modelhw2

February 2, 2020

1 Homework 2: Classification Methods

[1]:
```
# import packages
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 LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
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 math import exp
```

1.1 The Bayes Classifier

1. For classification problems, the test error rate is minimized by a simple classifier that assigns each observation to the most likely class given its predictor values: Pr(Y = j|X = x0) where x0 is the test observation and each possible class is represented by J. This is a conditional probability that Y = j, given the observed predictor vector x0. This classifier is known as the Bayes classifier. If the response variable is binary (i.e. two classes), the Bayes classifier corresponds to predicting class one if Pr(Y = 1|X = x0) > 0.5, and class two otherwise. Produce a graph illustrating this concept. Specifically, implement the following elements in your program:

a. Set your random number generator seed.
b. Simulate a dataset of N = 200 with X1, X2 where X1, X2 are random uniform variables between [−1, 1].
c. CalculateY =X1+X12+X2+X2+ ,where N(=0, 2 =0.25).
d. Y is defined in terms of the log-odds of success on the domain [−∞, +∞]. Calculate the probability of success bounded between [0, 1].
e. Plot each of the data points on a graph and use color to indicate if the observation was a success or a failure.
f. Overlay the plot with Bayes decision boundary, calculated using X1,X2.
g. Give your plot a meaningful title and axis labels.
h. The colored background grid is optional.

[17]:

```python
# set random seed
np.random.seed(2019)
#Simulate a dataset of N = 200 with X1, X2
X = np.random.uniform(-1,1,(2,200))
#CalculateY =X1+X12+X2+X2+ ,where N( =0, 2 =0.25).
Y = X[0,:] + X[0,:]**2 + X[1,:] + X[1,:]**2 +  np.random.normal( 0, 0.5, 200)
#Y is defined in terms of the log-odds of success on the domain [-∞, +∞].
#Calculate the probability of success bounded between [0, 1].
def logit2prob(x):
    odds=np.exp(np.array(x))
    prob=odds/(1 + odds)
    return prob
prob=logit2prob(Y)
probtf=prob>0.5
prob
```

