Trees

This week…

Lecture will cover…

- Decision Trees (in context of regression)
- Bagging
- Boosting
- Random Forests

You should read…

- ESL/ISLR chapters on trees, random forest, boosting & bagging

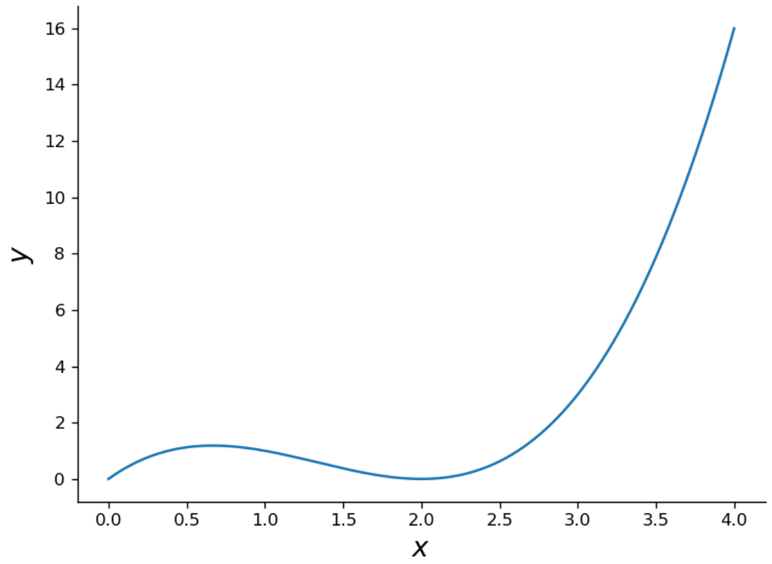
Heuristic For Function Approximation

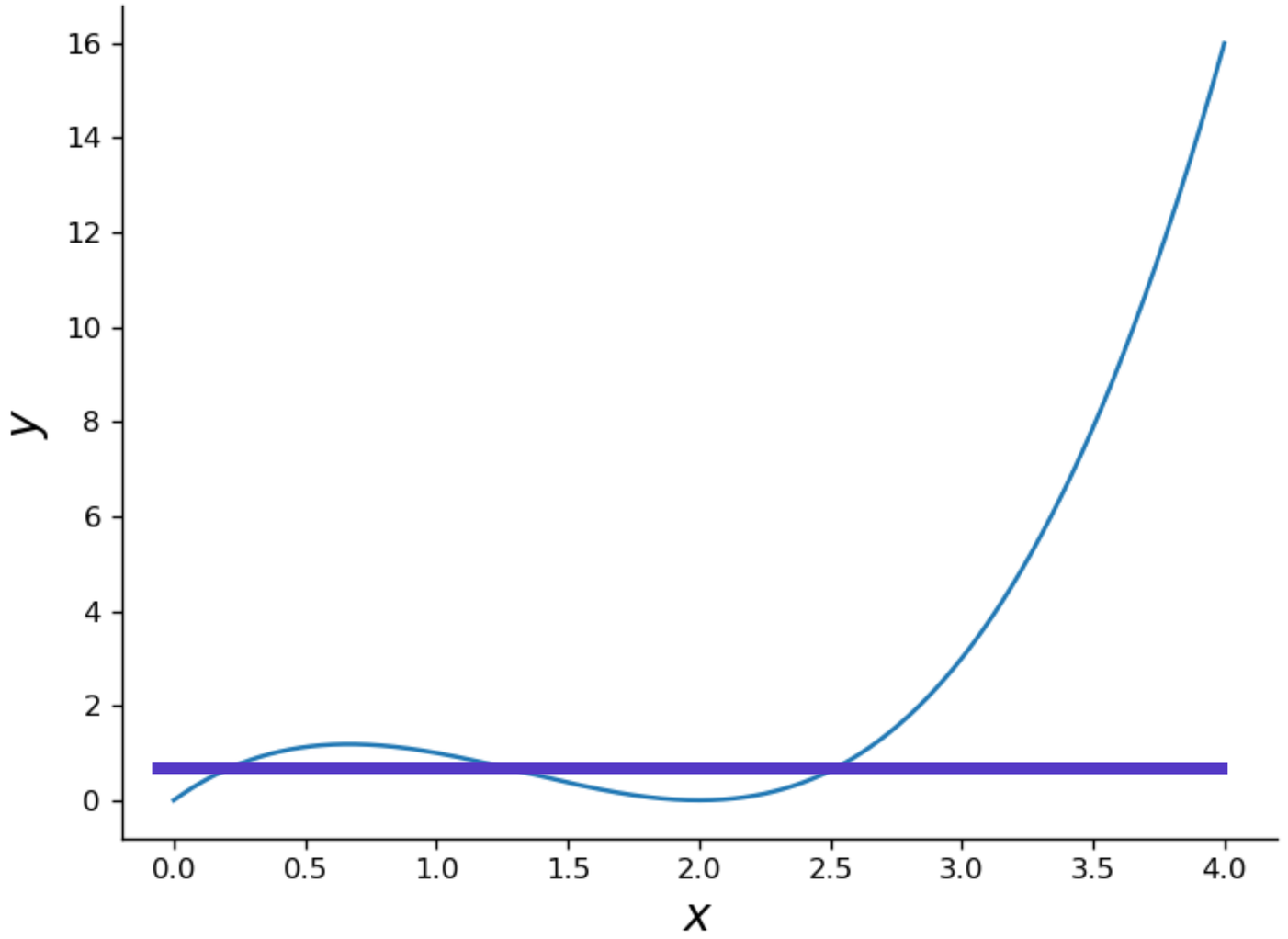
Here is a function we want to learn

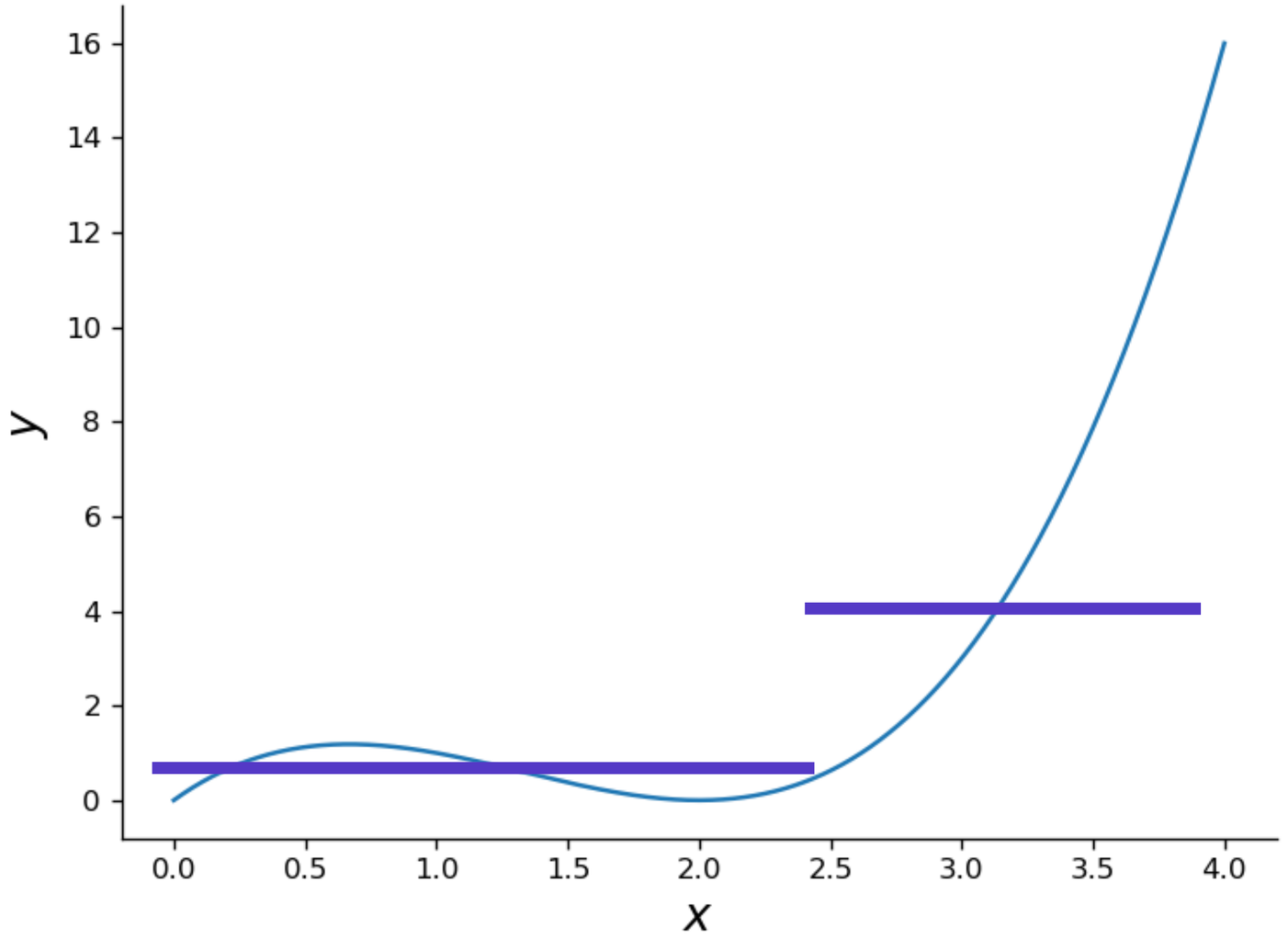
Want a completely non-parametric way to make predictions

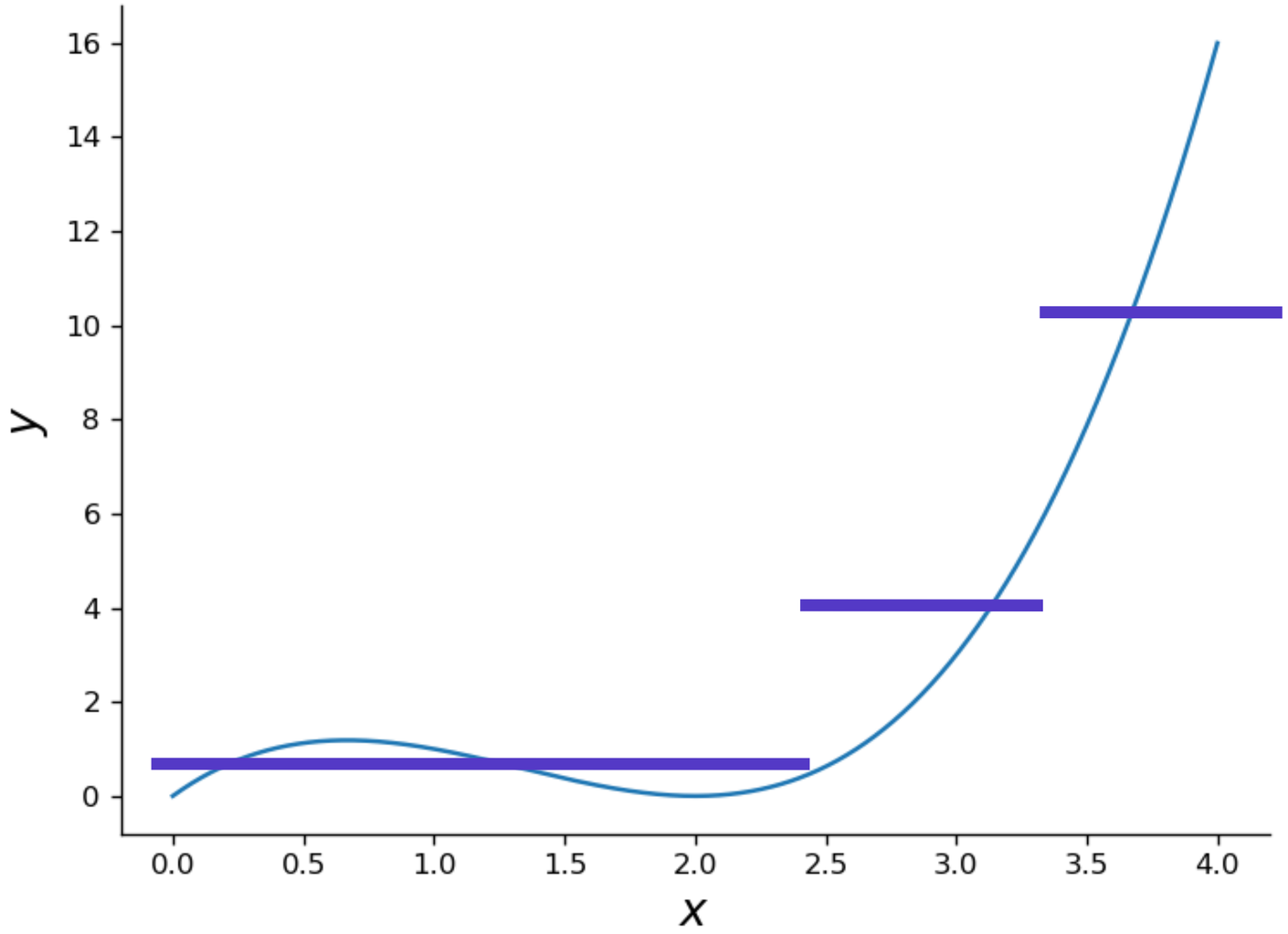
Why not approximate with a flat line?

Let’s start by creating a heuristic, and we can formalize it later




16
14
12
10
8
6
4
2
0
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x


16
14
12
10
8
6
4
2
0
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x

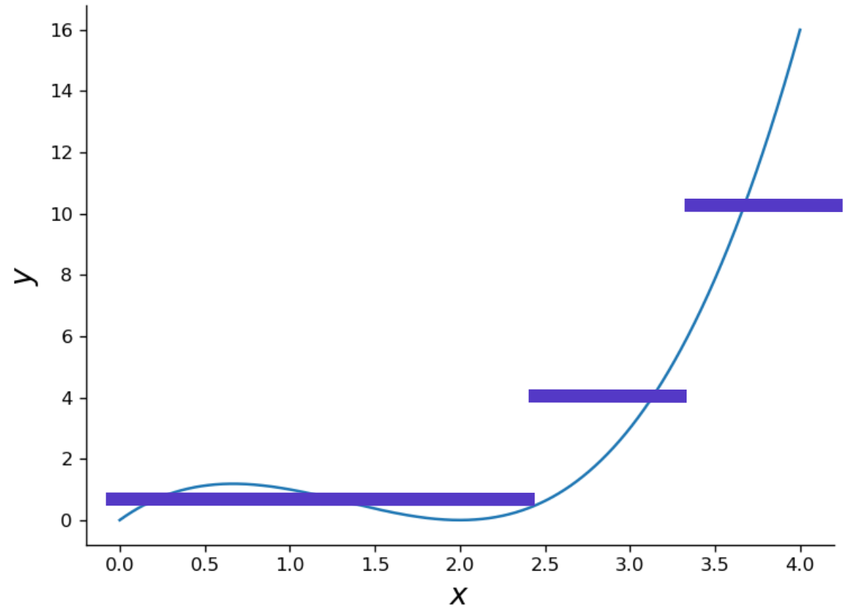

16
14
12
10
8
6
4
2
0
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x

Strategy

Strategy is to approximate the function with constant functions

Number of constant functions is tuneable

Pick a the height of the constant functions to minimize some loss



This is a decision tree!

