Regression

Jiaren Zhou, B.Eng.
E-Mail: iuk.zhou@gmail.com

Betreuer: Florian Particke M.Sc.

Prof. Dr.-Ing. Jörn Thielecke
Professur für Informationstechnik
Schwerpunkt Navigation und Ortsbestimmung
Telefon: +49 9131 8525-118
Fax: +49 9131 8525-102
E-Mail: joern.thielecke@fau.de

LIKE

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

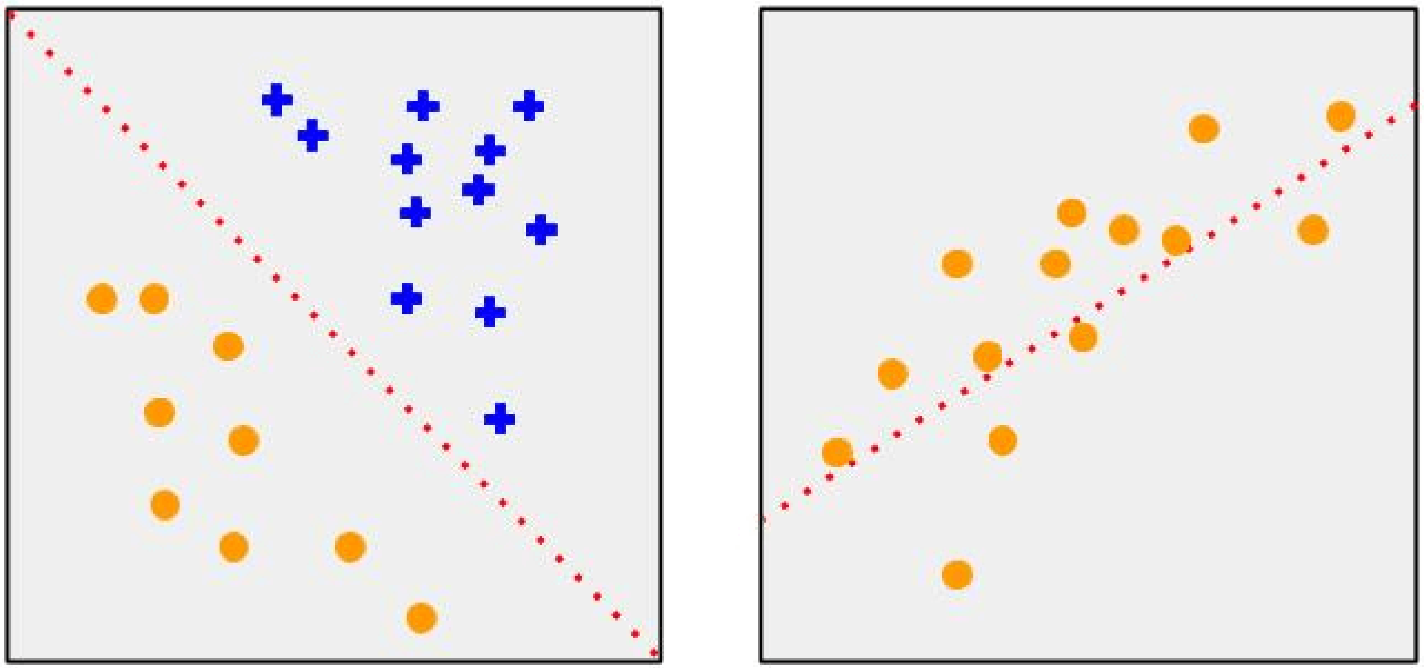
Contents

LIKE

I. Regression vs. Classification

"*Classification*" is the task of predicting a discrete class label.

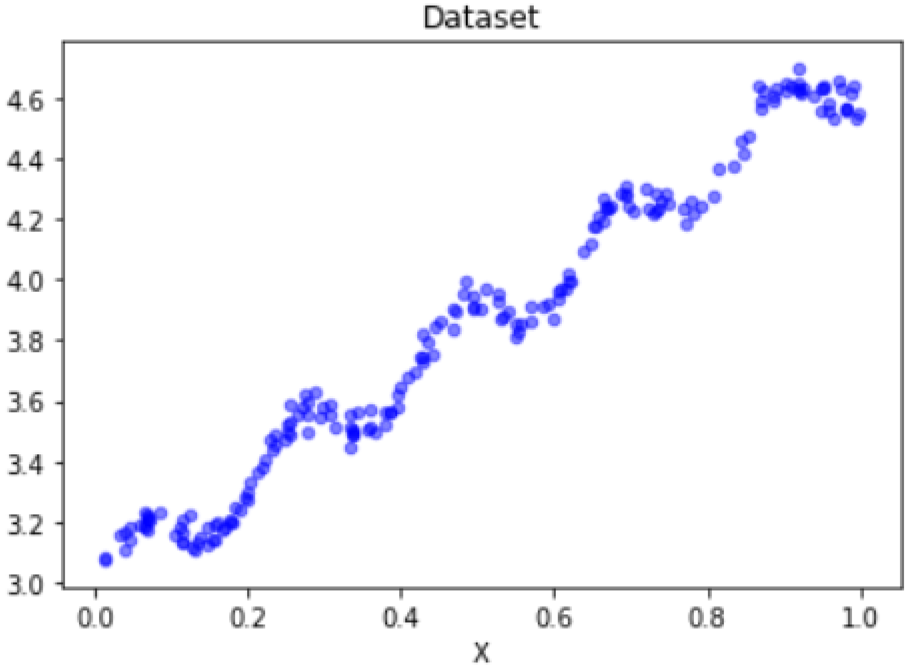
"*Regression*" is the task of predicting a continuous quantity.



Classification Regression

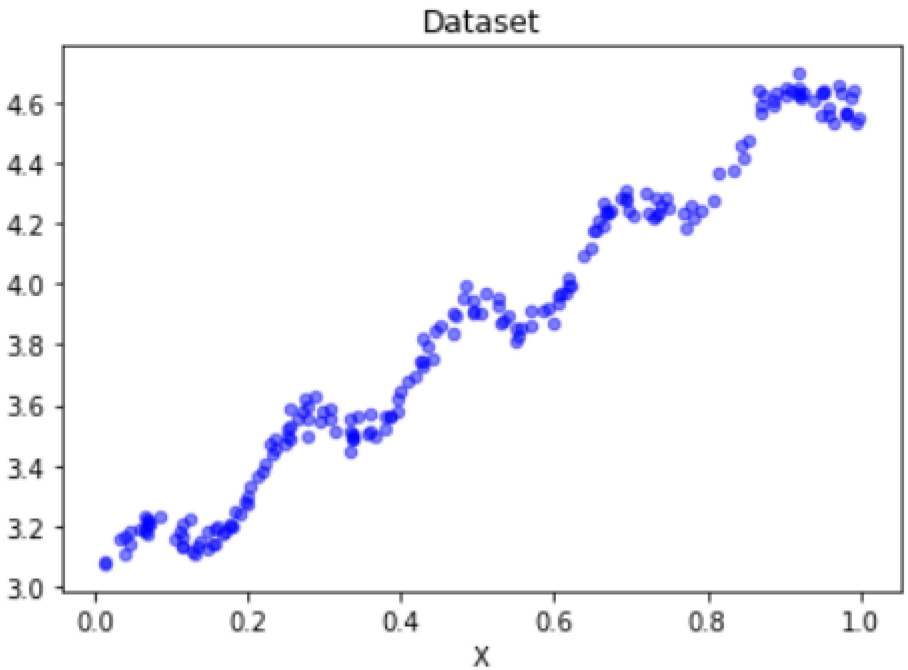
LIKE

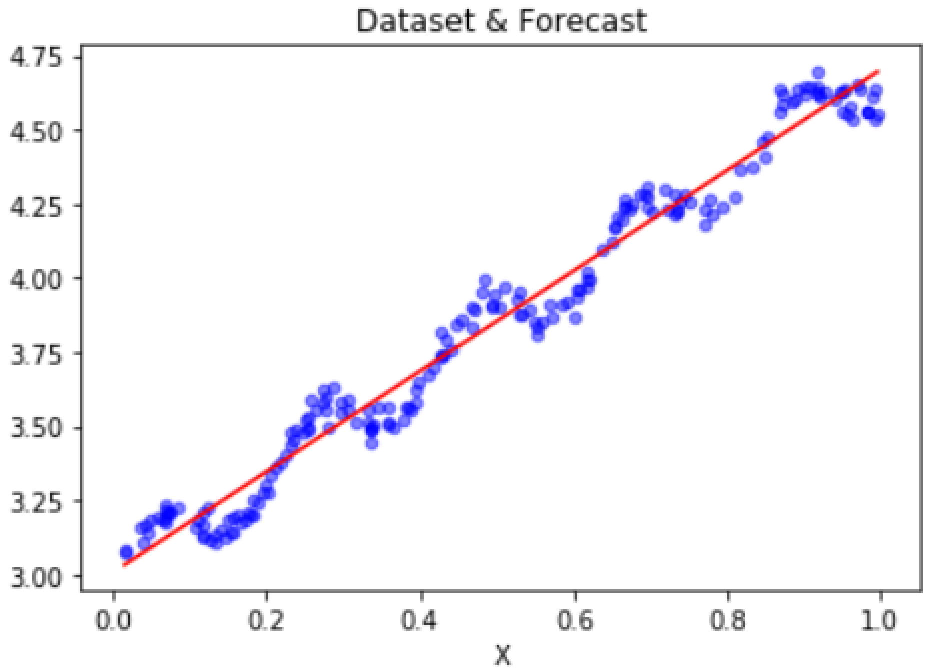
II. Linear Regression



LIKE

II. Linear Regression



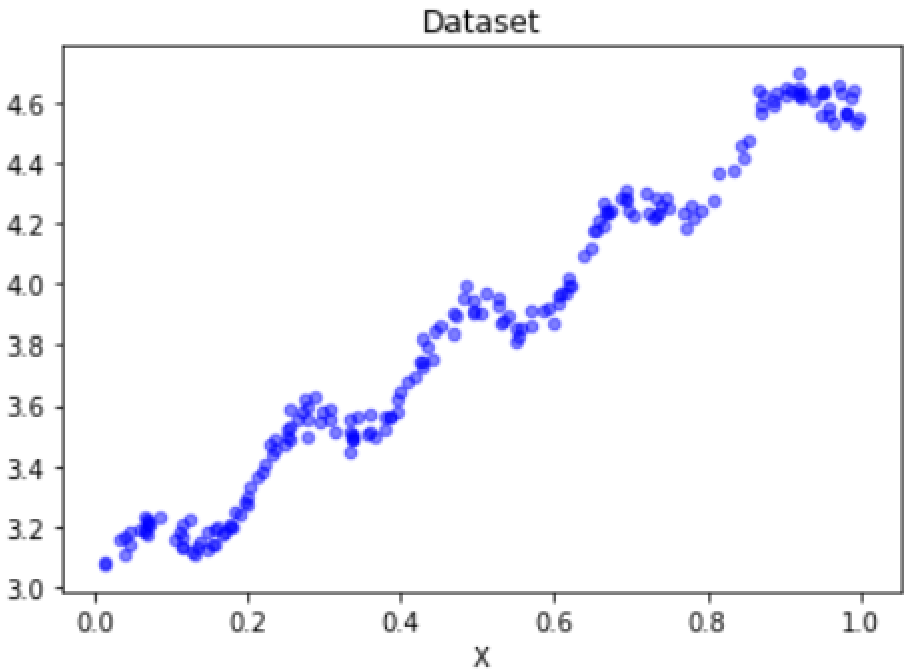


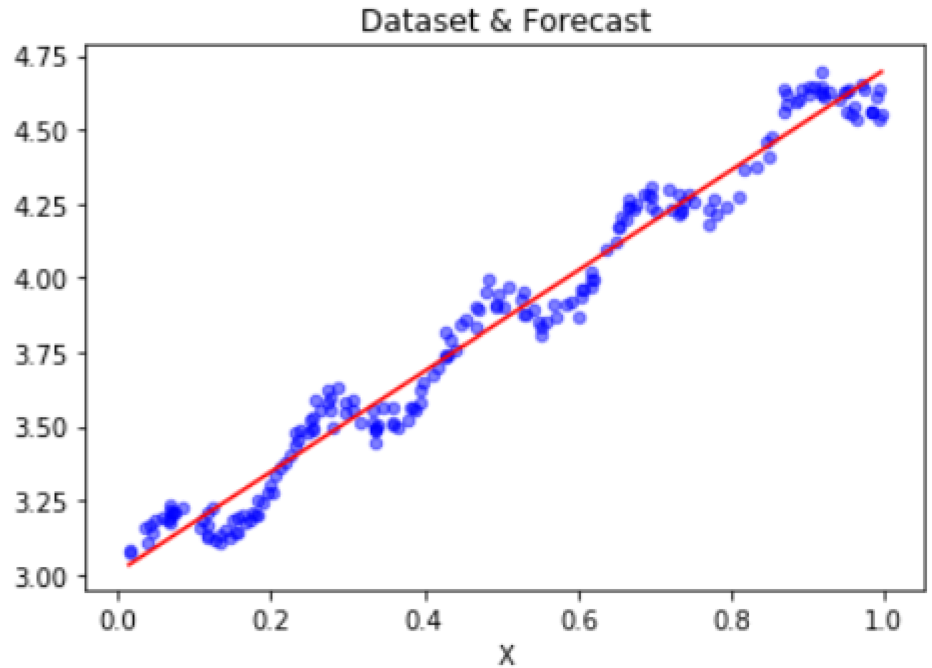
y = ω_1 * x + ω_0

ω_1 : 1.7
ω_0 : 3.0

LIKE

II. Linear Regression





$$y = \omega_1 * x + \omega_0$$

how to find the best **weight** ?

$\omega_1 : 1.7$
$\omega_0 : 3.0$

How to find the best **weight**:

find the *least squared error*

$$\sum_{i=1}^{m}(y_i - x_i^T\omega)^2$$

The expression in *matrix* is

$$(Y - X\omega)^T(Y - X\omega)$$

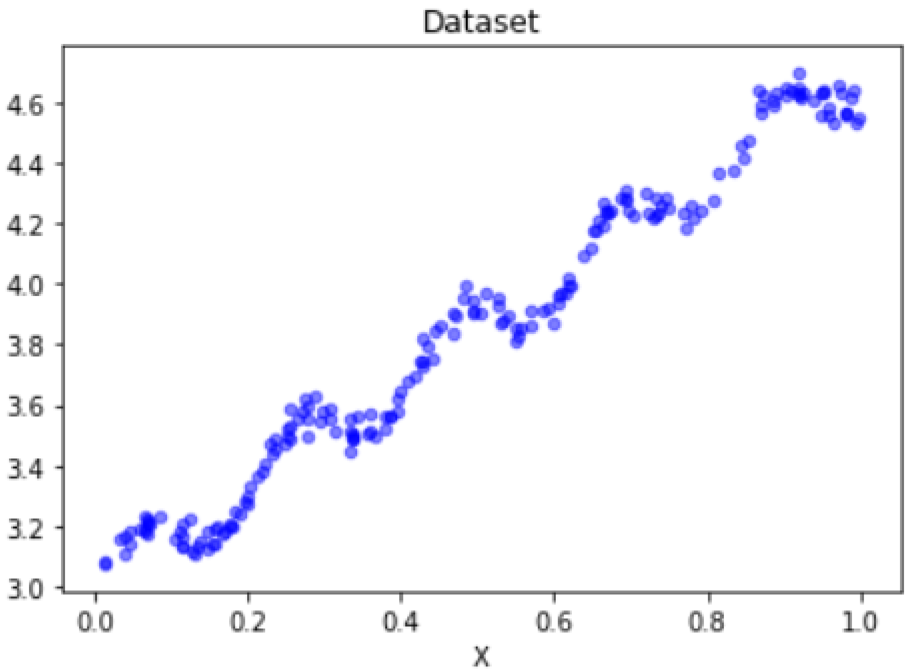
After the *derivation* with respect to ω, we get