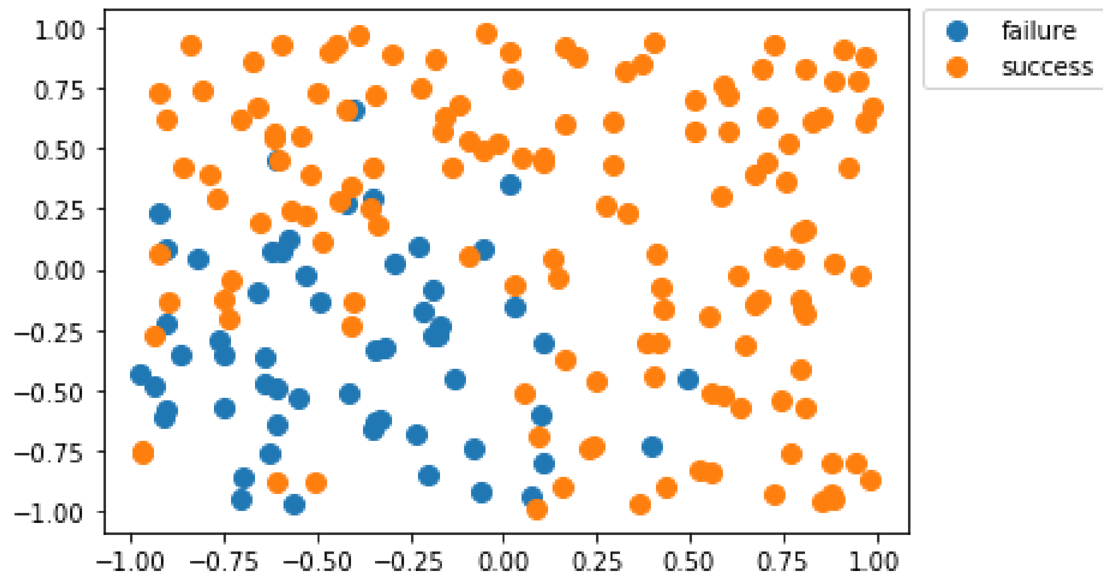
[17]:

```
array([0.82849187, 0.4608178 , 0.51176518, 0.60518982, 0.90938358,
       0.59006533, 0.96321944, 0.75761982, 0.94968088, 0.48865303,
       0.52303872, 0.90609121, 0.74523342, 0.64021733, 0.79025966,
       0.96359111, 0.53661513, 0.47997726, 0.32117233, 0.56016197,
       0.5738233 , 0.56393526, 0.47938579, 0.5309953 , 0.59683373,
       0.54477001, 0.57012001, 0.44425388, 0.83193474, 0.37961348,
       0.69761183, 0.85921594, 0.6296643 , 0.94861151, 0.84212708,
       0.71464094, 0.6528656 , 0.63641372, 0.46855042, 0.72024099,
       0.83368541, 0.2684377 , 0.50721412, 0.84434893, 0.60625281,
       0.91224952, 0.75334057, 0.75879306, 0.57100548, 0.98242394,
       0.4034332 , 0.46750637, 0.47205267, 0.82830874, 0.67958694,
       0.84624222, 0.38998106, 0.60138008, 0.43033285, 0.58575614,
       0.72992691, 0.44086161, 0.21546613, 0.83660791, 0.70091667,
       0.65390153, 0.8829348 , 0.72034761, 0.46351842, 0.85942801,
       0.29297696, 0.40447946, 0.65573051, 0.83797823, 0.63467034,
       0.7319606 , 0.97979748, 0.22591391, 0.78143954, 0.64677296,
       0.27131764, 0.66862327, 0.60950501, 0.48062551, 0.93391771,
       0.56902771, 0.38568622, 0.348932  , 0.62494238, 0.34982382,
       0.35166567, 0.81298598, 0.59648047, 0.86957333, 0.71943754,
       0.45679997, 0.48496706, 0.72286644, 0.29538359, 0.79582533,
       0.46760114, 0.67828772, 0.67883433, 0.65168941, 0.83776143,
       0.61611539, 0.98303505, 0.59678071, 0.60085863, 0.52494716,
       0.53873482, 0.63684489, 0.50774389, 0.62360852, 0.63971866,
       0.95834086, 0.70657401, 0.37470773, 0.5739381 , 0.34114826,
       0.43596874, 0.80545546, 0.16948187, 0.57523725, 0.47504385,
       0.7978173 , 0.91738198, 0.76164394, 0.86292607, 0.5341799 ,
       0.43914779, 0.44464322, 0.3554489 , 0.27323623, 0.89913037,
       0.81451059, 0.64782501, 0.92054441, 0.89216021, 0.31424726,
       0.9623672 , 0.92904984, 0.9808496 , 0.69442947, 0.50784292,
       0.48764802, 0.41643588, 0.69705644, 0.2148158 , 0.50237935,
       0.69171392, 0.40122293, 0.75039878, 0.81354675, 0.9422711 ,
```

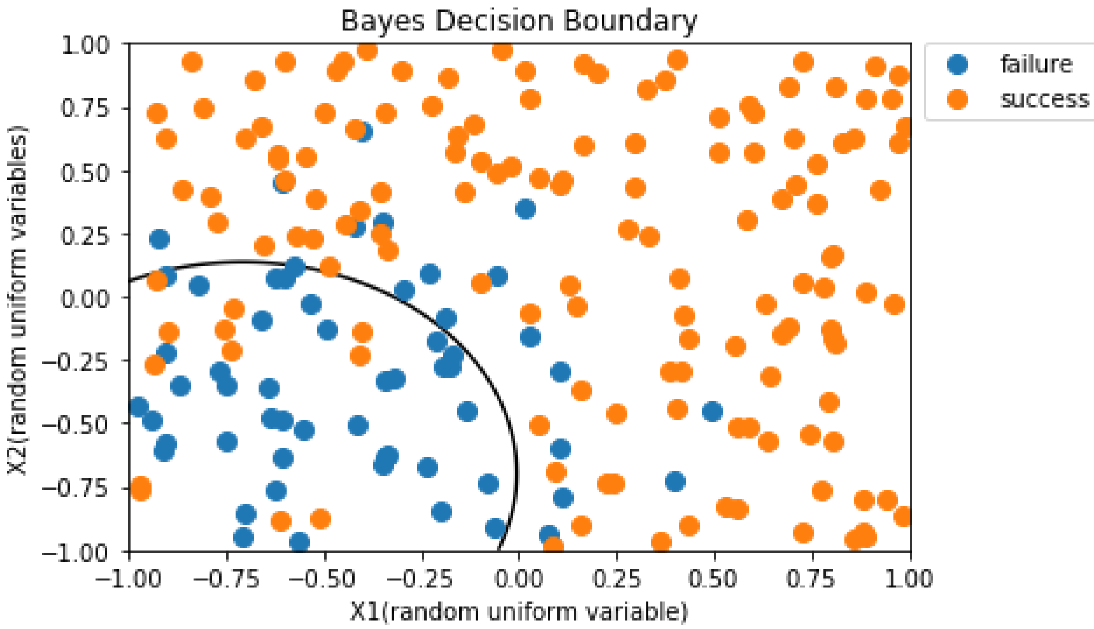
```
0.84681109, 0.28927443, 0.67763596, 0.32093472, 0.57147493,
0.39206698, 0.70836429, 0.3435361 , 0.54154384, 0.69181114,
0.27088757, 0.61302608, 0.88045321, 0.54098313, 0.85206581,
0.34219011, 0.35369037, 0.5021619 , 0.69137368, 0.77531435,
0.53858279, 0.35298938, 0.6533829 , 0.67150222, 0.81353448,
0.8856535 , 0.84202067, 0.24510023, 0.70479358, 0.957143  ,
0.44740698, 0.36650411, 0.82491493, 0.91924921, 0.70098476,
0.52979499, 0.24490036, 0.86720487, 0.45146117, 0.59474726,
0.62640475, 0.70520926, 0.23437945, 0.67202643, 0.90593173])
```

