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

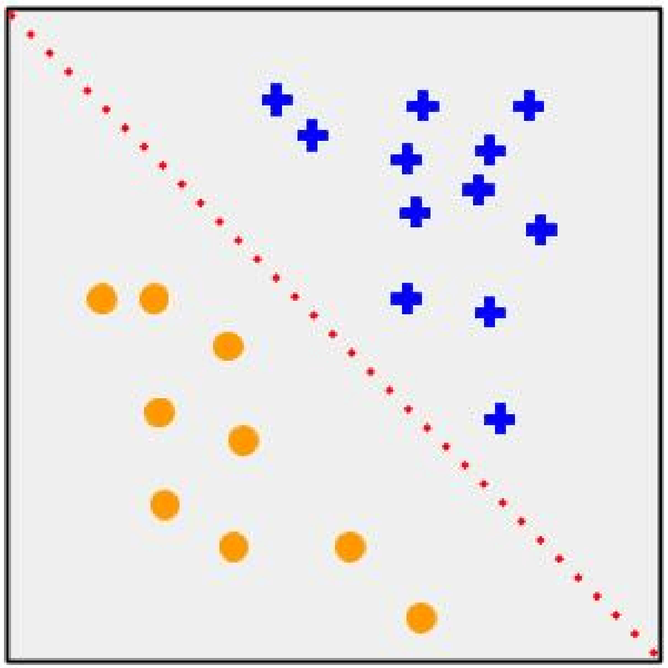
Contents

LIKE

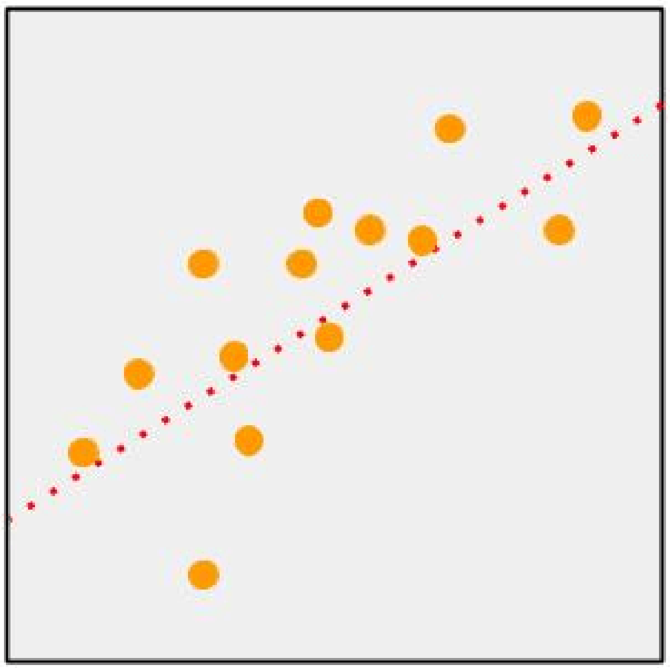
I. Regression vs. Classification

"*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

Task: House Price

Equation:

$$HousePrice = \omega 1 * HouseSize + \omega 2 * AgeofHouse$$

Input: HouseSize, AgeofHouse

Weights: ω1, ω2

II. Linear Regression

Let's assume that our input and output data are matrix X and Y:

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

and we put regression weights into a vector: $\omega = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix}$

then we could predict the value y1 for the given dataset x1

$$y_1 = \begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \end{bmatrix}^T * \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix}$$

LIKE

II. Linear Regression

The next step is to find the weights with the given training datasets by minimizing the squared error between actual and predicted value.

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

The corresponding expression in matrix is $(Y - X\omega)^T(Y - X\omega)$

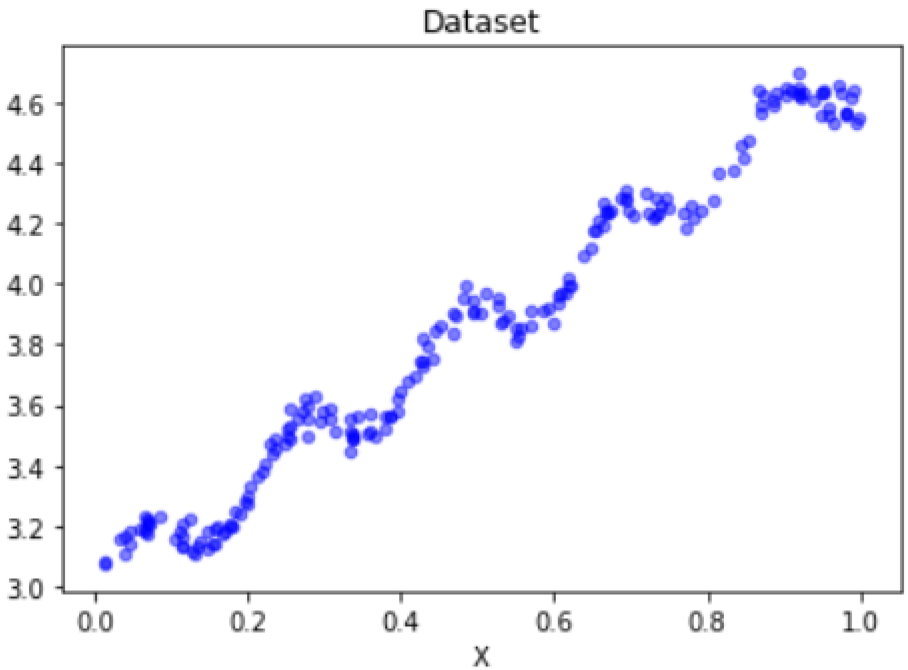
After the derivation, we get $X^T(Y - X\omega)$

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

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

LIKE

II. Linear Regression

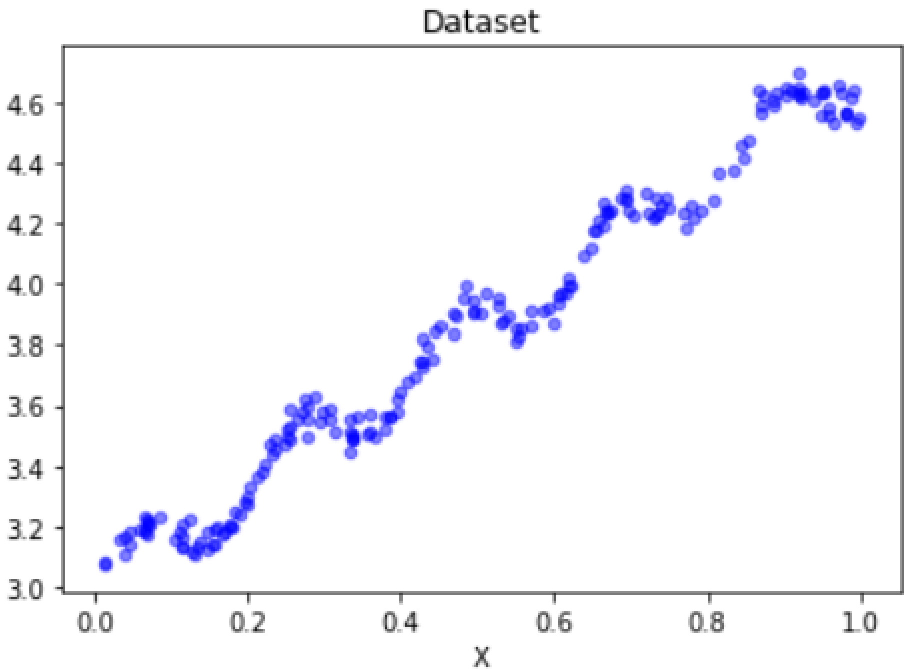


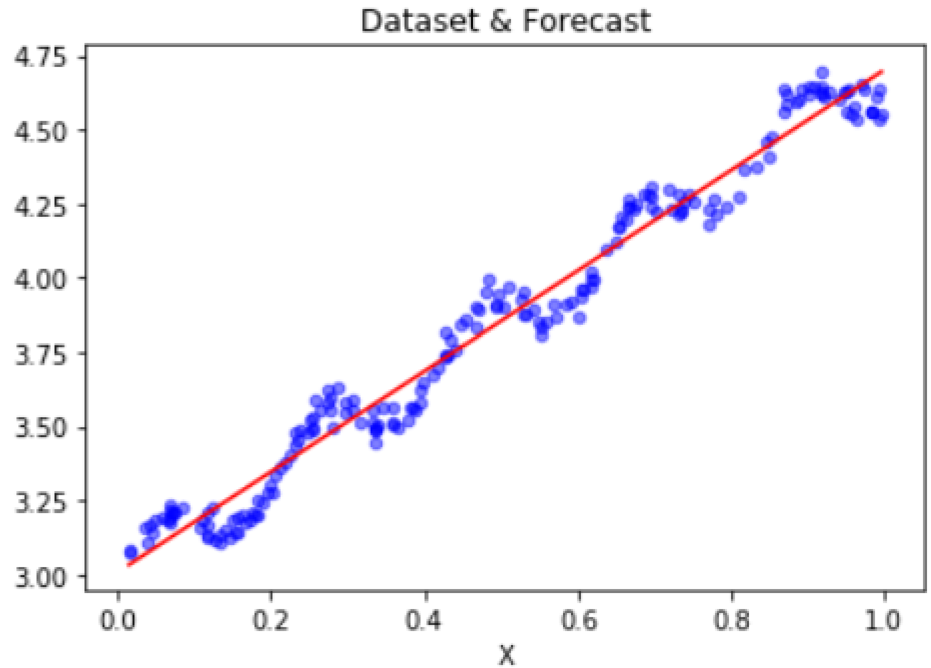
$$\hat{\omega} = (X^T X)^{-1} X^T Y$$

weights:

[[3.00774324] [1.69532264]]