More Formally

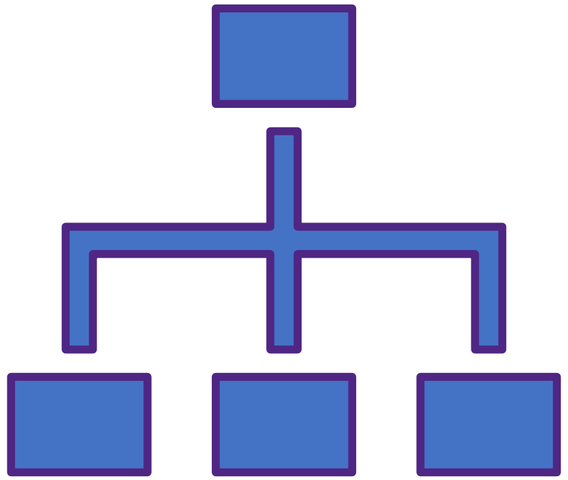
Trees recursively split the feature space into (hyper)-rectangles and fit a constant function to each (hyper)-rectangle

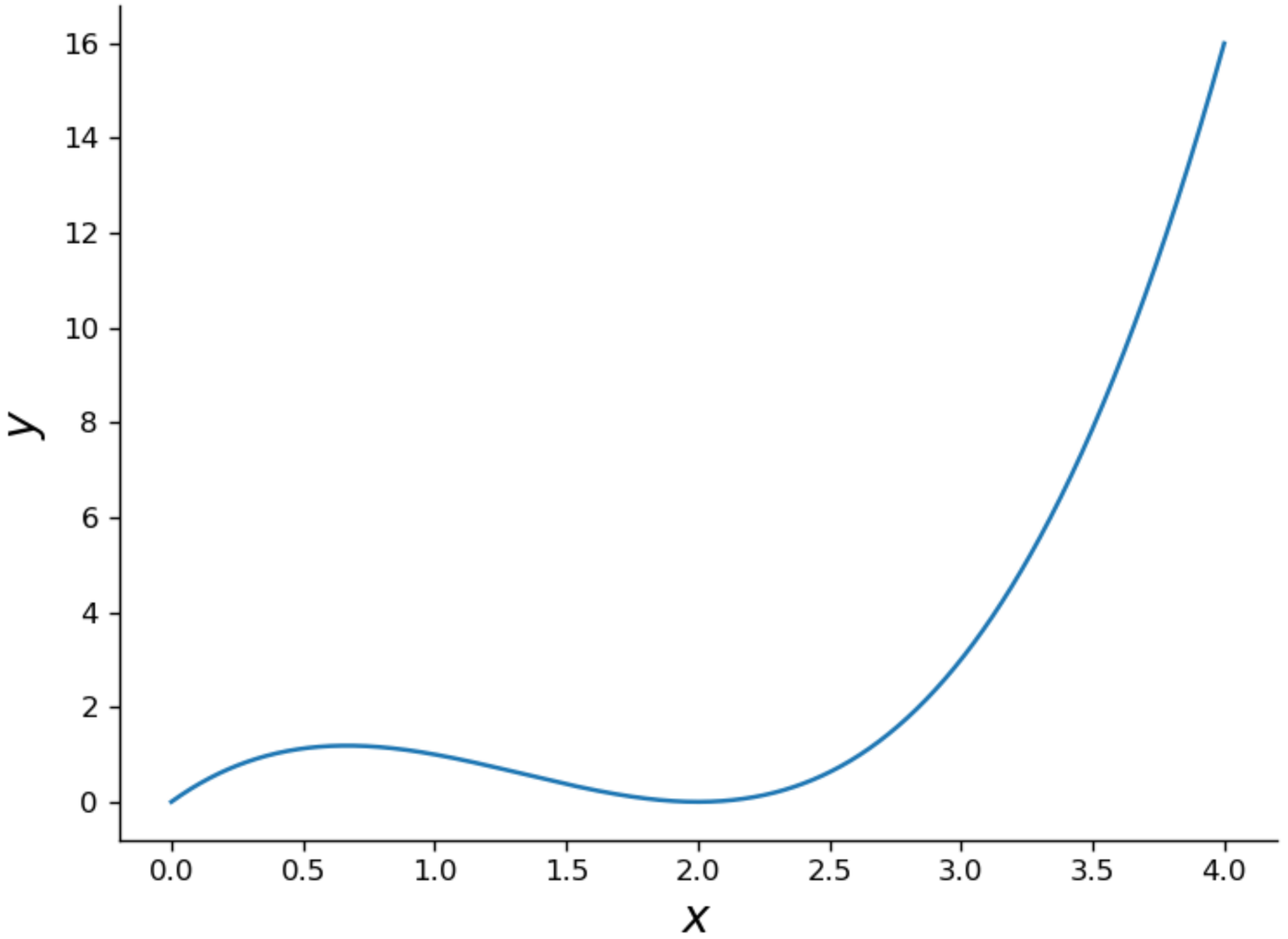
Greedily decide where to split by determining which split leads to largest gain in accuracy/reduction in loss

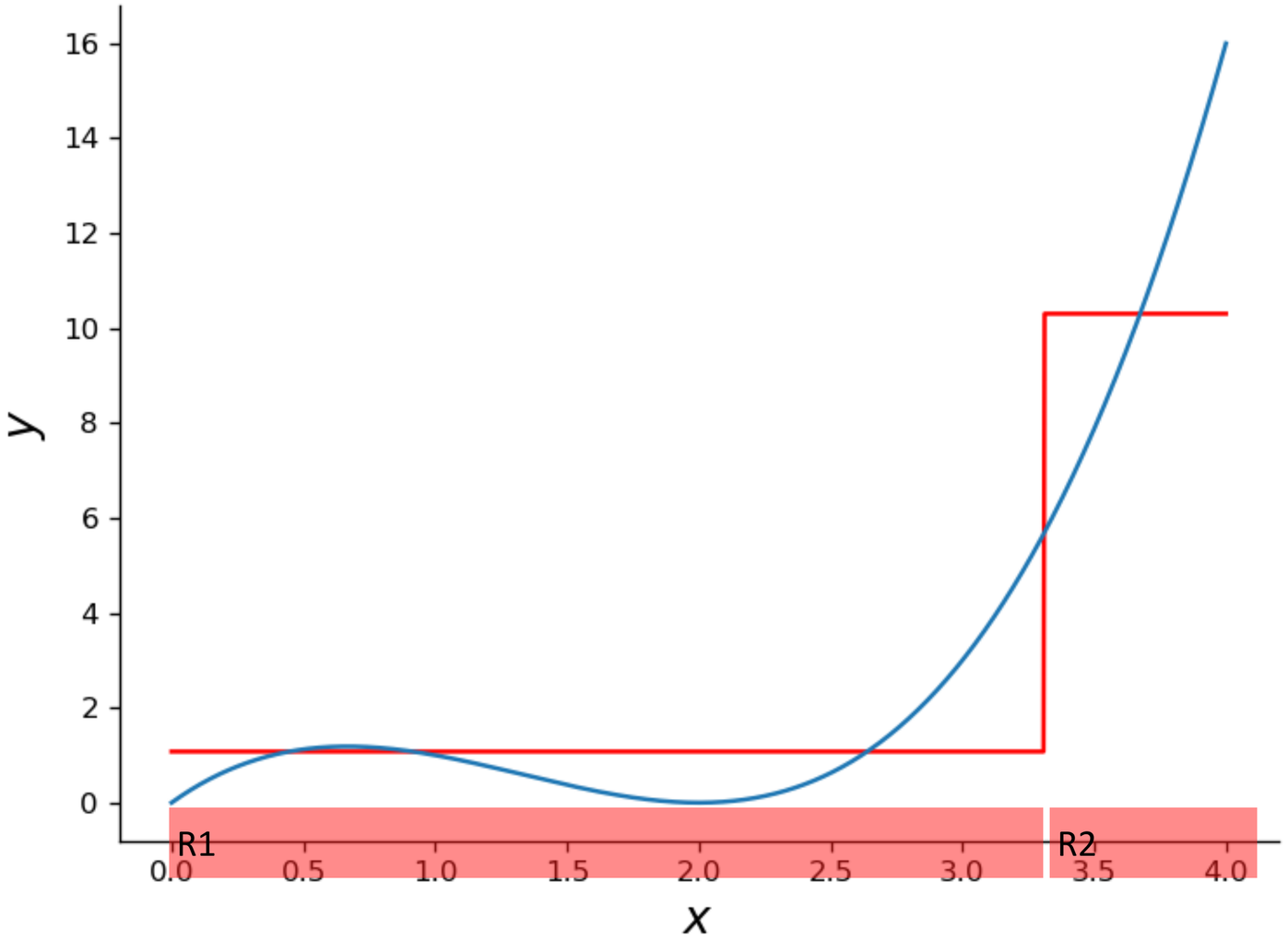
Terminate when

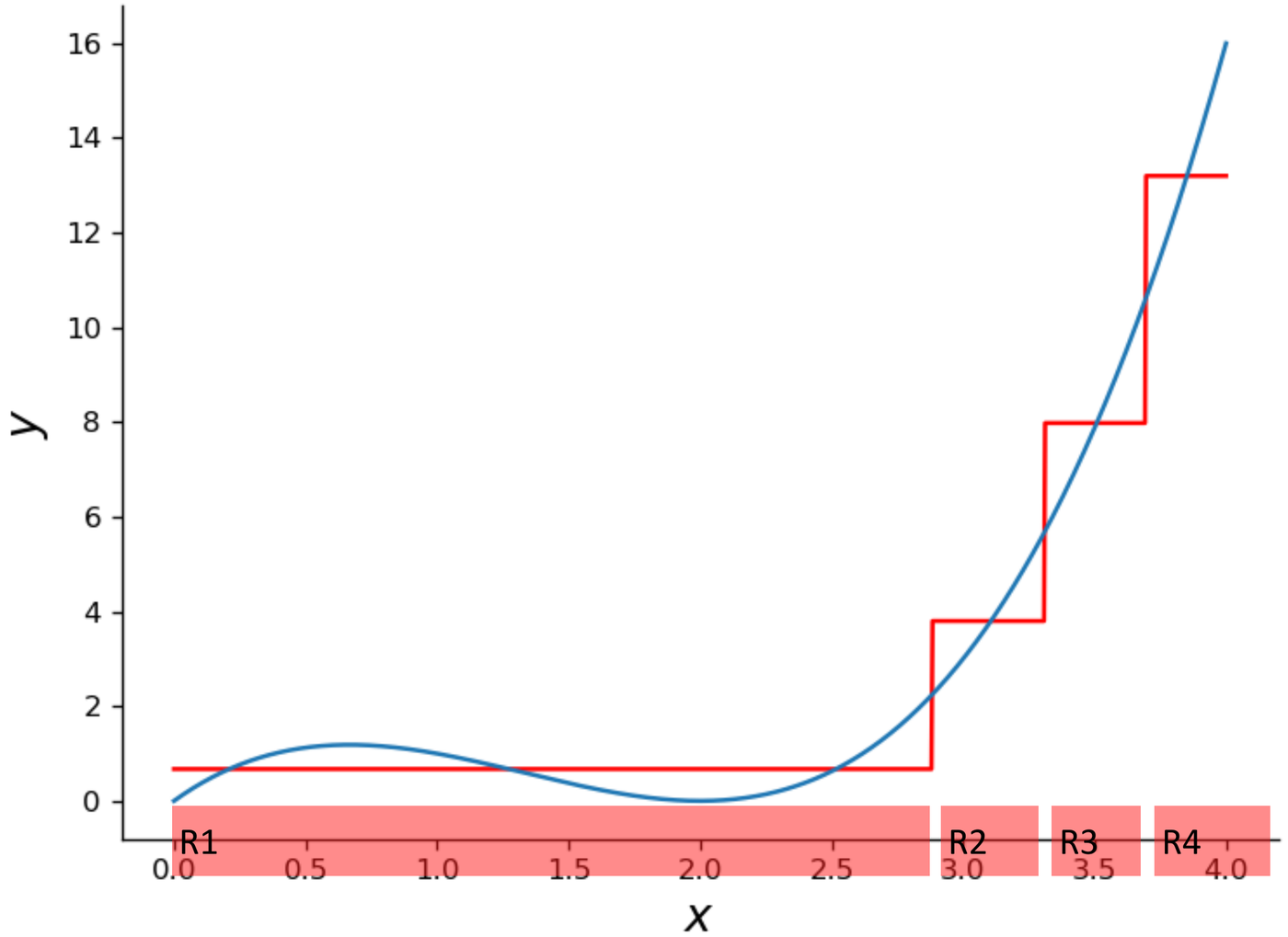
- Made enough splits, or
- Rectangles have some minimum number of observations

