Project 4 - Data-Driven Classification

Austin Phillips
ECE 4850
UVU Electrical and Computer Engineering
Orem, Utah
austin.phillips@uvu.edu

Angel Rodriguez
ECE 4850
UVU Electrical and Computer Engineering
Orem, Utah
angel.rodriguez@uvu.edu

***Abstract*—Varying data-driven classification models to classify data will be explored including linear regression, quadratic regression, 1-nearest neighbor, 5-nearest neighbor, and 15-nearest neighbor. The performances on a set of data will be empirically characterized.**

***Index Terms*—linear regression (LR), quadratic regression (QR), k-nearest neighbors (KNN), population, training data, testing data**

I. Introduction to Classification

The data are produced according to a model described later. Figure 1 shows the training data for d = 2 dimensional data. The 'x' data is for class 0; the 'o' data is for class 1. This training data is in the file classasgntrain1.dat. The first two columns of this (ascii) file are the x and y coordinates of N = 1000 points of the class 0 data; the second two columns of the data are the x and y coordinates of the class 1 data. Instances of data are generated by calling the MATLAB function gendat2, as follows:

x = gendat2(class, N);

The MATLAB function was called for class 0 and class 1 for N = 5000 and stored into the file outputFile.dat to be used for test data later.

Let N_0 be the number of points of training data from class 0, and let N_1 be the number of points of training data from class 1. Let N = $N_0 + N_1$ be the total number of training data. Let $\mathbf{x}_i$ denote the coordinates of a training vector (thinking of this as a column vector), and let $\mathbf{y}_i$ denote the corresponding class value. That is, either $y_i = 0$ or $y_i = 1$, depending on the class the point comes from. Then the set of data $(\mathbf{x}_i, \mathbf{y}_i)$,i = 1,2,...,N is the total set of training data. For some of the discussions below, we assume that the data are ordered so that the first N_0 training points are from class 0 and the next N_1 training points are from class 1.

II. Developing Linear Classification Regions Using Training Population

For some classification problems, it suffices to provide a straight line (or plane, in higher dimensions) dividing the two classes. Linear regression is one means of determining such a dividing line (or plane).

Let $\mathbf{x} = [x_1, x_2, ..., x_d]^T$

be a d-dimensional vector. A linear function of **x** is 1

The term β_0 is to change the "y-intercept" of the line, to account for non-zero mean data. In the usual regression problem, each point $\mathbf{x}_i$ has associated with it some value y_i. The parameters of the line $\beta = [\beta_1, \beta_2, ..., \beta_d]^T$ are chosen in such a way to get the best match possible over all training points. It is desired to minimize Equation 2 where x_{ij} is the jth coordinate of the training data vector $\mathbf{x}_i$, and where RSS stands for "residual sum of squares" (that is, the sum of the squares of the "residuals" or errors). This can be written in more convenient form as follows. First, let augmented matrix **X** be defined as in Equation 3. That is, stack a 1 on top of the column vector $\mathbf{x}_i$. Then the linear function can be written as shown in Equation 4. Now let y be the stack of data output values for all the data shown in 5 and **X** be the N × (d + 1) matrix formed by stacking the data as rows (including the 1) shown in 6. It turns out that $\hat{\boldsymbol{\beta}}$ can be found via Equation 7.

For a binary classification problem, the **y** data are modified somewhat. Next, form the N × 2 matrix Y whose 2 columns indicate respective class membership of the given data point. That is, form the X matrix (N × (d + 1)) and the corresponding Y matrix (N × 2) as shown in Equation 8. The matrix Y is called the indicator response matrix. Then find the estimated coefficient matrix $hat\beta$ via Equation 9. The estimated indicator response matrix is Equation 10. To use the classifier after training, suppose that the vector desired to classify is **x**. Form the indicator response vector displayed in 11. This is a vector with K = 2 elements in it, $\hat{y}^T = [\hat{y}_1, \hat{y}_2]$. The estimated class $\hat{k}$ corresponds to the column with the largest value and is defined in 12

$$f(\mathbf{x}) = \beta_0 + \sum_{j=1}^{d} (x_j \beta_j) \tag{1}$$

$$\mathcal{RSS}(\mathbf{B}) = \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{d} x_{ij}\beta_j)^2 \tag{2}$$

$$\mathbf{X} = \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} \tag{3}$$

$$f(\mathbf{x}) = \boldsymbol{\beta}^T \mathbf{X} \tag{4}$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_N \end{bmatrix} \tag{5}$$

$$X = \begin{bmatrix} 1 & x_1^T \\ 1 & x_2^T \\ \vdots & \vdots \\ 1 & x_N^T \end{bmatrix} \tag{6}$$

$$\hat{\boldsymbol{\beta}} = (X^T X)^{-1} X^T \mathbf{y} \tag{7}$$

$$X = \begin{bmatrix} 1 & x_1^T \\ 1 & x_2^T \\ \vdots & \vdots \\ 1 & x_{N_0}^T \\ 1 & x_{N_{0+1}}^T \\ 1 & x_{N_{0+2}}^T \\ \vdots & \vdots \\ 1 & x_N^T \end{bmatrix} \quad Y = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \end{bmatrix} \tag{8}$$

$$\hat{\beta} = (X^T X)^{-1} X^T Y \tag{9}$$

$$\hat{Y} = X^T \hat{\beta} \tag{10}$$

$$\hat{y} = \begin{bmatrix} 1 & \mathbf{x}^T \end{bmatrix} \hat{\beta} \tag{11}$$

$$\hat{k} = arg_{k \epsilon \{0,1,\ldots,K-1\}} \hat{\boldsymbol{y}}_{\boldsymbol{k}} \tag{12}$$

III. Adding Test Population to Linear Classification Regions

Using the Python code depicted in Section VIII, the sample population (training data) was plotted as shown in Figure 1. Then, the classification regions were added as well as the test data from outputFile.dat in the same plot where green represents class 0 and red represents class 1 as shown in Figure 2. Here, it can be observed that the linear classifications are okay, but have room for improvement since a significant portion of the red test data is within the green region. The same can be observed within the training data plot since the red points seem to go into what one would predict would be within the green classification region. This confirms that the test data was generated in a very similar manner to however the training data was generated and that linear regression is doing what it is meant to do.

