30100HW2

February 2, 2020

[10]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
import math
from tabulate import tabulate
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QDA
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
from sklearn.metrics import roc_curve, auc
```

1 1

[4]:
```
#a
np.random.seed(123)
```

[5]:
```
#b
x1 = np.random.uniform(-1, 1, 200)
x2 = np.random.uniform(-1, 1, 200)
```

[6]:
```
#c
y = x1 + x1**2 + x2 + x2**2 + np.random.normal(0, 0.5, 200)
```

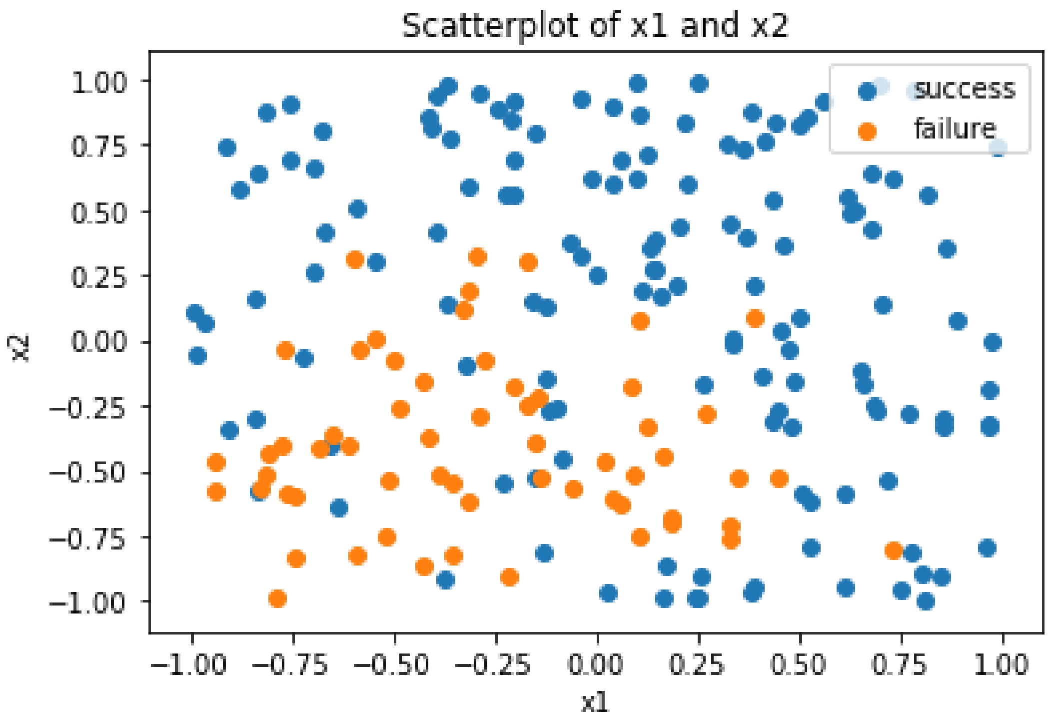
d

$$\log \frac{p}{1-p} = y$$

$$p = 1 - \frac{1}{e^y + 1}$$

[11]:
```
p = 1 - 1 / (math.e ** y + 1)
```

[17]:
```
#e
plt.scatter(x1[p > 0.5], x2[p > 0.5])
plt.scatter(x1[p <= 0.5], x2[p <= 0.5])
```



[22]:
```
df = np.stack([x1, x2], axis=1)
df = pd.DataFrame(df)
nb = GaussianNB()
nb.fit(df, p > 0.5)
```

[22]:
```
GaussianNB(priors=None, var_smoothing=1e-09)
```

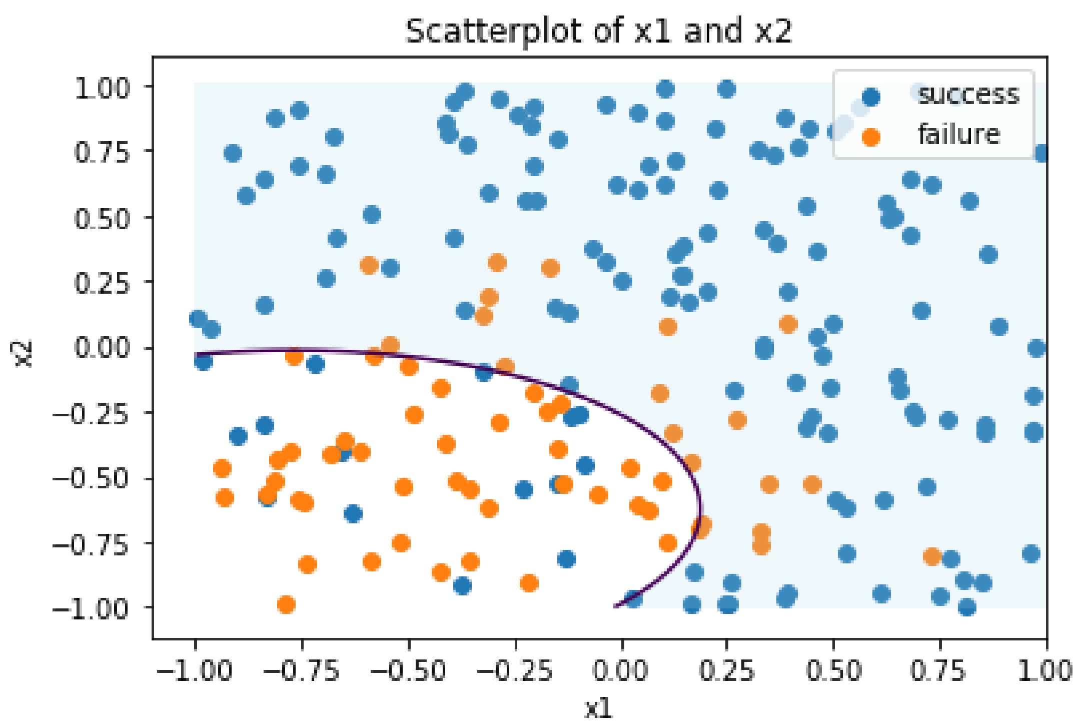
[30]:
```
xv, yv = np.meshgrid(np.linspace(-1, 1, 100), np.linspace(-1, 1, 100))
z = nb.predict_proba(np.c_[xv.ravel(), yv.ravel()])
z
```

[30]:
```
array([[6.75839408e-01, 3.24160592e-01],
       [6.78623963e-01, 3.21376037e-01],
       [6.81143326e-01, 3.18856674e-01],
       …,
       [4.84036422e-06, 9.99995160e-01],
       [4.37940135e-06, 9.99995621e-01],
       [3.95773146e-06, 9.99996042e-01]])
```

[34]:
```python
z = z[:, 1].reshape((100, 100))
z
```

[34]:
```
array([[0.32416059, 0.32137604, 0.31885667, …, 0.96907699, 0.97193926,
         0.97457239],
        [0.31291202, 0.31017975, 0.30770836, …, 0.96748567, 0.97049075,
         0.97325607],
        [0.30248471, 0.29980375, 0.2973794 , …, 0.96590999, 0.96905604,
         0.97195192],
        …,
        [0.99950124, 0.99949484, 0.99948897, …, 0.99999236, 0.99999309,
         0.99999376],
        [0.99960233, 0.99959723, 0.99959254, …, 0.99999391, 0.99999449,
         0.99999502],
        [0.99968384, 0.99967979, 0.99967606, …, 0.99999516, 0.99999562,
         0.99999604]])
```

[39]:
```python
#fgh
plt.scatter(x1[p > 0.5], x2[p > 0.5])
plt.scatter(x1[p <= 0.5], x2[p <= 0.5])
plt.contour(xv, yv, z, [0.5])
plt.contourf(xv, yv, z, [0.5,1], colors='lightblue', alpha=0.2)
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Scatterplot of x1 and x2')
plt.legend(['success','failure'], loc=1);
```



2 1

If the Bayes boundary is linear, QDA should perform better on the training set, because even if the boundary is linear, there are overlapping areas of classification, which enables the more flexible QDA to fit better. But regarding the test set, QDA always overfits the training set, which leads to bad performance on the test set compared with LDA, which can properly depict the boundary.

2.1 a

[52]:
```
def simulate(n, nonlinear=0):
    df_err = []
    for _ in range(1000):
        #i
        x1_2 = np.random.uniform(-1,1,n)
        x2_2 = np.random.uniform(-1,1,n)
        y_2 = x1_2 + x2_2 + (x1_2**2)*nonlinear + (x2_2**2)*nonlinear + np.
 ↪random.normal(0, 1, n)
        classifier = y_2 >= 0
        x_2 = np.stack([x1_2, x2_2], axis=1)

        #ii
        x_train, x_test, c_train, c_test = train_test_split(x_2, classifier,␣
 ↪test_size=0.3)

        #iii
        lda = LDA()
        lda.fit(x_train, c_train)
        qda = QDA()
        qda.fit(x_train, c_train)

        #iv
        lda_train_err = 1 - lda.score(x_train,c_train)
        lda_test_err = 1 - lda.score(x_test,c_test)
        qda_train_err = 1 - qda.score(x_train,c_train)
        qda_test_err = 1 - qda.score(x_test,c_test)
        df_err.append([lda_train_err, lda_test_err, qda_train_err,␣
 ↪qda_test_err])
    return df_err


df_err = simulate(1000)
```

2.2 b

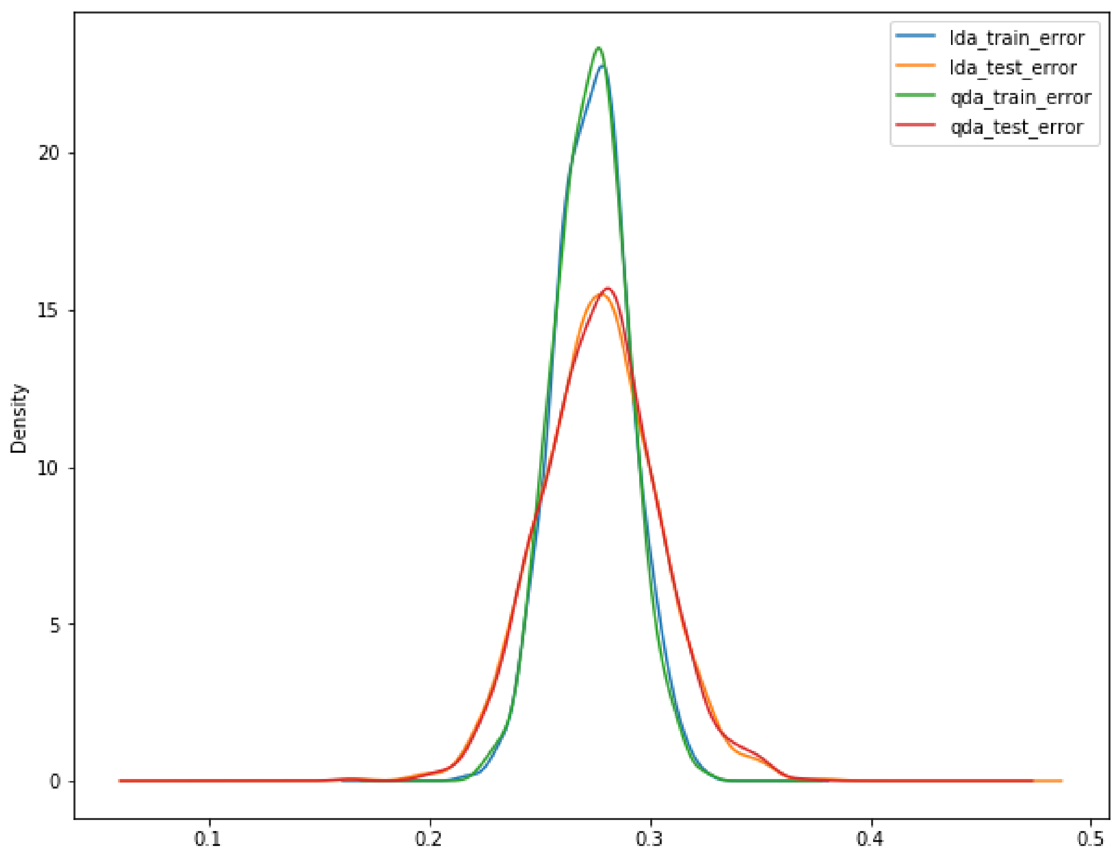
[49]:
```
df_err = pd.DataFrame(df_err, columns=["lda_train_error", "lda_test_error",␣
↪"qda_train_error", "qda_test_error"])
df_err.describe()
```

[49]:

	lda_train_error	lda_test_error	qda_train_error	qda_test_error
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.274150	0.276870	0.273421	0.277347
std	0.017011	0.026439	0.016788	0.026304
min	0.215714	0.166667	0.221429	0.163333
25%	0.262857	0.260000	0.261429	0.260000
50%	0.274286	0.276667	0.274286	0.276667
75%	0.284286	0.293333	0.284286	0.293333
max	0.325714	0.380000	0.325714	0.370000

[51]:
```
df_err.plot.density(figsize=(10, 8))
```

[51]: <matplotlib.axes._subplots.AxesSubplot at 0x147ad212820>



According to the above graph, the distribution of the error rates generated by QDA on the training set is skewed more to the right than IDA, and regarding the test set, LDA performs slightly better than QDA. Thus, this graph supports the above preposition.

3 3

If the Bayes boundary is non-linear, QDA should still perform better on the training set because of its complexity. In this case, LDA does not match the pattern of the boundary, so it performs worse than QDA on the test set, with QDA the better choice in both cases.

3.1 a

[53]:
```
#call the function in part 2
df_err_3 = simulate(1000, 1)
```

3.2 b

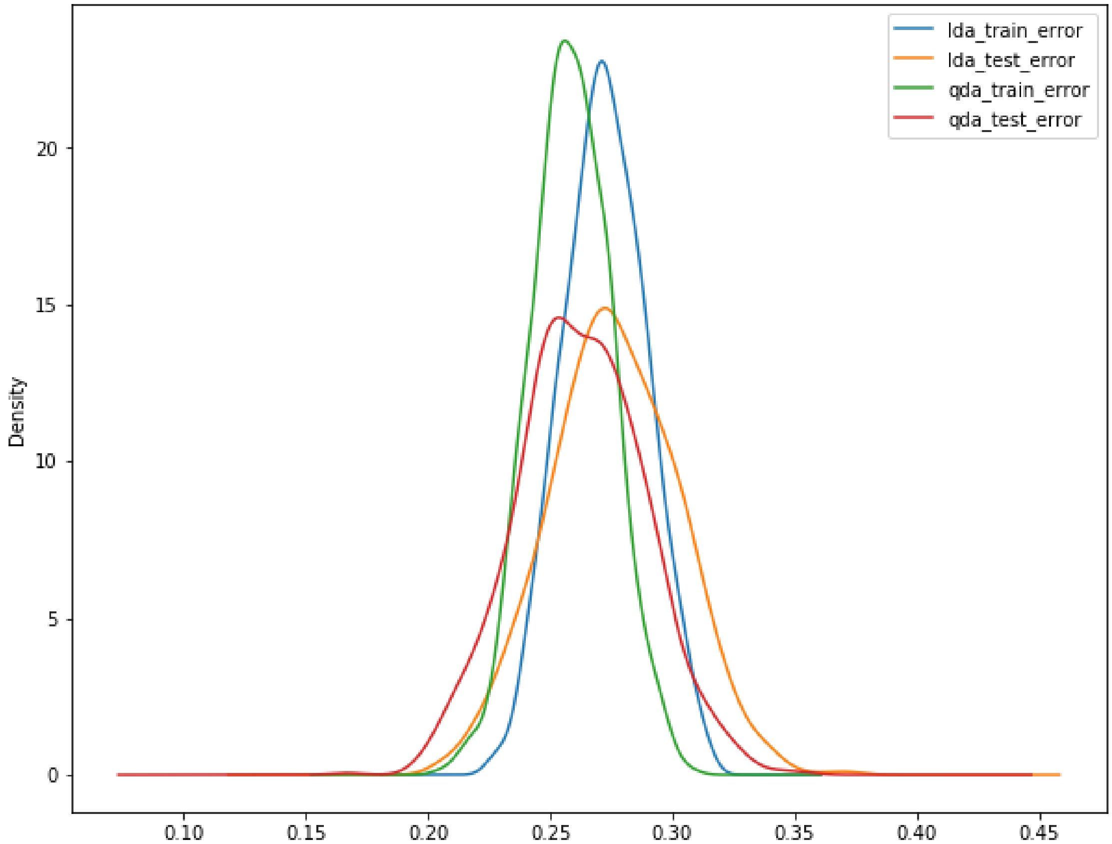
[54]:
```
df_err_3 = pd.DataFrame(df_err_3, columns=["lda_train_error", "lda_test_error",␣
↪"qda_train_error", "qda_test_error"])
df_err_3.describe()
```

[54]:

	lda_train_error	lda_test_error	qda_train_error	qda_test_error
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.272360	0.275973	0.258631	0.262437
std	0.016898	0.026298	0.016342	0.026035
min	0.225714	0.203333	0.204286	0.166667
25%	0.261071	0.256667	0.247143	0.245833
50%	0.271429	0.276667	0.258571	0.263333
75%	0.284286	0.293333	0.270000	0.280000
max	0.315714	0.373333	0.308571	0.353333

[55]:
```
df_err_3.plot.density(figsize=(10, 8))
```

[55]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x147acfc5340>
```



According to the above graph, QDA training error rate is significantly better than IDA training error rate and regarding the test set, QDA still performs better than IDA, which completely proves the above preposition.

4 4

When the sample size is relatively small, LDA can avoid the problem of overfitting, with QDA likely to overfit the training set. When sample size is large enough, it becomes capable of handling all the terms necessary for the model. Hence the test error rate of QDA will improve relative to LDA as sample size increases.

4.1 a

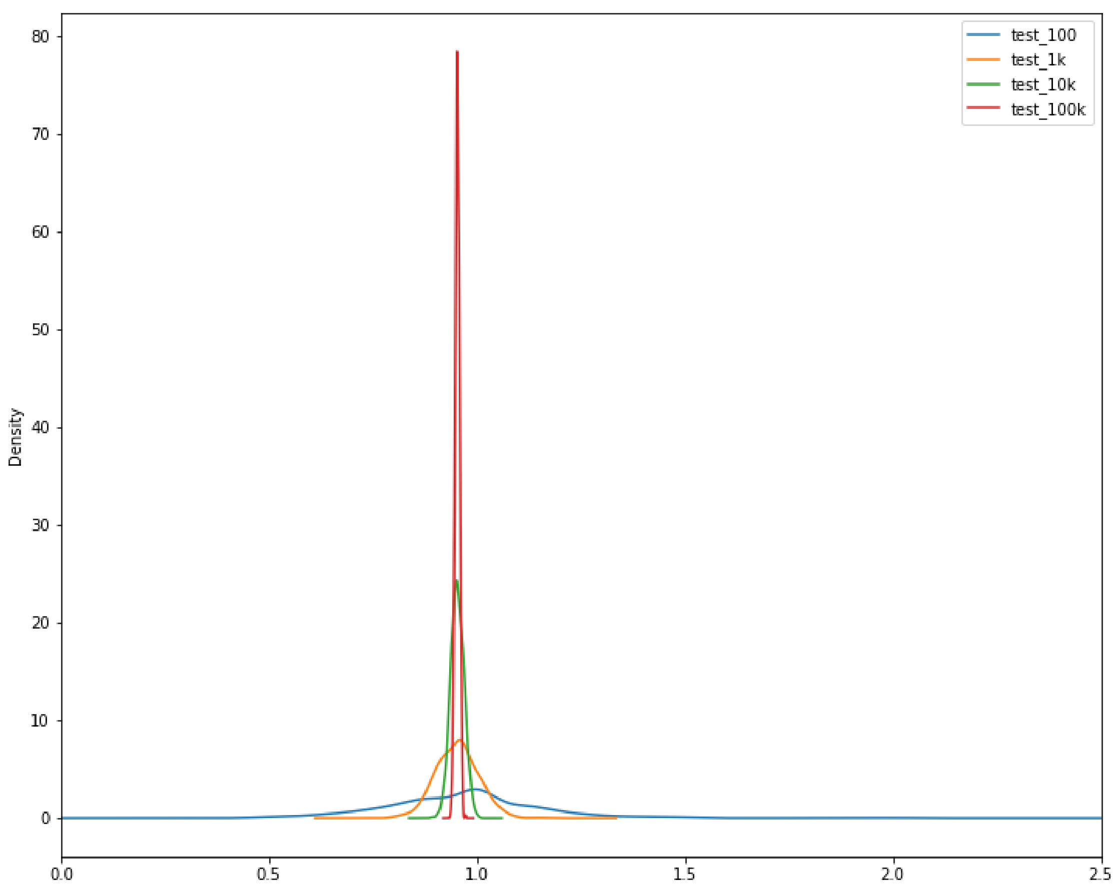
[106]:
```
df_1e02 = simulate(100, 1)
df_1e03 = simulate(1000, 1)
df_1e04 = simulate(10000, 1)
df_1e05 = simulate(100000, 1)
```

4.2 b

[107]:
```
df_1e02 = pd.DataFrame(df_1e02, columns=["lda_train_100", "lda_test_100",␣
↪"qda_train_100", "qda_test_100"]).iloc[:, [1, 3]]
df_1e03 = pd.DataFrame(df_1e03, columns=["lda_train_1k", "lda_test_1k",␣
↪"qda_train_1k", "qda_test_1k"]).iloc[:, [1, 3]]
df_1e04 = pd.DataFrame(df_1e04, columns=["lda_train_10k", "lda_test_10k",␣
↪"qda_train_10k", "qda_test_10k"]).iloc[:, [1, 3]]
df_1e05 = pd.DataFrame(df_1e05, columns=["lda_train_100k", "lda_test_100k",␣
↪"qda_train_100k", "qda_test_100k"]).iloc[:, [1, 3]]
```

[114]:
```
df_sum = pd.concat([df_1e02, df_1e03, df_1e04, df_1e05], axis=1)
# calculate the ratio of qda error rate to lda error rate
df_sum['test_100'] = df_sum['qda_test_100'] / df_sum['lda_test_100']
df_sum['test_1k'] = df_sum['qda_test_1k'] / df_sum['lda_test_1k']
df_sum['test_10k'] = df_sum['qda_test_10k'] / df_sum['lda_test_10k']
df_sum['test_100k'] = df_sum['qda_test_100k'] / df_sum['lda_test_100k']
df_sum[['test_100', 'test_1k', 'test_10k', 'test_100k']].plot.
↪density(figsize=(12, 10), xlim=(0, 2.5))
```

[114]: <matplotlib.axes._subplots.AxesSubplot at 0x147afbecf10>



As showed in the above graph, the distributions of the test error rate ratio is more to the left as sample size increases, which means that the qda test error rate decreases compared with lda and proves my preposition.

5 5

5.1 a

[118]:
```
df_mh = pd.read_csv('E:/R/Working Directory/mental_health.csv')
df_mh.dropna(inplace=True)
df_mh
```

[118]:

	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
…	…	…	…	…	…	…	…	…
2822	1.0	2.0	37.0	14.0	0	0	1.0	5.8849
2823	1.0	2.0	30.0	12.0	0	1	1.0	3.4774
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

[1165 rows x 8 columns]

[121]:
```
vote96 = df_mh['vote96']
x5 = df_mh.drop(columns=['vote96'])
x5_train, x5_test, v_train, v_test = train_test_split(x5, vote96, test_size=0.3)
```

5.2 b

[122]:
```
#i
log = LogisticRegression()
log.fit(x5_train,v_train)
log_err = 1 - log.score(x5_test, v_test)

#ii
lda = LDA()
lda.fit(x5_train,v_train)
lda_err = 1 - lda.score(x5_test, v_test)

#iii
qda = QDA()
```

```python
qda.fit(x5_train,v_train)
qda_err = 1 - qda.score(x5_test, v_test)

#iv
nb = GaussianNB()
nb.fit(x5_train, v_train)
nb_err = 1 - nb.score(x5_test, v_test)

#v
knn_list = []
knn_err_list = []
for i in range(1, 11):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(x5_train,v_train)
    knn_err = 1 - knn.score(x5_test, v_test)
    knn_list.append(knn)
    knn_err_list.append(knn_err)
```

5.3 c

[123]:
```python
print(tabulate([['Logistic Regression Model', log_err],
 ['LDA', lda_err],
 ['QDA', qda_err],
 ['Naive Bayes', nb_err],
 ['KNN, K=1', knn_err_list[0]],
 ['KNN, K=2', knn_err_list[1]],
 ['KNN, K=3', knn_err_list[2]],
 ['KNN, K=4', knn_err_list[3]],
 ['KNN, K=5', knn_err_list[4]],
 ['KNN, K=6', knn_err_list[5]],
 ['KNN, K=7', knn_err_list[6]],
 ['KNN, K=8', knn_err_list[7]],
 ['KNN, K=9', knn_err_list[8]],
 ['KNN, K=10', knn_err_list[9]]],
 headers = ['Type', 'Test Error Rate']))
```

```
Type                         Test Error Rate
-------------------------  -----------------
Logistic Regression Model           0.311429
LDA                                 0.317143
QDA                                 0.322857
Naive Bayes                         0.328571
KNN, K=1                            0.368571
KNN, K=2                            0.417143
KNN, K=3                            0.345714
KNN, K=4                            0.357143
KNN, K=5                            0.331429
```

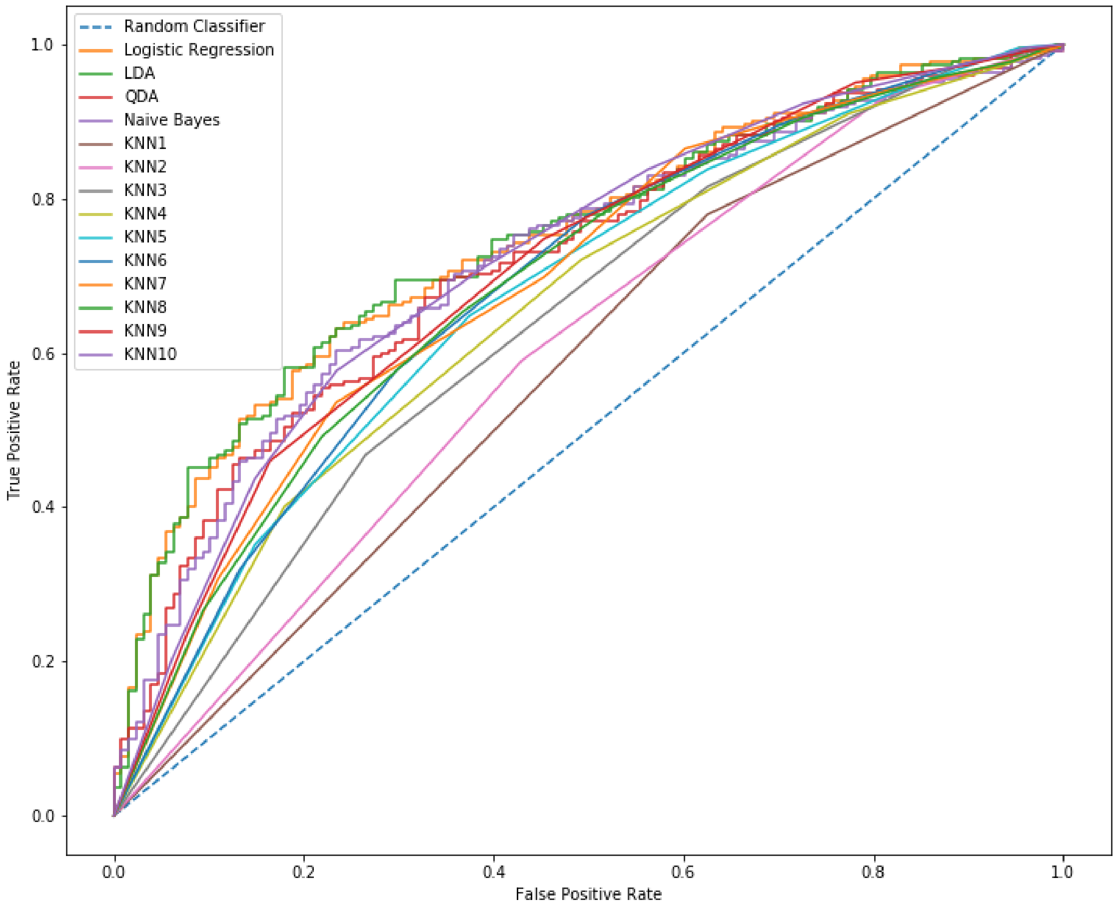
```
KNN, K=6                              0.322857
KNN, K=7                              0.305714
KNN, K=8                              0.325714
KNN, K=9                              0.32
KNN, K=10                             0.308571
```

[129]:

```
def auc_roc(model, name):
    probs = model.predict_proba(x5_test)[:, 1]
    auc = roc_auc_score(v_test, probs)
    print(name+ ': AUC = %f' % (auc))
    fpr, tpr, _ = roc_curve(v_test, probs)
    # 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(v_test)
rand_fpr, rand_tpr, _ = roc_curve(v_test, rand_probs)
plt.plot(rand_fpr, rand_tpr, linestyle='--', label='Random Classifier')
model_list = [log, lda, qda, nb]
model_list.extend(knn_list)
name_list = ['Logistic Regression', 'LDA', 'QDA', "Naive Bayes" ,
         ␣
 ↪'KNN1','KNN2','KNN3','KNN4','KNN5','KNN6','KNN7','KNN8','KNN9','KNN10']
for i, model in enumerate(model_list):
    auc_roc(model, name_list[i])
```

```
Logistic Regression: AUC = 0.746059
LDA: AUC = 0.746446
QDA: AUC = 0.717413
Naive Bayes: AUC = 0.720756
KNN1: AUC = 0.577140
KNN2: AUC = 0.599363
KNN3: AUC = 0.638302
KNN4: AUC = 0.656690
KNN5: AUC = 0.673582
KNN6: AUC = 0.686128
KNN7: AUC = 0.692075
KNN8: AUC = 0.690034
KNN9: AUC = 0.705201
KNN10: AUC = 0.719735
```



5.4 d

Accroding to the above analyses, the best model for this dataset is LDA, which produces the highest AUC, proving that it strikes a great balance of precision and recall. Its test error rate is also among the lowest.