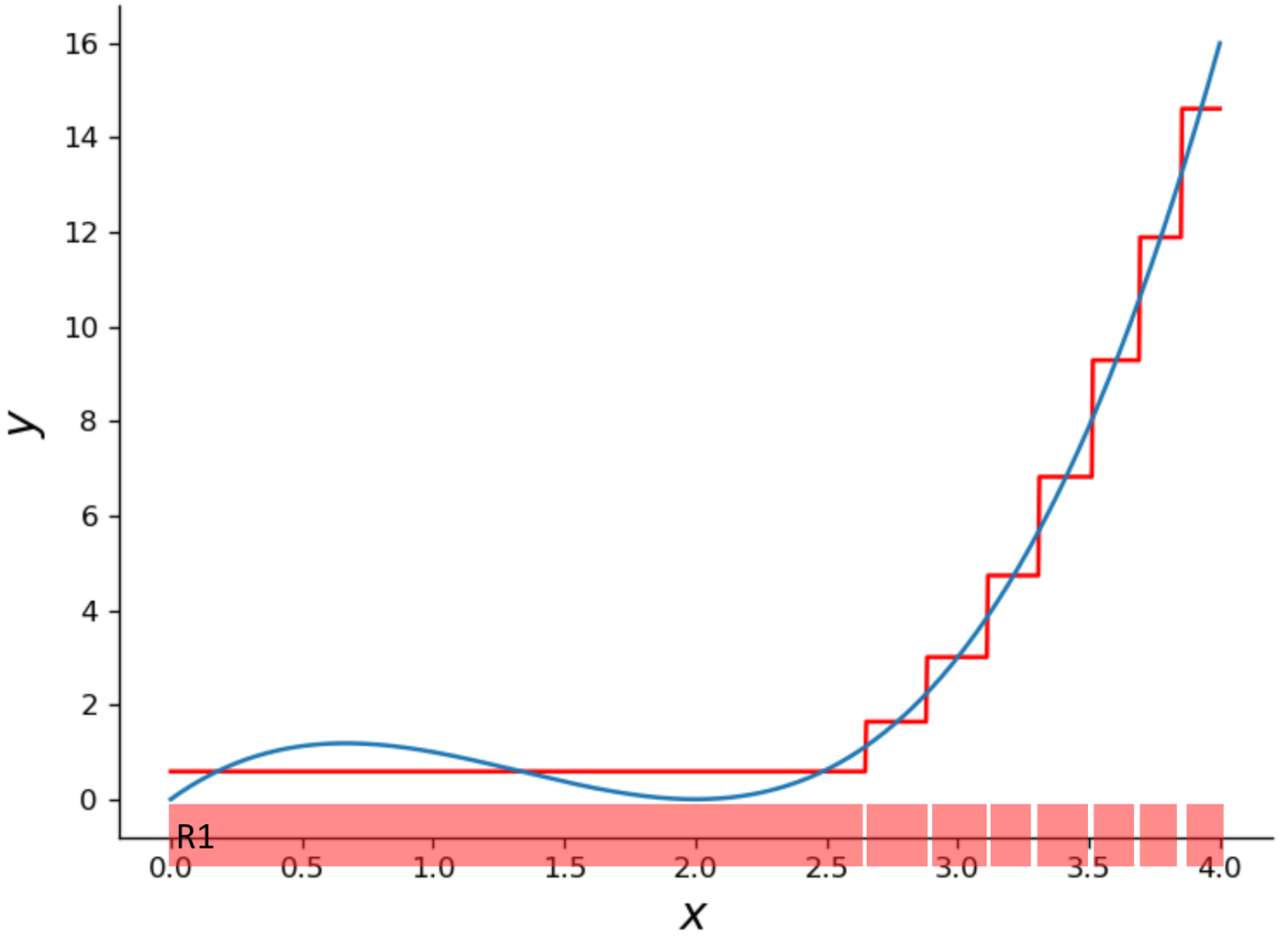
Let's see an example

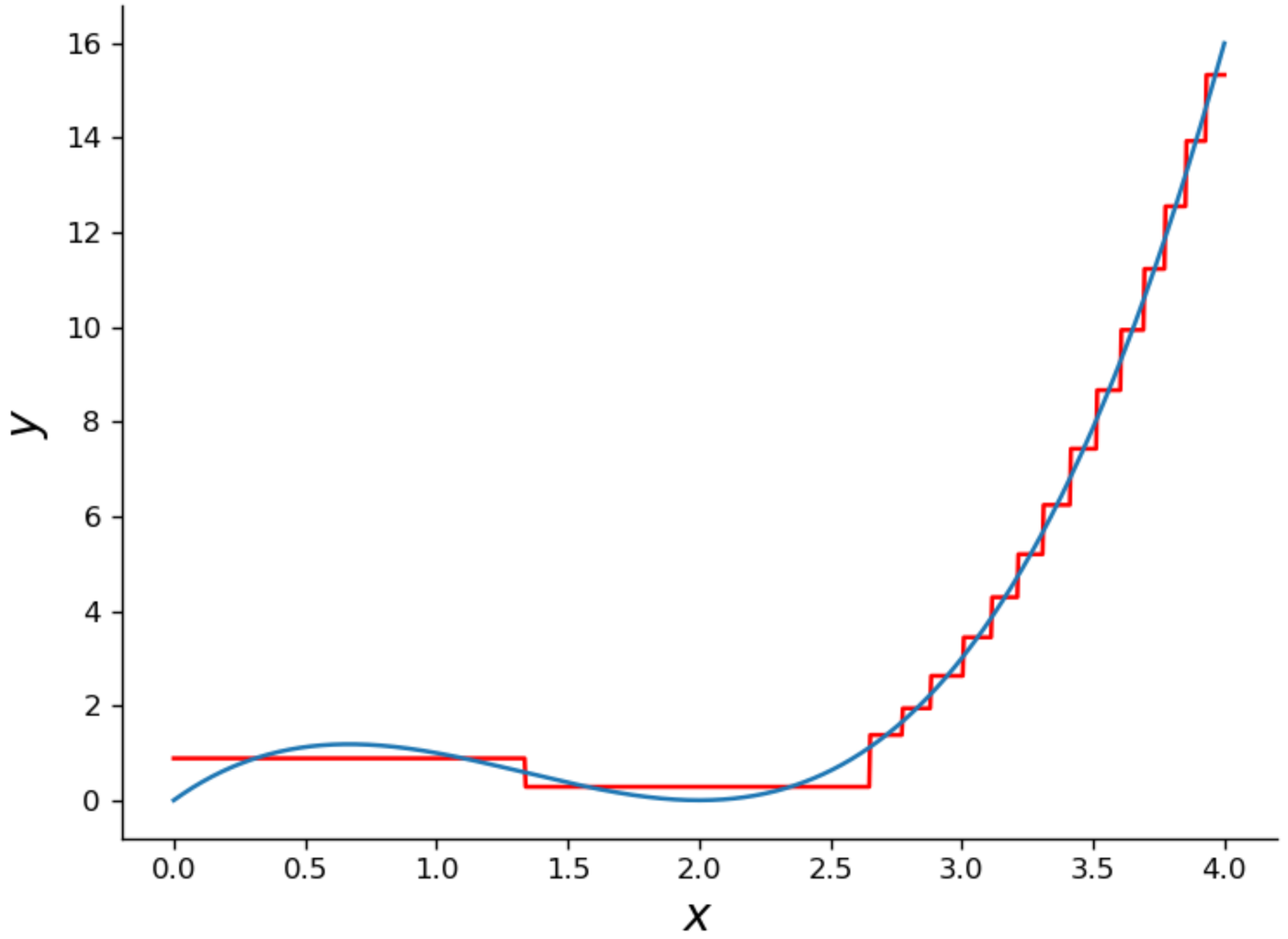


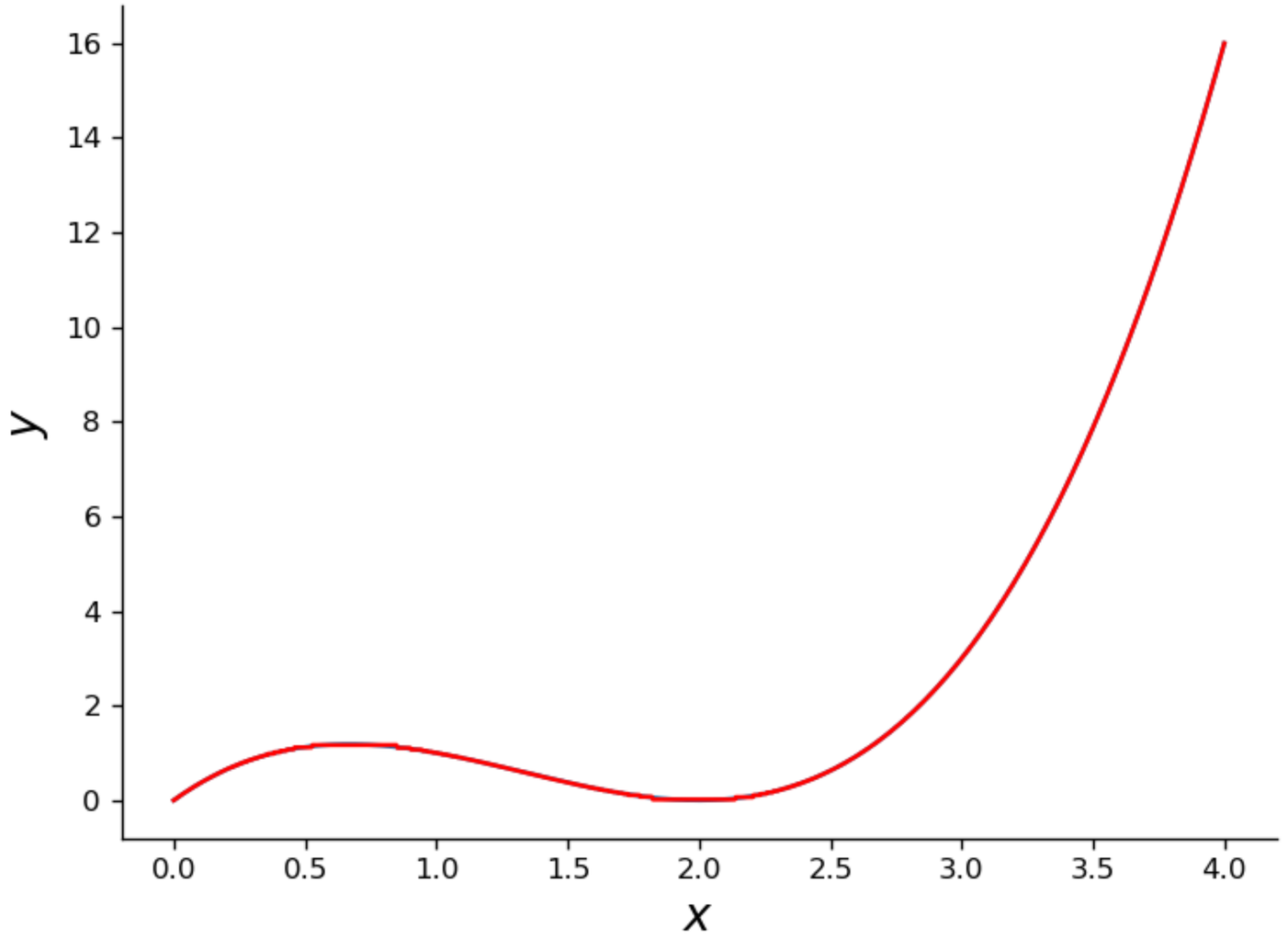

16
14
12
10
8
6
4
2
0
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x


16
14
12
10
8
6
4
2
0
y
R1
R2
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x

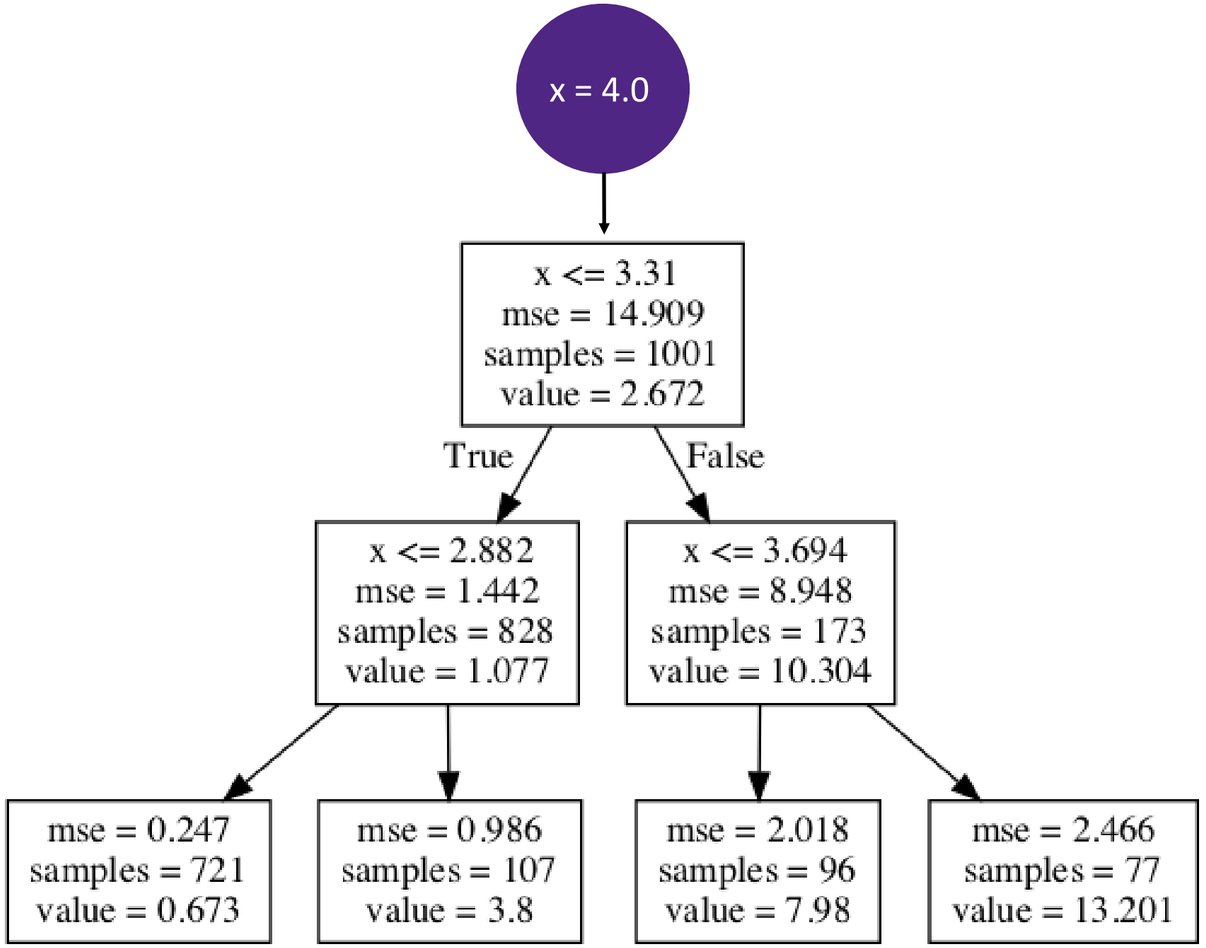

16
14
12
10
8
6
4
2
0
y
R1
R2
R3
R4
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x


16
14
12
10
8
6
4
2
0
y
R1
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x

