Homework 2

February 2, 2020

Student: Dimitrios Tanoglidis

Due: Sunday, February 2, 2020

[22]:
```python
#Import stuff
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import rcParams
#import seaborn as sns
rcParams['font.family'] = 'serif'

# Adjust rc parameters to make plots pretty
def plot_pretty(dpi=200, fontsize=8):

    import matplotlib.pyplot as plt

    plt.rc("savefig", dpi=dpi)
    plt.rc('text', usetex=True)
    plt.rc('font', size=fontsize)
    plt.rc('xtick', direction='in')
    plt.rc('ytick', direction='in')
    plt.rc('xtick.major', pad=10)
    plt.rc('xtick.minor', pad=5)
    plt.rc('ytick.major', pad=10)
    plt.rc('ytick.minor', pad=5)
    plt.rc('lines', dotted_pattern = [0.5, 1.1])

    return
plot_pretty()
```

1 The Bayes Classifier

a) Random generator seed

[23]:
```python
np.random.seed(seed=42)
```

b) Uniform dataset X_1, X_2 where X_1, X_2 are random uniform variables between $[-1, 1]$.

[24]:
```
N = 200
X_1 = np.random.uniform(-1.0,+1.0,200)
X_2 = np.random.uniform(-1.0,+1.0,200)
```

c) Calculate $Y = X_1 + X_1^2 + X_2 + X_2^2 + \epsilon$

First create the error term $\epsilon \sim N(\mu = 0, \sigma^2 = 0.25)$.

[25]:
```
mu = 0.0 # mean
sig = np.sqrt(0.25) # standard deviation
err = np.random.normal(mu, sig, 200)
```

[26]:
```
Y = X_1 + X_1**2.0 + X_2 + X_2**2.0 + err
```

d) If we treat Y as the log-odds of sucess, then we can calculate the probability of success as:

$$Y = \log\left(\frac{P}{1-P}\right) \Rightarrow \boxed{P = \frac{e^Y}{1+e^Y}} \tag{1}$$

[27]:
```
Prob = np.exp(Y)/(1.0 + np.exp(Y))
```

e + f + g +h) Here I calculate the Bayes decision boundary and create the plot as described in the prompt.

First, Create a meshgrid over the X_1 - X_2 space

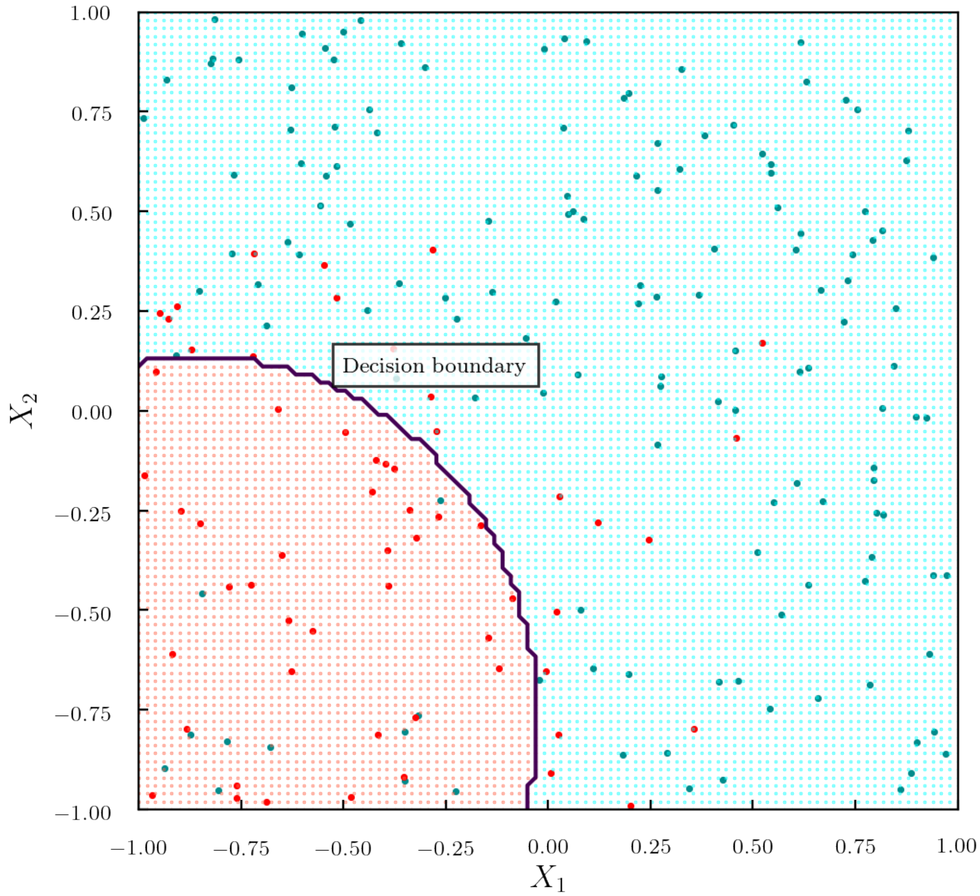
[28]:
```
x1 = np.linspace(-1.0,1.0,100)
x2 = np.linspace(-1.0,1.0,100)
x1g, x2g = np.meshgrid(x1,x2)
```

Now train the Bayes classifier based on the above training set and predict on the grid. Use the scikit implementation assuming Gaussian likelihood of features (Gaussian NB).