IV. Implementation in Python

Different methods of classifcation were accomplished with the following functions:

1) $linear\_regression(x0, x1, y)$ - linear regression function where the x0 vector is the property representing the x-coordinate of the population, the x1 vector is the property representing the y-coordinate of the population, and the y vector states the class which the point is in. Calculates **X** which is used to generate β and returns it.
2) $least\_squares\_est(x0, x1, beta)$ - guesstimating an output given an input and a linear regression where the x0 vector is the property representing the x-coordinate of the population, the x1 vector is the property representing the y-coordinate of the population, and beta is the returned variable from $linear\_regression(x0, x1, y)$:. Calculates **X** which is used to generate $\hat{y}$ and returns it.
3) $gen\_2D\_array(N, upper, lower)$ - returns values used to create square matrix for creating classification regions where N is the number of points, upper is the upper boundary of both the x and y coordinates, and lower is the lower boundary of both the x and y coordinates.
4) $quadratic\_regression(x0, x1, y)$ - similar to $linear\_regression(x0, x1, y)$ except **X** includes $[1, x_1, x_2, x_1^2, x_1x_2, x_2^2]$
5) $quad\_least\_squares_est(x0, x1, beta)$ - similar to $least\_squares\_est(x0, x1, beta)$ except **X** includes $[1, x_1, x_2, x_1^2, x_1x_2, x_2^2]$
6) $get\_distance(x1, y1, x2, y2)$ - calculates the distance between two points where x1/y1 correspond to first coordinate and x2/y2 correspond to second coordinate.
7) $nearest\_neighbor\_var(unknown, known, num_nearest)$ - calculates nearest number where $num_nearest$ is used to specify the k-value in KNN (1, 5, or 15 will be used).
8) $draw\_setup(self)$ - plots the sample/given data in choices of 2-dimensional or 3-dimensional graphs.

V. Implementing Quadratic Classification Regions

The classifier function is described in Equation 13 where the second summation introduces the quadratic dependency. Stack the information as follows. Let $\mathbf{x}_i = [x_1, x_2]^T$ training data point. Form a row of the data matrix **X** as shown in Equation 14, then form the **X** matrix as the stack of such rows for all training data. The $\hat{\beta}$ matrix has K columns, each of the form as shown in Equation 15, So that there are six unknowns. Given the **Y** matrix as before, the problem has the same structure as before. The least-squares estimate of $\hat{\beta}$ is Equation 16. Similarly repeating the process of linear regression the results are displayed in Figure 3

$$f(\mathbf{x}) = \beta_0 + \sum_{j=1}^{d} x_i \beta_i + \sum_{ij} \beta_{ij} x_j x_j \tag{13}$$

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1 x_2 & x_2^2 \end{bmatrix} \tag{14}$$

$$\hat{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_1 1 \\ \beta_1 2 \\ \beta_2 1 \end{bmatrix} \tag{15}$$

