    # Each run is independent, so we can speed things up a little
    # bit by running it in parallel.
    k.err <- foreach (i=1:100, .combine = c) %dopar% {

        library(class)
        data <- data[sample(nrow(data)),] # Randomize the data set

        train <- data[1:train.size, 1:10]
        test <- data[(train.size+1):nrow(data), 1:10]
        train.cl <- factor(data[1:train.size, 11])
        test.cl <- factor(data[(train.size+1):nrow(data), 11]);

        predict.cl <- knn(train, test, train.cl, k=K)
        sum(test.cl != predict.cl) / nrow(test)
    }

    stopImplicitCluster()
    mean(k.err)
}

plot(seq(1,101,2), err)

# This was our best performing k value.
k <- which.min(err)
min.err <- min(err)
acc <- 1.0 - min.err

# K = 15
print(paste("Min K: ", 1+(k-1)*2))
# About 83.801%
print(paste("KNN - Accuracy: ", acc))
# 2902.621 ~ 48.377m
print(proc.time() - start.time)
```

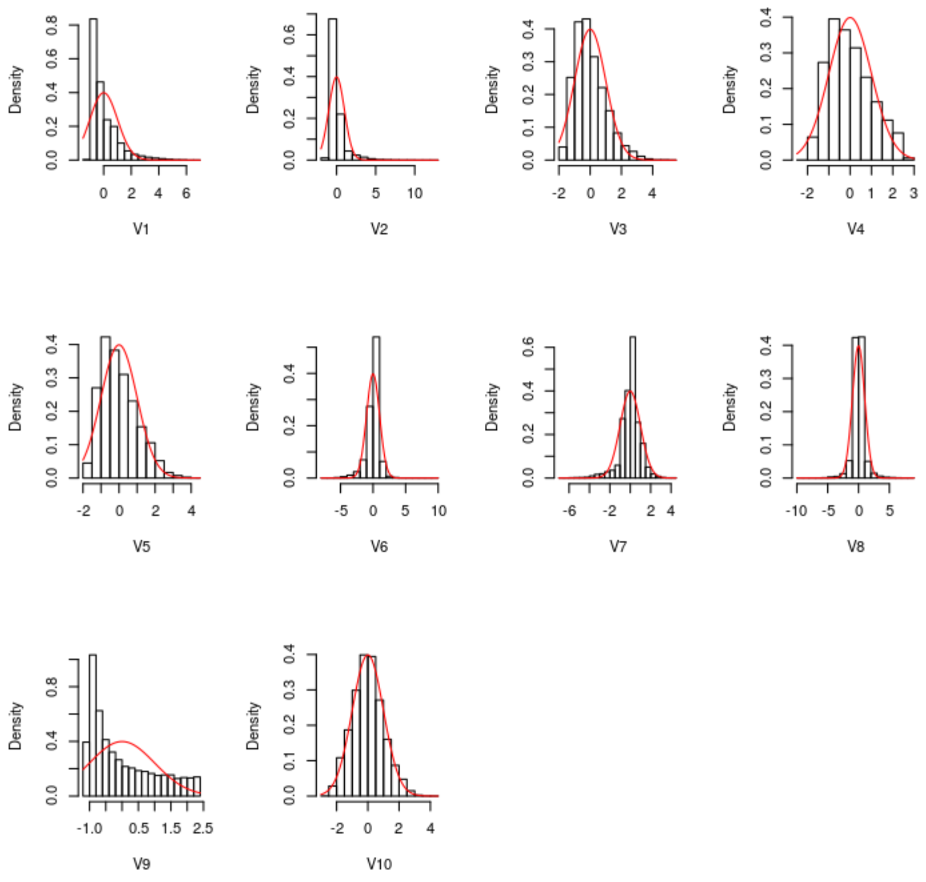
LDA

Both the hzTest and uniPlot indicated that the data was not multi-variate normal. From the plot we can see that some features look almost normal, but overall the data does not follow a normal distribution. Because of this, we can expect LDA and QDA to not perform well.
Running these methods anyway produced the expected results, both performed at about 78% accuracy on test, which is less than I was able to achieve with KNN.
Unlike with KNN, normalizing the data did not seem to affect the performance of the classifier, which is to be expected as they do not rely on distances as KNN does.