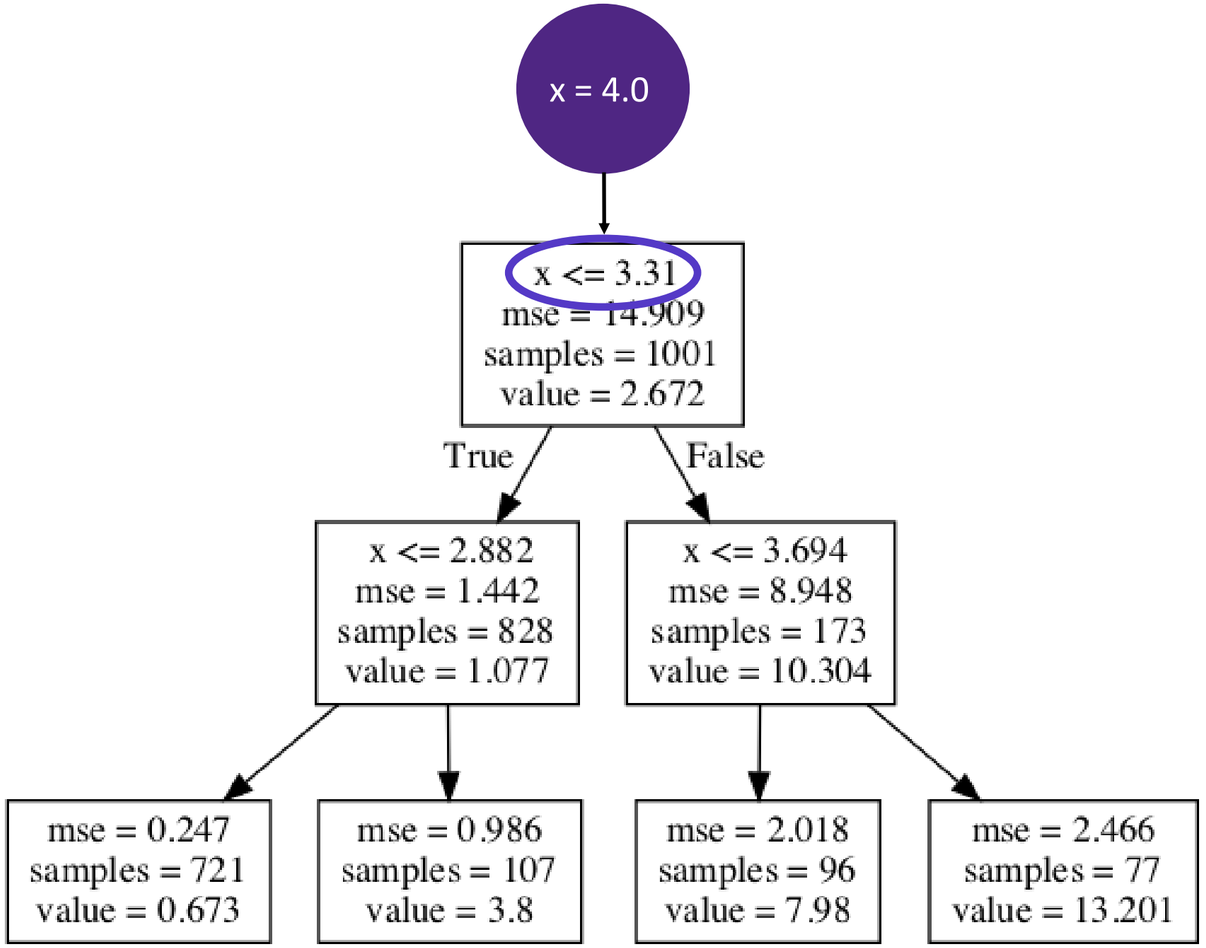

16
14
12
10
8
6
4
2
0
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x


16
14
12
10
8
6
4
2
0
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x

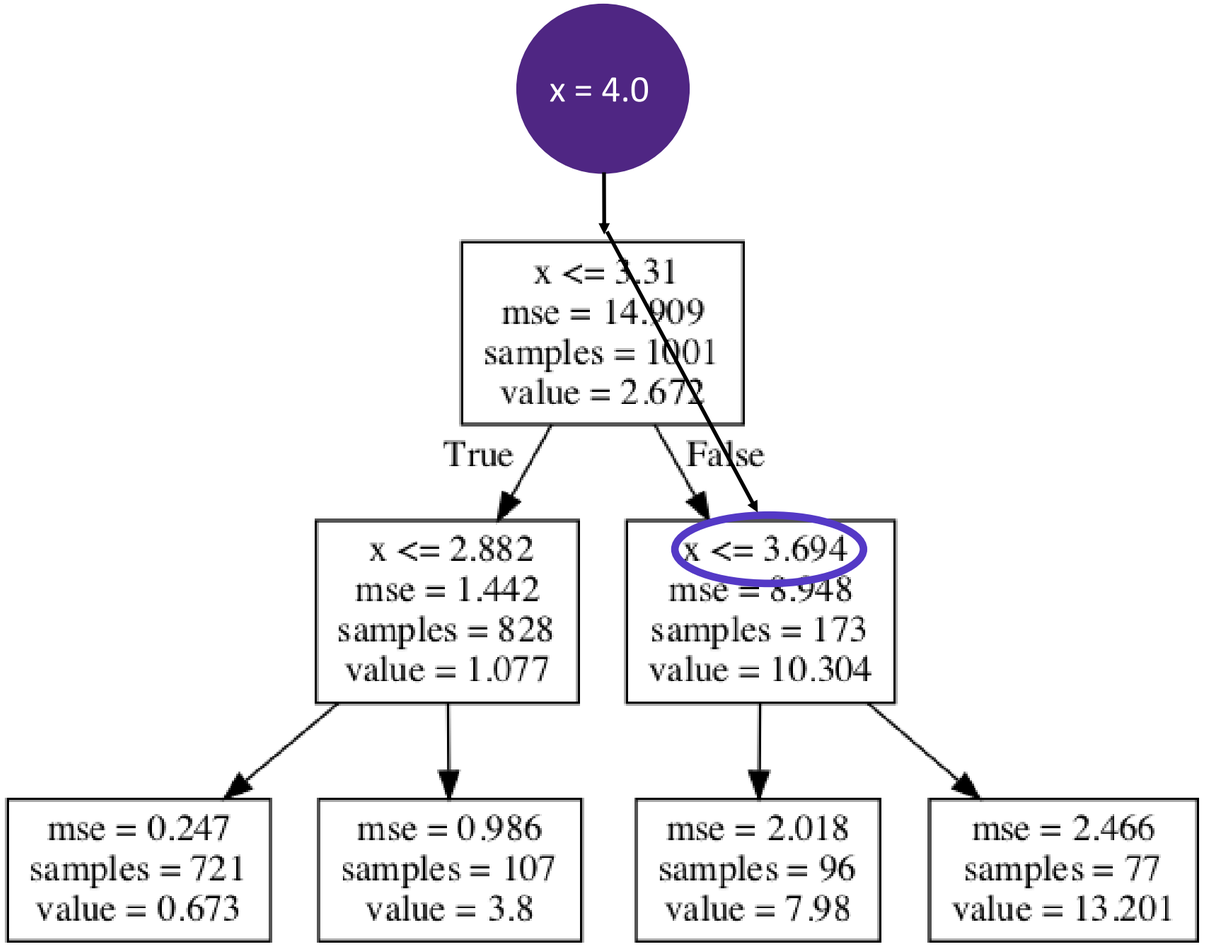
Visualizing The Process



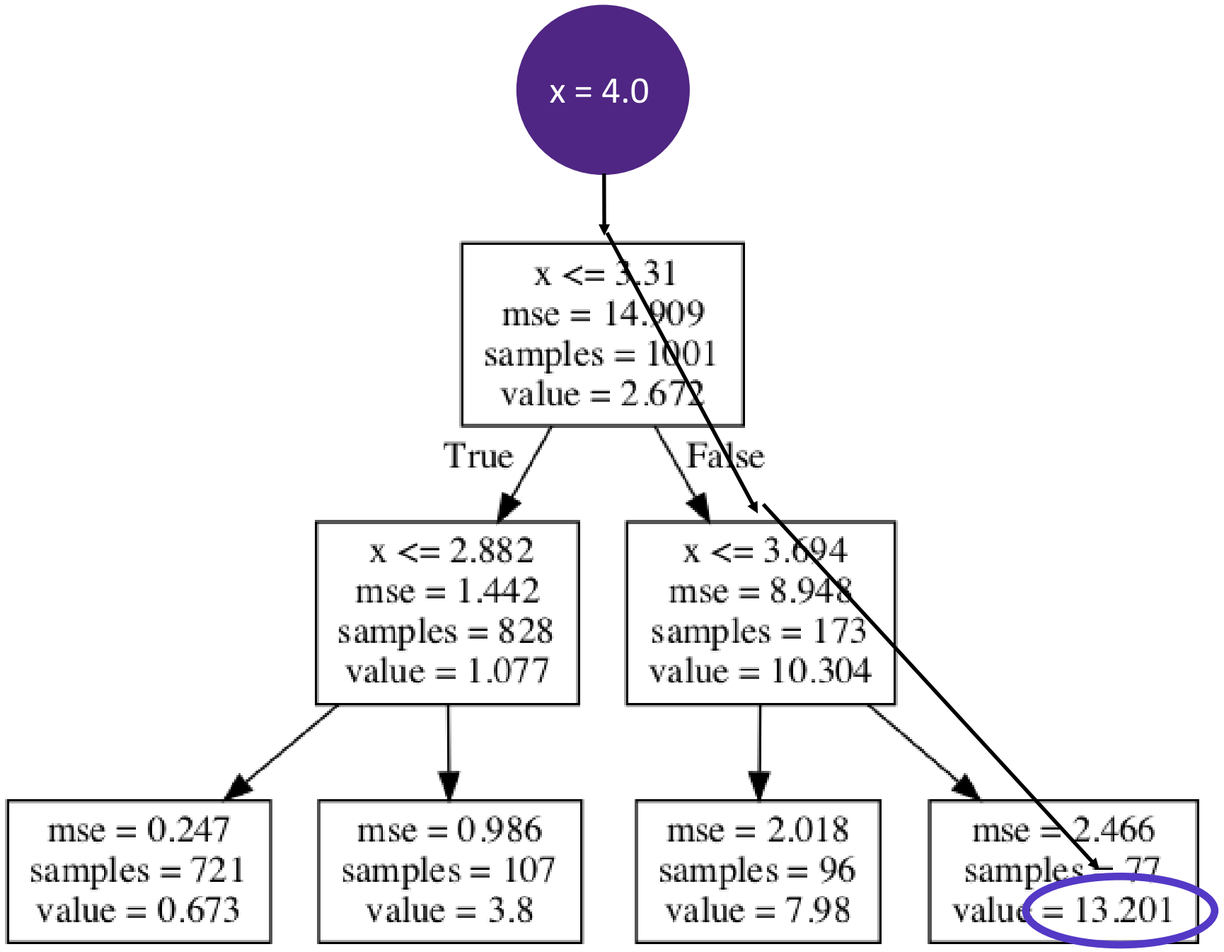
Visualizing The Process



Visualizing The Process

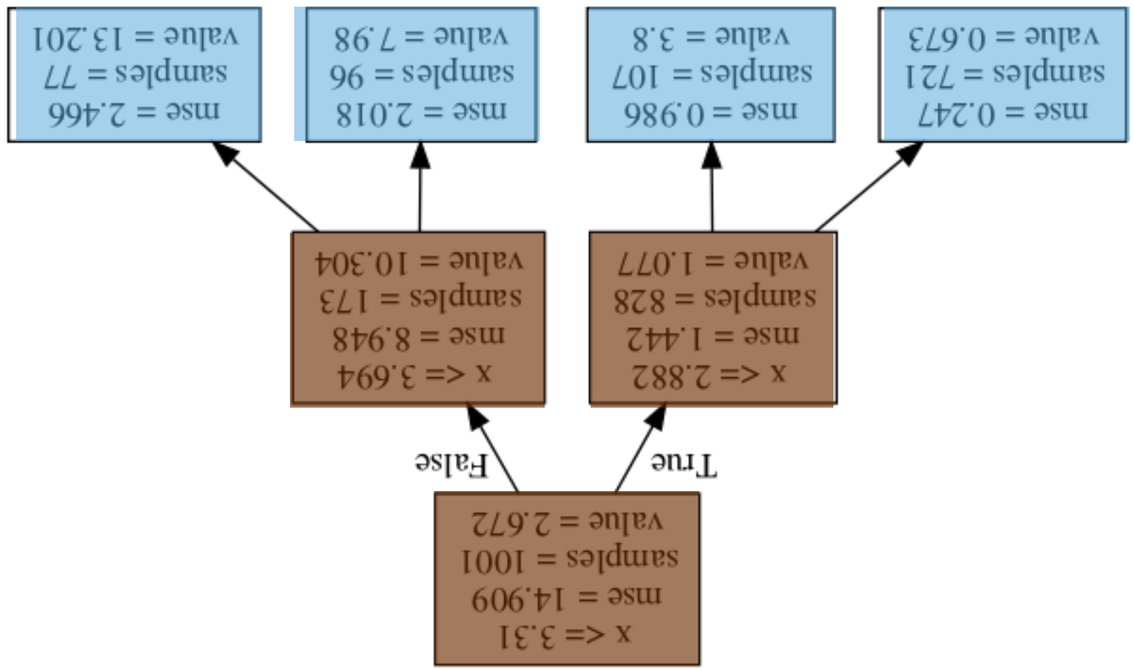


Visualizing The Process



Why Call These Methods “Trees”?

Call terminal nodes “leaves”

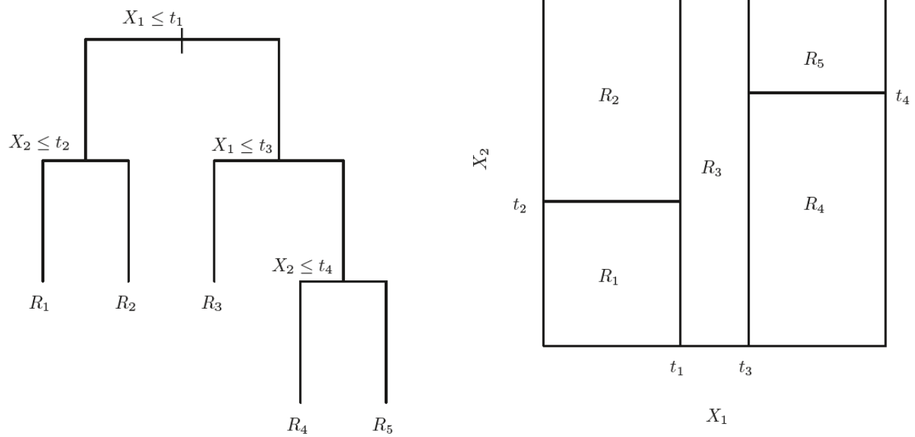


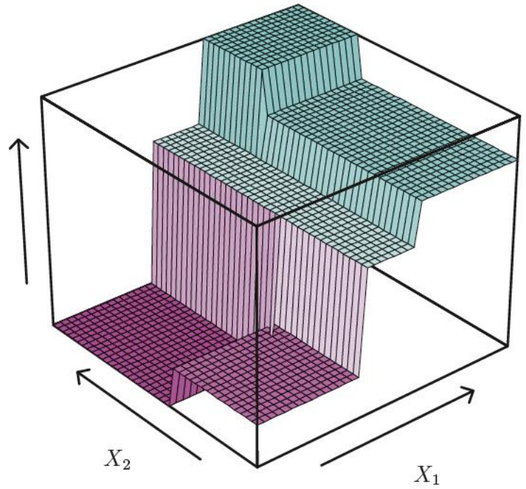
Method should be called Decision Roots

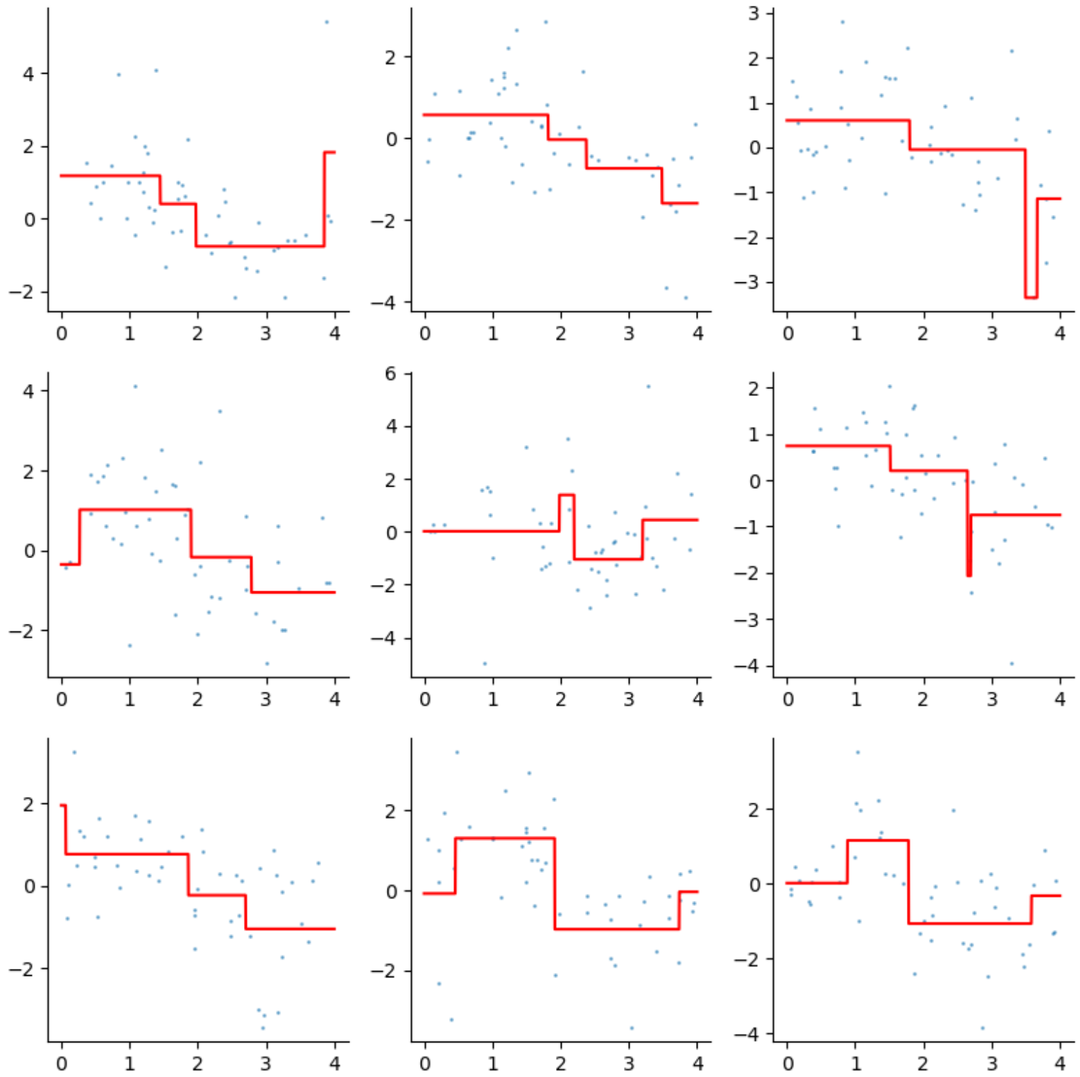
What About More Dimensions

Same process, but now we can split on other variables!

Node still is a decision on one feature.







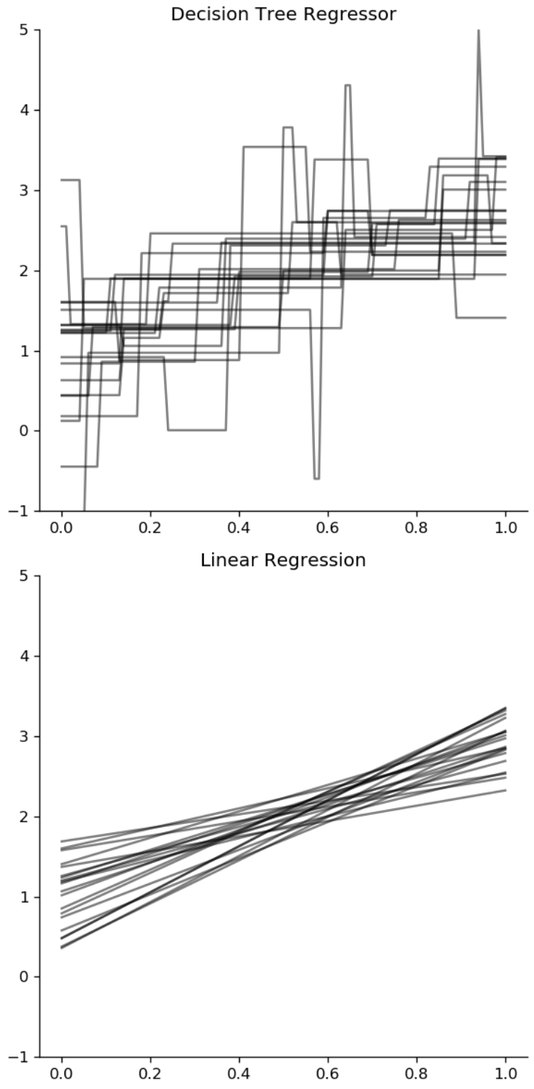
Trees Are High Variance

Data from same data generating process

Tree depth is 2 (not that much)

Wildly different predictions from data which is from same dataset

That is what high variance means



Trees Are High Variance

Simulate 20 datasets from from y = 2x + 1

Student t noise with df = 10

Tree predictions vary quite a lot.

Linear regression shows less variation

Trees are an unstable model! (Ironically, this became one of their strengths as we'll see later)

Effect of an error in the top split is propagated down to all of the splits below

Trees also lack smoothness! Recursive splits mean the resulting estimates are discontinuous. Performance in regression can be degraded in underlying processes is smooth

Summary

PROS

Trees are highly flexible!

Trees are non-parametric!

Trees are invariant to scale and can handle categorical predictors naturally!

Trees are **INTERPRETABLE** and easy to described to managers, marketers, etc.

CONS

Tree starts to fit the noise (overfit) and not the signal when we go too deep. (**Important to cross validate!**)

Trees lack smoothness for regression

Sklearn has a great list of pros and cons should you need to look them up.

