Perspective Homework II
Yuxin Wu (yuxinwu@uchicago.edu)
February 2nd, 2020

In [1]:

```
import pandas as pd
import numpy as np
import random
import math
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
import matplotlib.patches as mpatches
from sklearn import metrics
from sklearn.metrics import roc_auc_score
```

The Bayes Classifier
Question I

In [2]:

```
random.seed()
```

In [3]:

```
x1 = np.random.uniform(-1,1,200)
x2 = np.random.uniform(-1,1,200)
```

In [4]:

```
x = []
i = 0
while i < 200:
    x.append([x1[i], x2[i]])
    i += 1
```

In [5]:

```
mu, sigma = 0, 0.25**(1/2)
s = np.random.normal(mu, sigma, 200)
```

In [6]:

```
i = 0
lst = []
while i < 200:
    lst.append((x1[i], x2[i], s[i]))
    i += 1
```

In [7]:

```
lst_y = []
for i in lst:
    x1 = i[0]
    x2 = i[1]
    err = i[2]
    y = x1 + x1**2 + x2 + x2**2 + err
    lst_y.append(y)
```

In [8]:

```
success_rate = []
for i in lst_y:
```

```
    p = math.exp(i) / (1 + math.exp(i))
    success_rate.append(p)
```

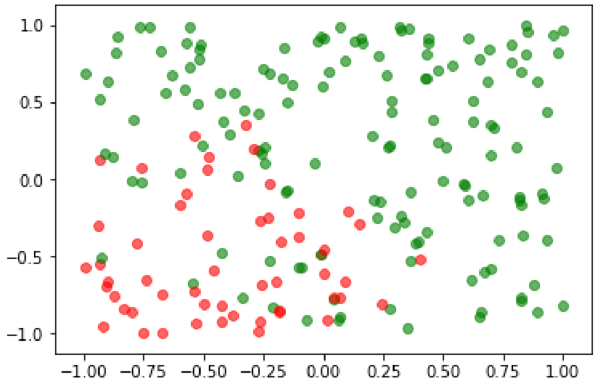
In [9]:

```
rate_table = []
i = 0
while i < 200:
    k = (lst[i][0], lst[i][1], success_rate[i])
    rate_table.append(k)
    i += 1
```

In [10]:

```
for i in rate_table:
    if i[2] > 0.5:
        ax = plt.subplot()
        ax.scatter(i[0], i[1], c='green', alpha=0.6)
    else:
        ax = plt.subplot()
        ax.scatter(i[0], i[1], c='red', alpha=0.6)
plt.show()
```

```
C:\Users\yw214\Anaconda3\lib\site-packages\matplotlib\figure.py:98: MatplotlibDeprecationWarning:
Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.
In a future version, a new instance will always be created and returned.  Meanwhile, this warning
can be suppressed, and the future behavior ensured, by passing a unique label to each axes instanc
e.
  "Adding an axes using the same arguments as a previous axes "
```



In [11]:

```
y = []
for i in rate_table:
    if i[2] > 0.5:
        y.append(1)
    else:
        y.append(0)
```

In [12]:

```
x = pd.DataFrame(x)
y = pd.DataFrame(y)
```

In [13]:

```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(x, y)
```

```
C:\Users\yw214\Anaconda3\lib\site-packages\sklearn\utils\validation.py:761: DataConversionWarning:
A column-vector y was passed when a 1d array was expected. Please change the shape of y to
(n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

Out[13]:

```
GaussianNB(priors=None, var_smoothing=1e-09)
```

In [14]:

```python
p = np.linspace(-1,1)
q = np.linspace(-1,1)
xv, yv = np.meshgrid(p,q)
Z = gnb.predict_proba(np.c_[xv.ravel(), yv.ravel()])
Z = Z[:,1].reshape(xv.shape)
```

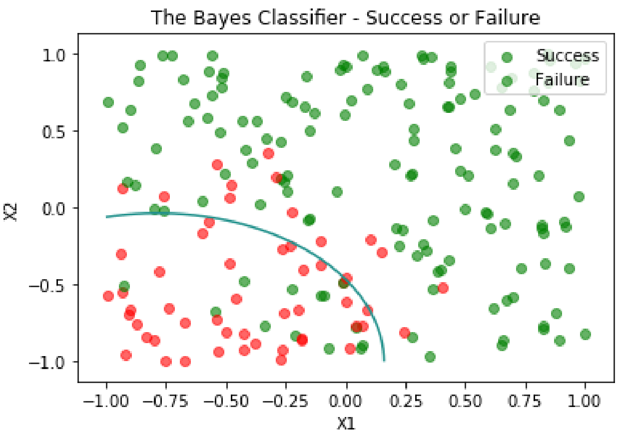
In [15]:

