# Homework 2: Classification Method

YIMIN LI

1. The Bayes Classifier

In [164]:
```
import sys
!{sys.executable} -m pip install tabulate
```

```
Requirement already satisfied: tabulate in c:\users\qmun\anaconda3\lib\s
ite-packages (0.8.6)
```

In [165]:
```
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.naive_bayes import GaussianNB as GNB
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as
LDA
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
as QDA
from sklearn.linear_model import LogisticRegression as LR
from sklearn.neighbors import KNeighborsClassifier as KNN
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import train_test_split
from tabulate import tabulate
from statistics import mean
```

In [166]:
```
# Generate Seed
np.random.seed(2020)
```

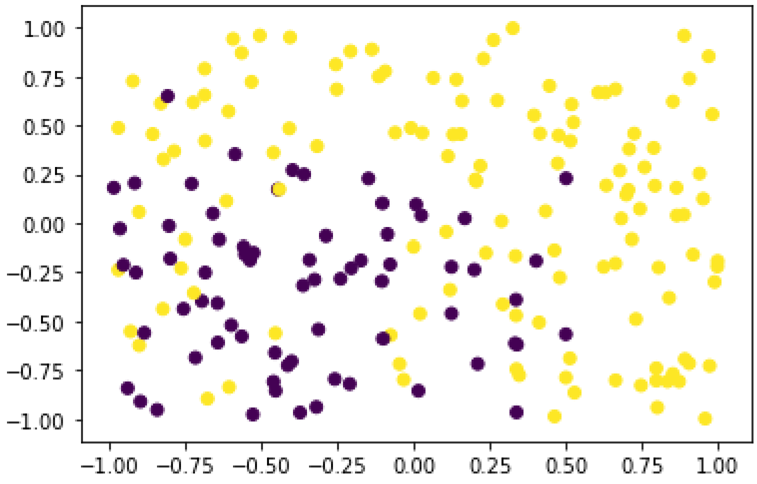
In [167]:
```
# Generate Uniform Variables
x1 = np.random.uniform(-1, 1, 200)
x2 = np.random.uniform(-1, 1, 200)
```

In [168]:
```
# Calculate Y
epislon = np.random.normal(0, 0.5, 200)
y = x1 + x1 * x1 + x2 + x2 * x2 + epislon
```

In [84]:
```
# According to the definition, y = log(prob / (1 - prob))
#Then, solving this formula, we would get
prob = np.exp(y) / (np.exp(y) + 1)
```

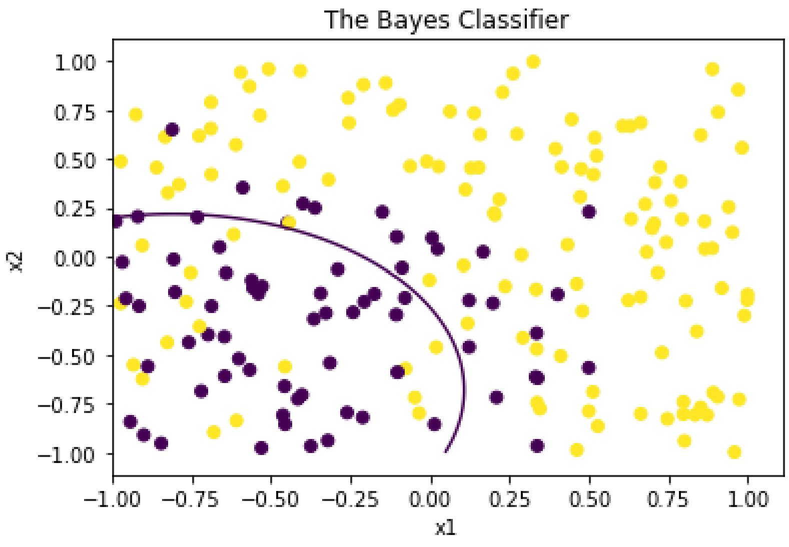
In [169]:
```
# Draw the draft
success = np.where(prob > 0.5, True, False)
plt.scatter(x1, x2, c = success)
```

