Homework 2: Classification Methods

Chiayun Chang

Question 1

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------------------------- tidyverse 1.3.0 --

## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

## -- Conflicts ------------------------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(broom)
library(rsample)
library(corrplot)
```

```
## corrplot 0.84 loaded
```

```
library(dplyr)
library(ISLR)
library(caret)
```

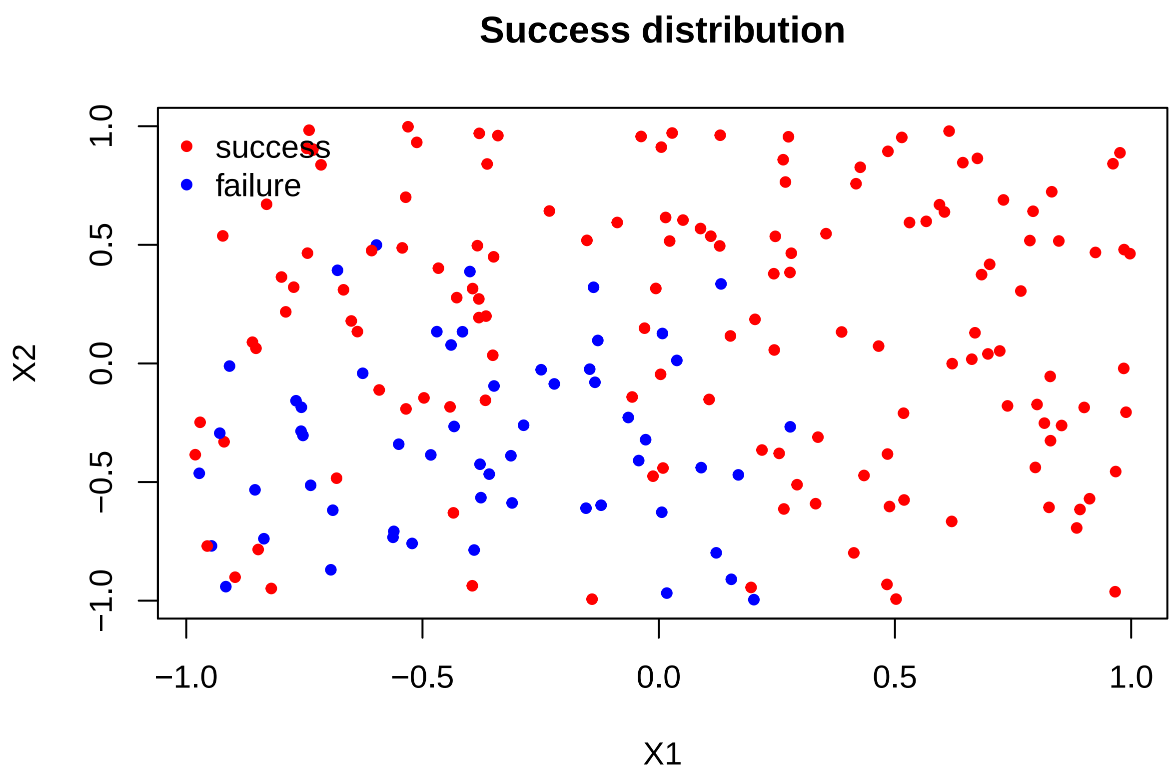
```
## Loading required package: lattice

##
## Attaching package: 'caret'

## The following object is masked from 'package:purrr':
##
##     lift
```

Genrate random uniform data

```
set.seed(1234)
N <- 200
X1 <- runif(N, -1, 1)
X2 <- runif(N, -1, 1)
Y <- X1 + X1^2 + X2 + X2^2 + rnorm(N, 0, 0.5)
Pr_Suc <- exp(Y)/(1+exp(Y))
```

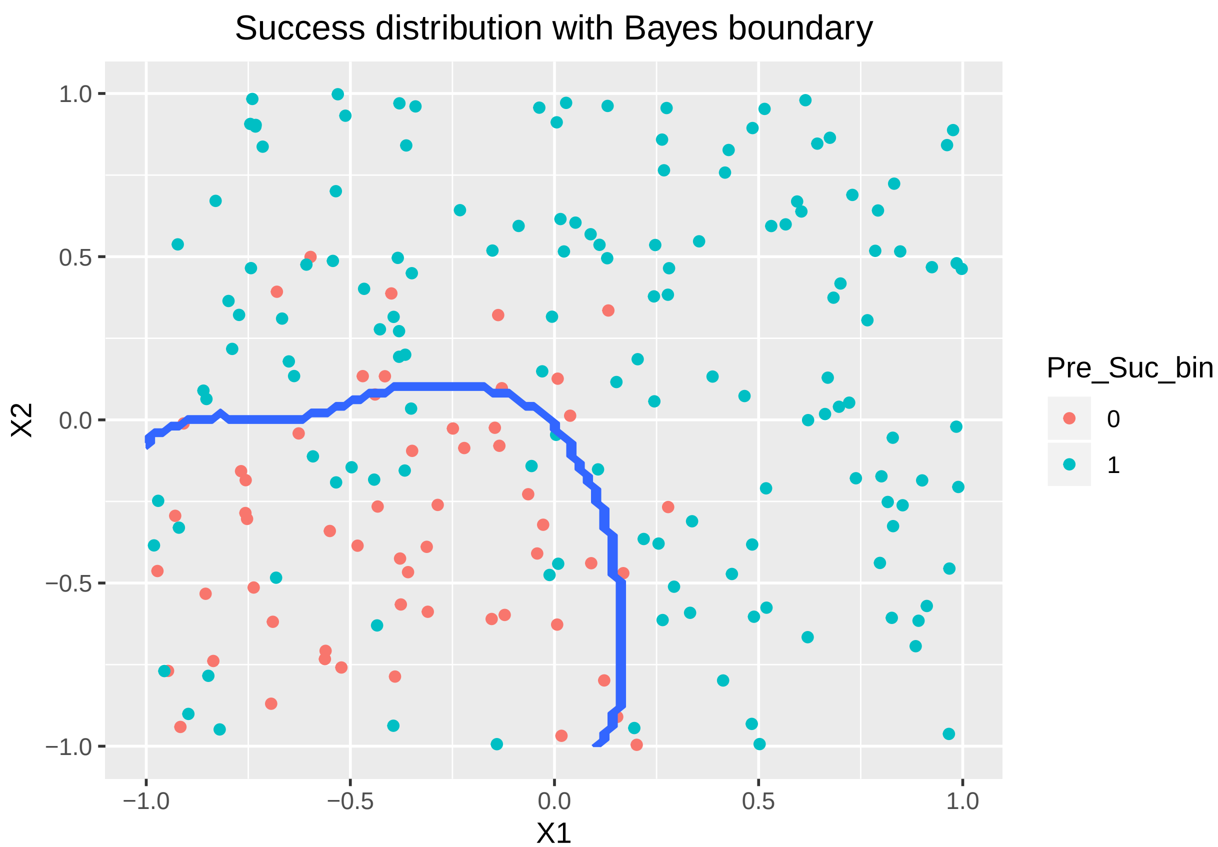
Plot