```
  Henze-Zirkler's Multivariate Normality Test
---------------------------------------------
  data : data[data[, 11] == "h", 1:10]

  HZ      : 19.21569
  p-value : 0

  Result  : Data are not multivariate normal.
---------------------------------------------
```



```
  Henze-Zirkler's Multivariate Normality Test
---------------------------------------------
  data : data[data[, 11] == "g", 1:10]

  HZ      : 12.72463
  p-value : 0

  Result  : Data are not multivariate normal.
---------------------------------------------
```



```r
#########
# Setup #
#########

library('foreach')
library('doParallel')
library('MVN')

# Read, scale, & center our data.
data <- read.csv("magic04.data", header=F, sep=",")
data[1:nrow(data), 1:10] = scale(data[1:nrow(data), 1:10], center=T, scale=T)
train.size <- 13000 # Given by homework specification
start.time <- proc.time()

#####################
# Test for Normality #
#####################

sink("g.norm.txt", append=F, split=F)
data <- data[sample(nrow(data)),] # Randomize the data set
hz = hzTest(data[data[,11]=='g',1:10], cov=TRUE, qqplot=FALSE)
print(hz)
# uniPlot(data, type="histogram")
sink()

#######
# LDA #
#######

registerDoParallel(8)

err <- foreach (i=1:100, .combine = c) %dopar% {
    library(MASS)

    data <- data[sample(nrow(data)),] # Randomize the data set

    train <- data[1:train.size, 1:10]
    test <- data[(train.size+1):nrow(data), 1:10]
    train.cl <- factor(data[1:train.size, 11])
    test.cl <- factor(data[(train.size+1):nrow(data), 11]);

    model <- lda(x = train, grouping = train.cl)
    predict.cl <- predict(model, test)$class
    sum(test.cl != predict.cl) / nrow(test)
}

stopImplicitCluster()
acc <- 1.0 - mean(err)

# About 78.429%
print(paste("LDA - Accuracy: ", acc))
print(proc.time() - start.time)
```

QDA

Due to the poor performance of LDA, and the non-linear nature of the input data set, we do not expect QDA to perform any better. It may have one assumption less than LDA, the assumption it keeps still fails. Running the algorithm produces a mean accuracy roughly equivalent to that produced by LDA.

```
#########
# Setup #
#########

library('foreach')
library('doParallel')

# Read, scale, & center our data.
data <- read.csv("magic04.data", header=F, sep=",")
data[1:nrow(data), 1:10] = scale(data[1:nrow(data), 1:10], center=T, scale=T)
train.size <- 13000 # Given by homework specification
start.time <- proc.time()

#######
# QDA #
#######

registerDoParallel(8)

err <- foreach (i=1:100, .combine = c) %dopar% {
    library(MASS)

    data <- data[sample(nrow(data)),] # Randomize the data set

    train <- data[1:train.size, 1:10]
    test <- data[(train.size+1):nrow(data), 1:10]
    train.cl <- factor(data[1:train.size, 11])
    test.cl <- factor(data[(train.size+1):nrow(data), 11]);
```

```r

    model <- qda(x = train, grouping = train.cl)
    predict.cl <- predict(model, test)$class
    sum(test.cl != predict.cl) / nrow(test)
}

stopImplicitCluster()
acc <- 1.0 - mean(err)

# About 78.4276%
print(paste("QDA - Accuracy: ", acc))
print((proc.time() - start.time))
```

Naive Bayes (Normal)

Naive Bayes was less interesting, again, as was the case with LDA, the normal assumption of the data set failed, but I went ahead and ran it anyway. The results were unexciting, serving as the worst performing classifier over 100 random test cases.

```r
#########
# Setup #
#########

library('foreach')
library('doParallel')

# Read, scale, & center our data.
data <- read.csv("magic04.data", header=F, sep=",")
data[1:nrow(data), 1:10] = scale(data[1:nrow(data), 1:10], center=T, scale=T)
train.size <- 13000 # Given by homework specification
start.time <- proc.time()

########################
# Naive Bayes (Normal) #
########################

registerDoParallel(8)

err <- foreach (i=1:100, .combine = c) %dopar% {
    library(klaR)
    library(caret)

    data <- data[sample(nrow(data)),] # Randomize the data set

    train <- data[1:train.size, 1:10]
    test <- data[(train.size+1):nrow(data), 1:10]
    train.cl <- factor(data[1:train.size, 11])
    test.cl <- factor(data[(train.size+1):nrow(data), 11]);

    model <- NaiveBayes(x = train, grouping = train.cl, usekernel=FALSE)
    predict.cl <- predict(model, test)$class
    sum(test.cl != predict.cl) / nrow(test)
}

stopImplicitCluster()
acc <- 1.0 - mean(err)

# About 72.6714%
print(paste("Naive Bayes (Normal) - Accuracy: ", acc))
print(proc.time() - start.time)
```

Naive Bayes (Kernel)

Removing the normal assumption of the input data set increased the processing time required to train a model using Naive Bayes with a Kernel density estimation. But this resulted in an improvement in accuracy over Naive Bayes with a Normal assumption. Despite this it still performed worse than LDA and QDA, much less KNN.