II. Linear Regression





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

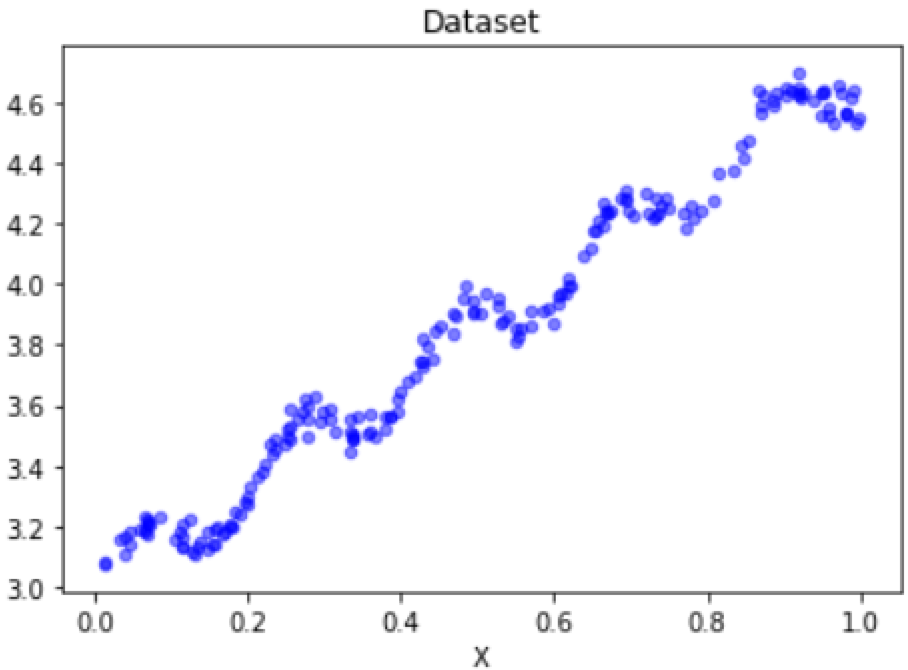
Y_pred = X * weights

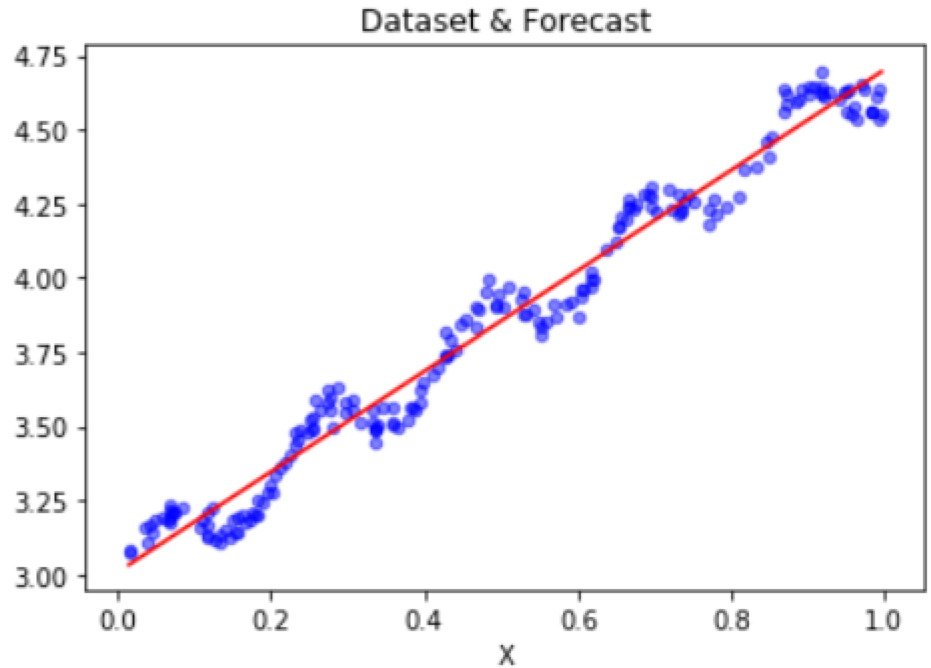
weights:

[[3.00774324] [1.69532264]]

LIKE

II. Linear Regression





But how far do they differ from each other?

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

[0.98647356, 1.]])

underfitting problem?

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.

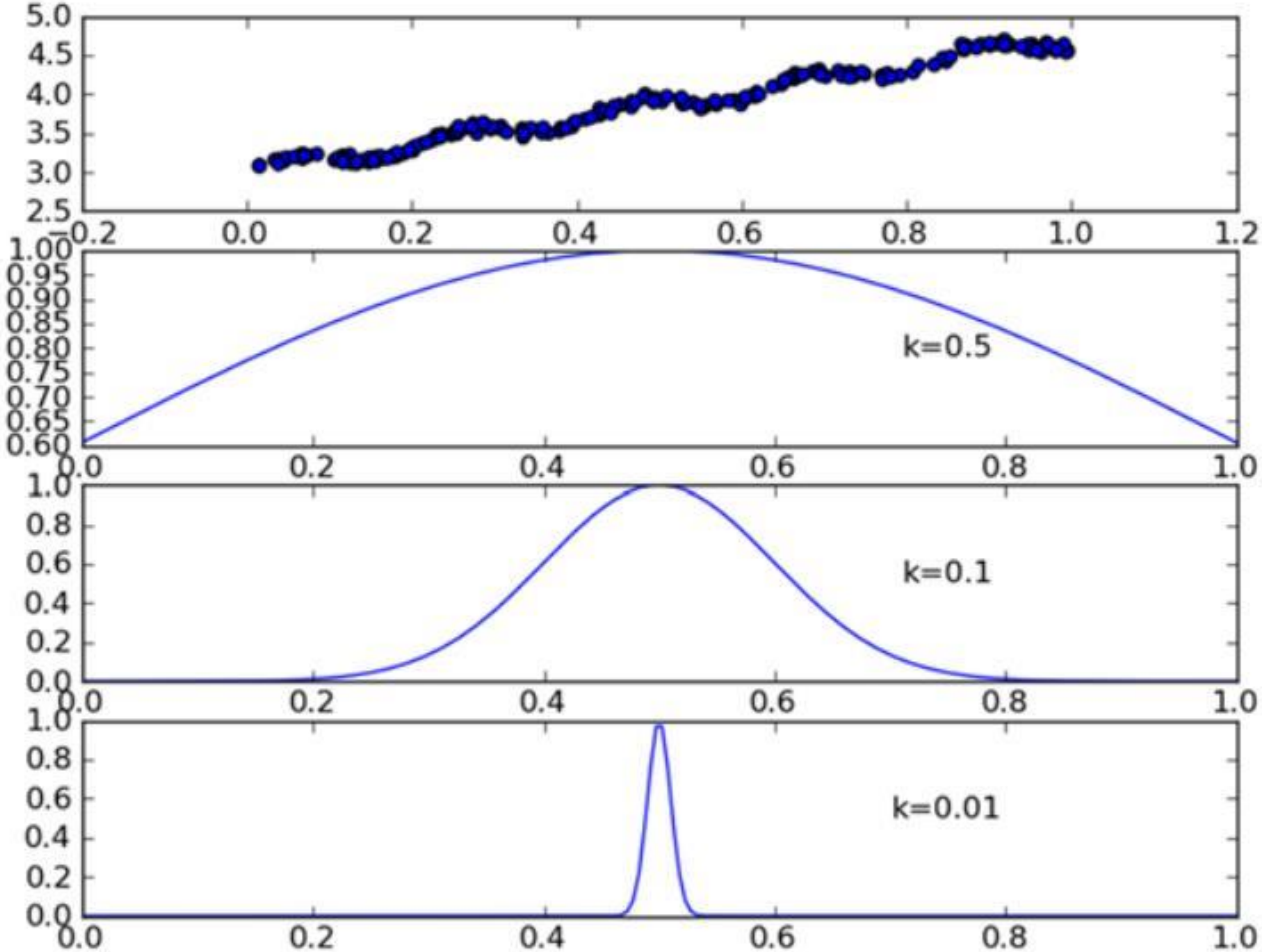
$$\hat{\omega} = (X^T W X)^{-1} X^T W Y$$