```r
plot(X1, X2, col = ifelse(Pr_Suc > 0.5, 2, 4), pch = 20, main = 'Success distribution', xlab = 'X1', yla

legend('topleft', c('success', 'failure'), col = c(2, 4), pch = 20, bty = 'n')
```




```r
Pre_Suc_bin <- ifelse(Pr_Suc > 0.5, 1, 0)
X <- as.data.frame(cbind(X1, X2, Pre_Suc_bin)) %>%
  mutate(Pre_Suc_bin = as.factor(Pre_Suc_bin))

train_control <- trainControl(
  method = "cv",
  number = 10
)

q1.nb <- train(
  x = X[,1:2],
  y = X$Pre_Suc_bin,
  method = "nb",
  trControl = train_control
)

x1 <- x2 <- seq(-1, 1, length.out= 100 )
new <- expand.grid(X1 = x1,X2 = x2)
new$Pre_Suc_bin <- predict(q1.nb, newdata = new)

ggplot(X, aes(x = X1, y = X2)) +
```

```r
  geom_point(aes(color = Pre_Suc_bin)) +
  geom_contour(data = new, aes(z = as.numeric(Pre_Suc_bin)))+
  ggtitle('Success distribution with Bayes boundary')+
  theme(plot.title = element_text(hjust = 0.5))
```



Question 2

In [447]:
```
import random
import numpy as np
import pandas as pd
import sklearn.model_selection
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as
LDA
from sklearn.metrics import confusion_matrix
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
as QDA
from tabulate import tabulate
import math
import matplotlib.pyplot as plt
import seaborn as sns
```

In [350]:
```
#Generate data
random.seed(5566)
X1 = np.random.uniform(-1,1,1000)
X2 = np.random.uniform(-1,1,1000)

Y_sim = X1 + X2 + np.random.uniform(0,1,1000)
Y_sim_bin = Y_sim >= 0

X1.shape = (1,1000)
X2.shape = (1,1000)
Y_sim.shape = (1,1000)
Y_sim_bin.shape = (1,1000)
Y_sim_set = np.concatenate((X1, X2, Y_sim_bin), axis=0)
```

In [315]:
```
# Notes:
# Another way to conduct ramdom sampling.
#numpy.random.shuffle(Y_sim_set)
#X_train, X_test, Y_train, Y_test = Y_sim_set[0:2,:700] ,Y_sim_set[0:2,:
300],Y_sim_set[2,:700] ,Y_sim_set[2,:300]
#Y_train.shape = (1,700)
#Y_test.shape = (1,300)
```

In [351]:
```
#Split data
Y_sim_set = np.transpose(Y_sim_set)
df = pd.DataFrame(Y_sim_set)
X = df[[0,1]]
Y = df[2]

X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size = 0.3
, random_state = 38)

X_train = X_train.to_numpy()
Y_train = Y_train.to_numpy()
```

In [353]:
```
#Calculating error rate
Y_pred = lda.predict(X_test)
con_matrix = confusion_matrix(Y_test, Y_pred)
er_rate = (con_matrix[1][0] + con_matrix[0][1])/300
er_rate
```

Out[353]: 0.09

In [409]:
```
def simulation_lda (sd):
    '''
    Return error rate of each simulation.
    Input:
        sd: integer, random seed for the simulation
    Output:
        er_rate: float, (TypeI + Type II)/total number of sample
    '''

    #generate data
    random.seed(sd)
    X1 = np.random.uniform(-1,1,1000)
    X2 = np.random.uniform(-1,1,1000)