Out[169]:
```
<matplotlib.collections.PathCollection at 0x21ccad5e908>
```



In [170]:
```
# Process the boundary by using GNB
X = np.stack(((x1),(x2)), axis=1)
gnb = GNB()
gnb.fit(X, success)
xx, yy = np.meshgrid(np.linspace(-1, 1, 100), np.linspace(-1, 1, 100))
Z = gnb.predict_proba(np.c_[xx.ravel(), yy.ravel()])
Z = Z[:, 1].reshape(xx.shape)
```

In [172]:
```
# Label the graph and draw the contour
success = np.where(prob > 0.5, True, False)
plt.scatter(x1, x2, c = success)
plt.contour(xx, yy, Z, [0.5])
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('The Bayes Classifier')
plt.show()
```



Exploring Simulated Differences between LDA and QDA

1. If Bayes decision boundary is linear, QDA would perform better on the training set since it would fit closer due to its higher flexibility. On the test set, LDA would perform better because QDA might overfit.

In [173]:
```
# i. Simulate 1000 times
def simulation():
    lda_train_error = []
    lda_test_error = []
    qda_train_error = []
    qda_test_error = []
    for i in range(1000):
        x1 = np.random.uniform(-1, 1, 1000)
        x2 = np.random.uniform(-1, 1, 1000)
        epislon = np.random.normal(0, 1, 1000)
        y = x1 + x2 + epislon
        y_simulated = y >= 0
        x = np.column_stack((x1, x2))
     # Split the dataset
        x_train, x_test, y_train, y_test = train_test_split(x, y_simulat
ed, test_size = 0.3)
     # Training dataset
        lda = LDA()
        lda.fit(x_train, y_train)
        qda = QDA()
        qda.fit(x_train, y_train)
     # Calculate the error
        lda_train = 1 - lda.score(x_train, y_train)
        lda_train_error.append(lda_train)
        lda_test = 1 - lda.score(x_test, y_test)
        lda_test_error.append(lda_test)
        qda_train = 1 - qda.score(x_train, y_train)
        qda_train_error.append(qda_train)
        qda_test = 1 - qda.score(x_test, y_test)
        qda_test_error.append(qda_test)
    return (lda_train_error, lda_test_error, qda_train_error, qda_test_e
rror)
```

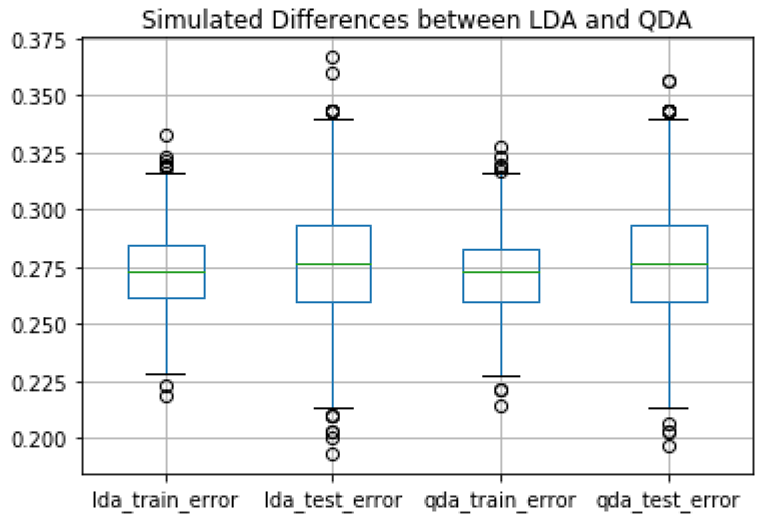
In [174]:

```
error_ls = simulation()
lda_train_error = error_ls[0]
lda_test_error = error_ls[1]
qda_train_error = error_ls[2]
qda_test_error = error_ls[3]
# Print out Tabulate
print(tabulate([['LDA Train', mean(lda_train_error)], ['LDA Test', mean(
lda_test_error)],
               ['QDA Train', mean(qda_train_error)], ['QDA Test', mean(q
da_test_error)]],
               headers = ['Dataset', 'Error Rate']))
# Draw Boxplot
df = pd.DataFrame(error_ls).T
df.rename(columns = {0: 'lda_train_error', 1: 'lda_test_error',
                     2: 'qda_train_error', 3: 'qda_test_error'}, inplace
=True)
df.boxplot()
plt.title("Simulated Differences between LDA and QDA")
```

```
Dataset      Error Rate
---------  ------------
LDA Train      0.272807
LDA Test       0.276613
QDA Train      0.271891
QDA Test       0.27677
```

Out[174]:

```
Text(0.5, 1.0, 'Simulated Differences between LDA and QDA')
```



As is shown in the boxplot and table, LDA's train error is slightly larger than QDA while QDA's test error is slightly larger than LDA's. It shows that QDA performs better in training set and LDA performs better in test set, which proves our prior explanation from an empirical perspective. However, it is also worth noting that the differences of error in test and training set for LDA and QDA are very small.