$$\hat{\beta} = (X^T X)^{-1} X^T Y \tag{16}$$

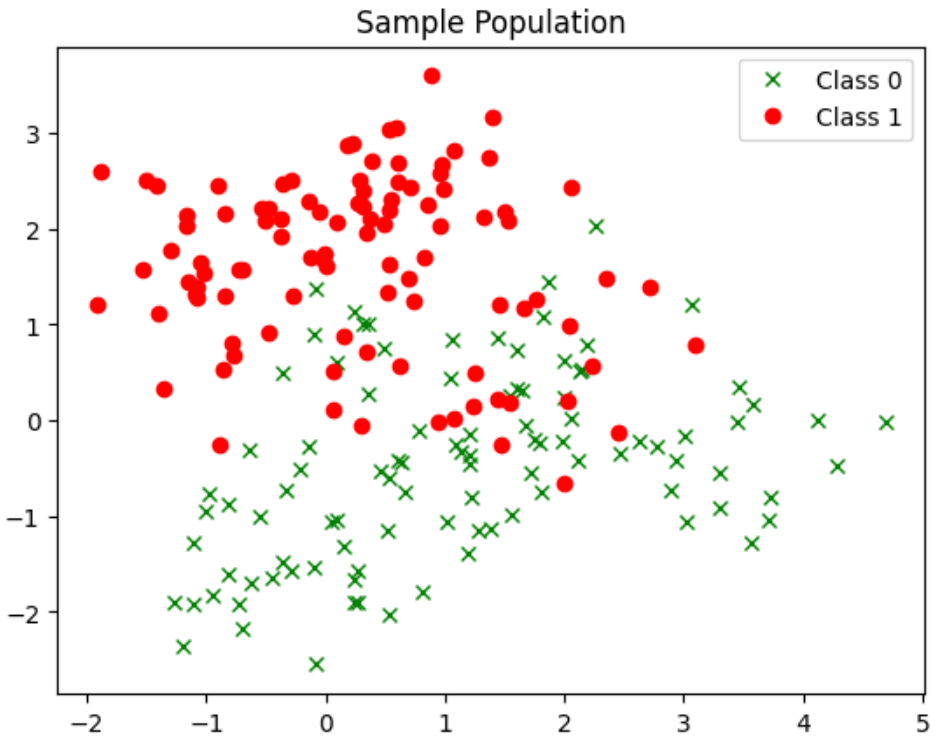

Fig. 1. Scatter plot of training data

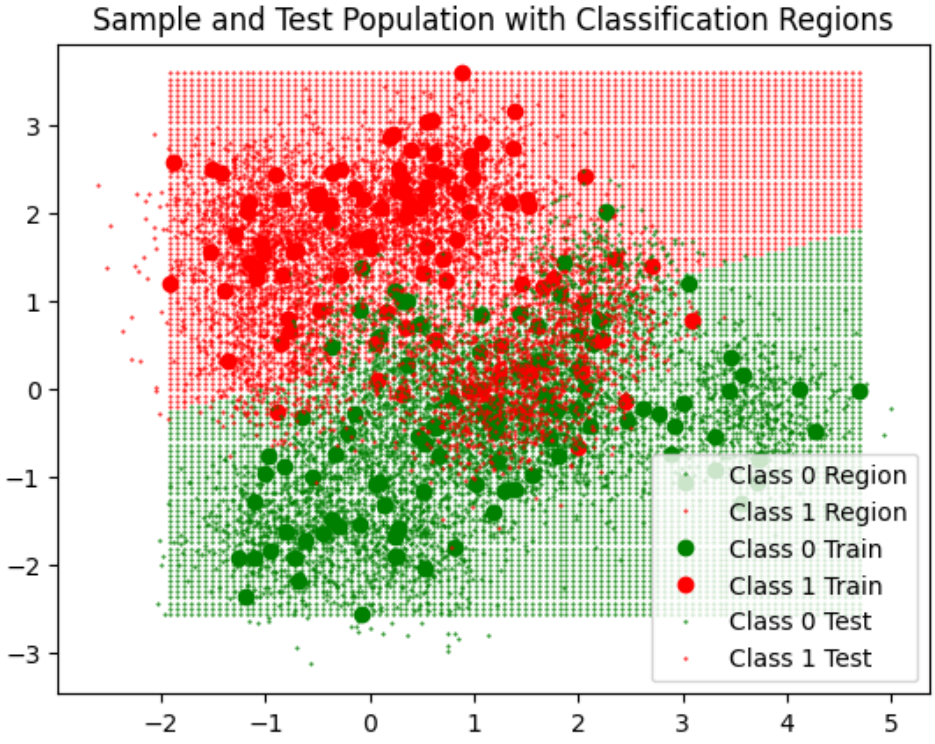

Fig. 2. Scatter plot of linear classification regions with training and testing data overlapped

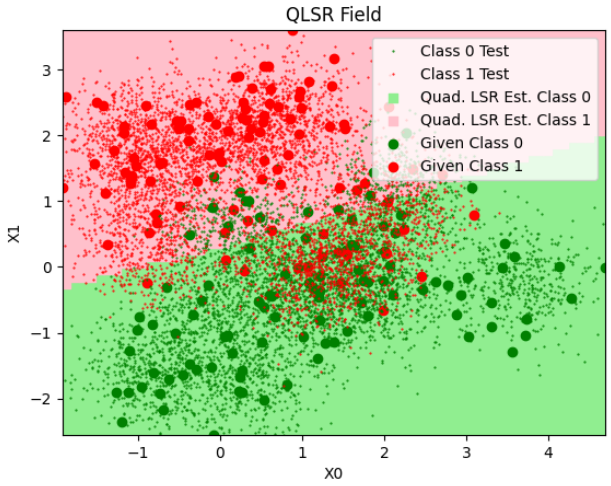

Fig. 3. Scatter plot of quadratic classification regions with training and testing data overlapped

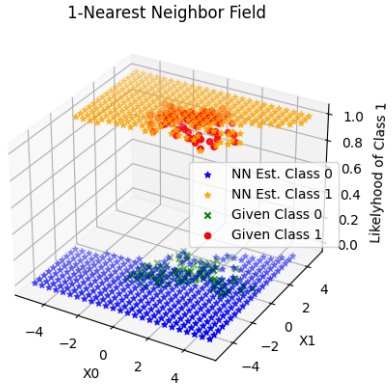

Fig. 4. Scatter plot showing the results of 1-Nearest Neighbor on points around the training data along with training data

VI. K-Nearest Neighbors (KNN)

The k-nearest neighbor classifier operates on the principle that you start to look like the people that you hang around with. That is, the classification of a vector **x** should be represented by classifications of neighbors near to it. Let $N_k(\mathbf{x})$ be the set of the k training vectors x_i nearest to the point **x**. Each training input x_i has a corresponding output (classification) value y_i associated with it. The k-nearest neighbor rule forms a classification function displayed in Equation 17. That is, it computes the average of the y values for the k nearest training points to **x** . If $f(\mathbf{x})$ ¡ 0.5, then the classifier decides that **x** is in class 1. Else, the classifier decides that **x** is in class 0. Using the functions defined previously, results of KNN can be observed within Figures 4 through 12 where 3-D means of plotting are used.

$$f(\mathbf{x}) = \frac{1}{k} \sum_{x_i k(x)} y_i \tag{17}$$

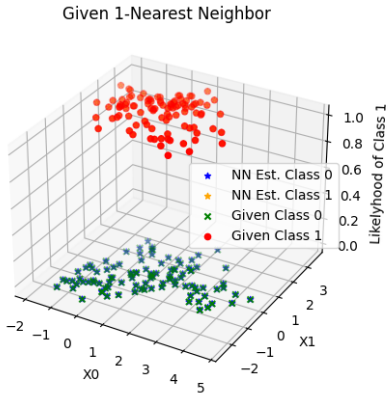

Fig. 5. Scatter plot showing how 1-NN would interpret the training data along with training data

VII. Conclusion

The results of the project can be observed within Table I where the error rates of training data are low and the error rates of the testing dats are all about 20 percent. This was unexpected as it was assumed that linear regression would be significantly worse. Analyzing the quadratic regression results further, it appears that it acts very similar to linear regression and appears very linear. However, it is quadratic in nature, but appears linear in the sense that a quadratic function appears linear when zoomed in far enough. Out of all the KNN results, 5-nearest neighbor performs better by a very slim difference. 1-nearest neighbor is the only method which has perfect results for the training data which makes sense being that it was trained on the training set. It is concluded that all methods of classification used are just as effective as each other for the given training and testing data sets.

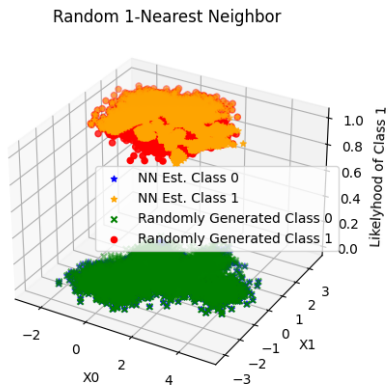

Fig. 6. Scatter plot showing how 1-NN would a generated dataset along with the generated dataset

TABLE I
ERROR RATE

Method	Training	Testing
Linear Regression	0.145	0.2061
Quadratic Regression	0.145	0.2053
1-Nearest Neighbor	0.0	0.2183
5-Nearest Neighbor	0.12	0.2054
15-Nearest Neighbor	0.16	0.1958

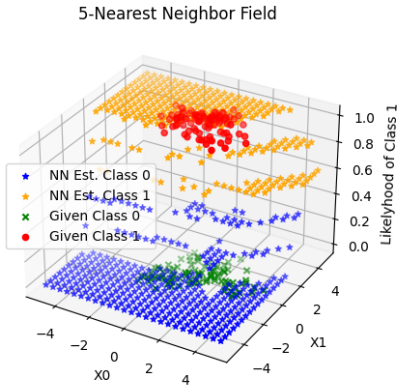

Fig. 7. Scatter plot showing the results of 5-Nearest Neighbor on points around the training data along with training data