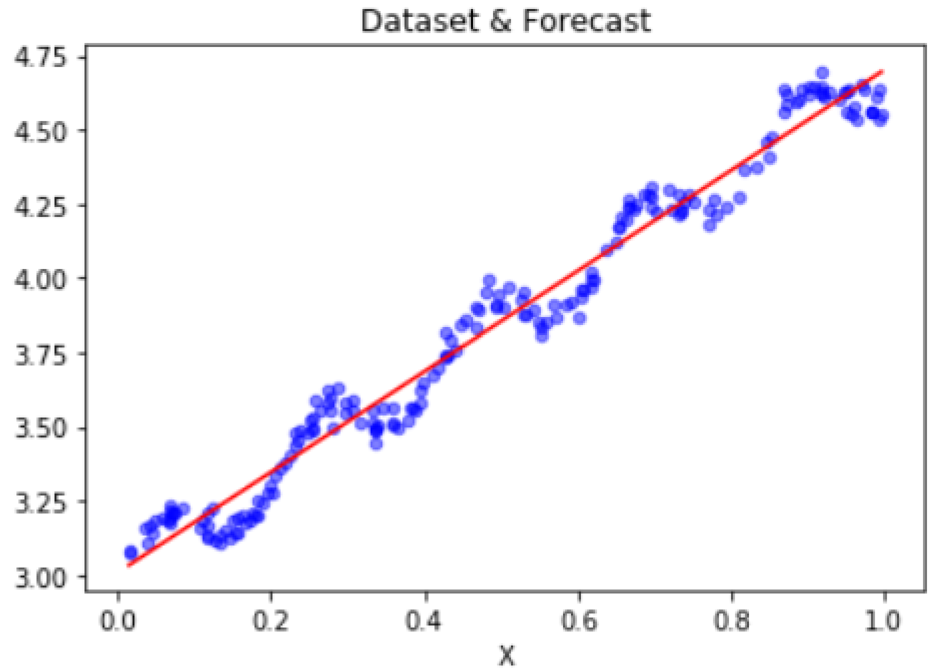
$$X^T(Y - X\omega)$$

Setting this to zero and solve for ω to get the following equation

$$\hat{\omega} = (X^TX)^{-1}X^TY$$

II. Linear Regression





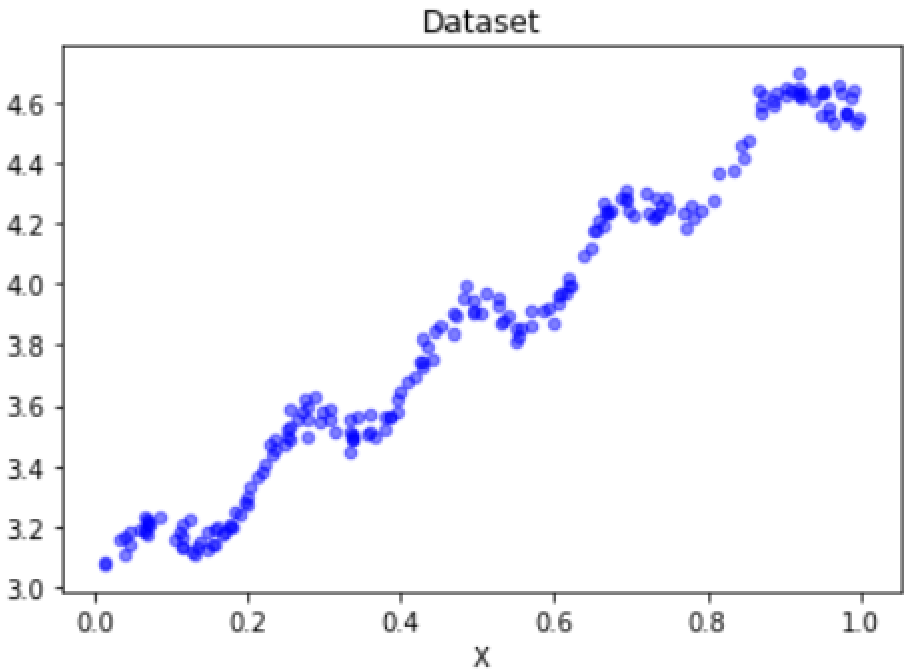
$$\hat{\omega} = (X^T X)^{-1} X^T Y$$

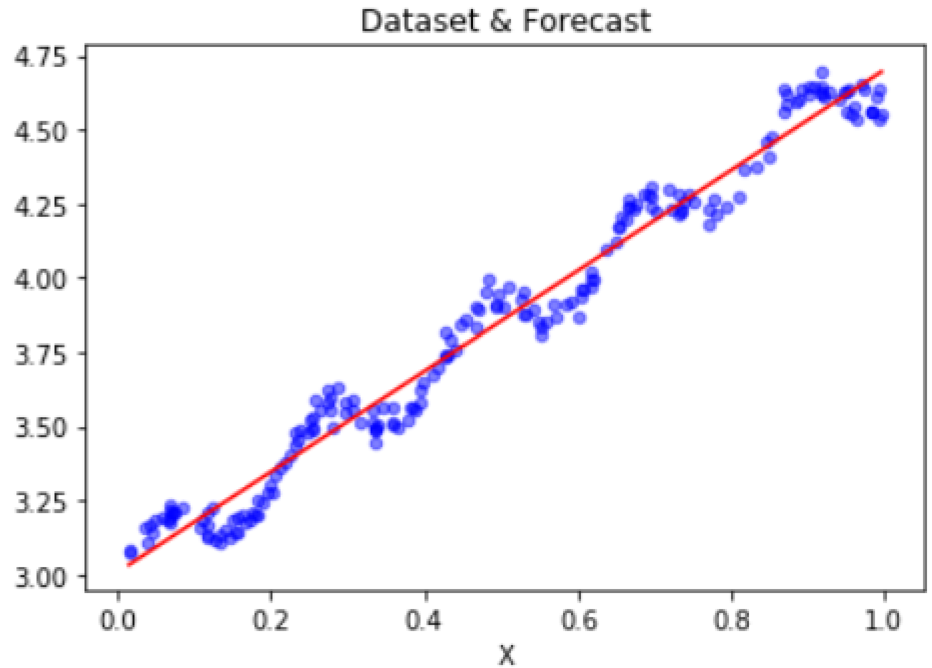
Y = w_1 * X + w_0

weights: [[3.00774324]
[1.69532264]]

w_0 : 3.0
w_1 : 1.7

II. Linear Regression



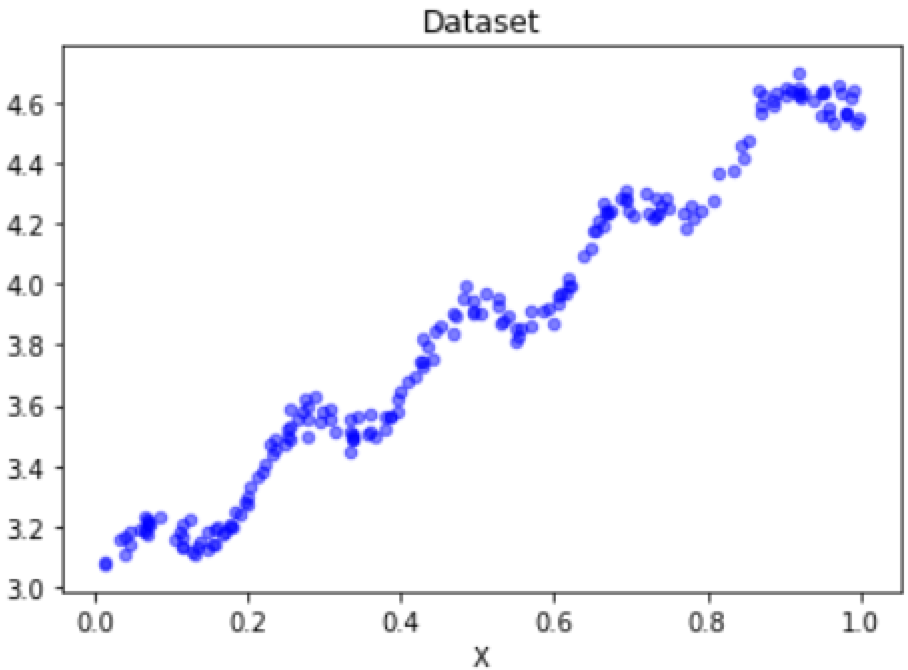


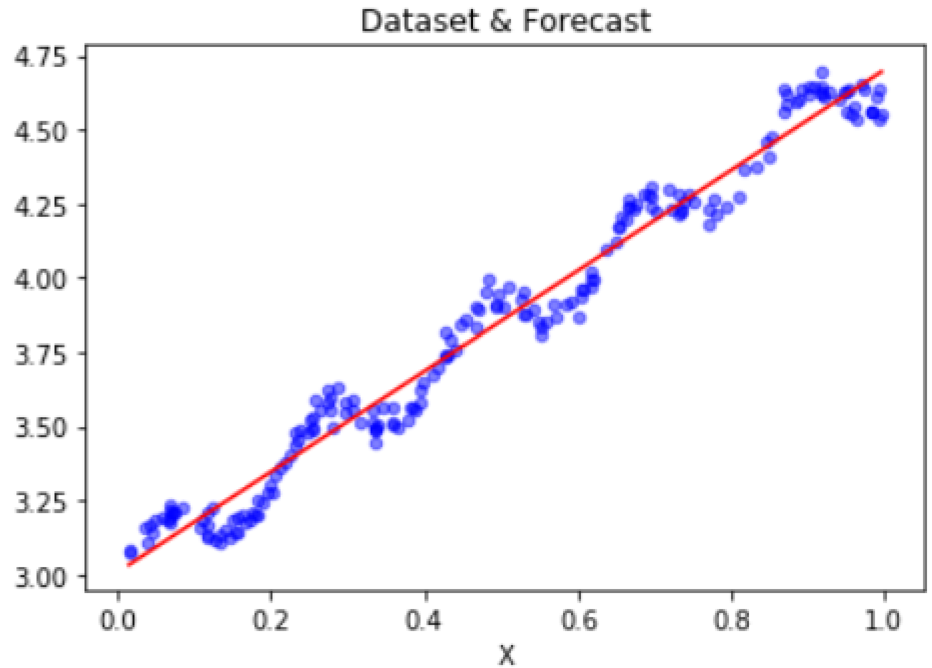
But how far do they differ from each other?

numpy.corrcoef(pred, actual): array([[1. , 0.98647356],
[0.98647356, 1.]])

LIKE

II. Linear Regression





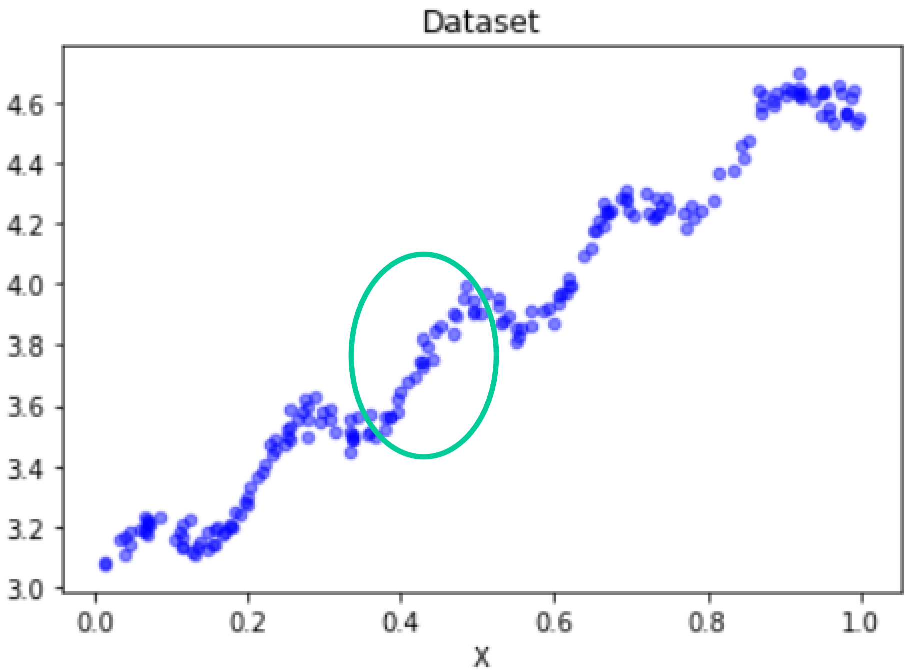
But how far do they differ from each other?

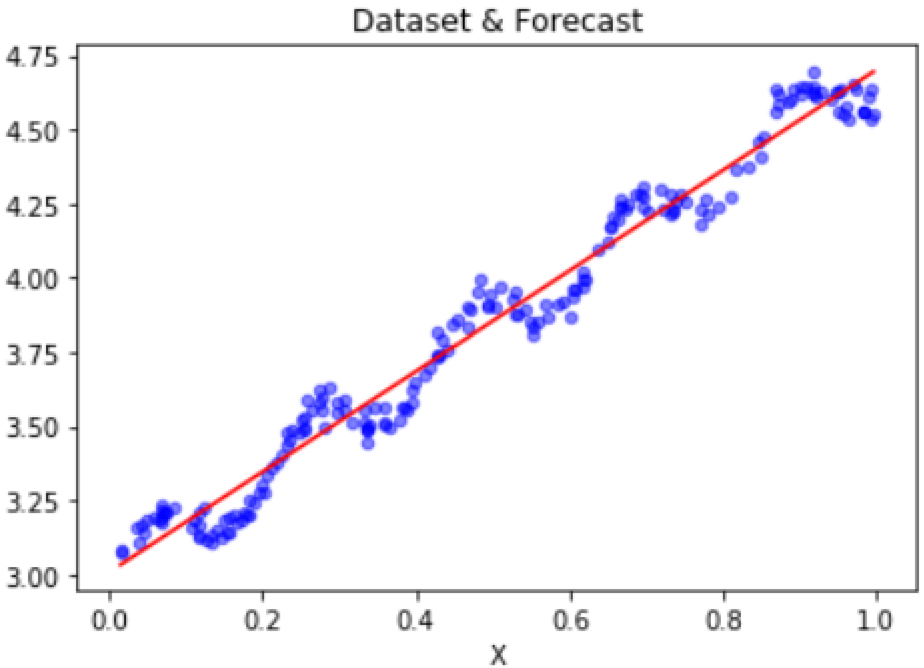
numpy.corrcoef(pred, actual): array([[1. , 0.98647356],
[0.98647356, 1.]])

More precise ?

LIKE

II. Linear Regression





But how far do they differ from each other?

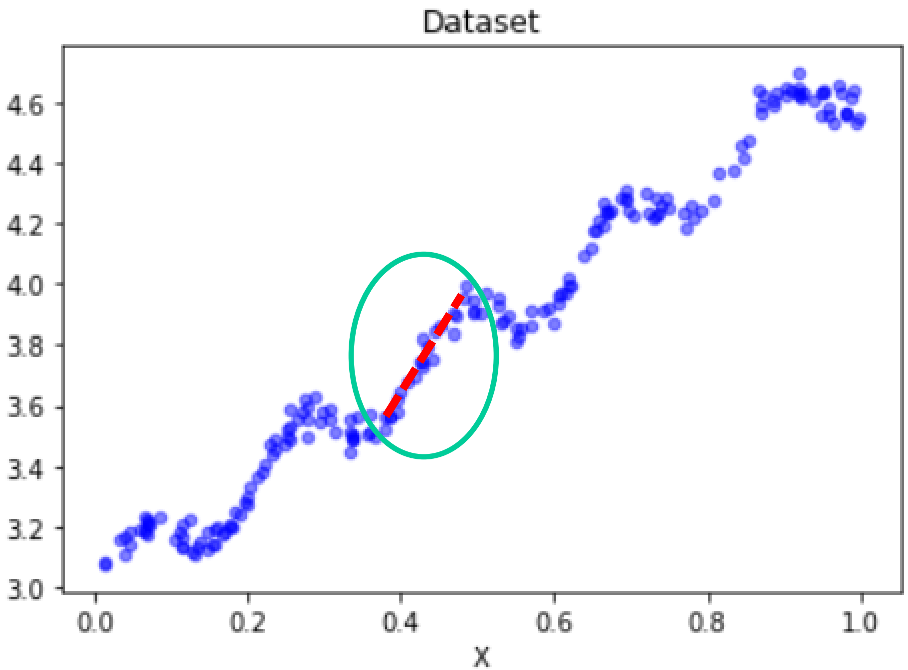
numpy.corrcoef(pred, actual): array([[1. , 0.98647356],

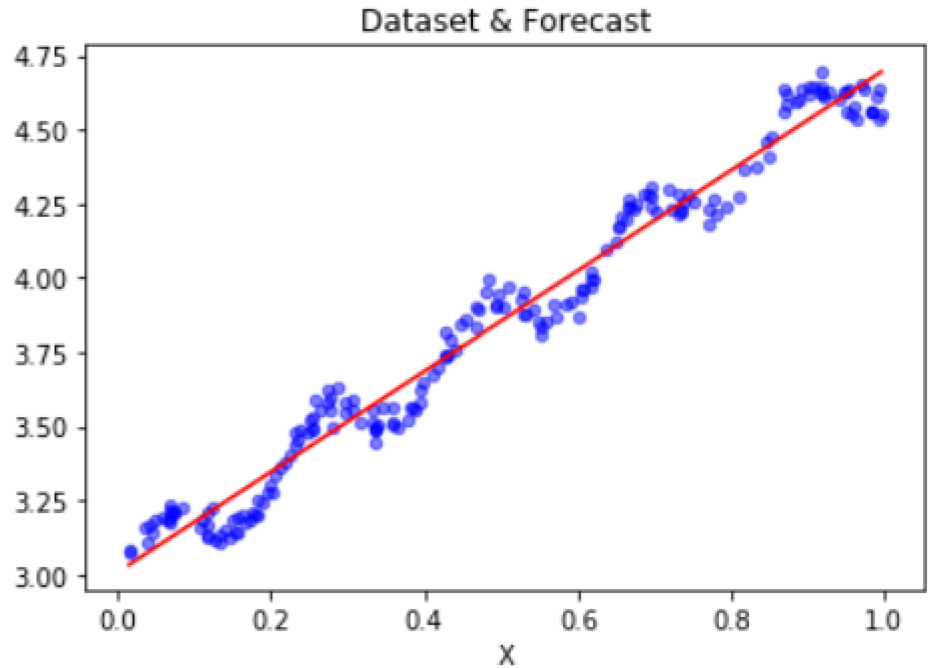
[0.98647356, 1.]])

More precise ?

LIKE

II. Linear Regression





But how far do they differ from each other?