[29]:
```
from sklearn.naive_bayes import GaussianNB

# Define a feature matrix of the training set
x_ft_tr = np.zeros([200,2])
x_ft_tr[:,0] = X_1;x_ft_tr[:,1] = X_2
# Define classes
y_cl = np.zeros(200)
y_cl[Prob>0.5]  = 1 #Set class equal to "1" if Prob > 0.5

# Define feature matrix of the grid space
x_ft_gr = np.zeros([10000,2])
x_ft_gr[:,0] = x1g.ravel();x_ft_gr[:,1] = x2g.ravel()

# Fit on training data and predict on grid data
```

```
gnb = GaussianNB()
y_pred = gnb.fit(x_ft_tr, y_cl).predict(x_ft_gr)

#Reshape everything
y_pr = y_pred.reshape(100,100)
```

[33]:
```
import seaborn as sns
sns.reset_orig()
plt.figure(figsize=(5.5,5.5))

# Print trainin data
plt.scatter(X_1[Prob>0.5],X_2[Prob>0.5], c='darkcyan', s= 2.5)
plt.scatter(X_1[Prob<0.5],X_2[Prob<0.5], c='r', s=2.5)

# Print grid
plt.scatter(x1g[y_pr>0.5],x2g[y_pr>0.5], s=0.05, c='cyan')
plt.scatter(x1g[y_pr<0.5],x2g[y_pr<0.5], s=0.05, c='tomato')

#plot decision boundary
plt.contour(x1g,x2g,y_pr,levels=1)

plt.text(-0.5,0.1, 'Decision boundary', bbox=dict(facecolor='w', alpha=0.8))

plt.xlabel('$X_1$',fontsize=12);plt.ylabel('$X_2$',fontsize=12)
plt.xlim(-1.0,1.0)
plt.ylim(-1.0,1.0)
plt.show()
```



2 Exploring Simulated Differences between LDA and QDA

2.1 Linear Bayes decision boundary

a) Repeating the process

[10]:
```
# Import train/test split, LDA, QDA from scikit-learn
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QDA
# Despite the name, the one below gives the error rate
from sklearn.metrics import zero_one_loss as Error_Rate
```

[80]:
```
N_rep = 1000 # Number of repetitions
np.random.seed(seed=2046)
# Create arrays to save the error rates of the two models
# Initialize them here
```

```
Err_LDA_train = np.zeros(N_rep) # LDA on train set
Err_LDA_test = np.zeros(N_rep) # LDA on test set
Err_QDA_train = np.zeros(N_rep) #QDA on train set
Err_QDA_test = np.zeros(N_rep) #QDA on test set

for i in range(N_rep):
    # Simulate X1, X2
    N_obs = 1000 # Number of observations
    X_1 = np.random.uniform(-1.0,+1.0,N_obs)
    X_2 = np.random.uniform(-1.0,+1.0,N_obs)
    # Simulate error
    err = np.random.normal(0,1.0,N_obs)

    # Define simulated Y
    Y = X_1 + X_2 + err

    # Now combine X_1, X_2 into a feature matrix
    X_ft = np.zeros([N_obs,2])
    X_ft[:,0] = X_1; X_ft[:,1] = X_2
    # Make Y binary
    Y[Y>0.0] = 1.0
    Y[Y<0.0] = - 1.0
    # ========================================
    # ========================================
    # Split into train/test
    X_train, X_test, y_train, y_test = train_test_split(X_ft, Y,
                                                         test_size=0.30,␣
↪random_state=42)

    # Fit LDA and QDA
    lda_model = LDA()
    lda_model.fit(X_train,y_train)
    qda_model = QDA()
    qda_model.fit(X_train,y_train)
    # =======================================
    # =======================================
    # Predictions now

    # Predict on the training set
    y_pr_LDA_train = lda_model.predict(X_train)
    y_pr_QDA_train = qda_model.predict(X_train)

    # Predict on the test set
    y_pr_LDA_test = lda_model.predict(X_test)
    y_pr_QDA_test = qda_model.predict(X_test)
    # ========================================
    # ========================================
```

```
    # Calculate LDA and QDA training and testing error rates
    Err_LDA_train[i] = Error_Rate(y_train, y_pr_LDA_train)
    Err_LDA_test[i] = Error_Rate(y_test, y_pr_LDA_test)

    Err_QDA_train[i] = Error_Rate(y_train, y_pr_QDA_train)
    Err_QDA_test[i] = Error_Rate(y_test, y_pr_QDA_test)
```

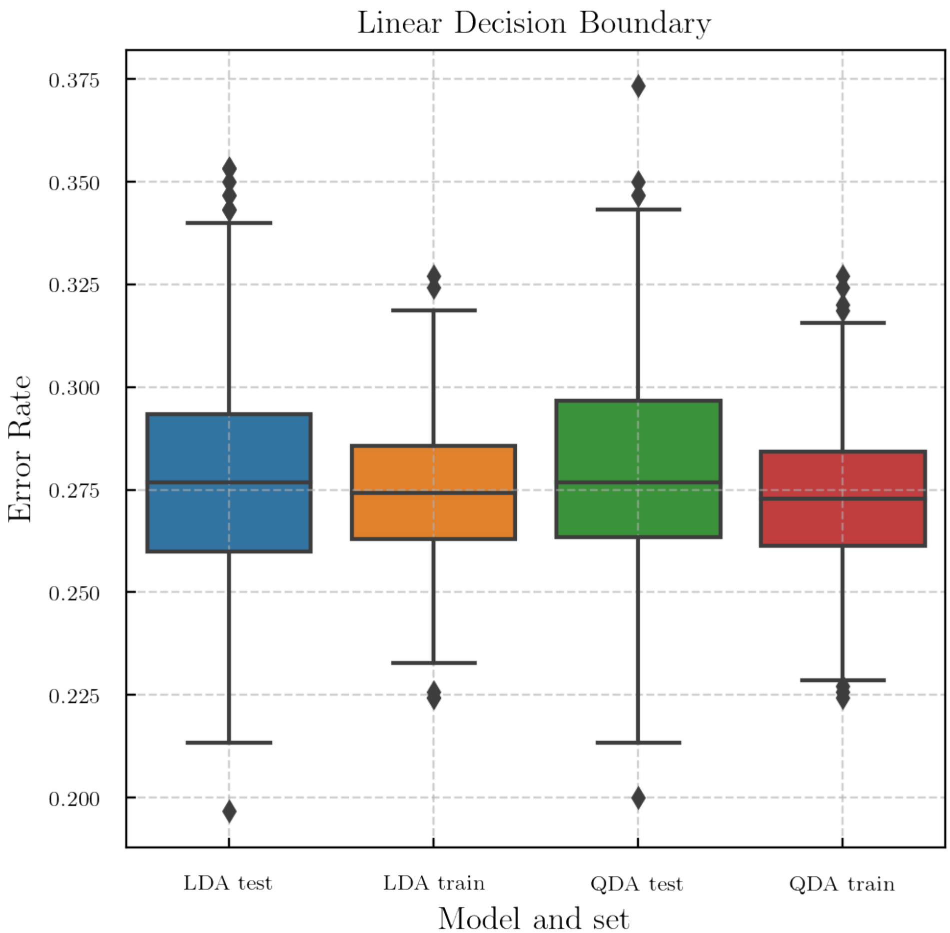
b) Summarizing the findings.

