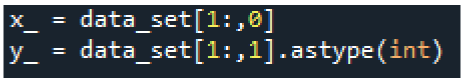
ENGR421 HW-4

Berke Can Rizai, 69282

After importing the necessary libraries, with the np.genfromtxt we take the dataset inside python.

Here x_ and y_ are data and value columns for the data.

First column is x and second is y.

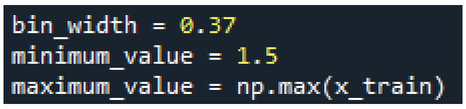


```
x_ = data_set[1:,0]
y_ = data_set[1:,1].astype(int)
```

N is the shape[0] of dataset which is the total number of elements.

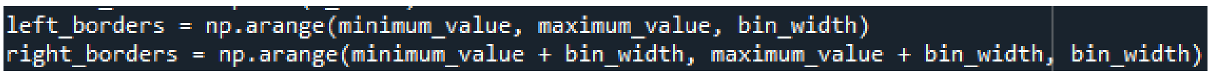
With the [:number] and [number:] we split the array so that :number is all the elements of the array except the number indexed element.

Bin width, minimum value and maximum value are initialized as follows;



```
bin_width = 0.37
minimum_value = 1.5
maximum_value = np.max(x_train)
```

For the regressogram, we initialize the left and right borders as follows,



```
left_borders = np.arange(minimum_value, maximum_value, bin_width)
right_borders = np.arange(minimum_value + bin_width, maximum_value + bin_width, bin_width)
```

Scoring is done on the next line and is saved to p_hat variable.

p_hat = np.asarray([np.sum(y_train[(left_borders[b] < x_train) & (x_train <= right_borders[b])] / np.sum((left_borders[b] < x_train) & (x_train <= right_borders[b]))) for b in range(len(left_borders))])

Formula is,

$$g(x) = \frac{\sum_{i=1}^{N} b(x, x_i) y_i}{\sum_{i=1}^{N} b(x, x_i)}$$

where

$$b(x, x_i) = \begin{cases} 1 & \text{if } x_i \text{ is in the same bin with } x \\ 0 & \text{otherwise} \end{cases}$$

This is very similar to what we have done on the lab, we take the mean values of the y, target value, for each bin.

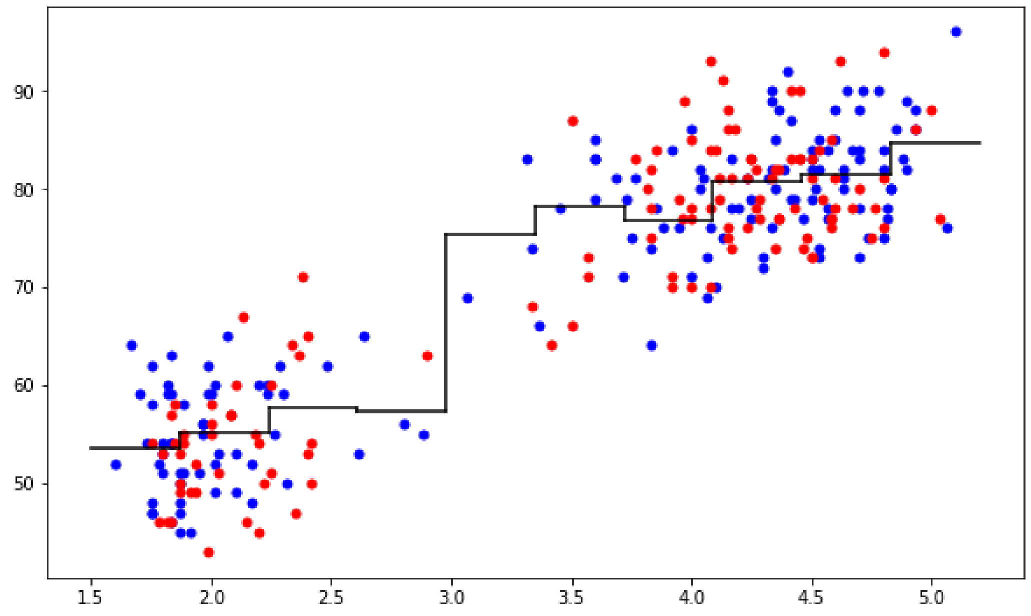
[(left_borders[b] < x_train) & (x_train <= right_borders[b])] returns a boolean for each of the bins and we take y_train values with this. Np.sum sums the values and we divide them to number of items in bin.

We draw points and draw the regressogram with the;

```
plt.figure(figsize = (10, 6))
plt.plot(x_train, y_train, "b.", markersize = 10)
plt.plot(x_test, y_test, "r.", markersize = 10)

for b in range(len(left_borders)):
    plt.plot([left_borders[b], right_borders[b]], [p_hat[b], p_hat[b]], "k-")
for b in range(len(left_borders) - 1):
    plt.plot([right_borders[b], right_borders[b]], [p_hat[b], p_hat[b + 1]], "k-")
plt.show()
```

We get the following result.



