In [73]:

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.discriminant_analysis import LinearDiscriminantAnal
ysis
from sklearn.discriminant_analysis import QuadraticDiscriminantA
nalysis
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, roc_curve, auc
from sklearn.model_selection import train_test_split
from math import exp
```

1 The Bayes Classifier

In [20]:

```
#a. Set random number generator seed
np.random.seed(2019)
```

In [21]:

```
#b. Simulate a dataset of N = 200 with X1, X2
x1 = np.random.uniform(-1, 1, 200)
x2 = np.random.uniform(-1, 1, 200)
```

In [22]:

```
#c. CalculateY =X1+X12+X2+X2+ε,whereε~N(μ=0,σ2=0.25).
y = x1 + x1**2 + x2 + x2**2 + np.random.normal(0, 0.5, 200)
```

In [23]:

```
#d. Y is defined in terms of the log-odds of success on the doma
in [−∞, +∞]. Calculate the probability of success bounded betwee
n [0, 1].
def cal_prob(x):
    p = np.exp(np.array(x))/(1 + np.exp(np.array(x)))
    return p

p = cal_prob(Y)
p_tf=p>0.5
p
```

Out[23]:

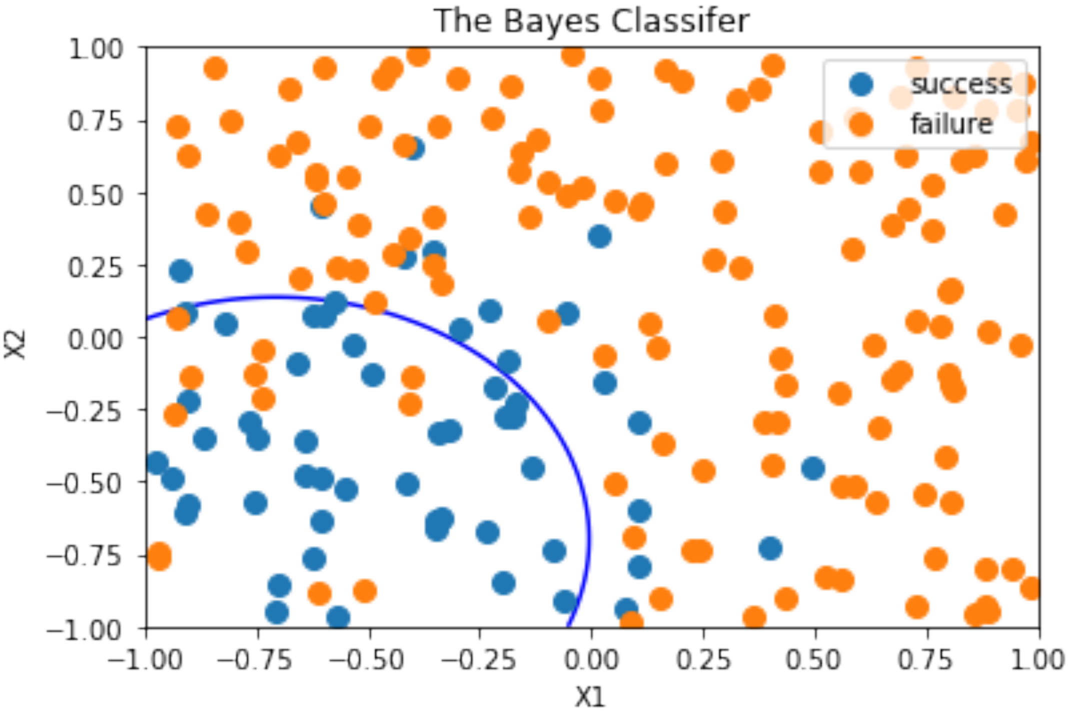
```
array([0.82849187, 0.4608178 , 0.51176518, 0.6051898
2, 0.90938358,
       0.59006533, 0.96321944, 0.75761982, 0.9496808
8, 0.48865303,
       0.52303872, 0.90609121, 0.74523342, 0.6402173
3, 0.79025966,
       0.96359111, 0.53661513, 0.47997726, 0.3211723
3, 0.56016197,
       0.5738233 , 0.56393526, 0.47938579, 0.5309953
, 0.59683373,
       0.54477001, 0.57012001, 0.44425388, 0.8319347
4, 0.37961348,
       0.69761183, 0.85921594, 0.6296643 , 0.9486115
1, 0.84212708,
       0.71464094, 0.6528656 , 0.63641372, 0.4685504
2, 0.72024099,
       0.83368541, 0.2684377 , 0.50721412, 0.8443489
3, 0.60625281,
       0.91224952, 0.75334057, 0.75879306, 0.5710054
8, 0.98242394,
       0.4034332 , 0.46750637, 0.47205267, 0.8283087
4, 0.67958694,
       0.84624222, 0.38998106, 0.60138008, 0.4303328
5, 0.58575614,
       0.72992691, 0.44086161, 0.21546613, 0.8366079
1, 0.70091667,
       0.65390153, 0.8829348 , 0.72034761, 0.4635184
2, 0.85942801,
       0.29297696, 0.40447946, 0.65573051, 0.8379782
```