In graphical and tabular forms

[81]:
```
import pandas as pd

# Create dataframe
df_Err = pd.DataFrame({"LDA train" : Err_LDA_train,
                       "LDA test" : Err_LDA_test,
                       "QDA train" : Err_QDA_train,
                       "QDA test" : Err_QDA_test})
```

[85]:
```
sns.reset_orig()
plt.figure(figsize=(5.5,5.5))

sns.boxplot(df_Err)

plt.grid(ls='--', alpha=0.6)
plt.title("Linear Decision Boundary",fontsize=12)
plt.ylabel("Error Rate", fontsize=12);plt.xlabel("Model and set", fontsize=12)
plt.show()
```

```
//anaconda/envs/python2/lib/python2.7/site-packages/seaborn/categorical.py:2171:
UserWarning: The boxplot API has been changed. Attempting to adjust your
arguments for the new API (which might not work). Please update your code. See
the version 0.6 release notes for more info.
  warnings.warn(msg, UserWarning)
```



And we can also summarize in Tabular form

[87]:
```
df_Err.describe()
```

[87]:

	LDA test	LDA train	QDA test	QDA train
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.278027	0.273989	0.278450	0.272989
std	0.025378	0.016911	0.025286	0.017079
min	0.196667	0.224286	0.200000	0.224286
25%	0.260000	0.262857	0.263333	0.261429
50%	0.276667	0.274286	0.276667	0.272857
75%	0.293333	0.285714	0.296667	0.284286
max	0.353333	0.327143	0.373333	0.327143

Although slightly, we see that the mean (and with std) the error rate on the test set of the LDA model is smaller than that of the QDA model.

2.2 Non-Linear Bayes decision boundary

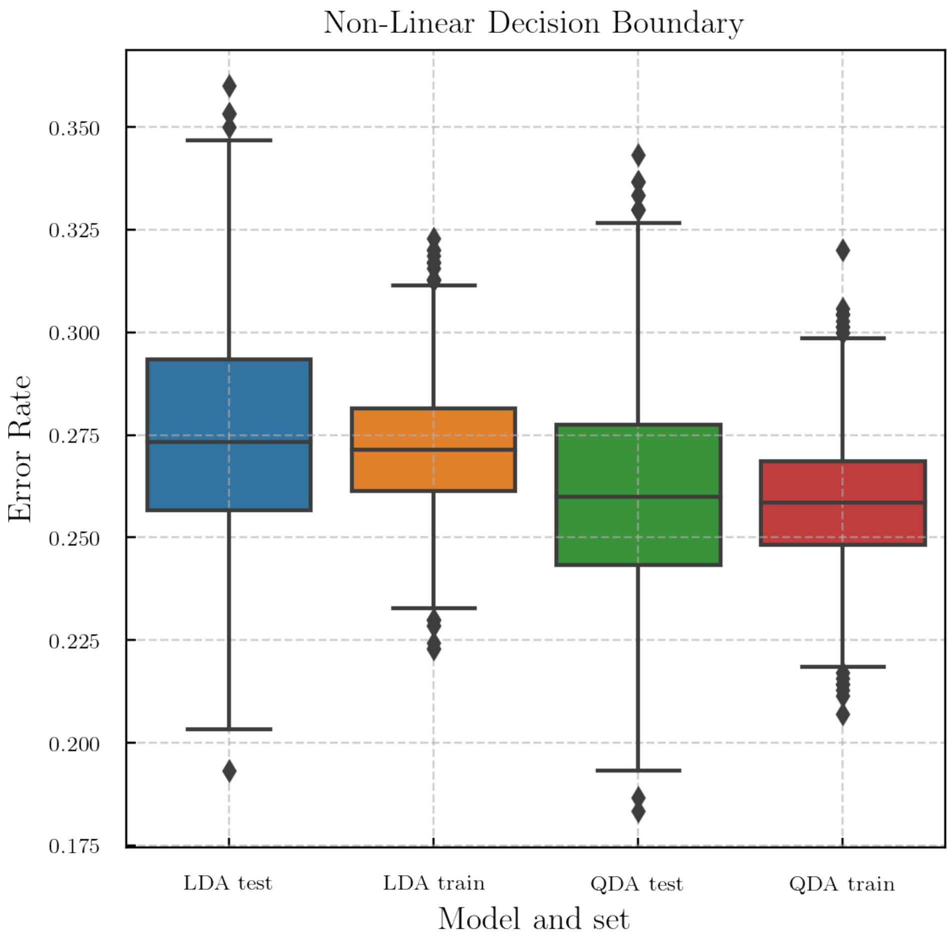
[88]:
```
N_rep = 1000 # Number of repetitions


# Create arrays to save the error rates of the two models
# Initialize them here
Err_LDA_train = np.zeros(N_rep) # LDA on train set
Err_LDA_test = np.zeros(N_rep) # LDA on test set
Err_QDA_train = np.zeros(N_rep) #QDA on train set
Err_QDA_test = np.zeros(N_rep) #QDA on test set


for i in range(N_rep):
    # Simulate X1, X2
    N_obs = 1000 # Number of observations
    X_1 = np.random.uniform(-1.0,+1.0,N_obs)
    X_2 = np.random.uniform(-1.0,+1.0,N_obs)
    # Simulate error
    err = np.random.normal(0,1.0,N_obs)

    # Define simulated Y
    Y = X_1 + X_1**2.0 + X_2 + X_2**2.0 + err

    # Now combine X_1, X_2 into a feature matrix
    X_ft = np.zeros([N_obs,2])
    X_ft[:,0] = X_1; X_ft[:,1] = X_2
    # Make Y binary
    Y[Y>0.0] = 1.0
    Y[Y<0.0] = - 1.0
    # ========================================
    # ========================================
    # Split into train/test
    X_train, X_test, y_train, y_test = train_test_split(X_ft, Y,
                                                        test_size=0.30,␣
↪random_state=42)

    # Fit LDA and QDA
    lda_model = LDA()
    lda_model.fit(X_train,y_train)
    qda_model = QDA()
    qda_model.fit(X_train,y_train)
    # =======================================
    # =======================================
    # Predictions now

    # Predict on the training set
```

```python
    y_pr_LDA_train = lda_model.predict(X_train)
    y_pr_QDA_train = qda_model.predict(X_train)

    # Predict on the test set
    y_pr_LDA_test = lda_model.predict(X_test)
    y_pr_QDA_test = qda_model.predict(X_test)
    # ========================================
    # ========================================
    # Calculate LDA and QDA training and testing error rates
    Err_LDA_train[i] = Error_Rate(y_train, y_pr_LDA_train)
    Err_LDA_test[i] = Error_Rate(y_test, y_pr_LDA_test)

    Err_QDA_train[i] = Error_Rate(y_train, y_pr_QDA_train)
    Err_QDA_test[i] = Error_Rate(y_test, y_pr_QDA_test)
```