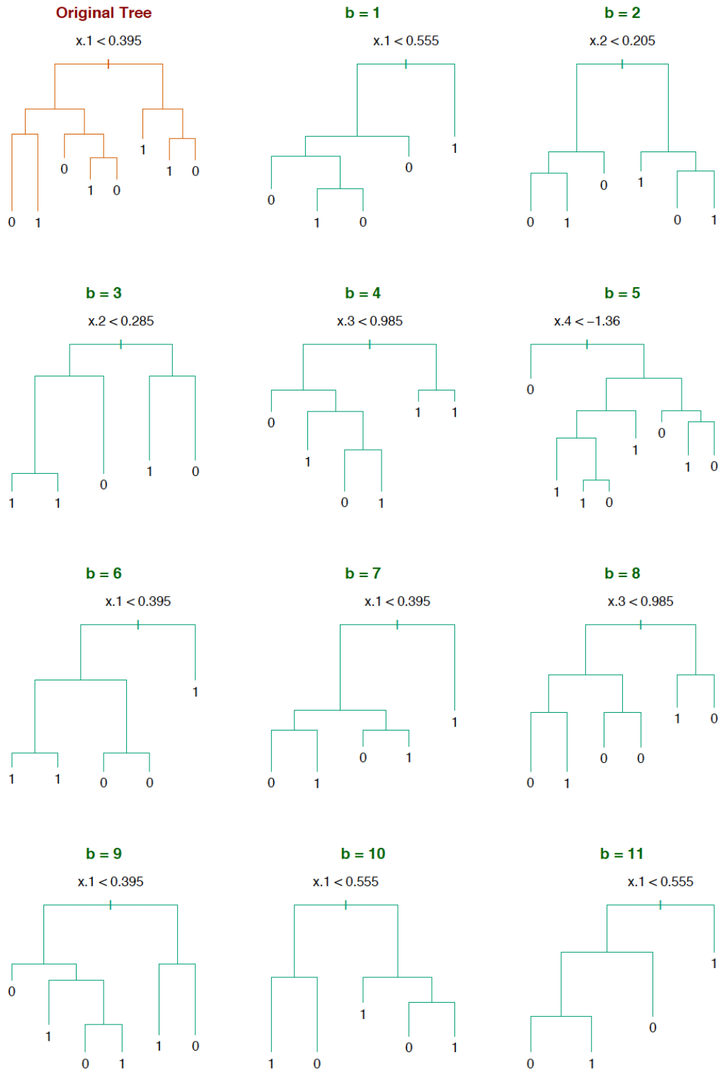
How Can We Reduce Variance?

Trees have good properties. Can we combat the overfitting?

Yes. A few ways to do this, including but not limited to:

- Bagging
- Boosting
- Creating a Random Forest

Bagging



Bagging

Short for “**B**ootstrap **Agg**regation”.

Bootstrap B datasets

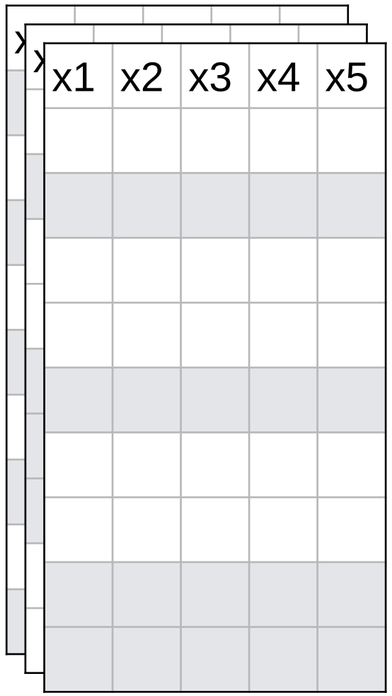
For b =1…B, fit a deep tree (high variance)

Combine trees and average B predictions

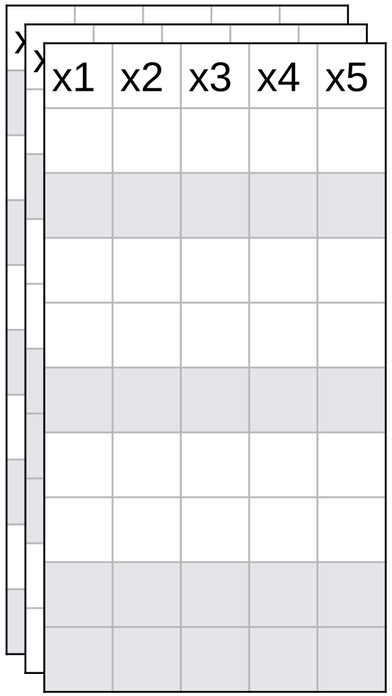
Can be done for other estimators too!

x1	x2	x3	x4	x5

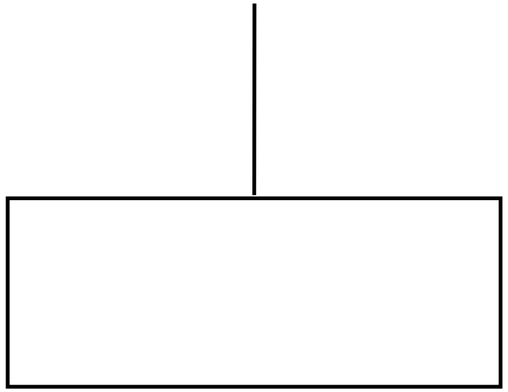
Take our
training data

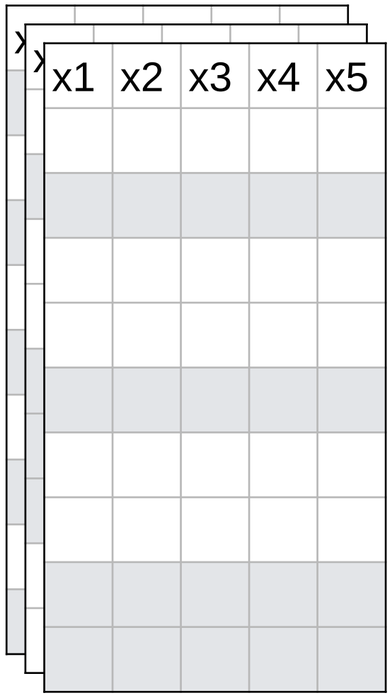


Bootstrap
B training datasets

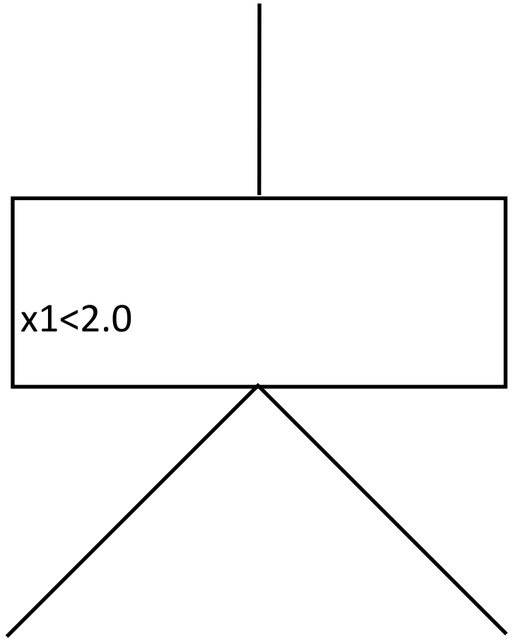


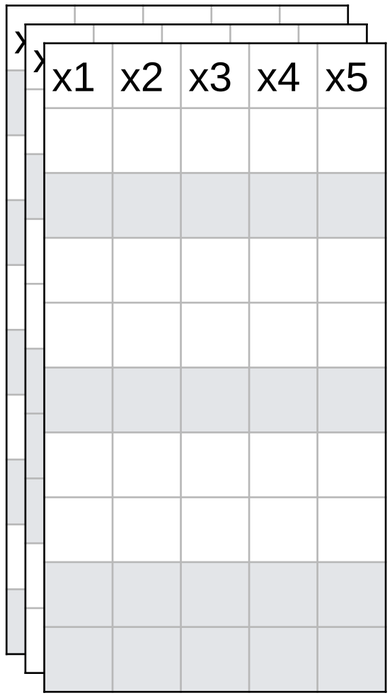
For the first split,
Show the tree the data



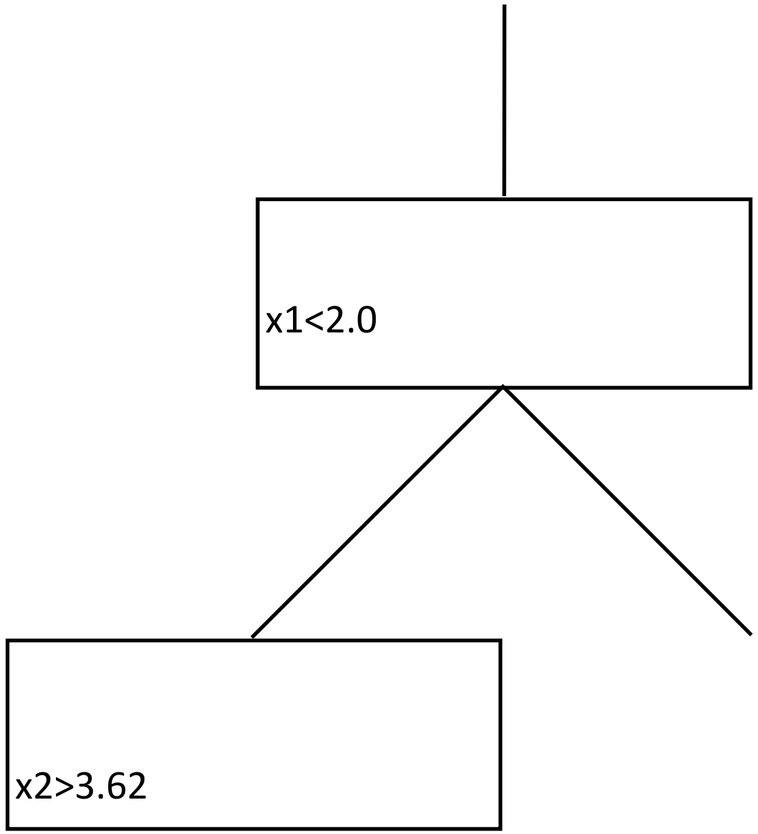


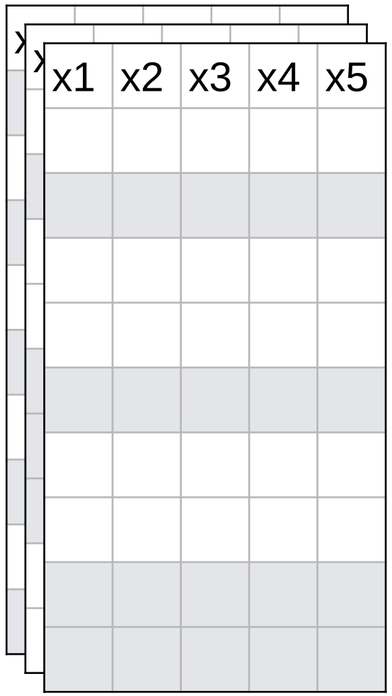
Decide the split
based on the features seen



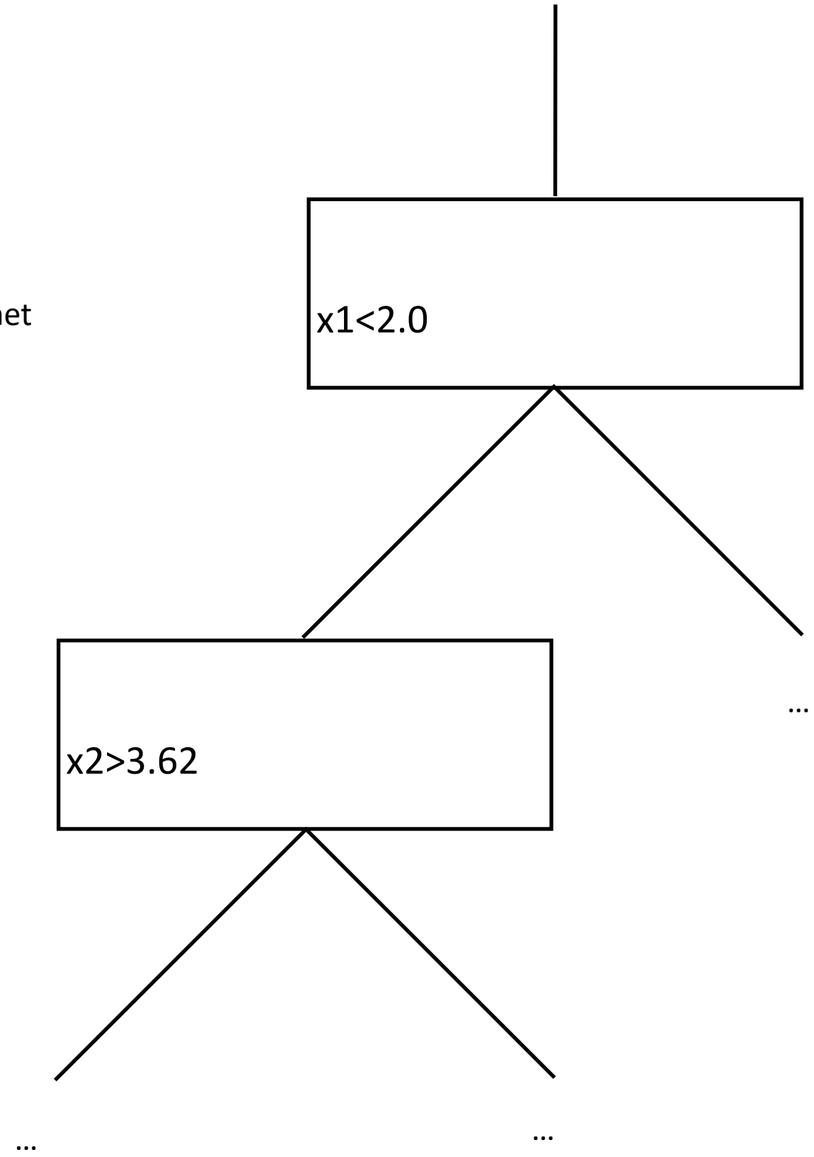


Recursively split nodes





Keep going until
stopping criterion met



Calculating the output

Now, take an element $\mathrm{x}_{\mathrm{new}} = (x_1, x_2, x_3, \ldots, x_n)$.

To get an estimate for the case:

1. Take the bag of estimators $\{t_j(x_{new})\}_{j=1}^{B}$, with B the total number of trees you just calculated.
2. For each tree t_j, calculate its estimate $\mathrm{y}_j = t_j(x_{new})$.
3. The final output (prediction!) is the average of all trees

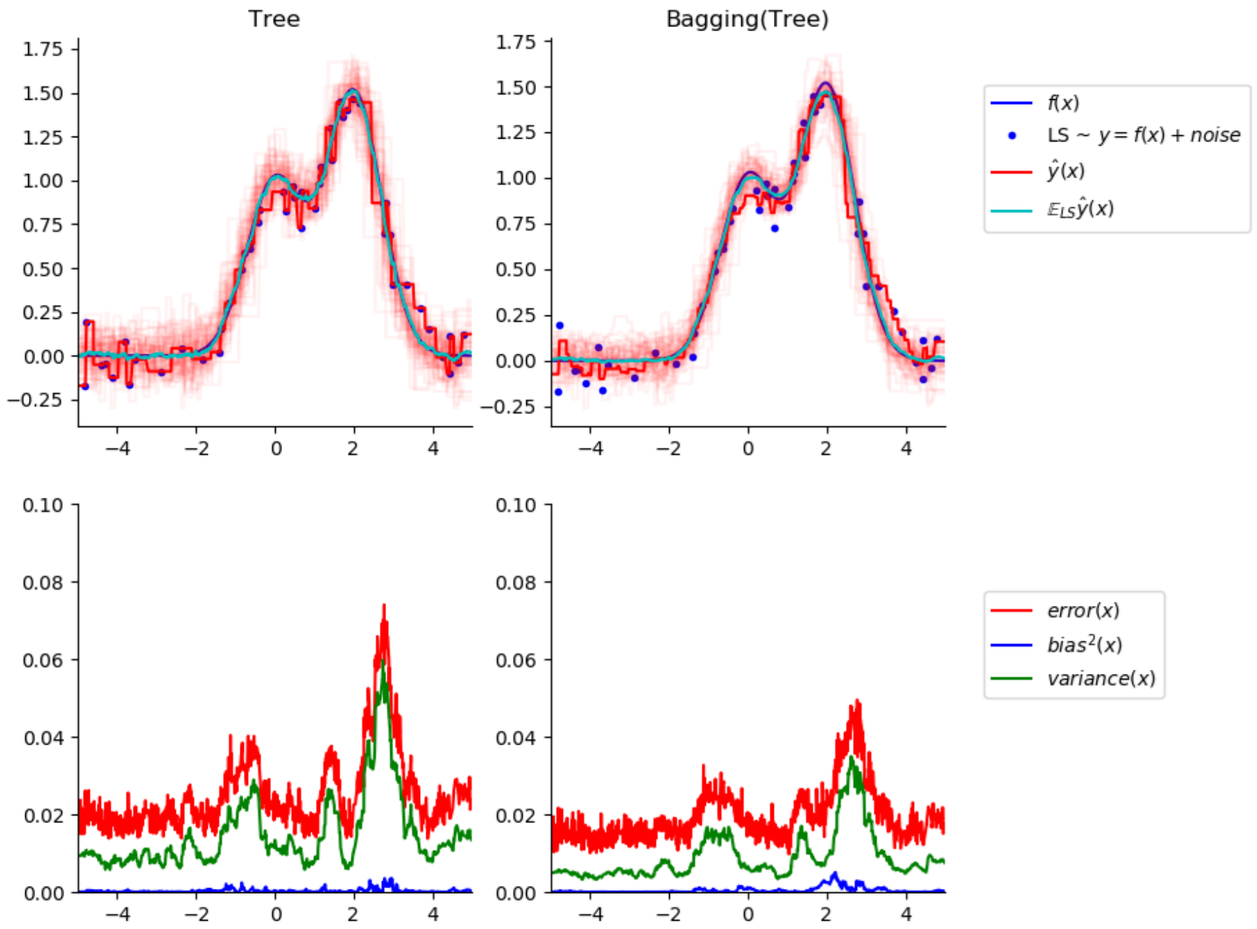
$$y_{new} = \sum_{j}^{B} \frac{t_j(x_{new})}{B}$$

Why Does This Work?

