In [552]:

```
import sklearn
import sklearn.naive_bayes
import sklearn.tree
import sklearn.ensemble
import sklearn.neural_network
import sklearn.decomposition
from sklearn import datasets, linear_model
from sklearn.model_selection import train_test_split

import nltk
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors
import seaborn
import scipy as sp

import collections
import os
import os.path
import random
import re
import glob
import pandas
import requests
import json
import math

%matplotlib inline
```

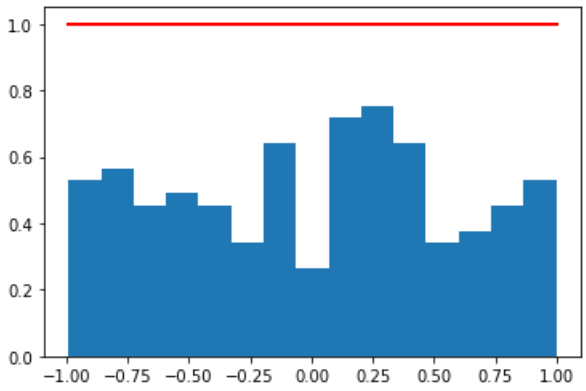
Bayes Classifier

1) (a-h)

In [553]:

```
#numpy random.uniform, put in low high, minus 1 or 1, and then sample size
np.random.seed(0)

#Generating values for X1
X1 = np.random.uniform(-1,1,200)
count, bins, ignored = plt.hist(X1, 15, normed=True)
plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
plt.show()
```

```
/Users/Sruti/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:6:
MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.1. Use 'density' inst
ead.
```

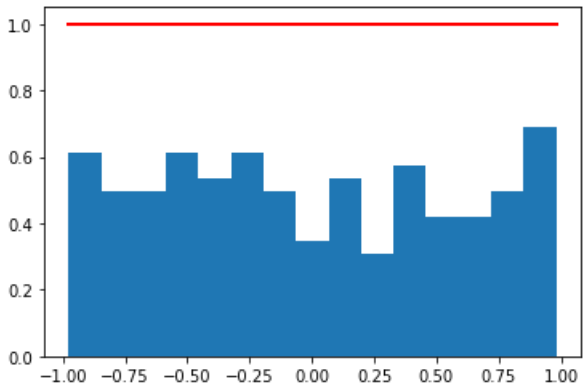


In [554]:

In [594]:

```
#Generating values for X2
X2 = np.random.uniform(-1,1,200)
count, bins, ignored = plt.hist(X2, 15, normed=True)
plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
plt.show()
```

```
/Users/Sruti/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.1. Use 'density' inst
ead.
  This is separate from the ipykernel package so we can avoid doing imports until
```



In [600]:

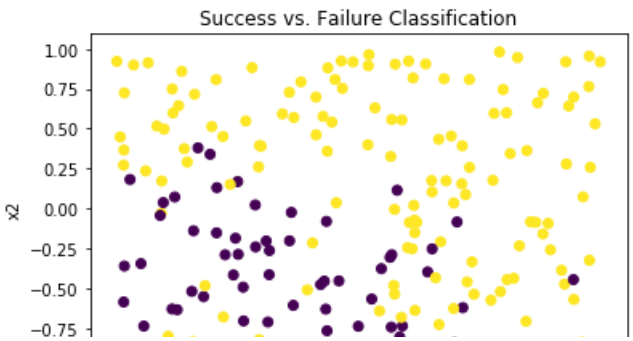
```
#Generating error term values
epsilon = np.random.normal(0, .5, 200) #variance .25, sd = .5

Y = X1 + (X1**2) + X2 + (X2**2)  + epsilon
```

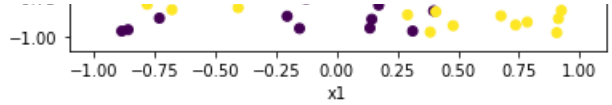
In [601]:

```
#Output = logodds, converting to probabilities
Y_prob = np.exp(Y)/(1+np.exp(Y))
Y_prob

#Using probabilites to classify (0.5 threshold)
Y_input = np.where(Y_prob > 0.5, 1, 0)
```

In [619]:

```
#Scatterplot of X1, X2 before overlaying Bayes decision boundary
plt.scatter(X1, X2, c = Y_input)
plt.title("Success vs. Failure Classification")
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()

#Had some problems plotting legend- manually generated:
plot_legend = [["Success", " Yellow"], ["Failure", "Purple"]]
plot_legend = pandas.DataFrame(plot_legend)
plot_legend.rename(columns={0:'Type', 1: 'Color'}, inplace=True)
plot_legend
```





Out[619]:

	Type	Color
0	Success	Yellow
1	Failure	Purple

In [ ]:

```
#Creating mesh for boundary
xx, yy = np.meshgrid(np.linspace(-1,1,100), np.linspace(-1,1,100))
Z = gnb.predict_proba(np.c_[xx.ravel(), yy.ravel()]) #flattening to reduce 3 dim to 2
Z = Z[:,1].reshape(xx.shape)
```

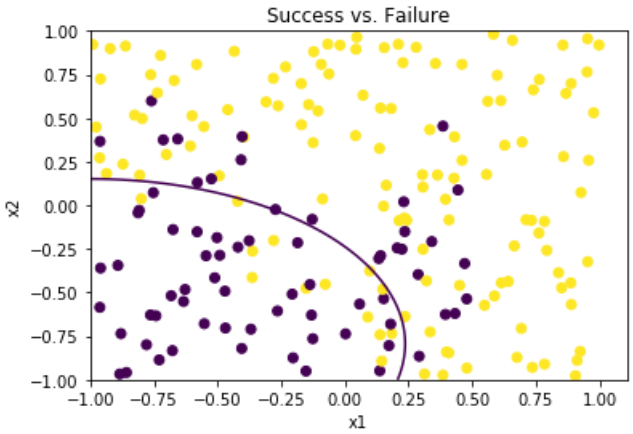
In [355]:

```
x1_x2 = np.array([X1, X2])
x1_x2 = x1_x2.transpose()
```