[18]:
```
data = np.vstack((X, probtf))
df = pd.DataFrame(dict(x=data[0,:], y=data[1,:], label=data[2,:]))
def namechange(i):
    if i==0:
        return 'failure'
    else:
        return 'success'
df['label']=df['label'].apply(namechange)
groups = df.groupby('label')
#Plot each of the data points on a graph and use color to indicate
#if the observation was a success or a failure.
fig, ax = plt.subplots()
for name, group in groups:
    ax.plot(group.x, group.y, marker='o', linestyle='', ms=8, label=name)
ax.legend()
ax.legend(bbox_to_anchor=(1.02, 1), loc=2, borderaxespad=0.)
```

[18]: <matplotlib.legend.Legend at 0x119b69908>



[28]:
```
#Overlay the plot with Bayes decision boundary, calculated using X1,X2.
nb = GaussianNB().fit(X.transpose(), probtf)
xlim = (-1, 1)
ylim = (-1, 1)
xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1],100),
                     np.linspace(ylim[0], ylim[1], 100))
Z = nb.predict_proba(np.c_[xx.ravel(), yy.ravel()])
Z = Z[:,1].reshape(xx.shape)

data = np.vstack((X, probtf))
df = pd.DataFrame(dict(x=data[0,:], y=data[1,:], label=data[2,:]))
def namechange(i):
    if i==0:
        return 'failure'
    else:
        return 'success'
df['label']=df['label'].apply(namechange)
groups = df.groupby('label')
fig, ax = plt.subplots()
for name, group in groups:
    ax.plot(group.x, group.y, marker='o', linestyle='', ms=8, label=name)
ax.contour(xx, yy, Z,  [0.5], colors='k')
ax.legend()
ax.legend(bbox_to_anchor=(1.02, 1), loc=2, borderaxespad=0.)
# Give your plot a meaningful title and axis labels.
plt.xlabel('X1(random uniform variable)')
plt.ylabel('X2(random uniform variables)')
plt.title('Bayes Decision Boundary')
plt.show()
```



1.2 Exploring Simulated Differences between LDA and QDA

1.2.1 2. If the Bayes decision boundary is linear, do we expect LDA or QDA to perform better on the training set? On the test set?

On the training set, QDA would perform better because it is more flexible than LDA On the testing set, LDA would perform better because the Bayes decision boundary is linear and QDA might overfit

a. Repeat the following process 1000 times

i. Simulate a dataset of 1000 observations with X1 , X2 Uniform(−1, +1). Y is a binary response variable defined by a Bayes decision boundary of f(X) = X1 + X2, where values 0 or greater are coded TRUE and values less than 0 or coded FALSE. Whereas your simulated Y is a function of X1 + X2 + where N (0, 1). That is, your simulated Y is a function of the Bayes decision boundary plus some irreducible error.
ii. Randomly split your dataset into 70/30% training/test sets.
iii. Use the training dataset to estimate LDA and QDA models.
iv. Calculate each model's training and test error rate.