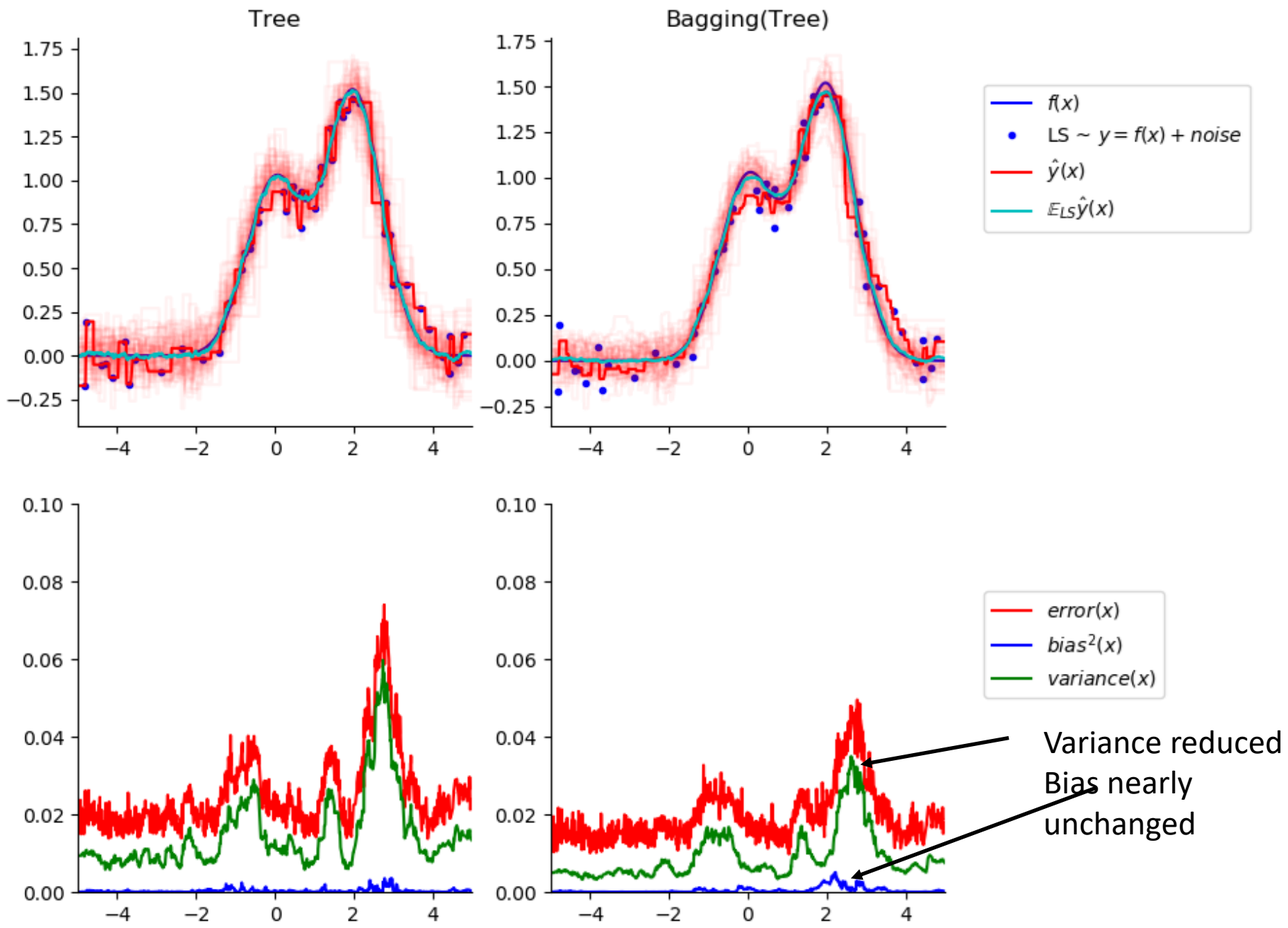
When you average random variables each with variance σ^2 ,then the variance of the average looks like σ^2/B

As $B \rightarrow \infty$, variance of the average becomes smaller.

Variance of Bagging



Variance of Bagging



Bagging For Classification

Suppose problem is to predict one of K classes

Trees in the bag can vote for which class

Result is a vector of proportions [p1, p2, p3, …] where pi is the proportion of the B trees voting for class i

Important: You might be tempted to treat these as probabilities. They are not.

If bagged classifier has smooth decision function, just average the decision function values. Usually leads to lower variance

Bagging For Classification

There is an elegant mathematical argument that bagging will reduce MSE in regression (pg. 285)

In classification, not always true; bagging good classifier will can make it better, while bagging a bad one can make it worse.

Furthermore, interpretation is lost as bagged trees are not trees anymore.

Summary of Bagging

Reduce variance by adding adding more trees and bootstrapping

Result is B trees, each grown on a bootstrapped version of the training data

Predictions from each of the B trees is then averaged and used as a final prediction

But, still a problem…

Correlation

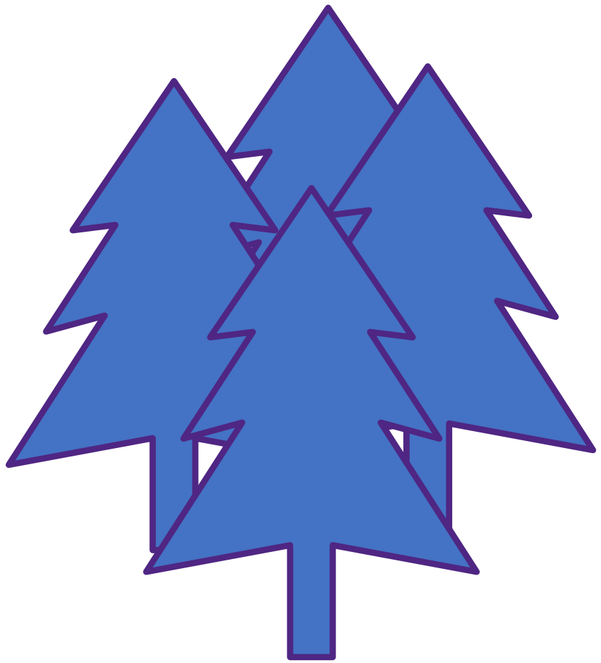
Suppose that we fit a bagged estimator onto data with lots of features

Suppose further that just a handful of those features are important

All B trees will make splits based on that handful of features

All B trees are *correlated,* thus benefits of averaging are limited, and variance of average is

$$\boldsymbol{\rho\sigma^2 + (1-\rho)\sigma^2/B}$$



Random Forest

Trees, Bagging & Correlation

Trees are a highly flexible model, but suffer from high variance

Bagging reduces this variance by averaging over trees trained on bootstrapped samples of our training data

As we've mentioned, bagged trees can look very similar (i.e. they are correlated)

This correlation limits the benefit of averaging

A Wacky Idea

What if I made a bagged estimator, but…

Instead of showing the estimator all the features, I only show it **a random subset?**

What if this **subset changed each time** the tree had to make a split?

I'd have a collection of trees (one might say a forest), which have been grown…randomly…

Random Forest

Bootstrap B datasets.

Grow B trees

- At each split, allow the tree to decide where to split based on a random sample of the features

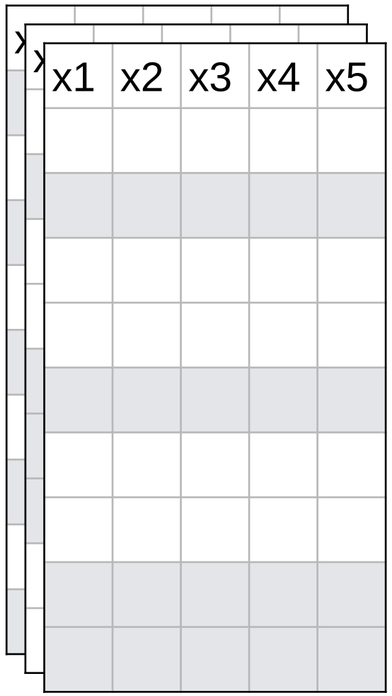
Average predictions from the B trees

The main idea is to reduce variance in bagging by reducing correlation of the trees. Achieved by showing subset of features.

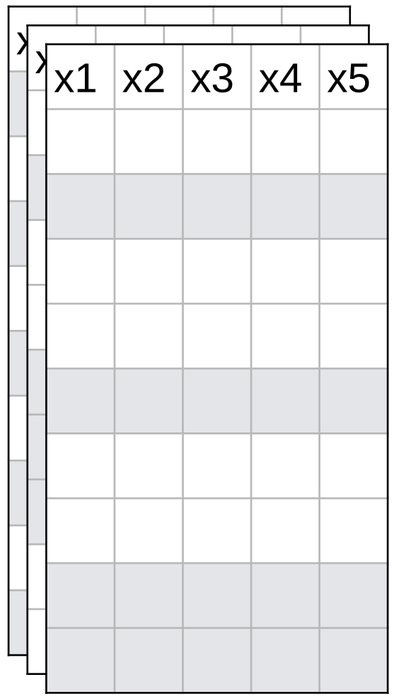
Let’s walk through how one tree might be grown

x1	x2	x3	x4	x5

Take our
training data

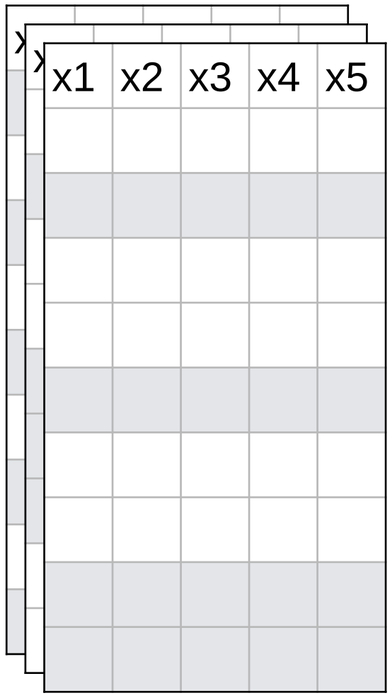


Bootstrap
B training datasets

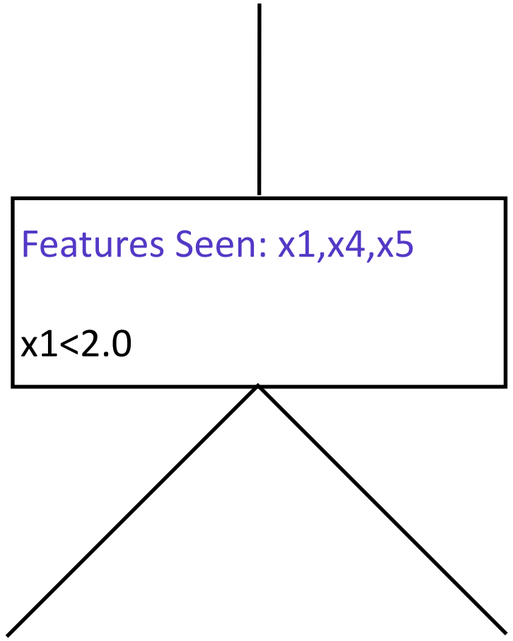


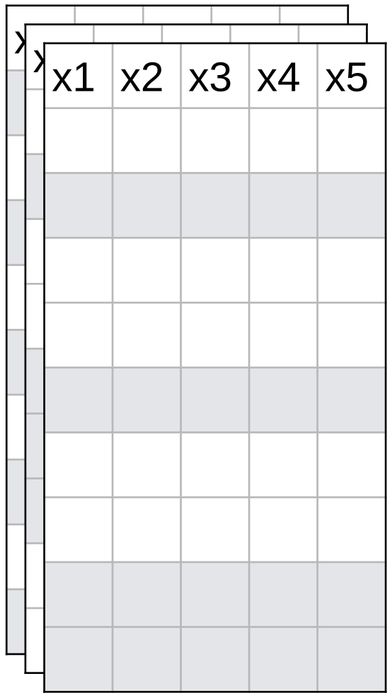
For the first split,
Show the tree a random
Subset of the features
(and optionally the samples)

Features Seen: x1,x4,x5

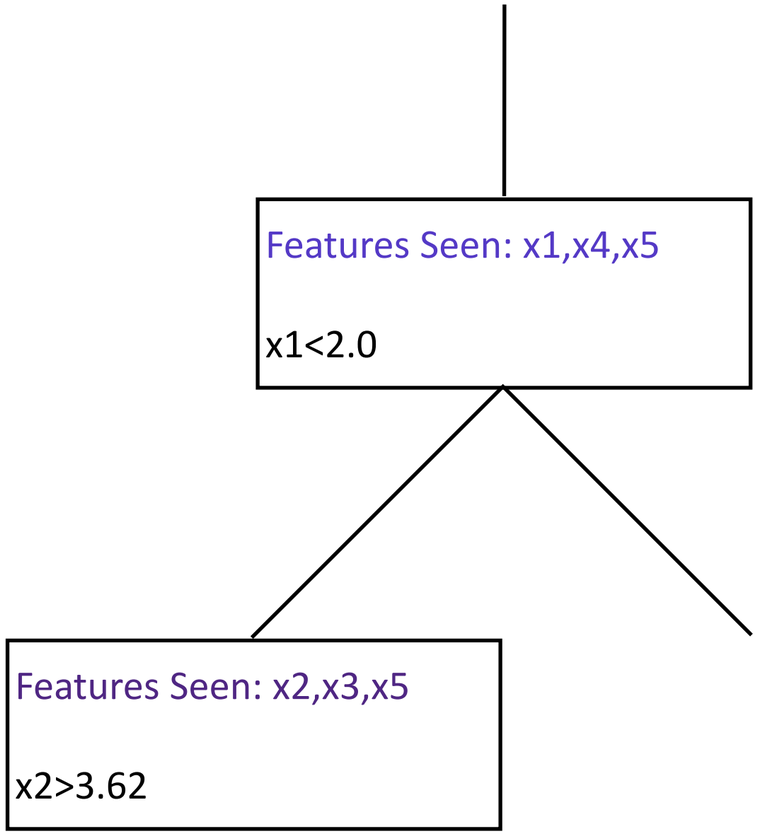


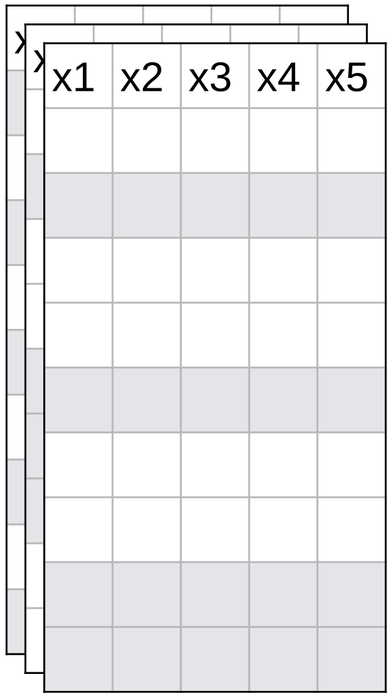
Decide the split
based on the features seen