    Y_sim = X1 + X2 + np.random.uniform(0,1,1000)
    #Y_sim1 = np.exp(Y_sim) / (1 + np.exp(Y_sim))
    Y_sim_bin = Y_sim >= 0

    X1.shape = (1,1000)
    X2.shape = (1,1000)
    Y_sim.shape = (1,1000)
    Y_sim_bin.shape = (1,1000)
    Y_sim_set = np.concatenate((X1, X2, Y_sim_bin), axis=0)

    #Split data
    Y_sim_set = np.transpose(Y_sim_set)
    df = pd.DataFrame(Y_sim_set)
    X = df[[0,1]]
    Y = df[2]
    X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size =
0.3, random_state = 38)

    X_train = X_train.to_numpy()
    Y_train = Y_train.to_numpy()

    #perform LDA
    lda = LDA(n_components=1)
    lda.fit(X_train, Y_train)

    Y_pred_test = lda.predict(X_test)
    con_matrix = confusion_matrix(Y_test, Y_pred_test)
    er_rate_test = (con_matrix[1][0] + con_matrix[0][1])/300

    Y_pred_train = lda.predict(X_train)
    con_matrix2 = confusion_matrix(Y_train, Y_pred_train)
    er_rate_train = (con_matrix2[1][0] + con_matrix2[0][1])/700

    return [er_rate_test, er_rate_train]
```

In [410]:
```
# do 1000 LDA simulations
er_rate_LDA_train = []
er_rate_LDA_test = []
sd = 3
for i in range (1000):
    er_rate_LDA_train.append(simulation_lda(sd)[1])
    er_rate_LDA_test.append(simulation_lda(sd)[0])
    sd+=1
```

In [428]:
```
def simulation_qda (sd):
    '''
    Return error rate of each simulation.
    Input:
        sd: integer, random seed for the simulation
    Output:
        er_rate: float, (TypeI + Type II)/total number of sample
    '''

    #generate data
    random.seed(sd)
    X1 = np.random.uniform(-1,1,1000)
    X2 = np.random.uniform(-1,1,1000)

    Y_sim = X1 + X1*X1 +X2 + X2*X2 + np.random.uniform(0,1,1000)

    Y_sim_bin = Y_sim >= 0

    X1.shape = (1,1000)
    X2.shape = (1,1000)
    Y_sim.shape = (1,1000)
    Y_sim_bin.shape = (1,1000)
    Y_sim_set = np.concatenate((X1, X2, Y_sim_bin), axis=0)

    #Split data
    Y_sim_set = np.transpose(Y_sim_set)
    df = pd.DataFrame(Y_sim_set)
    X = df[[0,1]]
    Y = df[2]
    X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size =
0.3, random_state = 38)

    X_train = X_train.to_numpy()
    Y_train = Y_train.to_numpy()

    #perform LDA
    qda = QDA()
    qda.fit(X_train, Y_train)

    Y_pred_test = qda.predict(X_test)
    con_matrix = confusion_matrix(Y_test, Y_pred_test)
    er_rate__test = (con_matrix[1][0] + con_matrix[0][1])/300

    Y_pred_train = qda.predict(X_train)
    con_matrix2 = confusion_matrix(Y_train, Y_pred_train)
    er_rate__train = (con_matrix2[1][0] + con_matrix2[0][1])/700



    return [er_rate_test, er_rate_train]
```

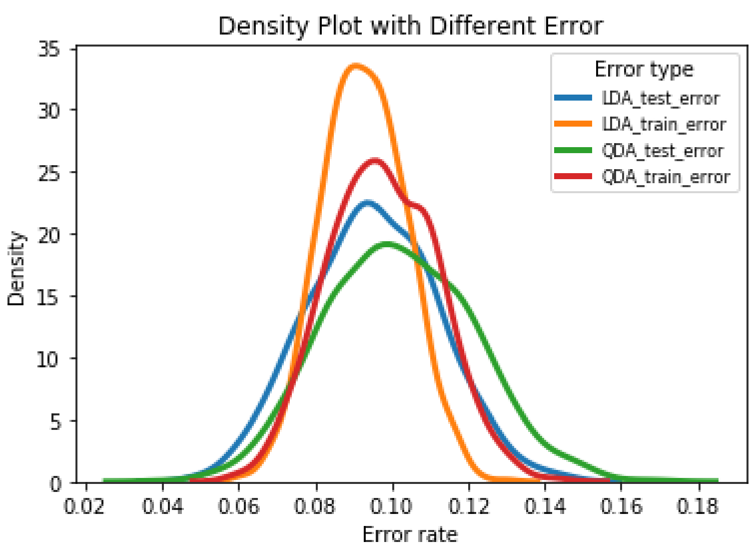
In [429]:
```
# do 1000 QDA simulations, with the same seed
er_rate_QDA_train = []
er_rate_QDA_test = []
sd = 3
for i in range (1000):
    er_rate_QDA_train.append(simulation_qda(sd)[1])
    er_rate_QDA_test.append(simulation_qda(sd)[0])
    sd+=1
```

In [463]:
```
print(tabulate([['LDA_test_error', np.mean(er_rate_LDA_test)],
 ['LDA_train_error', np.mean(er_rate_LDA_train)],
 ['QDA_test_error', np.mean(er_rate_QDA_test)],
 ['QDA_train_error', np.mean(er_rate_QDA_train)]],
             headers = ['Error type', 'Error rate'] ))
```