In [357]:

```
gnb = GaussianNB()
gnb.fit(x1_x2, Y_input)
xthing = np.array([X1_m.ravel(), X2_m.ravel()]).transpose()
print(xthing.shape)
y_pred = gnb.predict_proba(xthing)

#gnb = GaussianNB()
#y_pred = gnb.fit(X_train, y_train).predict_proba(X_test)
#y_pred

plt.scatter(X1, X2, c = Y_input)
plt.title("Success vs. Failure Boundary Classification")
plt.xlabel('x1')
plt.ylabel('x2')
plt.contour(xx, yy, Z, [0.5])
plt.show()
```

```
(4000000, 2)
```



Exploring Simulated Differences between LDA and QDA

2 (a & b)

Given its higher flexiblity, we would expect the QDA model to perform better on the training set- however, this flexibility might also

lead to overfitting. Therefore, we would expect the LDA model to perform better, given that it assumes linear dependence and not quadratic dependence.

b) However, the respective LDA and QDA error rates are almost the same (note similarity in variance in the boxplots below; mean error rate of LDA train = 0.474461, of LDA test = 0.499200, of QDA train = 0.465729,and of QDA test = 0.499400), indicating that these models perform roughly as well as each other. One potential reason for this is that QDA assumes both linear and quadratic dependency.

In [386]:

```
X1 = np.random.uniform(-1,1,1000)
X2 = np.random.uniform(-1,1,1000)
dataset = np.array([X1, X2])

epsilon = np.random.normal(0, 1, 1000)
Y_sim = X1 + X2 + epsilon
Y_sim

Y_sim_contain = Y_sim >=0
Y_sim_contain


X_split = np.column_stack((X1, X2))
```

In [289]:

```
# split X and y into training and testing sets
X_train_lda,X_test_lda,y_train_lda,y_test_lda=train_test_split(X_split,Y_sim_contain,test_size=0.3
,random_state=0)
```

In [290]:

```
lda_analysis = LDA()
lda.fit(X_train_lda, y_train_lda)
#LDA fit
```

Out[290]:

```
LinearDiscriminantAnalysis(n_components=None, priors=None, shrinkage=None,
                           solver='svd', store_covariance=False, tol=0.0001)
```

In [291]:

```
qda_analysis = QDA()
qda.fit(X_train_lda, y_train_lda)
#QDA fit
```

Out[291]:

```
QuadraticDiscriminantAnalysis(priors=None, reg_param=0.0,
                              store_covariance=False, tol=0.0001)
```

In [308]:

```
qda_lda_errors = [["LDA Training Error Rate", 1-lda.score(X_train_lda, y_train_lda)],
["LDA Test Error Rate", 1-lda.score(X_test_lda, y_test_lda)],
["QDA Training Error Rate", 1-qda.score(X_train_lda, y_train_lda)],
["QDA Test Error Rate", 1-qda.score(X_test_lda, y_test_lda)]]

errors_table = pandas.DataFrame(qda_lda_errors)
errors_table
```

Out[308]:

	0	1
0	LDA Training Error Rate	0.291429
1	LDA Test Error Rate	0.236667
2	QDA Training Error Rate	0.298571

3	QDA Test Error Rate	0.236667

In [ ]:

```
#Note: Talked with Selina about how to approach this question, but we generated
#our functions separately and understand what they are doing/how they work.

#Repeating x1000
error_lda_train = []
error_lda_test = []
error_qda_train = []
error_qda_test = []
for i in range(1000):
    X1 = np.random.uniform(-1,1,1000)
    X2 = np.random.uniform(-1,1,1000)
    dataset = np.array([X1, X2])
    epsilon = np.random.normal(0, 1, 1000)
    Y_sim = X1 + X2 + epsilon
    Y_sim_contain = Y_sim >=0
    X_train_lda,X_test_lda,y_train_lda,y_test_lda=train_test_split(X_split,Y_sim_contain,test_size=
0.3,random_state=0)
    lda_analysis = LDA()
    lda.fit(X_train_lda, y_train_lda)
    qda_analysis = QDA()
    qda.fit(X_train_lda, y_train_lda)
    error_lda_train.append(1-lda.score(X_train_lda, y_train_lda))
    error_lda_test.append(1-lda.score(X_test_lda, y_test_lda))
    error_qda_train.append(1-qda.score(X_train_lda, y_train_lda))
    error_qda_test.append(1-qda.score(X_test_lda, y_test_lda))
```

In [315]:

```
qda_lda_errors_1000 = [error_lda_train, error_lda_test, error_qda_train, error_qda_test]
```

In [375]:

```
#Putting results in dataframe
qda_lda_errors_1000 = pandas.DataFrame(qda_lda_errors_1000)
qda_lda_errors_1000.rename(index={0:'LDA Training Error',1:'LDA Test Error', 2: 'QDA Training
Error', 3: 'QDA Test Error'}, inplace=True)
```

In [381]:

```
qda_lda_errors_1000.rename(columns={'Error rate':'0'}, inplace=True)
qda_lda_errors_1000 = qda_lda_errors_1000.transpose()
qda_lda_errors_1000
#The rows correspond to each of the 1000 iterations
#Table shows training error output
```

Out[381]:

	LDA Training Error	LDA Test Error	QDA Training Error	QDA Test Error
0	0.457143	0.466667	0.460000	0.463333
1	0.451429	0.523333	0.434286	0.513333
2	0.457143	0.496667	0.442857	0.526667
3	0.492857	0.473333	0.472857	0.473333
4	0.491429	0.493333	0.465714	0.526667
...	...	...	...	...
995	0.481429	0.506667	0.468571	0.506667
996	0.447143	0.510000	0.462857	0.516667
997	0.492857	0.536667	0.468571	0.510000
998	0.474286	0.530000	0.474286	0.530000
999	0.491429	0.473333	0.474286	0.463333

1000 rows × 4 columns

In [382]:

```
qda_lda_errors_1000.describe()
```

Out[382]:

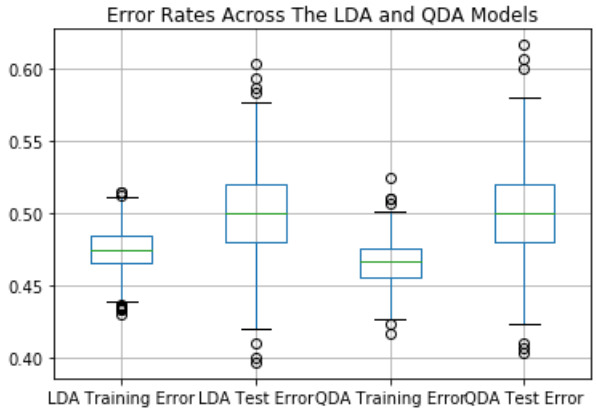
	LDA Training Error	LDA Test Error	QDA Training Error	QDA Test Error
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.474461	0.499200	0.465729	0.499400
std	0.014679	0.029266	0.014145	0.029531
min	0.430000	0.396667	0.417143	0.403333
25%	0.465714	0.480000	0.455714	0.480000
50%	0.474286	0.500000	0.467143	0.500000
75%	0.484286	0.520000	0.475714	0.520000
max	0.514286	0.603333	0.524286	0.616667

In [383]:

```
qda_lda_errors_1000.boxplot()
plt.title("Error Rates Across The LDA and QDA Models")
```

Out[383]:

```
Text(0.5, 1.0, 'Error Rates Across The LDA and QDA Models')
```



3 (a & b)

If the decision boundary is non-linear, we would expect QDA to perform better (in both the training and test) because LDA would not capture the quadratic dependence in the data. In other words, LDA is too inflexible to adequately model the relationship. However, the output table and boxplot I've generated below show negligible differences in mean error rate across QDA and LDA in both training and test, which is a surprising result.

In [575]:

```
X1_3 = np.random.uniform(-1,1,1000)
X2_3 = np.random.uniform(-1,1,1000)
dataset_3 = np.array([X1, X2])


epsilon_3 = np.random.normal(0, 1, 1000)
Y_sim_3 = X1_3 + (X1_3**2) + X2_3 + (X2_3**2) + epsilon_3
Y_sim_3

Y_sim_contain_3 = Y_sim_3 >=0
Y_sim_contain_3
```

```
X_split_3 = np.column_stack((X1_3, X2_3))
```

In [576]:

```
#Do 1000 times
error_lda_train_3 = []
error_lda_test_3 = []
error_qda_train_3 = []
error_qda_test_3 = []
for i in range(1000):
    X1_3 = np.random.uniform(-1,1,1000)
    X2_3 = np.random.uniform(-1,1,1000)
    dataset_3 = np.array([X1_3, X2_3])
    epsilon_3 = np.random.normal(0, 1, 1000)
    Y_sim_3 = X1_3 + (X1_3**2) + X2_3 + (X2_3**2) + epsilon_3
    Y_sim_contain_3 = Y_sim_3 >=0
    X_train_3,X_test_3,y_train_3,y_test_3=train_test_split(X_split_3,Y_sim_contain_3,test_size=0.3
,random_state=0)
    lda_analysis_3 = LDA()
    lda.fit(X_train_3, y_train_3)
    qda_analysis_3 = QDA()
    qda.fit(X_train_3, y_train_3)
    error_lda_train_3.append(1-lda.score(X_train_3, y_train_3))
    error_lda_test_3.append(1-lda.score(X_test_3, y_test_3))
    error_qda_train_3.append(1-qda.score(X_train_3, y_train_3))
    error_qda_test_3.append(1-qda.score(X_test_3, y_test_3))
```

In [577]:

```
qd_errors_1000 = [error_lda_train_3, error_lda_test_3, error_qda_train_3, error_qda_test_3]
```

In [578]:

```
qd_errors_1000 = pandas.DataFrame(qd_errors_1000)
qd_errors_1000.rename(index={0:'LDA Training Error',1:'LDA Test Error', 2: 'QDA Training Error', 3:
'QDA Test Error'}, inplace=True)

qd_errors_1000.rename(columns={'Error rate':'0'}, inplace=True)
qd_errors_1000 = qd_errors_1000.transpose()
#The rows correspond to each of the 1000 iterations
```

In [579]:

```
qd_errors_1000
```

Out[579]:

	LDA Training Error	LDA Test Error	QDA Training Error	QDA Test Error
0	0.342857	0.270000	0.342857	0.270000
1	0.337143	0.343333	0.337143	0.343333
2	0.324286	0.330000	0.324286	0.330000
3	0.362857	0.323333	0.361429	0.343333
4	0.345714	0.260000	0.345714	0.260000
...	...	...	...	...
995	0.307143	0.356667	0.307143	0.356667
996	0.327143	0.303333	0.327143	0.303333
997	0.324286	0.293333	0.324286	0.293333
998	0.338571	0.393333	0.338571	0.393333
999	0.327143	0.353333	0.327143	0.353333

1000 rows × 4 columns

In [581]:

```
qd_errors_1000.describe()
```

Out[581]:

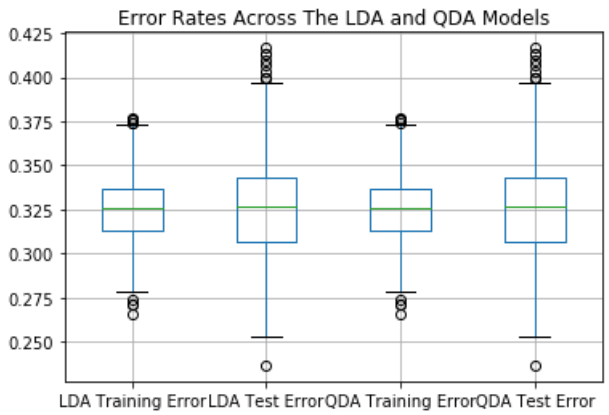
	LDA Training Error	LDA Test Error	QDA Training Error	QDA Test Error
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.325436	0.325780	0.325419	0.325857
std	0.017846	0.026824	0.017831	0.026871
min	0.265714	0.236667	0.265714	0.236667
25%	0.312857	0.306667	0.312857	0.306667
50%	0.325714	0.326667	0.325714	0.326667
75%	0.337143	0.343333	0.337143	0.343333
max	0.377143	0.416667	0.377143	0.416667

In [582]:

```
qd_errors_1000.boxplot()
plt.title("Error Rates Across The LDA and QDA Models")
```

Out[582]:

```
Text(0.5, 1.0, 'Error Rates Across The LDA and QDA Models')
```



4 (a & b)

As the sample size n increases, we should expect the error rate for LDA and QDA (for both test and training) to reduce. However, given that we are working with a non-linear decision-boundary in this case, QDA should fit better than LDA. As anticipated, QDA had the lowest error rate values (they decreased further as n grew larger). If the boxplot is unclear: the plots correspond to lda train, lda test, qda train, and qda test from left to right.