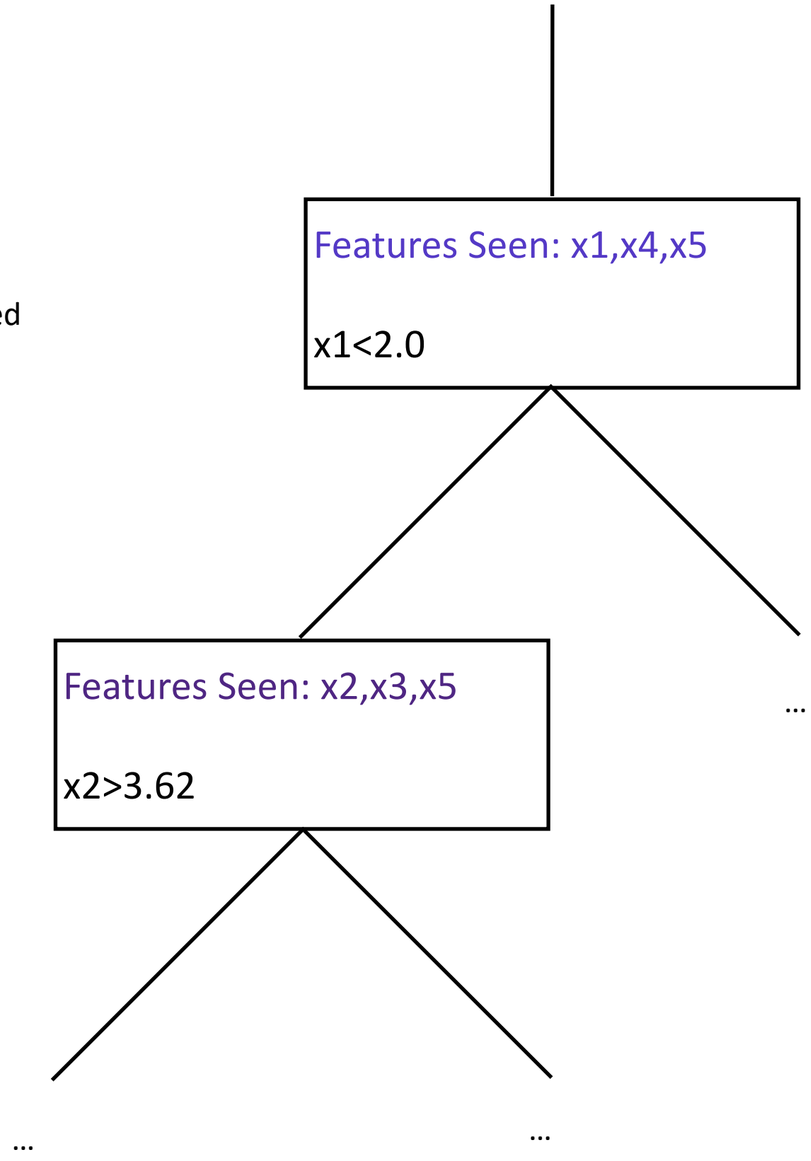


Recursively split nodes, showing a random sample of the features in daughter nodes





Keep going until
Max depth reached

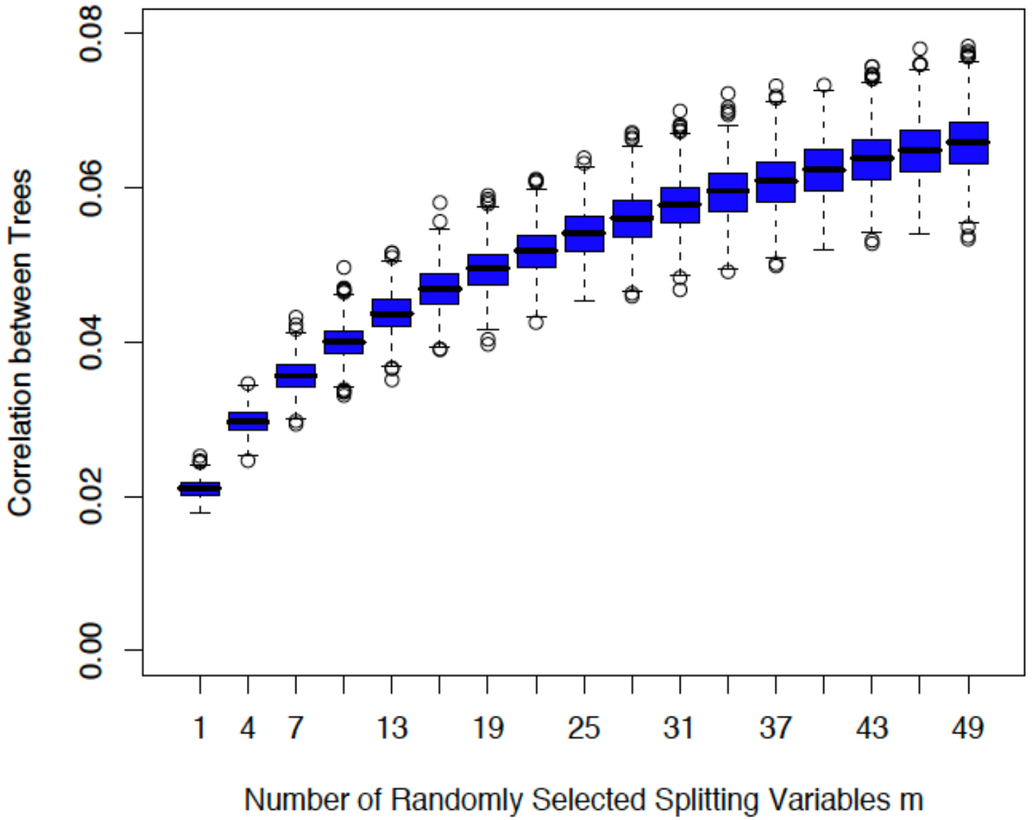


You can predict just like before: Take the average of the predictions (or the vote for each tree).

Why Does RF Further Reduce Correlation

Simply, pairs of tree predictions tend to be less correlated if they do not use the same splitting variables.

If we show each tree fewer variables, then there is less overlap in the features used to split on between trees.



Applied Preference

Random forests are popular among data scientists because:

- They are easily parallelized
- Low bias, and lower variance than something like a tree or bagging
- We can examine how important a feature was to predicting the data

Analogy: Group of experts getting to an agreement.

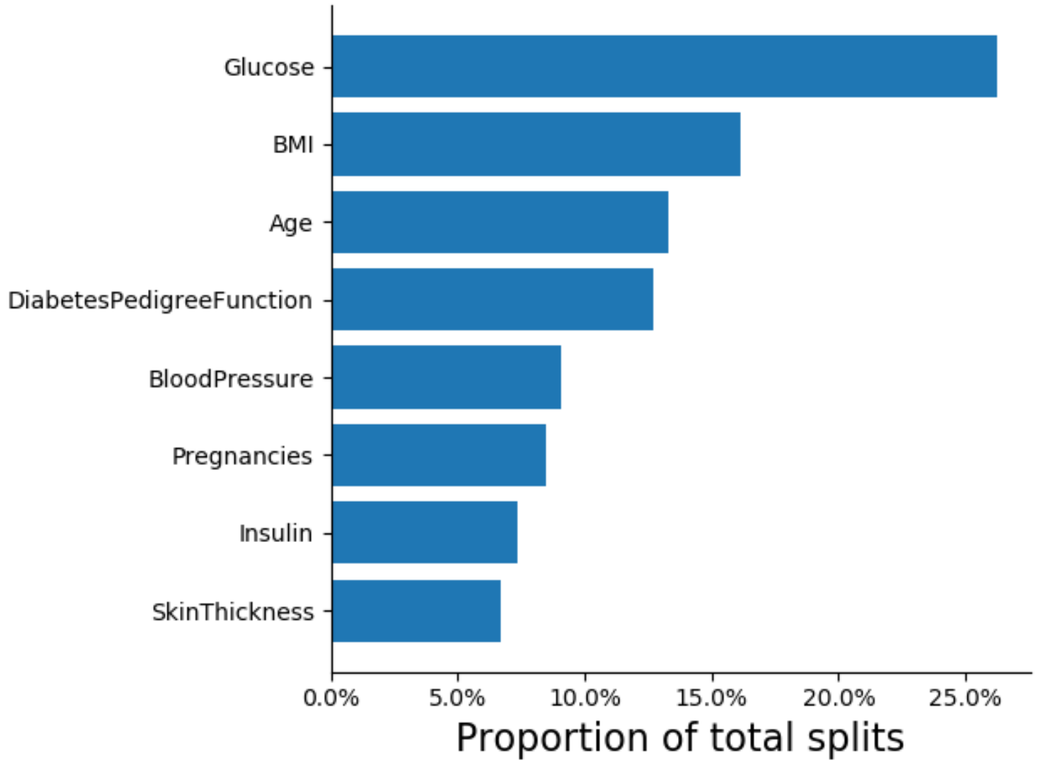
Feature Importance

The improvement in the split criterion is the importance measure attributed to the feature.

Aggregate the improvement across all features. Once model is trained, make a plot like this

More sophisticated approaches can calculate the average loss reduction after adding an element.

You can even do this with profit or costs.



Out of Bag Error

Not every observation (x,y) appears in each bootstrapped dataset.

If we use the trees that did not see (x,y) to make predictions on x, we can essentially validate how well our model is performing.

Call this the “Out of Bag” (OOB) error.

An OOB error estimate is almost identical to that obtained by N-fold cross- validation. Unlike many other nonlinear estimators, random forests can be fit in one sequence, with cross-validation being per- formed along the way.

Once the OOB error stabilizes, the training can be terminated.

Can a RF Overfit?

Yes (although in a perfect world it should not)

The average of fully grown trees can result in too rich a model, and incur unnecessary variance

Small gains in performance by controlling the depths of the individual trees grown in random forests.

Using full-grown trees seldom costs much, and results in one less tuning parameter.

Summary of Random Forests

Bagged estimator, but only split on a random subsample of the features

Random subsample of features decorrelates trees

Can measure feature importance

Can validate as model is being trained through OOB error

Stochastic Gradient Boosting

Boosting

Remember that going too deep in trees leads to overfitting

However, not going far enough means we can't learn complex structure

Boosting applies a weak learner (usually a tree with one split, called a *stump*) in a very neat way in order to learn complex structure

Let's take a look at AdaBoost.M1

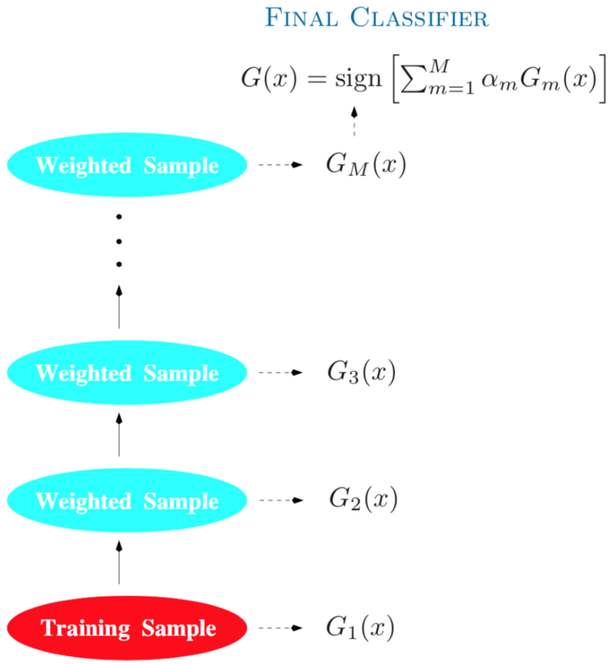
AdaBoost.M1

Fit a weak learner to the sample

Reweight the observations

Fit a weak learner to the reweighed sample

Weight the predictions of the learners



AdaBoost.M1

Algorithm 10.1 *AdaBoost.M1.*

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M: M weak classifiers
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

They weight the contribution of each respective classifier. Their effect is to give higher influence to the more accurate classifiers in the sequence.

Weighted majority vote to produce the final prediction

Boosting

Back in 2016, nearly every kaggle competition winner used some form of boosting (esp. Gradient Boosting).

It works great for smallish sample sizes compared to Random Forests. It can be flexible and accurate with little work.

Libraries like xgboost and lightgbm have *fast* implementations and APIs very similar to sklearn; no need to learn anything new!

Let's take a look at why boosting can be so accurate

Boosting Fits an Additive Model

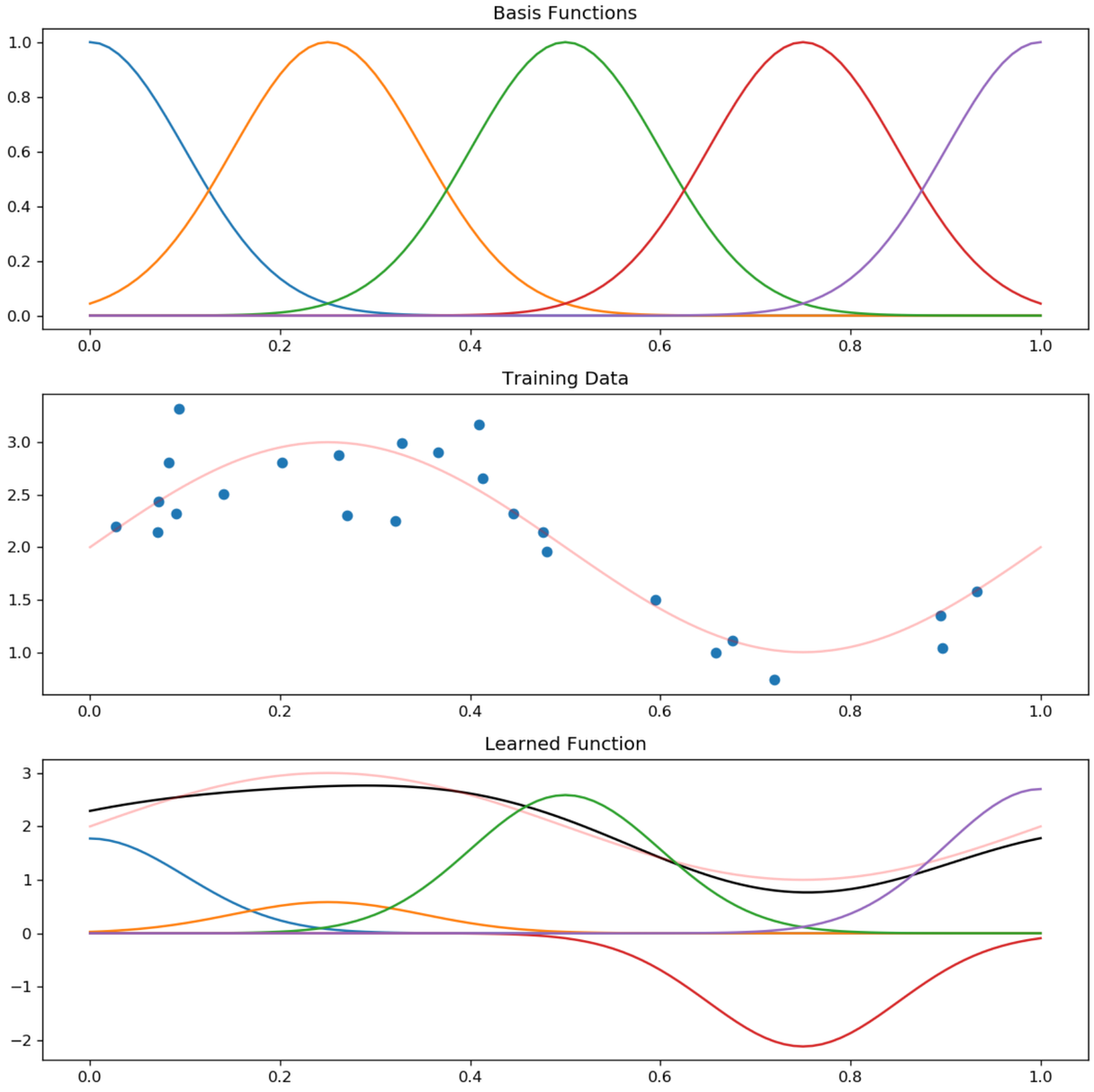
An additive model is one which is expressed as

$$g(x) = \beta_0 + \sum_i f_i(\mathbf{x})$$

Basis functions are simple functions which can be combined together to fit complex structure.