```python
for i in rate_table:
    if i[2] > 0.5:
        ax = plt.subplot()
        ax.scatter(i[0], i[1], c='green', alpha=0.6)
    else:
        ax = plt.subplot()
        ax.scatter(i[0], i[1], c='red', alpha=0.6)

plt.contour(xv, yv, Z, 1)
plt.xlabel("X1")
plt.ylabel("X2")

plt.title('The Bayes Classifier - Success or Failure')
plt.legend(['Success', "Failure"], loc = 1)
plt.show()
```

```
C:\Users\yw214\Anaconda3\lib\site-packages\matplotlib\figure.py:98: MatplotlibDeprecationWarning:
Adding an axes using the same arguments as a previous axes currently reuses the earlier instance.
In a future version, a new instance will always be created and returned.  Meanwhile, this warning
can be suppressed, and the future behavior ensured, by passing a unique label to each axes instanc
e.
  "Adding an axes using the same arguments as a previous axes "
```



Exploring Simulated Differences between LDA and QDA
Question II

If the Bayes decision boundary is linear, do we expect LDA or QDA to perform better on the training set? On the test set?
If the Bayes decision boundary is linear, QDA will perform better on training set. Because its higher flexiblity may yield a closer fit. While LDA will perform better on test set. Because QDA could overfit the linearity on the Bayes decision boundary.

In [16]:

```python
count = 0
lda_train_error_total = 0
lda_test_error_total = 0
qda_train_error_total = 0
qda_test_error_total = 0
```

```python
error_lst = []

while count < 1000:
    x1 = np.random.uniform(-1,1,1000)
    x2 = np.random.uniform(-1,1,1000)
    mu, sigma = 0, 1
    err = np.random.normal(mu, sigma, 1000)
    y = x1 + x2 + err

    x = []
    i = 0
    while i < 1000:
        x.append([x1[i], x2[i]])
        i += 1

    Y = []
    i = 0
    while i < 1000:
        if y[i] >= 0:
            Y.append(True)
        else:
            Y.append(False)
        i += 1

    x_train, x_test, y_train, y_test = train_test_split(x, Y, test_size=0.3)

    lda = LinearDiscriminantAnalysis()
    lda.fit(x_train, y_train)

    lda_train_error = (1 - lda.score(x_train, y_train))
    lda_test_error = (1 - lda.score(x_test, y_test))

    qda = QuadraticDiscriminantAnalysis()
    qda.fit(x_train, y_train)

    qda_train_error = (1 - qda.score(x_train, y_train))
    qda_test_error = (1 - qda.score(x_test, y_test))

    lda_train_error_total += lda_train_error
    lda_test_error_total += lda_test_error
    qda_train_error_total += qda_train_error
    qda_test_error_total += qda_test_error

    error_lst.append([lda_train_error, lda_test_error, qda_train_error, qda_test_error])
    count += 1
```

In [17]:

```python
print("Avg Error Rate of LDA Training Set: ", lda_train_error_total / 1000)
print("Avg Error Rate of LDA Testing Set: ",lda_test_error_total / 1000)
print("Avg Error Rate of QDA Training Set: ", qda_train_error_total / 1000)
print("Avg Error Rate of QDA Testining Set: ", qda_test_error_total / 1000)
```

```
Avg Error Rate of LDA Training Set:  0.2740414285714293
Avg Error Rate of LDA Testing Set:  0.27845333333333283
Avg Error Rate of QDA Training Set:  0.27304714285714327
Avg Error Rate of QDA Testining Set:  0.27877999999999975
```

In [18]:

```python
df = pd.DataFrame(error_lst)
df.head()
```

Out[18]:

	0	1	2	3
0	0.274286	0.300000	0.275714	0.300000
1	0.290000	0.240000	0.292857	0.233333
2	0.304286	0.293333	0.305714	0.286667
3	0.282857	0.280000	0.277143	0.296667

	0	1	2	3
4	0.270000	0.313333	0.268571	0.316667

In [19]:

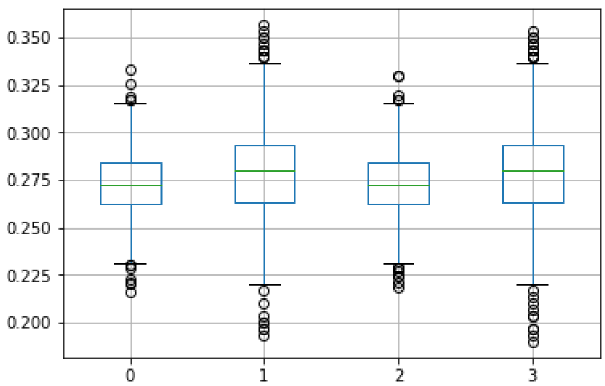
```
print(df.describe())
print("0: Avg Error Rate of LDA Training Set")
print("1: Avg Error Rate of LDA Testing Set")
print("2: Avg Error Rate of QDA Training Set")
print("3: Avg Error Rate of QDA Testining Set")
```