While I was unable to run n = 100,000 for this assignment, I think that the error rate would have been the lowest in this case (particularly so for the QDA test and training models).

In [626]:

```
#Do 1000 times for n= 100, 1000, 10000, and 100000 for both QDA and LDA
def for_n(n):
    error_lda_train_n = []
    error_lda_test_n = []
    error_qda_train_n = []
    error_qda_test_n = []
    for i in range(n):
        X1_n = np.random.uniform(-1,1, n)
        X2_n = np.random.uniform(-1,1, n)
        X_split_n = np.column_stack((X1_n, X2_n))
        dataset_n = np.array([X1_n, X2_n])
        epsilon_n = np.random.normal(0, 1, n)
        Y_sim_n = X1_n + (X1_n**2) + X2_n + (X2_n**2) + epsilon_n
        Y_sim_contain_n = Y_sim_n >=0
```

```
        X_train_n,X_test_n,y_train_n,y_test_n=train_test_split(X_split_n,Y_sim_contain_n,test_size=
0.3,random_state=0)
        lda_analysis_n = LDA()
        lda.fit(X_train_n, y_train_n)
        qda_analysis_n = QDA()
        qda.fit(X_train_n, y_train_n)
        error_lda_train_n.append(1-lda.score(X_train_n, y_train_n))
        error_lda_test_n.append(1-lda.score(X_test_n, y_test_n))
        error_qda_train_n.append(1-qda.score(X_train_n, y_train_n))
        error_qda_test_n.append(1-qda.score(X_test_n, y_test_n))
    return error_lda_train_n, error_lda_test_n, error_qda_train_n, error_qda_test_n

#n = 100:
error_lda_train_100, error_lda_test_100, error_qda_train_100, error_qda_test_100 = for_n(100)
```

In [630]:

```
#n = 10,000
error_lda_train_10000, error_lda_test_10000, error_qda_train_10000, error_qda_test_10000 =
for_n(10000)
```

In [628]:

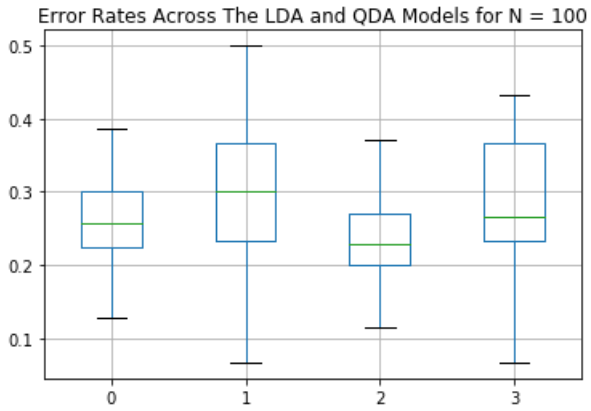
```
#n = 1,000
error_lda_train_1000, error_lda_test_1000, error_qda_train_1000, error_qda_test_1000 = for_n(1000)
```

In [627]:

```
errors_n_100 = [error_lda_train_100, error_lda_test_100, error_qda_train_100, error_qda_test_100]
errors_n_100 = pandas.DataFrame(errors_n_100)
errors_n_100 = errors_n_100.transpose()
errors_n_100.boxplot()
plt.title("Error Rates Across The LDA and QDA Models for N = 100")
```

Out[627]:

```
Text(0.5, 1.0, 'Error Rates Across The LDA and QDA Models for N = 100')
```

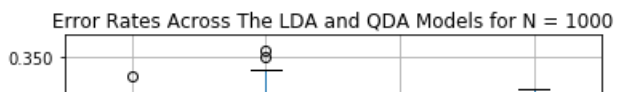


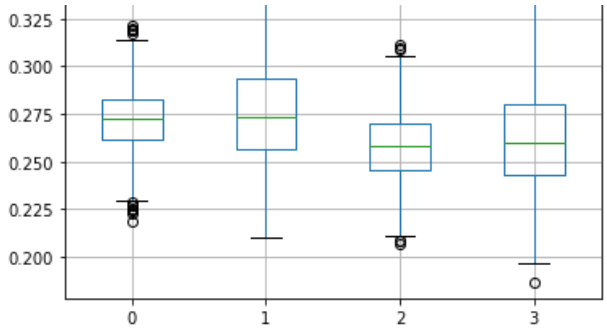
In [629]:

```
errors_n_1000 = [error_lda_train_1000, error_lda_test_1000, error_qda_train_1000,
error_qda_test_1000]
errors_n_1000 = pandas.DataFrame(errors_n_1000)
errors_n_1000 = errors_n_1000.transpose()
errors_n_1000.boxplot()
plt.title("Error Rates Across The LDA and QDA Models for N = 1000")
```

Out[629]:

```
Text(0.5, 1.0, 'Error Rates Across The LDA and QDA Models for N = 1000')
```





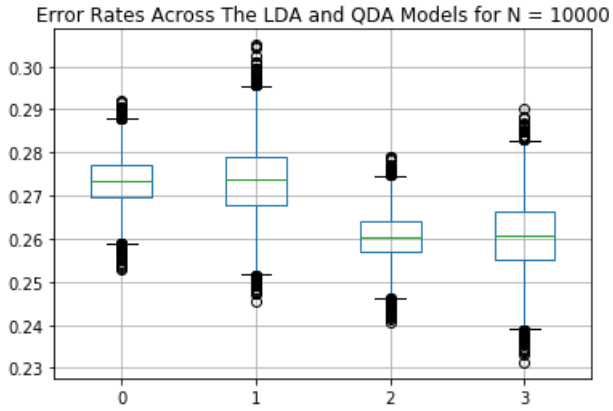
I tried to run n = 100,000, but it never finished running.