b) Summarize the findings

[89]:
```python
# Create dataframe
df_Err = pd.DataFrame({"LDA train" : Err_LDA_train,
                      "LDA test" : Err_LDA_test,
                      "QDA train" : Err_QDA_train,
                      "QDA test" : Err_QDA_test})

plt.figure(figsize=(5.5,5.5))

sns.boxplot(df_Err)
plt.title("Non-Linear Decision Boundary",fontsize=12)
plt.grid(ls='--', alpha=0.6)
plt.ylabel("Error Rate", fontsize=12);plt.xlabel("Model and set", fontsize=12)
plt.show()
```

```
//anaconda/envs/python2/lib/python2.7/site-packages/seaborn/categorical.py:2171:
UserWarning: The boxplot API has been changed. Attempting to adjust your
arguments for the new API (which might not work). Please update your code. See
the version 0.6 release notes for more info.
  warnings.warn(msg, UserWarning)
```



[90]:
```
# And in tabular form
df_Err.describe()
```

[90]:

	LDA test	LDA train	QDA test	QDA train
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.275323	0.271704	0.261943	0.258353
std	0.026186	0.016743	0.026143	0.016259
min	0.193333	0.222857	0.183333	0.207143
25%	0.256667	0.261429	0.243333	0.248214
50%	0.273333	0.271429	0.260000	0.258571
75%	0.293333	0.281429	0.277500	0.268571
max	0.360000	0.322857	0.343333	0.320000

Both from visual inspection and from the table above we see that the QDA model performs much better than the LDA one, both on the training and test sets.

2.3 LDA/QDA Error Rate and Sample Size

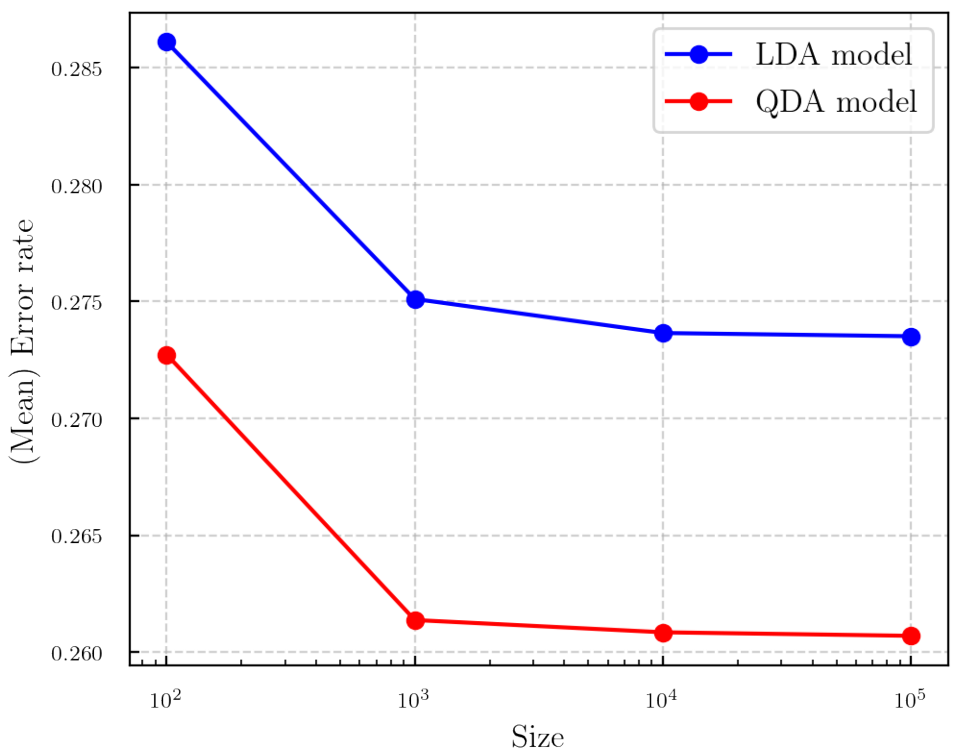
Here I will estimate the mean Error rate for the LDA and QDA models

[96]:
```
N_rep = 1000 # Number of repetitions

N_sizes = [1e2,1e3,1e4,1e5]
Errors_LDA = np.zeros(4)
Errors_QDA = np.zeros(4)

for j in range(4):
    N_obs = int(N_sizes[j])

    Err_LDA_test = np.zeros(N_rep) # LDA on test set
    Err_QDA_test = np.zeros(N_rep) #QDA on test set

    for i in range(N_rep):
        # Simulate X1, X2
        X_1 = np.random.uniform(-1.0,+1.0,N_obs)
        X_2 = np.random.uniform(-1.0,+1.0,N_obs)
        # Simulate error
        err = np.random.normal(0,1.0,N_obs)

        # Define simulated Y
        Y = X_1 + X_1**2.0 + X_2 + X_2**2.0 + err

        # Now combine X_1, X_2 into a feature matrix
        X_ft = np.zeros([N_obs,2])
        X_ft[:,0] = X_1; X_ft[:,1] = X_2
        # Make Y binary
        Y[Y>0.0] = 1.0
        Y[Y<0.0] = - 1.0
        # ========================================
        # ========================================
        # Split into train/test
        X_train, X_test, y_train, y_test = train_test_split(X_ft, Y,
                                                            test_size=0.30,␣
 ↪random_state=42)

        # Fit LDA and QDA
        lda_model = LDA()
        lda_model.fit(X_train,y_train)
        qda_model = QDA()
        qda_model.fit(X_train,y_train)

        # Predict on the test set
        y_pr_LDA_test = lda_model.predict(X_test)
```

```
        y_pr_QDA_test = qda_model.predict(X_test)

        Err_LDA_test[i] = Error_Rate(y_test, y_pr_LDA_test)
        Err_QDA_test[i] = Error_Rate(y_test, y_pr_QDA_test)

    Errors_LDA[j] = np.mean(Err_LDA_test)
    Errors_QDA[j] = np.mean(Err_QDA_test)
```