```
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.274041     0.278453     0.273047     0.278780
std       0.016904     0.025210     0.016656     0.025111
min       0.215714     0.193333     0.218571     0.190000
25%       0.262857     0.263333     0.262857     0.263333
50%       0.272857     0.280000     0.272857     0.280000
75%       0.284286     0.293333     0.284286     0.293333
max       0.332857     0.356667     0.330000     0.353333
0: Avg Error Rate of LDA Training Set
1: Avg Error Rate of LDA Testing Set
2: Avg Error Rate of QDA Training Set
3: Avg Error Rate of QDA Testining Set
```

In [20]:

```
df.boxplot()
print("0: Avg Error Rate of LDA Training Set")
print("1: Avg Error Rate of LDA Testing Set")
print("2: Avg Error Rate of QDA Training Set")
print("3: Avg Error Rate of QDA Testining Set")
```

```
0: Avg Error Rate of LDA Training Set
1: Avg Error Rate of LDA Testing Set
2: Avg Error Rate of QDA Training Set
3: Avg Error Rate of QDA Testining Set
```



As it is shown in the result, QDA performs better on the trainning set and LDA performs better on test set. This proves my answer above.

Exploring Simulated Differences between LDA and QD
Question III

If the Bayes decision boundary is non-linear, do we expect LDA or QDA to perform better on the training set? On the test set?
If the decision boundary is non-linear, we would expect that QDA performs better on both the training set and test set. Because QDA's flexibility will perform better under a non-linear situation.

In [21]:

```
count = 0
error_lst = []
```

```
lda_train_error_total = 0
lda_test_error_total = 0
qda_train_error_total = 0
qda_test_error_total = 0

while count < 1000:
    x1 = np.random.uniform(-1,1,1000)
    x2 = np.random.uniform(-1,1,1000)
    mu, sigma = 0, 1
    err = np.random.normal(mu, sigma, 1000)
    y = x1 + x1**2 + x2 + x2**2 + err

    x = []
    i = 0
    while i < 1000:
        x.append([x1[i], x2[i]])
        i += 1

    Y = []
    i = 0
    while i < 1000:
        if y[i] >= 0:
            Y.append(True)
        else:
            Y.append(False)
        i += 1

    x_train, x_test, y_train, y_test = train_test_split(x, Y, test_size=0.3)

    lda = LinearDiscriminantAnalysis()
    lda.fit(x_train, y_train)

    lda_train_error = (1 - lda.score(x_train, y_train))
    lda_test_error = (1 - lda.score(x_test, y_test))

    qda = QuadraticDiscriminantAnalysis()
    qda.fit(x_train, y_train)

    qda_train_error = (1 - qda.score(x_train, y_train))
    qda_test_error = (1 - qda.score(x_test, y_test))

    lda_train_error_total += lda_train_error
    lda_test_error_total += lda_test_error
    qda_train_error_total += qda_train_error
    qda_test_error_total += qda_test_error

    error_lst.append([lda_train_error, lda_test_error, qda_train_error, qda_test_error])

    count += 1
```

In [22]:

```
print("Error Rate of LDA Training Set: ", lda_train_error_total / 1000)
print("Error Rate of LDA Testing Set: ",lda_test_error_total / 1000)
print("Error Rate of QDA Training Set: ", qda_train_error_total / 1000)
print("Error Rate of QDA Testining Set: ", qda_test_error_total / 1000)
```

```
Error Rate of LDA Training Set:  0.27284142857142896
Error Rate of LDA Testing Set:  0.2744966666666664
Error Rate of QDA Training Set:  0.25924571428571475
Error Rate of QDA Testining Set:  0.2617399999999999
```

In [23]:

```
df = pd.DataFrame(error_lst)
df.head()
```

Out[23]:

	0	1	2	3
0	0.270000	0.300000	0.262857	0.290000
1	0.261429	0.226667	0.255714	0.230000

	0	1	2	3
2	0.272857	0.293333	0.261429	0.323333
3	0.277143	0.276667	0.252857	0.263333
4	0.270000	0.246667	0.257143	0.250000

In [24]:

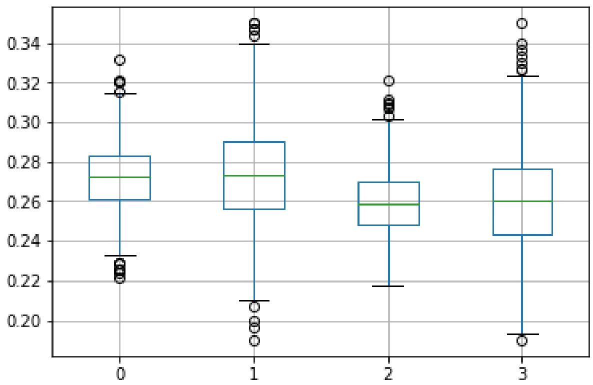
```
print(df.describe())
print("0: Avg Error Rate of LDA Training Set")
print("1: Avg Error Rate of LDA Testing Set")
print("2: Avg Error Rate of QDA Training Set")
print("3: Avg Error Rate of QDA Testining Set")
```

```
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.272841     0.274497     0.259246     0.261740
std       0.016310     0.026162     0.015822     0.025354
min       0.221429     0.190000     0.217143     0.190000
25%       0.261429     0.256667     0.248571     0.243333
50%       0.272857     0.273333     0.258571     0.260000
75%       0.282857     0.290000     0.270000     0.276667
max       0.331429     0.350000     0.321429     0.350000
0: Avg Error Rate of LDA Training Set
1: Avg Error Rate of LDA Testing Set
2: Avg Error Rate of QDA Training Set
3: Avg Error Rate of QDA Testining Set
```

In [25]:

```
df.boxplot()
print("0: Avg Error Rate of LDA Training Set")
print("1: Avg Error Rate of LDA Testing Set")
print("2: Avg Error Rate of QDA Training Set")
print("3: Avg Error Rate of QDA Testining Set")
```

```
0: Avg Error Rate of LDA Training Set
1: Avg Error Rate of LDA Testing Set
2: Avg Error Rate of QDA Training Set
3: Avg Error Rate of QDA Testining Set
```



As it is shown in the result, QDA performs better on both the trainning set and the test set. This proves my answer above.

Exploring Simulated Differences between LDA and QD Question IV

In general, as sample size n increases, do we expect the test error rate of QDA relative to LDA to improve, decline, or be unchanged? Why?
QDA would have a better performance with a large sample size. QDA is more flexible than LDA and so has higher variance, but with a large sample size, the variance will not be a big concern.

In [26]:

In [26]:

```
def qfour(n):

    count = 0
    error_lst = []

    while count < 1000:
        x1 = np.random.uniform(-1,1,n)
        x2 = np.random.uniform(-1,1,n)
        mu, sigma = 0, 1
        err = np.random.normal(mu, sigma, n)
        y = x1 + x1 ** 2 + x2 + x2 ** 2 + err

        Y = y > 0
        x = np.column_stack((x1, x2))

        x_train, x_test, y_train, y_test = train_test_split(x, Y, test_size=0.3)

        lda = LinearDiscriminantAnalysis()
        lda.fit(x_train, y_train)

        lda_train_error = (1 - lda.score(x_train, y_train))
        lda_test_error = (1 - lda.score(x_test, y_test))

        qda = QuadraticDiscriminantAnalysis()
        qda.fit(x_train, y_train)

        qda_train_error = (1 - qda.score(x_train, y_train))
        qda_test_error = (1 - qda.score(x_test, y_test))

        error_lst.append([lda_train_error, lda_test_error, qda_train_error, qda_test_error])

        count += 1

    return error_lst
```

In [27]:

```
dt_100_lst = qfour(100)
dt_1000_lst = qfour(1000)
dt_10000_lst = qfour(10000)
dt_100000_lst = qfour(100000)
```

In [28]:

```
dt_100_lst = pd.DataFrame(dt_100_lst)
dt_1000_lst = pd.DataFrame(dt_1000_lst)
dt_10000_lst = pd.DataFrame(dt_10000_lst)
dt_100000_lst = pd.DataFrame(dt_100000_lst)
```

In [29]:

```
dt_100_lst.describe()
```

Out[29]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.265200	0.287367	0.242900	0.273933
std	0.053818	0.082835	0.048673	0.083485
min	0.114286	0.066667	0.085714	0.033333
25%	0.228571	0.233333	0.214286	0.233333
50%	0.257143	0.300000	0.242857	0.266667
75%	0.300000	0.333333	0.271429	0.333333
max	0.457143	0.566667	0.400000	0.533333

In [30]:

```
dt_1000_lst.describe()
```

Out[30]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.272840	0.275127	0.258937	0.261993
std	0.017277	0.026324	0.016675	0.025490
min	0.221429	0.186667	0.195714	0.170000
25%	0.260000	0.256667	0.247143	0.246667
50%	0.272857	0.273333	0.258571	0.261667
75%	0.284286	0.293333	0.270000	0.276667
max	0.327143	0.363333	0.314286	0.343333

In [31]:

```
dt_10000_lst.describe()
```

Out[31]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.273263	0.273331	0.260324	0.260469
std	0.005238	0.008066	0.005060	0.008044
min	0.255143	0.248667	0.244714	0.233667
25%	0.269857	0.267667	0.257000	0.255000
50%	0.273143	0.273333	0.260286	0.260333
75%	0.276714	0.279000	0.263714	0.266000
max	0.289714	0.299333	0.276714	0.284667

In [32]:

```
dt_100000_lst.describe()
```

Out[32]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.273399	0.273530	0.260556	0.260711
std	0.001718	0.002562	0.001672	0.002513
min	0.268429	0.266433	0.254871	0.253867
25%	0.272214	0.271825	0.259400	0.258967
50%	0.273407	0.273567	0.260521	0.260733
75%	0.274614	0.275233	0.261743	0.262367
max	0.278943	0.282200	0.265200	0.268867

In [33]:

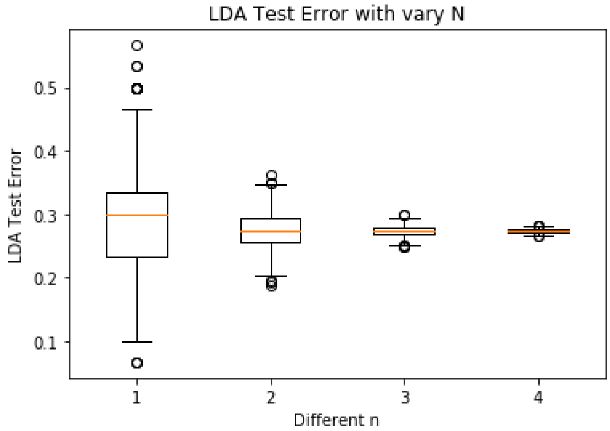
```
lda_test_error_100 = pd.DataFrame(dt_100_lst[1])
lda_test_error_1000 = pd.DataFrame(dt_1000_lst[1])
lda_test_error_10000 = pd.DataFrame(dt_10000_lst[1])
lda_test_error_100000 = pd.DataFrame(dt_100000_lst[1])
```

In [34]:

```
lda_test_100 = list(dt_100_lst[1])
lda_test_1000 = list(dt_1000_lst[1])
lda_test_10000 = list(dt_10000_lst[1])
lda_test_100000 = list(dt_100000_lst[1])
lda_plot_data = [lda_test_100,lda_test_1000,lda_test_10000,lda_test_100000]
```

In [35]:

```
plt.boxplot(lda_plot_data)
plt.xlabel("Different n")
plt.ylabel("LDA Test Error")
plt.title("LDA Test Error with vary N")
plt.show()
print("1 : N = 100")
print("2 : N = 1000")
print("3 : N = 10000")
print("4 : N = 100000")
```



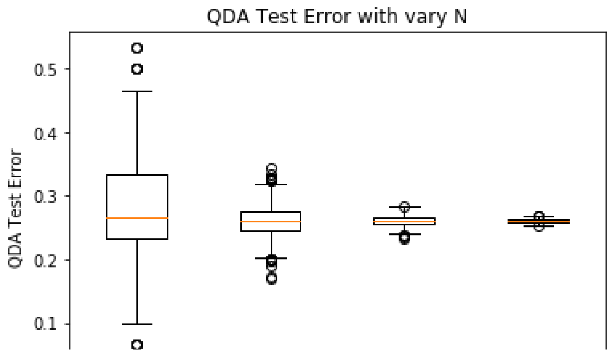

```
1 : N = 100
2 : N = 1000
3 : N = 10000
4 : N = 100000
```

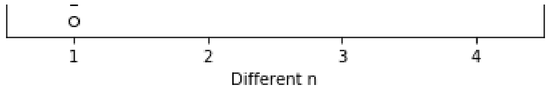
In [36]:

```
qda_test_100 = list(dt_100_lst[3])
qda_test_1000 = list(dt_1000_lst[3])
qda_test_10000 = list(dt_10000_lst[3])
qda_test_100000 = list(dt_100000_lst[3])
qda_plot_data = [qda_test_100,qda_test_1000,qda_test_10000,qda_test_100000]
```

In [37]:

```
plt.boxplot(qda_plot_data)
plt.xlabel("Different n")
plt.ylabel("QDA Test Error")
plt.title("QDA Test Error with vary N")
plt.show()
print("1 : N = 100")
print("2 : N = 1000")
print("3 : N = 10000")
print("4 : N = 100000")
```





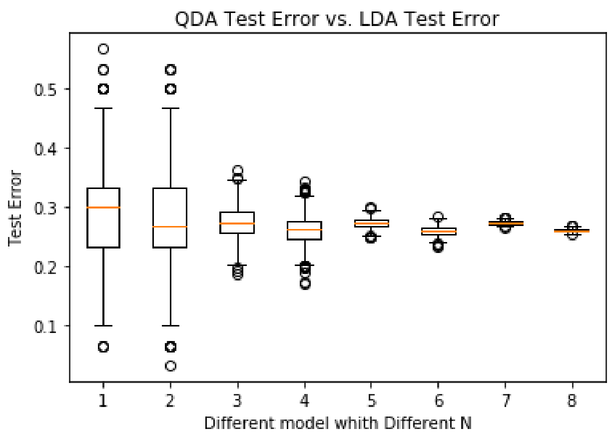

```
1 : N = 100
2 : N = 1000
3 : N = 10000
4 : N = 100000
```

In [38]:

```
plot_data = [lda_test_100,qda_test_100, lda_test_1000, qda_test_1000, lda_test_10000, qda_test_1000
0,\
                 lda_test_100000, qda_test_100000]
```

In [39]:

```
plt.boxplot(plot_data)
plt.xlabel("Different model whith Different N")
plt.ylabel("Test Error")
plt.title("QDA Test Error vs. LDA Test Error")
plt.show()
print("1 : LDA_100")
print("2 : QDA_100")
print("3 : LDA_1000")
print("4 : QDA_1000")
print("5 : LDA_10000")
print("6 : QDA_10000")
print("7 : LDA_100000")
print("8 : QDA_100000")
```




```
1 : LDA_100
2 : QDA_100
3 : LDA_1000
4 : QDA_1000
5 : LDA_10000
6 : QDA_10000
7 : LDA_100000
8 : QDA_100000
```

Modeling voter turnout
Question V

In [43]:

```
data = pd.read_csv("mental_health.csv")
data.head()
data = data.dropna()
```

In [44]:

```
y = data["vote96"]
```

In [45]:

```
x = data.loc[:,['mhealth_sum','age', "educ", "black", "female", "married", "inc10"] ]
```

In [46]:

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)
```

In [47]:

```
from sklearn.linear_model import LogisticRegression
logis = LogisticRegression()
logis.fit(x_train, y_train)
logis_test_error = 1 - logis.score(x_test, y_test)
```

```
C:\Users\yw214\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:433: FutureWarning: De
fault solver will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
```

In [48]:

```
lda = LinearDiscriminantAnalysis()
lda.fit(x_train, y_train)
lda_test_error = 1 - lda.score(x_test, y_test)
```

In [49]:

```
qda = QuadraticDiscriminantAnalysis()
qda.fit(x_train, y_train)
qda_test_error = 1 - qda.score(x_test, y_test)
```

In [50]:

```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(x_train, y_train)
gnb_test_error = 1 - gnb.score(x_test, y_test)
```

In [51]:

```
from sklearn import neighbors
#N = 1
nbrs1 = neighbors.KNeighborsClassifier(n_neighbors=1, metric='euclidean')
nbrs1.fit(x_train, y_train)
n1 = 1 - nbrs1.score(x_test, y_test)


#N = 2
nbrs2 = neighbors.KNeighborsClassifier(n_neighbors=2, metric='euclidean')
nbrs2.fit(x_train, y_train)
n2 = 1 - nbrs2.score(x_test, y_test)


#N = 3
nbrs3 = neighbors.KNeighborsClassifier(n_neighbors=3, metric='euclidean')
nbrs3.fit(x_train, y_train)
n3 = 1 - nbrs3.score(x_test, y_test)

#N = 4
nbrs4 = neighbors.KNeighborsClassifier(n_neighbors=4, metric='euclidean')
nbrs4.fit(x_train, y_train)
n4 = 1 - nbrs4.score(x_test, y_test)

#N = 5
nbrs5 = neighbors.KNeighborsClassifier(n_neighbors=5, metric='euclidean')
nbrs5.fit(x_train, y_train)
n5 = 1 - nbrs5.score(x_test, y_test)
```

```
#N = 6
nbrs6 = neighbors.KNeighborsClassifier(n_neighbors=6, metric='euclidean')
nbrs6.fit(x_train, y_train)
n6 = 1 - nbrs6.score(x_test, y_test)

#N = 7
nbrs7 = neighbors.KNeighborsClassifier(n_neighbors=7, metric='euclidean')
nbrs7.fit(x_train, y_train)
n7 = 1 - nbrs7.score(x_test, y_test)

#N = 8
nbrs8 = neighbors.KNeighborsClassifier(n_neighbors=8, metric='euclidean')
nbrs8.fit(x_train, y_train)
n8 = 1 - nbrs8.score(x_test, y_test)

#N = 9
nbrs9 = neighbors.KNeighborsClassifier(n_neighbors=9, metric='euclidean')
nbrs9.fit(x_train, y_train)
n9 = 1 - nbrs9.score(x_test, y_test)

#N = 10
nbrs10 = neighbors.KNeighborsClassifier(n_neighbors=10, metric='euclidean')
nbrs10.fit(x_train, y_train)
n10 = 1 - nbrs10.score(x_test, y_test)
```