In [631]:

```
errors_n_10000 = [error_lda_train_10000, error_lda_test_10000, error_qda_train_10000, error_qda_tes
t_10000]
errors_n_10000 = pandas.DataFrame(errors_n_10000)
errors_n_10000 = errors_n_10000.transpose()
errors_n_10000.boxplot()
plt.title("Error Rates Across The LDA and QDA Models for N = 10000")
```

Out[631]:

```
Text(0.5, 1.0, 'Error Rates Across The LDA and QDA Models for N = 10000')
```



Modeling Voter Turnout

5 (a-d)

Logistic Regression

In [516]:

```
mental_health = pandas.read_csv('/Users/Sruti/Desktop/MACS30100/week_4/mental_health.csv',
index_col=False)
```

In [517]:

```
mental_health
```

Out[517]:

	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

	vote96	mhealth_sum	age	educ	black	female	married	inc10
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
...	...	...	...	...	...	...	...	...
2827	1.0	NaN	73.0	14.0	0	1	1.0	2.2737
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
2831	NaN	6.0	20.0	10.0	0	0	0.0	4.0124

2832 rows × 8 columns

In [518]:

```
mental_health = mental_health.dropna()
```

In [519]:

```
# create training and testing vars
feature_cols = ['mhealth_sum', 'age', 'educ', 'black', 'female', 'married', 'inc10']
X = mental_health[feature_cols] # Features
y = mental_health.vote96 # Target variable
```

In [520]:

```
# split X and y into training and testing sets
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3,random_state=0)
```

In [521]:

```
# instantiate the model (using the default parameters)
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression()

# fit the model with data
logreg.fit(X_train,y_train)

y_pred = logreg.predict(X_test)
```

```
/Users/Sruti/opt/anaconda3/lib/python3.7/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 [522]:

```
logreg_error = 1-logreg.score(X_test, y_test)
logreg_error
```

Out[522]:

```
0.3057142857142857
```

In [523]:

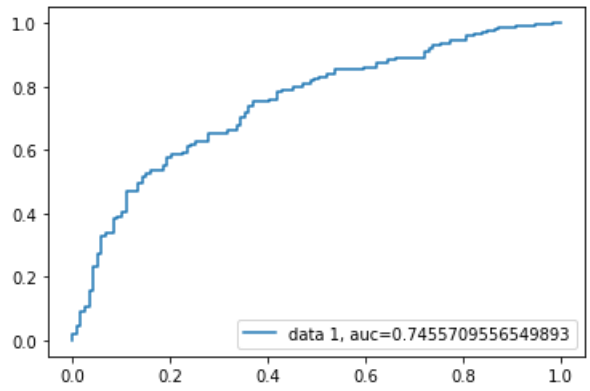
```
from sklearn import metrics
cnf_matrix = metrics.confusion_matrix(y_test, y_pred)
cnf_matrix
```

Out[523]:

```
array([[ 38,  81],
       [ 26, 205]])
```

In [524]:

```
y_pred_proba = logreg.predict_proba(X_test)[::,1]
fpr, tpr, _ = metrics.roc_curve(y_test,  y_pred_proba)
auc = metrics.roc_auc_score(y_test, y_pred_proba)
plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
plt.legend(loc=4)
plt.show()
```



In [525]:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis,
QuadraticDiscriminantAnalysis
from sklearn.metrics import confusion_matrix, classification_report, precision_score
```

Linear Discriminant Model

In [527]:

```
# Splitting the dataset into the Training set and Test set
X_train_lda, X_test_lda, y_train_lda, y_test_lda = train_test_split(X, y, test_size = 0.3)


# Implement LDA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
lda = LDA()
X_train_lda = lda.fit_transform(X_train_lda, y_train_lda)
y_pred = lda.transform(X_test_lda)

#Generated logodds, converting to probabilities
y_prob = np.exp(y_pred)/(1+np.exp(y_pred))
y_prob

#Using probabilites to classify (0.5 threshold)
y_input = np.where(y_prob > 0.5, 1, 0)
```

In [528]:

```
lda_error = 1-lda.score(X_test_lda, y_test_lda)
lda_error
```

Out[528]:

```
0.26857142857142857
```

In [529]:

```
cnf_matrix = metrics.confusion_matrix(y_test_lda, y_input)
cnf_matrix
```

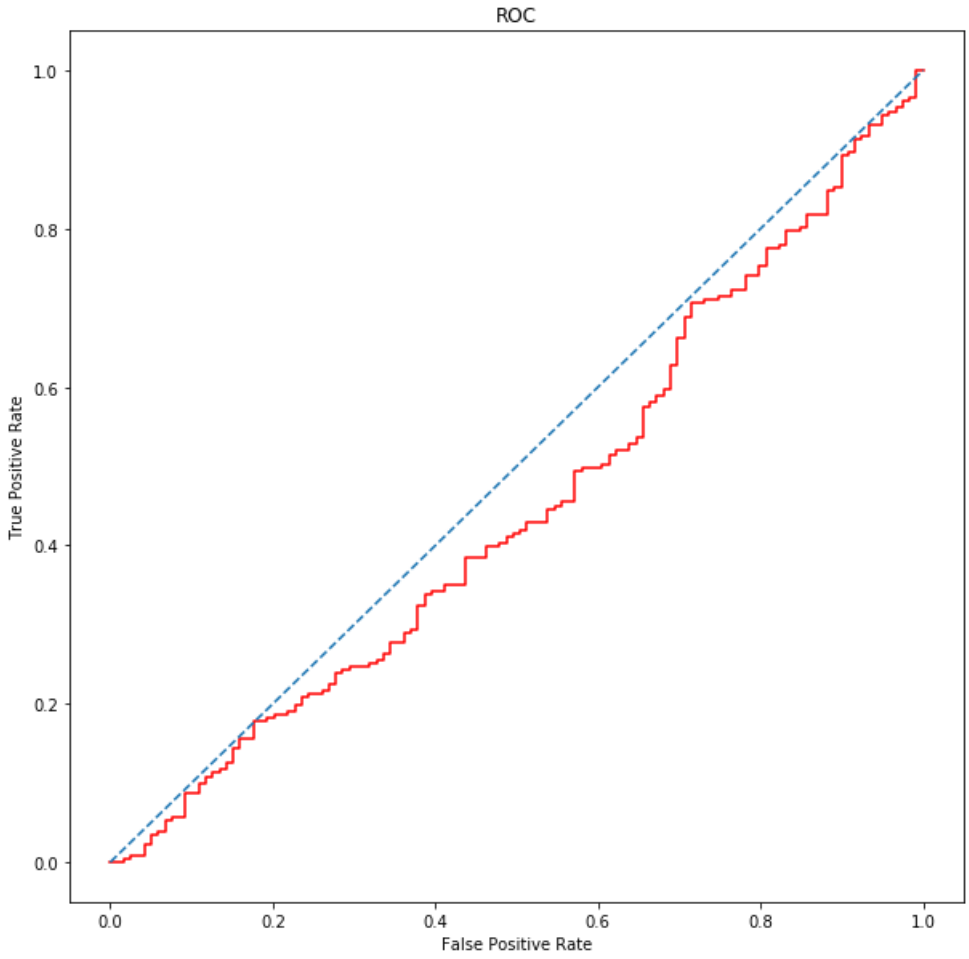
Out[529]:

```
array([[ 79,  28],
       [ 90, 153]])
```