```
Error type         Error rate
---------------  ------------
LDA_test_error      0.0952233
LDA_train_error     0.0927971
QDA_test_error      0.10203
QDA_train_error     0.0977114
```

In [459]:
```
labels = ['LDA_test_error','LDA_train_error','QDA_test_error','QDA_train
_error']
i = 0
er_array = [[er_rate_LDA_test],[er_rate_LDA_train],[er_rate_QDA_test],[e
r_rate_QDA_train]]
for l in labels:

    sns.distplot(er_array[i], hist = False, kde = True,
                 kde_kws = {'linewidth': 3},
                 label = l)
    i+=1

plt.legend(prop={'size': 8}, title = 'Error type')
plt.title('Density Plot with Different Error')
plt.xlabel('Error rate')
plt.ylabel('Density')
```

Out[459]:
```
Text(0, 0.5, 'Density')
```



Description

In this case, if the Baysien decision boundary is linear, for the training data, QDA will be able to fit better than LDA because it can actually account for the variances that the LDA cannot. However, this will also mean that QDA is more likely to overfit, and the LDA will outperform QDA in predicting. This make sense because the true boundary is linear to begin with.

The mean of the types of error are not far from eachother. QDA has higher testing rate and training rate than LDA, in support of the observation mentioned above.

Question 3

In [4]:
```
import random
import numpy as np
import pandas as pd
import sklearn.model_selection
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as
LDA
from sklearn.metrics import confusion_matrix
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
as QDA
from tabulate import tabulate
import math
import matplotlib.pyplot as plt
import seaborn as sns
```

In [4]:
```
def simulation_lda (sd):
    '''
    Return error rate of each simulation.
    Input:
        sd: integer, random seed for the simulation
    Output:
        er_rate: list of floats, (TypeI + Type II)/total number of sampl
e, for training and testing error
    '''

    #generate data
    random.seed(sd)
    X1 = np.random.uniform(-1,1,1000)
    X2 = np.random.uniform(-1,1,1000)

    Y_sim = X1 + X1*X1 + X2 + X2*X2 + np.random.uniform(0,1,1000)
    Y_sim_bin = Y_sim >= 0

    X1.shape = (1,1000)
    X2.shape = (1,1000)
    Y_sim.shape = (1,1000)
    Y_sim_bin.shape = (1,1000)
    Y_sim_set = np.concatenate((X1, X2, Y_sim_bin), axis=0)

    #Split data
    Y_sim_set = np.transpose(Y_sim_set)
    df = pd.DataFrame(Y_sim_set)
    X = df[[0,1]]
    Y = df[2]
    X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size =
0.3, random_state = 38)

    X_train = X_train.to_numpy()
    Y_train = Y_train.to_numpy()

    #perform LDA
    lda = LDA(n_components=1)
    lda.fit(X_train, Y_train)

    Y_pred_test = lda.predict(X_test)
    con_matrix = confusion_matrix(Y_test, Y_pred_test)
    er_rate_test = (con_matrix[1][0] + con_matrix[0][1])/300

    Y_pred_train = lda.predict(X_train)
    con_matrix2 = confusion_matrix(Y_train, Y_pred_train)
    er_rate_train = (con_matrix2[1][0] + con_matrix2[0][1])/700

    return [er_rate_test, er_rate_train]
```

In [5]:
```
def simulation_qda (sd):
    '''
    Return error rate of each simulation.
    Input:
        sd: integer, random seed for the simulation
    Output:
        er_rate: list of floats, (TypeI + Type II)/total number of sampl
e, for training and testing error
    '''

    #generate data
    random.seed(sd)
    X1 = np.random.uniform(-1,1,1000)
    X2 = np.random.uniform(-1,1,1000)

    Y_sim = X1 + X1*X1 +X2 + X2*X2 + np.random.uniform(0,1,1000)

    Y_sim_bin = Y_sim >= 0

    X1.shape = (1,1000)
    X2.shape = (1,1000)
    Y_sim.shape = (1,1000)
    Y_sim_bin.shape = (1,1000)
    Y_sim_set = np.concatenate((X1, X2, Y_sim_bin), axis=0)

    #Split data
    Y_sim_set = np.transpose(Y_sim_set)
    df = pd.DataFrame(Y_sim_set)
    X = df[[0,1]]
    Y = df[2]
    X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size =
0.3, random_state = 38)

    X_train = X_train.to_numpy()
    Y_train = Y_train.to_numpy()

    #perform LDA
    qda = QDA()
    qda.fit(X_train, Y_train)

    Y_pred_test = qda.predict(X_test)
    con_matrix = confusion_matrix(Y_test, Y_pred_test)
    er_rate__test = (con_matrix[1][0] + con_matrix[0][1])/300