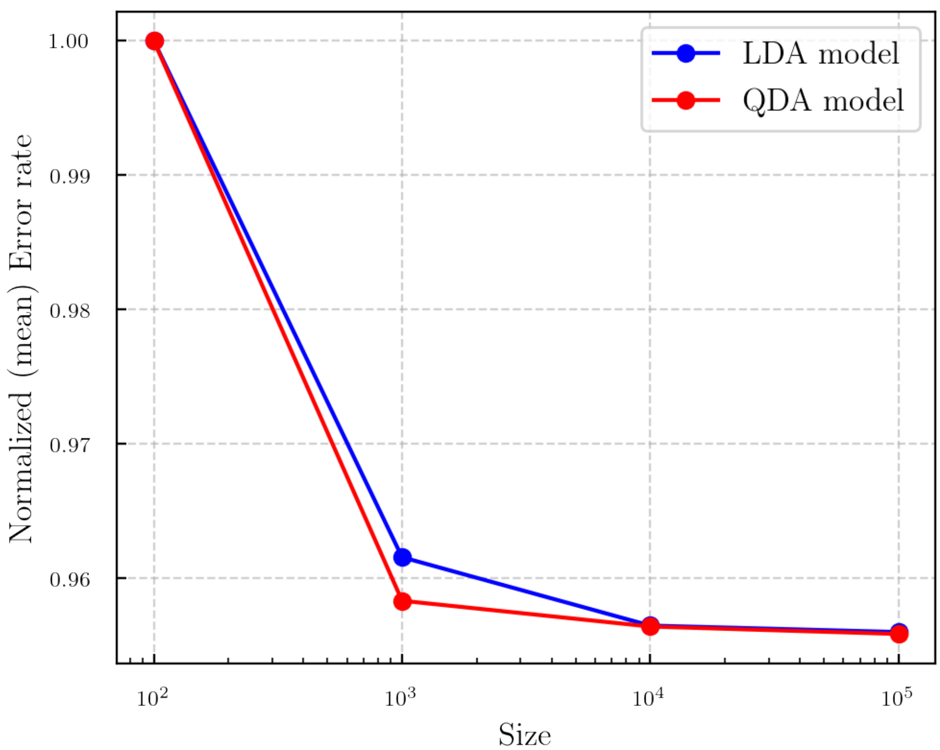
We can now plot Error rate as a function of size.

[114]:
```
plt.figure(figsize=(5.5,4.5))

plt.plot(N_sizes, Errors_LDA, marker='o',linewidth=1.5, c='blue', label='LDA␣
 ↪model')
plt.plot(N_sizes, Errors_QDA,marker='o', linewidth=1.5, c='red', label='QDA␣
 ↪model')

plt.grid(ls='--', alpha=0.6)
plt.xscale("log")
plt.xlabel("Size",fontsize=12);plt.ylabel("(Mean) Error rate", fontsize=12)
plt.legend(loc='upper right', fontsize=12)
plt.show()
```



However the above does not allow us to see easily which one changes more with the change of the size, because QDA always performs better. For that reason I will plot a normalized version of the above plot.

[115]:
```
plt.figure(figsize=(5.5,4.5))

plt.plot(N_sizes, Errors_LDA/Errors_LDA[0], marker='o',linewidth=1.5, c='blue',␣
↪label='LDA model')
plt.plot(N_sizes, Errors_QDA/Errors_QDA[0],marker='o', linewidth=1.5, c='red',␣
↪label='QDA model')

plt.grid(ls='--', alpha=0.6)
plt.xscale("log")
plt.xlabel("Size",fontsize=12);plt.ylabel("Normalized (mean) Error rate",␣
↪fontsize=12)
plt.legend(loc='upper right', fontsize=12)
plt.show()
```



The error rate of the QDA model drops more rapidly with size, but this trend does not continue for sizes $> 10^4$.

3 Modeling voter turnout

First let's import pandas to read csv and the other (that we haven's already imported) classification algorithms implementations from scikit-learn.

[116]:
```
import pandas as pd
from sklearn.linear_model import LogisticRegression as LogReg
from sklearn.neighbors import KNeighborsClassifier as KNN
```

Let's read the `mental_health.cvs` file first.

[117]:
```
ment_h_df = pd.read_csv('mental_health.csv')

# And print header
ment_h_df.head(10)
```

[117]:

	vote96	mhealth_sum	age	educ	black	female	married	inc10
0	1.0	0.0	60.0	12.0	0	0	0.0	4.8149
1	1.0	NaN	27.0	17.0	0	1	0.0	1.7387
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
4	0.0	NaN	35.0	16.0	0	1	0.0	4.8149
5	1.0	NaN	33.0	16.0	0	0	0.0	2.5412
6	0.0	NaN	43.0	12.0	0	0	0.0	4.8149
7	0.0	6.0	29.0	13.0	0	0	0.0	10.6998
8	1.0	2.0	39.0	18.0	0	1	1.0	NaN
9	0.0	NaN	45.0	15.0	0	0	0.0	7.2223

We see that there are many rows that contain `NaN` values. Let's drop them.

[118]:
```
mh_df = ment_h_df.dropna() # This is our final dataframe
mh_df.head(10) #Print first rows to see the difference
```

[118]:

	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
13	1.0	2.0	48.0	20.0	0	0	1.0	8.8273
14	0.0	9.0	20.0	12.0	0	1	0.0	7.2223
16	0.0	12.0	27.0	11.0	0	1	0.0	1.2037
19	1.0	2.0	28.0	16.0	0	0	1.0	7.2223
21	1.0	0.0	72.0	14.0	0	0	1.0	4.0124

Now extract the features (`mhealth_sum, age, educ, black,female,married,inc10`) and the responce variable (`vote96`) as numpy arrays to feed scikit-learn.