[5]:
```
def linearsim(n): #take the number of simulations as parameter
        np.random.seed(n)
        #Simulate a dataset of 1000 observation
        X = np.random.uniform(-1,1,(2,1000))
        #Y is a binary response variable defined by a Bayes decision boundary of
        #f(X) = X1 + X2, , where values 0 or greater are coded TRUE and values␣
↪less than 0
```

```
        #or coded FALSE. Whereas your simulated Y is a function of X1 + X2 +
        #where    N (0, 1). That is, your simulated Y is a function of the
        #Bayes decision boundary plus some irreducible error.
        Y = (X[0,:] + X[1,:] + np.random.normal(size = 1000))>=0
        data=np.vstack((X, Y))
        seq = np.arange(1000)
        np.random.shuffle(seq)
        #Randomly split your dataset into 70/30% training/test sets
        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
```

[6]:

```
# 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=linearsim(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
```

b. Summarize all the simulations' error rates and report the results in tabular and graphical form. Use this evidence to support your answer.

[7]:
```
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()
```

[7]:

	LDA_train	LDA_test	QDA_train	QDA_test
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.274107	0.275237	0.273340	0.275590
std	0.016798	0.025362	0.016640	0.025507
min	0.220000	0.193333	0.217143	0.196667
25%	0.262857	0.256667	0.262857	0.260000
50%	0.274286	0.276667	0.274286	0.276667
75%	0.284286	0.293333	0.284286	0.293333
max	0.324286	0.360000	0.320000	0.360000

[8]:
```
# mean error rates
df.describe().loc[['mean']]
```

[8]:

	LDA_train	LDA_test	QDA_train	QDA_test
mean	0.274107	0.275237	0.27334	0.27559

[9]:
```
# Draw the boxplot
sns.boxplot(data=df)
plt.ylabel('Error rate')
plt.title("Error Rate over 1000 Simulations")
plt.show()
```

When the Bayes decision boundary is linear, the average error rate of LDA is 27.41% for the training set, 27.52% for the testing set; the average error rate of QDA is 27.33% for the training set, 27.56% for the testing set. So QDA performs better than LDA on the training set, while LDA performs better on the the testing set.

1.2.2 3. 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?

On both the training set and the testing set, QDA would perform better because its flexibility helps it perform better in non-linear situations.

a. Repeat the following process 1000 times.

i. Simulate a dataset of 1000 observations with X1 , X2 Uniform(−1, +1). Y is a binary response variable defined by a Bayes decision boundary of f(X) = X1 + X12 + X2 + X2, where values 0 or greater are coded TRUE and values less than 0 or coded FALSE. Whereas your simulated Y is a function of X1 +X12 +X2 +X2 + where N(0,1). That is, your simulated Y is a function of the Bayes decision boundary plus some irreducible error.
ii. Randomly split your dataset into 70/30% training/test sets.
iii. Use the training dataset to estimate LDA and QDA models.
iv. Calculate each model's training and test error rate.

[10]:
```
def nonlinearsim(n): #take the number of simulations as parameter
        np.random.seed(n)
        #Simulate a dataset of 1000 observation
        X = np.random.uniform(-1,1,(2,1000))
```

```
    #Y is a binary response variable defined by a Bayes decision boundary␣
↪of
    #f(X) = X1 + X12 + X2 + X2, where values 0 or greater are coded TRUE␣
↪and values
    #less than 0 or coded FALSE. Whereas your simulated Y is a function of
    #X1 +X12 +X2 +X2 + where    N(0,1). That is, your simulated Y is a
    #function of the Bayes decision boundary plus some irreducible error.
    Y = (X[0,:] + X[0,:]**2 + X[1,:] + X[1,:]**2 + np.random.normal(size =␣
↪1000))>=0
    data=np.vstack((X, Y))
    seq = np.arange(1000)
    np.random.shuffle(seq)
    #Randomly split your dataset into 70/30% training/test sets
    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
```

[11]:
```
# 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=nonlinearsim(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
```

b. Summarize all the simulations' error rates and report the results in tabular and graphical form. Use this evidence to support your answer.