```
#########
# Setup #
#########

library('foreach')
library('doParallel')

# Read, scale, & center our data.
data <- read.csv("magic04.data", header=F, sep=",")
data[1:nrow(data), 1:10] = scale(data[1:nrow(data), 1:10], center=T, scale=T)
train.size <- 13000 # Given by homework specification
start.time <- proc.time()

########################
# Naive Bayes (Kernel) #
########################

registerDoParallel(8)

err <- foreach (i=1:100, .combine = c) %dopar% {
    library(klaR)
    library(caret)

    data <- data[sample(nrow(data)),] # Randomize the data set

    train <- data[1:train.size, 1:10]
    test <- data[(train.size+1):nrow(data), 1:10]
    train.cl <- factor(data[1:train.size, 11])
    test.cl <- factor(data[(train.size+1):nrow(data), 11]);

    model <- NaiveBayes(x = train, grouping = train.cl, usekernel=TRUE)
    predict.cl <- predict(model, test)$class
    sum(test.cl != predict.cl) / nrow(test)
}

stopImplicitCluster()
acc <- 1.0 - mean(err)

# About 76.2375%
print(paste("Naive Bayes (Kernel) - Accuracy: ", acc))
print((proc.time() - start.time))
```

Problem 2.

(a) Classify the test point (0, 1) using QDA and calculate the posterior class probabilities. Do the calculations by hand.

(b) Classify the test point (0, 1) using naive Bayes assuming normality and calculate the posterior class probabilities. Do the calculations by hand.

(c) Verify your results for both classifiers using Matlab, R, etc. You can use the built-in functions.

Solution

Part (a)

First we need to calculate the class means $\hat{\mu}_0$ and $\hat{\mu}_1$

$$\begin{aligned}
\hat{\mu}_0 &= \frac{1}{3}([0.6585, 0.2444] + [2.2460, 0.5281] + [-2.7665, -3.8303]) \\
&= \frac{1}{3}[0.138, -3.8303] \\
&= [0.046, -1.019267] \\
\hat{\mu}_1 &= \frac{1}{3}([-1.2565, 3.4912] + [-0.7973, 1.2288] + [1.1170, 2.2637]) \\
&= \frac{1}{3}[-0.9368, 6.9837] \\
&= [-0.3122667, 2.3279000]
\end{aligned}$$

Next calculate the covariance matrix for each class.

$$\begin{aligned}
\hat{\Sigma}_0 = &\frac{1}{2}(([0.6585, 0.2444] - \hat{\mu}_0)([0.6585, 0.2444] - \hat{\mu}_0)^\intercal \\
&+ ([2.2460, 0.5281] - \hat{\mu}_0)([2.2460, 0.5281] - \hat{\mu}_0)^\intercal \\
&+ ([-2.7665, -3.8303] - \hat{\mu}_0)([-2.7665, -3.8303] - \hat{\mu}_0)^\intercal \\
&= \frac{1}{2}(\begin{bmatrix} 0.33516 & 0.77400 \\ 0.77400 & 1.59685 \end{bmatrix} + \begin{bmatrix} 4.8400 & 3.4042 \\ 3.4042 & 2.394 \end{bmatrix} + \begin{bmatrix} 7.9102 & 7.9060 \\ 7.9060 & 7.9019 \end{bmatrix}) \\
&= \begin{bmatrix} 6.5627 & 6.0421 \\ 6.0421 & 5.9466 \end{bmatrix}
\end{aligned}$$

$$\begin{aligned}
\hat{\Sigma}_1 &= \frac{1}{2}(([-1.2565, 3.4912] - \hat{\mu}_1)([-1.2565, 3.4912] - \hat{\mu}_1)^\intercal \\
&+ ([-0.7973, 1.2288] - \hat{\mu}_1)([-0.7973, 1.2288] - \hat{\mu}_1)^\intercal \\
&+ ([1.1170, 2.2637] - \hat{\mu}_1)([1.1170, 2.2637] - \hat{\mu}_1)^\intercal \\
&= \frac{1}{2}(\begin{bmatrix} 0.89158 & -1.09843 \\ -1.09843 & 1.35327 \end{bmatrix} + \begin{bmatrix} 0.23526 & 0.53310 \\ 0.53310 & 1.20802 \end{bmatrix} + \begin{bmatrix} 2.0426033 & -0.0917589 \\ -0.0917589 & 2.0426033 \end{bmatrix}) \\
&= \begin{bmatrix} 1.58482 & -0.32854 \\ -0.32854 & 1.28270 \end{bmatrix}
\end{aligned}$$

We would like to maximize $\hat{P}[Y = k]\hat{f}(X = x|Y = k)$ over $k \in \{0, 1\}$ for our input x=(0,1).

$$\underset{k\in\{0,1\}}{\arg\max}\,\hat{P}[Y = k]\left(\frac{1}{(2\pi)^{\frac{d}{2}}|\hat{\Sigma}_k|^{\frac{1}{2}}}\right)\exp(\frac{1}{2}(x - \hat{\mu}_k)^\intercal\hat{\Sigma}_k^{-1}(x - \hat{\mu}_k))$$

case: k = 0

$$\hat{P}[Y = 0] = \frac{3}{6} = \frac{1}{2}$$
$$|\hat{\Sigma}_0| = 2.5180 \Rightarrow |\hat{\Sigma}_0|^{\frac{1}{2}} = 1.5868$$
$$\left(\frac{1}{(2\pi)^{\frac{2}{2}}|\hat{\Sigma}_0|^{\frac{1}{2}}}\right) = (\frac{1}{9.9702}) = 0.10030$$
$$\exp(\frac{1}{2}(x - \hat{\mu}_0)^\intercal\hat{\Sigma}_0^{-1}(x - \hat{\mu}_0)) =$$
$$\exp(-\frac{1}{2} * 11.078) = \exp(-5.5389) = 0.0039308$$
$$\hat{P}[Y = 0]\hat{f}(X = x|Y = 0) = .00019713$$