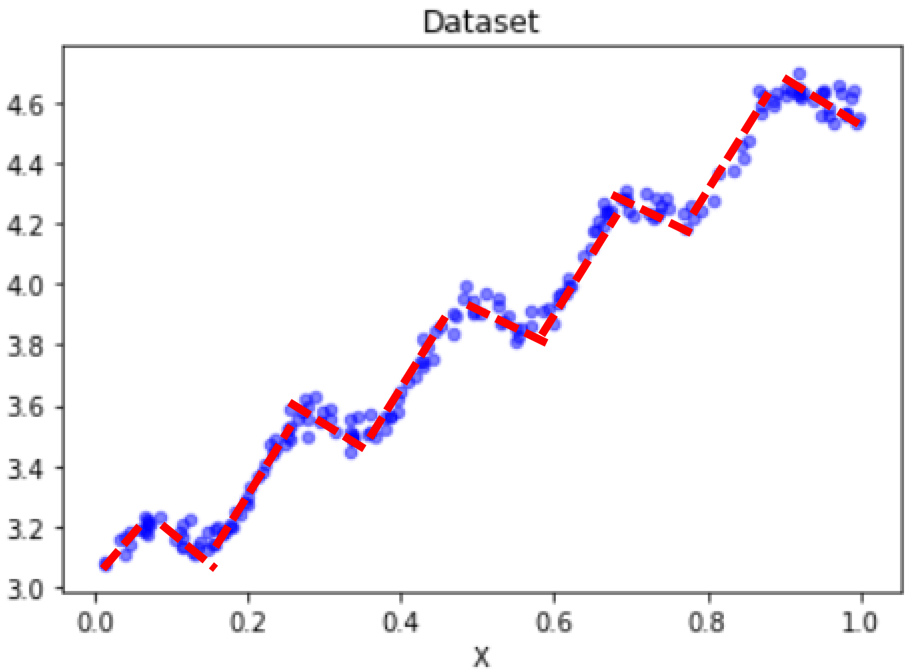
numpy.corrcoef(pred, actual): array([[1. , 0.98647356],

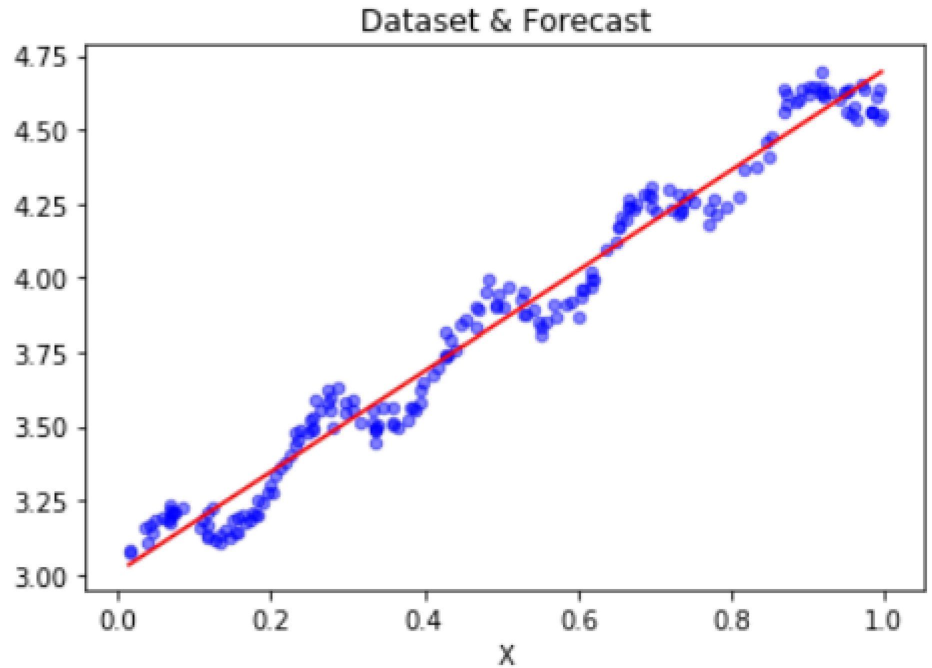
[0.98647356, 1.]])

More precise ?

LIKE

II. Linear Regression





But how far do they differ from each other?

numpy.corrcoef(pred, actual): array([[1. , 0.98647356],
[0.98647356, 1.]])

More precise ?

III. Locally Weighted Linear Regression

More weights are given to those data points which are close to the data point of interest, then the least-squares regression similar to the linear regression will be carried out.

$$\hat{\omega} = (X^T X)^{-1} X^T Y \longrightarrow \hat{\omega} = (X^T W X)^{-1} X^T W Y$$

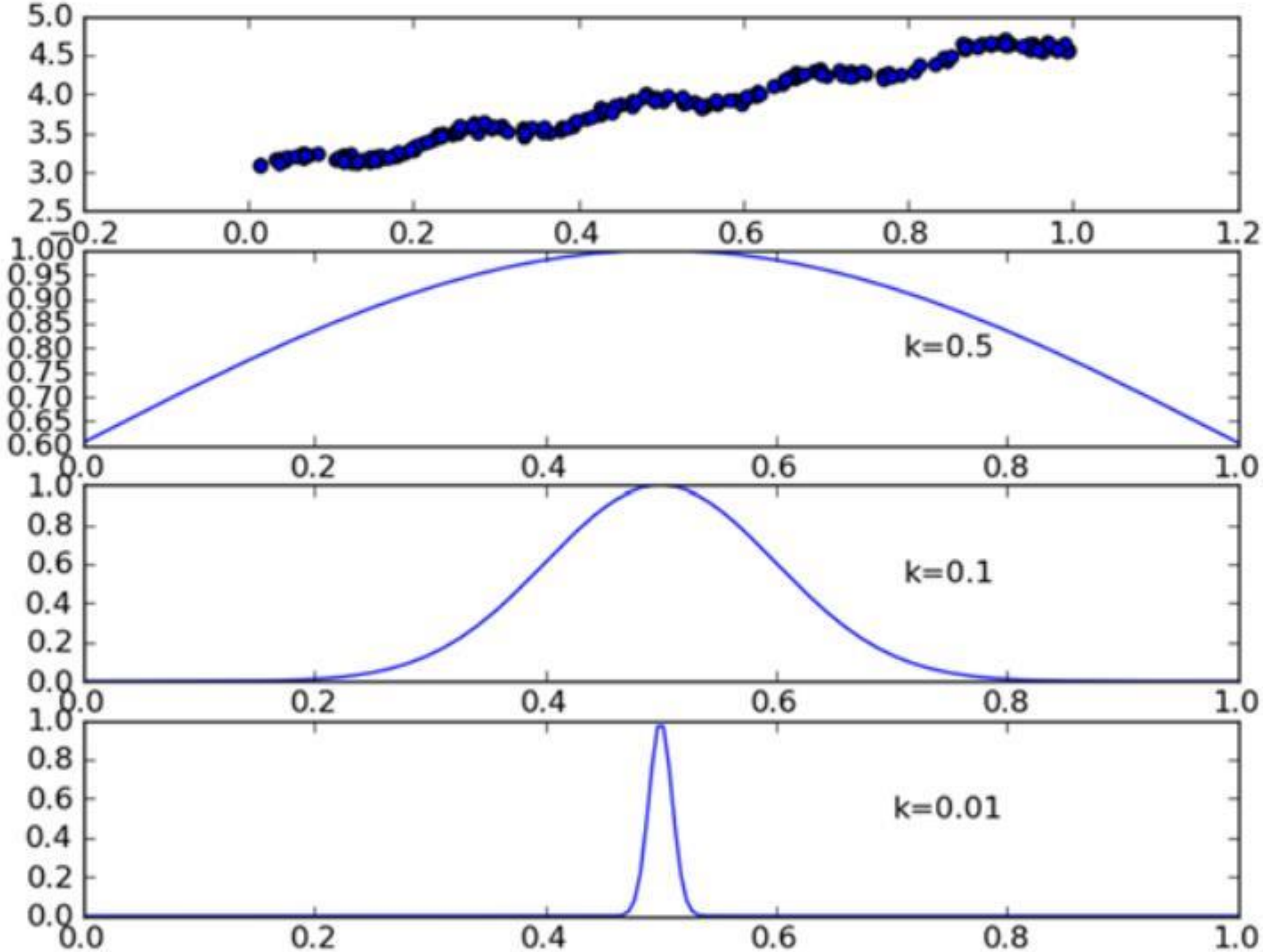
W is a matrix and will be generated by a ***kernel*** function, which shall give nearby points more weights than other points. The mostly used kernel is *Gaussian* and assigns the weights by

$$\omega(i,i) = exp\left(\frac{|x^i - x|}{-2\kappa^2}\right)$$

LIKE

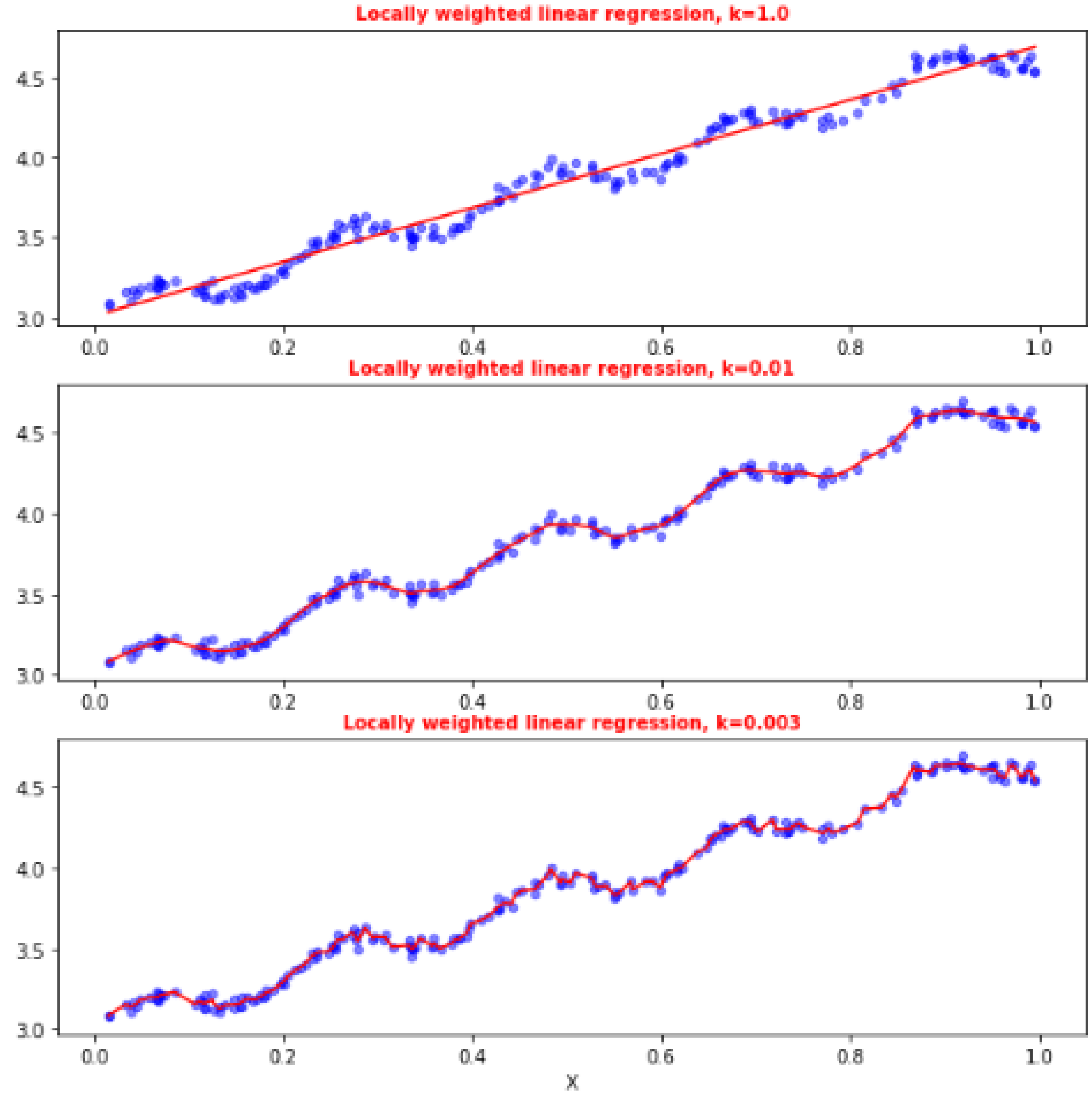
III. Locally Weighted Linear Regression

$$\omega(i,i) = exp\left(\frac{\left|x^i - x\right|}{-2\kappa^2}\right)$$



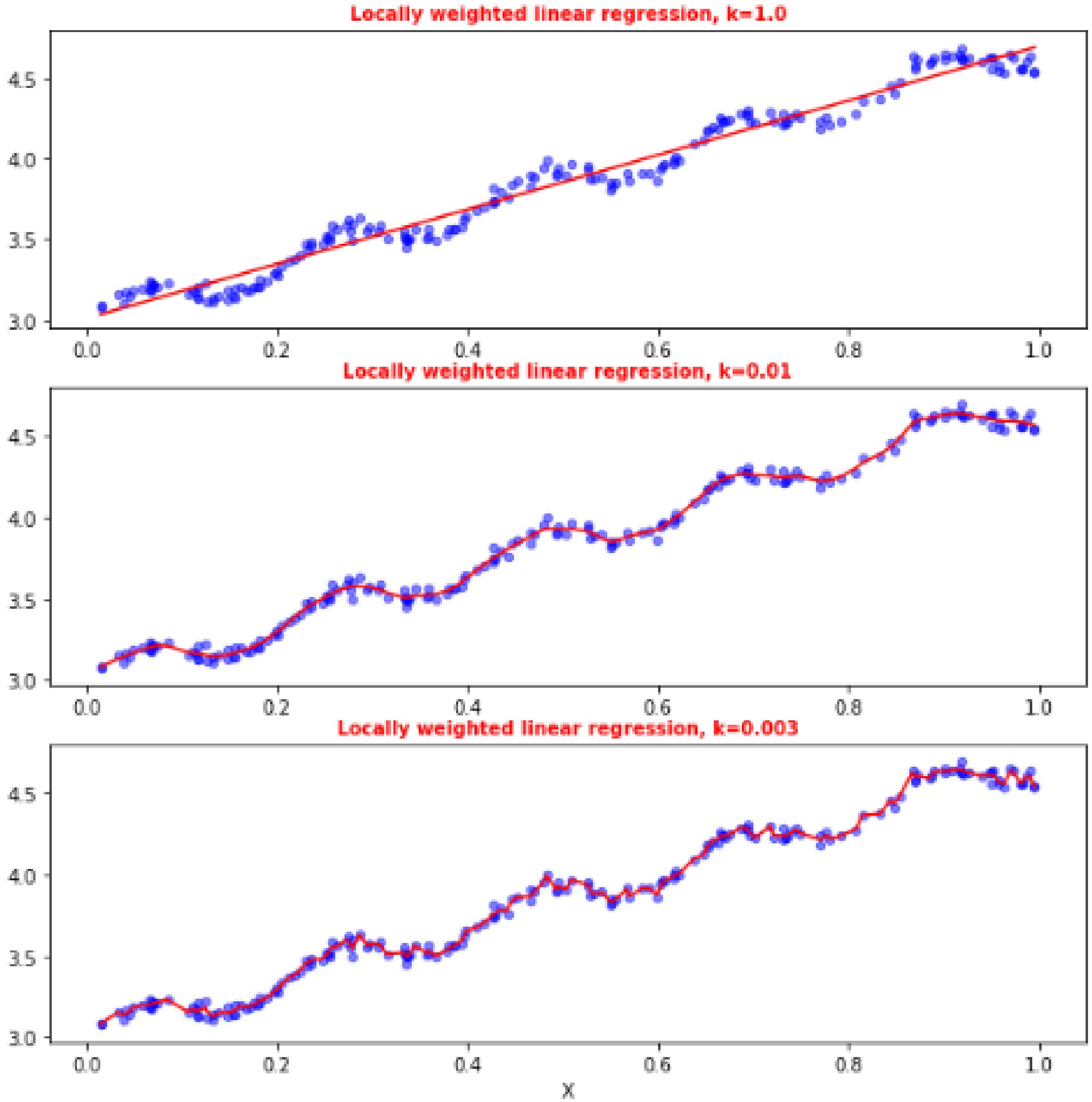
LIKE

III. Locally Weighted Linear Regression



LIKE

III. Locally Weighted Linear Regression



overfitting!

LIKE

Example: predicting the age of an abalone

Testdata: abalone.txt

Feature_X:								Age_Y:
[[1.	0.455	0.365	0.095	0.514	0.2245	0.101	0.15]	[[15.]
[1.	0.35	0.265	0.09	0.225	0.0995	0.048	0.07]	[7.]
[-1.	0.53	0.42	0.135	0.677	0.2565	0.141	0.21]	[9.]
[…	…	…	…	…	…	…	…]	[10.]
[0.	0.33	0.255	0.08	0.205	0.0895	0.039	0.05]]	[7.]]

