HW2 Yijia Lyu lugalyu@uchicago.edu 12245291

In [2]:

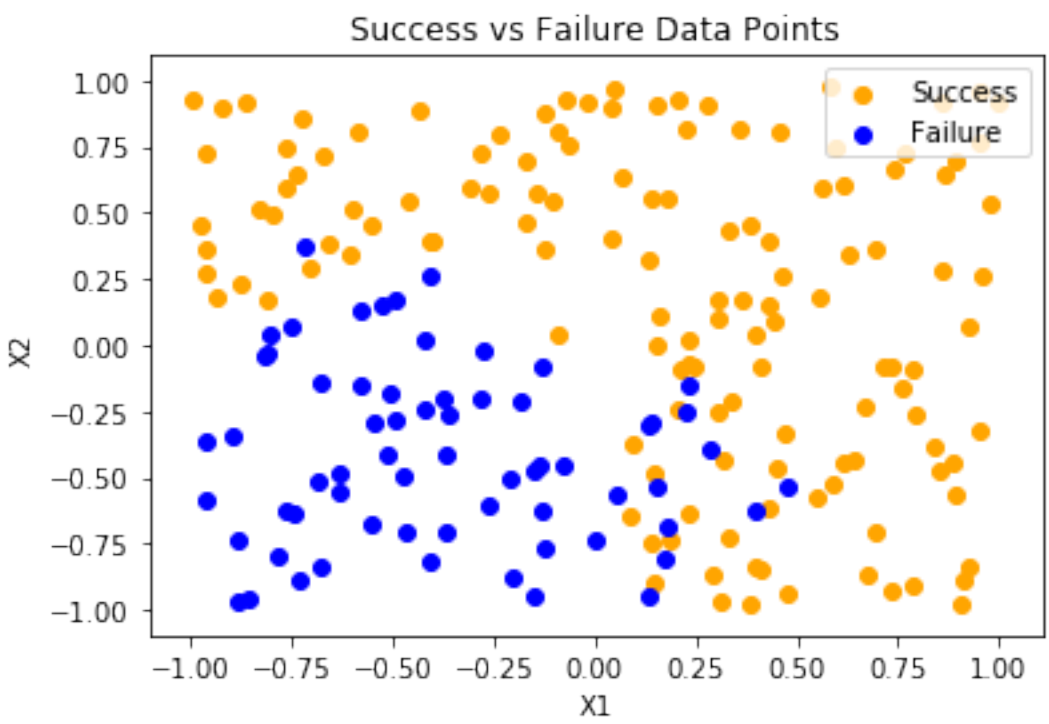
```
import random
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import math

from sklearn.discriminant_analysis import LinearDiscriminantAnal
ysis as LDA
from sklearn.discriminant_analysis import QuadraticDiscriminantA
nalysis as QDA

from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import roc_auc_score, roc_curve, auc
```

The Bayes Classifier

Question1

In [29]:

```
#set random seed
np.random.seed(0)
#simulate dataset
x1 = np.random.uniform(low=-1,high=1,size=200)
x2 = np.random.uniform(low=-1,high=1,size=200)
error = np.random.normal(0,0.25,200)
#calculate y
y = x1+x1**2+x2+x2**2+error
suc_pbb = math.e**y/(1+math.e**y)
label= np.where(suc_pbb>0.5,True,False)
#plot datapoints
plt.scatter(x1[label],x2[label], color='orange')
plt.scatter(x1[~label],x2[~label], color='blue')
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Success vs Failure Data Points')
plt.legend(['Success', 'Failure'], loc=1);
```

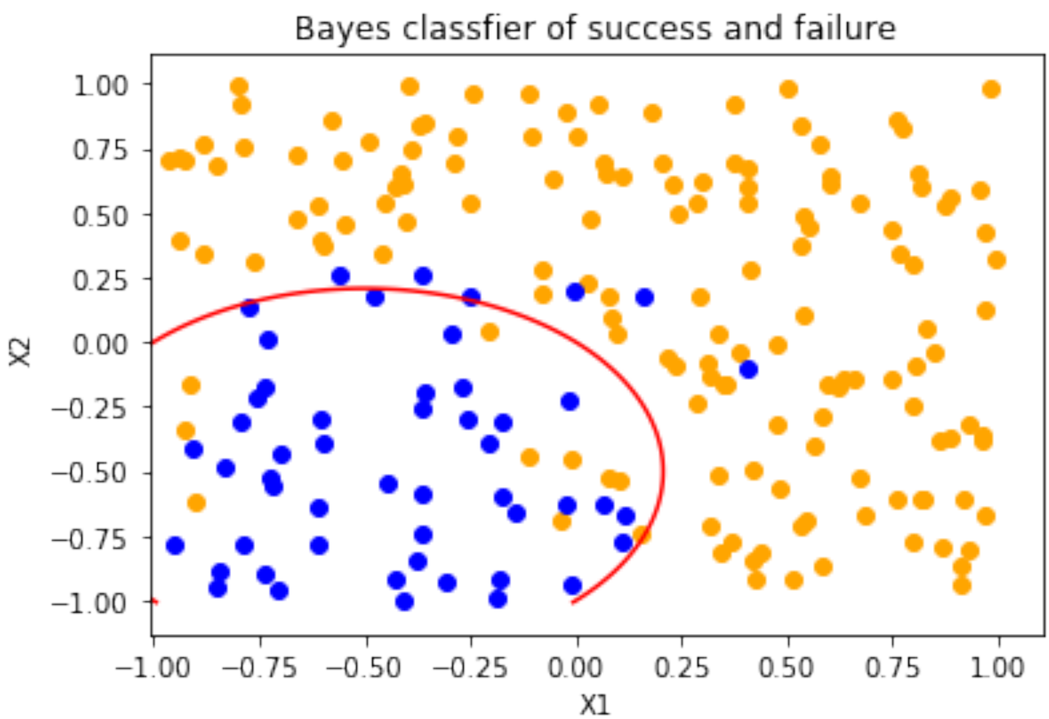


In [31]:

```
#simulate dataset
x1 = np.random.uniform(-1,1,200)
x2 = np.random.uniform(-1,1,200)
eps = np.random.normal(0,0.25,200)
#calculate y
y = x1+x1*x1+x2+x2*x2+eps
Y_exp = np.exp(y)
suc_prob = Y_exp/(1+Y_exp)
label= np.where(suc_prob>0.5,True,False)
#plot datapoints
plt.scatter(x1[label],x2[label], color='orange')
plt.scatter(x1[~label],x2[~label], color='blue')
x1 = np.arange(-1.01, 1.01, 0.01)
x2 = np.arange(-1.01, 1.01, 0.01)
X1, X2 = np.meshgrid(x1, x2)
y = X1 + X1 ** 2 + X2 + X2 ** 2
suc_pbb = math.e**y/(1+math.e**y)
plt.contour(X1, X2, suc_pbb, levels=[0.5], colors='red')
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Bayes classfier of success and failure')
```

Out[31]:

```
Text(0.5, 1.0, 'Bayes classfier of success and failu
re')
```



Exploring Simulated Differences between LDA and QDA

Question2:
QDA has better performance for its higher flexibility on the testing set while LDA performs better on the training set as the Bayes decision boundary is linear and QDA in this case might overfit.

In [4]:

```
lda_train_error = []
qda_train_error = []
lda_test_error = []
qda_test_error = []
for n in range(1000):
    random.seed(n)
    #stimulate dataset
    X1 = np.array([random.uniform(-1,1) for i in range(1000)])
    X2 = np.array([random.uniform(-1,1) for i in range(1000)])
    err = np.array([np.random.normal(loc=0.0, scale=1, size=None
)\
                    for i in range(1000)])
    #simulate y
    y = X1 + X2 + err
    y = y >= 0
    X = np.stack(((X1), (X2)), axis=1)
    train_x, test_x, train_y, test_y = \
    train_test_split(X, y, test_size=0.3)

    # fit LDA:
    clf = LDA()
    clf.fit(train_x,train_y)
    y_predict = clf.predict(test_x)
    y_train_hat = clf.predict(train_x)
    lda_train_error.append\
    (sum(np.ones(len(y_train_hat))[y_train_hat!=train_y])/len(tr
ain_y))
    lda_test_error.append\
    (sum(np.ones(len(test_y))[y_predict!=test_y])/len(test_y))

    #fit QDA:
    clf = QDA()
    clf.fit(train_x,train_y)
    y_predict = clf.predict(test_x)
    y_train_hat = clf.predict(train_x)
    qda_train_error.append\
    (sum(np.ones(len(y_train_hat))[y_train_hat!=train_y])/len(tr
ain_y))
    qda_test_error.append\
    (sum(np.ones(len(test_y))[y_predict!=test_y])/len(test_y))
```

In [5]:

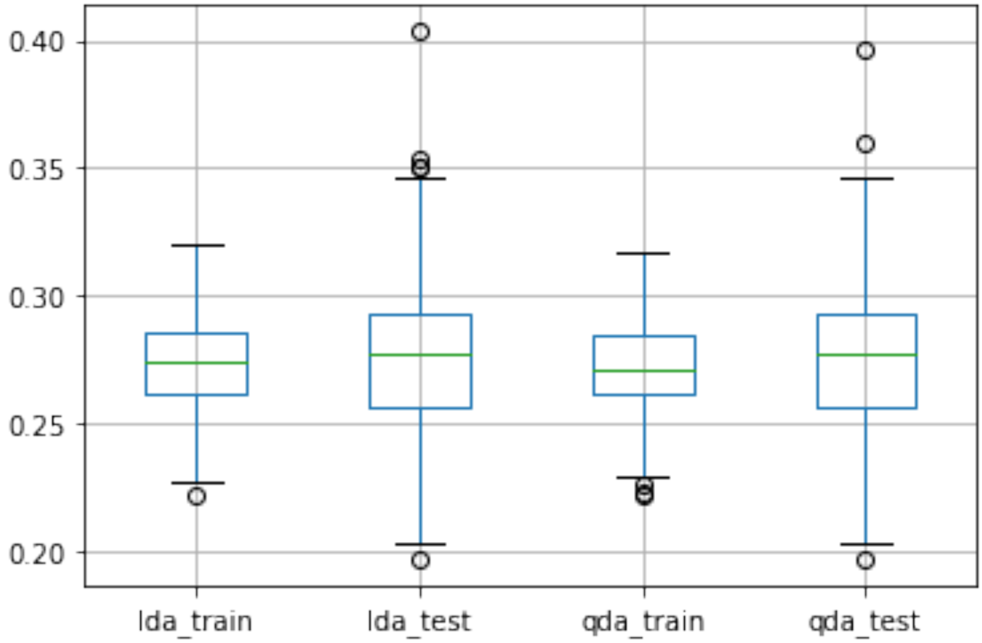
```
df_result = pd.DataFrame({'lda_train':lda_train_error,\
                          'lda_test':lda_test_error,\
                          'qda_train':qda_train_error,\
                          'qda_test':qda_test_error})
df_result.describe()
```

Out[5]:

	lda_train	lda_test	qda_train	qda_test
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.273866	0.275513	0.272747	0.275603
std	0.016949	0.026008	0.016644	0.026039
min	0.221429	0.196667	0.221429	0.196667
25%	0.261429	0.256667	0.261429	0.256667
50%	0.273571	0.276667	0.271429	0.276667
75%	0.285714	0.293333	0.284286	0.293333
max	0.320000	0.403333	0.317143	0.396667

In [6]:

```
df_result.boxplot()
plt.show()
```



When it is a linear case, the result of QDA and IDA has no big difference. But QDA has a slightly lower error rate in training set and a slightly higher error rate in test set than LDA.

In [ ]:

```

```

Question3

As the Bayes decision boundary is not linear, QDA performs better and produces less errors in both training and test sets.

In [ ]:

```
lda_train_error = []
qda_train_error = []
lda_test_error = []
qda_test_error = []
for n in range(1000):
    random.seed(n)
    #stimulate dataset
    X1 = np.array([random.uniform(-1,1) for i in range(1000)])
    X2 = np.array([random.uniform(-1,1) for i in range(1000)])
    err = np.array([np.random.normal(loc=0.0, scale=1, size=None
)\
                    for i in range(1000)])
    #simulate y
    y = X1 + X2 + X1*X1 + X2*X2 + err
    y = y >= 0
    X = np.stack(((X1), (X2)), axis=1)
    train_x, test_x, train_y, test_y = train_test_split(X, y, te
st_size=0.3)

    # LDA training and testing:
    clf = LDA()
    clf.fit(train_x,train_y)
    y_predict = clf.predict(test_x)
    y_train_hat = clf.predict(train_x)
    lda_train_error.append(sum(np.ones(len(y_train_hat))\
                               [y_train_hat!=train_y])/len(train
_y))
    lda_test_error.append(sum(np.ones(len(test_y))\
                              [y_predict!=test_y])/len(test_y))

    #QDA training and testing:
    clf = QDA()
    clf.fit(train_x,train_y)
    y_predict = clf.predict(test_x)
    y_train_hat = clf.predict(train_x)
    qda_train_error.append(sum(np.ones(len(y_train_hat))\
                               [y_train_hat!=train_y])/len(train
_y))
    qda_test_error.append(sum(np.ones(len(test_y))\
                              [y_predict!=test_y])/len(test_y))
```

In [37]:

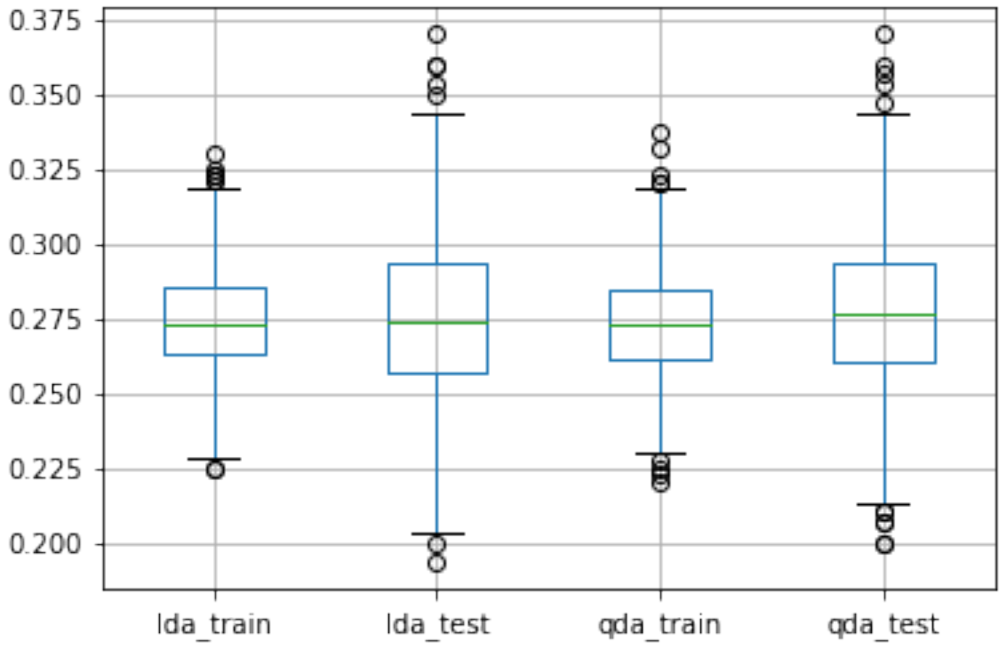
```
df_result = pd.DataFrame({'lda_train':lda_train_error,'lda_test'
:lda_test_error,\
                          'qda_train':qda_train_error,'qda_test'
:qda_test_error})
df_result.describe()
```

Out[37]:

	lda_train	lda_test	qda_train	qda_test
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.273924	0.276053	0.273010	0.276143
std	0.017474	0.026531	0.017215	0.026403
min	0.224286	0.193333	0.220000	0.200000
25%	0.262857	0.256667	0.261429	0.260000
50%	0.272857	0.273333	0.272857	0.276667
75%	0.285714	0.293333	0.284286	0.293333
max	0.330000	0.370000	0.337143	0.370000

In [38]:

```
df_result.boxplot()
plt.show()
```



In this non-linear case, QDA performs better and produces less errors in both training and test sets.

In [ ]:

```

```

Question4

As n increases, the test error rate of both QDA and LDA would decrease, but QDA decreases faster, so it has better performance on test error as larger data could reduce the overfitting problem.

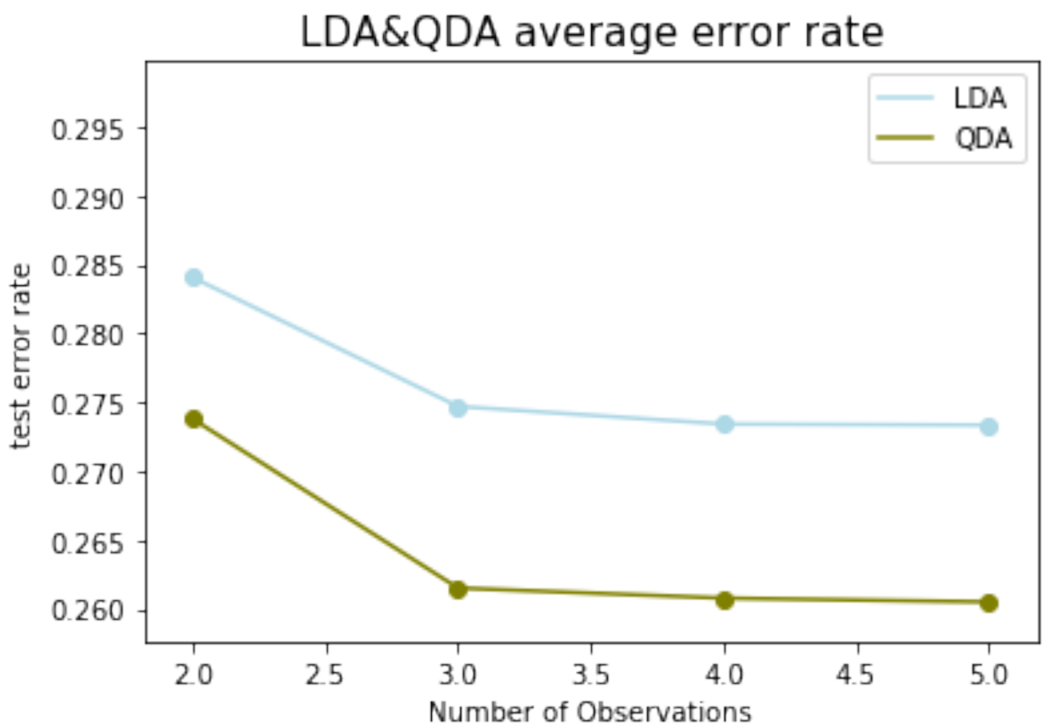
In [7]:

```
LDA_train_error = []
LDA_test_error = []
QDA_train_error = []
QDA_test_error = []
N_lst = [1e02, 1e03, 1e04, 1e05]
for N in N_lst:
    N = int(N)
    lda_train_error = []
```

```
    lda_train_error = []
    qda_train_error = []

    lda_test_error = []
    qda_test_error = []
    for time in range(1000):
        random.seed(time)
        #stimulate dataset
        X1 = np.array([random.uniform(-1,1) for i in range(N)])
        X2 = np.array([random.uniform(-1,1) for i in range(N)])
        e = np.array([np.random.normal(loc=0.0, scale=1, size=No
ne)\
                      for i in range(N)])
        y = X1 + X2 + X1**2 + X2**2 + e
        y = y >= 0
        X = np.stack(((X1), (X2)), axis=1)
        train_x, test_x, train_y, test_y = \
        train_test_split(X, y, test_size=0.3)

        #LDA training and testing
        clf = LDA()
        clf.fit(train_x,train_y)
        y_predict = clf.predict(test_x)
        y_train_hat = clf.predict(train_x)
        lda_train_error.append(sum(np.ones(len(y_train_hat))\
                                   [y_train_hat!=train_y])/len(t
rain_y))
        lda_test_error.append(sum(np.ones(len(test_y))\
                                  [y_predict!=test_y])/len(test_
y))
        #QDA training and testing
        clf = QDA()
        clf.fit(train_x,train_y)
        y_predict = clf.predict(test_x)
        y_train_hat = clf.predict(train_x)
        qda_train_error.append(sum(np.ones(len(y_train_hat))\
                                   [y_train_hat!=train_y])/len(t
rain_y))
        qda_test_error.append(sum(np.ones(len(test_y))\
                                  [y_predict!=test_y])/len(test_
y))
    LDA_train_error.append(np.mean(lda_train_error))
    QDA_train_error.append(np.mean(qda_train_error))
    LDA_test_error.append(np.mean(lda_test_error))
    QDA_test_error.append(np.mean(qda_test_error))
```

In [17]:

```
plt.scatter(np.log10(np.array(N_list)), LDA_test_error,color='li
ghtblue')
plt.scatter(np.log10(np.array(N_list)), QDA_test_error,color='ol
ive')
plt.plot(np.log10(np.array(N_list)), LDA_test_error,color= 'ligh
tblue', label='LDA')
plt.plot(np.log10(np.array(N_list)), QDA_test_error,color= 'oliv
e', label ='QDA')
plt.ylabel('test error rate',size=10)
plt.xlabel('Number of Observations',size=10)
plt.title('LDA&QDA average error rate',size=15)
plt.legend()
plt.show()
```



The graph tells that QDA obviously produces lower test error rate in non-linear cases although both the two lines are decreasing when we have increasing number of observations.

In [ ]:

Modeling voter turnout

Question5

In [23]:

```
#data cleaning
df = pd.read_csv('mental_health.csv')
df = df.dropna()
df.head(5)
```

Out[23]:

	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 [26]:

```
#spliting data
X = df[['mhealth_sum','age','educ','black','female','married','i
nc10']]
y = df['vote96']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_s
ize=0.3, shuffle = True)
```

In [27]:

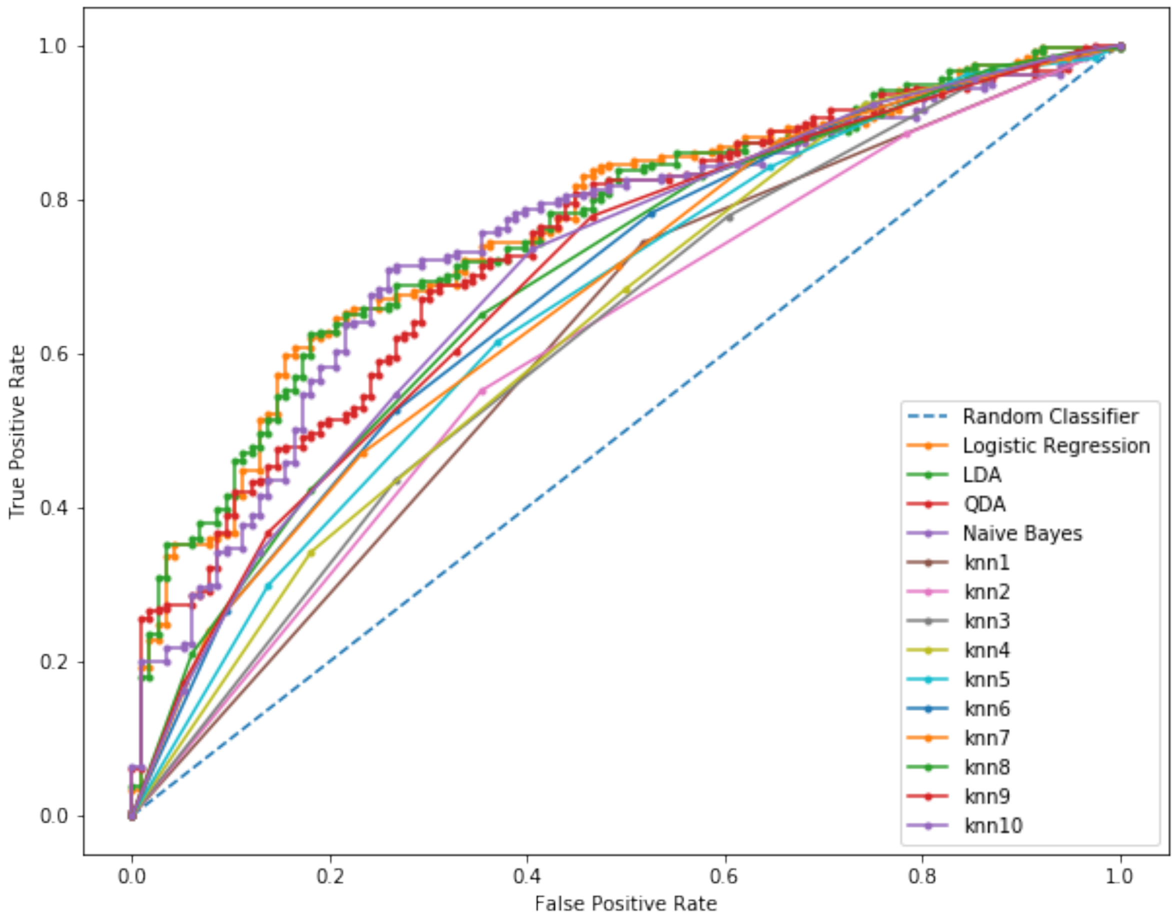
```
#Logistic Regression
lr = LogisticRegression()
lr.fit(X_train,y_train)
lr_error = 1 - lr.score(X_test, y_test)

#LDA
lda = LDA()
lda.fit(X_train,y_train)
lda_error = 1 - lda.score(X_test, y_test)

#QDA
qda = QDA()
qda.fit(X_train,y_train)
qda_error = 1 - qda.score(X_test, y_test)

#NAIVE BAYES
gnb = GaussianNB()
gnb.fit(X_train, y_train)
gnb_error = 1 - gnb.score(X_test, y_test)

#KNN
def KNN_fit_error(n):
    KNN = KNeighborsClassifier(n_neighbors=n)
    KNN.fit(X_train,y_train)
    KNN_error = 1 - KNN.score(X_test, y_test)
    return KNN, KNN_error
KNN1, KNN1_error = KNN_fit_error(1)
KNN2, KNN2_error = KNN_fit_error(2)
KNN3, KNN3_error = KNN_fit_error(3)
KNN4, KNN4_error = KNN_fit_error(4)
KNN5, KNN5_error = KNN_fit_error(5)
KNN6, KNN6_error = KNN_fit_error(6)
KNN7, KNN7_error = KNN_fit_error(7)
KNN8, KNN8_error = KNN_fit_error(8)
KNN9, KNN9_error = KNN_fit_error(9)
KNN10, KNN10_error = KNN_fit_error(10)
```

```
/opt/anaconda3/lib/python3.7/site-packages/sklearn/l
inear_model/logistic.py:432: FutureWarning: Default
solver will be changed to 'lbfgs' in 0.22. Specify a
solver to silence this warning.
  FutureWarning)
```

In [33]:

```
def get_roc_auc(model, label):
# predict model probabilities
    pbb = model.predict_proba(X_test)
# filter positive probabilities
    pbb = pbb[:, 1]
#calculate auc score
    auc = roc_auc_score(y_test, pbb)
    print(label+ ': AUC = %.3f' % (auc))
# calculate roc curves
    fpr, tpr, _ = roc_curve(y_test, pbb)
# plot the roc curve for the model
    plt.plot(fpr, tpr, marker='.', label=label)

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

def plot_random_classifier():
    plt.figure(figsize=(10,8))
# generate a random prediction line
    random_pbb = [0 for _ in range(len(y_test))]
# calculate auc scores
    random_auc = roc_auc_score(y_test, random_pbb)
    random_fpr, random_tpr, _ = roc_curve(y_test, random_pbb)
    plt.plot(random_fpr, random_tpr, linestyle='--', label='Rand
om Classifier')

plot_random_classifier()
models = [lr, lda, qda, gnb, KNN1, KNN2, KNN3, KNN4, KNN5, KNN6,
KNN7, KNN8, KNN9, KNN10]
labels = ['Logistic Regression', 'LDA', 'QDA', "Naive Bayes" ,
'knn1','knn2','knn3','knn4','knn5','knn6','knn7','knn8','knn9','
knn10']
count = 0
for model in models:
    get_roc_auc(model, labels[count])
    count+=1
```

```
Logistic Regression: AUC = 0.762
LDA: AUC = 0.762
QDA: AUC = 0.738
Naive Bayes: AUC = 0.743
knn1: AUC = 0.613
knn2: AUC = 0.610
knn3: AUC = 0.622
knn4: AUC = 0.638
knn5: AUC = 0.658
knn6: AUC = 0.681
knn7: AUC = 0.673
knn8: AUC = 0.693
knn9: AUC = 0.695
knn10: AUC = 0.700
```



5d.Answer

From the graph, considering both error rate and ROC/AUC, if we define 'best performance' to be lower test error rate and higher AUC score, Naive Bayes and QDA seem to be the best two. The two models satisfy our goal on accuracy.