In [530]:

```
auc_roc = metrics.roc_auc_score(y_test, y_input)
print(auc_roc)

from sklearn.metrics import roc_curve, auc

false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(false_positive_rate, true_positive_rate)

roc_auc = auc(false_positive_rate, true_positive_rate)

def plot_roc(roc_auc):
    plt.figure(figsize = (10,10))
    plt.title("ROC")
    plt.plot(false_positive_rate, true_positive_rate, color='red', label='AUC = %0.2f' % roc_auc)
    plt.plot([0,1], [0,1], linestyle= '--')
    plt.axis('tight')
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')

plot_roc(roc_auc)
#Not sure why a couple of these ROC plots are graphing backwards
```

```
0.4588744588744589
```



Quadratic Discriminant Analysis

In [531]:

```
from sklearn import discriminant_analysis
```

In [532]:

```
# Splitting the dataset into the Training set and Test set
```

```
# Splitting the dataset into the Training set and Test set
X_train_qda, X_test_qda, y_train_qda, y_test_qda = train_test_split(X, y, test_size = 0.3)

# Implement LDA
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QDA
qda = QDA()
qda.fit(X_train_qda, y_train_qda)
y_pred = qda.predict_proba(X_test_qda)

#Generated logodds, converting to probabilities
y_pred

#Using probabilites to classify (0.5 threshold)
y_input = np.where(y_pred > 0.5, 1, 0)
y_input = y_input[:,1]
```

In [533]:

```
qda_error = 1-qda.score(X_test_qda, y_test_qda)
qda_error
```

Out[533]:

```
0.28
```

In [534]:

```
cnf_matrix = metrics.confusion_matrix(y_test_qda, y_input)
cnf_matrix
```

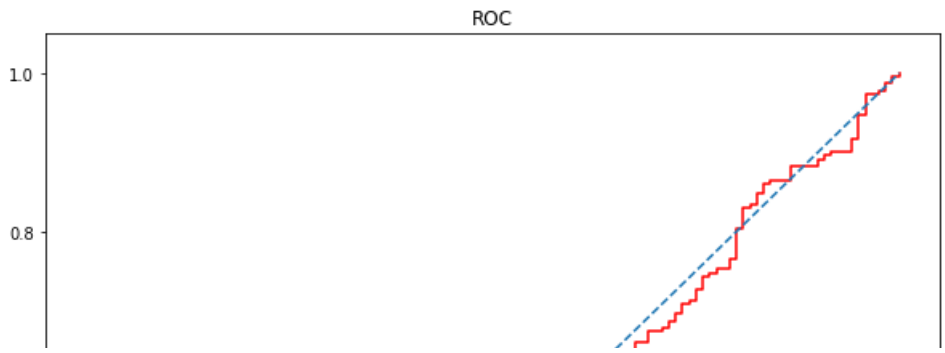
Out[534]:

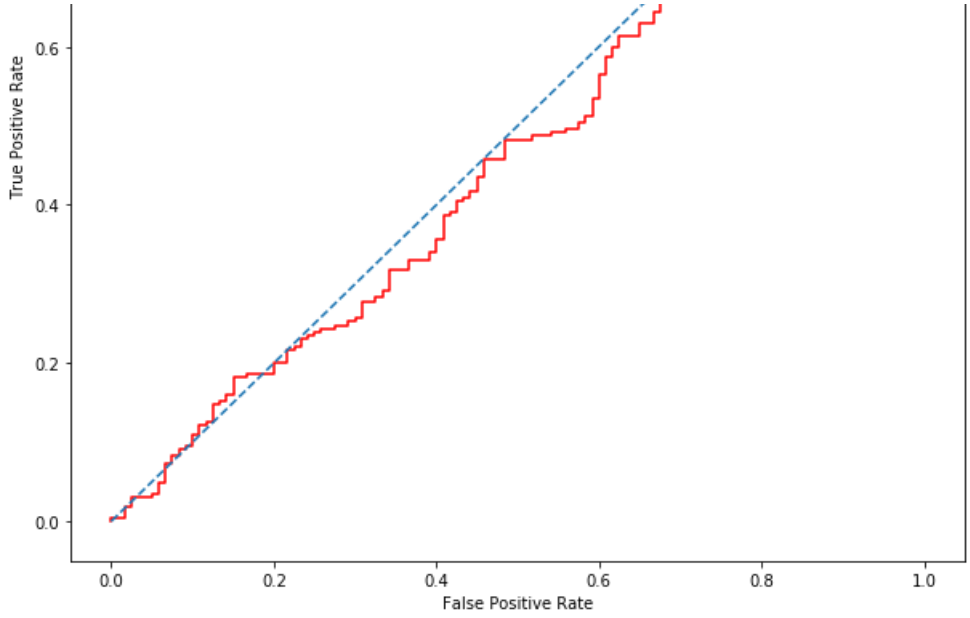
```
array([[ 60,  60],
       [ 38, 192]])
```

In [612]:

```
auc_roc = metrics.roc_auc_score(y_test_qda, y_input)
print(auc_roc)

from sklearn.metrics import roc_curve, auc

false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test_qda, y_prob)
roc_auc = auc(false_positive_rate, true_positive_rate)

roc_auc = auc(false_positive_rate, true_positive_rate)

def plot_roc(roc_auc):
    plt.figure(figsize = (10,10))
    plt.title("ROC")
    plt.plot(false_positive_rate, true_positive_rate, color='red', label='AUC = %0.2f' % roc_auc)
    plt.plot([0,1], [0,1], linestyle= '--')
    plt.axis('tight')
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')

plot_roc(roc_auc)
```

```
0.5137681159420291
```