[119]:
```
 #Responce variable
vote96 = np.asarray(mh_df['vote96'])

# Featrues
mhealth_sum = np.asarray(mh_df['mhealth_sum'])
age = np.asarray(mh_df['age'])
educ = np.asarray(mh_df['educ'])
black = np.asarray(mh_df['black'])
female = np.asarray(mh_df['female'])
married = np.asarray(mh_df['married'])
inc10 = np.asarray(mh_df['inc10'])

N_len = len(age)
print(N_len)
```

```
1165
```

Construct a feature matrix now

[120]:
```
X_ft = np.column_stack((mhealth_sum,age,educ,black,female,married,inc10))
```

a) Now let's split the data into a training and test set.

[121]:
```
X_train, X_test, y_train, y_test = train_test_split(X_ft, vote96,
                                                     test_size=0.30,␣
 ↪random_state=42)
```

b) Fit the models now and Predict on the test set.

c) Logistic Regression

[124]:
```
clf = LogReg(random_state=0).fit(X_train, y_train)
# Predict
y_pr_lr = clf.predict(X_test)
```

ii) Linear Discriminant Model (LDA)

[125]:
```
lda_model = LDA()
lda_model.fit(X_train,y_train)
# Predict
y_pr_lda = lda_model.predict(X_test)
```

iii) Quadratic Discriminant Model (QDA)

[126]:
```
qda_model = QDA()
qda_model.fit(X_train,y_train)
# Predict
y_pr_qda = qda_model.predict(X_test)
```

iv) Naive Bayes

```
[127]: gnb = GaussianNB()
gnb.fit(X_train, y_train)
# Predict
y_pr_nb = gnb.predict(X_test)
```

v) KNN for $k = 1, ..., 10$ using Euclidean distance metrics.

```
[128]: # First Fit
kNN_1 = KNN(n_neighbors=1, metric='euclidean', weights='distance').
 ↪fit(X_train,y_train)
kNN_2 = KNN(n_neighbors=2, metric='euclidean', weights='distance').
 ↪fit(X_train,y_train)
kNN_3 = KNN(n_neighbors=3, metric='euclidean', weights='distance').
 ↪fit(X_train,y_train)
kNN_4 = KNN(n_neighbors=4, metric='euclidean', weights='distance').
 ↪fit(X_train,y_train)
kNN_5 = KNN(n_neighbors=5, metric='euclidean', weights='distance').
 ↪fit(X_train,y_train)
kNN_6 = KNN(n_neighbors=6, metric='euclidean', weights='distance').
 ↪fit(X_train,y_train)
kNN_7 = KNN(n_neighbors=7, metric='euclidean', weights='distance').
 ↪fit(X_train,y_train)
kNN_8 = KNN(n_neighbors=8, metric='euclidean', weights='distance').
 ↪fit(X_train,y_train)
kNN_9 = KNN(n_neighbors=9, metric='euclidean', weights='distance').
 ↪fit(X_train,y_train)
kNN_10 = KNN(n_neighbors=10, metric='euclidean', weights='distance').
 ↪fit(X_train,y_train)

# Then Predict
y_pr_kNN1 = kNN_1.predict(X_test)
y_pr_kNN2 = kNN_2.predict(X_test)
y_pr_kNN3 = kNN_3.predict(X_test)
y_pr_kNN4 = kNN_4.predict(X_test)
y_pr_kNN5 = kNN_5.predict(X_test)
y_pr_kNN6 = kNN_6.predict(X_test)
y_pr_kNN7 = kNN_7.predict(X_test)
y_pr_kNN8 = kNN_8.predict(X_test)
y_pr_kNN9 = kNN_9.predict(X_test)
y_pr_kNN10 = kNN_10.predict(X_test)
```

c) Model performance metrics.

Now we calculate the performance metrics.

i) Start with the error rate

[129]:
```
# Logistic regression
Err_lr = Error_Rate(y_test, y_pr_lr)
# LDA
Err_lda = Error_Rate(y_test, y_pr_lda)
# QDA
Err_qda = Error_Rate(y_test, y_pr_qda)
# Naive Bayes
Err_NB = Error_Rate(y_test, y_pr_nb)
# kNN - 10 models
Err_kNN1 = Error_Rate(y_test, y_pr_kNN1)
Err_kNN2 = Error_Rate(y_test, y_pr_kNN2)
Err_kNN3 = Error_Rate(y_test, y_pr_kNN3)
Err_kNN4 = Error_Rate(y_test, y_pr_kNN4)
Err_kNN5 = Error_Rate(y_test, y_pr_kNN5)
Err_kNN6 = Error_Rate(y_test, y_pr_kNN6)
Err_kNN7 = Error_Rate(y_test, y_pr_kNN7)
Err_kNN8 = Error_Rate(y_test, y_pr_kNN8)
Err_kNN9 = Error_Rate(y_test, y_pr_kNN9)
Err_kNN10 = Error_Rate(y_test, y_pr_kNN10)
```

ii) ROC curve and AUC

Here I will do the following, I will plot the ROC curves of all models except the ones of KNN in one plot and all the kNN curves in another plot.

For the ROC curve I have to predict probabilities for the positive class, so I will do that as well.

[130]:
```
from sklearn.metrics import roc_curve, auc

# Predict probabilites of the class numbered 1
y_proba_lr = clf.predict_proba(X_test)[:,1] #Linear Regression
y_proba_lda = lda_model.predict_proba(X_test)[:,1] #LDA
y_proba_qda = qda_model.predict_proba(X_test)[:,1] # QDA
y_proba_nb = gnb.predict_proba(X_test)[:,1]# Naive Bayes

# knns
y_proba_kNN1 = kNN_1.predict_proba(X_test)[:,1]
y_proba_kNN2 = kNN_2.predict_proba(X_test)[:,1]
y_proba_kNN3 = kNN_3.predict_proba(X_test)[:,1]
y_proba_kNN4 = kNN_4.predict_proba(X_test)[:,1]
y_proba_kNN5 = kNN_5.predict_proba(X_test)[:,1]
y_proba_kNN6 = kNN_6.predict_proba(X_test)[:,1]
y_proba_kNN7 = kNN_7.predict_proba(X_test)[:,1]
y_proba_kNN8 = kNN_8.predict_proba(X_test)[:,1]
y_proba_kNN9 = kNN_9.predict_proba(X_test)[:,1]
y_proba_kNN10 = kNN_10.predict_proba(X_test)[:,1]
```