    Y_pred_train = qda.predict(X_train)
    con_matrix2 = confusion_matrix(Y_train, Y_pred_train)
    er_rate__train = (con_matrix2[1][0] + con_matrix2[0][1])/700



    return [er_rate_test, er_rate_train]
```

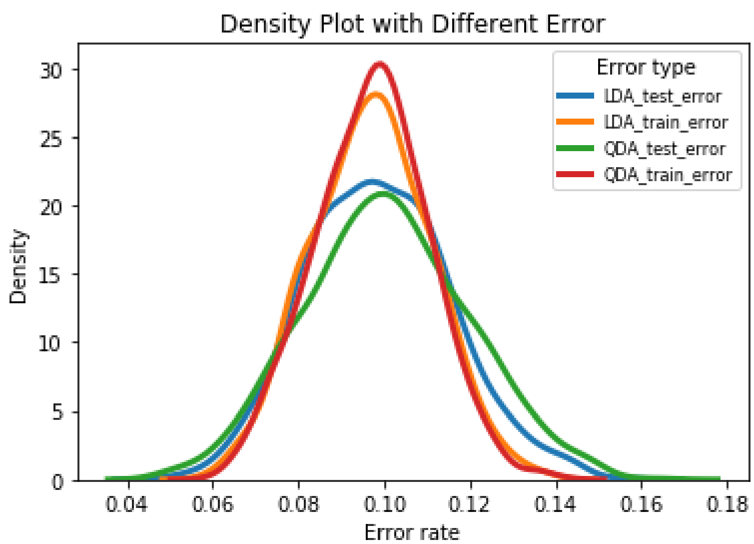
In [6]:
```
# do 1000 LDA simulations
er_rate_LDA_train = []
er_rate_LDA_test = []
sd = 3
for i in range (1000):
    er_rate_LDA_train.append(simulation_lda(sd)[1])
    er_rate_LDA_test.append(simulation_lda(sd)[0])
    sd+=1

# do 1000 QDA simulations, with the same seed
er_rate_QDA_train = []
er_rate_QDA_test = []
sd = 3
for i in range (1000):
    er_rate_QDA_train.append(simulation_qda(sd)[1])
    er_rate_QDA_test.append(simulation_qda(sd)[0])
    sd+=1
```

In [10]:
```
print(tabulate([['LDA_test_error', np.mean(er_rate_LDA_test)],
 ['LDA_train_error', np.mean(er_rate_LDA_train)],
 ['QDA_test_error', np.mean(er_rate_QDA_test)],
 ['QDA_train_error', np.mean(er_rate_QDA_train)]],
             headers = ['Error type', 'Error rate'] ))
```

```
Error type         Error rate
---------------  ------------
LDA_test_error      0.0987033
LDA_train_error     0.0969529
QDA_test_error      0.10063
QDA_train_error     0.09715
```

In [9]:
```
labels = ['LDA_test_error','LDA_train_error','QDA_test_error','QDA_train
_error']
i = 0
er_array = [[er_rate_LDA_test],[er_rate_LDA_train],[er_rate_QDA_test],[e
r_rate_QDA_train]]
for l in labels:

    sns.distplot(er_array[i], hist = False, kde = True,
                 kde_kws = {'linewidth': 3},
                 label = l)
    i+=1

plt.legend(prop={'size': 8}, title = 'Error type')
plt.title('Density Plot with Different Error')
plt.xlabel('Error rate')
plt.ylabel('Density')
```

Out[9]:
```
Text(0, 0.5, 'Density')
```



Description

As shown above, the mean of the four types of error rates do not differ much. The differnce between testing error and training error for QDA and LDA shows that, QDA might have some issues with overfitting. QDA's training error is well lower than its testing error, whereas the differece between LDA's error rate is smaller.

Question 4

In [1]:
```
import random
import numpy as np
import pandas as pd
import sklearn.model_selection
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.metrics import confusion_matrix
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as
from tabulate import tabulate
import math
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
def simulation_qda (sd,n):

    err_rate = []

    for i in range(1000):
        X1 = np.random.uniform(-1,1,n)
        X2 = np.random.uniform(-1,1,n)

        Y_sim = X1 + X2 + np.random.uniform(0,1,n)
        Y_sim_bin = Y_sim >= 0

        X1.shape = (1,n)
        X2.shape = (1,n)
        Y_sim.shape = (1,n)
        Y_sim_bin.shape = (1,n)
        Y_sim_set = np.concatenate((X1, X2, Y_sim_bin), axis=0)

        Y_sim_set = np.transpose(Y_sim_set)
        df = pd.DataFrame(Y_sim_set)
        X = df[[0,1]]
        Y = df[2]
        X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size

        X_train = X_train.to_numpy()
        Y_train = Y_train.to_numpy()

        qda = QDA()
        qda.fit(X_train, Y_train)

        Y_pred_test = qda.predict(X_test)
        con_matrix = confusion_matrix(Y_test, Y_pred_test)
        er_rate__test = (con_matrix[1][0] + con_matrix[0][1])/n /0.3
        err_rate.append(er_rate__test)
        sd +=1

    return err_rate
```

In [17]:
```
err_rate_100 = simulation_qda (78,100)
```

In [18]:
```
err_rate_1000 = simulation_qda (78,1000)
```

In [5]:
```
err_rate_10000 = simulation_qda (78,10000)
```

In [6]:
```
err_rate_100000 = simulation_qda (78,100000)
```

In [14]:
```
def simulation_lda (sd,n):

    err_rate = []

    for i in range(1000):
        X1 = np.random.uniform(-1,1,n)
        X2 = np.random.uniform(-1,1,n)

        Y_sim = X1 + X2 + np.random.uniform(0,1,n)
        Y_sim_bin = Y_sim >= 0

        X1.shape = (1,n)
        X2.shape = (1,n)
        Y_sim.shape = (1,n)
        Y_sim_bin.shape = (1,n)
        Y_sim_set = np.concatenate((X1, X2, Y_sim_bin), axis=0)