case: k = 1

$$\hat{P}[Y = 1] = \frac{3}{6} = \frac{1}{2}$$
$$|\hat{\Sigma}_1| = 1.9249 \Rightarrow |\hat{\Sigma}_1|^{\frac{1}{2}} = 1.3874$$
$$\left(\frac{1}{(2\pi)^{\frac{2}{2}}|\hat{\Sigma}_1|^{\frac{1}{2}}}\right) = \frac{1}{8.7174} = 0.11471$$
$$\exp(-\frac{1}{2}(x - \hat{\mu}_1)^\intercal\hat{\Sigma}_1^{-1}(x - \hat{\mu}_1)) =$$
$$\exp(-\frac{1}{2} * 1.3752) = \exp(-0.6876) = 0.50278$$
$$\hat{P}[Y = 1]\hat{f}(X = x|Y = 1) = 0.028838$$

Noting that $0.00019713 < 0.028838$, k=1 maximizes our function, thus we predict a class label of 1 for (0,1). The posterior class probability is given by the following function.

$$\begin{aligned} P[Y=k|X=x] &= \frac{P[Y=k]f_{1\ldots d}(X=x|Y=k)}{\sum_{i=0}^{k} P[Y=i]f_{Y\ldots d}(X=x|Y=i)} \\ &= \frac{0.028838}{0.028838+0.00019713} \\ &= 0.99321 \end{aligned}$$

Thus we find that we have a posterior class probability of 99.321% for class k=1.

Part (b)

The naive Bayes classification uses the same class norms $\hat{\mu}_0$ and $\hat{\mu}_1$.
However we will need to compute $\hat{\sigma}^2$ values for each feature of each class.

case: k = 0
First compute the variance of each feature.

$$\begin{aligned} \hat{\sigma}_0^2 &= \frac{1}{3} * \sum_{j \in C_0} (x_i - \hat{\mu}_i)^2 \\ &= \frac{1}{3}(((0.6585, 0.2444) - \hat{\mu}_0)^2 + ((2.2460, 0.5281) - \hat{\mu}_0)^2 + ((-2.7665, -3.8303) - \hat{\mu}_0)^2) \\ &= \frac{1}{3}(13.125, 11.893) \\ &= (4.3751, 3.9644) \end{aligned}$$

Now we can compute $f_i((0,1)|Y=0)$ for use in our classifier.

$$\begin{aligned} f((0,1)|Y=0) &= \frac{1}{\sqrt{2\pi\hat{\sigma}_0^2}} e^{-\frac{(<0,1>-\hat{\mu}_0)^2}{2\hat{\sigma}_0^2}} \\ &= \frac{1}{(5.2431, 4.9909)}(0.99976, 0.59794) \\ &= (0.19068, 0.11981) \end{aligned}$$

We would like to maximize the following equation over all k $\hat{P}[Y=0]\prod_{i=1}^{d} f_i(X_i|Y=k)$.

$$\begin{aligned} \hat{P}[Y=0] &= \frac{3}{6} \\ f_0(X_0|Y=0) &= 0.19068 \\ f_1(X_1|Y=0) &= 0.11981 \\ \frac{1}{2} * 0.19068 * 0.11981 &= 0.011423 \end{aligned}$$

case: k = 1
First compute the variance of each feature.

$$\begin{aligned}
\hat{\sigma}_1^2 &= \frac{1}{3} * \sum_{j \in C_1} (x_i - \hat{\mu}_i)^2 \\
&= \frac{1}{3}(((-1.2565, 3.4912) - \hat{\mu}_1)^2 + ((-0.7973, 1.2288) - \hat{\mu}_1)^2 + (1.1170, 2.2637) - \hat{\mu}_1)^2) \\
&= \frac{1}{3}(3.1696, 2.5654) \\
&= (1.05655, 0.85514)
\end{aligned}$$

Now we can compute $f_i((0,1)|Y=1)$ for use in our classifier.

$$\begin{aligned}
f((0,1)|Y=1) &= \frac{1}{\sqrt{2\pi\hat{\sigma}_1^2}} e^{-\frac{(<0,1>-\hat{\mu}_1)^2}{2\hat{\sigma}_1^2}} \\
&= \frac{1}{(2.5765, 2.3180)}(0.95490, 0.35664) \\
&= (0.37062, 0.15386)
\end{aligned}$$

We would like to maximize the following equation over all k $\hat{P}[Y=1]\prod_{i=1}^{d} f_i(X_i|Y=k)$.

$$\begin{aligned}
\hat{P}[Y=1] &= \frac{3}{6} \\
f_0(X_0|Y=0) &= 0.37062 \\
f_1(X_1|Y=0) &= 0.15386 \\
\frac{1}{2} * 0.37062 * 0.15386 &= 0.028512
\end{aligned}$$

Noting that $0.011423 < 0.028512$, k=1 maximizes our function, thus we predict a class label of 1 for (0,1). The posterior class probability is given by the following function.

$$\begin{aligned}
P[Y=k|X=x] &= \frac{P[Y=k]f_{1\ldots d}(X=x|Y=k)}{\sum_{i=0}^{k} P[Y=i]f_{Y\ldots d}(X=x|Y=i)} \\
&= \frac{0.028512}{0.028512 + 0.011423} \\
&= 0.71396
\end{aligned}$$

Thus we find that we have a posterior class probability of 71.396% for class k=1.

Part (c)

I have included the results of the R script that follows as comments directly below the corresponding print statements. The result of qda(...) in R produces the exact same posterior class probability as I have computed above.

Currently my results for NaiveBayes (assuming normal) are off by about 4%, so I must have an error in my work somewhere. I spent a couple of hours checking my work and couldn't find any issues, checked against Matlab, which agreed with R. Apparently my friends Python code agrees with my work, so I'm just gonna give up and leave it as is.