Now estimate fale positive rate/ true postitive rate, necessary to plot the ROC curve and the area under each curve (AUC)

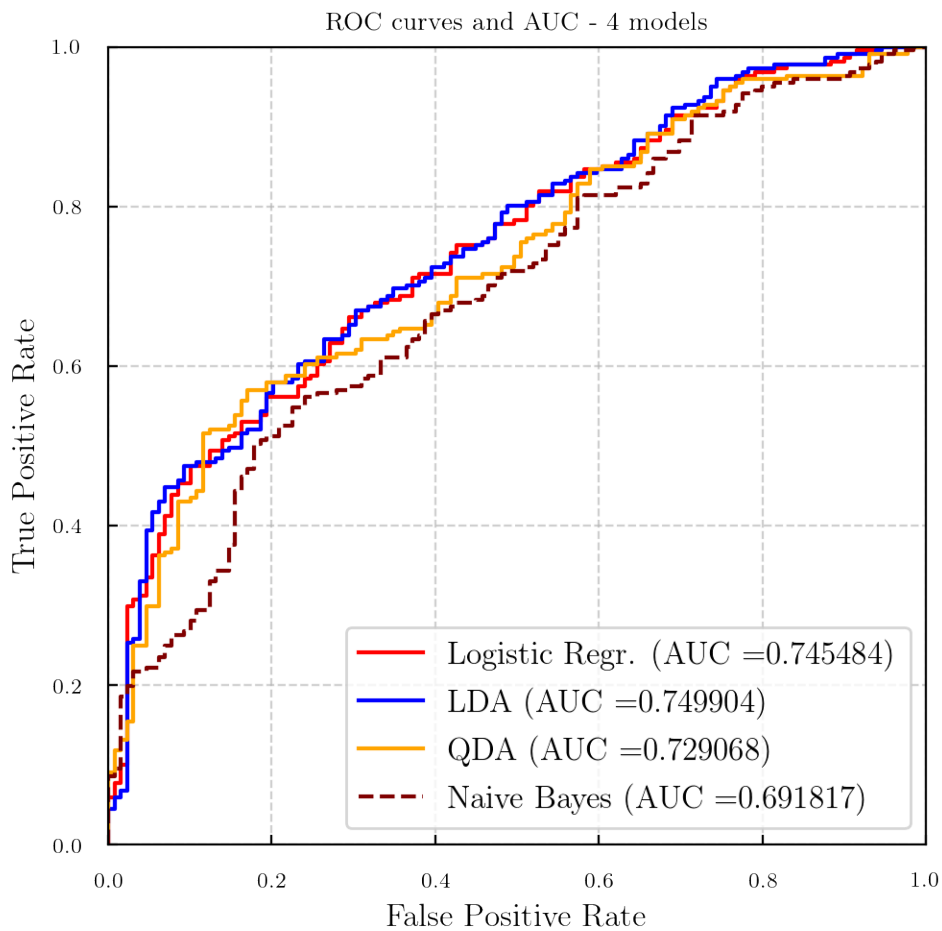
```
[131]: # fpr, tpr for each model, in order to plot the ROC curve
fpr_lr, tpr_lr, thr = roc_curve(y_test, y_proba_lr) #lr
fpr_lda, tpr_lda, thr = roc_curve(y_test, y_proba_lda)#LDA
fpr_qda, tpr_qda, thr = roc_curve(y_test, y_proba_qda)#QDA
fpr_nb, tpr_nb, thr = roc_curve(y_test, y_proba_nb)#Naive Bayes
# All kNNs
fpr_knn1, tpr_knn1, thr = roc_curve(y_test, y_proba_kNN1)
fpr_knn2, tpr_knn2, thr = roc_curve(y_test, y_proba_kNN2)
fpr_knn3, tpr_knn3, thr = roc_curve(y_test, y_proba_kNN3)
fpr_knn4, tpr_knn4, thr = roc_curve(y_test, y_proba_kNN4)
fpr_knn5, tpr_knn5, thr = roc_curve(y_test, y_proba_kNN5)
fpr_knn6, tpr_knn6, thr = roc_curve(y_test, y_proba_kNN6)
fpr_knn7, tpr_knn7, thr = roc_curve(y_test, y_proba_kNN7)
fpr_knn8, tpr_knn8, thr = roc_curve(y_test, y_proba_kNN8)
fpr_knn9, tpr_knn9, thr = roc_curve(y_test, y_proba_kNN9)
fpr_knn10, tpr_knn10, thr = roc_curve(y_test, y_proba_kNN10)
# =====================================================
# =====================================================
# Calcuate the AUCs
AUC_lr = auc(fpr_lr,tpr_lr) # Logistic Regression
AUC_lda = auc(fpr_lda,tpr_lda) # LDA
AUC_qda = auc(fpr_qda,tpr_qda) # QDA
AUC_nb = auc(fpr_nb,tpr_nb) # Naive Bayes

# And now for the kNN models
AUC_knn1 = auc(fpr_knn1,tpr_knn1)
AUC_knn2 = auc(fpr_knn2,tpr_knn2)
AUC_knn3 = auc(fpr_knn3,tpr_knn3)
AUC_knn4 = auc(fpr_knn4,tpr_knn4)
AUC_knn5 = auc(fpr_knn5,tpr_knn5)
AUC_knn6 = auc(fpr_knn6,tpr_knn6)
AUC_knn7 = auc(fpr_knn7,tpr_knn7)
AUC_knn8 = auc(fpr_knn8,tpr_knn8)
AUC_knn9 = auc(fpr_knn9,tpr_knn9)
AUC_knn10 = auc(fpr_knn10,tpr_knn10)
```