In [52]:

```
print("Logistic Error Rate : ", logis_test_error)
print("LDA Error Rate : ", lda_test_error)
print("QDA Error Rate : ", qda_test_error)
print("Naive Byes Error Rate : ", gnb_test_error)
print("KNN (N = 1) Error Rate : ", n1)
print("KNN (N = 2) Error Rate : ", n2)
print("KNN (N = 3) Error Rate : ", n3)
print("KNN (N = 4) Error Rate : ", n4)
print("KNN (N = 5) Error Rate : ", n5)
print("KNN (N = 6) Error Rate : ", n6)
print("KNN (N = 7) Error Rate : ", n7)
print("KNN (N = 8) Error Rate : ", n8)
print("KNN (N = 9) Error Rate : ", n9)
print("KNN (N = 10) Error Rate : ", n10)
```

```
Logistic Error Rate :  0.27428571428571424
LDA Error Rate :  0.2828571428571428
QDA Error Rate :  0.2857142857142857
Naive Byes Error Rate :  0.2885714285714286
KNN (N = 1) Error Rate :  0.2857142857142857
KNN (N = 2) Error Rate :  0.36571428571428577
KNN (N = 3) Error Rate :  0.30000000000000004
KNN (N = 4) Error Rate :  0.31999999999999995
KNN (N = 5) Error Rate :  0.31999999999999995
KNN (N = 6) Error Rate :  0.3314285714285714
KNN (N = 7) Error Rate :  0.31714285714285717
KNN (N = 8) Error Rate :  0.3371428571428572
KNN (N = 9) Error Rate :  0.30000000000000004
KNN (N = 10) Error Rate :  0.31714285714285717
```

In [53]:

```
plt.figure(figsize=(10,10))
probs = logis.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
logis_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'b', label = 'Logis AUC = %0.2f' % logis_roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')

probs = lda.predict_proba(x_test)
preds = probs[:,1]
```

```
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
lda_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'g', label = 'LDA AUC = %0.2f' % lda_roc_auc)
plt.legend(loc = 'lower right')

probs = qda.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
qda_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'orange', label = 'QDA AUC = %0.2f' % qda_roc_auc)
plt.legend(loc = 'lower right')

probs = gnb.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
gnb_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'purple', label = 'GNB AUC = %0.2f' % gnb_roc_auc)
plt.legend(loc = 'lower right')

probs = gnb.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
gnb_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'purple', label = 'KNN1 AUC = %0.2f' % gnb_roc_auc)
plt.legend(loc = 'lower right')

probs = nbrs1.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
nbrs1_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'black', label = 'KNN2 AUC = %0.2f' % nbrs1_roc_auc)
plt.legend(loc = 'lower right')

probs = nbrs2.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
nbrs2_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'grey', label = 'KNN2 AUC = %0.2f' % nbrs2_roc_auc)
plt.legend(loc = 'lower right')

probs = nbrs3.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
nbrs3_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'pink', label = 'KNN3 AUC = %0.2f' % nbrs3_roc_auc)
plt.legend(loc = 'lower right')

probs = nbrs4.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
nbrs4_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'cyan', label = 'KNN4 AUC = %0.2f' % nbrs4_roc_auc)
plt.legend(loc = 'lower right')

probs = nbrs5.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
nbrs5_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'magenta', label = 'KNN5 AUC = %0.2f' % nbrs5_roc_auc)
plt.legend(loc = 'lower right')

probs = nbrs6.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
nbrs6_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'tan', label = 'KNN6 AUC = %0.2f' % nbrs6_roc_auc)
plt.legend(loc = 'lower right')

probs = nbrs7.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
nbrs7_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'lime', label = 'KNN7 AUC = %0.2f' % nbrs7_roc_auc)
plt.legend(loc = 'lower right')

probs = nbrs8.predict_proba(x_test)
preds = probs[:,1]
```

