Machine Learning II

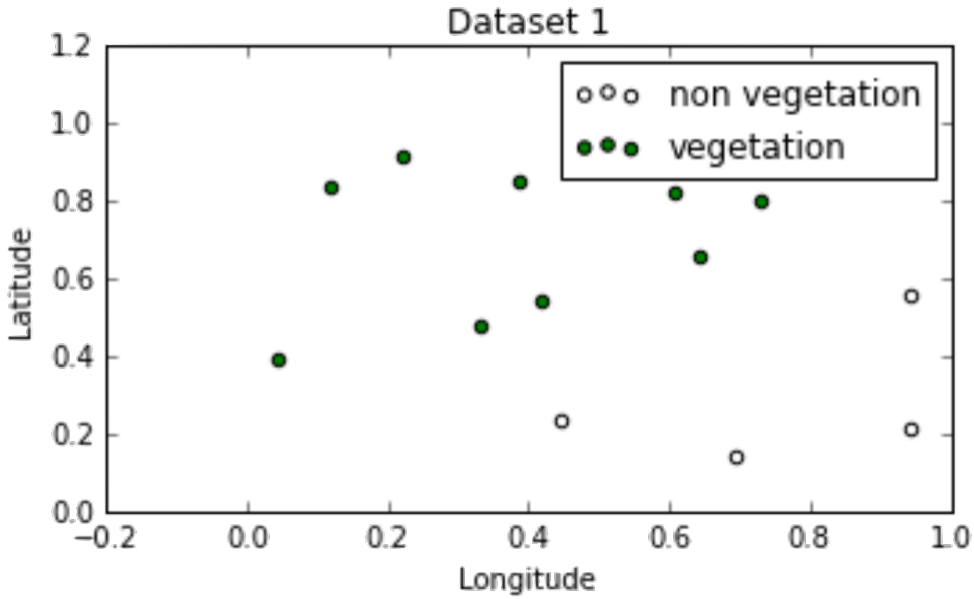
JSC 370: Data Science II

Outline

- Motivation
- Decision Trees
- Random Forests
- Bagging
- Boosting
- Gradient Boosting, XGBoost

Geometry of Data

Logistic regression for classification works best when the classes are well-separated in the feature space