Now make some nice plots

First the ROC curves of all models except the kNN ones, and then the kNN models

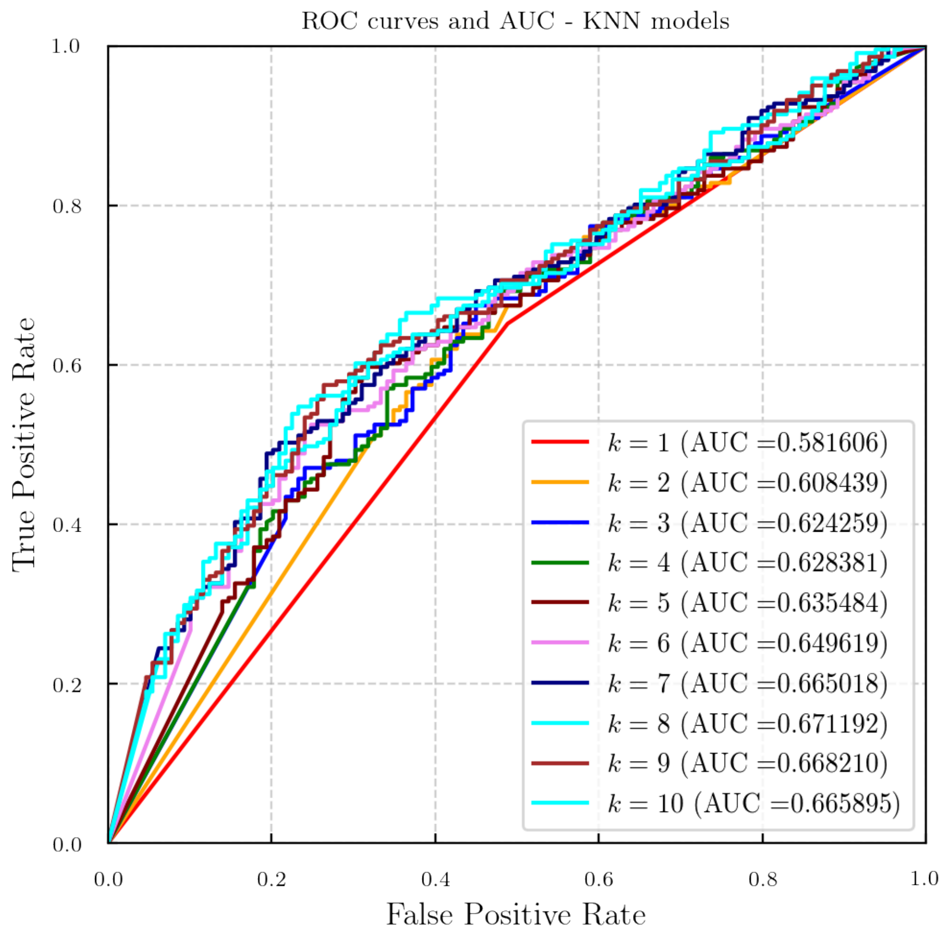
```
[151]: plt.figure(figsize=(5.5,5.5))

plt.plot(fpr_lr, tpr_lr, c='red', label='Logistic Regr. (AUC ={0:3f})'.
 ↪format(AUC_lr))
plt.plot(fpr_lda, tpr_lda,  c='blue', label='LDA (AUC ={0:3f})'.format(AUC_lda))
plt.plot(fpr_qda, tpr_qda, c='orange',label='QDA (AUC ={0:3f})'.format(AUC_qda)␣
 ↪)
```

```
plt.plot(fpr_nb, tpr_nb, c='maroon',ls='--',label='Naive Bayes (AUC ={0:3f})'.
 ↪format(AUC_nb))

plt.title('ROC curves and AUC - 4 models')
plt.grid(ls='--', alpha=0.6)
plt.xlabel('False Positive Rate',fontsize=12);plt.ylabel('True Positive␣
 ↪Rate',fontsize=12)
plt.xlim(0.0,1.0);plt.ylim(0.0,1.0)

plt.legend(loc='lower right', fontsize=12)
plt.show()
```



[156]:
```
plt.figure(figsize=(5.5,5.5))

plt.plot(fpr_knn1, tpr_knn1,c='red', label='$k=1$ (AUC ={0:3f})'.
 ↪format(AUC_knn1))
```

```
plt.plot(fpr_knn2, tpr_knn2,c='orange', label='$k=2$ (AUC ={0:3f})'.
 ↪format(AUC_knn2))
plt.plot(fpr_knn3, tpr_knn3,c='blue', label='$k=3$ (AUC ={0:3f})'.
 ↪format(AUC_knn3))
plt.plot(fpr_knn4, tpr_knn4,c='green', label='$k=4$ (AUC ={0:3f})'.
 ↪format(AUC_knn4))
plt.plot(fpr_knn5, tpr_knn5,c='maroon', label='$k=5$ (AUC ={0:3f})'.
 ↪format(AUC_knn5))
plt.plot(fpr_knn6, tpr_knn6,c='violet', label='$k=6$ (AUC ={0:3f})'.
 ↪format(AUC_knn6))
plt.plot(fpr_knn7, tpr_knn7,c='navy', label='$k=7$ (AUC ={0:3f})'.
 ↪format(AUC_knn7))
plt.plot(fpr_knn8, tpr_knn8,c='aqua', label='$k=8$ (AUC ={0:3f})'.
 ↪format(AUC_knn8))
plt.plot(fpr_knn9, tpr_knn9,c='brown', label='$k=9$ (AUC ={0:3f})'.
 ↪format(AUC_knn9))
plt.plot(fpr_knn10, tpr_knn10,c='cyan', label='$k=10$ (AUC ={0:3f})'.
 ↪format(AUC_knn10))

plt.title('ROC curves and AUC - KNN models')
plt.grid(ls='--', alpha=0.6)
plt.xlabel('False Positive Rate',fontsize=12);plt.ylabel('True Positive␣
 ↪Rate',fontsize=12)
plt.xlim(0.0,1.0);plt.ylim(0.0,1.0)

plt.legend(loc='lower right', fontsize=10)
plt.show()
```



The maximum AUC for all above models (knn and the rest) comes from the LDA model (AUC = 0.7449) followed closely by the one from the Logistic Regression model (AUC = 0.7455).

It seems that the LDA model performs the best. Let's check the error rates as well

[160]:
```
print(Err_lr)
print(Err_lda)
print(Err_qda)
print(Err_NB)
```

```
0.31714285714285717
0.31999999999999995
0.3342857142857143
0.34571428571428575
```

[163]:
```
print(Err_kNN1)
print(Err_kNN10)
```

```
0.4
```

```
0.37428571428571433
```

The minimum error rate now comes from the Logistic regression model, closely followed by the LDA model. It seems that these two models are almost equivalent.