<u>Training</u> set and <u>test</u> set are *identical*:

k=0.1, squared error : 56.82523568972884
k=1.0, squared error : 429.8905618700651
k=10, squared error : 549.1181708826451

LIKE

Example: predicting the age of an abalone

Training set and test set are *identical*:

k=0.1, squared error : 56.82523568972884
k=1.0, squared error : 429.8905618700651
k=10, squared error : 549.1181708826451

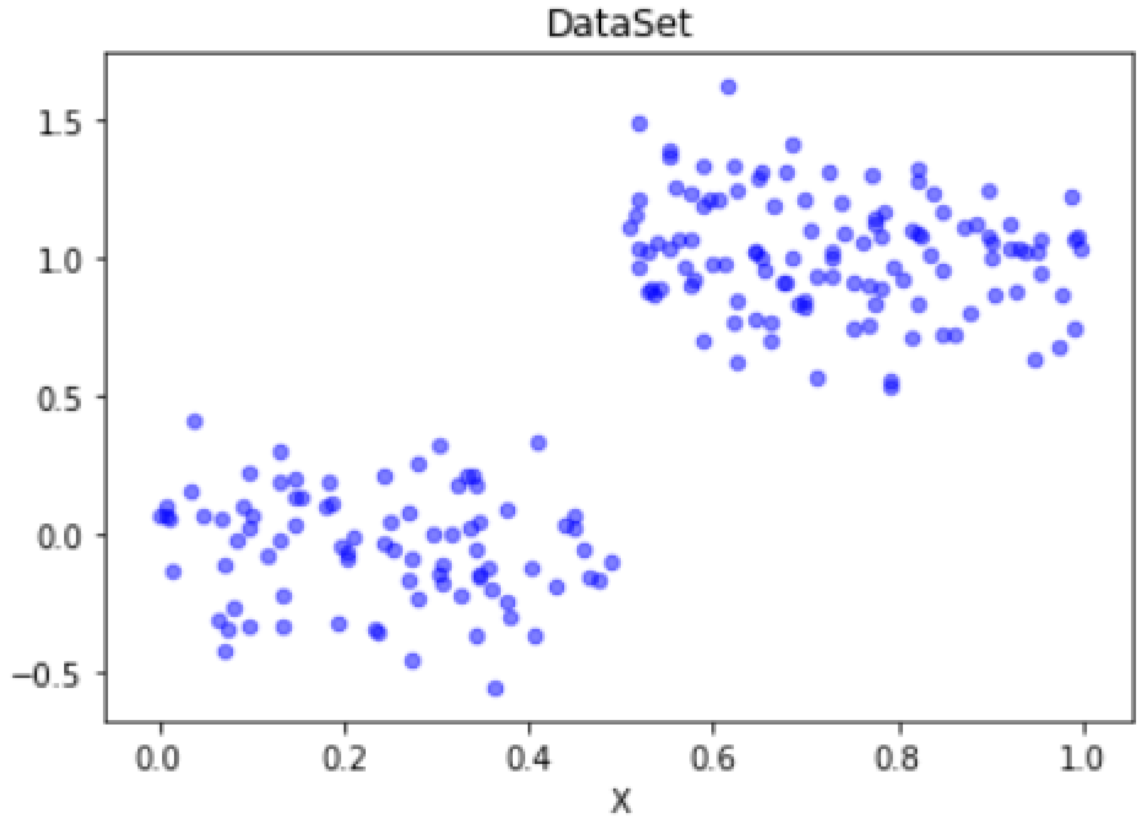
Training set and test set are *different*:

k=0.1, squared error : 41317.161723642595
k=1.0, squared error : 573.526144189767
k=10, squared error : 517.5711905387598

Linear regression: squared error : 518.6363153249638

LIKE

How to deal with the *nonlinearities* in real life?



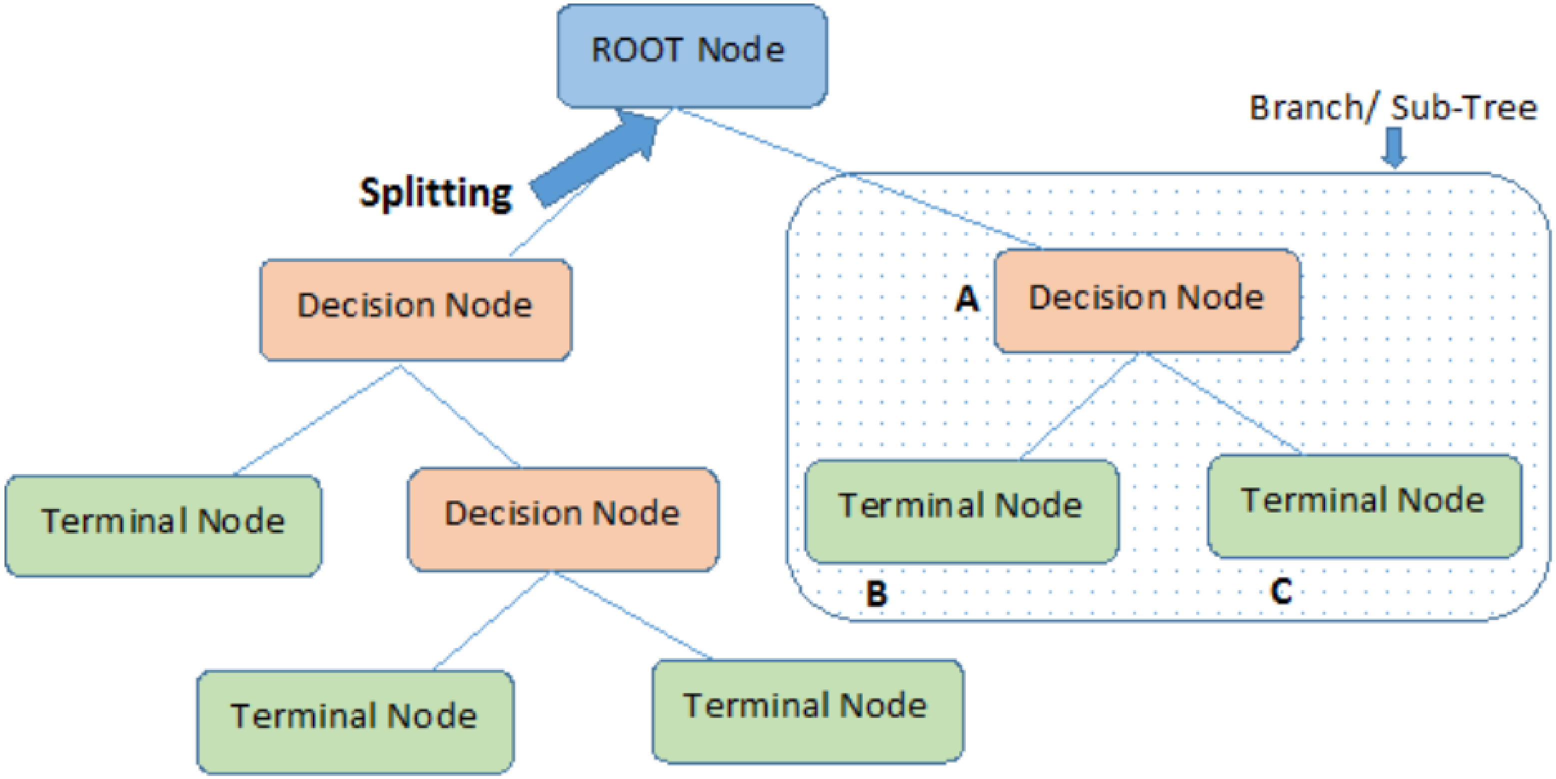
LIKE

How to deal with the *nonlinearities* in real life?

Tree Regression

IV: Tree-based Regression

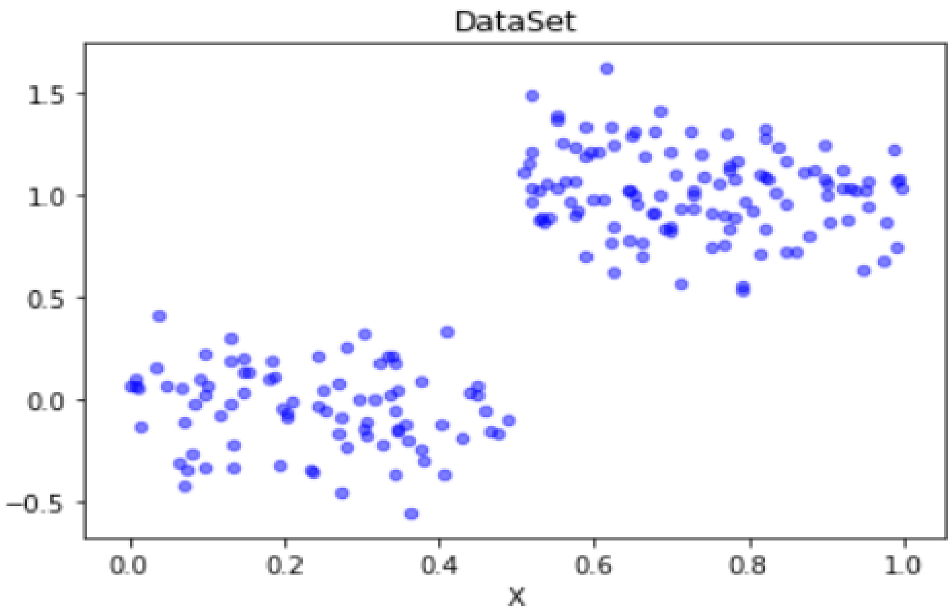
Tree structure



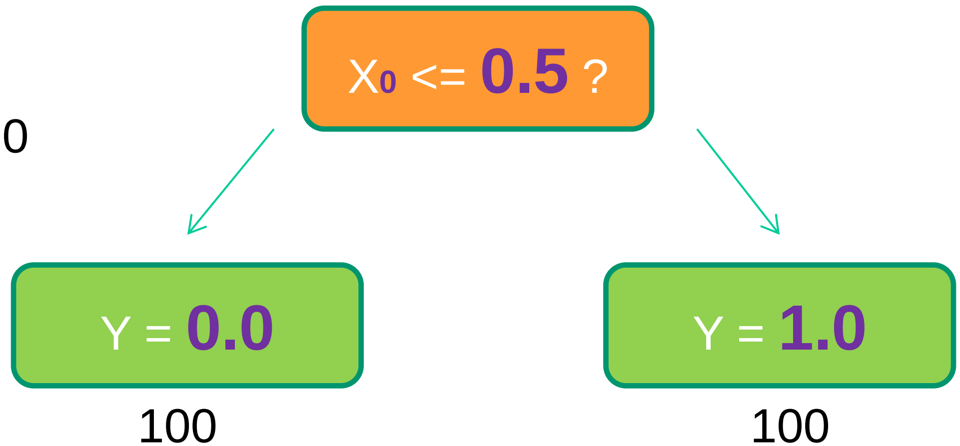
Note:- A is parent node of B and C.

LIKE

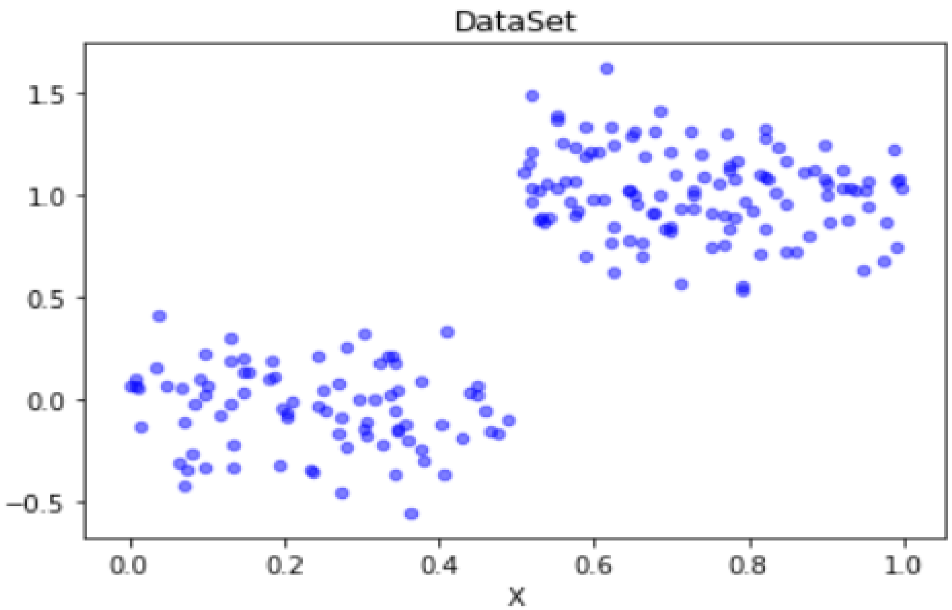
IV: Tree-based Regression



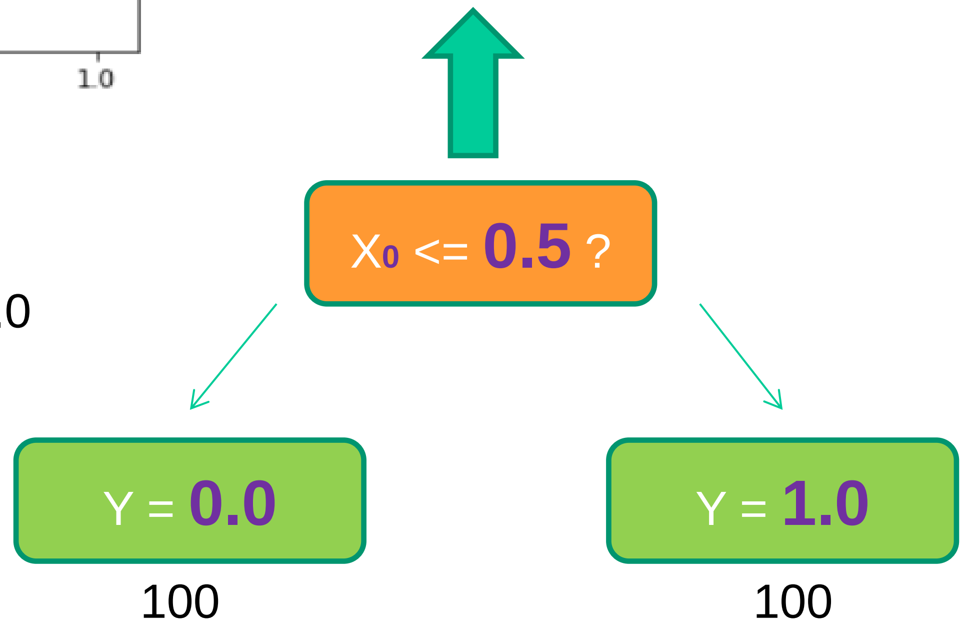
Feature Index: 0
Threshold value: 0.5
Leaf node value: 0.0 or 1.0
Number of Samples: 100



IV: Tree-based Regression



How to find the splitting feature and splitting point ?



Feature Index:	0
Threshold value:	0.5
Leaf node value:	0.0 or 1.0
Number of Samples:	100

IV: Tree-based Regression

How to deal with the ***nonlinearities*** in real life?

Tree regression:

It implements a new algorithm called **CART** (**C**lassification **A**nd **R**egression **T**rees). It is well-known and well-documented tree-building algorithm that makes ***binary splits*** to handle continuous variables.

By doing this we choose a **feature** and make **values** *greater* than the desired go on the **right** side of the tree and all the other values go on the **left** side.

LIKE

How to make a binary split?

We need to select: splitting *feature 'xj'* and a splitting *point 's'* so that we can divide data into two regions R1 and R2:

$$R_1(j,s) = \{x | x^{(j)} \le s\}, \quad R_2(j,s) = \{x | x^{(j)} > s\}$$

Then we calculate the *average* value for each generated region by

$$\hat{c}_1 = ave(y_i | x_i \in R_1(j,s)), \quad \hat{c}_2 = ave(y_i | x_i \in R_2(j,s))$$

Goal: find such (ĉ 1, ĉ 2) which gives the **minimum** of total squared error as follows

$$\min_{j,s} [\min_{\hat{c}_1} \sum_{x_i \in R_1(j,s)} (y_i - \hat{c}_1)^2 + \min_{\hat{c}_2} \sum_{x_i \in R_2(j,s)} (y_i - \hat{c}_2)^2]$$

LIKE

Example: binary split in the tree regression

Let's split it by the value of the second feature, threshold value: 0.5

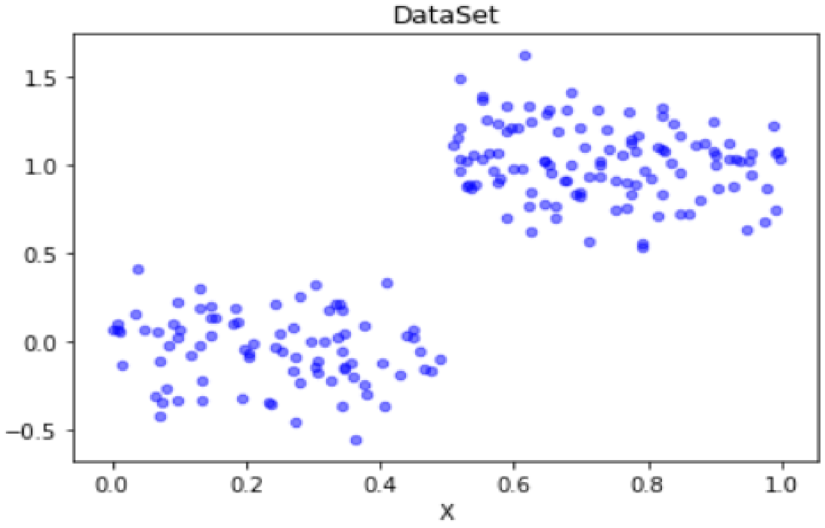
```
matrix([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]])
```

```
Left: [[1. 0. 0. 0.]
       [0. 0. 1. 0.]
       [0. 0. 0. 1.]]
```

```
Right: [[0. 1. 0. 0.]]
```

LIKE

IV: Tree-based Regression



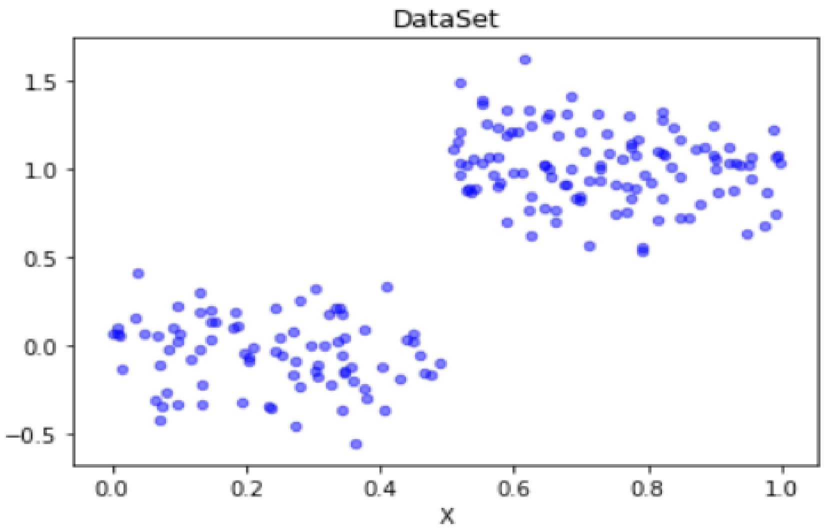
Stop Conditions:

Mind. error reduction: 1

Mind. data instances: 4

LIKE

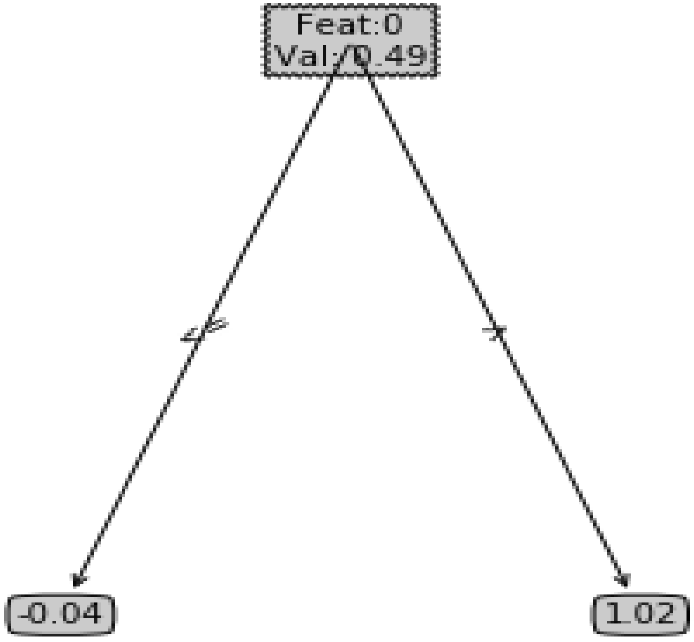
IV: Tree-based Regression



Stop Conditions:

Mind. error reduction: 1

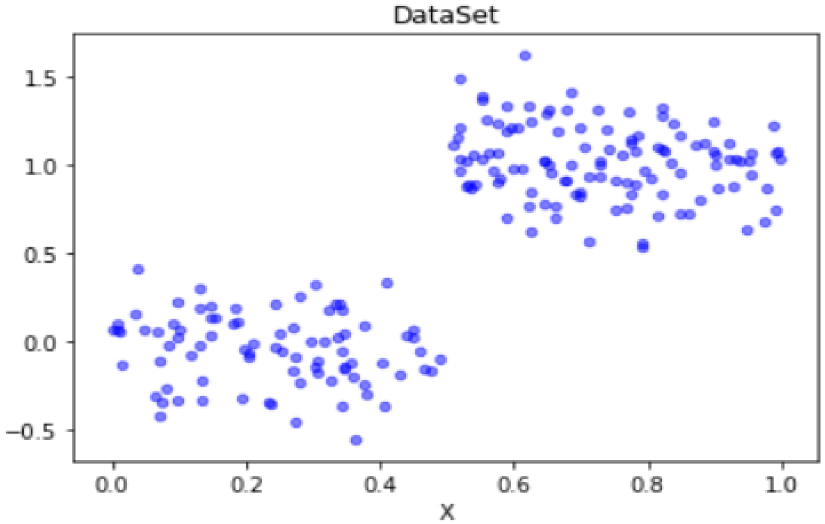
Mind. data instances: 4

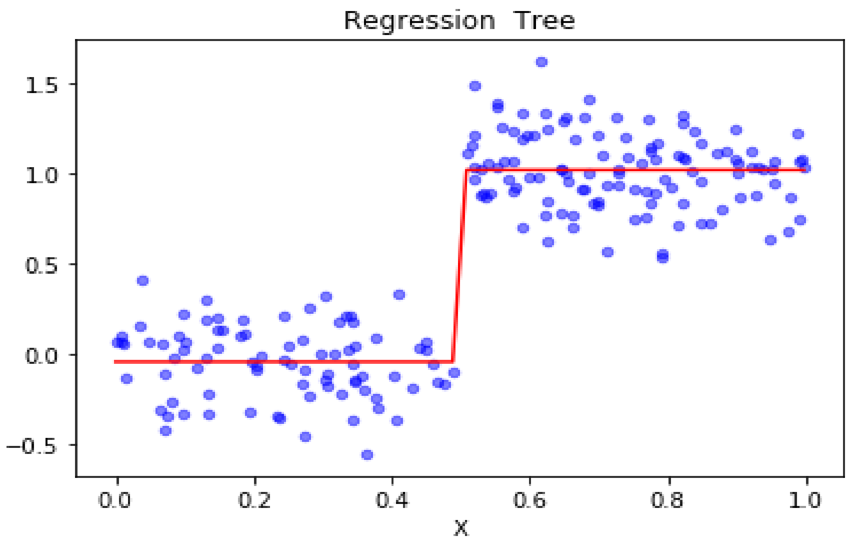



```
print(createTree(data_mat))
{'spInd': 0,
'spVal': 0.48813,
'left': -0.044650285714285714,
'right': 1.0180967672413792 }
```

LIKE

IV: Tree-based Regression

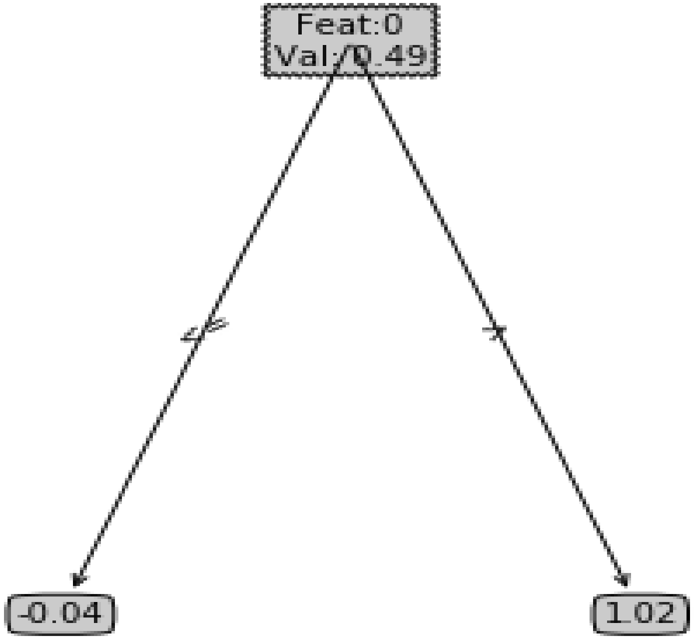




Stop Conditions:

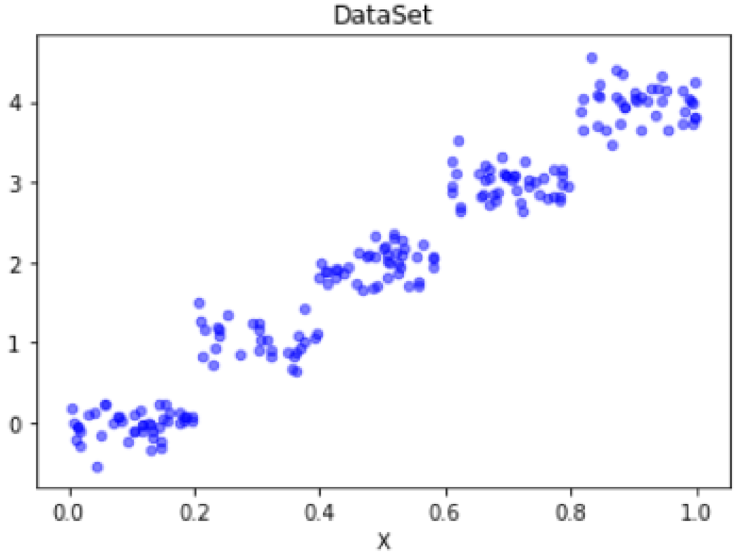
Mind. error reduction: 1

Mind. data instances: 4



LIKE

IV: Tree-based Regression

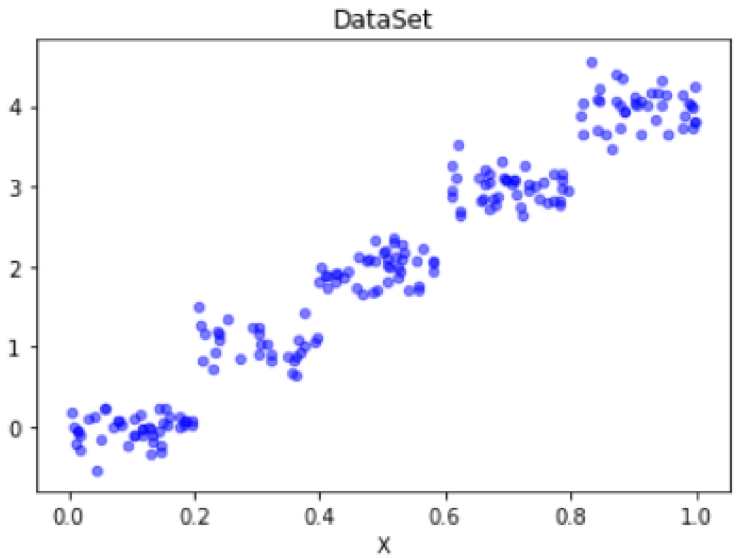


Stop Conditions:

Mind. error reduction: 1

Mind. data instances: 4

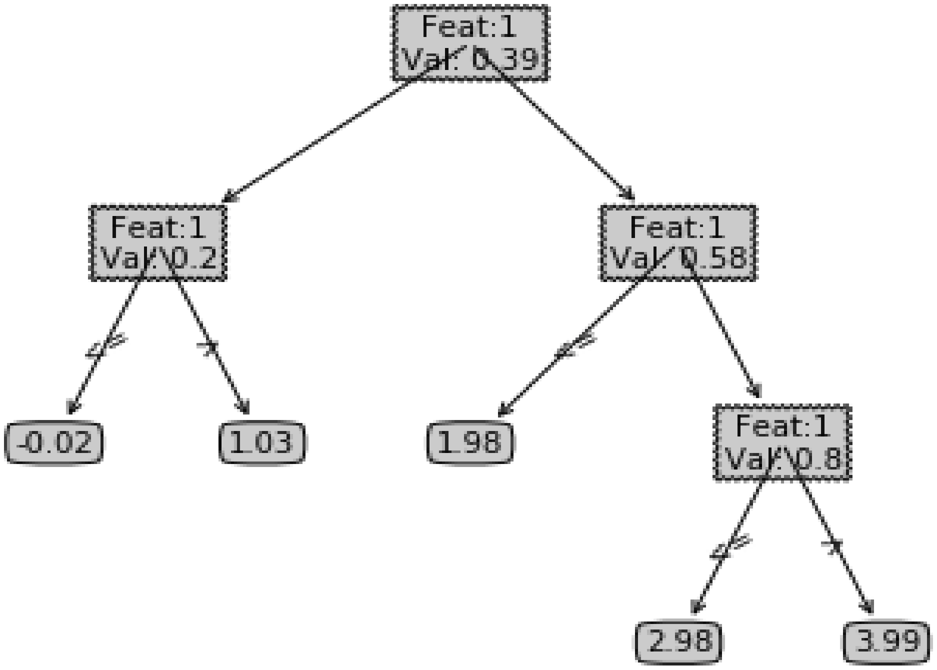
IV: Tree-based Regression



Stop Conditions:

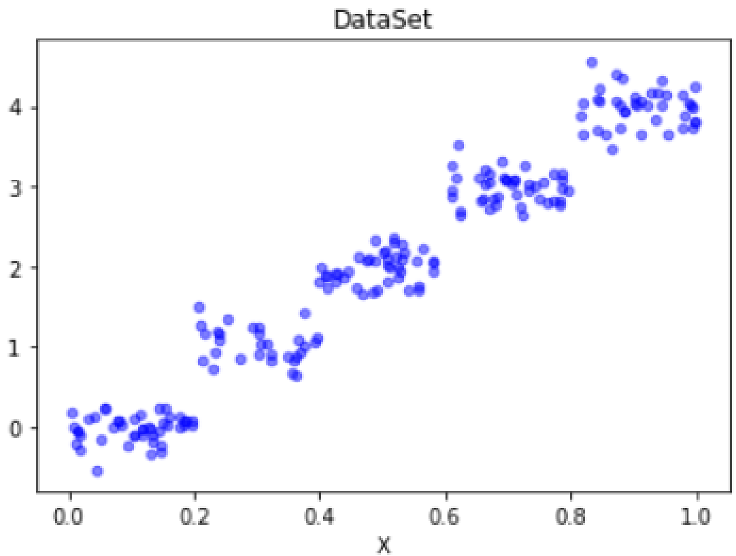
Mind. error reduction: 1

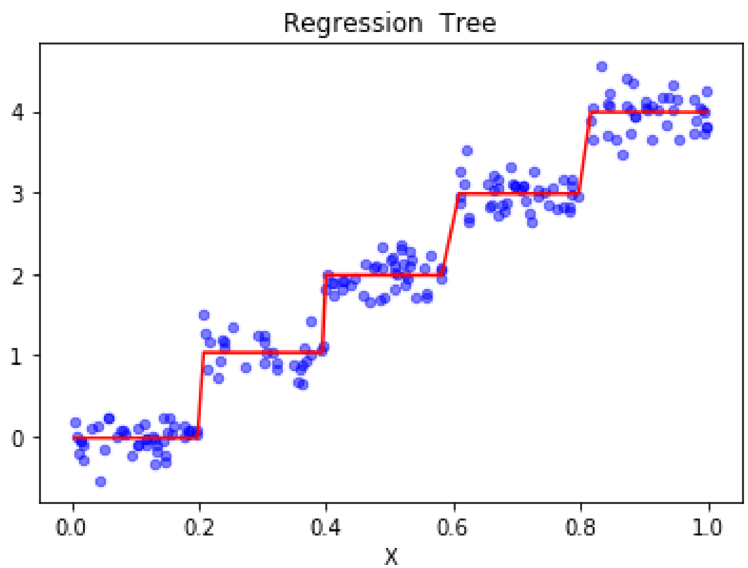
Mind. data instances: 4



LIKE

IV: Tree-based Regression

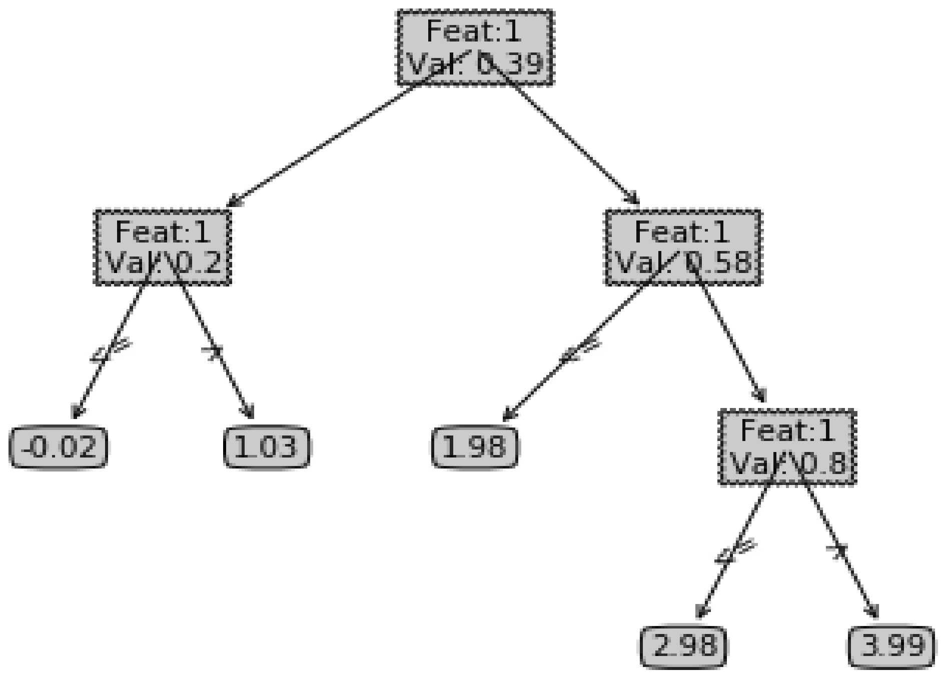




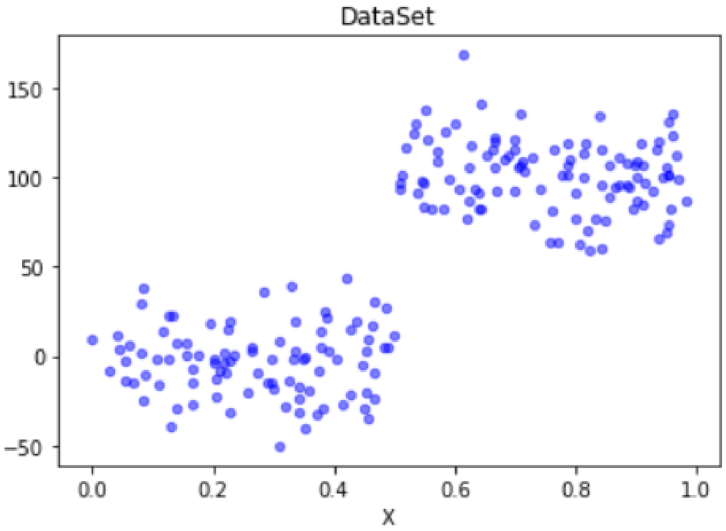
Stop Conditions:

Mind. error reduction: 1

Mind. data instances: 4



IV: Tree-based Regression

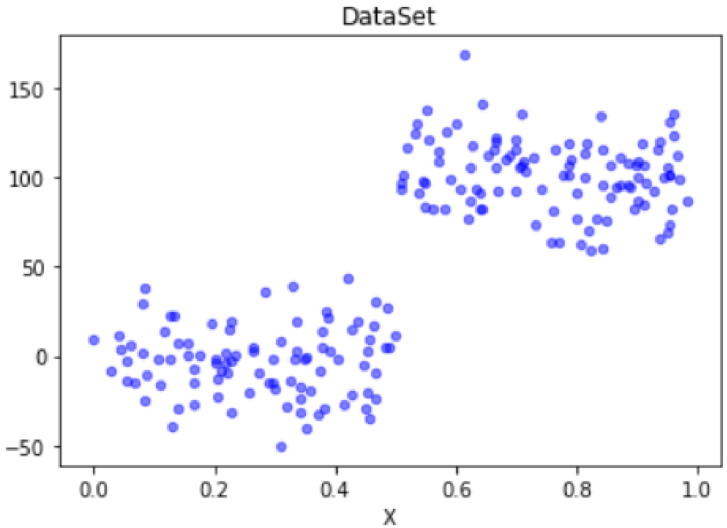


Stop Conditions:

Mind. error reduction: 1

Mind. data instances: 4

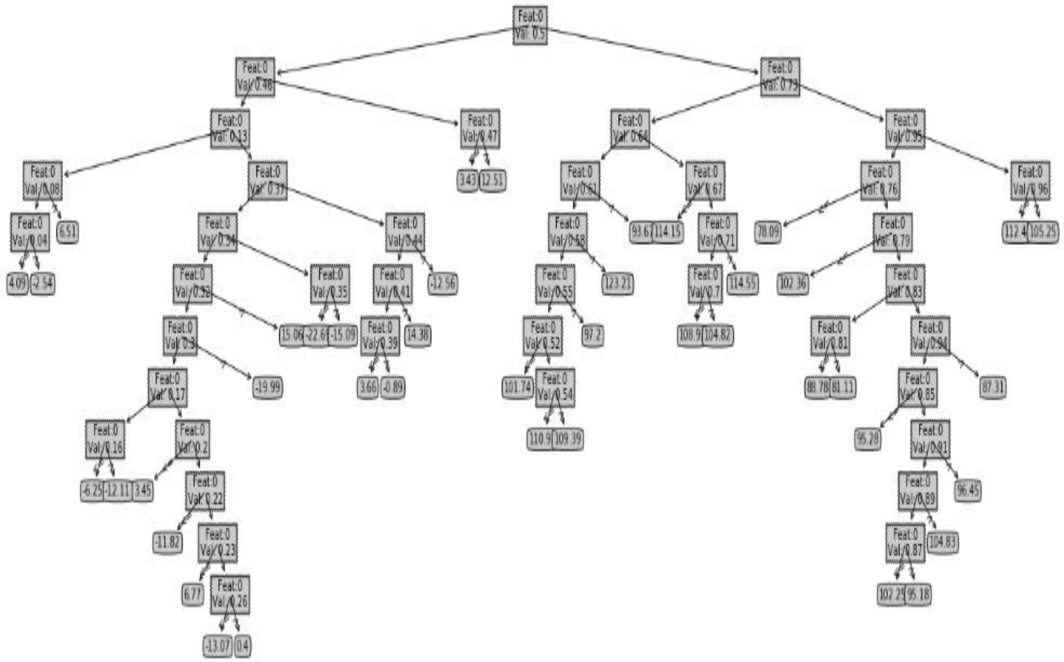
IV: Tree-based Regression



Stop Conditions:

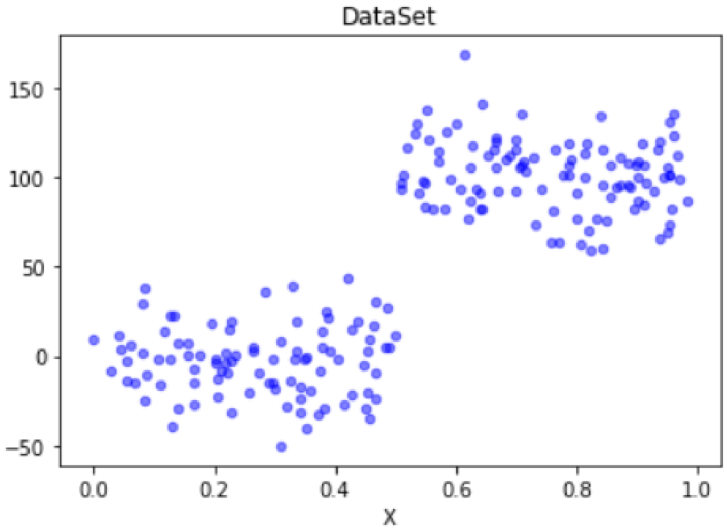
Mind. error reduction: 1

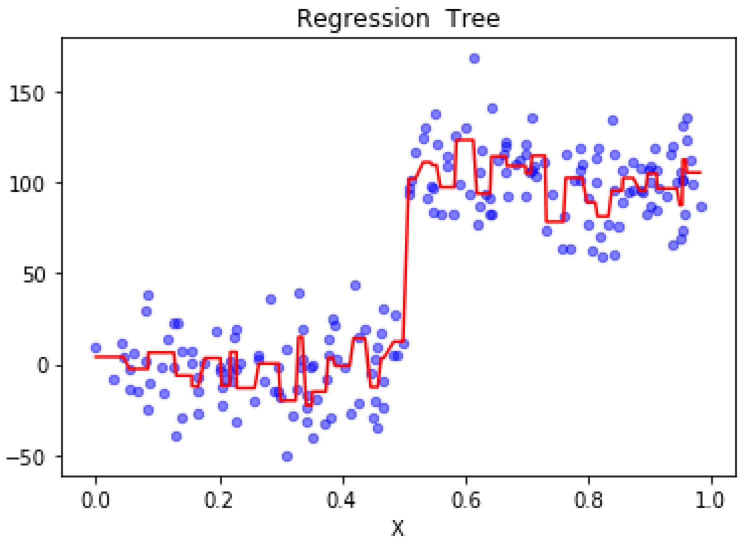
Mind. data instances: 4



LIKE

IV: Tree-based Regression

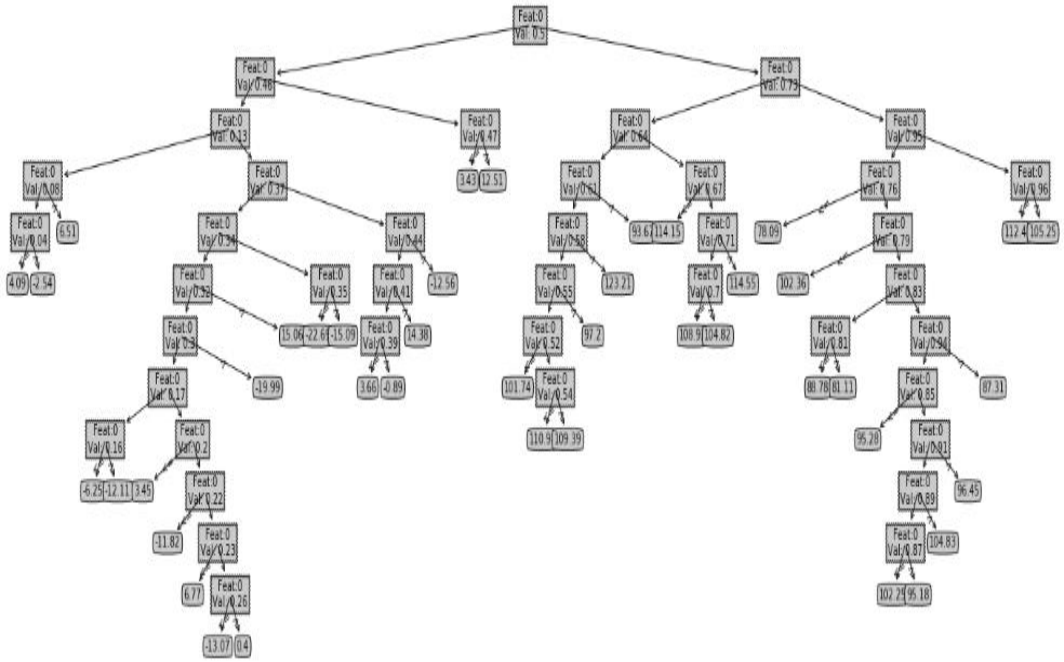




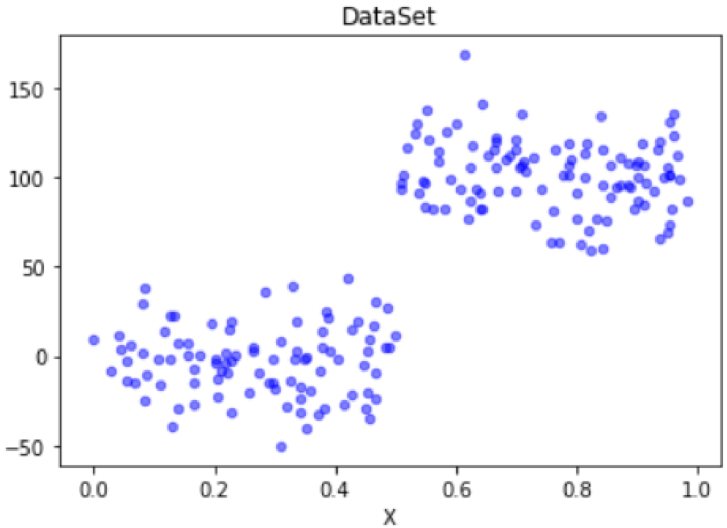
Stop Conditions:

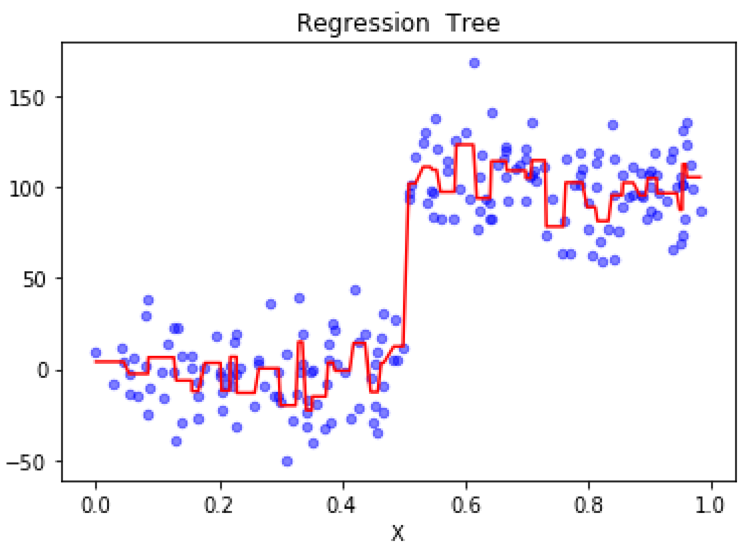
Mind. error reduction: 1

Mind. data instances: 4



IV: Tree-based Regression

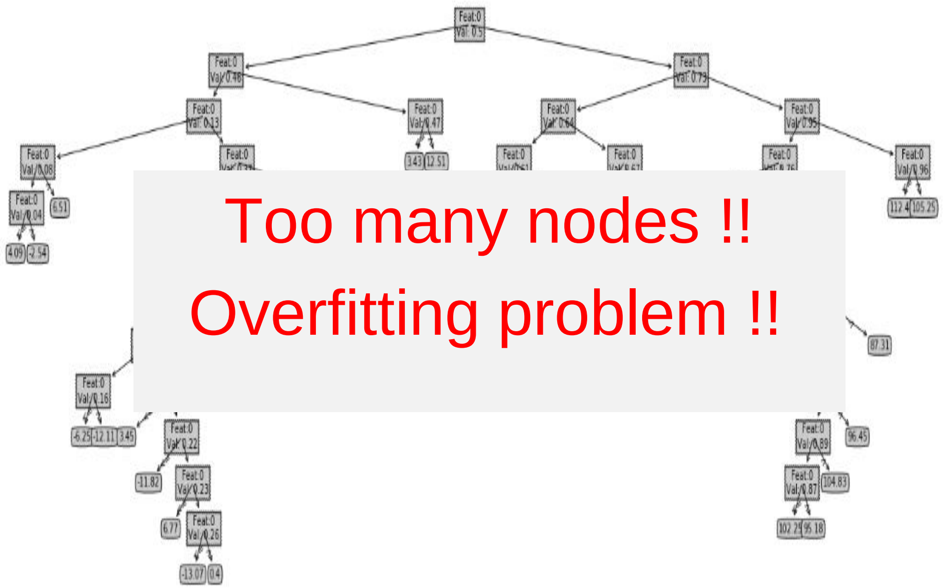




Stop Conditions:

Mind. error reduction: 1

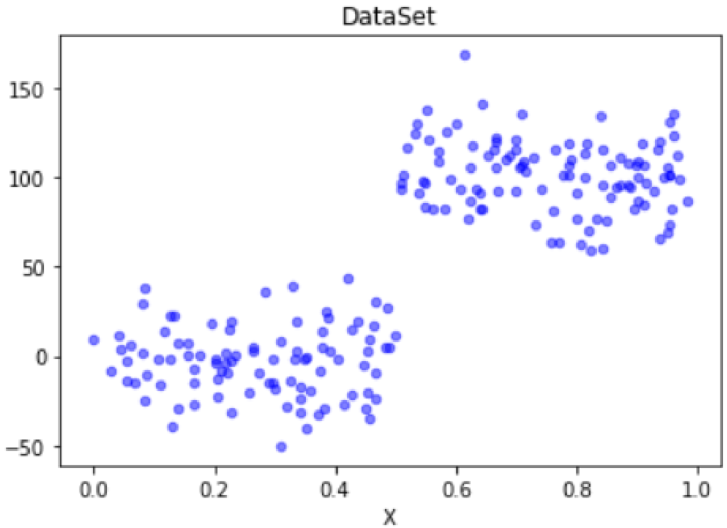
Mind. data instances: 4



Too many nodes !!

Overfitting problem !!

IV: Tree-based Regression



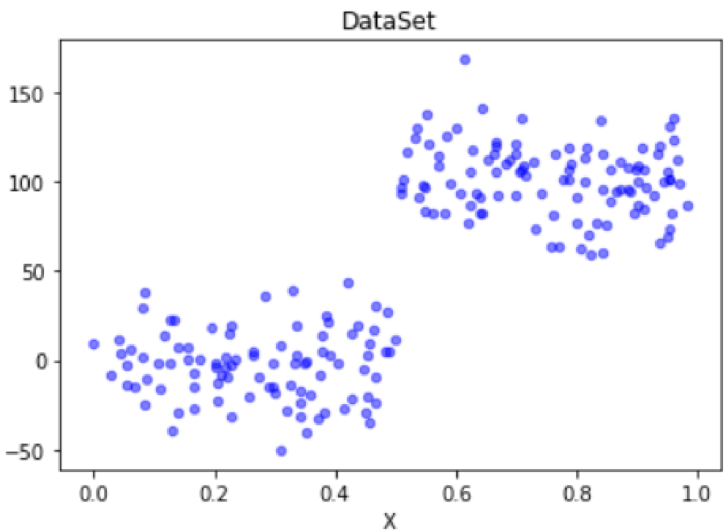
Stop Conditions:

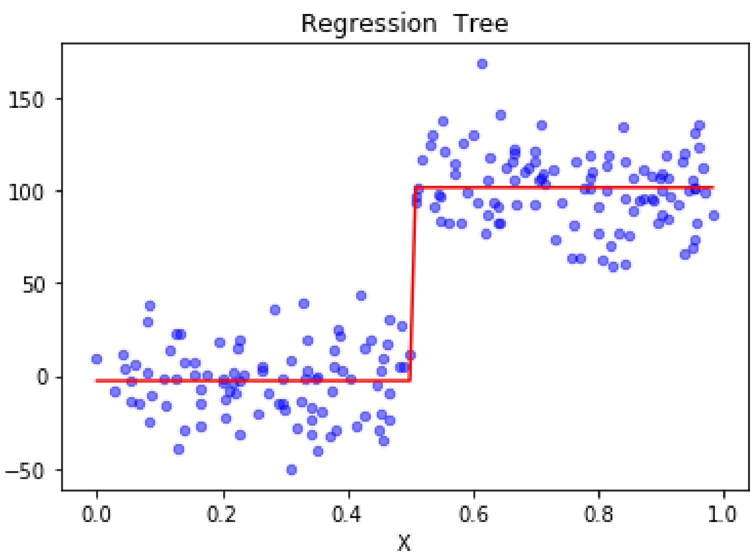
Mind. error reduction: ~~1~~ 10000

Mind. data instances: 4

LIKE

IV: Tree-based Regression

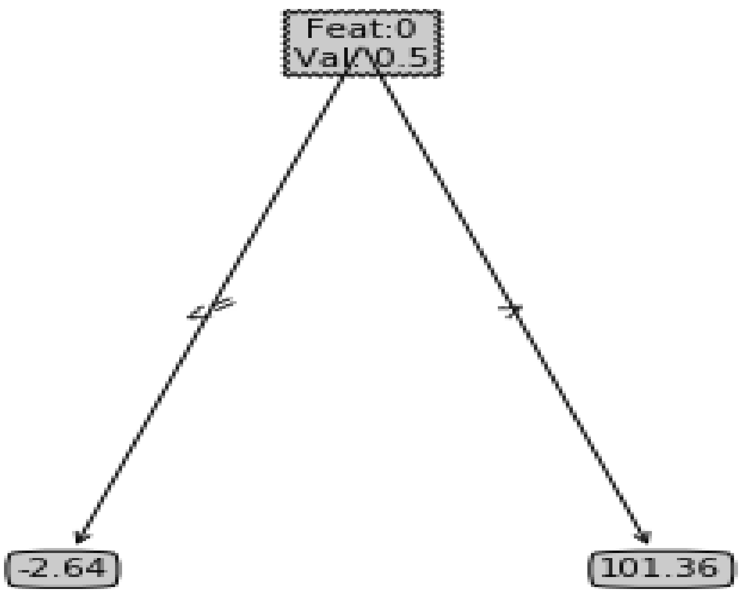




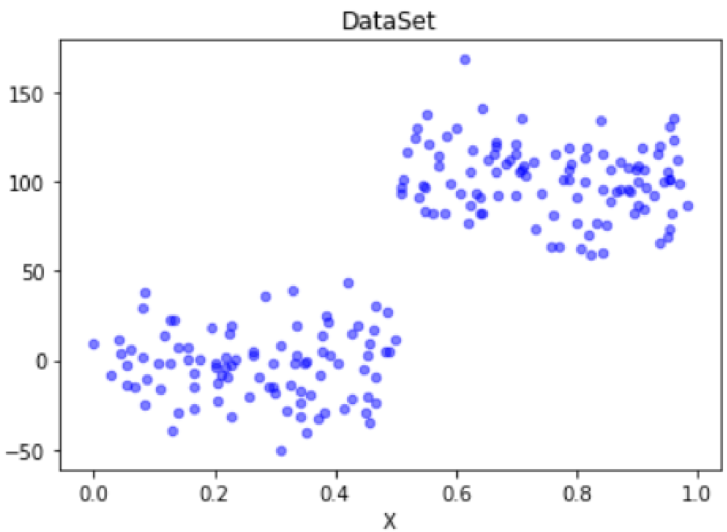
Stop Conditions:

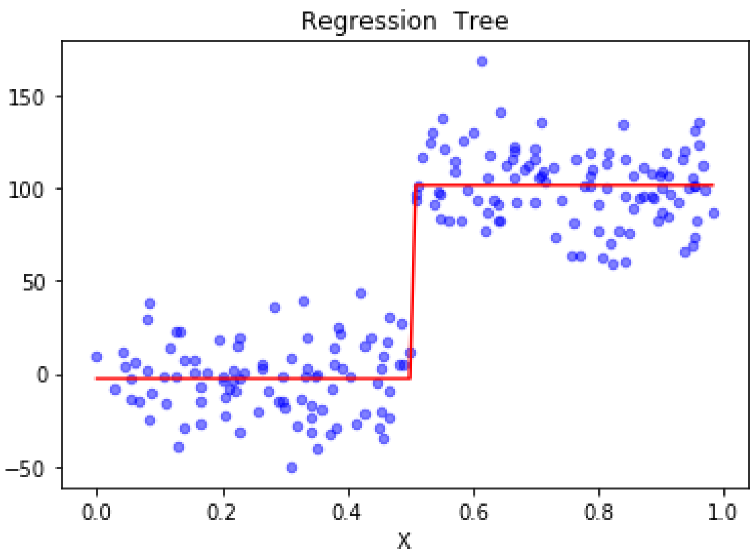
Mind. error reduction: ~~1~~ 10000

Mind. data instances: 4



IV: Tree-based Regression





Stop Conditions:

Mind. error reduction: ~~1~~ 10000

Mind. data instances: 4

Is there any solution without *user intervention*?

→ Postpruning

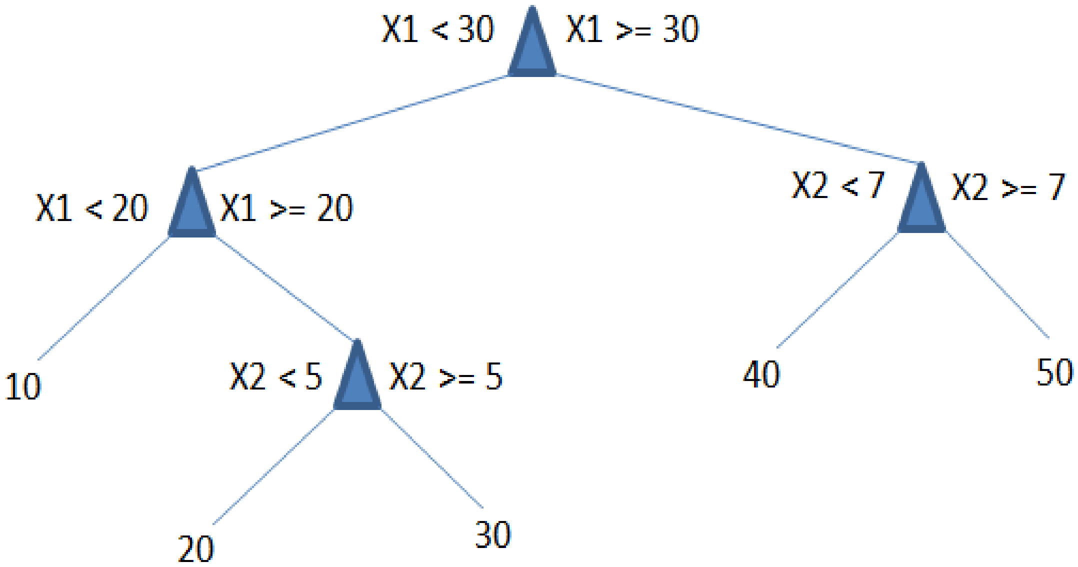
LIKE

How to implement the postpruning?

- Make the regression tree to a large depth
- Start at the bottom and combine every two leaf nodes(left and right) into a new terminal node, if such error after merge is smaller than the original one

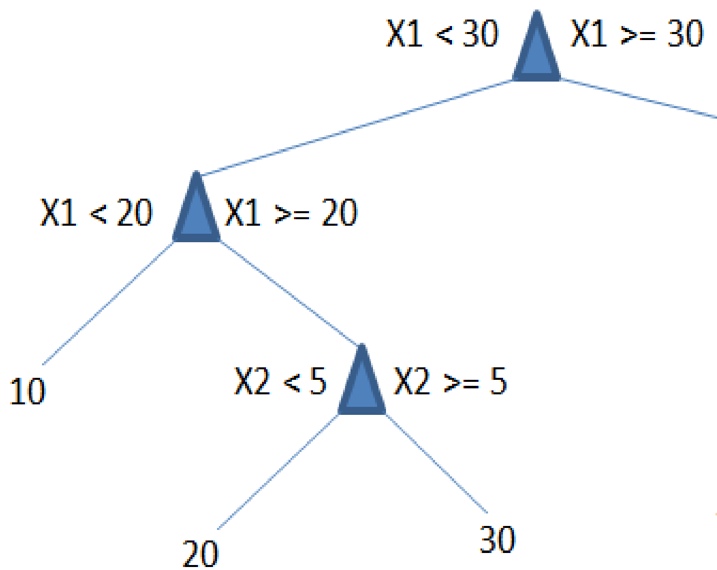
How to implement the postpruning?

- Make the regression tree to a large depth
- Start at the bottom and combine every two leaf nodes(left and right) into a new terminal node, if such error after merge is smaller than the original one



How to implement the postpruning?

- Make the regression tree to a large depth
- Start at the bottom and combine every two leaf nodes(left and right) into a new terminal node, if such error after merge is smaller than the original one



Test_data

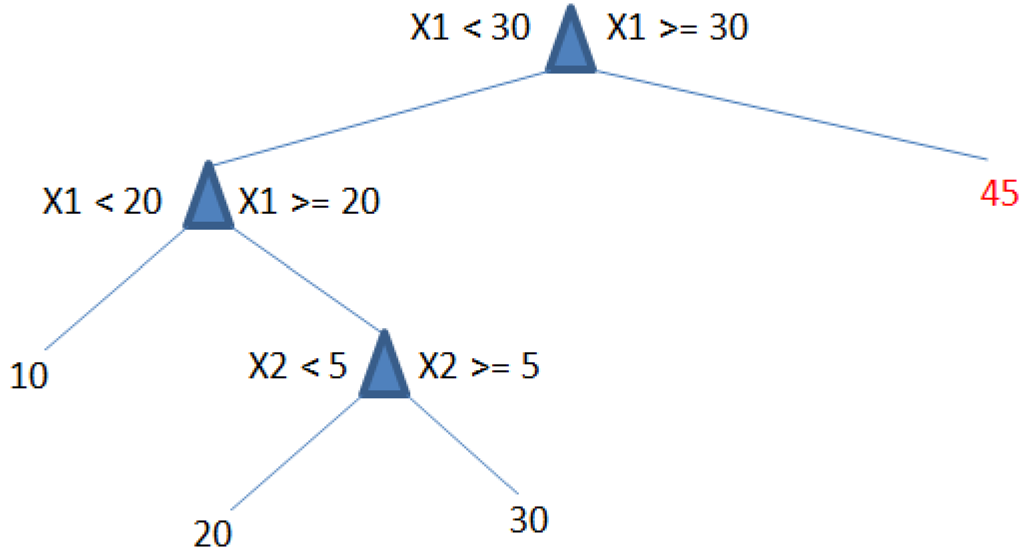
Calculate *Merge-Error* with 45

Calculate *Orig.-Error* with 40 and 50

$$\sum_{x_i \in R(2,7)} (y_i - 45)^2 < \sum_{x_i \in R_1(2,7)} (y_i - 40)^2 + \sum_{x_i \in R_2(2,7)} (y_i - 50)^2$$

How to implement the postpruning?

- Make the regression tree to a large depth
- Start at the bottom and combine every two leaf nodes(left and right) into a new terminal node, if such error after merge is smaller than the original one

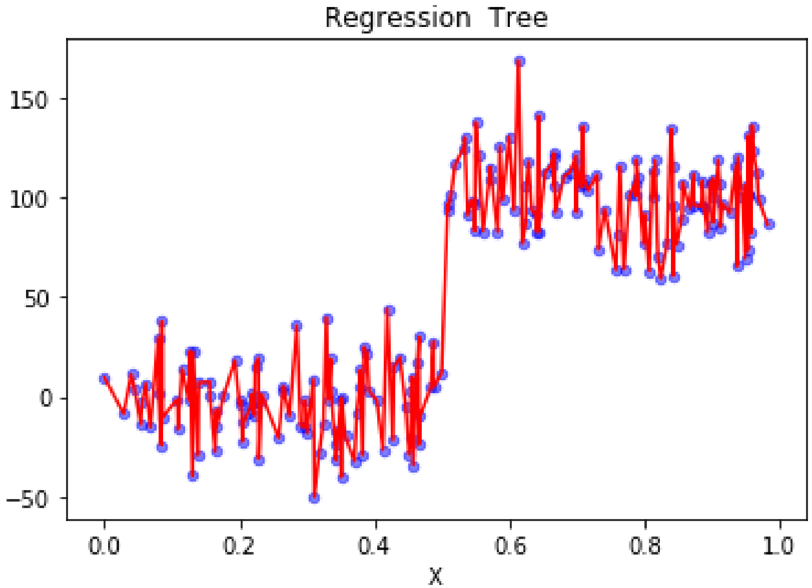


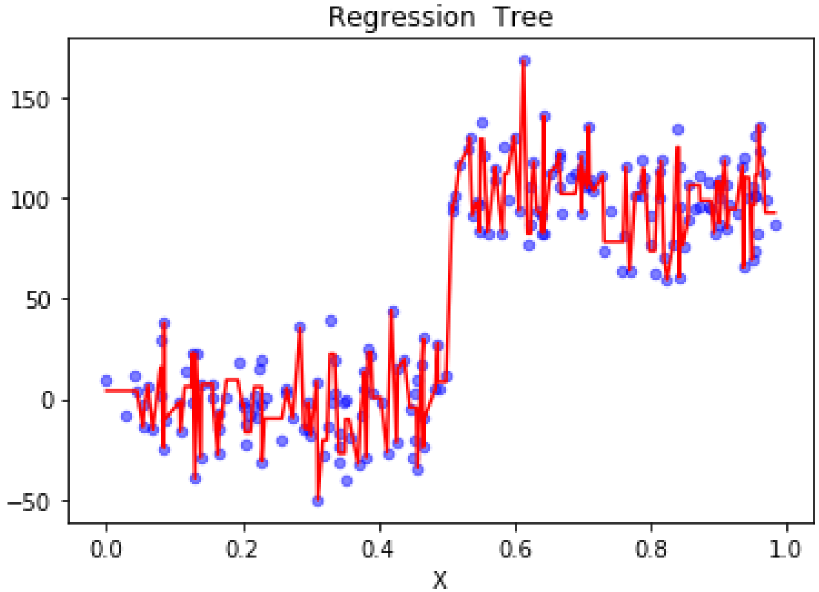
If *Merge-Error* < *Orig.-Error*

then **merge**

LIKE

Prepruning vs. Postpruning





Prepruning

Sum of leaf nodes: 200

tree depth: 25

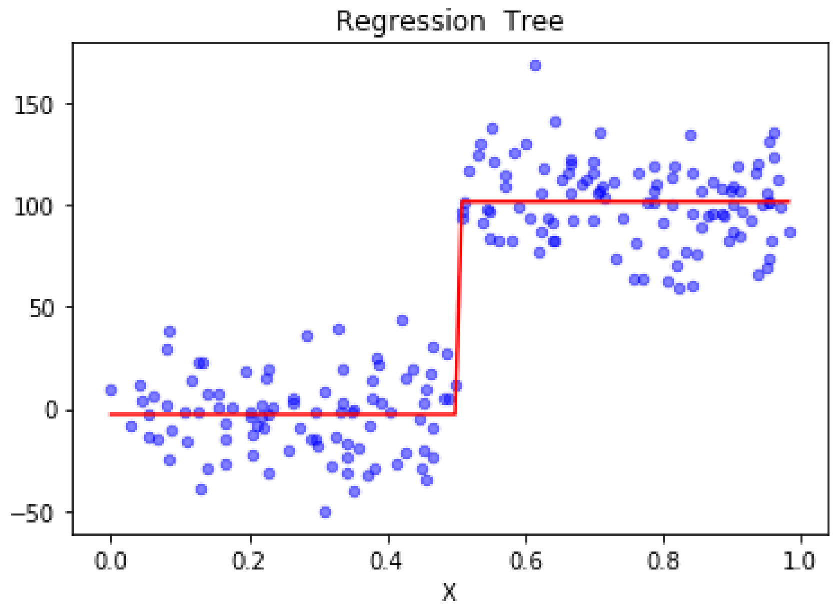
Postpruning

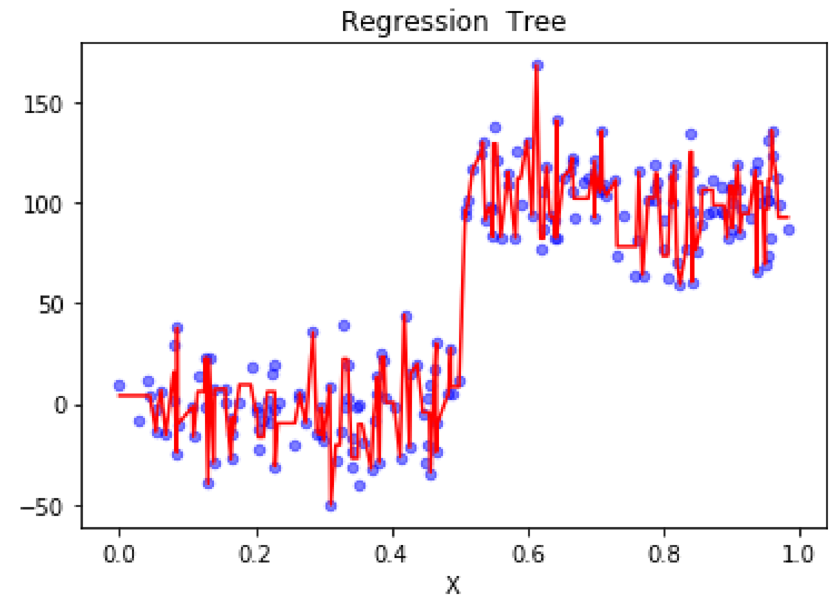
Sum of leaf nodes: 141

tree depth: 23

LIKE

Prepruning vs. Postpruning

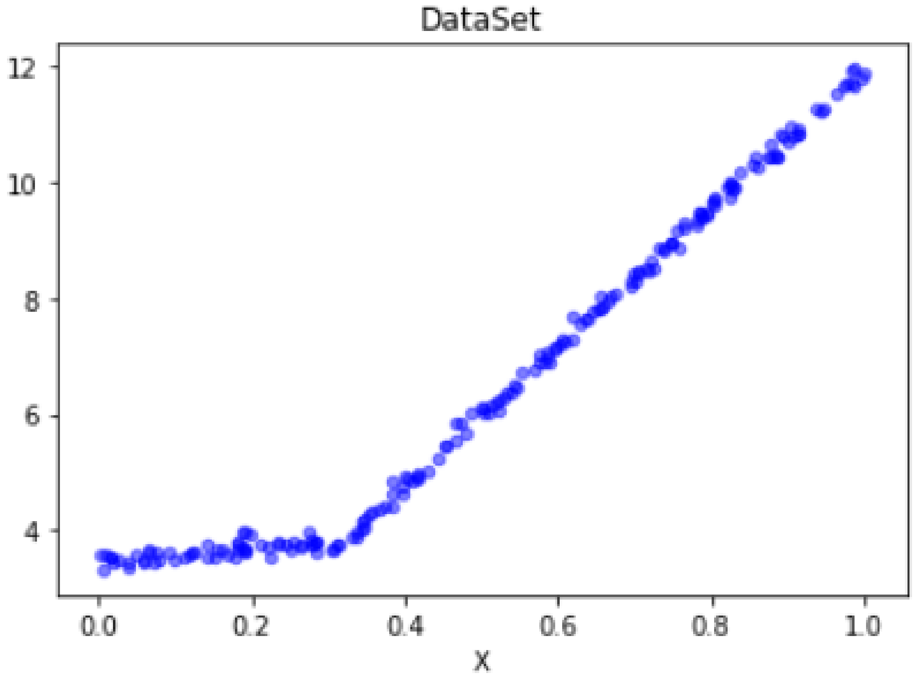




Some nodes were pruned off the tree, but it wasn't reduced to two nodes as we had hoped. It turns out that <u>postpruning</u> isn't as effective as <u>prepruning</u>.