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

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

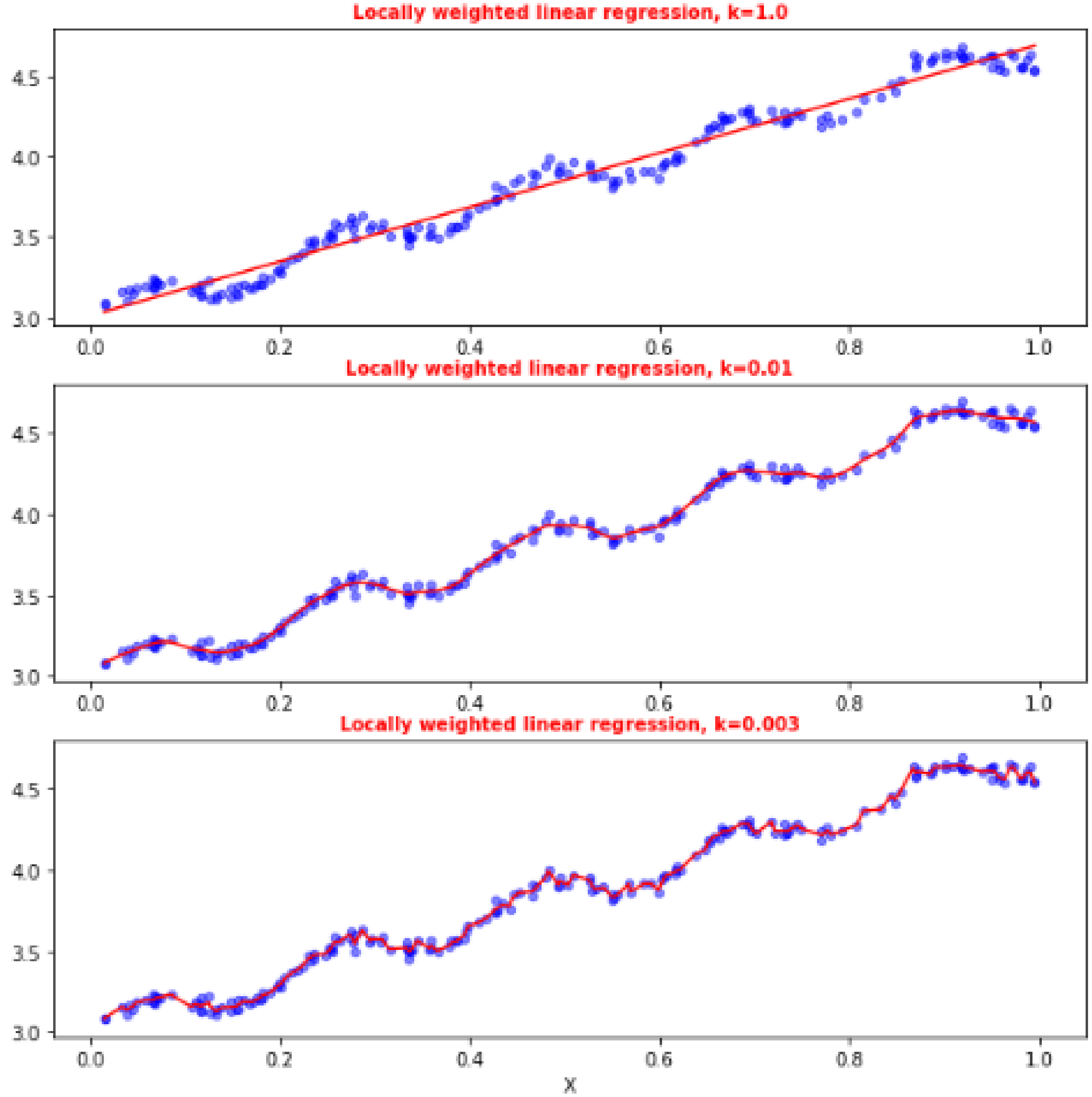
LIKE

III. Locally Weighted Linear Regression



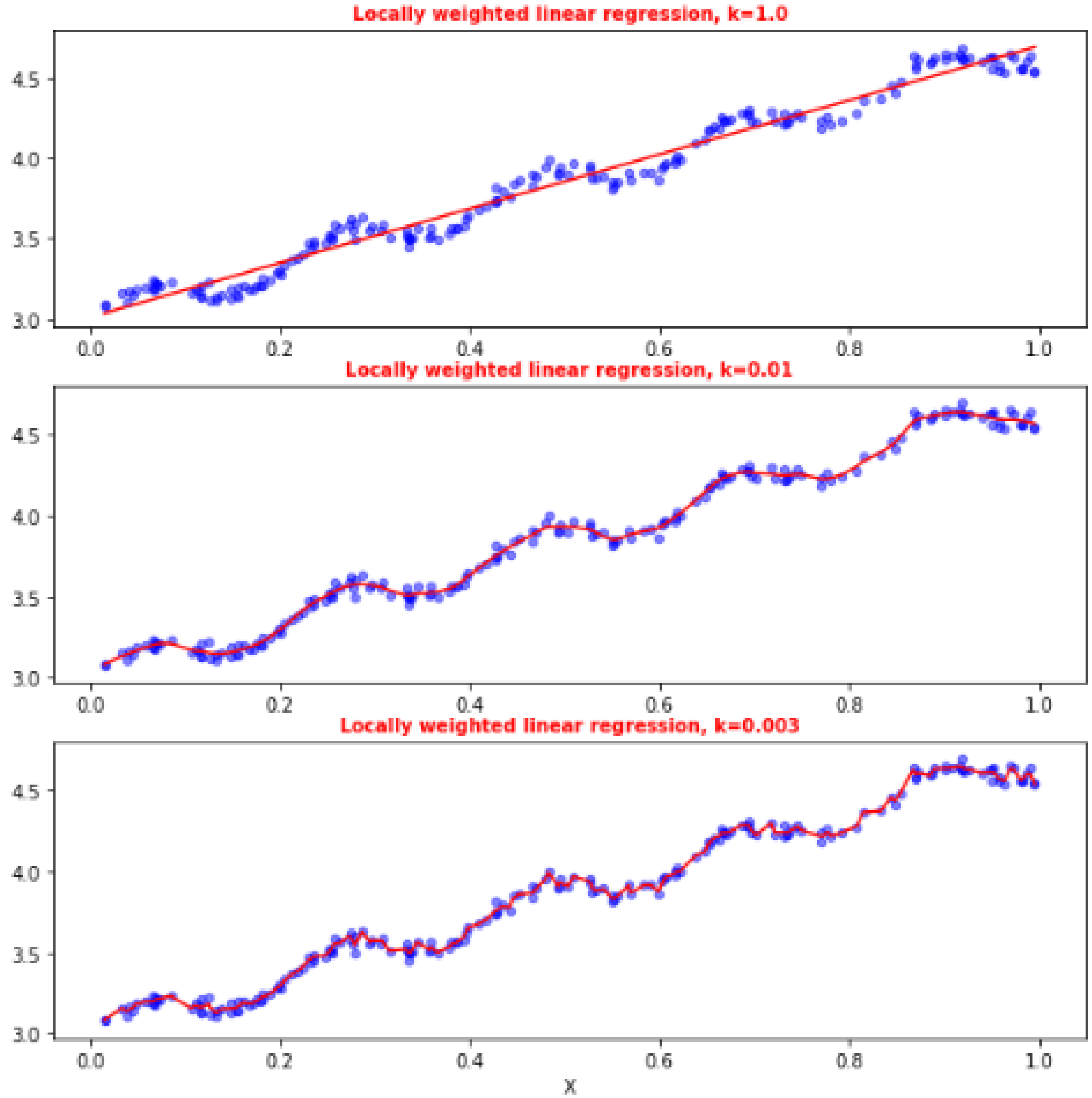
LIKE

III. Locally Weighted Linear Regression



LIKE

III. Locally Weighted Linear Regression



overfitting!

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.]]

Training set and test set are *identical*:

k=0.1, the Error: 56.82523568972884
k=1.0, the Error: 429.8905618700651
k=10, the Error: 549.1181708826451

Example: predicting the age of an abalone

Training set and test set are *identical*:

k=0.1, the Error: 56.82523568972884
k=1.0, the Error: 429.8905618700651
k=10, the Error: 549.1181708826451

Training set and test set are *different*:

k=0.1, the Error: 41317.161723642595
k=1.0, the Error: 573.526144189767
k=10, the Error: 517.5711905387598

LIKE

Example: predicting the age of an abalone

Training set and test set are different,
Linear regression vs. Locally weighted linear regression

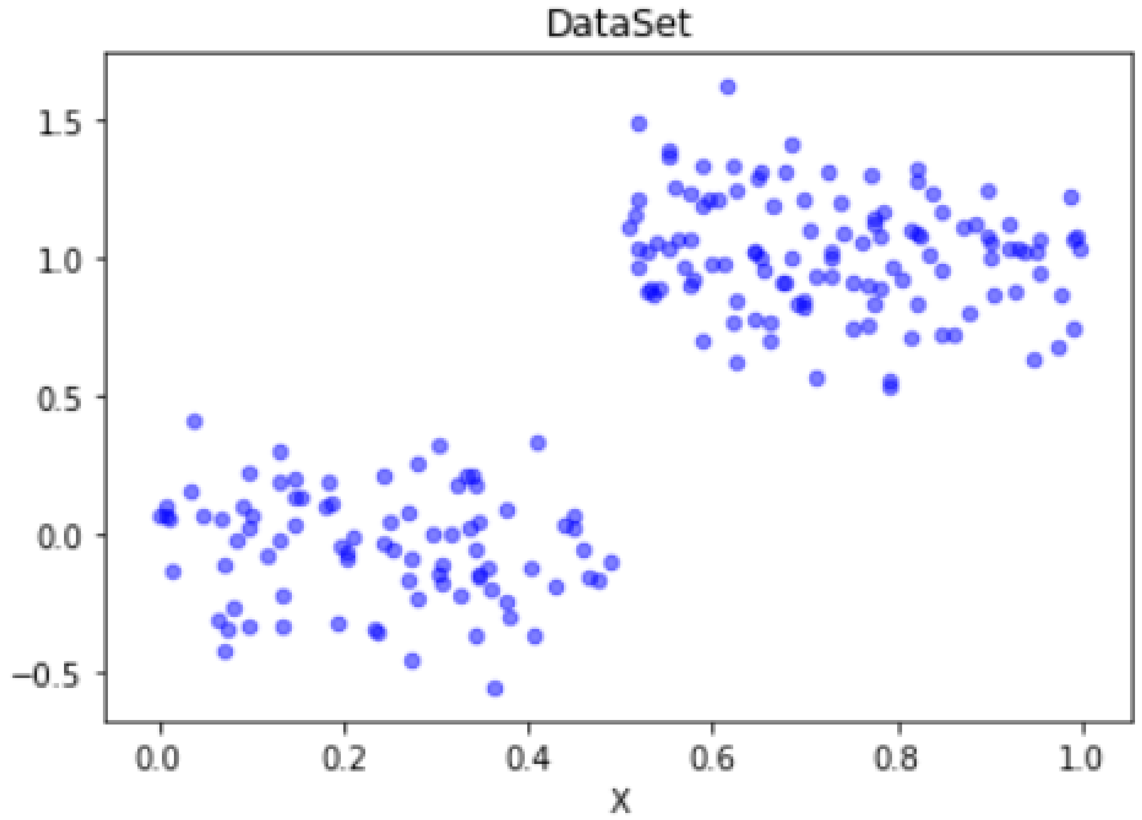
lwlr with k=1: the Error: 573.526144189767
Linear regression: the Error: 518.6363153249638

Simple *linear regression* works almost as well as the *locally weighted linear regression*. This demonstration illustrates one fact: in order to find the best model, we have to see how the model works on **unknown** data.