        Y_sim_set = np.transpose(Y_sim_set)
        df = pd.DataFrame(Y_sim_set)
        X = df[[0,1]]
        Y = df[2]
        X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size

        X_train = X_train.to_numpy()
        Y_train = Y_train.to_numpy()

        lda = LDA()
        lda.fit(X_train, Y_train)

        Y_pred_test = lda.predict(X_test)
        con_matrix = confusion_matrix(Y_test, Y_pred_test)
        er_rate__test = (con_matrix[1][0] + con_matrix[0][1])/n /0.3
        err_rate.append(er_rate__test)
        sd +=1

    return err_rate
```

In [23]:
```
lerr_rate_100 = simulation_lda (78,100)
```

In [25]:
```
lerr_rate_1000 = simulation_lda (78,1000)
```

In [26]:
```
lerr_rate_10000 = simulation_lda (78,10000)
```

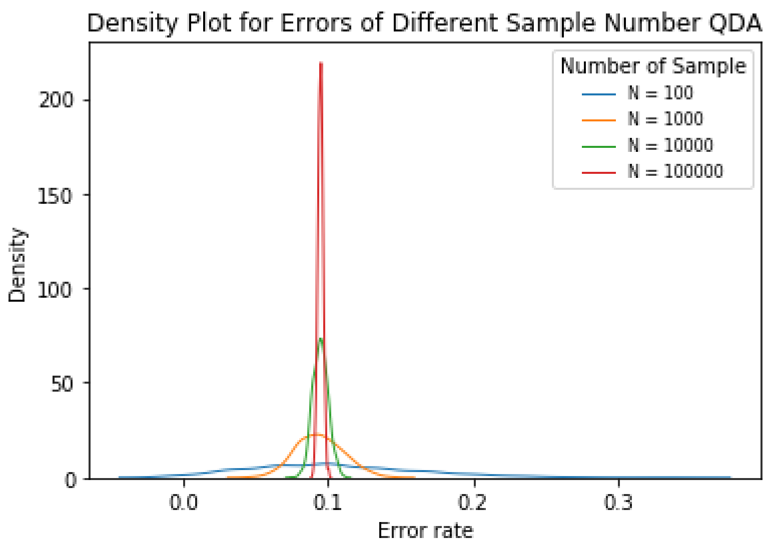
In [27]:
```
lerr_rate_100000 = simulation_lda (78,100000)
```

In [28]:
```
print(tabulate(
[
 ['N = 100', np.mean(err_rate_100)],
 ['N = 1000', np.mean(err_rate_1000)],
 ['N = 10000', np.mean(err_rate_10000)],
 ['N = 100000', np.mean(err_rate_100000)]

],
  headers = ['Number of Sample for QDAs', 'Error rate'] ))
print(tabulate(
[
 ['N = 100', np.mean(lerr_rate_100)],
 ['N = 1000', np.mean(lerr_rate_1000)],
 ['N = 10000', np.mean(lerr_rate_10000)],
 ['N = 100000', np.mean(lerr_rate_100000)]

],
  headers = ['Number of Sample for LDAs', 'Error rate'] ))
```

```
Number of Sample for QDAs      Error rate
---------------------------  ------------
N = 100                          0.103667
N = 1000                         0.09431
N = 10000                        0.0948723
N = 100000                       0.0949423
Number of Sample for LDAs      Error rate
---------------------------  ------------
N = 100                          0.101367
N = 1000                         0.0946167
N = 10000                        0.0939303
N = 100000                       0.0939042
```

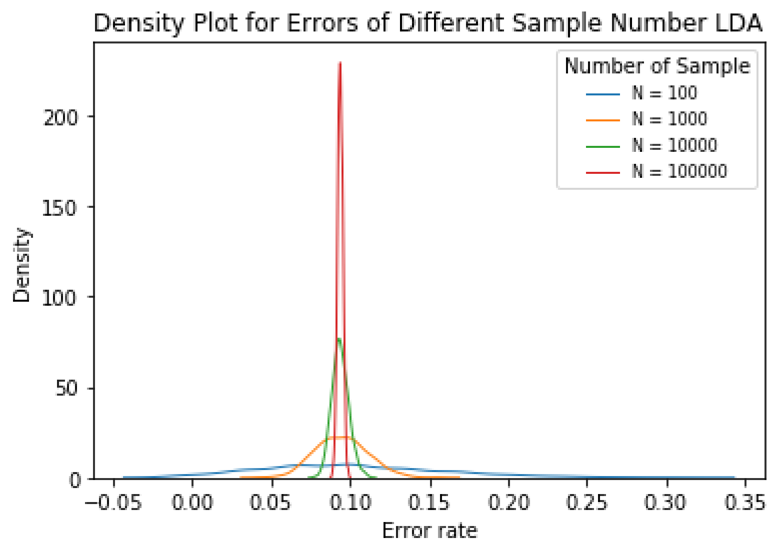
In [30]:
```
labels = ['N = 100', 'N = 1000', 'N = 10000', 'N = 100000']
i = 0
er_array = [[err_rate_100],[err_rate_1000],[err_rate_10000],[err_rate_10000

for l in labels:

    sns.distplot(er_array[i], hist = False, kde = True,
                 kde_kws = {'linewidth': 1},
                 label = l)
    i+=1