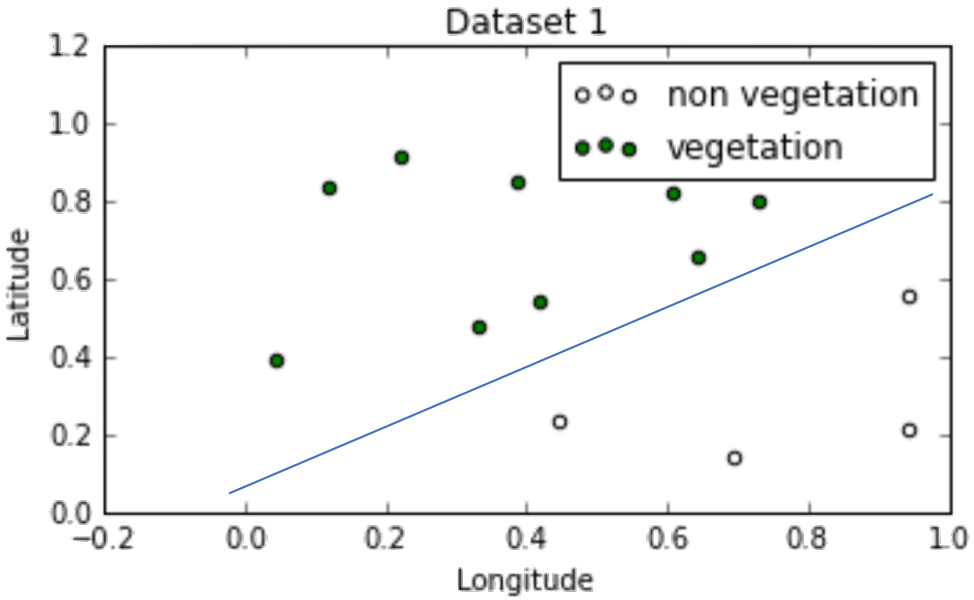


Geometry of Data

- **The decision boundary** is defined where the probability of being in class 1 and class 0 are equal, i.e.
  - $P(Y = 1) = 1 - P(Y = 1) \quad \Rightarrow. \quad P(Y = 1) = 0.5,$
- Which is equivalent to when the `log-odds=0`:
  - $\boldsymbol{x}\beta = 0,$
- this equation defines a line or a hyperplane.

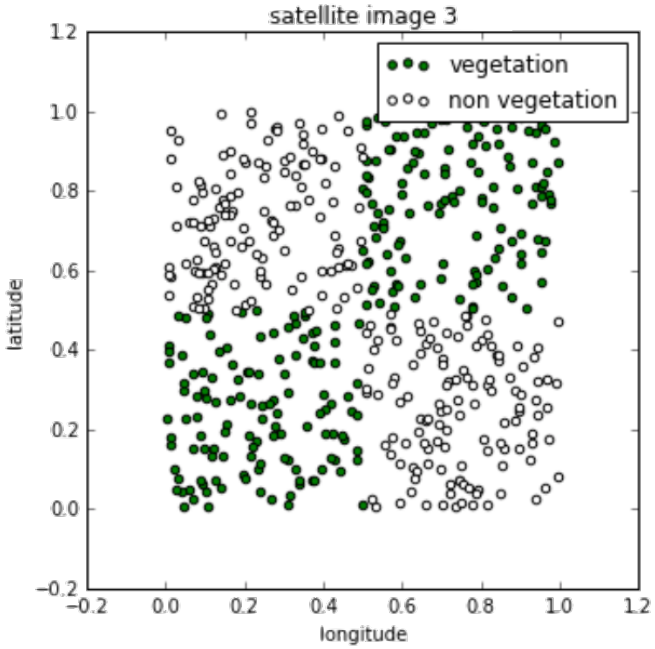
Geometry of Data

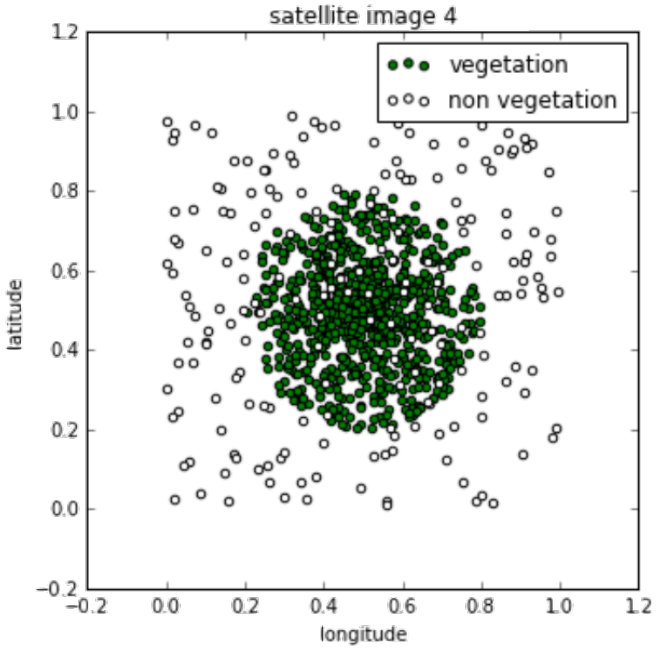
- **Question:** Can you guess the equation that defines the decision boundary below?
- $-0.8x_1 + x_2 = 0 \implies x_2 = 0.8x_1 \Rightarrow Latitude = 0.8\, Lon$