LIKE

IV: Tree-based Regression

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



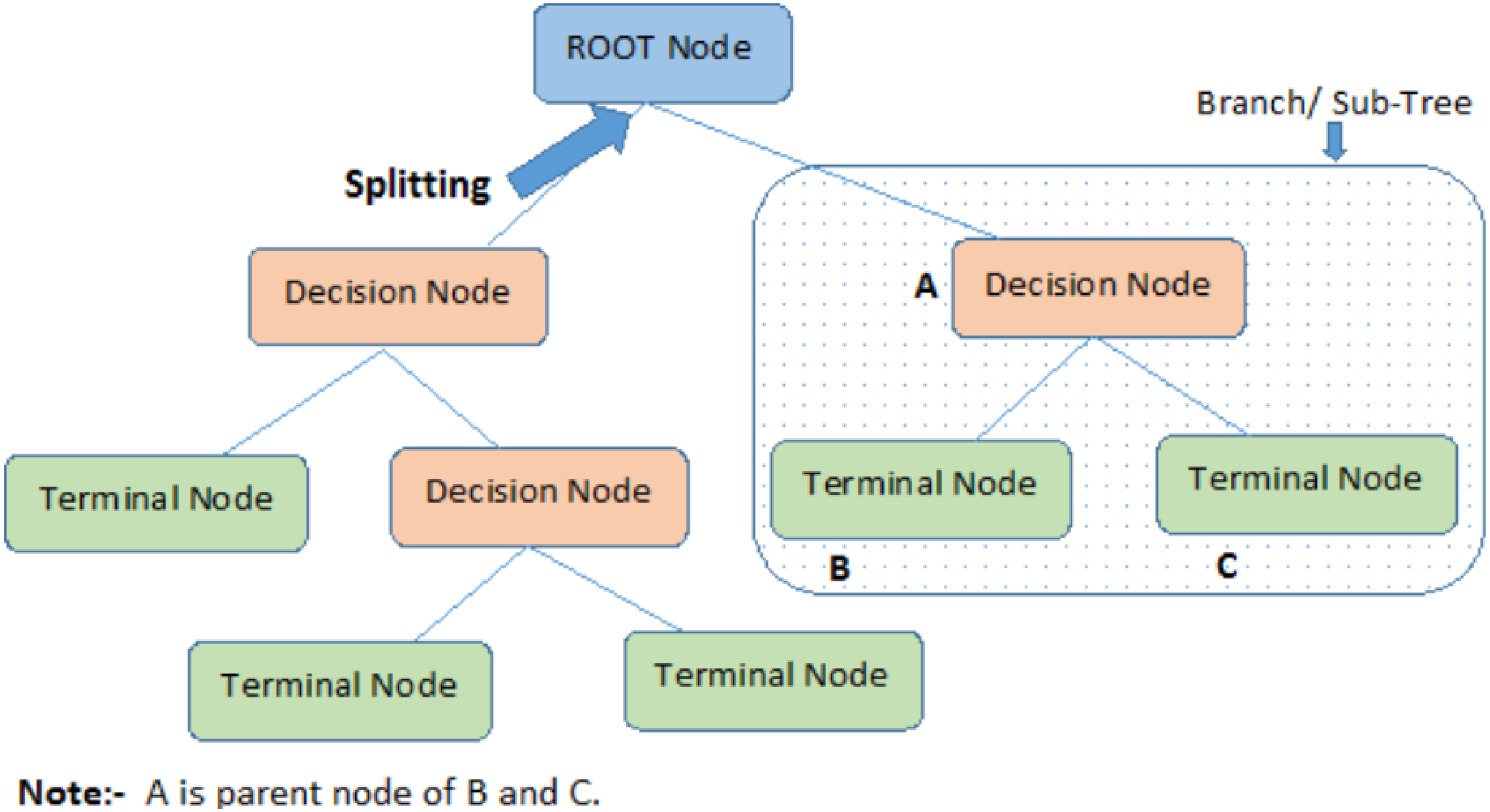
LIKE

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

Tree Regression

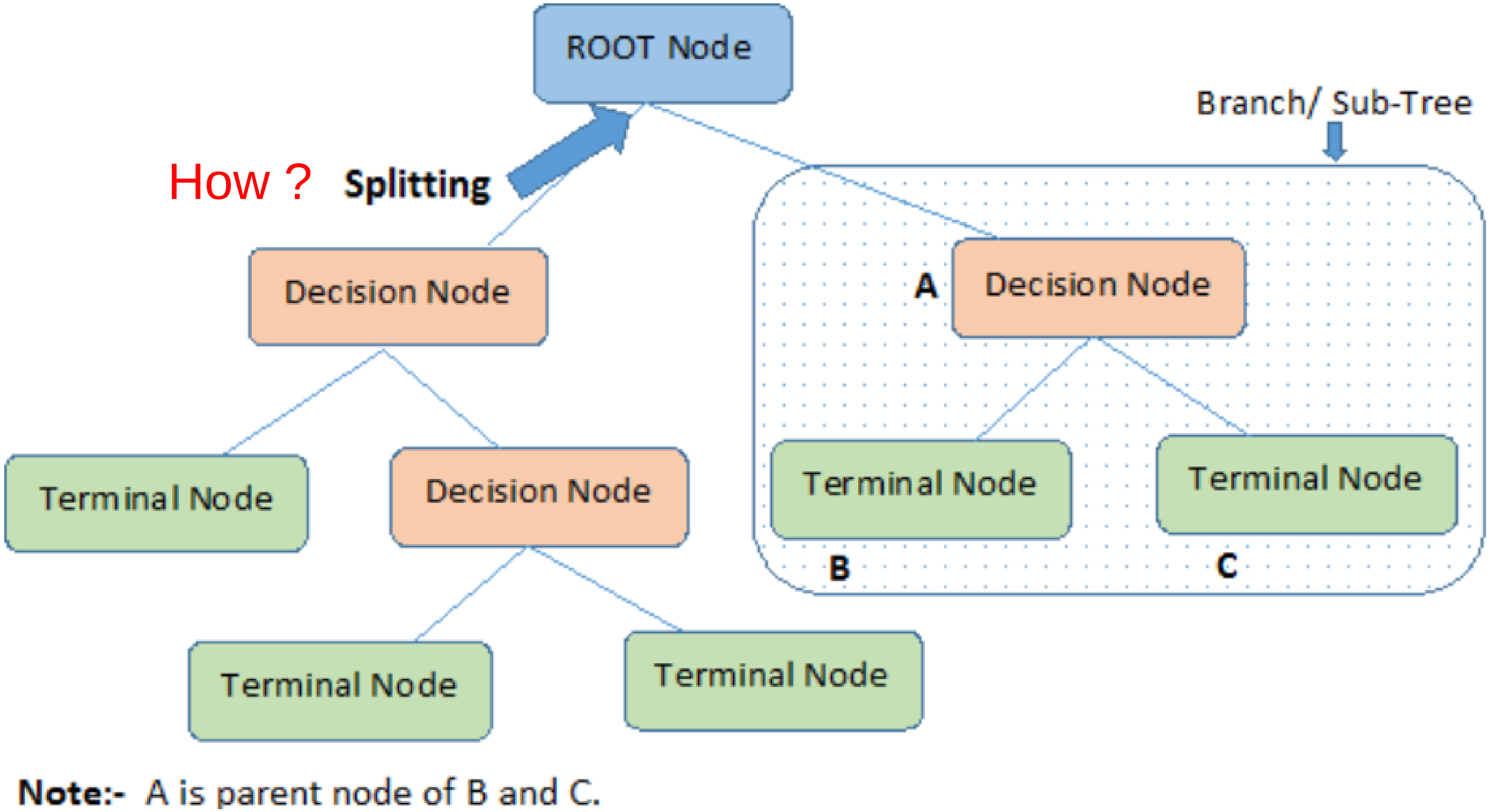
IV: Tree-based Regression

Tree structure



LIKE

Tree structure



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

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

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

Pseudo-code of choosing a best split

How to make a binary split?

For every unique value:
    Split the dataset into two
        Measure the error of these two splits
        If the error is less than bestError, then bestSplit to this split and update bestError
Return bestSplit feature and threshold

LIKE

Pseudo-code of choosing a best split

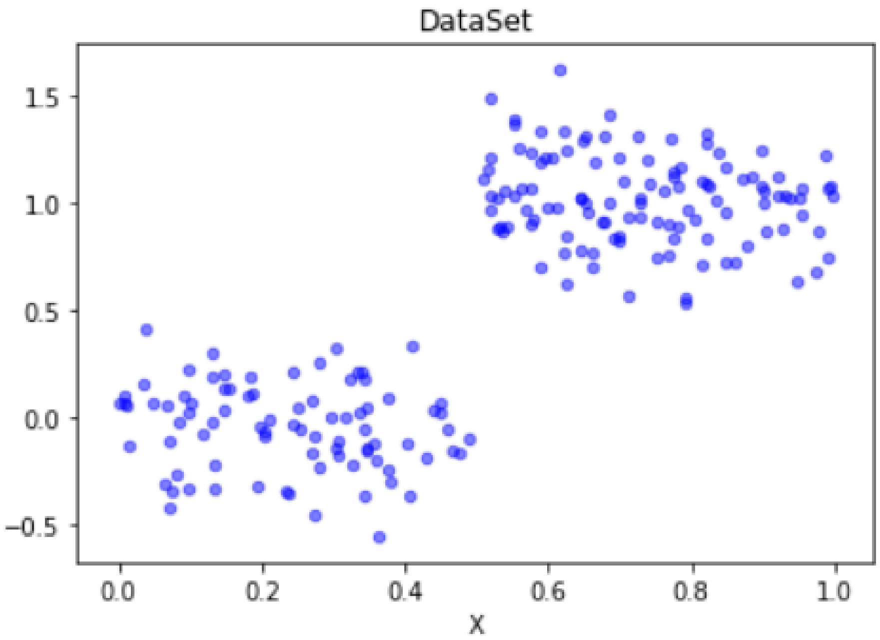
How to make a binary split?

For every unique value:
Split the dataset into two
Measure the error of these two splits
If the error is less than bestError, then bestSplit to this split and update bestError
Return bestSplit feature and threshold

Stop Conditions: min. error-reduction & min. data instances
If (actual error – best error < error reduction) or
(left_matrix.numInstance < min.Inst) or
(right_matrix.numInstance < min.Inst) → split stops!

LIKE

IV: Tree-based Regression



User-defined parameter:

Error reduction: 1

Min.data instances: 4

Result:

Splitting feature: 0 (here only the feature from x-data)

Splitting point: 0.48813 (x-value)

Pseudo-code of creating a regression tree

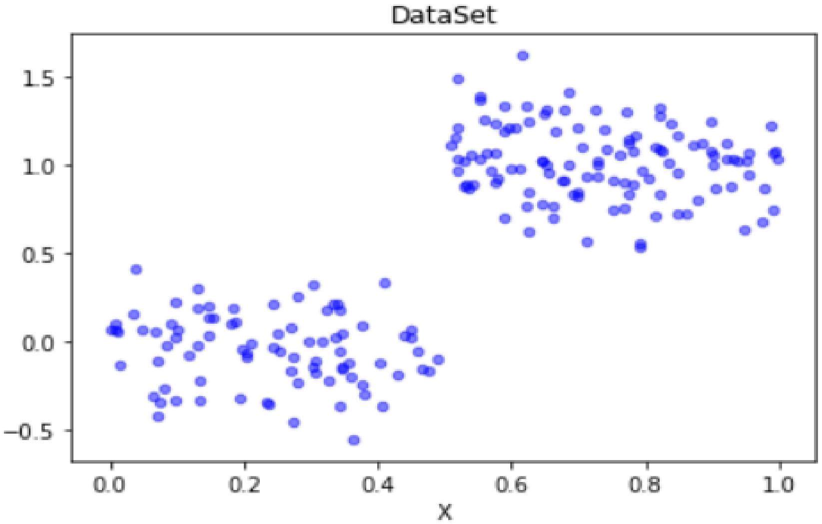
How to make a regression tree?

Find the best feature to split on:

If we can’t split the data, this node becomes a leaf node
Make a binary split of the data
Call createTree() on the right split of the data
Call createTree() on the left split of the data

LIKE

IV: Tree-based Regression



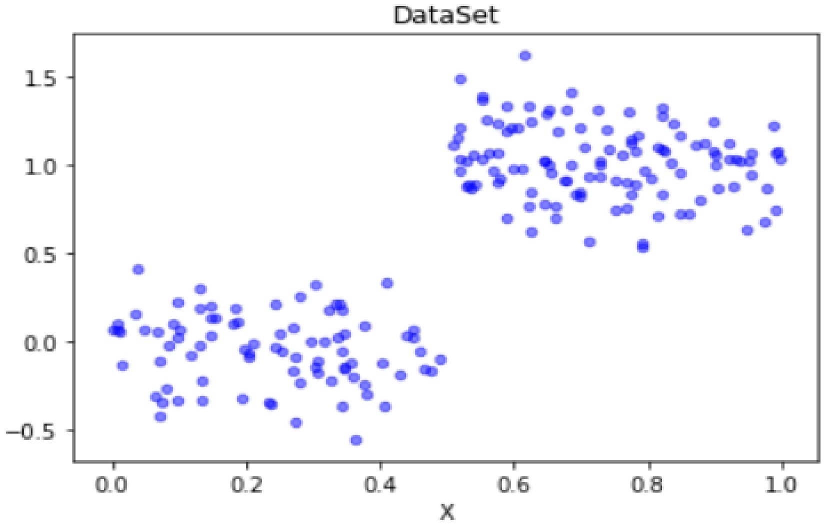
User-defined parameter:

Mind. error eduction: 1

Mind. data instances: 4

LIKE

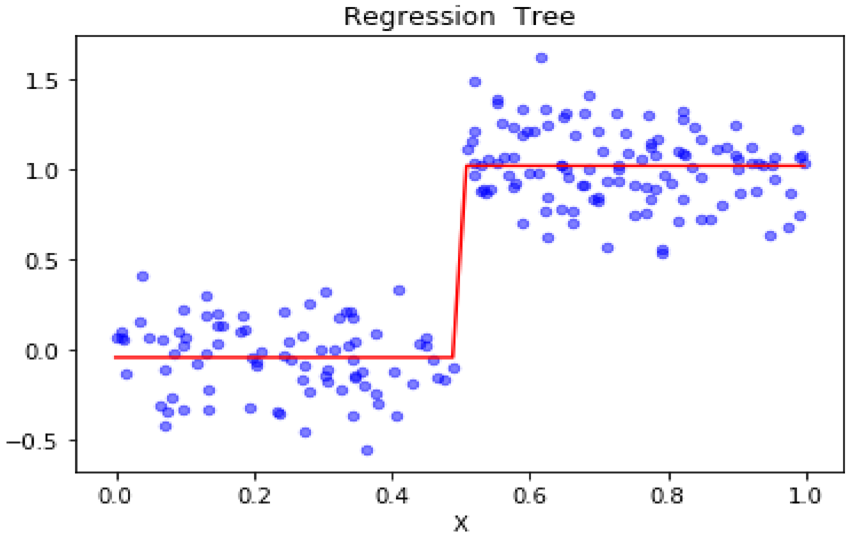
IV: Tree-based Regression



User-defined parameter:

Mind. error eduction: 1

Mind. data instances: 4



print(createTree(data_mat))