We can employ ***<u>both</u>*** to give the best possible model.

IV: Tree-based Regression

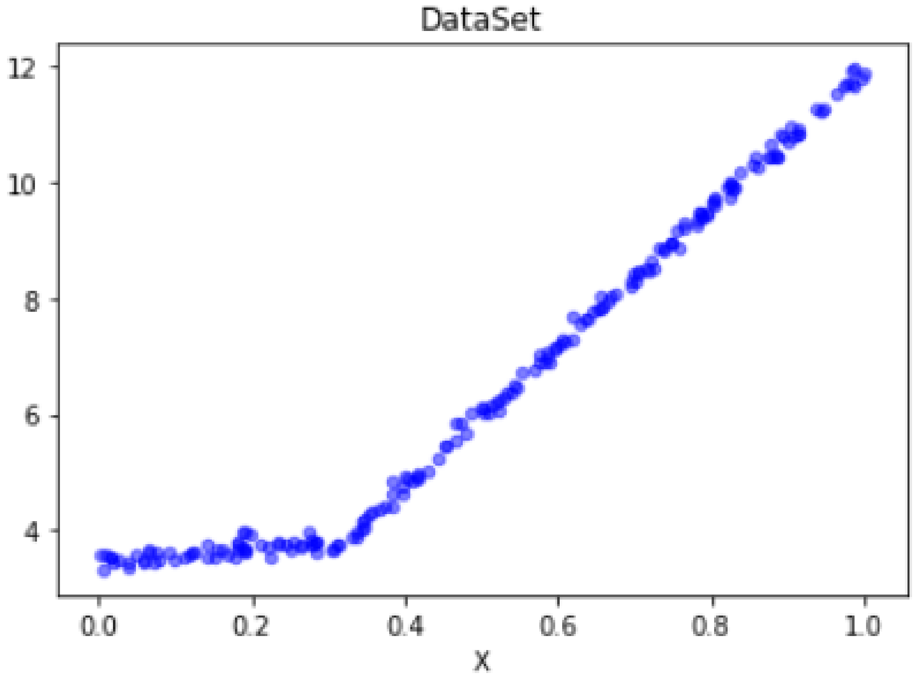


Would it be better to model this dataset as

- a bunch of constant values (many leaf nodes) or
- two straight lines (1: from _0.0_ to _0.3_; 2: from _0.3_ to _1.0_)

LIKE

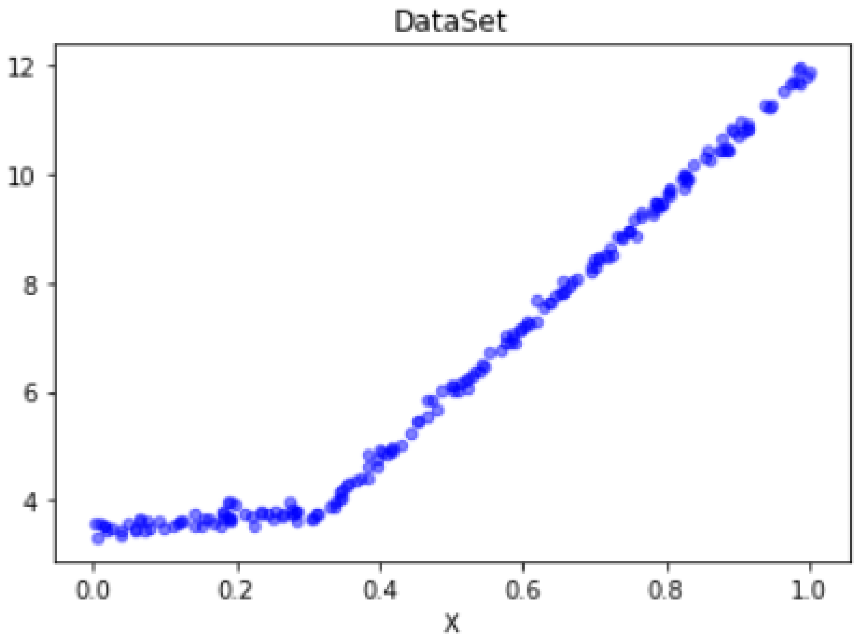
IV: Tree-based Regression



Model Tree = Tree + Linear regression

A way to model dataset as a piecewise linear model at each leaf node. Piecewise linear means that the model consists of multiple linear segments

IV: Tree-based Regression



Result:

{'spInd': 0, 'spVal': 0.285477,

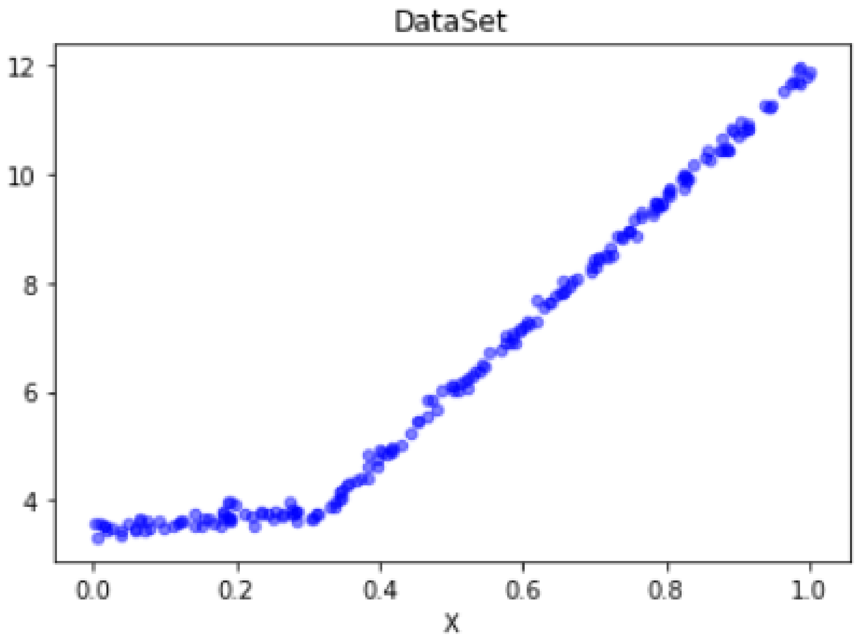
'left': matrix([[3.46877936], [1.18521743]]),

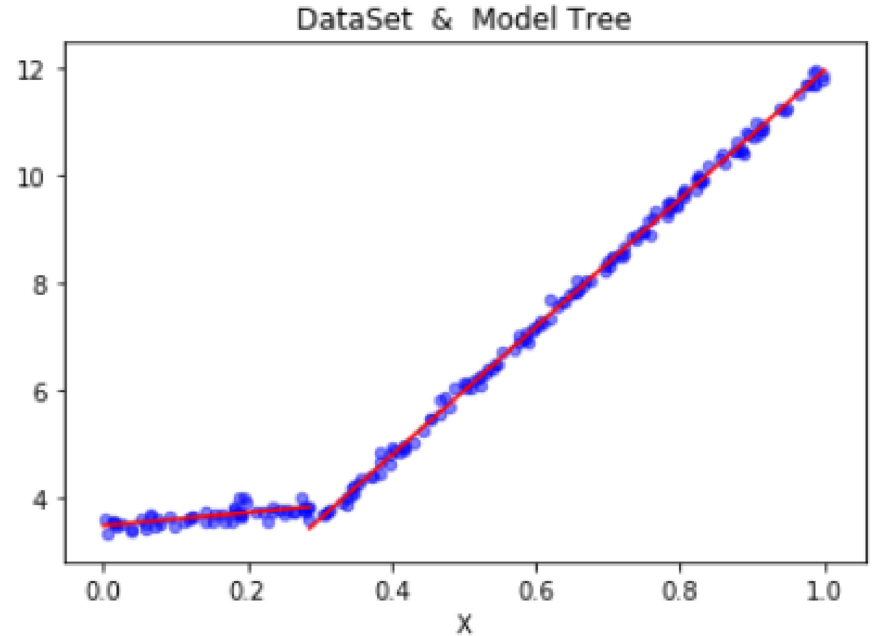
'right': matrix([[1.69855694e-03], [1.19647739e+01]])}

$$y = 0 + 11.96 * x \quad and \quad y = 3.47 + 1.20 * x$$

LIKE

IV: Tree-based Regression





Result:

{'spInd': 0, 'spVal': 0.285477,

'left': matrix([[3.46877936], [1.18521743]]),

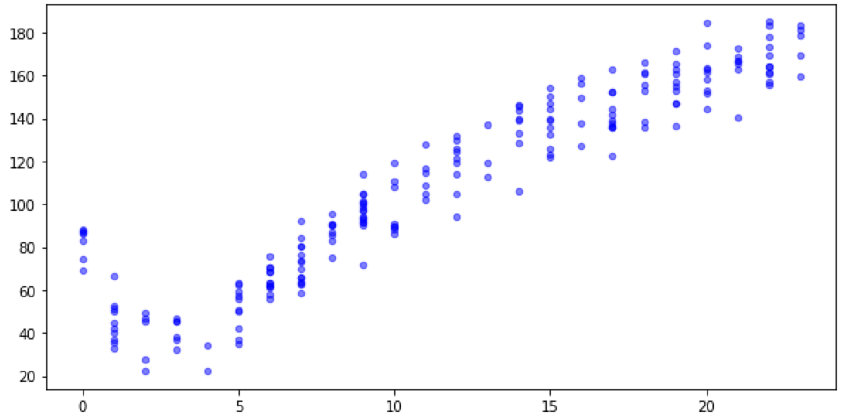
'right': matrix([[1.69855694e-03], [1.19647739e+01]])}

$$y = 0 + 11.96 * x \quad and \quad y = 3.47 + 1.20 * x$$

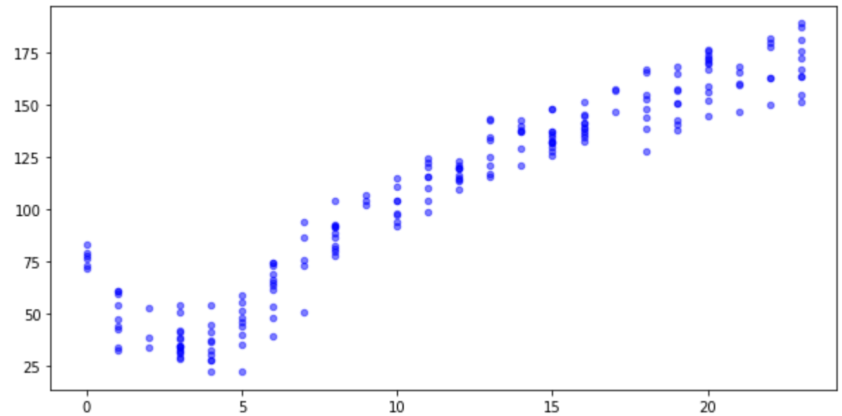
LIKE

V: Linear Regression vs. Tree Regression

Training-data



Testing-data



Evaluation method:

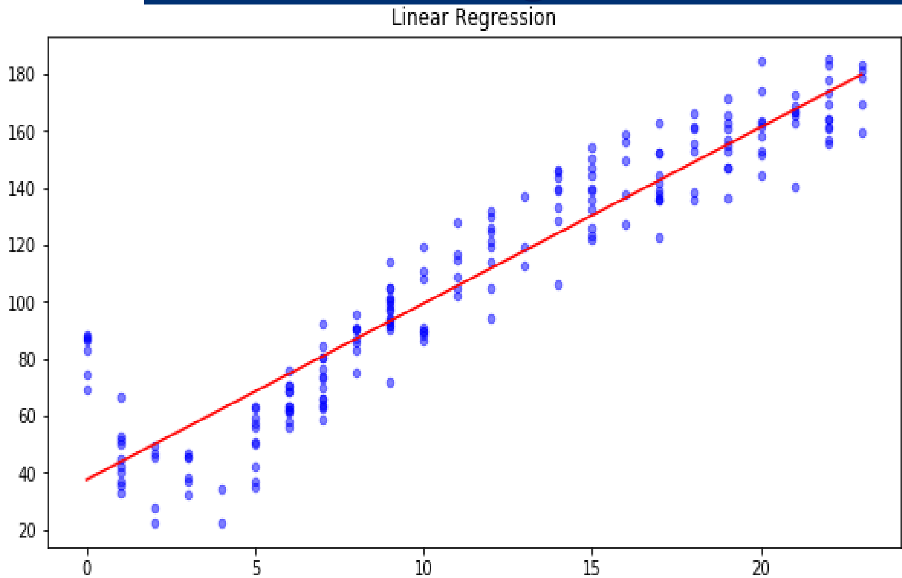
- Correlation coefficients

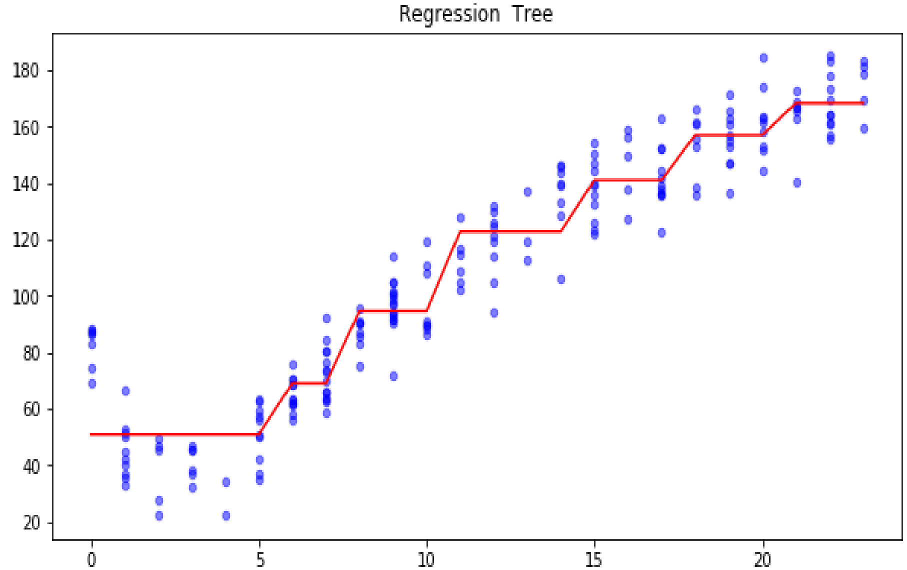
Regression methods:

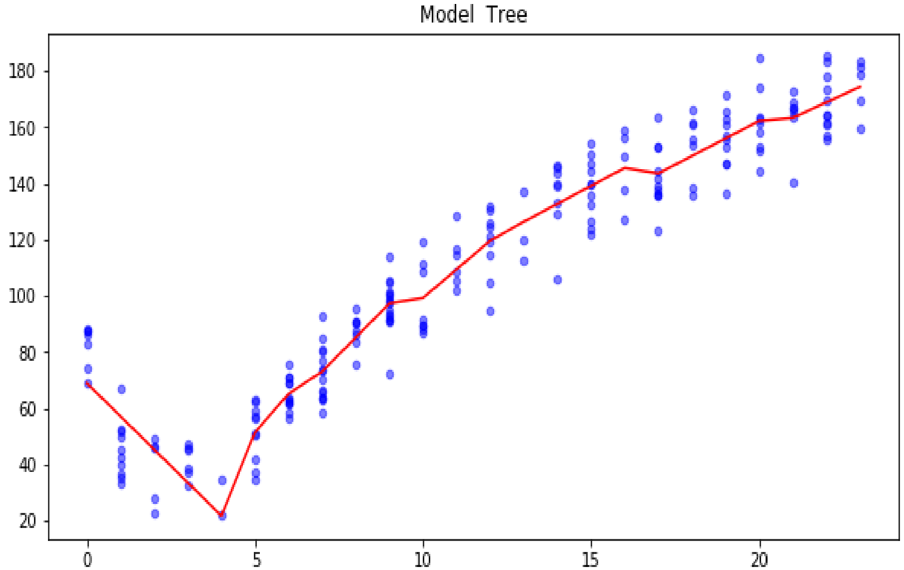
- Linear Regression(LR)
- Regression Tree (RT)
- Model Tree (MT)

LIKE

V: Linear Regression vs. Tree Regression







Correlation coefficients:

- LR: 0.94346842356
- RT: 0.96408523182
- MT: 0.97604121913

LIKE

VI: Summary

Regression is the process of predicting a target value, which makes it become one of the most useful tools in statistics.

Linear methods:

- Linear regression: does a good job, when the dataset is simple and linear.
- Locally weighted linear regression: the forecast would be more precise. Overfitting problem should be avoided.

Non-linear methods:

- Regression tree: breaks up the predicted value into piecewise constant segments.
- Model Tree: implements the linear regression equations at each leaf node.
- Pre- and Postpruning: can effectively reduce the complexity of tree and help avoid the overfitting problem.

Reference

- Harrington, Peter. 2012. *Machine Learning in Action*. Shelter Island (N.Y.): Manning Publications Co.
- Li Hang. 2012. Statistic Learning Methods. Tsinghua University Publications Co.
- Analytics Vidhya: A Complete Tutorial on Tree Based Modeling from Scratch. APRIL 12, 2016 https://www.analyticsvidhya.com/blog/2016/04/complete-tutorial-tree-based-modeling-scratch-in-python