[12]:
```
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()
```

[12]:

	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

[13]:
```
# mean error rates
df.describe().loc[['mean']]
```

[13]:

	LDA_train	LDA_test	QDA_train	QDA_test
mean	0.272387	0.274833	0.25902	0.261623

[14]:
```
# Draw the boxplot
sns.boxplot(data=df)
plt.ylabel('Error rate')
plt.title("Error Rate over 1000 Simulations")
plt.show()
```

When the Bayes decision boundary is nonlinear, the average error rate of LDA is 27.24% for the training set, 27.48% for the testing set; the average error rate of QDA is 25.90% for the training set, 26.16% for the testing set. So QDA performs better than LDA on both the training set and the testing set.

1.2.3 4. 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?

As sample size n increases, the test error rate of both QDA and LDA would decrease, and QDA would always perform better, I would expect the test error rate of QDA relative to LDA to decrease, because QDA would be less affected by overfitting problem as sample size increases, and its flexibility allows it to fit data better than LDA.

a. Use the non-linear Bayes decision boundary approach from part (2) and vary n across your simulations (e.g., simulate 1000 times for n = c(1e02, 1e03, 1e04, 1e05).

[2]:
```
def nonlinearsim_new(n, m): #take the number of simulations and the number of
                                            #obesrvations as parameter
        np.random.seed(n)
        #Simulate a dataset of 1000 observation
        X = np.random.uniform(-1,1,(2,m))
        #Y is a binary response variable defined by a Bayes decision boundary␣
 ↪of
        #f(X) = X1 + X12 + X2 + X2, where values 0 or greater are coded TRUE␣
 ↪and values
```

```
        #less than 0 or coded FALSE. Whereas your simulated Y is a function of
        #X1 +X12 +X2 +X2 +  where    N(0,1). That is, your simulated Y is a␣
↪function of
        #the Bayes decision boundary plus some irreducible error.
        Y = (X[0,:] + X[0,:]**2 + X[1,:] + X[1,:]**2 + np.random.normal(size =␣
↪m))>=0
        data=np.vstack((X, Y))
        seq = np.arange(m)
        np.random.shuffle(seq)
        #Randomly split your dataset into 70/30% training/test sets
        train_idx = seq[:int(0.7*m)]
        test_idx = seq[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
```

[3]:
```
# 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= nonlinearsim_new(j,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()
```