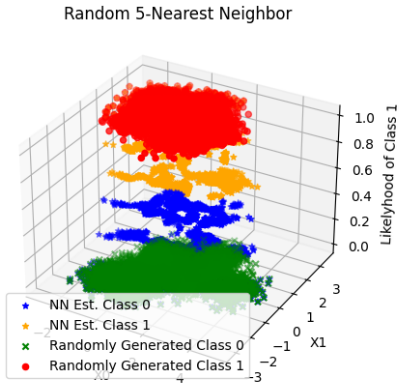

Fig. 9. Scatter plot showing how 5-NN would a generated dataset along with the generated dataset

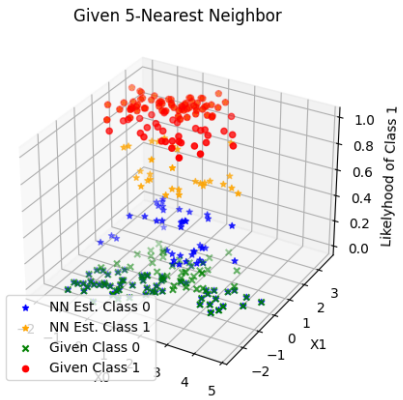

Fig. 8. Scatter plot showing how 5-NN would interpret the training data along with training data

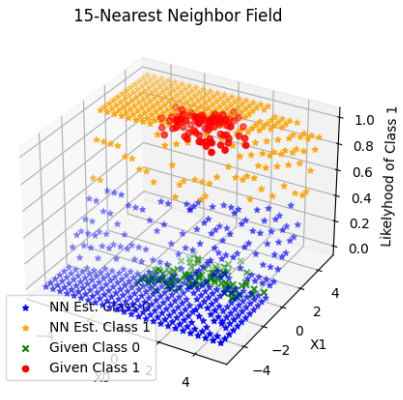

Fig. 10. Scatter plot showing the results of 15-Nearest Neighbor on points around the training data along with training data

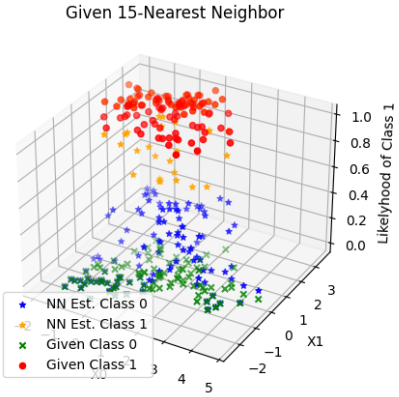

Fig. 11. Scatter plot showing how 15-NN would interpret the training data along with training data

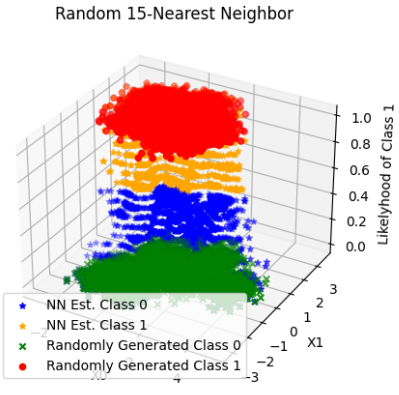

Fig. 12. Scatter plot showing how 15-NN would a generated dataset along with the generated dataset