plt.legend(prop={'size': 8}, title = 'Number of Sample')
plt.title('Density Plot for Errors of Different Sample Number QDA')
plt.xlabel('Error rate')
plt.ylabel('Density')
```

Out[30]: Text(0, 0.5, 'Density')



In [31]:
```
labels = ['N = 100', 'N = 1000', 'N = 10000', 'N = 100000']
i = 0
er_array = [[lerr_rate_100],[lerr_rate_1000],[lerr_rate_10000],[lerr_rate_1

for l in labels:

    sns.distplot(er_array[i], hist = False, kde = True,
                 kde_kws = {'linewidth': 1},
                 label = l)
    i+=1

plt.legend(prop={'size': 8}, title = 'Number of Sample')
plt.title('Density Plot for Errors of Different Sample Number LDA')
plt.xlabel('Error rate')
plt.ylabel('Density')
```

Out[31]: Text(0, 0.5, 'Density')



Description

The non linear Bayes boundary should give the more flexible QDA an edge in fitting. As the number of samples goes up, with the same number of predictors, the more flexible QDA would perform better. This is becuase the model would be able to account for all the variances.

The results above, however, does not show that QDA significantly outperforms the LDA. To get a more notable difference in performance, we might have to try with a simulated data where the classification boundary is even more non-linear.

Question 5

In [53]:
```
import random
import numpy as np
import pandas as pd
import sklearn.model_selection
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.metrics import confusion_matrix
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as
from tabulate import tabulate
import math
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.naive_bayes import GaussianNB as NB
from sklearn.linear_model import LogisticRegression as LR
from sklearn.neighbors import KNeighborsClassifier as KNN
from sklearn.metrics import auc
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
```

In [4]:
```
mental_health = pd.read_csv('mental_health.csv')
```

In [5]:
```
df = mental_health.dropna()
df
```

Out[5]:

	vote96	mhealth_sum	age	educ	black	female	married	inc10
0	1.0	0.0	60.0	12.0	0	0	0.0	4.8149
2	1.0	1.0	36.0	12.0	0	0	1.0	8.8273
3	0.0	7.0	21.0	13.0	0	0	0.0	1.7387
7	0.0	6.0	29.0	13.0	0	0	0.0	10.6998
11	1.0	1.0	41.0	15.0	1	1	1.0	8.8273
...	...	...	...	...	...	...	...	...
2822	1.0	2.0	37.0	14.0	0	0	1.0	5.8849
2823	1.0	2.0	30.0	12.0	0	1	1.0	3.4774
2828	1.0	1.0	40.0	12.0	0	1	0.0	1.7387
2829	1.0	2.0	73.0	6.0	0	0	1.0	2.2737
2830	1.0	4.0	47.0	12.0	0	0	0.0	3.4774

1165 rows × 8 columns

In [6]:
```
predictors = ['mhealth_sum','age','educ','black','female','married','inc10'
X_train, X_test, Y_train, Y_test = train_test_split(df[predictors], df['vot
```

In [7]:
```
#Logistic Regression
lr = LR()
lr.fit(X_train,Y_train)
lr_err_rate = 1-lr.score(X_test, Y_test)
```

In [51]:
```
#LDA
lda = LDA()
lda.fit(X_train,Y_train)
lda_err_rate = 1-lda.score(X_test, Y_test)

#QDA
qda = QDA()
qda.fit(X_train,Y_train)
qda_err_rate = 1-qda.score(X_test, Y_test)
```

In [10]:
```
#Naive Bayes
nb = NB()
nb.fit(X_train, Y_train)
nb_err_rate = 1-nb.score(X_test, Y_test)
```

In [23]:
```
# KNN
def simulation_KNN(n):
    Knn = KNN(n)
    Knn.fit(X_train,Y_train)
    return 1-Knn.score(X_test, Y_test)

knn_err_1_10 = []

for n in range(1,11):
    knn_err_1_10.append(simulation_KNN(n))
```

In [27]:
```
print(tabulate(
[
['Logistic Regression', lr_err_rate],
['LDA ', lda_err_rate],
['QDA ', qda_err_rate],
['Naive Bayes', nb_err_rate],
['KNN(n=1)', knn_err_1_10[0]],
['KNN(n=2)', knn_err_1_10[1]],
['KNN(n=3)', knn_err_1_10[2]],
['KNN(n=4)', knn_err_1_10[3]],
['KNN(n=5)', knn_err_1_10[4]],
['KNN(n=6)', knn_err_1_10[5]],
['KNN(n=7)', knn_err_1_10[6]],
['KNN(n=8)', knn_err_1_10[7]],
['KNN(n=9)', knn_err_1_10[8]],
['KNN(n=10)',knn_err_1_10[9]]
],
headers = ['Model test error', 'Error Rate']))
```