Naive Bayes

In [536]:

```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
gnb = GaussianNB()
y_pred = gnb.fit(X_train, y_train).predict_proba(X_test)
y_pred


#Using probabilites to classify (0.5 threshold)
y_pred

#Using probabilites to classify (0.5 threshold)
y_input = np.where(y_pred > 0.5, 1, 0)
y_input = y_input[:,1]
y_input
#need to vary the threshold to generate the error curves, how to do that?
```

Out[536]:

```
array([1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
       0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1,
       1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,
       0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0,
       1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,
       1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0,
       0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1,
       1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0,
       1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0])
```

In [537]:

```
naive_bayes_error = 1-gnb.score(X_test, y_test)
naive_bayes_error
```

Out[537]:

```
0.3085714285714286
```

In [538]:

```
cnf_matrix = metrics.confusion_matrix(y_test, y_input)
cnf_matrix
```

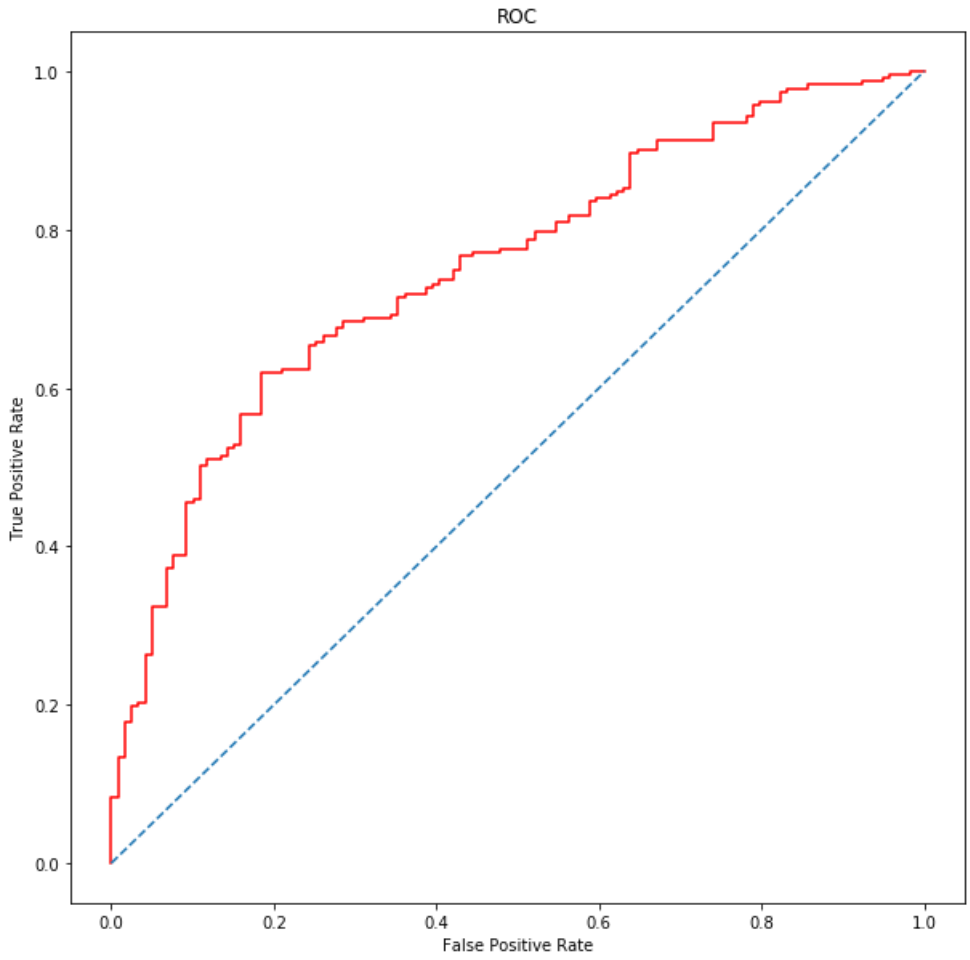
Out[538]:

```
array([[ 49,  70],
       [ 38, 193]])
```

In [539]:

```
auc_roc = metrics.roc_auc_score(y_test, y_input)
print(auc_roc)

from sklearn.metrics import roc_curve, auc

false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_pred[:,1])
roc_auc = auc(false_positive_rate, true_positive_rate)

roc_auc = auc(false_positive_rate, true_positive_rate)

def plot_roc(roc_auc):
    plt.figure(figsize = (10,10))
    plt.title("ROC")
    plt.plot(false_positive_rate, true_positive_rate, color='red', label='AUC = %0.2f' % roc_auc)
    plt.plot([0,1], [0,1], linestyle= '--')
    plt.axis('tight')
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')

plot_roc(roc_auc)
```

```
0.6236312706900942
```



K-Nearest Neighbors

In [540]:

```
from sklearn.neighbors import KNeighborsClassifier

def knn(n):
    neigh = KNeighborsClassifier(n_neighbors= n, metric='euclidean')
    y_pred = neigh.fit(X_train, y_train).predict_proba(X_test)
    y_input = np.where(y_pred > 0.5, 1, 0)
    y_input = y_input[:,1]
    knn_error = 1-neigh.score(X_test, y_test)
    auc_roc = metrics.roc_auc_score(y_test, y_input)
    return y_input, knn_error, auc_roc

n1, n1_error, auc_roc1 = knn(1)
n2, n2_error, auc_roc2 = knn(2)
n3, n3_error, auc_roc3 = knn(3)
n4, n4_error, auc_roc4 = knn(4)
n5, n5_error, auc_roc5 = knn(5)
n6, n6_error, auc_roc6 = knn(6)
n7, n7_error, auc_roc7 = knn(7)
n8, n8_error, auc_roc8 = knn(8)
n9, n9_error, auc_roc9 = knn(9)
n10,n10_error, auc_roc10 = knn(10)
```