```
3, 0.63467034,
        0.7319606 , 0.97979748, 0.22591391, 0.7814395
4, 0.64677296,
        0.27131764, 0.66862327, 0.60950501, 0.4806255
1, 0.93391771,
        0.56902771, 0.38568622, 0.348932  , 0.6249423
8, 0.34982382,
        0.35166567, 0.81298598, 0.59648047, 0.8695733
3, 0.71943754,
        0.45679997, 0.48496706, 0.72286644, 0.2953835
9, 0.79582533,
        0.46760114, 0.67828772, 0.67883433, 0.6516894
1, 0.83776143,
        0.61611539, 0.98303505, 0.59678071, 0.6008586
3, 0.52494716,
        0.53873482, 0.63684489, 0.50774389, 0.6236085
2, 0.63971866,
        0.95834086, 0.70657401, 0.37470773, 0.5739381
, 0.34114826,
        0.43596874, 0.80545546, 0.16948187, 0.5752372
5, 0.47504385,
        0.7978173 , 0.91738198, 0.76164394, 0.8629260
7, 0.5341799 ,
        0.43914779, 0.44464322, 0.3554489 , 0.2732362
3, 0.89913037,
        0.81451059, 0.64782501, 0.92054441, 0.8921602
1, 0.31424726,
        0.9623672 , 0.92904984, 0.9808496 , 0.6944294
7, 0.50784292,
        0.48764802, 0.41643588, 0.69705644, 0.2148158
, 0.50237935,
        0.69171392, 0.40122293, 0.75039878, 0.8135467
5, 0.9422711 ,
        0.84681109, 0.28927443, 0.67763596, 0.3209347
2, 0.57147493,
        0.39206698, 0.70836429, 0.3435361 , 0.5415438
4, 0.69181114,
        0.27088757, 0.61302608, 0.88045321, 0.5409831
3, 0.85206581,
        0.34219011, 0.35369037, 0.5021619 , 0.6913736
8, 0.77531435,
        0.53858279, 0.35298938, 0.6533829 , 0.6715022
2, 0.81353448,
        0.8856535 , 0.84202067, 0.24510023, 0.7047935
8, 0.957143  ,
```

```
       0.44740698, 0.36650411, 0.82491493, 0.9192492
1, 0.70098476,
       0.52979499, 0.24490036, 0.86720487, 0.4514611
7, 0.59474726,
       0.62640475, 0.70520926, 0.23437945, 0.6720264
3, 0.90593173])
```

In [27]:

```
#e-h.
nb = GaussianNB().fit(X.transpose(), p_tf)
xx, yy = np.meshgrid(np.linspace(-1, 1, 100), np.linspace(-1, 1,
100))
Z = nb.predict_proba(np.c_[xx.ravel(), yy.ravel()])
Z = Z[:,1].reshape(xx.shape)

data = np.vstack((X, p_tf))
df = pd.DataFrame(dict(x=data[0,:], y=data[1,:], label=data[2,:]
))

groups = df.groupby('label')
f, x = plt.subplots()
for name, group in groups:
    x.plot(group.x, group.y, marker='o', linestyle='', ms=8, lab
el=name)
x.contour(xx, yy, Z,  [0.5], colors='blue')

plt.xlabel('X1')
plt.ylabel('X2')
plt.title('The Bayes Classifer')
plt.legend(['success','failure'], loc=1)
plt.show()
```



2 Exploring Simulated Differences between LDA and QDA

2

If the Bayes boundary is linear, QDA will perform better on the training set. Because the higher the flexiblity, the closer fit will QDA gets. But LDA will perform better on the test set,because QDA could overfits the training set, which leads to a bad performance on the Bayes decision boundary.

In [39]:

```
#a. Repeat the following process 1000 times.

def linear(n):
        #Simulate a dataset of 1000 observation
        x1_2 = np.random.uniform(-1,1,1000)
        x2_2 = np.random.uniform(-1,1,1000)
        Y = (x1_2 + x2_2 + np.random.normal(size = 1000))>=0
        data=np.vstack((x1_2, x2_2, Y))
        s = np.arange(1000)
        np.random.shuffle(s)

        #Randomly split the dataset into 70/30% training/test se
ts
        train_idx = seq[:700]
        test_idx = seq[700:]
        train, test = data[:,train_idx], data[:,test_idx]
        x_train = train[:2,:].T
        y_train = train[2,:].T
        x_test=test[:2,:].T
        y_test=test[2,:].T

        #Use the training dataset to estimate LDA and QDA models
.
        # LDA model
        lda = LinearDiscriminantAnalysis()
        lda_model = lda.fit(x_train,y_train)
        # QDA model
        qda = QuadraticDiscriminantAnalysis()
        qda_model= qda.fit(x_train, y_train)

       # Calculate each model's training and test error rate.
        lda_train = 1 - lda_model.score(x_train, y_train)
        lda_test = 1 - lda_model.score(x_test, y_test)
        qda_train = 1 - qda_model.score(x_train, y_train)
        qda_test  = 1 - qda_model.score(x_test, y_test)

        return lda_train, lda_test, qda_train, qda_test
```

In [41]:

```
# Repeat the following process 1000 times.
lda_train_err = np.zeros(1000)
lda_test_err = np.zeros(1000)
qda_train_err = np.zeros(1000)
qda_test_err = np.zeros(1000)


for i in range(1000):
    lda_train, lda_test, qda_train, qda_test=linear(i)
    lda_train_err[i] = lda_train
    lda_test_err[i] = lda_test
    qda_train_err[i] = qda_train
    qda_test_err[i] = qda_test
```

In [42]:

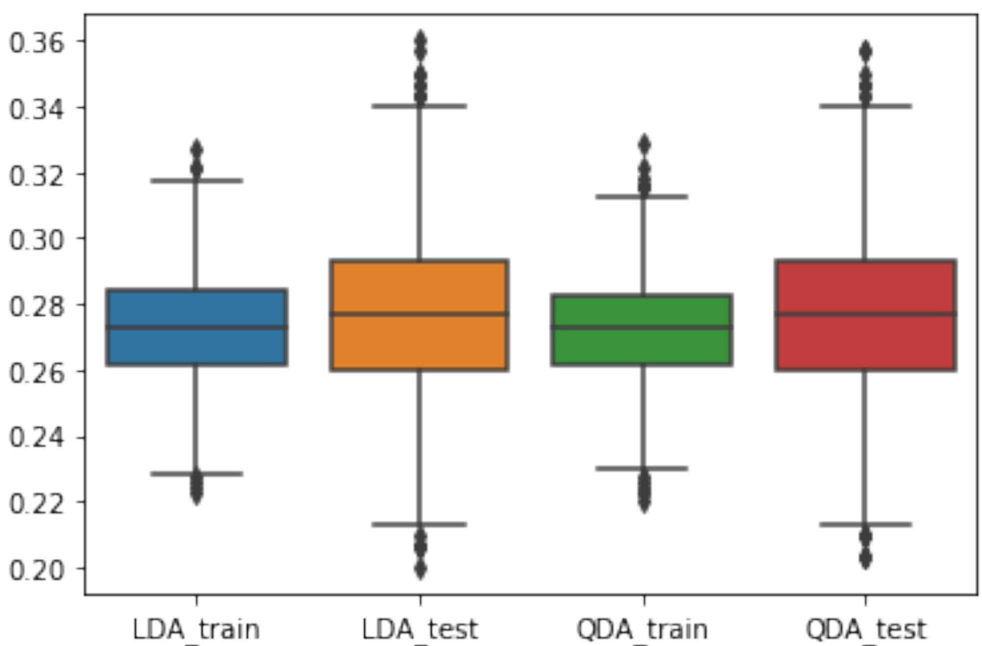
```
#b. Summarize all the simulations’ error rates and report the re
sults in tabular and graphical form.