VIII. Code

A. project4_angel.ipynb

```
import numpy as np
import matplotlib.pyplot as plt
from gendat2_python import gendat2

# loads the sample data where x0,y0 represents class 0 and x1,y1 represents class 1
x0, y0, x1, y1 = np.loadtxt('classasgntrain1.dat').T
N0 = x0.size
N1 = x1.size
N = N0 + N1
# this just plots the sample/given data
fig, ax = plt.subplots()
ax.plot(x0, y0, 'gx', label='Class 0')
ax.plot(x1, y1, 'ro', label='Class 1')
ax.legend()
plt.title("Sample Population")
plt.show()

x0, y0, x1, y1 = np.loadtxt('classasgntrain1.dat').T
# x0 = np.array([x0,y0])
# x1 = np.array([x1,y1])
print(f'x0: {x0.shape}')
print(f'x1: {x1.shape}')
print(f'N: {N}')

# Build the X matrix
class_0_matrix = np.array([np.ones(N0), x0, y0]).T
class_1_matrix = np.array([np.ones(N1), x1, y1]).T
X_aug = np.concatenate((class_0_matrix, class_1_matrix), axis=0)
# Build the indicator response matrix
Y0_array = np.array([np.ones(N0), np.zeros(N0)]).T
Y1_array = np.array([np.zeros(N1), np.ones(N1)]).T
Y = np.concatenate((Y0_array, Y1_array), axis=0)
# print(f'class_0_matrix: {class_0_matrix.shape}')
# print(f'class_1_matrix: {class_1_matrix.shape}')
print(f'X_aug: {X_aug.shape}')
print(f'Y: {Y.shape}')

# Find the parameter matrix
# Bhat = (X'*X) \ X'* Y;
Bhat = np.dot(np.dot(np.linalg.pinv(np.dot(X_aug.T, X_aug)), X_aug.T), Y)
# Find the approximate response
Yhat = np.dot(X_aug, Bhat)
Yhathard = Yhat > 0.5 # threshold into different classes
nerr = np.divide(np.sum(np.sum(np.absolute(Yhathard - Y))), 2); # count the total number of er
errrate_linregress_train = np.divide(nerr,N)
print(f'Bhat: {Bhat.shape}')
print(f'Yhat: {Yhat.shape}')
print(f'errrate_linregress_train: {errrate_linregress_train}')


# Now test on new (testing data)
Ntest0 = 5000 # number of class 0 points to generate
Ntest1 = 5000 # number of class 1 points to generate
```

```
# xtest0 = gendat2(0,Ntest0) # generate the test data for class 0
# xtest1 = gendat2(1,Ntest1) # generate the test data for class 1
x_test_class_0, y_test_class_0, x_test_class_1, y_test_class_1 = np.loadtxt('outputFile.dat').
print(f'x_test_class_0: {x_test_class_0.shape}')
print(f'y_test_class_0: {y_test_class_0.shape}')
xtest0 = np.array([x_test_class_0, y_test_class_0]) # generate the test data for class 0
xtest1 = np.array([x_test_class_1, y_test_class_1]) # generate the test data for class 1
print(f'xtest0: {xtest0.shape}')
print(f'xtest1: {xtest1.shape}')

nerr = 0
for i in range(Ntest0):
    # get xtest0[:,i]
    temp = xtest0.copy()
    temp = temp[:,i]
    yhat = np.dot(np.array([1, temp[0], temp[1]]), Bhat)
    if yhat[1] > yhat[0]: # error: chose class 0 over class 1
        nerr += 1
for i in range(Ntest1):
    # get xtest1[:,i]
    temp = xtest1.copy()
    temp = temp[:,i]
    yhat = np.dot(np.array([1, temp[0], temp[1]]), Bhat)
    if yhat[0] > yhat[1]: # error: chose class 0 over class 1
        nerr += 1
errrate_linregress_test = nerr / (Ntest0 + Ntest1)

# Get minimum and maximum values for all x and all y
class_0 = np.array([x0, y0])
class_1 = np.array([x1, y1])
x_values = np.concatenate((x0, x1))
y_values = np.concatenate((y0, y1))
xmin = np.min(x_values)
xmax = np.max(x_values)
ymin = np.min(y_values)
ymax = np.max(y_values)
print(f'xmin: {xmin}')
print(f'xmax: {xmax}')
xpl = np.linspace(xmin,xmax,100)
ypl = np.linspace(ymin,ymax,100)
# Make classification regions
# For each class, make array with 2 rows
redpts = np.array([[],[]]); # class 1 estimates
greenpts = np.array([[],[]]);# class 0 estimatees
for x in xpl:
    for y in ypl:
        yhat = np.dot(np.array([1, x, y]), Bhat)
        # Get [x;y] as column
        xy_array = np.array([[x], [y]])
        # print(f'xy_array: {xy_array.shape}')
        if yhat[0] > yhat[1]: # choose class 0
            greenpts = np.append(greenpts, xy_array, axis=1)
        else: # choose class 1
            redpts = np.append(redpts, xy_array, axis=1)
print(f'greenpts: {greenpts.shape}')
print(f'redpts: {redpts.shape}')
```

```
print(f'xtest0: {xtest0.shape}')
print(f'xtest1: {xtest1.shape}')
print(f'greenpts: {greenpts.shape}')
fig, ax = plt.subplots()
# Plot the classification regions)
ax.plot(greenpts[0,:], greenpts[1,:], 'g.', label='Class 0 Region', markersize=1)
ax.plot(redpts[0,:], redpts[1,:], 'r.', label='Class 1 Region', markersize=1)
# Plot test data for class 0 and class 1
ax.plot(xtest0[0,:], xtest0[1,:], 'g.', label='Class 0 Test', markersize=1)
ax.plot(xtest1[0,:], xtest1[1,:], 'r.', label='Class 1 Test', markersize=1)
# Plot training data for class 0 and class 1
ax.plot(x0, y0, 'gx', label='Class 0 Train')
ax.plot(x1, y1, 'rx', label='Class 1 Train')
ax.legend()
plt.title("Sample and Test Population with Classification Regions")
plt.show()
```

B. classasgn1.m

```
% classasgn1.m
clc, clear, close all;
% Sample classifier program
% Load the training data and divide into the different classes
load classasgntrain1.dat
x0 = classasgntrain1(:,1:2)'; % data vectors for class 0 (2 x N0)
N0 = size(x0,2);
x1 = classasgntrain1(:,3:4)'; % data vectors for class 1 (2 x N1)
N1 = size(x1,2);
N = N0 + N1;
% plot the data
clf;
plot(x0(1,:), x0(2,:),'go');
hold on;
plot(x1(1,:), x1(2,:),'ro');
xlabel('x_0');
ylabel('x_1');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Linear regression classifier
% Build the X matrix
X = [ones(N0,1) x0'; ones(N1,1) x1'];
% Build the indicator response matrix
Y = [ones(N0,1) zeros(N0,1); zeros(N1,1) ones(N1,1)];
% Find the parameter matrix
Bhat = (X'*X) \ X'* Y;
% Find the approximate response
Yhat = X*Bhat;
Yhathard = Yhat > 0.5; % threshold into different classes
nerr = sum(sum(abs(Yhathard - Y)))/2; % count the total number of errors
errrate_linregress_train = nerr / N;
% Now test on new (testing data)
Ntest0 = 5000; % number of class 0 points to generate
Ntest1 = 5000; % number of class 1 points to generate
xtest0 = gendat2(0,Ntest0); % generate the test data for class 0
xtest1 = gendat2(1,Ntest1); % generate the test data for class 1

%# create file, and write the title and number of columns
```

```
fid = fopen('outputFile.dat', 'wt');
fclose(fid);
%# append rest of data
dlmwrite('outputFile.dat', [xtest0' xtest1'], '-append', 'delimiter','\t')

nerr = 0;
for i=1:Ntest0
yhat = [1 xtest0(:,i)']*Bhat;
if(yhat(2) > yhat(1)) % error: chose class 1 over class 0
nerr = nerr+1;
 end
end
for i=1:Ntest1
 yhat = [1 xtest1(:,i)']*Bhat;
 if(yhat(1) > yhat(2)) % error: chose class 0 over class 1
 nerr = nerr+1;
 end
end
errrate_linregress_test = nerr / (Ntest0 + Ntest1);
% Plot the performance across the window (that is, plot the classification regions)
xmin = min([x0(1,:) x1(1,:)]); xmax = max([x0(1,:) x1(1,:)]);
ymin = min([x0(2,:) x1(2,:)]); ymax = max([x0(2,:) x1(2,:)]);
xpl = linspace(xmin,xmax,100);
ypl = linspace(ymin,ymax,100);
redpts = []; % class 1 estimates
greenpts = [];% class 0 estimatees
% loop over all points
for x = xpl
 for y = ypl
 yhat = [1 x y]*Bhat;
 if(yhat(1) > yhat(2)) % choose class 0 over class 1
 greenpts = [greenpts [x;y]];
 else
 redpts = [redpts [x;y]];
 end
 end
end
% Plot classification regions
plot(greenpts(1,:), greenpts(2,:),'g.','MarkerSize',0.25);
plot(redpts(1,:),redpts(2,:),'r.','MarkerSize',0.25);
% Plot training data
plot(xtest0(1,:), xtest0(2,:), 'g.','MarkerSize',0.5);
plot(xtest1(1,:), xtest1(2,:), 'r.','MarkerSize',0.5);
axis tight
```