To calculate error,

I hold all of the squared errors in the sums variable. For every item in the x_test, I check the corresponding bin with the int((x_test[i] - minimum_value) // 0.37). This finds the index of p_hat we need to look for each of the points.

Then, we take the square of the error which is (real - prediction) ** 2 where real value is y_test[i].

We add this error to sums.

```
sums = 0
for i in range(len(x_test)):
    indx = int((x_test[i] - minimum_value) // 0.37)
    prediction = p_hat[indx]
    errorsq = (y_test[i] - prediction) ** 2
    sums = sums + errorsq
```

Then, we take the sum, divide with the number of data points and take the squared root.

```
rmse = math.sqrt(sums / len(y_test) )
```

print("Regressogram => RMSE is " +str(rmse) +" when h is 0.37") prints the value.

Produces;

```
Regressogram => RMSE is 5.962617204275407 when h is 0.37
```

for the data.

Next, data_interval is the linearly spaced values between minimum and maximum value and the number is the number of values, in this case it is 2501. Np.linspace does it for us.

```
data_interval = np.linspace(minimum_value, np.max(x_train), 2501)
```

We use the Running Mean Smoother formula from book and lab.

Running Mean Smoother

$$g(x) = \frac{\sum_{i=1}^{N} w\left(\frac{x - x_i}{h}\right) y_i}{\sum_{i=1}^{N} w\left(\frac{x - x_i}{h}\right)}$$

where

$$w(u) = \begin{cases} 1 & \text{if } |u| \leq 1/2 \\ 0 & \text{otherwise} \end{cases}$$

Here the code that is corresponding to the formula is as follows;

The idea is that for each of the interval, bin is the space that is 0.5 to the right of the point and 0.5 to the left of the point instead of the fixed bins. With this, we have more smoother and more correct approach.

Calculation for the interval is as follows;

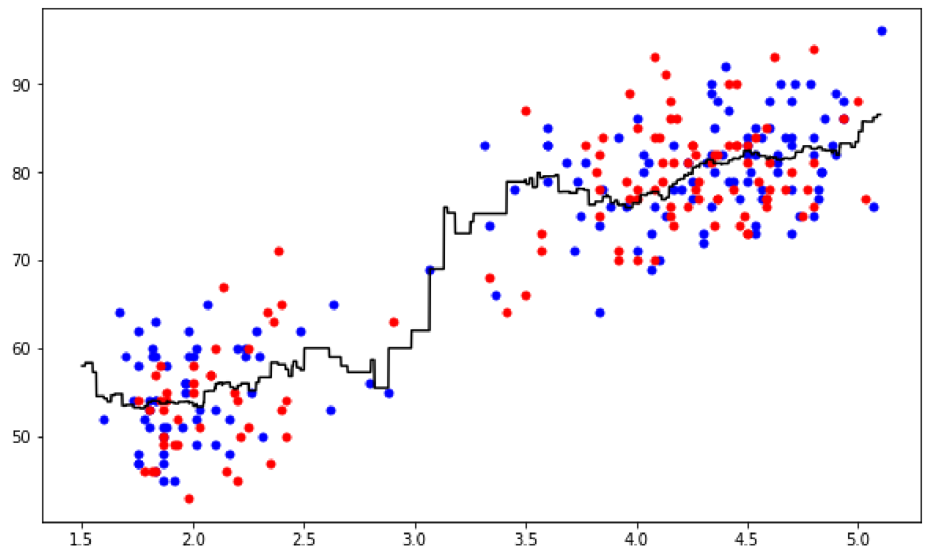
p_hat = np.asarray([np.sum(y_train[((x - 0.5 * bin_width) < x_train) & (x_train <= (x + 0.5 * bin_width))] / len(y_train[((x - 0.5 * bin_width) < x_train) & (x_train <= (x + 0.5 * bin_width))])) for x in data_interval])

Each value is stored on p_hat for the interval.

We again, plot the RMS and points with

```
plt.figure(figsize = (10, 6))
plt.plot(x_train, y_train, "b.", markersize = 10)
plt.plot(x_test, y_test, "r.", markersize = 10)
plt.plot(data_interval, p_hat, "k-")
plt.show()
```

Data_interval is the x and p_hat values are y for the plot function. And this produces,



For the prediction and the error, I used the bisect library. This library allows us to find which interval any of the values we have belongs to. For example if we give it x[i], it returns the index of the p_hat array that is predictive of the ith element.

Logic for finding RMSE is same, we sum errors in sums, we look up prediction with p_hat[indx].

```
sums = 0
for i in range(len(x_test)):
    indx = bisect.bisect_left(data_interval, x_test[i])
    prediction = p_hat[indx]
    errorsq = (y_test[i] - prediction) ** 2
    sums = sums + errorsq
```

After summing errors in our predictions we take the square root of the mean.

```
rmse = math.sqrt(sums / len(y_test) )


print("Running Mean Smoother => RMSE is " +str(rmse) +" when h is 0.37")
```

Result is;

```
Running Mean Smoother => RMSE is 6.089003211720321 when h is 0.37
```

Last one is the Kernel Smoother,

Here, the formula we use is

Kernel Smoother

$$g(x) = \frac{\sum_{i=1}^{N} K\left(\frac{x - x_i}{h}\right) y_i}{\sum_{i=1}^{N} K\left(\frac{x - x_i}{h}\right)}$$

where

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{u^2}{2}\right)$$

This is similar to the running mean smoother except the function K we apply.