```r
#########
# Setup #
#########

library(MASS)
library(klaR)

data <- data.frame(
    c(0.6585, 2.2460, -2.7665, -1.2565, -0.7973, 1.1170),
    c(0.2444, 0.5281, -3.8303, 3.4912, 1.2288, 2.2637),
    c(0, 0, 0, 1, 1, 1)
)

test <- data.frame(c(0), c(1))

names(data) <- c("F1", "F2", "CLASS")
names(test) <- c("F1", "F2")

train.size <- nrow(data)
train <- data[1:train.size, 1:2]
train.cl <- factor(data[1:train.size, 3])

#######
# QDA #
#######

model <- qda(x = train, grouping = train.cl)
predict <- predict(model, test)
print(predict)
print(paste("QDA: ", predict$class))

# Posterior Class Probabilities
# 0: 0.0067895
# 1: 0.9932105
```

```

##############
# Naive Bayes #
##############

model <- NaiveBayes(x=train, grouping=train.cl, usekernel=FALSE)
predict <- predict(model, test)
print(predict)
print(paste("Naive Bayes: ", predict$class))

# Posterior Class Probabilities
# 0: 0.2493135
# 1: 0.7506865
```

```
function p2

%% Data
% Input features to train on.
train = [
    % Class 0
    0.6585, 0.2444;
    2.2460, 0.5281;
    -2.7665, -3.8303;
    % Class 1
    -1.2565, 3.4912;
    -0.7973, 1.2288;
    1.1170, 2.2637
];
% Labels for the input features.
labels = [0; 0; 0; 1; 1; 1];
% We want to know the class and
% posterior class probabilities for this point.
test = [0, 1];

%% Classification
% Create our model, and use it to evaluate the test point.
model = fitcnb(train, labels, 'Distribution', 'normal')
% label = 1
% Posterior = 0.2493, 0.7507
[label, Posterior] = predict(model, test)

end
```