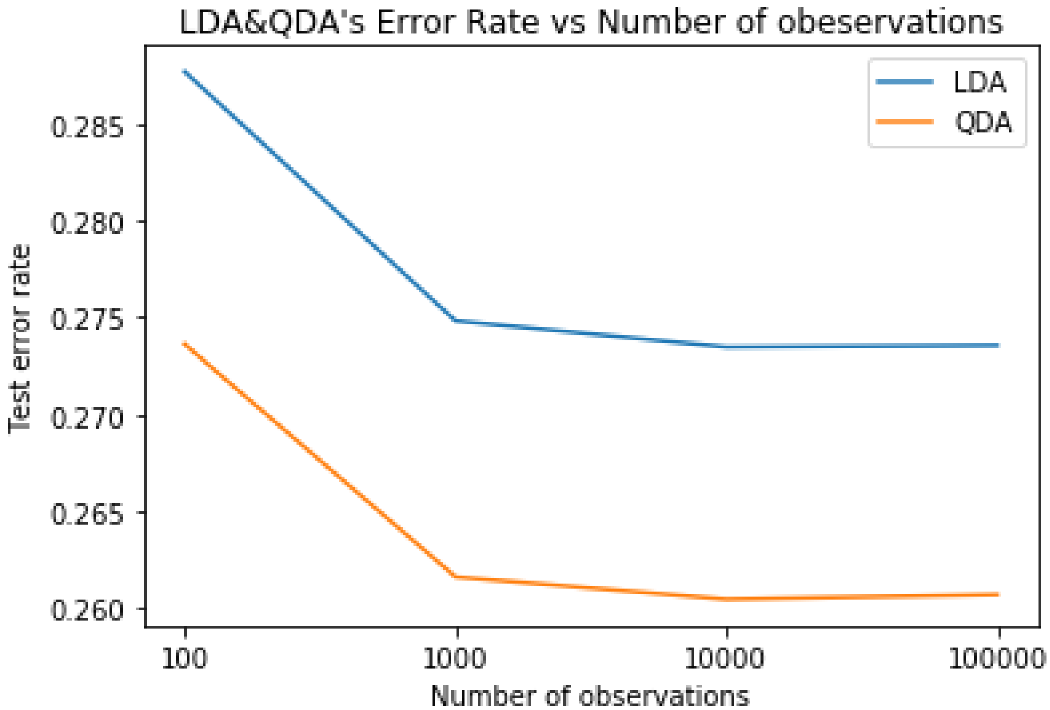
[14]:
```
dict = {'Sample Size':["100", "1000", "10000", "100000"],
        'LDA_error':lda_err,
        'QDA_error':qda_err}

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

[14]:

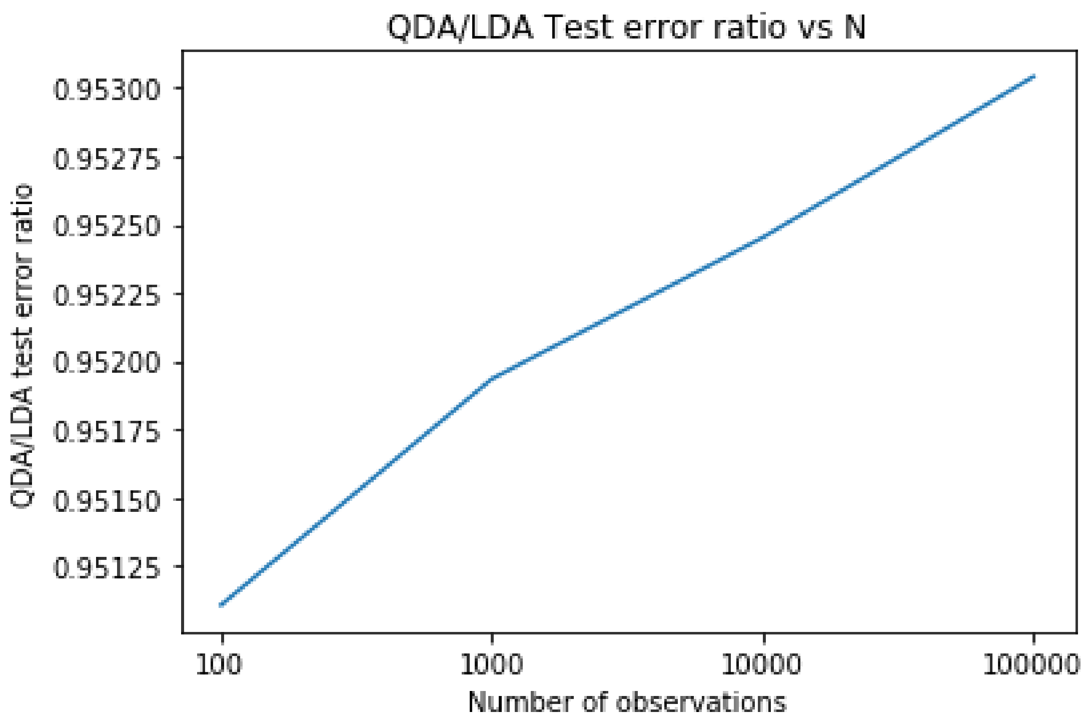
	Sample Size	LDA_error	QDA_error
0	100	0.287700	0.273633
1	1000	0.274833	0.261623
2	10000	0.273475	0.260472
3	100000	0.273557	0.260711

b. Plot the test error rate for the LDA and QDA models as it changes over all of these values of n. Use this graph to support your answer.

[5]:
```
# Graphical Interpretation
plt.plot([1,2,3,4], lda_err, label='LDA')
plt.plot([1,2,3,4], qda_err, label='QDA')
plt.legend()
plt.xticks([1,2,3,4], ['100', '1000', '10000', '100000'])
plt.xlabel('Number of observations')
plt.ylabel('Test error rate')
plt.title('LDA&QDA\'s Error Rate vs Number of obeservations')
plt.show()
```



[13]:
```
# Ratio Graphical Interpretation
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 N')
plt.show()
```



As the table and the graph show, both models predict better on the testing set as sample size increases, because large sample size allows the models to be closer to the true model. And QDA always perform better than LDA, because it performs better in catching non-linear patterns. However, as opposed to my former expectation, the test error rate of QDA relative to LDA actually increases. This could be explained by that when the sample size is large enough, both models fit data better and approach the real model, so no wonder their test error rate would become more similar as they both become more accurate. (or things might be different if I change the random seed)

1.3 Modeling voter turnout

1.3.1 5. Building several classifiers and comparing output.

[2]:
```
# import the data and remove the NANs
mh = pd.read_csv('mental_health.csv').dropna()
mh
```

[2]:

	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]
```

a. Split the data into a training and test set (70/30)

[3]:
```
np.random.seed(124)
seq = np.arange(mh.shape[0])
np.random.shuffle(seq)
train_idx = seq[:int(0.7*mh.shape[0])]
test_idx = seq[int(0.7*mh.shape[0]):]
train, test= mh.iloc[train_idx,:], mh.iloc[test_idx,:]
x_train = train.iloc[:,1:]
y_train = train['vote96']
x_test = test.iloc[:,1:]
y_test = test['vote96']
```

b. Using the training set and all important predictors, estimate the following models with vote96 as the response variable:

[4]:
```
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 settings)
nb = GaussianNB().fit(x_train, y_train)
models.append(('Naive Bayes', nb))

# v. K-nearest neighbors with K = 1,2,...,10 (that is, 10 separate models␣
↪varying K) and
#Euclidean distance metrics
m_knn = []
for k in range(1,11):
    knn= KNeighborsClassifier(n_neighbors=k).fit(x_train, y_train)
```

```
    models.append(('KNN_{}'.format(k), knn))
```

```
/anaconda3/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:433:
FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a
solver to silence this warning.
  FutureWarning)
```

c. Using the test set, calculate the following model performance metrics:

[5]:
```
# 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])
```

i. Error rate

[6]:
```
# calculate the error rates of the models
error_rates = [ (1 - accuracy_score(y_test, pred)) for pred in preds]

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

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

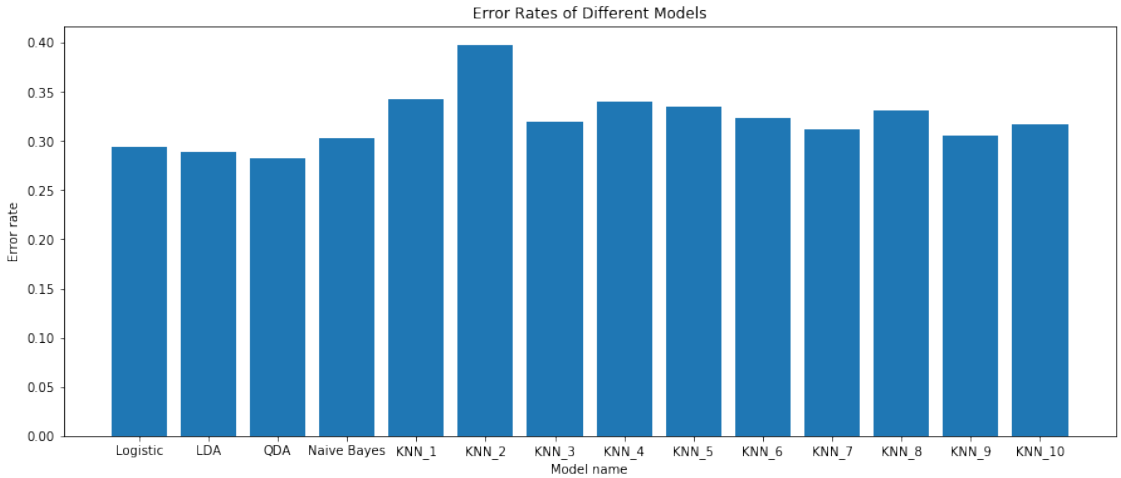
[6]:

	Model Name	Error rate
0	Logistic	0.294286
1	LDA	0.288571
2	QDA	0.282857
3	Naive Bayes	0.302857
4	KNN_1	0.342857
5	KNN_2	0.397143
6	KNN_3	0.320000
7	KNN_4	0.340000
8	KNN_5	0.334286
9	KNN_6	0.322857
10	KNN_7	0.311429
11	KNN_8	0.331429
12	KNN_9	0.305714
13	KNN_10	0.317143

[7]:
```
# Graphical Interpretation
plt.figure(figsize=(15,6))
```

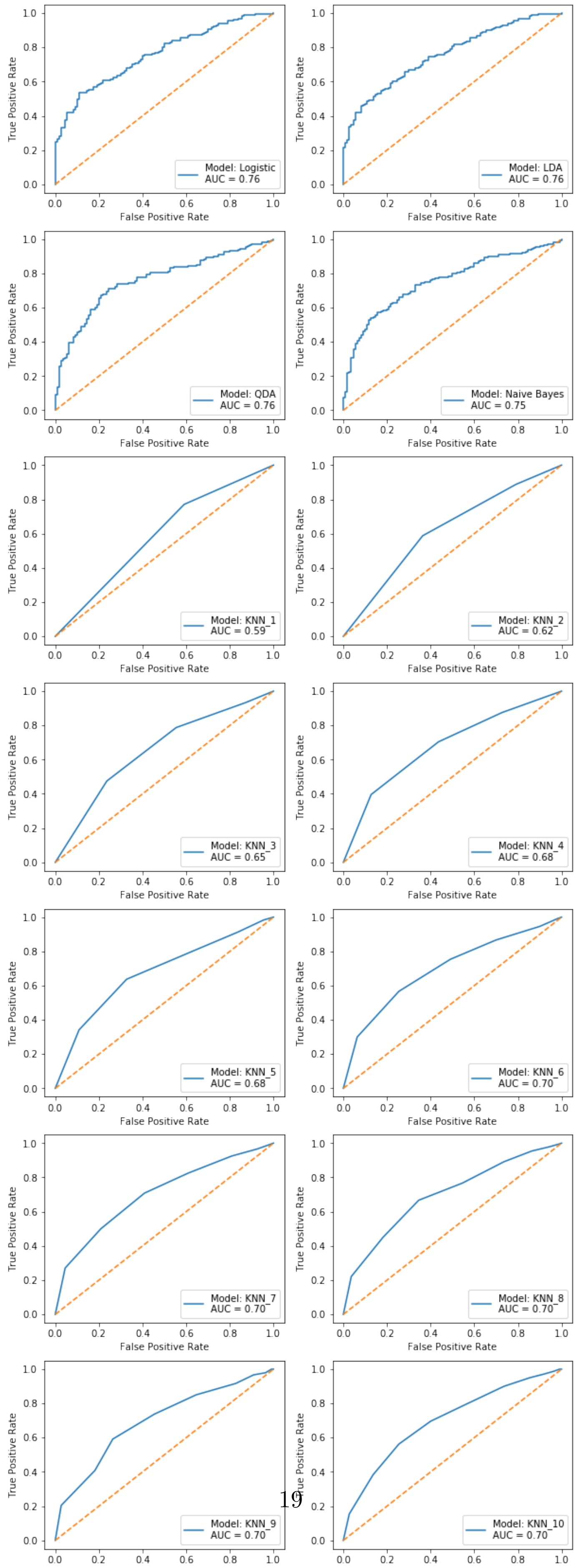
```python
plt.bar(range(len(names)), error_rates)
plt.xticks(range(len(names)),names)
plt.xlabel('Model name')
plt.ylabel('Error rate')
plt.title('Error Rates of Different Models')
```

[7]: Text(0.5, 1.0, 'Error Rates of Different Models')



ii. ROC curve(s) / Area under the curve (AUC)

[8]:
```python
rocs = []
aucs = []
for prob in probs:
    fpr, tpr, _ = roc_curve(y_test, prob)
    rocs.append((fpr, tpr))
    aucs.append(auc(fpr, tpr))
#Plot
fig = plt.figure(figsize=(10, 30))
for i, roc in enumerate(rocs):
    ax = fig.add_subplot(7, 2, i+1)
    ax.plot(roc[0], roc[1] , label='Model: '+ names[i] +
            '\nAUC = %0.2f' % aucs[i])
    ax.plot([0,1], [0,1], linestyle = 'dashed')
    ax.legend(loc = 'lower right')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
```


True Positive Rate
False Positive Rate
Model: Logistic
AUC = 0.76
Model: LDA
AUC = 0.76
Model: QDA
AUC = 0.76
Model: Naive Bayes
AUC = 0.75
Model: KNN_1
AUC = 0.59
Model: KNN_2
AUC = 0.62
Model: KNN_3
AUC = 0.65
Model: KNN_4
AUC = 0.68
Model: KNN_5
AUC = 0.68
Model: KNN_6
AUC = 0.70
Model: KNN_7
AUC = 0.70
Model: KNN_8
AUC = 0.70
Model: KNN_9
AUC = 0.70
Model: KNN_10
AUC = 0.70

d. Which model performs the best? Be sure to define what you mean by "best" and identify supporting evidence to support your conclusion(s). From my definition, the "best" model should be the one with the highest accuracy on the testing data, which means it should be featured with low error rate and high AUC. Therefore, my best-performance model is the QDA model, as it has the lowest error rate of 28.3%, and the highest AUC of 0.76.