df = pd.DataFrame({'LDA_train':lda_train_err,
                   'LDA_test':lda_test_err,
                   'QDA_train':qda_train_err,
                   'QDA_test':qda_test_err
                   })
df.describe()
```

Out[42]:

	LDA_train	LDA_test	QDA_train	QDA_test
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.273091	0.276090	0.272161	0.276303
std	0.016366	0.025906	0.016601	0.026090
min	0.222857	0.200000	0.220000	0.203333
25%	0.261429	0.260000	0.261429	0.260000
50%	0.272857	0.276667	0.272857	0.276667
75%	0.284286	0.293333	0.282857	0.293333
max	0.327143	0.360000	0.328571	0.356667

In [45]:

```
# Draw the boxplot
sns.boxplot(data=df)
plt.show()
```



According to the result, QDA performs better on the trainning set LDA performs better on test set. This supports my answer above.

3

If the Bayes boundary is nonlinear, QDA performs better on both the training set and test set. Because the high flexibility allows QDA to perform better on describing the nonlinear relationship.

In [53]:

```
#a. Repeat the following process 1000 times.

def nonlinear(n):
        np.random.seed(n)
        #Simulate a dataset of 1000 observation
        x1_2 = np.random.uniform(-1,1,1000)
        x2_2 = np.random.uniform(-1,1,1000)
        Y = (x1_2 + x1_2**2 + x2_2 + x2_2**2 + np.random.normal(
size = 1000))>=0
        data=np.vstack((x1_2, x2_2, Y))
        s = np.arange(1000)
        np.random.shuffle(s)

        #Randomly split your dataset into 70/30% training/test s
ets
        train_idx = s[:700]
        test_idx = s[700:]
        train, test = data[:,train_idx], data[:,test_idx]
        x_train = train[:2,:].T
        y_train = train[2,:].T
        x_test=test[:2,:].T
        y_test=test[2,:].T

        #Use the training dataset to estimate LDA and QDA models
.
        # LDA model
        lda = LinearDiscriminantAnalysis()
        lda_model = lda.fit(x_train,y_train)
        # QDA model
        qda = QuadraticDiscriminantAnalysis()
        qda_model= qda.fit(x_train, y_train)

       # Calculate each model's training and test error rate.
        lda_train = 1 - lda_model.score(x_train, y_train)
```

```
        lda_test = 1 - lda_model.score(x_test, y_test)
        qda_train = 1 - qda_model.score(x_train, y_train)
        qda_test  = 1 - qda_model.score(x_test, y_test)

        return lda_train, lda_test, qda_train, qda_test

lda_train_err = np.zeros(1000)
lda_test_err = np.zeros(1000)
qda_train_err = np.zeros(1000)
qda_test_err = np.zeros(1000)