You are already familiar with the radial basis function.

Taking linear combinations of basis functions allows us to get flexible fits


Basis Functions
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
Training Data
3.0
2.5
2.0
1.5
1.0
0.0
0.2
0.4
0.6
0.8
1.0
Learned Function
3
2
1
0
−1
−2
0.0
0.2
0.4
0.6
0.8
1.0

Boosting Fits an Additive Model

In boosting, the basis functions are the weak learners

The basis function expansion looks like

$$f(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma_m),$$

To fit this model, we need to solve

$$\min_{\{\beta_m, \gamma_m\}_1^M} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b(x_i; \gamma_m)\right).$$

Forward Stagewise Additive Modelling

Minimizing the loss can be computationally intensive, so we use “forward stagewise modelling”.

Approximate the solution by sequentially adding new basis functions to the expansion *without adjusting parameters and coefficients of basis functions already fit.*

Algorithm 10.2 *Forward Stagewise Additive Modeling.*

1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to M:
   (a) Compute

$$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$

   (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.

Forward Stagewise Additive Modelling

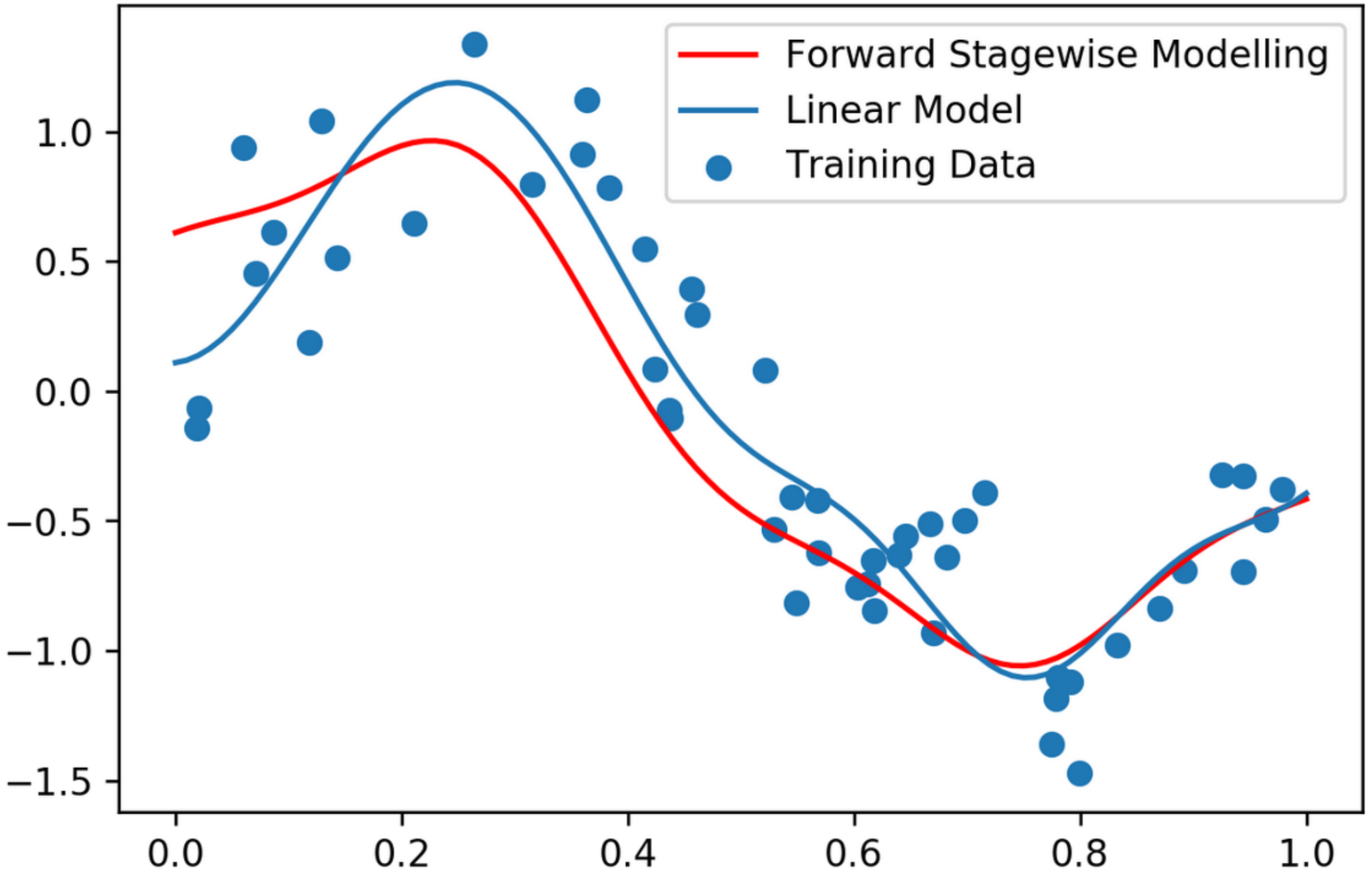
When using squared error loss, some nice math happens

$$L(y, f(x)) = (y - f(x))^2,$$

$$\begin{aligned} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)) &= (y_i - f_{m-1}(x_i) - \beta b(x_i; \gamma))^2 \\ &= (r_{im} - \beta b(x_i; \gamma))^2, \end{aligned}$$

So for squared error, we predict with our existing model, get the residuals, and then fit the a tree to the residuals. Then, that tree becomes part of our model

Not generally the procedure when loss is different from squared error


Forward Stagewise Modelling
Linear Model
Training Data
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0

Boosted Trees

For classification setting with exponential loss, forward stagewise additive modelling yields the adaboost algorithm

For regression using squared error loss, the next tree we add is the tree which best fits our residuals (we will see this is also the case with gradient boosting).

Forward stagewise modelling is a greedy processes, so solutions we obtain are a greedy approximation to the true minimizer.

Analogy: Sequential learning of non-experts!

Boosting Summary

Deep trees can overfit, but stumps have high bias.

If we fit an additive model through Forward Stagewise Additive Modelling, we can ensemble weak learners to make a flexible learner

If we are using MSE as our loss, Forward Stagewise Additive Modelling boils down to fitting a tree to the residuals at each step.

Gradient Descent

Gradient descent seeks a local optima by performing

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha \nabla f(\mathbf{x}_n) = (\mathbf{x}_{n-1} - \alpha \nabla f(\mathbf{x}_{n-1})) - \alpha \nabla f(\mathbf{x}_n) = \mathbf{x_0} - \sum_i \alpha \nabla f(\mathbf{x}_i)$$

Here, f is a loss function, alpha is the learning rate

Gradient descent is a “greedy” strategy, as it moves in the direction of steepest descent given its current position. Other movements could lead to smaller loss, but we don’t take those.

Boosting is Like Gradient Descent

Forward stagewise additive modelling is also very greedy. At each step, the solution tree is the one that maximally reduces the loss.

Thus, tree predictions in each boosting iterations are kind of like the components of the gradient; they move is in the direction where loss is reduced maximally.

Gradient boosting fits a tree to the negative gradient values of the loss using least squares and a weak learner (e.g. a stump).

When we use squared error loss, the gradients are the residuals (remember lecture 1?)

Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute

$$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$

   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute

$$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$

   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

Example of Gradient Boosting Using Squared Error

Height (m)	Favorite Color	Gender	Weight (kg)
1.6	Blue	Male	88
1.6	Green	Female	76
1.5	Blue	Female	56
1.8	Red	Male	73
1.5	Green	Male	77
1.4	Blue	Female	57

Example of Gradient Boosting Using Squared Error

Height (m)	Favorite Color	Gender	Weight (kg)
1.6	Blue	Male	88
1.6	Green	Female	76
1.5	Blue	Female	56
1.8	Red	Male	73
1.5	Green	Male	77
1.4	Blue	Female	57

Average Weight

71.2

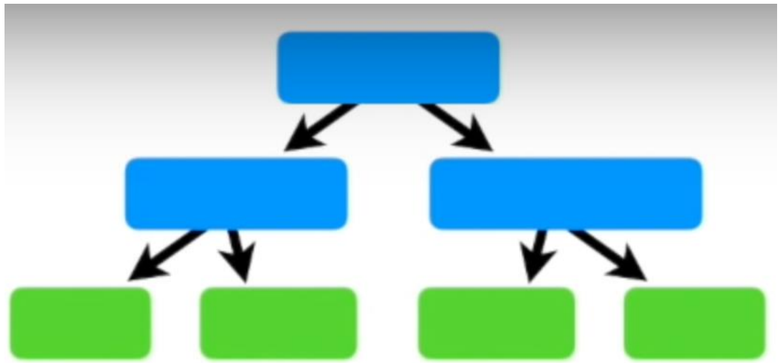
Example of Gradient Boosting Using Squared Error

Height (m)	Favorite Color	Gender	Weight (kg)	Residual
1.6	Blue	Male	88	16.8
1.6	Green	Female	76	4.8
1.5	Blue	Female	56	-15.2
1.8	Red	Male	73	1.8
1.5	Green	Male	77	5.8
1.4	Blue	Female	57	-14.2

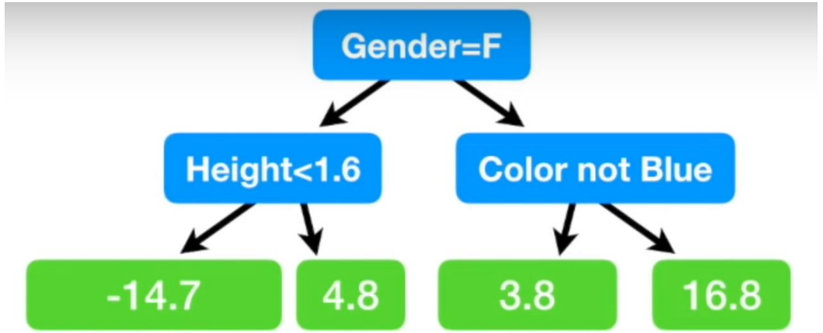
Average Weight

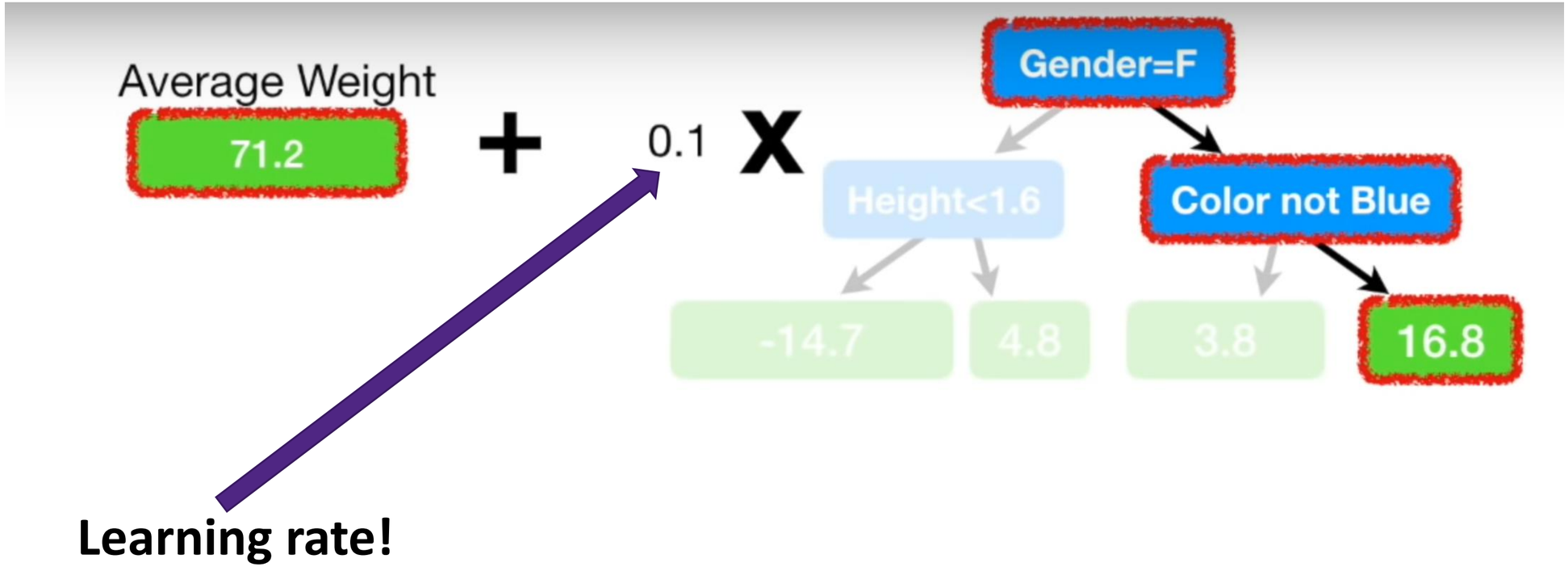
71.2

Height (m)	Favorite Color	Gender	Weight (kg)	Residual
1.6	Blue	Male	88	16.8
1.6	Green	Female	76	4.8
1.5	Blue	Female	56	-15.2
1.8	Red	Male	73	1.8
1.5	Green	Male	77	5.8
1.4	Blue	Female	57	-14.2

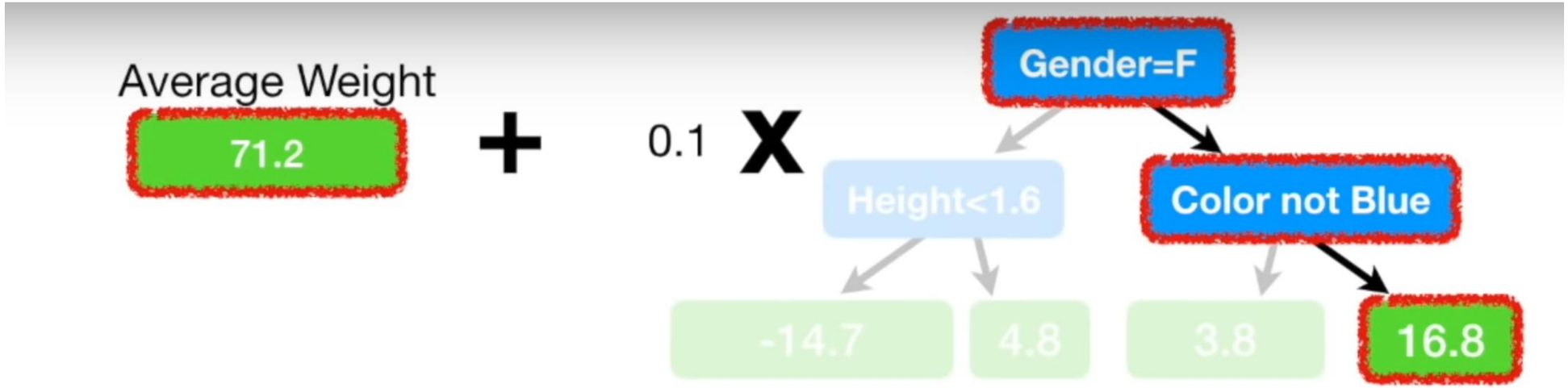


Height (m)	Favorite Color	Gender	Weight (kg)	Residual
1.6	Blue	Male	88	16.8
1.6	Green	Female	76	4.8
1.5	Blue	Female	56	-15.2
1.8	Red	Male	73	1.8
1.5	Green	Male	77	5.8
1.4	Blue	Female	57	-14.2





Learning rate!



Now the **Predicted Weight** = 71.2 + (0.1 × 16.8) = 72.9

Gradient Boosting Summary

Boosting fits weak learners through Forward Stage Additive Modelling

Gradient boosting fits weak learners to the gradient of the loss

This emulates something like gradient descent.

Closing Thoughts

Trees: recursively split the feature space based on reduction in loss/split criteria. They are high variance which means high error.

Bagging, Boosting, and Random Forests are all ways to deal with this high variance while retaining high flexibility.

If you are interested in more details, see chapters 9, 10, 15 of *Elements of Statistical Learning*

It should be your go-to model if you are only working with structured data.

Now let’s continue with a tutorial!