In the data_interval we calculate values with,

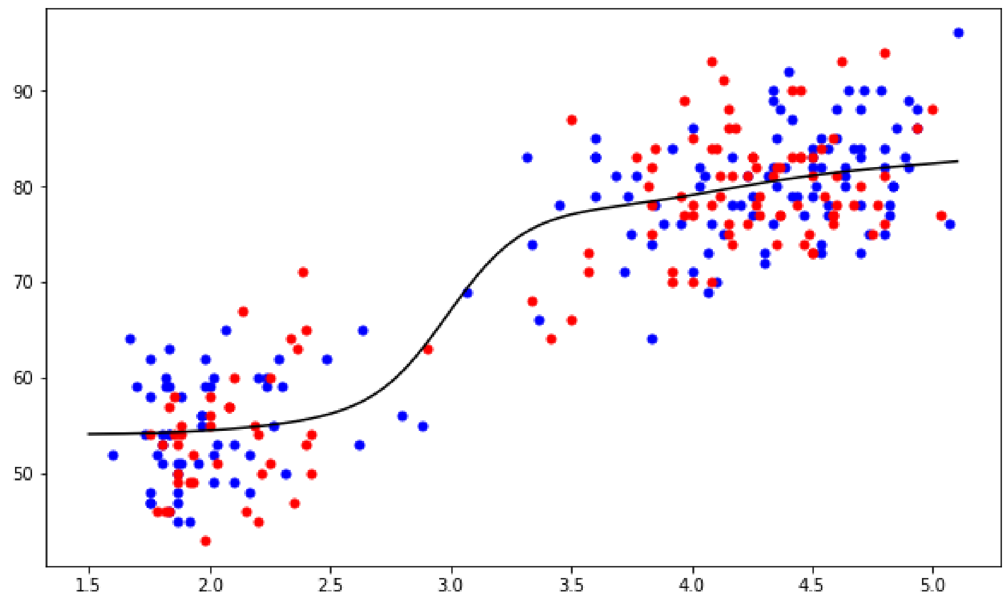
p_hat = np.asarray([np.sum(1.0 / np.sqrt(2 * math.pi) * np.exp(-0.5 * (x - x_train)**2 / bin_width**2) * y_train) / (np.sum(1/np.sqrt(2 * math.pi) * np.exp((-0.5 * (x - x_train)**2) / bin_width**2))) for x in data_interval])

Where np.exp takes exponent and math.pi is the pi number.

then we plot the function as well as the points with the same code.

```
plt.figure(figsize = (10, 6))
plt.plot(x_train, y_train, "b.", markersize = 10)
plt.plot(x_test, y_test, "r.", markersize = 10)
plt.plot(data_interval, p_hat, "k-")
plt.show()
```

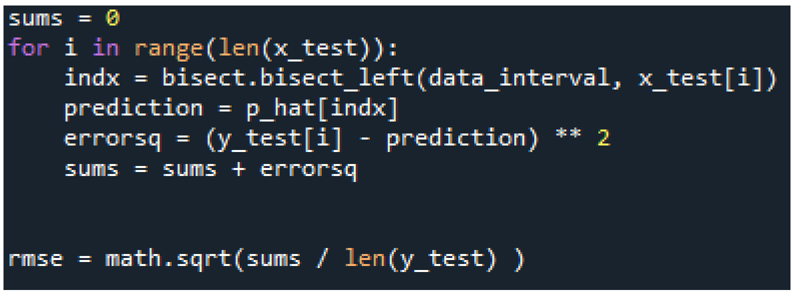
Produces the following plot,



Again, with the bisect, we find the corresponding interval for the x_test set values and get the prediction from the p_hat with the index.

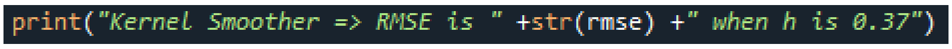
We add up all the squared errors we find by deducting prediction from real y and squaring it.

We then print the RMSE and we end up with the result;



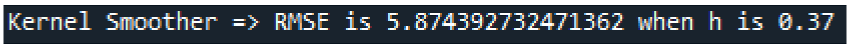
```
sums = 0
for i in range(len(x_test)):
    indx = bisect.bisect_left(data_interval, x_test[i])
    prediction = p_hat[indx]
    errorsq = (y_test[i] - prediction) ** 2
    sums = sums + errorsq


rmse = math.sqrt(sums / len(y_test) )
```



```
print("Kernel Smoother => RMSE is " +str(rmse) +" when h is 0.37")
```

Prints the result.



```
Kernel Smoother => RMSE is 5.874392732471362 when h is 0.37
```

Looking at the results, kernel smoother produces the best result as it gives the least error overall.