for i in range(1000):
    lda_train, lda_test, qda_train, qda_test=nonlinear(i)
    lda_train_err[i] = lda_train
    lda_test_err[i] = lda_test
    qda_train_err[i] = qda_train
    qda_test_err[i] = qda_test
```

In [50]:

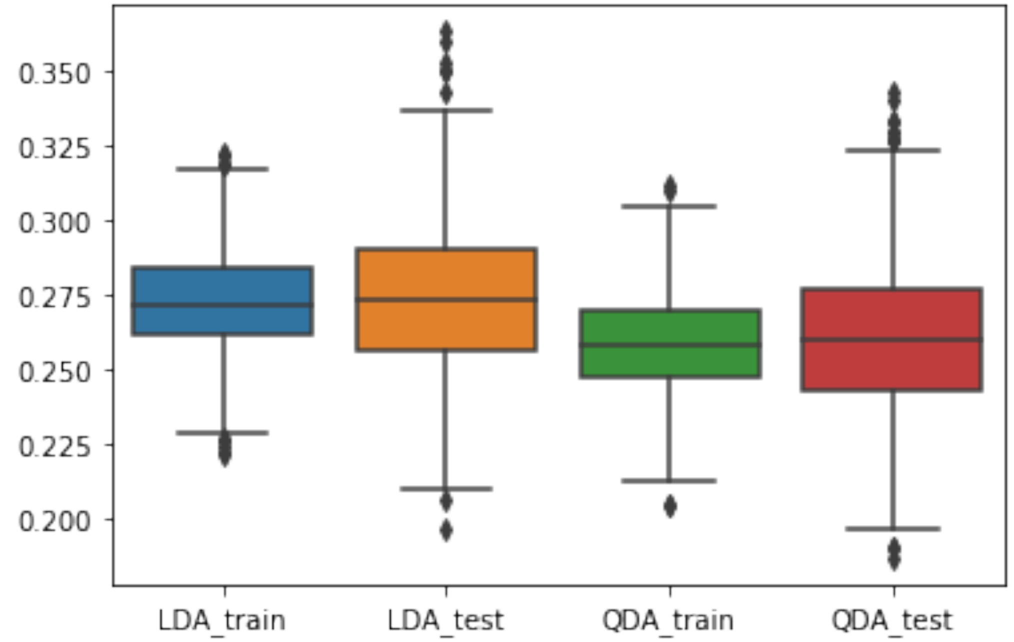
```
#b. Summarize all the simulations' error rates and report the re
sults in tabular and graphical form.
df = pd.DataFrame({'LDA_train':lda_train_err,
                   'LDA_test':lda_test_err,
                   'QDA_train':qda_train_err,
                   'QDA_test':qda_test_err
                   })
#Summarize all the simulations' error rates
df.describe()
```

Out[50]:

	LDA_train	LDA_test	QDA_train	QDA_test
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.272387	0.274833	0.259020	0.261623
std	0.016924	0.025222	0.016340	0.024093
min	0.221429	0.196667	0.204286	0.186667
25%	0.261429	0.256667	0.247143	0.243333
50%	0.271429	0.273333	0.258571	0.260000
75%	0.284286	0.290000	0.270000	0.276667
max	0.322857	0.363333	0.311429	0.343333

In [52]:

```
# Draw the boxplot
sns.boxplot(data=df)
plt.show()
```



According to the result, QDA performs better on both the trainning set and test set. This supports my answer above.

4

When the sample size is relatively small,the test prediction accuracy of LDA is better than QDA. Because LDA is less likely to be overfitting than QDA. When sample size is large enough, the test prediction accuracy of QDA is better than LDA.Because the higher the flexibility, the closer fit will QDA gets and variance won't be a problem in terms of the larger sample sizes.

In [93]:

```
np.random.seed(10)
def nonlinear_new(m):
        #Simulate a dataset of 1000 observation
        x1_3 = np.random.uniform(-1,1,m)
        x2_3 = np.random.uniform(-1,1,m)
        Y = (x1_3 + x1_3**2 + x2_3 + x2_3**2 + np.random.normal(
size = m))>=0
```

```
size = m))>=0
        data=np.vstack((x1_3, x2_3, Y))

        s = np.arange(m)
        np.random.shuffle(s)
        #Randomly split your dataset into 70/30% training/test s
ets
        train_idx = s[:int(0.7*m)]
        test_idx = s[int(0.7*m):]
        train, test = data[:,train_idx], data[:,test_idx]
        x_train = train[:2,:].T
        y_train = train[2,:].T
        x_test=test[:2,:].T
        y_test=test[2,:].T

        #Use the training dataset to estimate LDA and QDA models
.
        # LDA model
        lda = LinearDiscriminantAnalysis()
        lda_model = lda.fit(x_train,y_train)
        # QDA model
        qda = QuadraticDiscriminantAnalysis()
        qda_model= qda.fit(x_train, y_train)