C. *usefulfunctions.py*

```
import numpy as np
import matplotlib.pyplot as plt

class DrawOneDataset:
    def __init__(self, data, marker0, marker1, color0, color1, label0, label1, title):
        plt.switch_backend('QtAgg')
        self.data = data
        self.title = title
        self.marker0 = marker0
        self.marker1 = marker1
```

```
        self.color0 = color0
        self.color1 = color1
        self.label0 = label0
        self.label1 = label1
        self.fig = plt.figure()
        self.draw_setup()
        self.cid = self.fig.canvas.mpl_connect('close_event', self.on_close)
        self.run_flag = 1

    def draw_setup(self):
        # this just plots the sample/given data
        ax = self.fig.add_subplot(projection='3d')
        data_class0 = self.data[:, self.data[3]<self.data[2]]
        data_class1 = self.data[:, self.data[3]>self.data[2]]
        ax.scatter(data_class0[0], data_class0[1], data_class0[3], marker=self.marker0, color=
        ax.scatter(data_class1[0], data_class1[1], data_class1[3], marker=self.marker1, color=
        ax.set_xlabel('X0')
        ax.set_ylabel('X1')
        ax.set_zlabel('Likelyhood of Class 1')
        ax.legend()
        plt.title(self.title)

    def run_drawing(self):
        while self.run_flag:
            plt.draw()
            plt.pause(0.001)

    def on_close(self, event):
        self.run_flag = 0
        plt.close(fig=self.fig)

# class DrawTwoDataset:
#     def __init__(self, data0, data1, marker00, marker01, marker10, marker11, color00, color0
#         plt.switch_backend('QtAgg')
#         self.data0 = data0
#         self.data1 = data1
#         self.marker00 = marker00
#         self.marker01 = marker01
#         self.marker10 = marker10
#         self.marker11 = marker11
#         self.color00 = color00
#         self.color01 = color01
#         self.color10 = color10
#         self.color11 = color11
#         self.legend00 = legend00
#         self.legend01 = legend01
#         self.legend10 = legend10
#         self.legend11 = legend11
#         self.title = title
#         self.fig = plt.figure()
#         self.draw_setup()
#         self.cid = self.fig.canvas.mpl_connect('close_event', self.on_close)
#         self.run_flag = 1

#     def draw_setup(self):
#         # this just plots the sample/given data
```

```
#         ax = self.fig.add_subplot(projection='3d')
#         data_class0 = self.data0[:, self.data0[3]<self.data0[2]]
#         data_class1 = self.data0[:, self.data0[3]>self.data0[2]]
#         ax.scatter(data_class0[0], data_class0[1], data_class0[3], marker=self.marker00, col
#         ax.scatter(data_class1[0], data_class1[1], data_class1[3], marker=self.marker01, col
#         sample_class0 = self.data1[:, self.data1[3]<self.data1[2]]
#         sample_class1 = self.data1[:, self.data1[3]>self.data1[2]]
#         ax.scatter(sample_class0[0], sample_class0[1], sample_class0[3], marker=self.marker1
#         ax.scatter(sample_class1[0], sample_class1[1], sample_class1[3], marker=self.marker1
#         ax.set_xlabel('X0')
#         ax.set_ylabel('X1')
#         ax.set_zlabel('Likelyhood of Class 1')
#         ax.legend()
#         plt.title(self.title)

#     def run_drawing(self):
#         while self.run_flag:
#             plt.draw()
#             plt.pause(0.001)

#     def on_close(self, event):
#         self.run_flag = 0
#         plt.close(fig=self.fig)

class DrawTwoDataset:
    def __init__(self, data0, data1, marker00, marker01, marker10, marker11, color00, color01,
        self.data0 = data0
        self.data1 = data1
        self.marker00 = marker00
        self.marker01 = marker01
        self.marker10 = marker10
        self.marker11 = marker11
        self.color00 = color00
        self.color01 = color01
        self.color10 = color10
        self.color11 = color11
        self.legend00 = legend00
        self.legend01 = legend01
        self.legend10 = legend10
        self.legend11 = legend11
        self.title = title
        self.fig = plt.figure()
        self.draw_setup()

    def draw_setup(self):
        # this just plots the sample/given data
        ax = self.fig.add_subplot()
        data_class0 = self.data0[:, self.data0[3]<self.data0[2]]
        data_class1 = self.data0[:, self.data0[3]>self.data0[2]]
        ax.scatter(data_class0[0], data_class0[1], marker=self.marker00, color=self.color00, la
        ax.scatter(data_class1[0], data_class1[1], marker=self.marker01, color=self.color01, la
        sample_class0 = self.data1[:, self.data1[3]<self.data1[2]]
        sample_class1 = self.data1[:, self.data1[3]>self.data1[2]]
        ax.scatter(sample_class0[0], sample_class0[1], marker=self.marker10, color=self.color1
        ax.scatter(sample_class1[0], sample_class1[1], marker=self.marker11, color=self.color1
        ax.set_xlabel('X0')
```

```
        ax.set_ylabel('X1')
        ax.legend()
        ax.set(xlim=(-1.918,4.6887), ylim=(-2.549,3.5951))
        plt.title(self.title)

    def run_drawing(self):
        plt.show()


class DrawThreeDataset:
    def __init__(self, data0, data1, data2, title):
        plt.switch_backend('QtAgg')
        self.data0 = data0
        self.data1 = data1
        self.data2 = data2
        self.title = title
        self.fig = plt.figure()
        self.draw_setup()
        self.cid = self.fig.canvas.mpl_connect('close_event', self.on_close)
        self.run_flag = 1

    def draw_setup(self):
        # this just plots the sample/given data
        ax = self.fig.add_subplot(projection='3d')
        data_class0 = self.data0[:, self.data0[3]<self.data0[2]]
        data_class1 = self.data0[:, self.data0[3]>self.data0[2]]
        ax.scatter(data_class0[0], data_class0[1], data_class0[3], marker='*', color='green')
        ax.scatter(data_class1[0], data_class1[1], data_class1[3], marker='*', color='red')
        data1_class0 = self.data1[:, self.data1[3]<self.data1[2]]
        data1_class1 = self.data1[:, self.data1[3]>self.data1[2]]
        ax.scatter(data1_class0[0], data1_class0[1], data1_class0[3], marker='x', color='green
        ax.scatter(data1_class1[0], data1_class1[1], data1_class1[3], marker='o', color='red')
        data2_class0 = self.data2[:, self.data2[3]<self.data2[2]]
        data2_class1 = self.data2[:, self.data2[3]>self.data2[2]]
        ax.scatter(data2_class0[0], data2_class0[1], data2_class0[3], marker='D', color='green
        ax.scatter(data2_class1[0], data2_class1[1], data2_class1[3], marker='D', color='red')
        ax.set_xlabel('x0')
        ax.set_ylabel('x1')
        ax.set_zlabel('likelyhood of class 1')
        plt.title(self.title)

    def run_drawing(self):
        while self.run_flag:
            plt.draw()
            plt.pause(0.001)

    def on_close(self, event):
        self.run_flag = 0
        plt.close(fig=self.fig)
```