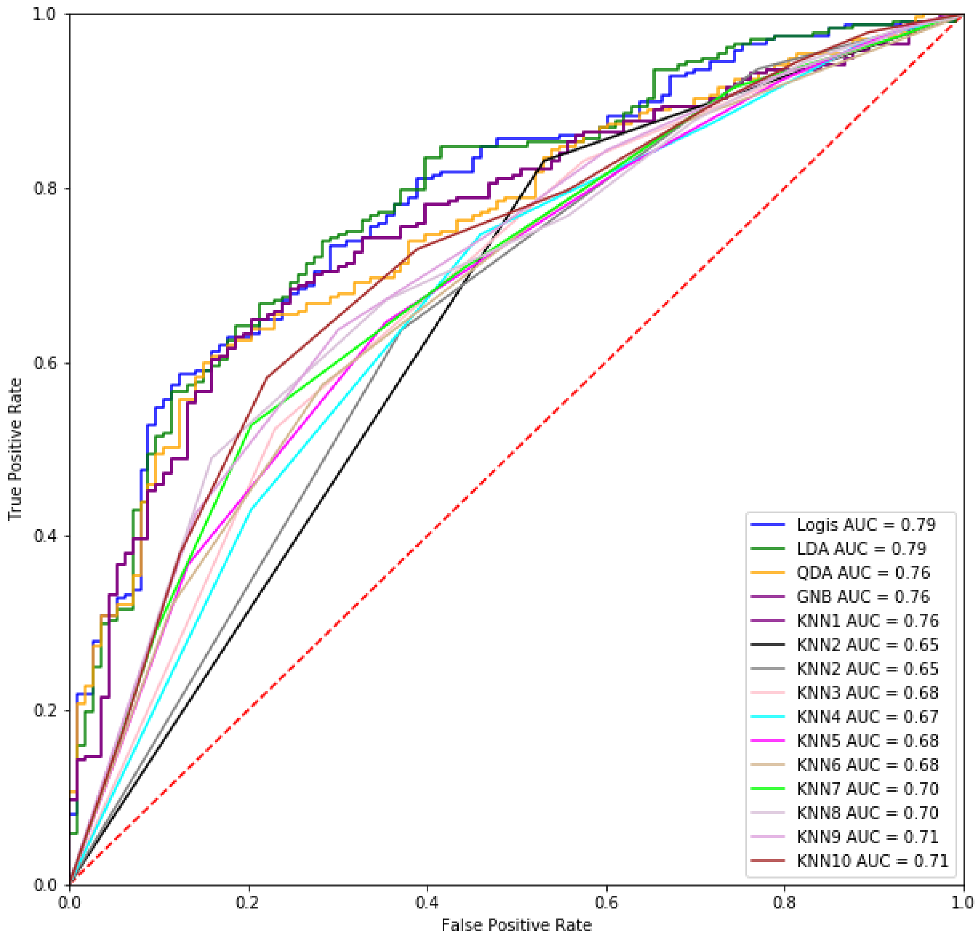
```
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
nbrs8_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'thistle', label = 'KNN8 AUC = %0.2f' % nbrs8_roc_auc)
plt.legend(loc = 'lower right')

probs = nbrs9.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
nbrs9_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'plum', label = 'KNN9 AUC = %0.2f' % nbrs9_roc_auc)
plt.legend(loc = 'lower right')

probs = nbrs10.predict_proba(x_test)
preds = probs[:,1]
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
nbrs10_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'brown', label = 'KNN10 AUC = %0.2f' % nbrs10_roc_auc)
plt.legend(loc = 'lower right')
```

Out[53]:

```
<matplotlib.legend.Legend at 0x1a56097ea90>
```



Which model performs the best? Be sure to define what you mean by “best” and identify supporting evidence to support your conclusion(s).

As it shown in the result/ graph printed above. In terms of the error rate, Logistic and LDA perform the best. QDA, GNB and KNN (N = 1) perform relatively good. In terms of the ROC/AUC, the best model would be Logistic and LDA, both of them have a relatively high AUC. QDA, GNB and KNN (N = 1) perform relatively good.
Therefore, I would say Logidtic and LDA would be my best models in terms of both accuracy and AUC/ROC.