       # Calculate each model’s training and test error rate.
        lda_train = 1 - lda_model.score(x_train, y_train)
        lda_test = 1 - lda_model.score(x_test, y_test)
        qda_train = 1 - qda_model.score(x_train, y_train)
        qda_test  = 1 - qda_model.score(x_test, y_test)

        return lda_train, lda_test, qda_train, qda_test

# simulating 1000 times for N = [100, 1000, 10000, 100000]
N = [10**2, 10**3, 10**4, 10**5]
lda_err = np.zeros(len(N))
qda_err = np.zeros(len(N))
for i in range(len(N)):
    lda_test_err = np.zeros(1000)
    qda_test_err = np.zeros(1000)


    for j in range(1000):
        lda_train, lda_test, qda_train, qda_test= nonlinear_new(
N[i])
        lda_test_err[j] = lda_test
        qda_test_err[j] = qda_test
```

```
    lda_err[i] = lda_test_err.mean()
    qda_err[i] =qda_test_err.mean()
```

In [94]:

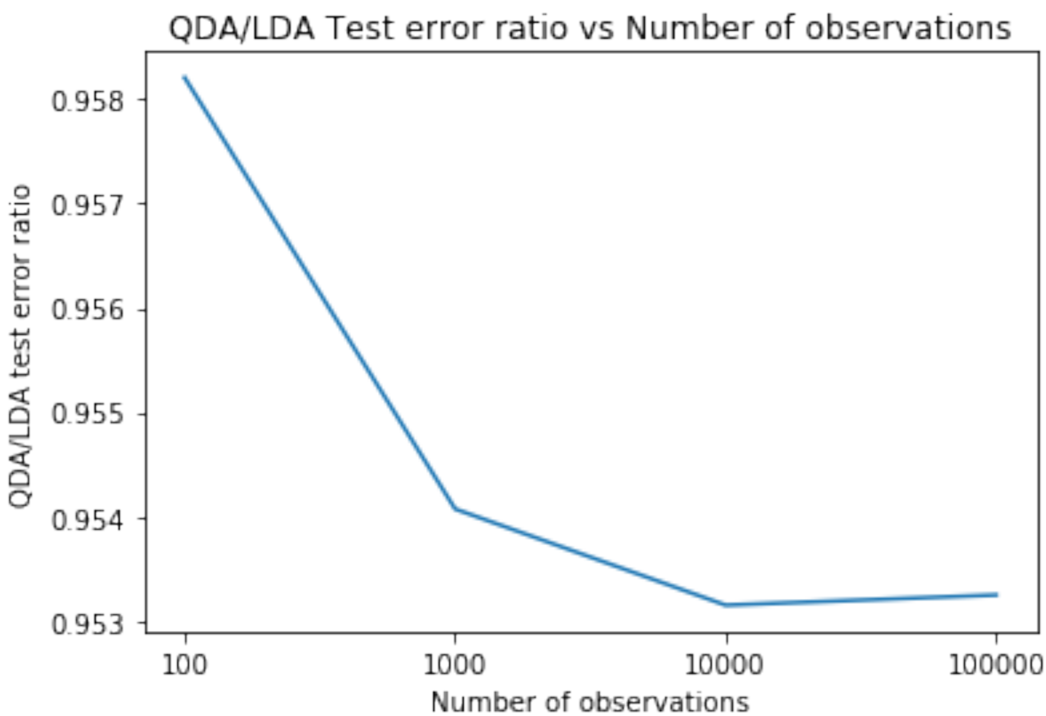
```
d = {'Sample Size':["100", "1000", "10000", "100000"],
        'LDA_error':lda_err,
        'QDA_error':qda_err}

df = pd.DataFrame(d)
df
```

Out[94]:

	Sample Size	LDA_error	QDA_error
0	100	0.286333	0.274367
1	1000	0.273327	0.260773
2	10000	0.273855	0.261025
3	100000	0.273349	0.260569

In [97]:

```
ratio=np.array(qda_err/lda_err)
plt.plot([1,2,3,4], ratio)
plt.xticks([1,2,3,4], ['100', '1000', '10000', '100000'])
plt.xlabel('Number of observations')
plt.ylabel('QDA/LDA test error ratio')
plt.title('QDA/LDA Test error ratio vs Number of observations')
plt.show()
```



According to the above graph, we can expect the test error rate of QDA relative to LDA to decline, which proves my answer above.

5 Modeling voter turnout

In [64]:

```
mh = pd.read_csv('mental_health.csv')
mh.dropna(inplace=True)
mh.head()
```

Out[64]:

	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

In [75]:

```
#a. Split the data into a training and test set (70/30)
np.random.seed(124)
vote96 = mh['vote96']
x = mh.drop(columns=['vote96'])
x_train, x_test, y_train, y_test = train_test_split(x, vote96, t
est_size=0.3)
```

In [76]:

```
#b. Using the training set and all important predictors, estimat
e the following models with vote96 as the response variable:
models=[]
#i. Logistic regression model
logit = LogisticRegression().fit(x_train, y_train)
models.append(('Logistic', logit))

#ii. Linear discriminant model
lda = LinearDiscriminantAnalysis().fit(x_train, y_train)
models.append(('LDA', lda))

#iii. Quadratic discriminant model
qda = QuadraticDiscriminantAnalysis().fit(x_train, y_train)
models.append(('QDA', qda))

#iv. Naive Bayes (you can use the default hyperparameter setting
s)
nb = GaussianNB().fit(x_train, y_train)
models.append(('Naive Bayes', nb))

#v. K-nearest neighbors with K = 1,2,...,10 (that is, 10 separat
e models varying K) and Euclidean distance metrics
m_knn = []
for k in range(1,11):
    knn= KNeighborsClassifier(n_neighbors=k).fit(x_train, y_trai
n)
    models.append(('KNN_{}'.format(k), knn))
```

```
//anaconda3/lib/python3.7/site-packages/sklearn/line
ar_model/logistic.py:432: FutureWarning: Default sol
ver will be changed to 'lbfgs' in 0.22. Specify a so
lver to silence this warning.
  FutureWarning)
```

In [92]:

```
#c. Using the test set, create the list for prediction,names of
the models,  prediction_probability,
preds = []
names=[]
probs = []

for name, model in models:
    preds.append(model.predict(x_test))
    names.append(name)
    probs.append(model.predict_proba(x_test)[:, 1])

error_rates = [ (1 - accuracy_score(y_test, pred)) for pred in p
reds]