{'spInd': 0,

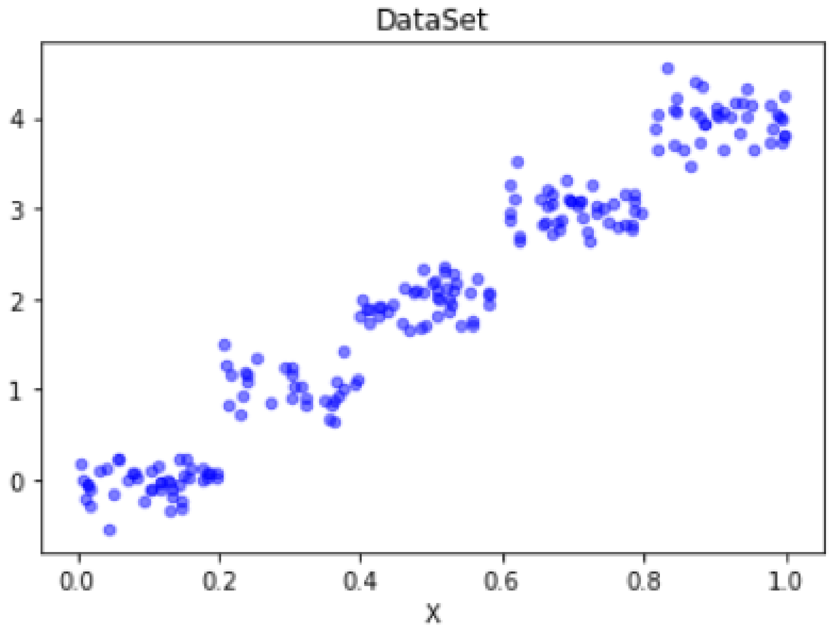
'spVal': 0.48813,

'left': -0.044650285714285 72,

'right': 1.0180967672413792 }

LIKE

IV: Tree-based Regression



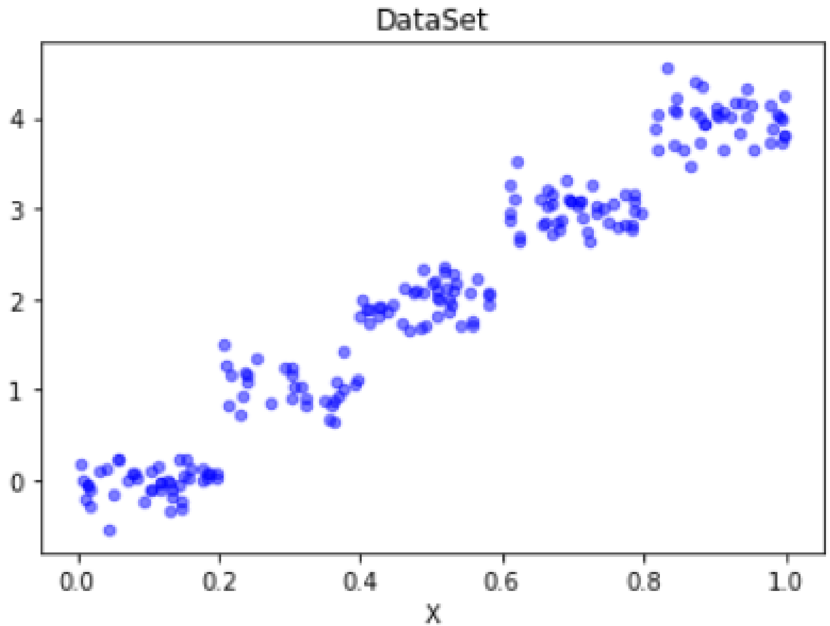
User-defined parameter:

Mind. error eduction: 1

Mind. data instances: 4

LIKE

IV: Tree-based Regression

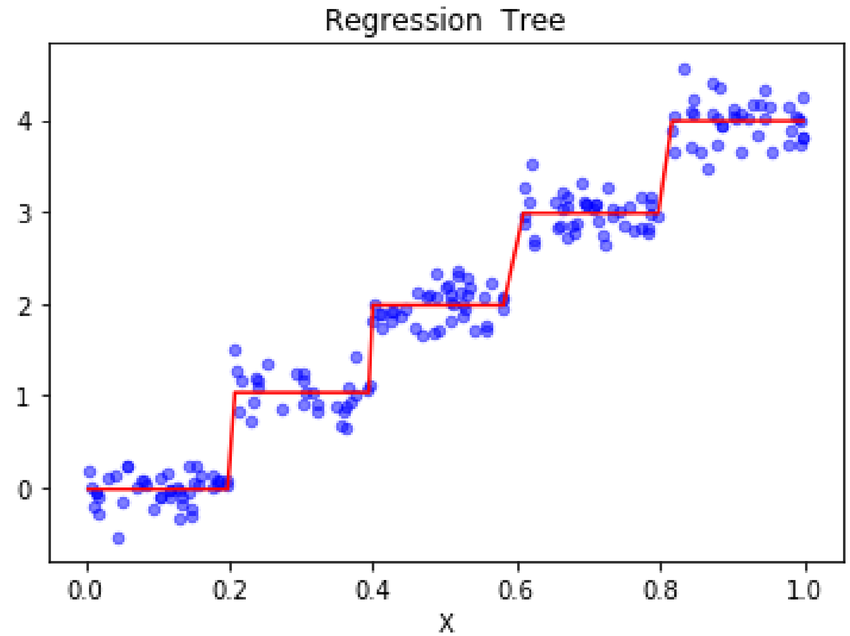


User-defined parameter:

Mind. error eduction: 1

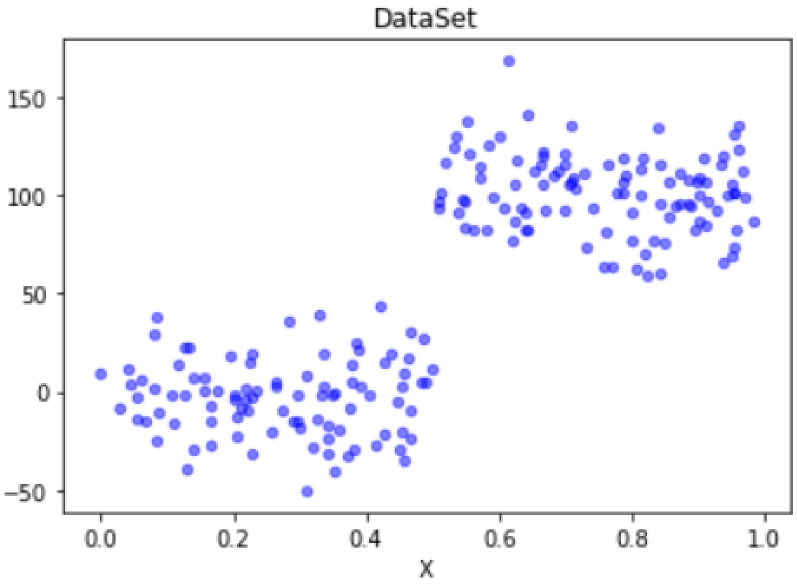
Mind. data instances: 4

{'spInd': 1, 'spVal': 0.39435, 'left': {'spInd': 1, 'spVal': 0.197834, 'left': -0.023838155555555553, 'right': 1.0289583666666666}, 'right': {'spInd': 1, 'spVal': 0.582002, 'left': 1.980035071428571, 'right': {'spInd': 1, 'spVal': 0.797583, 'left': 2.9836209534883724, 'right': 3.9871632}}}



LIKE

IV: Tree-based Regression

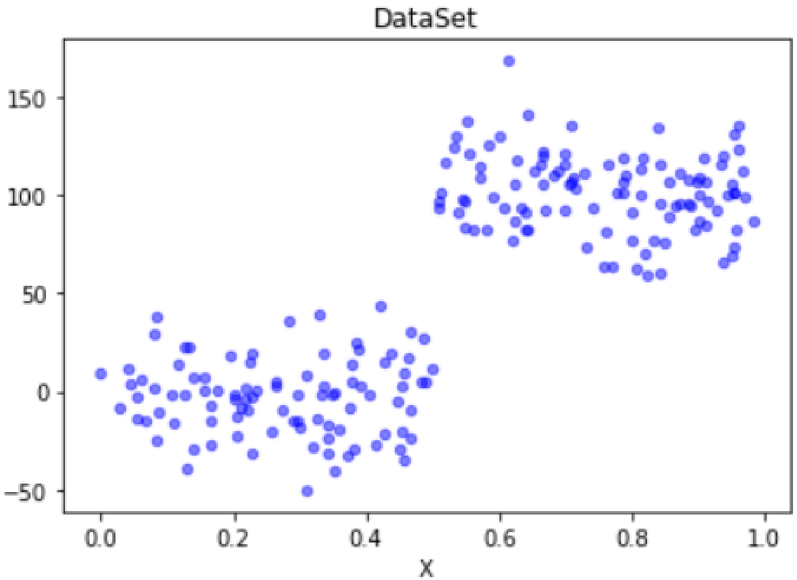


User-defined parameter:

Mind. error eduction: 1

Mind. data instances: 4

IV: Tree-based Regression

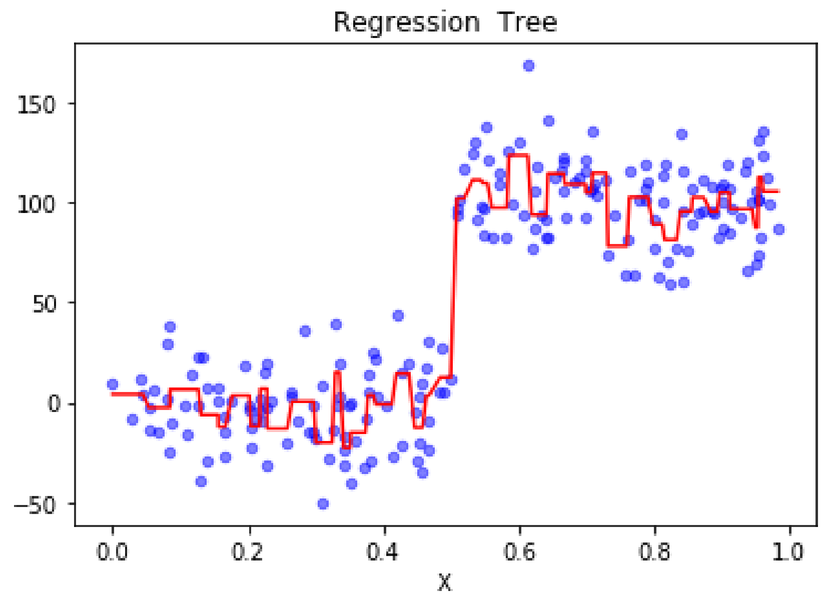


{'spInd': 0, 'spVal': 0.499171, 'left': {'spInd': 0, 'spVal': 0.457563, 'left': {'spInd': 0, 'spVal': 0.126833, 'left': {'spInd': 0, 'spVal': 0.084661, 'left': {'spInd': 0, 'spVal': 0.044737, 'left': 4.091626, 'right': -2.54439271428571 5}, 'right': 6.509843285714284}, 'right': {'spInd': 0, 'spVal': 0.373501, 'left': {'spInd': 0, 'spVal': 0.335182, 'left': {'spInd': 0, 'spVal': 0.324274, 'left': {'spInd': 0, 'spVal': 0.297107, 'left': {'spInd': 0, 'spVal': 0.166765, 'left': {'spInd': 0, 'spVal': 0.156067, 'left': -6.24790000000001, 'right': -12.1079725}, 'right': {'spInd': 0, 'spVal': 0.202161, 'left': 3.4496025, 'right': {'spInd': 0, 'spVal': 0.217214, 'left': -11.82227850000001, 'right': {'spInd': 0, 'spVal': 0.228473, 'left': 6.770429, 'right': {'spInd': 0, 'spVal': 0.25807, 'left': -13.070501, 'right': 0.40377471428571476}}}}}, 'right': -19.9941552}, 'right': 15.05929075}, 'right': {'spInd': 0, 'spVal': 0.350725, 'left': -22.693879600000002, 'right': -15.08511175}}, 'right': {'spInd': 0, 'spVal': 0.437652, 'left': {'spInd': 0, 'spVal': 0.412516, 'left': {'spInd': 0, 'spVal': 0.385021, 'left': 3.6584772500000016, 'right': -0.8923554999999995}, 'right': 14.38417875}, 'right': -12.558604833333334}}}, 'right': {'spInd': 0, 'spVal': 0.467383, 'left': 3.4331330000000007, 'right': 12.50675925}}, 'right': {'spInd': 0, 'spVal': 0.729397, 'left': {'spInd': 0, 'spVal': 0.640515, 'left': {'spInd': 0, 'spVal': 0.613004, 'left': {'spInd': 0, 'spVal': 0.582311, 'left': {'spInd': 0, 'spVal': 0.553797, 'left': {'spInd': 0, 'spVal': 0.51915, 'left': 101.73699325000001, 'right': {'spInd': 0, 'spVal': 0.543843, 'left': 110.979946, 'right': 109.38961049999999}}, 'right': 97.20018024999999}, 'right': 123.2101316}, 'right': 93.67344971428572}, 'right': {'spInd': 0, 'spVal': 0.666452, 'left': 114.1516242857143, 'right': {'spInd': 0, 'spVal': 0.706961, 'left': {'spInd': 0, 'spVal': 0.698472, 'left': 108.92921799999999, 'right': 104.82495374999999}, 'right': 114.554706}}}, 'right': {'spInd': 0, 'spVal': 0.952833, 'left': {'spInd': 0, 'spVal': 0.759504, 'left': 78.08564325, 'right': {'spInd': 0, 'spVal': 0.790312, 'left': 102.35780185714285, 'right': {'spInd': 0, 'spVal': 0.833026, 'left': {'spInd': 0, 'spVal': 0.811602, 'left': 88.78449880000001, 'right': 81.110152}, 'right': {'spInd': 0, 'spVal': 0.944221, 'left': {'spInd': 0, 'spVal': 0.85497, 'left': 95.27584316666666, 'right': {'spInd': 0, 'spVal': 0.910975, 'left': {'spInd': 0, 'spVal': 0.892999, 'left': {'spInd': 0, 'spVal': 0.872883, 'left': 102.25234449999999, 'right': 95.181793}, 'right': 104.825409}, 'right': 96.452867}}, 'right': 87.3103875}}}}, 'right': {'spInd': 0, 'spVal': 0.958512, 'left': 112.42895575000001, 'right': 105.24862350000001}}}}

User-defined parameter:

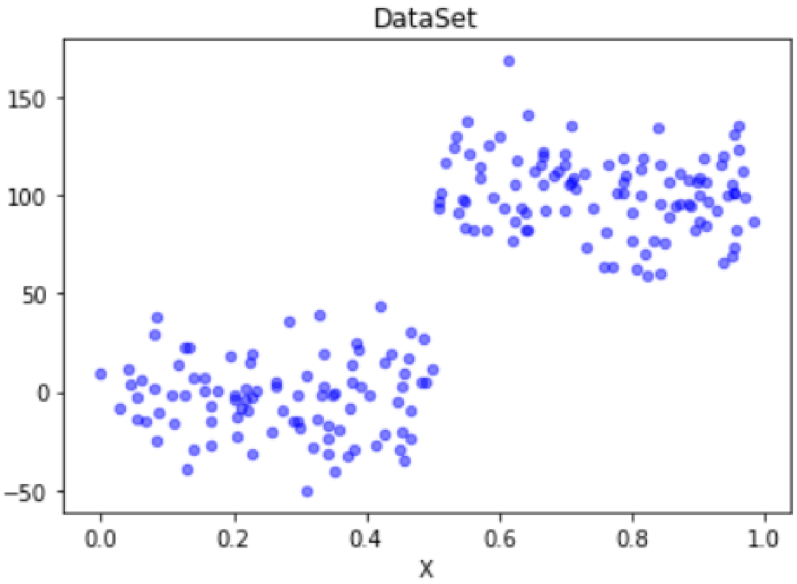
Mind. error eduction: 1

Mind. data instances: 4



LIKE

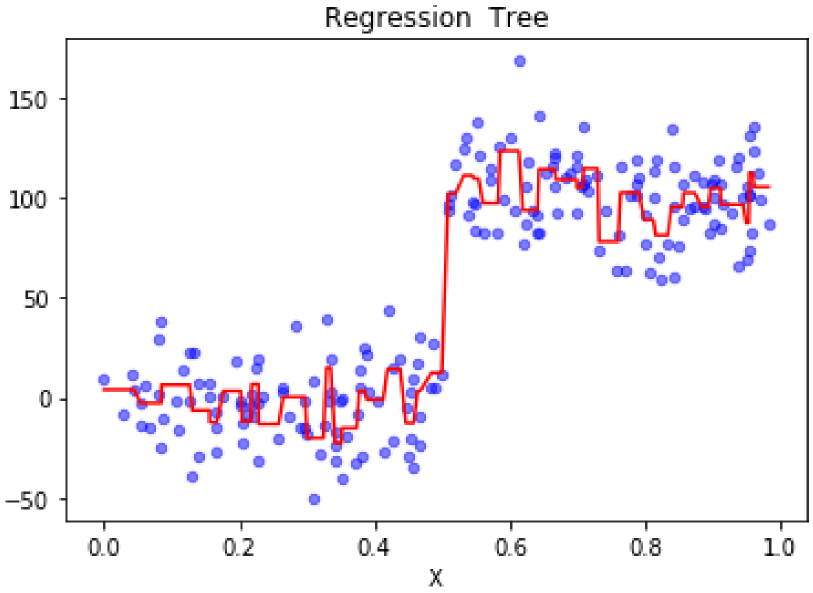
IV: Tree-based Regression



User-defined parameter:

Mind. error eduction: 1

Mind. data instances: 4

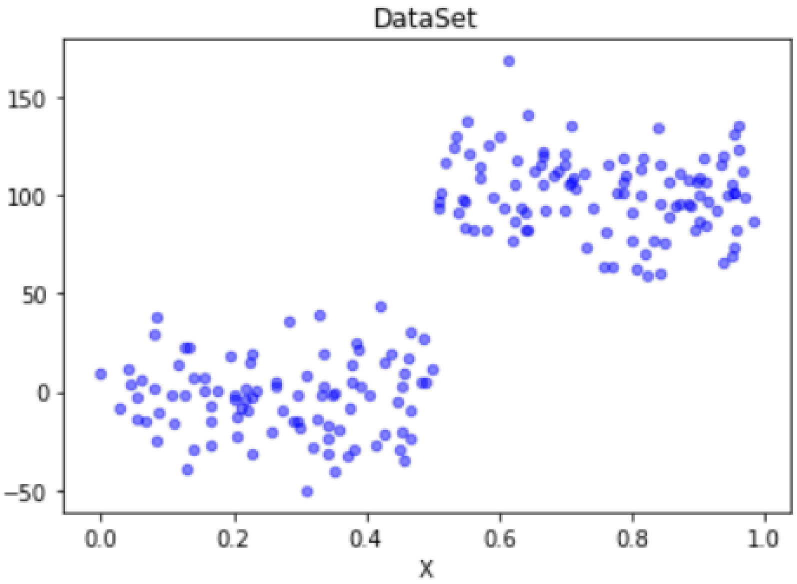


Too many nodes !!

Overfitting problem !!

LIKE

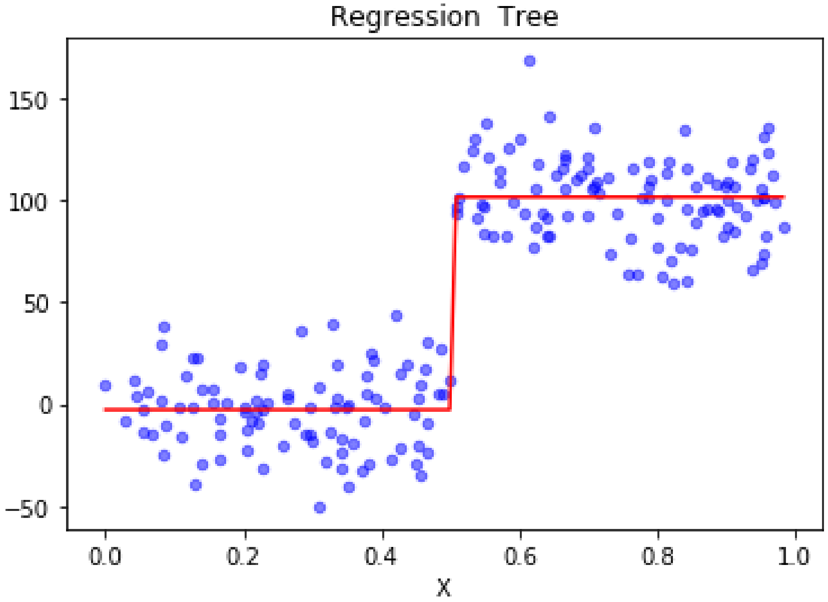
IV: Tree-based Regression



User-defined parameter:

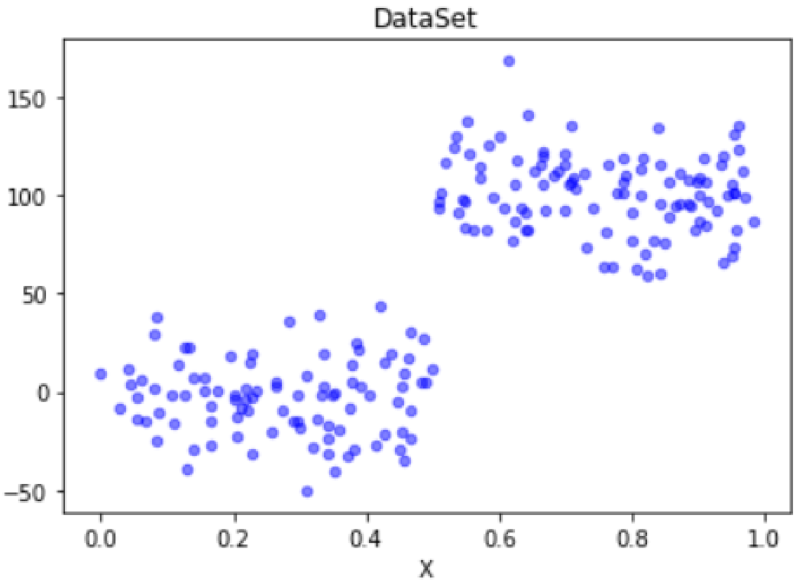
Mind. error eduction: ~~1~~ 10000

Mind. data instances: 4



LIKE

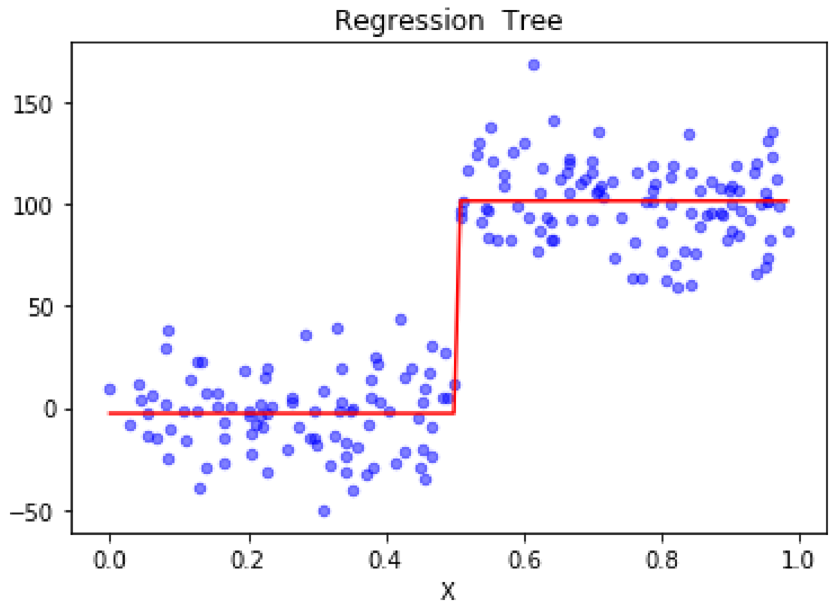
IV: Tree-based Regression



User-defined parameter:

Mind. error eduction: ~~1~~ 10000

Mind. data instances: 4



Is there any better method without user intervention?

→ Postpruning

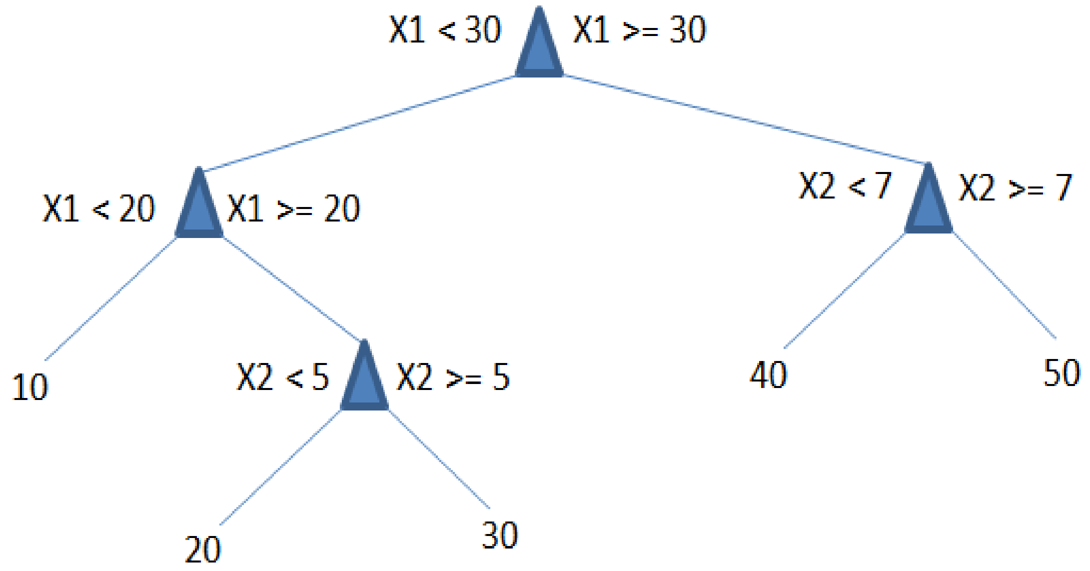
LIKE

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

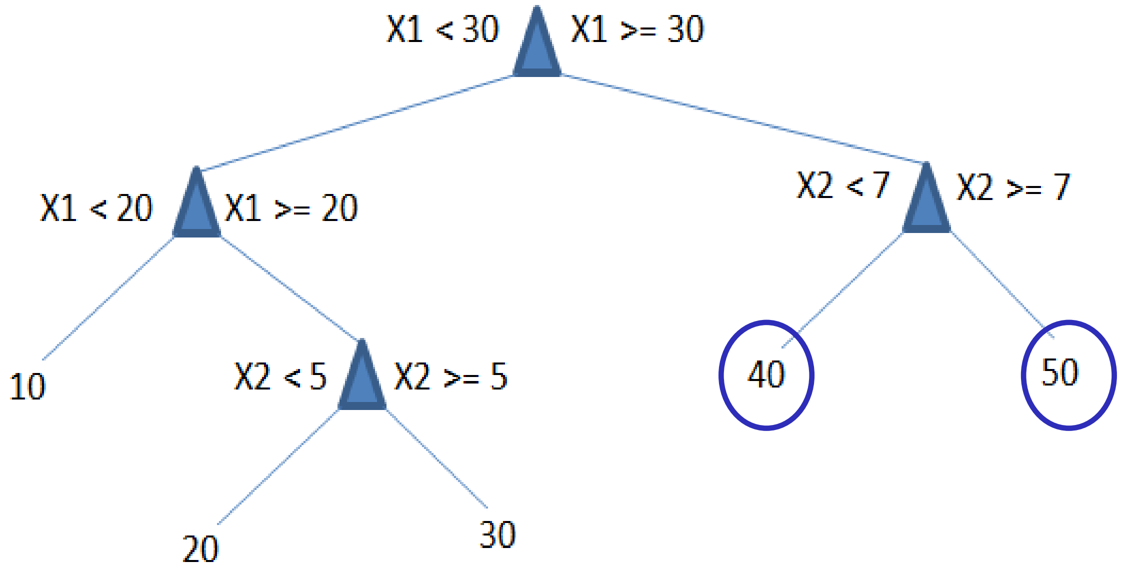
How to implement the postpruning?

- 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



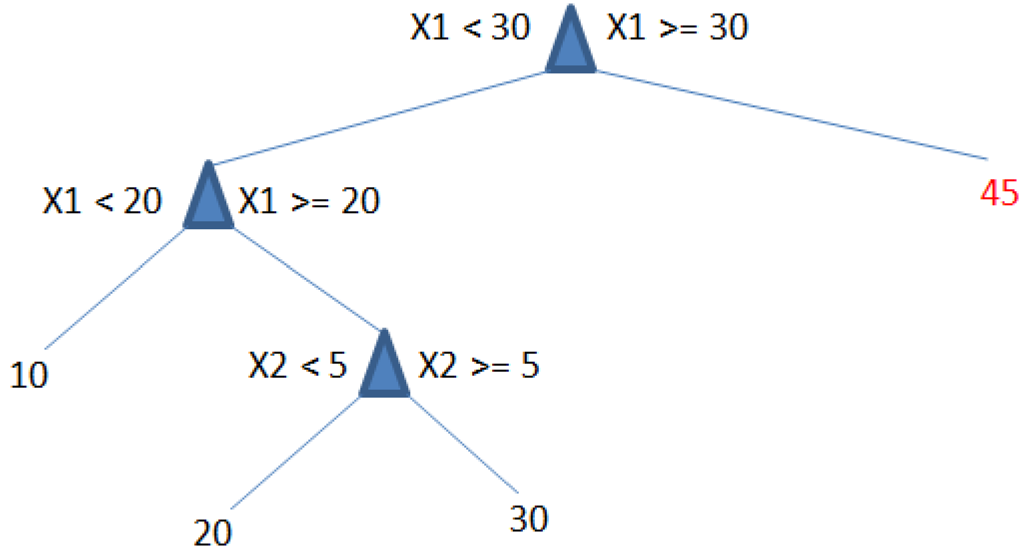
Calculate *Merge-Error* with 45

Calculate *Orig.-Error* with 40 and 50

LIKE

How to implement the postpruning?

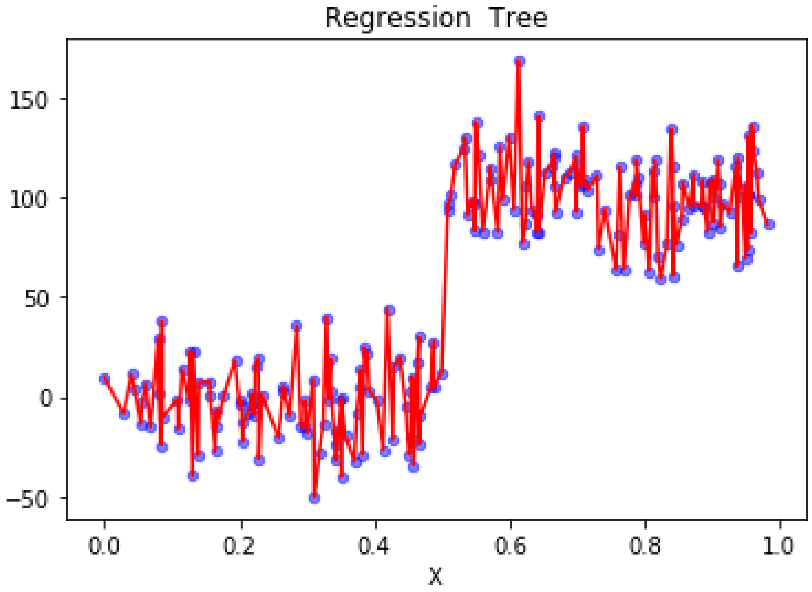
- Make the regression tree to a large depth
- Start at the bottom and combine every two leaf nodes(left and right) into a new terminal node, if such error after merge is smaller than the original one



If <u>*Merge-Error*</u> < <u>*Orig.-Error*</u>

then **merge**

Prepruning vs. Postpruning

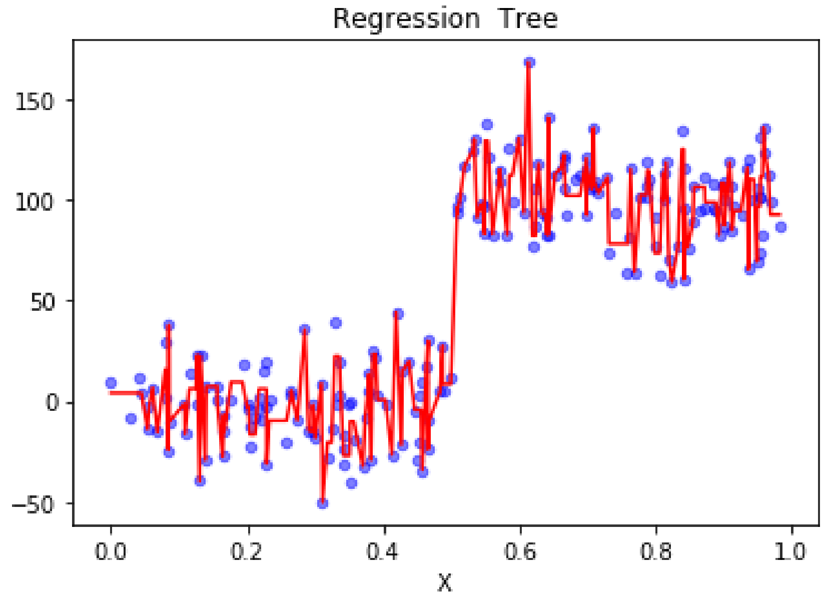


<u>Prepruning</u> (stop conditions)

Error reduction: 0

Min.data instances: 1

<u>Postpruning</u>



LIKE

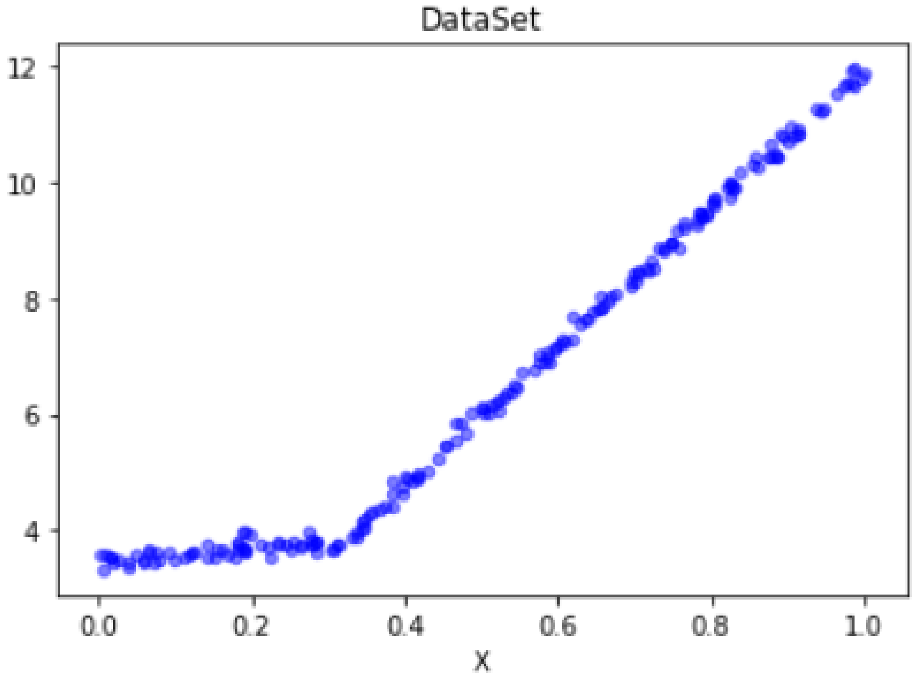
Prepruning vs. Postpruning

A large number of nodes were pruned off the tree, but it wasn't reduced to two nodes as we had hoped. It turns out that postpruning isn't as effective as prepruning.

We can employ *both* to give the best possible model.

LIKE

IV: Tree-based Regression

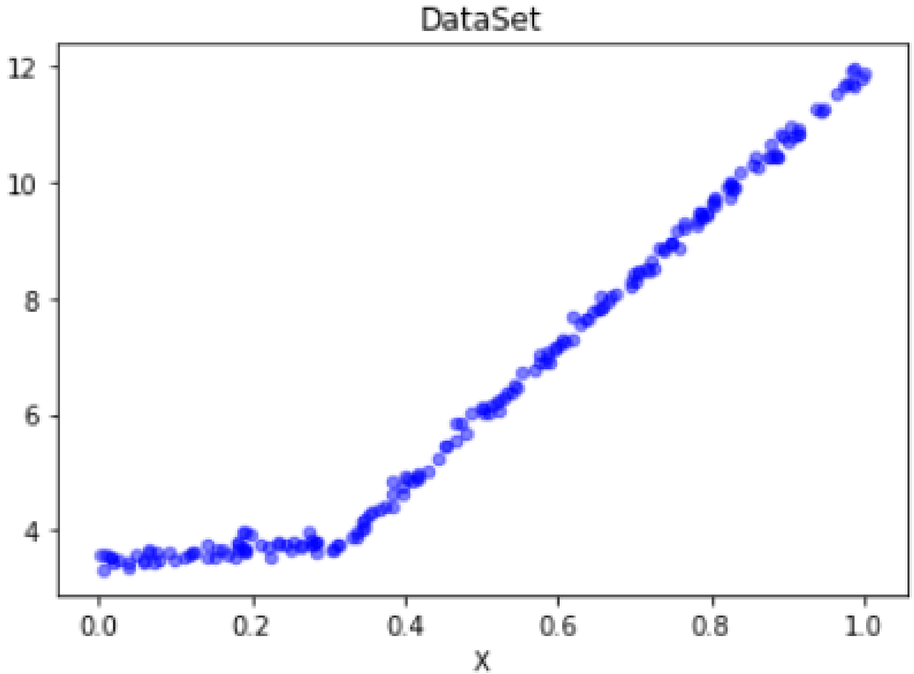


Would it be better to model this dataset as

- 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:

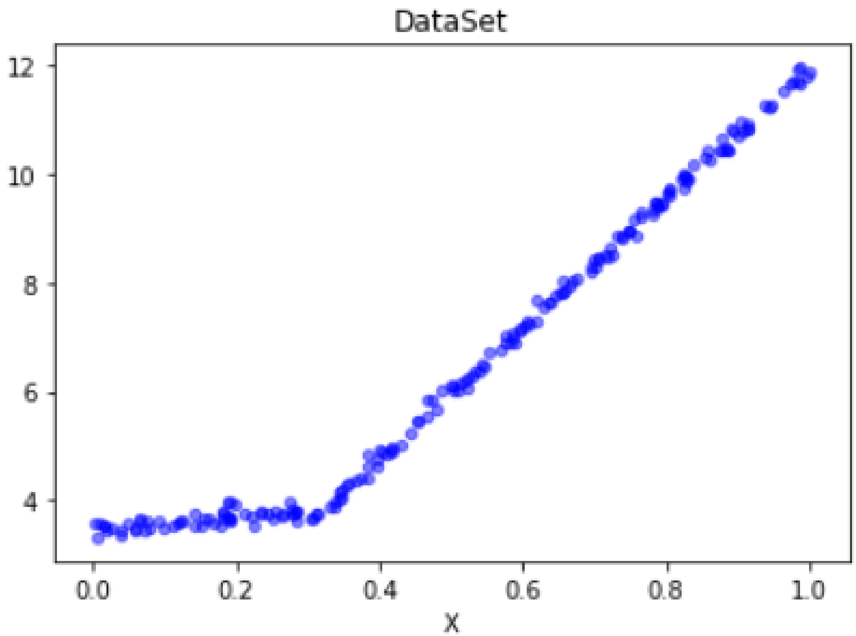
A way to model dataset as a <u>piecewise</u> linear model at each leaf node. Piecewise linear means that the model consists of multiple linear segments

LIKE

How to make the model tree?

- Use the tree-generating algorithm to break up the data into segment
- Use the linear regression to generate the linear model at the leaf nodes
- Use the least-squares method to determine the best weight

IV: Tree-based Regression



Result:

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

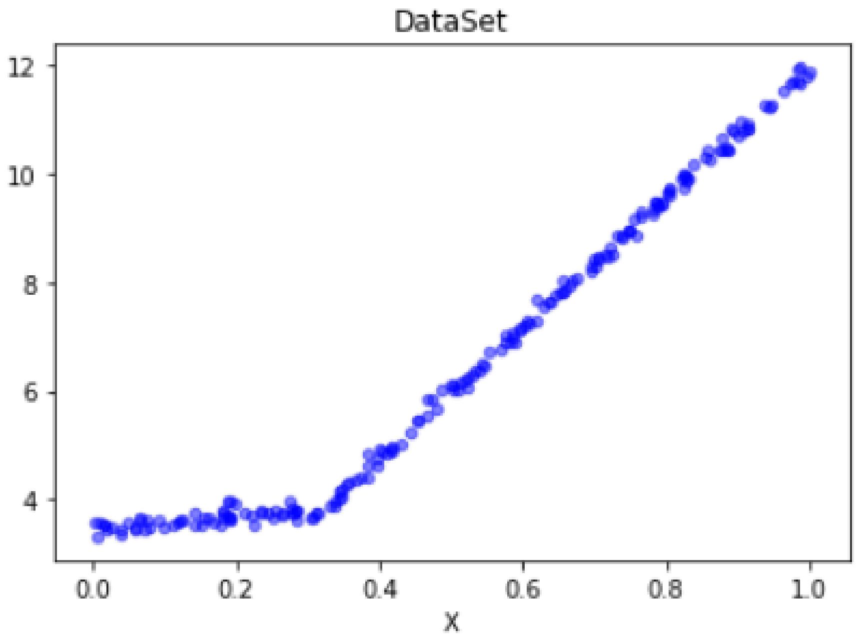
'left': matrix([[3.46877936], [1.18521743]]),

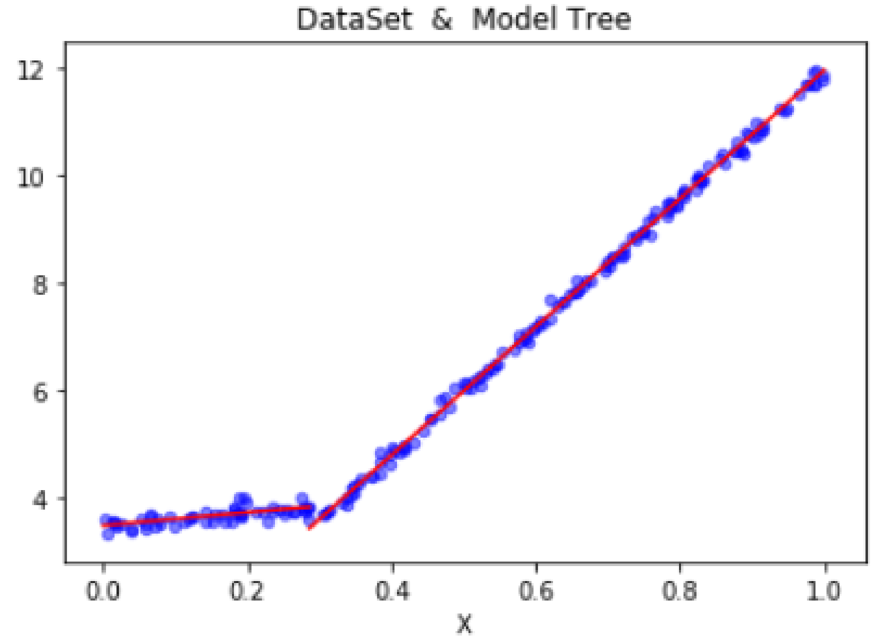
'right': matrix([[1.69855694e-03], [1.19647739e+01]])}

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

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]])}

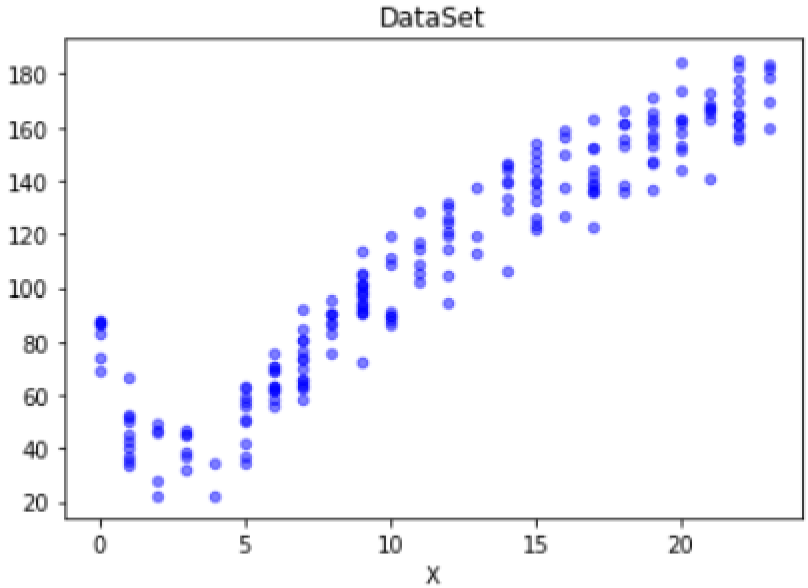
$$y = 0 + 11.96 * x \quad and \quad y = 3.47 + 1.20 * x$$

LIKE

V: Compare Linear Regression and Tree Regression

Regression methods:

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



Evaluation method:

- Numpy.corrcoef()

Correlation coefficients:

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

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.

LIKE

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