D. project4_austin.ipynb

```
import numpy as np
import matplotlib.pyplot as plt
from importlib import reload

# custom library
```

```
import useful_functions
# useful for when you need to reload your library without reloading the entire notebook
reload(useful_functions)

x0_class0, x1_class0, x0_class1, x1_class1 = np.loadtxt('classasgntrain1.dat').T
data_x0 = np.concatenate((x0_class0, x0_class1))
data_x1 = np.concatenate((x1_class0, x1_class1))
data_y_class0 = np.concatenate((np.ones(x0_class0.size), np.zeros(x0_class1.size)))
data_y_class1 = np.concatenate((np.zeros(x1_class0.size), np.ones(x1_class1.size)))
data = np.asanyarray((data_x0, data_x1, data_y_class0, data_y_class1))
print(f'This is x0 array: \n{data[0]}')
print()
print(f'This is x1 array: \n{data[1]}')
print()
print(f'This states if a point is in class 0: \n{data[2]}')
print()
print(f'This states if a point is in class 1: \n{data[3]}')
print()

random0_class0, random1_class0, random0_class1, random1_class1 = np.loadtxt('outputFile.dat').
random_x0 = np.concatenate((random0_class0, random0_class1))
random_x1 = np.concatenate((random1_class0, random1_class1))
random_y_class0 = np.concatenate((np.ones(random0_class0.size), np.zeros(random0_class1.size))
random_y_class1 = np.concatenate((np.zeros(random1_class0.size), np.ones(random1_class1.size))
random_data = np.asanyarray((random_x0, random_x1, random_y_class0, random_y_class1))
print(f'This is random_x0 array: \n{random_data[0]}')
print()
print(f'This is random_x1 array: \n{random_data[1]}')
print()
print(f'This states if a point is in class 0: \n{random_data[2]}')
print()
print(f'This states if a point is in class 1: \n{random_data[3]}')
print()

# fig = useful_functions.DrawOneDataset(data, 'x', 'o', 'green', 'red', 'Class 0', 'Class 1',
# fig.run_drawing()

# linear regression function from project 1
# NOTE: this didn't work on windows maybe
def linear_regression(x0, x1, y):
    '''x (numpy.ndarray): x-coordinates of training data
    y (numpy.ndarray): y-coordinates of training data'''
    X_aug = np.stack((np.ones(x0.size), x0, x1), axis=1)
    Beta = np.dot(np.dot(np.linalg.pinv(np.dot(X_aug.T, X_aug)), X_aug.T), y)
    return Beta

# gestimating an output given an input and a linear regression
# NOTE: this didn't work on windows maybe
def least_squares_est(x0, x1, beta):
    '''x (numpy.ndarray): x-coordinate which we want a prediction value for based on the model
    y_hat (numpy.ndarray): y-coordinate predictions based on given inputs x'''
    X_aug = np.stack((np.ones(x0.size), x0, x1), axis=1)
    y_hat = np.dot(X_aug, beta)
    return y_hat

    def gen_2D_array(N, upper, lower):
```

```
    x_extended0 = np.array([])
    for i in range(N):
        x_extended0 = np.append(x_extended0, np.linspace(lower, upper, N))
    x_extended1 = np.array([])
    for i in np.linspace(lower, upper, N):
        x_extended1 = np.append(x_extended1, np.repeat(i, N))

    return x_extended0, x_extended1

x_extended0, x_extended1 = gen_2D_array(100, -5, 5)
x_extended = np.vstack((x_extended0, x_extended1))

beta0 = linear_regression(data[0], data[1], data[2])
beta1 = linear_regression(data[0], data[1], data[3])

y0_hat = least_squares_est(x_extended0, x_extended1, beta0)
y1_hat = least_squares_est(x_extended0, x_extended1, beta1)
data_hat = np.vstack((x_extended0, x_extended1, y0_hat, y1_hat))

fig = useful_functions.DrawTwoDataset(data_hat, data, '*', '*', 'x', 'o',
                                      'blue', 'orange', 'green', 'red',
                                      'LSR Estimatied Class 0', 'LSR Estimated Class 1', 'Given C
fig.run_drawing()

y0_hat = least_squares_est(data[0], data[1], beta0)
y1_hat = least_squares_est(data[0], data[1], beta1)
data_hat = np.vstack((data[0], data[1], y0_hat, y1_hat))

value_vector = np.array([])
for i in range(data_hat[3].size):
    if data_hat[3][i] > 0.5:
        value_vector = np.append(value_vector, 1)
    else:
        value_vector = np.append(value_vector, 0)

error_rate = np.sum(np.abs(value_vector - data[3])) / data[3].size
print(f'Error rate of LSR on Given Data: {error_rate}')

fig = useful_functions.DrawTwoDataset(data_hat, data, '*', '*', 'x', 'o',
                                      'blue', 'orange', 'green', 'red',
                                      'LSR Estimatied Class 0', 'LSR Estimated Class 1', 'Given C
fig.run_drawing()

random0_hat = least_squares_est(random_data[0], random_data[1], beta0)
random1_hat = least_squares_est(random_data[0], random_data[1], beta1)
random_guess = np.vstack((random_data[0], random_data[1], random0_hat, random1_hat))

value_vector = np.array([])
for i in range(random_guess[3].size):
    if random_guess[3][i] > 0.5:
        value_vector = np.append(value_vector, 1)
    else:
        value_vector = np.append(value_vector, 0)

error_rate = np.sum(np.abs(value_vector - random_data[3])) / random_data[3].size
print(f'Error rate of LSR on Random Data: {error_rate}')
```

```
fig = useful_functions.DrawTwoDataset(random_guess, random_data, '*', '*', 'x', 'o',
                                      'blue', 'orange', 'green', 'red',
                                      'Estimation of Class 0', 'Estimation of Class 1', 'Randoml