dict = {'Model Name':names,
        'Error rate': error_rates}

df = pd.DataFrame(dict)
df
```

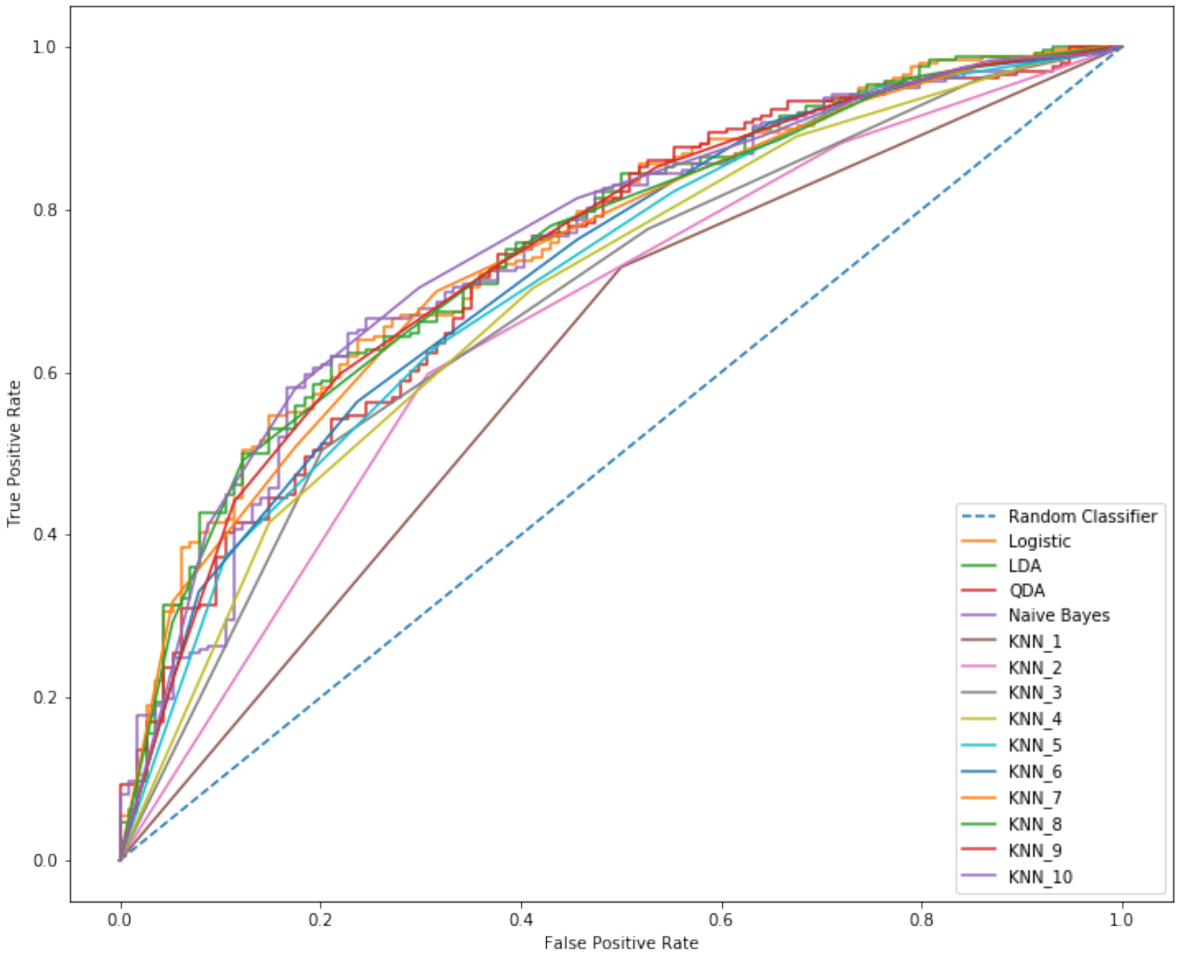
Out[92]:

	Model Name	Error rate
0	Logistic	0.271429
1	LDA	0.277143
2	QDA	0.300000
3	Naive Bayes	0.285714
4	KNN_1	0.345714
5	KNN_2	0.371429
6	KNN_3	0.322857
7	KNN_4	0.334286
8	KNN_5	0.300000
9	KNN_6	0.308571
10	KNN_7	0.291429
11	KNN_8	0.288571
12	KNN_9	0.274286
13	KNN_10	0.274286

In [113]:

```
def auc_roc(prob, name):
    fpr, tpr, _ = roc_curve(y_test, prob)
    aucs=auc(fpr, tpr)
    print(name+ ': AUC = %f' % (aucs))
# plot the roc curve for the model
    plt.plot(fpr, tpr, label=name)
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend()

plt.figure(figsize=(12,10))
rand_probs = [0] * len(y_test)
rand_fpr, rand_tpr, _ = roc_curve(y_test, rand_probs)
plt.plot(rand_fpr, rand_tpr, linestyle='--', label='Random Class
ifier')
for i, model in enumerate(models):
    auc_roc(probs[i], names[i])
```

```
Logistic: AUC = 0.757322
LDA: AUC = 0.756430
QDA: AUC = 0.736247
Naive Bayes: AUC = 0.745503
KNN_1: AUC = 0.614407
KNN_2: AUC = 0.660608
KNN_3: AUC = 0.685753
KNN_4: AUC = 0.695863
KNN_5: AUC = 0.715265
KNN_6: AUC = 0.726862
KNN_7: AUC = 0.744555
KNN_8: AUC = 0.749350
KNN_9: AUC = 0.747863
KNN_10: AUC = 0.761095
```



d.Accroding to the above analyses, the best models for this dataset are Logistic and LDA, which produces the highest AUC and lowest error rate.