```
Model test error       Error Rate
-------------------  ------------
Logistic Regression      0.265714
LDA                      0.262857
QDA                      0.28
Naive Bayes              0.268571
KNN(n=1)                 0.337143
KNN(n=2)                 0.408571
KNN(n=3)                 0.351429
KNN(n=4)                 0.374286
KNN(n=5)                 0.342857
KNN(n=6)                 0.354286
KNN(n=7)                 0.337143
KNN(n=8)                 0.328571
KNN(n=9)                 0.328571
KNN(n=10)                0.302857
```

In [34]:
```
# KNN retrun model
def simulation_KNN(n):
    Knn = KNN(n)
    Knn.fit(X_train,Y_train)
    return Knn

knn_models = []

for n in range(1,11):
    knn_models.append(simulation_KNN(n))
```

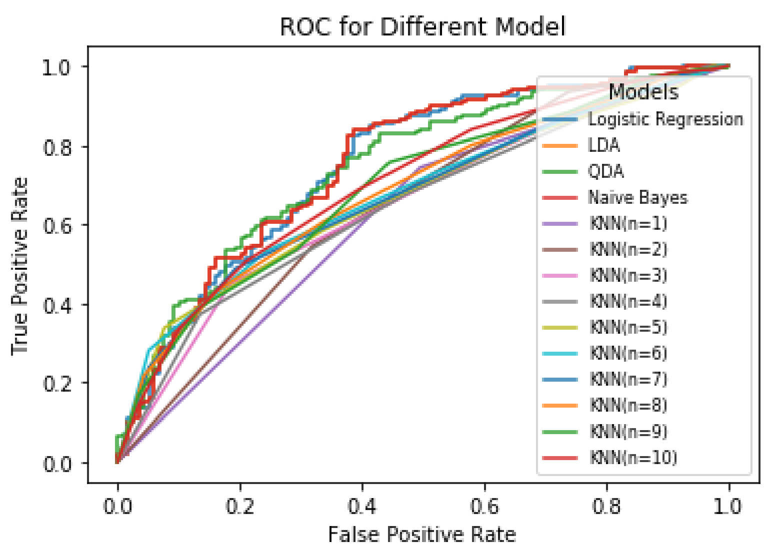
In [44]:
```
lr_proba = lr.predict_proba(X_test)
lr_auc = roc_auc_score(Y_test, lr_proba[:,1])

lda_proba = lda.predict_proba(X_test)
lda_auc = roc_auc_score(Y_test, lda_proba[:,1])

qda_proba = qda.predict_proba(X_test)
qda_auc = roc_auc_score(Y_test, qda_proba[:,1])

nb_proba = lda.predict_proba(X_test)
nb_auc = roc_auc_score(Y_test, nb_proba[:,1])

knn_aucs =[]
knn_probas = []
for n in range(1,11):
    knn = simulation_KNN(n)
    knn_proba = knn.predict_proba(X_test)
    knn_auc = roc_auc_score(Y_test, knn_proba[:,1])
    knn_aucs.append(knn_auc)
    knn_probas.append(knn_proba[:,1])
```

In [45]:
```
print(tabulate(
[
['Logistic Regression', lr_auc],
['LDA ', lda_auc],
['QDA ', qda_auc],
['Naive Bayes', nb_auc],
['KNN(n=1)', knn_aucs[0]],
['KNN(n=2)', knn_aucs[1]],
['KNN(n=3)', knn_aucs[2]],
['KNN(n=4)', knn_aucs[3]],
['KNN(n=5)', knn_aucs[4]],
['KNN(n=6)', knn_aucs[5]],
['KNN(n=7)', knn_aucs[6]],
['KNN(n=8)', knn_aucs[7]],
['KNN(n=9)', knn_aucs[8]],
['KNN(n=10)',knn_aucs[9]]
],
headers = ['Model', 'AUC']))
```

```
Model                     AUC
-------------------  --------
Logistic Regression  0.753938
LDA                  0.756484
QDA                  0.747608
Naive Bayes          0.756484
KNN(n=1)             0.624395
KNN(n=2)             0.650169
KNN(n=3)             0.657372
KNN(n=4)             0.650969
KNN(n=5)             0.668758
KNN(n=6)             0.68049
KNN(n=7)             0.677944
KNN(n=8)             0.686511
KNN(n=9)             0.692932
KNN(n=10)            0.707392
```

In [50]:
```
models = [('Logistic Regression',lr_proba[:,1]),
          ('LDA ', lda_proba[:,1]) ,
          ('QDA ', qda_proba[:,1]) ,
          ('Naive Bayes', nb_proba[:,1]),
          ('KNN(n=1)', knn_probas[0]),
          ('KNN(n=2)', knn_probas[1]),
          ('KNN(n=3)', knn_probas[2]),
          ('KNN(n=4)', knn_probas[3]),
          ('KNN(n=5)', knn_probas[4]),
          ('KNN(n=6)', knn_probas[5]),
          ('KNN(n=7)', knn_probas[6]),
          ('KNN(n=8)', knn_probas[7]),
          ('KNN(n=9)', knn_probas[8]),
          ('KNN(n=10)',knn_probas[9])]


for m in models:

    fpr,tpr, a = roc_curve(Y_test, m[1])
    plt.plot(fpr, tpr, label = m[0])


plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC for Different Model')
plt.legend(prop={'size': 8}, title = 'Models')
```

Out[50]: <matplotlib.legend.Legend at 0x1245fa610>



Description

Logistic regression, LDA, Naive Bayes have the lowerst test error rate. In general, the closer the ROC is to the top right, i.e. having higher specitivity and higher sensitivity, the better. As shown above, the Naive Bayes and logistic regression model performs better than the rest.

ROCs of LDA, logistic regression and Naive bayes tagled at some point. It could be harder to tell which is closer to the top left. To address this, it will be very useful to also look at the AUCs. Naive Bayes has the largest AUC, meaning it is closer to the top left than other models. In this sense, Naive Bayes is the overal best model.