fig.run_drawing()

# quadratic regression function from project 1
# NOTE: this didn't work on windows maybe
def quadratic_regression(x0, x1, y):
    '''x (numpy.ndarray): x-coordinates of training data
    y (numpy.ndarray): y-coordinates of training data'''
    X_aug = np.stack((np.ones(x0.size), x0, x0**2, x1, x1**2, x0*x1), axis=1)
    Beta = np.dot(np.dot(np.linalg.pinv(np.dot(X_aug.T, X_aug)), X_aug.T), y)
    return Beta

# gestimating an output given an input and a quadratic regression
# NOTE: this didn't work on windows maybe
def quad_least_squares_est(x0, x1, beta):
    '''x (numpy.ndarray): x-coordinate which we want a prediction value for based on the model
    y_hat (numpy.ndarray): y-coordinate predictions based on given inputs x'''
    X_aug = np.stack((np.ones(x0.size), x0, x0**2, x1, x1**2, x0*x1), axis=1)
    y_hat = np.dot(X_aug, beta)
    return y_hat

qbeta0 = quadratic_regression(data[0], data[1], data[2])
qbeta1 = quadratic_regression(data[0], data[1], data[3])

y0_hat = quad_least_squares_est(x_extended0, x_extended1, qbeta0)
y1_hat = quad_least_squares_est(x_extended0, x_extended1, qbeta1)
data_hat = np.vstack((x_extended0, x_extended1, y0_hat, y1_hat))

fig = useful_functions.DrawTwoDataset(data_hat, data, ',', ',', 'o', 'o',
                                      'lightgreen', 'pink', 'green', 'red',
                                      'Quad. LSR Est. Class 0', 'Quad. LSR Est. Class 1', 'Given
fig.run_drawing()

y0_hat = quad_least_squares_est(data[0], data[1], qbeta0)
y1_hat = quad_least_squares_est(data[0], data[1], qbeta1)
data_hat = np.vstack((data[0], data[1], y0_hat, y1_hat))

value_vector = np.array([])
for i in range(data_hat[3].size):
    if data_hat[3][i] > 0.5:
        value_vector = np.append(value_vector, 1)
    else:
        value_vector = np.append(value_vector, 0)

error_rate = np.sum(np.abs(value_vector - data[3])) / data[3].size
print(f'Error rate of LSR on Given Data: {error_rate}')

fig = useful_functions.DrawTwoDataset(data_hat, data, '*', '*', 'x', 'o',
                                      'blue', 'orange', 'green', 'red',
                                      'Quad. LSR Est. Class 0', 'Quad. LSR Est. Class 1', 'Given
fig.run_drawing()

random0_hat = quad_least_squares_est(random_data[0], random_data[1], qbeta0)
```

```
random1_hat = quad_least_squares_est(random_data[0], random_data[1], qbeta1)
random_guess = np.vstack((random_data[0], random_data[1], random0_hat, random1_hat))

value_vector = np.array([])
for i in range(random_guess[3].size):
    if random_guess[3][i] > 0.5:
        value_vector = np.append(value_vector, 1)
    else:
        value_vector = np.append(value_vector, 0)

error_rate = np.sum(np.abs(value_vector - random_data[3])) / random_data[3].size
print(f'Error rate of LSR on Random Data: {error_rate}')

fig = useful_functions.DrawTwoDataset(random_guess, random_data, '*', '*', 'x', 'o',
                                      'blue', 'orange', 'green', 'red',
                                      'Quad. LSR Est. Class 0', 'Quad. LSR Est. Class 1', 'Rando
fig.run_drawing()

def get_distance(x1,y1,x2,y2):
    """x1/y1 correspond to first coordinate
    and x2/y2 correspond to second coordinate"""
    return np.sqrt((x2-x1)**2 + (y2-y1)**2)

def nearest_neighbor_var(unknown, known, num_nearest):
    if num_nearest >= known[0].size:
        raise ValueError('The order of nearest neighbor must be less than the number of known
    output_array = np.zeros(unknown[0].size)
    temp_data = np.vstack((np.zeros(known[0].size), np.zeros(known[0].size)))
    for j in range(unknown[0].size):
        for i in range(known[0].size):
            temp_data[0][i] = get_distance(known[0][i], known[1][i], unknown[0][j], unknown[1]
            temp_data[1][i] = known[3][i]
        largest_distancex2 = 2*temp_data[0][np.argmax(temp_data[0])]
        for z in range(num_nearest):
            min_position = np.argmin(temp_data[0])
            output_array[j] += temp_data[1][min_position]
            temp_data[0][min_position] = largest_distancex2

    output_array = output_array / num_nearest

    return output_array

nearest_neighbor_count = 15
output_array = nearest_neighbor_var(x_extended, data, nearest_neighbor_count)
data_hat = np.vstack((x_extended, np.abs(output_array-1), output_array))

fig = useful_functions.DrawTwoDataset(data_hat, data, '*', '*', 'x', 'o',
                                      'blue', 'orange', 'green', 'red',
                                      'NN Est. Class 0', 'NN Est. Class 1', 'Given Class 0', 'Gi
fig.run_drawing()

nearest_neighbor_count = 15
output_array = nearest_neighbor_var(random_data, data, nearest_neighbor_count)
data_hat = np.vstack((random_data[0], random_data[1], 1-output_array, output_array))

error_rate = np.sum(np.abs(np.round(data_hat[3]) - random_data[3])) / random_data[3].size
```

```
print(f'Error rate of {nearest_neighbor_count}-NN on Random Data: {error_rate}')

fig = useful_functions.DrawTwoDataset(data_hat, random_data, '*', '*', 'x', 'o',
                                      'blue', 'orange', 'green', 'red',
                                      'NN Est. Class 0', 'NN Est. Class 1', 'Randomly Generated
fig.run_drawing()
```