In [541]:

```
knn_errors_auc = [["knn_1", n1_error, auc_roc1],
["knn_2", n2_error, auc_roc2],
["knn_3", n3_error, auc_roc3],
["knn_4", n4_error, auc_roc4], ["knn_5", n5_error, auc_roc5],
                  ["knn_6", n6_error, auc_roc6], ["knn_7", n7_error, auc_roc7],
              ["knn_8", n8_error, auc_roc8], ["knn_9", n9_error, auc_roc9],
                   ["knn_10", n10_error, auc_roc10]]

knn_errors_auc = pandas.DataFrame(knn_errors_auc)
knn_errors_auc
```

Out[541]:

	0	1	2
0	knn_1	0.314286	0.637637
1	knn_2	0.402857	0.609244
2	knn_3	0.314286	0.619302
3	knn_4	0.348571	0.623886
4	knn_5	0.331429	0.608352
5	knn_6	0.351429	0.611536
6	knn_7	0.320000	0.614973
7	knn_8	0.320000	0.639419
8	knn_9	0.314286	0.615228
9	knn_10	0.331429	0.616501

In [543]:

```
cnf_matrix = metrics.confusion_matrix(y_test, y_input)
cnf_matrix
```

Out[543]:

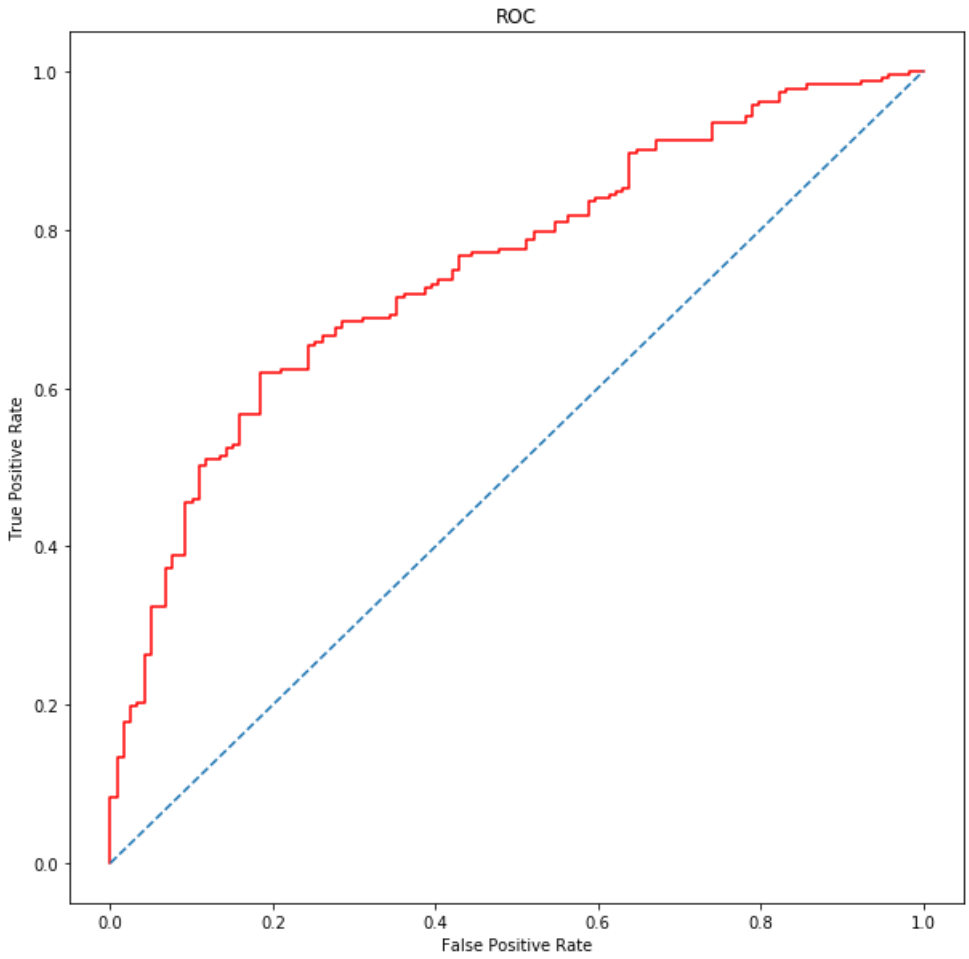
```
array([[ 49,  70],
       [ 38, 193]])
```

In [544]:

```
auc_roc = metrics.roc_auc_score(y_test, y_input)
print(auc_roc)
```

```
print(auc_rf)

from sklearn.metrics import roc_curve, auc

false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_pred[:,1])
roc_auc = auc(false_positive_rate, true_positive_rate)

roc_auc = auc(false_positive_rate, true_positive_rate)

def plot_roc(roc_auc):
    plt.figure(figsize = (10,10))
    plt.title("ROC")
    plt.plot(false_positive_rate, true_positive_rate, color='red', label='AUC = %0.2f' % roc_auc)
    plt.plot([0,1], [0,1], linestyle= '--')
    plt.axis('tight')
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')

plot_roc(roc_auc)
```

```
0.6236312706900942
```



In [545]:

```
#Error rates for all of the models used for Q5
q5_allER = [["knn_1", n1_error],
["knn_2", n2_error],
["knn_3", n3_error],
["knn_4", n4_error], ["knn_5", n5_error], ["knn_6", n6_error], ["knn_7", n7_error],
             ["knn_8", n8_error], ["knn_9", n9_error], ["knn_10", n10_error],
           ["Logistic Regression", logreg_error], ["LDA", lda_error],
           ["Naive Bayes", naive_bayes_error], ["QDA", qda_error]]

q5_allER = pandas.DataFrame(q5_allER)
q5_allER
```

Out[545]:

	0	1
0	knn_1	0.314286
1	knn_2	0.402857
2	knn_3	0.314286
3	knn_4	0.348571
4	knn_5	0.331429
5	knn_6	0.351429
6	knn_7	0.320000
7	knn_8	0.320000
8	knn_9	0.314286
9	knn_10	0.331429
10	Logistic Regression	0.305714
11	LDA	0.268571
12	Naive Bayes	0.308571
13	QDA	0.280000

In terms of ER (above), LDA and QDA performed the best with ER levels around .26 (best here corresponding with lower ER values). In terms of AUC, logistic regression performed best (roughly .7) (best here corresponding with higher ER values).

In [ ]: