```
import random as rd
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, roc_curve, auc
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis,
QuadraticDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
import seaborn as sns

import warnings
warnings.filterwarnings("ignore")
```

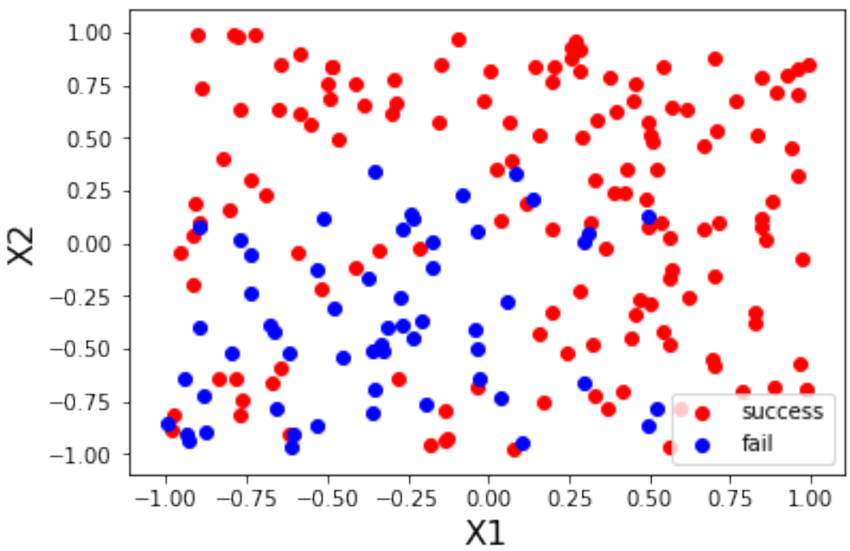
Question 1: The Bayes Classifier

```
rd.seed(1234) # set a random seed

X1 = np.random.uniform(-1, 1, 200)
X2 = np.random.uniform(-1, 1, 200) # produce two RVs in uniform distribution

eps = np.random.normal(0, 0.5, 200) # produce error in normal distribution

Y = X1 + X2 + X1**2 + X2**2 + eps
```

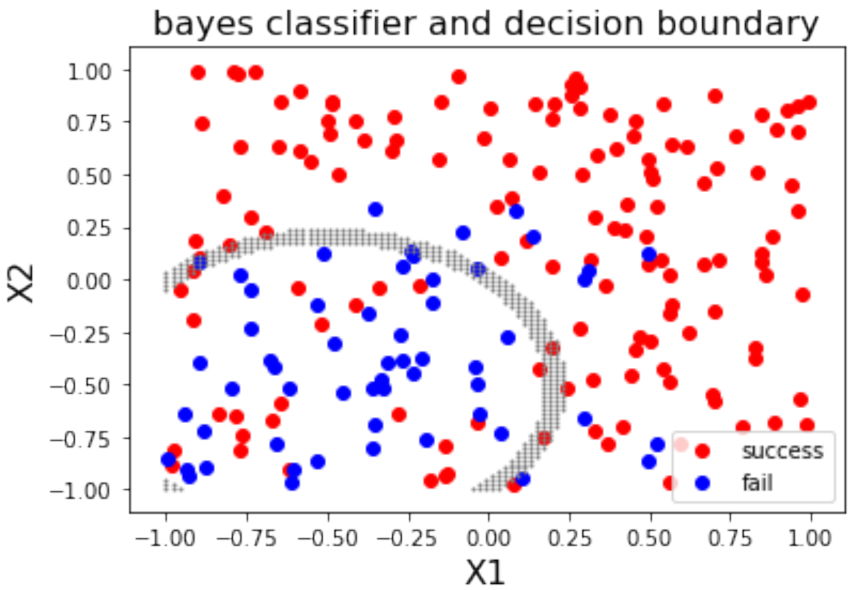
According to the question, Y is defined in terms of log odds, which is $Y = log(\frac{p}{1-p})$ and p is the probability of success. Next I will do the inverse calculation:

$$p = \frac{e^Y}{e^Y + 1}$$

```
p = np.exp(Y)/(np.exp(Y)+1) # calculate probability of success

df = pd.DataFrame({'Probability': p, 'X1': X1, 'X2': X2})

# Plot (cited from https://stackoverflow.com/questions/21654635/scatter-plots-in-pandas-pyplot-how-to-plot-by-category)
plt.scatter(X1[p > 0.5],X2[p > 0.5],c='r',label = 'success')
plt.scatter(X1[p <= 0.5],X2[p <= 0.5],c='b',label = 'fail')
plt.xlabel('X1',size = 16)
plt.ylabel('X2',size = 16)
plt.legend()
plt.show()
```




```
a = np.linspace(-1,1,100)
b = np.linspace(-1,1,100)
ab = np.array([a]*len(b)).reshape((len(a),len(b)))
c = ab.T
land = ab + c + ab**2 + c**2

plt.scatter(X1[p > 0.5], X2[p > 0.5], c='r', label = 'success')
plt.scatter(X1[p <= 0.5], X2[p <= 0.5], c='b', label = 'fail')

for i in range(len(a)):
    for j in range(len(b)):
        if land[i,j] <= 0.03 and land[i,j] >= -0.03:
            plt.scatter(a[i], b[j], s = 3, color = 'gray')
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Bayes Classifier and Decision Boundary')
plt.legend()
plt.show()
```



Question 2

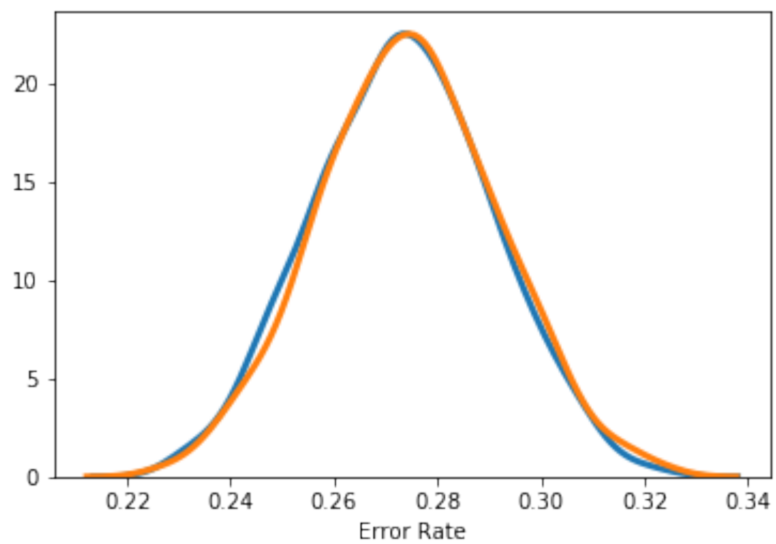
Theoretically, if the bayes decision boundary is linear, we would expect LDA to perform better than QDA on both training set and testing set since LDA relies on the assumption that predictors of each class have common covariance.

```
i = 1
train_error = {'LDA':[], 'QDA':[]}
test_error = {'LDA':[], 'QDA':[]}

while i < 1001:
    X1 = np.random.uniform(-1, 1, 1000)
    X2 = np.random.uniform(-1, 1, 1000)
    eps = np.random.normal(0, 1, 1000)
    y = X1 + X2 + eps
    Y = y >= 0
    X = pd.DataFrame({'X1':X1, 'X2':X2})
    X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,
random_state=123)
    model_LDA = LinearDiscriminantAnalysis().fit(X_train, y_train)
    model_LDA_test_pred = model_LDA.predict(X_test)
    model_LDA_train_pred = model_LDA.predict(X_train)
    model_LDA_test_err = 1 - accuracy_score(y_test, model_LDA_test_pred)
    model_LDA_train_err = 1 - accuracy_score(y_train, model_LDA_train_pred)
    train_error['LDA'].append(model_LDA_train_err)
    test_error['LDA'].append(model_LDA_test_err)

    model_QDA = QuadraticDiscriminantAnalysis().fit(X_train, y_train)
    model_QDA_test_pred = model_QDA.predict(X_test)
    model_QDA_train_pred = model_QDA.predict(X_train)
    model_QDA_test_err = 1 - accuracy_score(y_test, model_QDA_test_pred)
    model_QDA_train_err = 1 - accuracy_score(y_train, model_QDA_train_pred)
```

```
    train_error['QDA'].append(model_QDA_train_err)
    test_error['QDA'].append(model_QDA_test_err)

    i += 1

# tabular
df = pd.DataFrame({'QDA':[np.mean(train_error['QDA']),
np.mean(test_error['QDA'])],
                  'LDA':[np.mean(train_error['LDA']),
np.mean(test_error['LDA'])]},
                 index = ['Training Error Rate', 'Testing Error Rate'])
print(df)

# graph
## histogram of training error
df = pd.DataFrame({'QDA': train_error['QDA'], 'LDA': train_error['LDA']})
sns.distplot(df['QDA'], hist = False, kde = True, kde_kws = {'shade': False,
'linewidth': 3})
sns.distplot(df['LDA'], hist = False, kde = True, kde_kws = {'shade': False,
'linewidth': 3})
plt.xlabel('Error Rate')
```

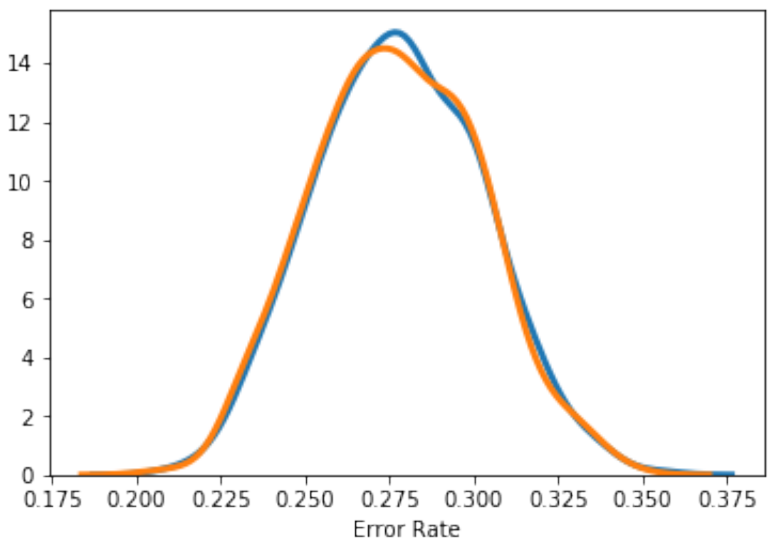
```
                          QDA       LDA
Training Error Rate  0.273356  0.274493
Testing Error Rate   0.277943  0.277220
```

```
Text(0.5, 0, 'Error Rate')
```



```
## histogram of testing error
df = pd.DataFrame({'QDA': test_error['QDA'], 'LDA': test_error['LDA']})
sns.distplot(df['QDA'], hist = False, kde = True, kde_kws = {'shade': False,
'linewidth': 3})
sns.distplot(df['LDA'], hist = False, kde = True, kde_kws = {'shade': False,
'linewidth': 3})
plt.xlabel('Error Rate')
```

```
Text(0.5, 0, 'Error Rate')
```



Based on the table of mean error and histogram of both training error and testing error, I can conclude that the performance of LDA and QDA are similar.

Question 3

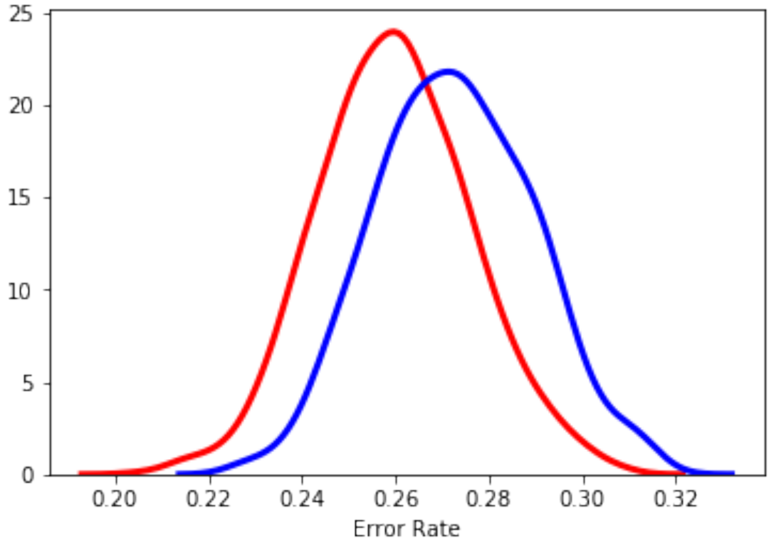
Theoretically, if the bayes decision boundary is non-linear, we would expect QDA to perform better than LDA on both training set and testing set since LDA relies on the assumption that predictors of each class have different covariance.

```
i = 1
train_error = {'LDA':[], 'QDA':[]}
test_error = {'LDA':[], 'QDA':[]}

while i < 1001:
    X1 = np.random.uniform(-1, 1, 1000)
    X2 = np.random.uniform(-1, 1, 1000)
    eps = np.random.normal(0, 1, 1000)
```

```
    y = X1 + X2 + X1**2 + X2**2 + eps
    Y = y >= 0
    X = pd.DataFrame({'X1':X1, 'X2':X2})
    X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,
random_state=123)
    model_LDA = LinearDiscriminantAnalysis().fit(X_train, y_train)
    model_LDA_test_pred = model_LDA.predict(X_test)
    model_LDA_train_pred = model_LDA.predict(X_train)
    model_LDA_test_err = 1 - accuracy_score(y_test, model_LDA_test_pred)
    model_LDA_train_err = 1 - accuracy_score(y_train, model_LDA_train_pred)
    train_error['LDA'].append(model_LDA_train_err)
    test_error['LDA'].append(model_LDA_test_err)

    model_QDA = QuadraticDiscriminantAnalysis().fit(X_train, y_train)
    model_QDA_test_pred = model_QDA.predict(X_test)
    model_QDA_train_pred = model_QDA.predict(X_train)
    model_QDA_test_err = 1 - accuracy_score(y_test, model_QDA_test_pred)
    model_QDA_train_err = 1 - accuracy_score(y_train, model_QDA_train_pred)
    train_error['QDA'].append(model_QDA_train_err)
    test_error['QDA'].append(model_QDA_test_err)

    i += 1

# tabular
df = pd.DataFrame({'QDA':[np.mean(train_error['QDA']),
np.mean(test_error['QDA'])],
                  'LDA':[np.mean(train_error['LDA']),
np.mean(test_error['LDA'])]},
                 index = ['Training Error Rate', 'Testing Error Rate'])
print(df)

# graph
df = pd.DataFrame({'QDA': train_error['QDA'], 'LDA': train_error['LDA']})
sns.distplot(df['QDA'], hist = False, kde = True, kde_kws = {'shade': False,
'linewidth': 3}, color = 'red')
sns.distplot(df['LDA'], hist = False, kde = True, kde_kws = {'shade': False,
'linewidth': 3}, color = 'blue')
plt.xlabel('Error Rate')
```

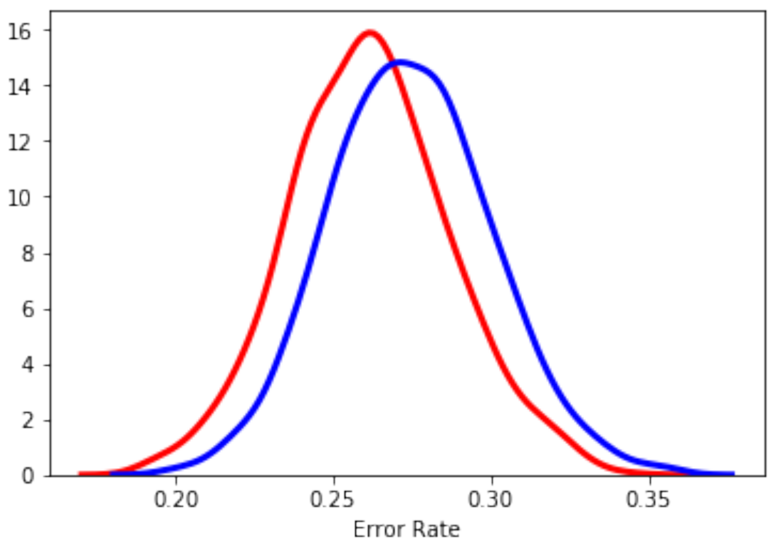
```
                          QDA       LDA
Training Error Rate  0.259717  0.272660
Testing Error Rate   0.261560  0.274503
```

```
Text(0.5, 0, 'Error Rate')
```




```
df = pd.DataFrame({'QDA': test_error['QDA'], 'LDA': test_error['LDA']})
sns.distplot(df['QDA'], hist = False, kde = True, kde_kws = {'shade': False,
'linewidth': 3}, color = 'red')
sns.distplot(df['LDA'], hist = False, kde = True, kde_kws = {'shade': False,
'linewidth': 3}, color = 'blue')
plt.xlabel('Error Rate')
```

```
Text(0.5, 0, 'Error Rate')
```



Based on the table of mean error rate and two figures, I can conclude that the performace of QDA is better than LDA.

Question 4

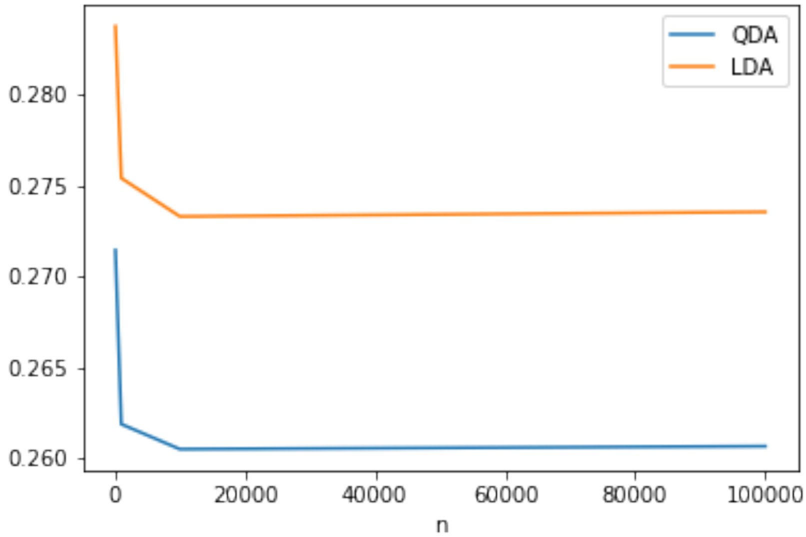
In general, the performance of QDA relative to LDA will improve as n increases since QDA is more complex than LDA and large sample can avoid overfitting. Thus, QDA can fit the data more accurately.

```
n = [100, 1000, 10000, 100000]
train = {'LDA':[], 'QDA':[]}
test = {'LDA':[], 'QDA':[]}
for n0 in n:
    i = 1
    train_error = {'LDA':[], 'QDA':[]}
    test_error = {'LDA':[], 'QDA':[]}

    while i < 1001:
        X1 = np.random.uniform(-1, 1, n0)
        X2 = np.random.uniform(-1, 1, n0)
        eps = np.random.normal(0, 1, n0)
        y = X1 + X2 + X1**2 + X2**2 + eps
        Y = y >= 0
        X = pd.DataFrame({'X1':X1, 'X2':X2})
        X_train, X_test, y_train, y_test = train_test_split(X, Y,
test_size=0.3, random_state=123)
        model_LDA = LinearDiscriminantAnalysis().fit(X_train, y_train)
        model_LDA_test_pred = model_LDA.predict(X_test)
        model_LDA_train_pred = model_LDA.predict(X_train)
        model_LDA_test_err = 1 - accuracy_score(y_test, model_LDA_test_pred)
        model_LDA_train_err = 1 - accuracy_score(y_train,
model_LDA_train_pred)
        train_error['LDA'].append(model_LDA_train_err)
        test_error['LDA'].append(model_LDA_test_err)

        model_QDA = QuadraticDiscriminantAnalysis().fit(X_train, y_train)
        model_QDA_test_pred = model_QDA.predict(X_test)
        model_QDA_train_pred = model_QDA.predict(X_train)
        model_QDA_test_err = 1 - accuracy_score(y_test, model_QDA_test_pred)
        model_QDA_train_err = 1 - accuracy_score(y_train,
model_QDA_train_pred)
        train_error['QDA'].append(model_QDA_train_err)
        test_error['QDA'].append(model_QDA_test_err)

        i += 1
    train['QDA'].append(np.mean(train_error['QDA']));
train['LDA'].append(np.mean(train_error['LDA']))
    test['QDA'].append(np.mean(test_error['QDA']));
test['LDA'].append(np.mean(test_error['LDA']))

train_df = pd.DataFrame({'n': n, 'QDA': train['QDA'], 'LDA': train['LDA']})
```

```
test_df = pd.DataFrame({'n': n, 'QDA': test['QDA'], 'LDA': test['LDA']})
```

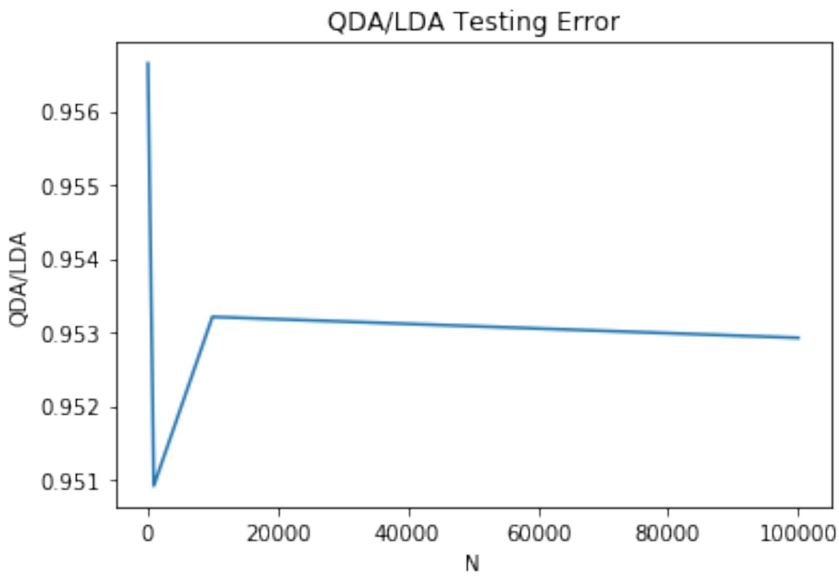
```
test_df = test_df.set_index('n')
test_df.plot.line()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a2705a710>
```




```
ratio = np.array(test['QDA'])/np.array(test['LDA'])
x = np.array([100, 1000, 10000, 100000])
plt.plot(x, ratio)
plt.ylabel('QDA/LDA')
plt.xlabel('N')
plt.title('QDA/LDA Testing Error')
```

```
Text(0.5, 1.0, 'QDA/LDA Testing Error')
```



The test error rate of QDA relative to LDA will decline before 1000 and increase as n becomes close to 10000 but finally decrease as n moves toward 100000. Though the trend is not monotonous, the performace of QDA is better than LDA as n becomes larger.

Question 5: Modeling voter turnout

```
data = pd.read_csv('problem-set-2/mental_health.csv') # load data
data.isna().apply(sum) # count on NA value
```

```
vote96          219
mhealth_sum    1418
age               4
educ             12
black             0
female            0
married           1
inc10           329
dtype: int64
```

Through the missing value statistic, I find that some vote96 values are missing, which is inconvininet when building a prediction model. Thus, I should remove observations with missing voting value.

```
# handle NA in vote96
index = pd.notna(data['vote96'])
data = data.loc[index]
data.isna().apply(sum)/len(data)
```

```
vote96         0.000000
mhealth_sum    0.494068
age            0.001531
educ           0.003062
black          0.000000
female         0.000000
married        0.000000
inc10          0.112132
dtype: float64
```

After removing those observations, there are still some missing value in the data. For the variable named mhealth_sum, I find that nearly half of the variable is missing. Thus, using techniques such as imputation to handle this situation would be inappropriate and I will remove those observations. However, I will not delete other observations which has some missing values and I will fill the NA value instead. Then, I split the original data into training set and testing set.

```
# handle NA in other variables
index = pd.notna(data['mhealth_sum'])
data = data.loc[index]
data.isna().apply(sum)/len(data)

data['age'] = data['age'].fillna(data['age'].median())
data['educ'] = data['educ'].fillna(data['educ'].mode()[0])
data['inc10'] = data['inc10'].fillna(data['inc10'].median())

# split the dataset
y = data['vote96']
X = data.loc[:, data.columns != 'vote96']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=123)
```

Logistic regression

```
# logitic regression model
model_1 = LogisticRegression().fit(X_train, y_train)
model_1_pred = model_1.predict(X_test)
model_1_errorrate = 1 - accuracy_score(y_test, model_1_pred)
model_1_roc = roc_curve(y_test, model_1_pred, pos_label=1)
fpr, tpr, thresholds = model_1_roc
model_1_auc = auc(fpr, tpr)
print('     Error Rate           AUC')
print(model_1_errorrate, model_1_auc)
```

```
    Error Rate            AUC
0.292191435768262 0.6051004420569638
```

LDA

```
# LDA model
model_2 = LinearDiscriminantAnalysis().fit(X_train, y_train)
model_2_pred = model_2.predict(X_test)
model_2_errorrate = 1 - accuracy_score(y_test, model_2_pred)
model_2_roc = roc_curve(y_test, model_2_pred, pos_label=1)
fpr, tpr, thresholds = model_2_roc
model_2_auc = auc(fpr, tpr)
print('    Error Rate           AUC')
print(model_2_errorrate, model_2_auc)
```

```
    Error Rate            AUC
0.2821158690176322 0.6229785686307425
```

QDA

```
# QDA model
model_3 = QuadraticDiscriminantAnalysis().fit(X_train, y_train)
model_3_pred = model_3.predict(X_test)
model_3_errorrate = 1 - accuracy_score(y_test, model_3_pred)
model_3_roc = roc_curve(y_test, model_3_pred, pos_label=1)
fpr, tpr, thresholds = model_3_roc
model_3_auc = auc(fpr, tpr)
print('    Error Rate           AUC')
print(model_3_errorrate, model_3_auc)
```

```
    Error Rate            AUC
0.3148614609571788 0.631735773040121
```

Naive Bayes

```
# NB
model_4 = GaussianNB().fit(X_train, y_train)
model_4_pred = model_4.predict(X_test)
model_4_errorrate = 1 - accuracy_score(y_test, model_4_pred)
model_4_roc = roc_curve(y_test, model_4_pred, pos_label=1)
fpr, tpr, thresholds = model_4_roc
model_4_auc = auc(fpr, tpr)
print('    Error Rate          AUC')
print(model_4_errorrate, model_4_auc)
```

```
    Error Rate          AUC
0.3047858942065491 0.6343797213362431
```

KNN

```
# KNN
print('          Error Rate          AUC')
for i in range(1,11):
    model = KNeighborsClassifier(n_neighbors=i).fit(X_train, y_train)
    model_pred = model.predict(X_test)
    model_errorrate = 1 - accuracy_score(y_test, model_pred)
    model_roc = roc_curve(y_test, model_pred, pos_label=1)
    fpr, tpr, thresholds = model_roc
    model_auc = auc(fpr, tpr)
    print('K = {} {} {}'.format(i, model_errorrate, model_auc))
```

```
          Error Rate          AUC
K = 1 0.36020151133501255 0.5952940518157909
K = 2 0.38035264483627207 0.615396452352974
K = 3 0.3123425692695214 0.6218174696435566
K = 4 0.35012594458438284 0.6131721783895696
K = 5 0.309823677581864 0.6186699121481729
K = 6 0.309823677581864 0.6372894633764199
K = 7 0.3148614609571788 0.6063454759106933
K = 8 0.309823677581864 0.6339040904258296
K = 9 0.3022670025188917 0.6176906720384981
K = 10 0.3022670025188917 0.6329248503161546
```

Based on the error rate and auc value, I think the best model is LDA. This is because the error rate of this model is the smallest among those candidates and the AUC is also relatively large.