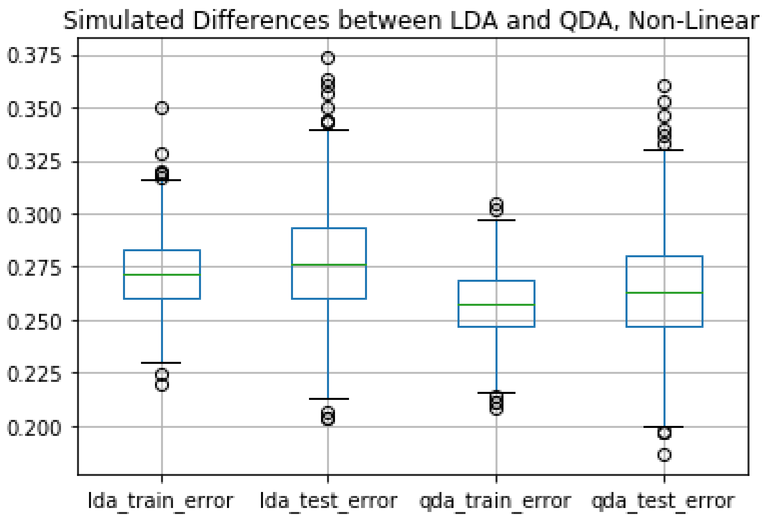
1. If Bayes decision boundary is non-linear, we would expect QDA to perform better both in training and test set due to its flexibility.

In [175]:
```
# i. Simulate 1000 times
def simulation_non_linear():
    lda_train_error_non = []
    lda_test_error_non = []
    qda_train_error_non = []
    qda_test_error_non = []
    for i in range(1000):
        x1 = np.random.uniform(-1, 1, 1000)
        x2 = np.random.uniform(-1, 1, 1000)
        epislon = np.random.normal(0, 1, 1000)
        y = x1 + x1 * x1 + x2 + x2 * x2 + epislon
        y_simulated = y >= 0
        x = np.column_stack((x1, x2))
     # Split the dataset
        x_train, x_test, y_train, y_test = train_test_split(x, y_simulat
ed, test_size = 0.3)
     # Training dataset
        lda = LDA()
        lda.fit(x_train, y_train)
        qda = QDA()
        qda.fit(x_train, y_train)
     # Calculate the error
        lda_train_non = 1 - lda.score(x_train, y_train)
        lda_train_error_non.append(lda_train_non)
        lda_test_non = 1 - lda.score(x_test, y_test)
        lda_test_error_non.append(lda_test_non)
        qda_train_non = 1 - qda.score(x_train, y_train)
        qda_train_error_non.append(qda_train_non)
        qda_test_non = 1 - qda.score(x_test, y_test)
        qda_test_error_non.append(qda_test_non)
    return (lda_train_error_non, lda_test_error_non, qda_train_error_non
, qda_test_error_non)
```

In [176]:
```
error_ls_non = simulation_non_linear()
lda_train_error_non = error_ls_non[0]
lda_test_error_non = error_ls_non[1]
qda_train_error_non = error_ls_non[2]
qda_test_error_non = error_ls_non[3]
# Print out Tabulate
print(tabulate([['LDA Train', mean(lda_train_error_non)], ['LDA Test', m
ean(lda_test_error_non)],
               ['QDA Train', mean(qda_train_error_non)], ['QDA Test', me
an(qda_test_error_non)]],
               headers = ['Dataset', 'Error Rate']))
# Draw Boxplot
df = pd.DataFrame(error_ls_non).T
df.rename(columns = {0: 'lda_train_error', 1: 'lda_test_error',
                     2: 'qda_train_error', 3: 'qda_test_error'}, inplace
=True)
df.boxplot()
plt.title("Simulated Differences between LDA and QDA, Non-Linear")
```

```
Dataset      Error Rate
---------  ------------
LDA Train      0.271717
LDA Test       0.27593
QDA Train      0.257947
QDA Test       0.262583
```

Out[176]:
```
Text(0.5, 1.0, 'Simulated Differences between LDA and QDA, Non-Linear')
```



As is shown in the boxplot and table, LDA's both train and test errors are larger than qda. It shows that QDA performs better in both training and test set, which proves our prior explanation from an empirical perspective. it is also worth noting that the differences of error between LDA and QDA are much more significant than the linear-one.

1. Generally speaking, since QDA is more flexible than LDA and has higher variance, QDA performs better when sample size n increases, so that the test error rate of QDA relative to LDA would decrease as sample size n increases and the variance of the classifier is not a major concern.

In [177]:
```
 #i. Simulate multiple times
def simulation_multiple(n):
    lda_train_multi = []
    lda_test_multi = []
    qda_train_multi = []
    qda_test_multi = []
    for i in range(1000):
        x1 = np.random.uniform(-1, 1, n)
        x2 = np.random.uniform(-1, 1, n)
        epislon = np.random.normal(0, 1, n)
        y = x1 + x2 + epislon
        y_simulated = y >= 0
        x = np.column_stack((x1, x2))
     # Split the dataset
        x_train, x_test, y_train, y_test = train_test_split(x, y_simulat
ed, test_size = 0.3)
     # Training dataset
        lda = LDA()
        lda.fit(x_train, y_train)
        qda = QDA()
        qda.fit(x_train, y_train)
     # Calculate the error
        lda_train = 1 - lda.score(x_train, y_train)
        lda_train_multi.append(lda_train)
        lda_test = 1 - lda.score(x_test, y_test)
        lda_test_multi.append(lda_test)
        qda_train = 1 - qda.score(x_train, y_train)
        qda_train_multi.append(qda_train)
        qda_test = 1 - qda.score(x_test, y_test)
        qda_test_multi.append(qda_test)
    return (lda_train_multi, lda_test_multi, qda_train_multi, qda_test_m
ulti)
```

In [178]:

```
error_ls_100 = simulation_multiple(100)
error_ls_1000 = simulation_multiple(1000)
error_ls_10000 = simulation_multiple(10000)
error_ls_100000 = simulation_multiple(100000)

df_100 = pd.DataFrame(error_ls_100).T
df_100.rename(columns = {0: 'lda_train_100', 1: 'lda_test_100',
                         2: 'qda_train_100', 3: 'qda_test_100'}, inplace
=True)
df_1000 = pd.DataFrame(error_ls_1000).T
df_1000.rename(columns = {0: 'lda_train_1000', 1: 'lda_test_1000',
                         2: 'qda_train_1000', 3: 'qda_test_1000'}, inpla
ce=True)
df_10000 = pd.DataFrame(error_ls_10000).T
df_10000.rename(columns = {0: 'lda_train_10000', 1: 'lda_test_10000',
                         2: 'qda_train_10000', 3: 'qda_test_10000'}, inp
lace=True)
df_100000 = pd.DataFrame(error_ls_100000).T
df_100000.rename(columns = {0: 'lda_train_100000', 1: 'lda_test_100000',

                         2: 'qda_train_100000', 3: 'qda_test_100000'}, i
nplace=True)

df = pd.concat([df_100, df_1000, df_10000, df_100000], axis=1)
df.head(10)
```

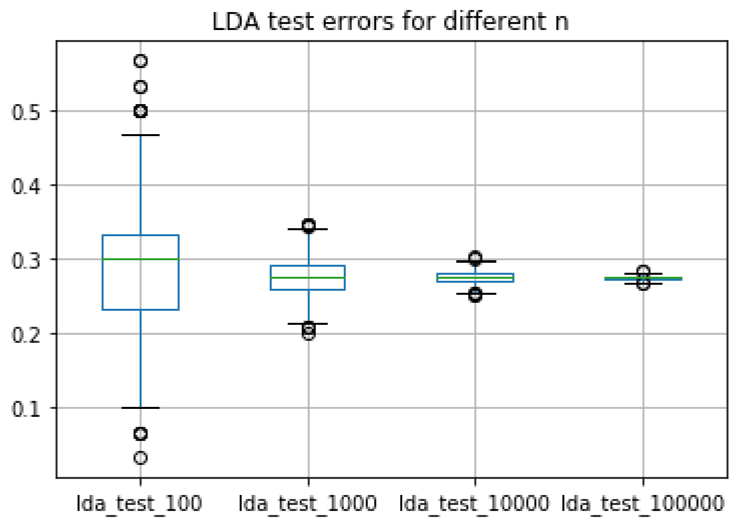
Out[178]:

| | lda_train_100 | lda_test_100 | qda_train_100 | qda_test_100 | lda_train_1000 | lda_test_1000 | qda_trai |
|---|---|---|---|---|---|---|---|
| 0 | 0.228571 | 0.533333 | 0.228571 | 0.500000 | 0.250000 | 0.283333 | 0. |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0.228571 | 0.233333 | 0.228571 | 0.200000 | 0.281429 | 0.243333 | 0. |
| 2 | 0.300000 | 0.100000 | 0.242857 | 0.100000 | 0.290000 | 0.300000 | 0. |
| 3 | 0.271429 | 0.300000 | 0.257143 | 0.333333 | 0.282857 | 0.303333 | 0. |
| 4 | 0.271429 | 0.366667 | 0.271429 | 0.333333 | 0.291429 | 0.266667 | 0. |
| 5 | 0.271429 | 0.266667 | 0.257143 | 0.233333 | 0.272857 | 0.276667 | 0. |
| 6 | 0.228571 | 0.366667 | 0.214286 | 0.366667 | 0.282857 | 0.310000 | 0. |
| 7 | 0.171429 | 0.233333 | 0.171429 | 0.233333 | 0.247143 | 0.260000 | 0. |
| 8 | 0.271429 | 0.200000 | 0.242857 | 0.266667 | 0.258571 | 0.300000 | 0. |
| 9 | 0.200000 | 0.266667 | 0.185714 | 0.300000 | 0.275714 | 0.283333 | 0. |

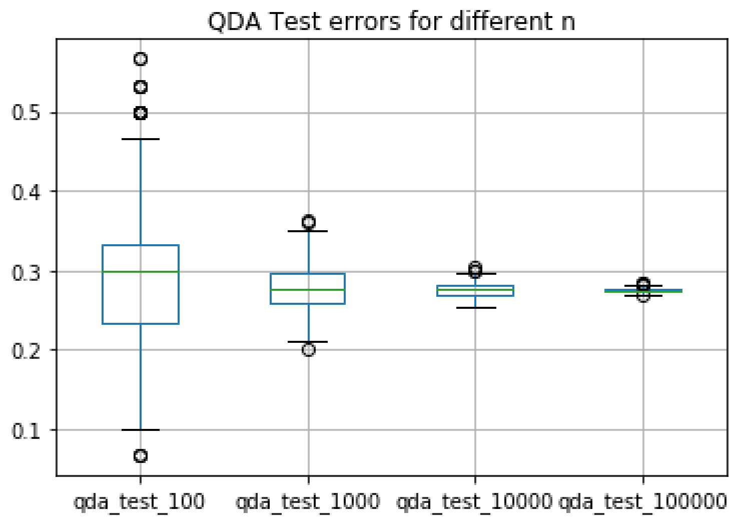
In [179]:
```
df[['lda_test_100', 'lda_test_1000', 'lda_test_10000', 'lda_test_100000'
]].boxplot()
plt.title('LDA test errors for different n')
```

Out[179]:
```
Text(0.5, 1.0, 'LDA test errors for different n')
```



In [180]:
```
df[['qda_test_100', 'qda_test_1000', 'qda_test_10000', 'qda_test_100000'
]].boxplot()
plt.title('QDA Test errors for different n')
```

Out[180]:
```
Text(0.5, 1.0, 'QDA Test errors for different n')
```



From the tables above, we would apparently see that both QDA and LDA test error changes as sample size n grows. However, QDA's test error reduces more dramatically than LDA. Thus,the test error rate of QDA relative to LDA would decrease as sample size n grows.

1. Modeling Voter Turnout

In [186]:
```
url = "https://raw.githubusercontent.com/macss-model20/problem-set-2/mas
ter/mental_health.csv"
df = pd.read_csv(url)
df.dropna(inplace=True)
y = df['vote96']
x = df[df.columns[1:]]
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.
3)
```

In [187]:
```
# Logistic Regression Model
lr = LR()
lr.fit(x_train, y_train)
lr_error = 1 - lr.score(x_test, y_test)
# Linear Discriminant Model
lda = LDA()
lda.fit(x_train, y_train)
lda_error = 1 - lda.score(x_test, y_test)
# Quadratic Discriminant Model
qda = QDA()
qda.fit(x_train, y_train)
qda_error = 1 - qda.score(x_test, y_test)
# Naive Bayes
nb = GNB()
nb.fit(x_train, y_train)
nb_error = 1 - nb.score(x_test, y_test)
# K-nearest neighbors and Euclidean distance metrics
knn_list = []
knn_list_err = []
for i in range(1, 11):
    knn = KNN(n_neighbors = i)
    knn.fit(x_train, y_train)
    knn_error = 1 - knn.score(x_test, y_test)
    knn_list.append(knn)
    knn_list_err.append(knn_error)
```

```
C:\Users\qmun\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.
py:432: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22
. Specify a solver to silence this warning.
  FutureWarning)
```

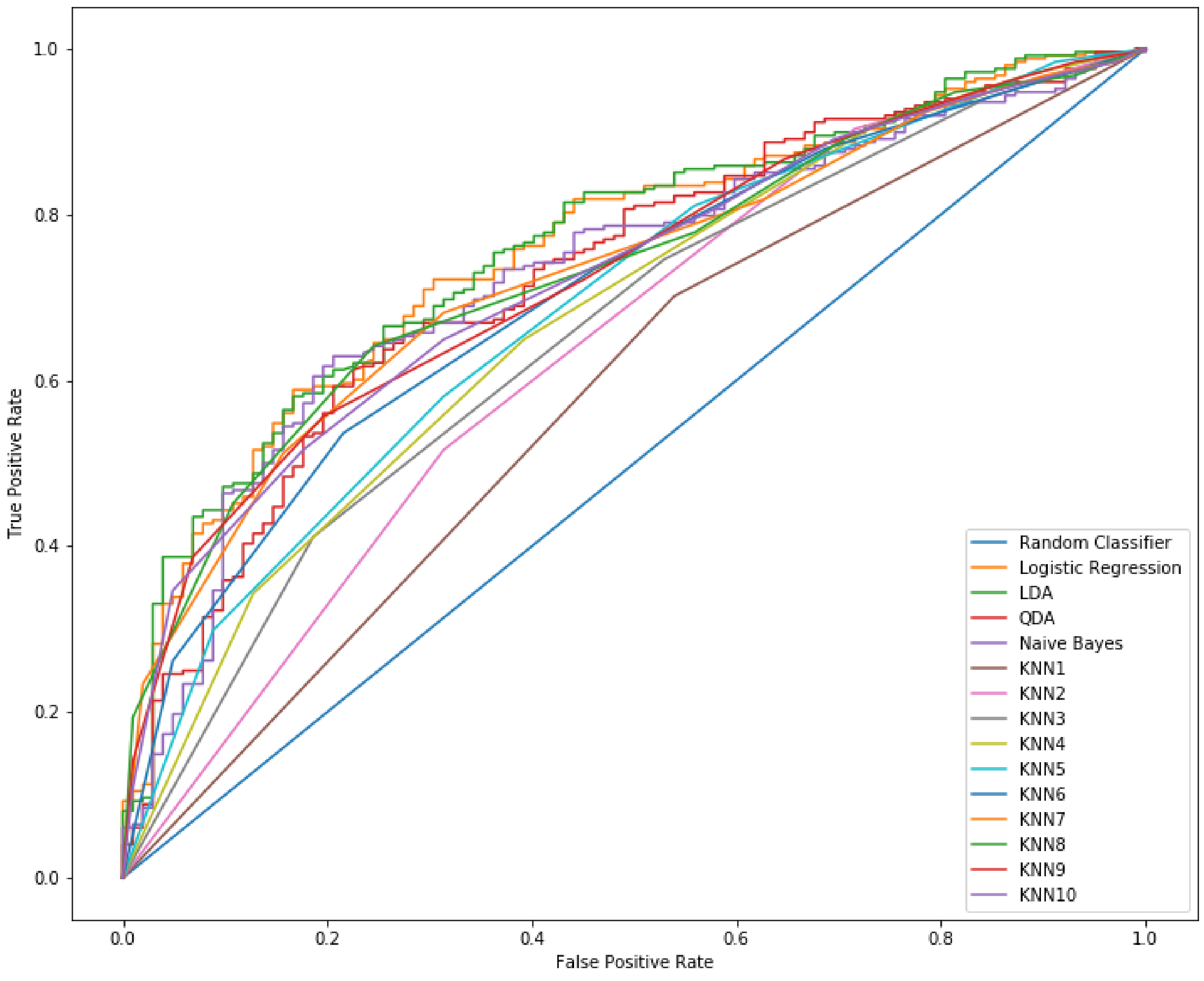
In [188]:
```
print(tabulate([['Logistic Regression', lr_error],
                ['Linear Discriminant', lda_error],
                ['Quadratic Discriminant', qda_error],
                ['Naive Bayes', nb_error],
                ['KNN, K=1', knn_list_err[0]],
                ['KNN, K=2', knn_list_err[1]],
                ['KNN, K=3', knn_list_err[2]],
                ['KNN, K=4', knn_list_err[3]],
                ['KNN, K=5', knn_list_err[4]],
                ['KNN, K=6', knn_list_err[5]],
                ['KNN, K=7', knn_list_err[6]],
                ['KNN, K=8', knn_list_err[7]],
                ['KNN, K=9', knn_list_err[8]],
                ['KNN, K=10', knn_list_err[9]]],
               headers = ['Type', 'Error rate']))
```

```
Type                      Error rate
----------------------  ------------
Logistic Regression         0.277143
Linear Discriminant         0.265714
Quadratic Discriminant      0.302857
Naive Bayes                 0.285714
KNN, K=1                    0.368571
KNN, K=2                    0.434286
KNN, K=3                    0.334286
KNN, K=4                    0.362857
KNN, K=5                    0.297143
KNN, K=6                    0.322857
KNN, K=7                    0.311429
KNN, K=8                    0.32
KNN, K=9                    0.282857
KNN, K=10                   0.305714
```

In [189]:
```
def roc_auc(model, name):
    probs = model.predict_proba(x_test)[:,1]
    model_auc = roc_auc_score(y_test, probs)
    print(name + ' : ' + str(model_auc))
    fpr, tpr, _ = roc_curve(y_test, probs)
    plt.plot(fpr, tpr, label=name)

plt.figure(figsize=(12,10))
random_probs = [0] * len(y_test)
random_auc = roc_auc_score(y_test, random_probs)
random_fpr, random_tpr, _ = roc_curve(y_test, random_probs)
plt.plot(random_fpr, random_tpr, label='Random Classifier')

models = [lr, lda, qda, nb] + knn_list
names = ['Logistic Regression', 'LDA', 'QDA', 'Naive Bayes', 'KNN1', 'KN
N2',
         'KNN3', 'KNN4', 'KNN5', 'KNN6', 'KNN7', 'KNN8', 'KNN9', 'KNN10'
]
for i in range(len(models)):
    roc_auc(models[i], names[i])

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
```

```
Logistic Regression : 0.7581633459835547
LDA : 0.7625513915243517
QDA : 0.7281981340923467
Naive Bayes : 0.7280400063251107
KNN1 : 0.5811986084756484
KNN2 : 0.6367805186590765
KNN3 : 0.6533641682479443
KNN4 : 0.6737824161922834
KNN5 : 0.6860373181530677
KNN6 : 0.7052103099304237
KNN7 : 0.7263994307400379
KNN8 : 0.7323292220113852
KNN9 : 0.7269133459835547
KNN10 : 0.7243437697659708
```

```
Out[189]: <matplotlib.legend.Legend at 0x21ccb3b3588>
```



In terms of test error, LDA has the highest accuracy while KNN model (except for K=9) performs worst in test error. In terms of ROC, LDA has the highest AUC among others, and Logisitic Regression ranks the second. So whether "best" refers to test error or ROC, LDA both performs the best and Logistic Regression Model ranks the 2nd in both perspectives.