Geometry of Data

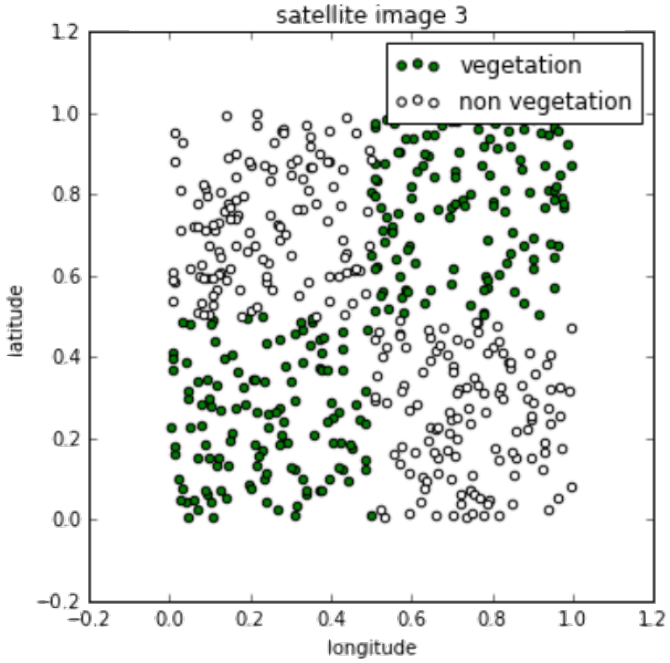
- **Question:** How about these?





Geometry of Data

•Notice that in all of the datasets the classes are still well-separated in the feature space, but ***the decision boundaries cannot be described by single equations***:



Geometry of Data

- While logistic regression models with linear boundaries are intuitive to interpret by examining the impact of each predictor on the log-odds of a positive classification, it is less straightforward to interpret nonlinear decision boundaries in context:

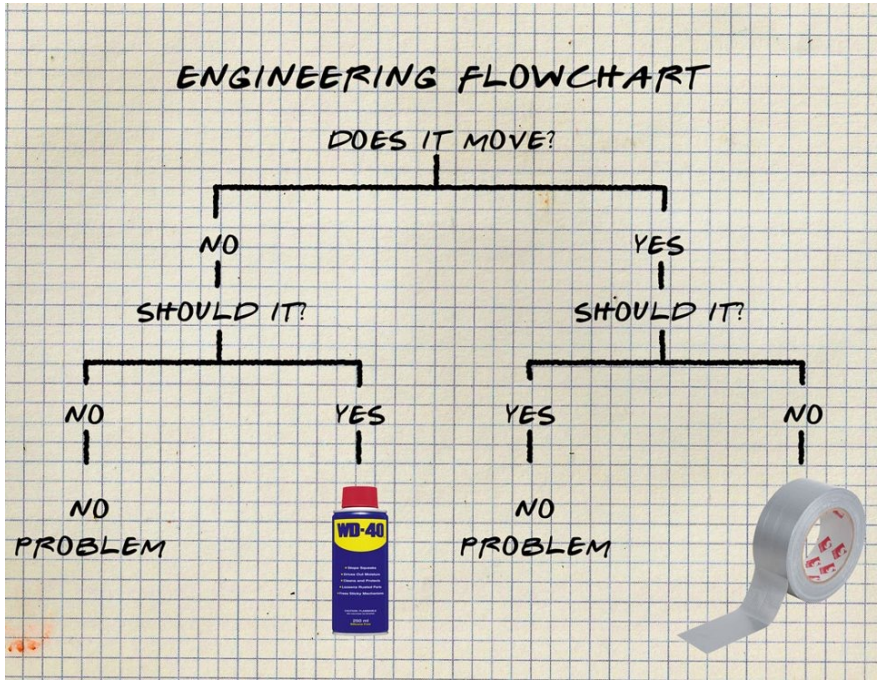
- $(x_3+2x_2) - x_1^2 + 10 = 0$

- It would be desirable to build models that:

1. allow for ***complex decision boundaries***.
2. are also ***easy to interpret***.

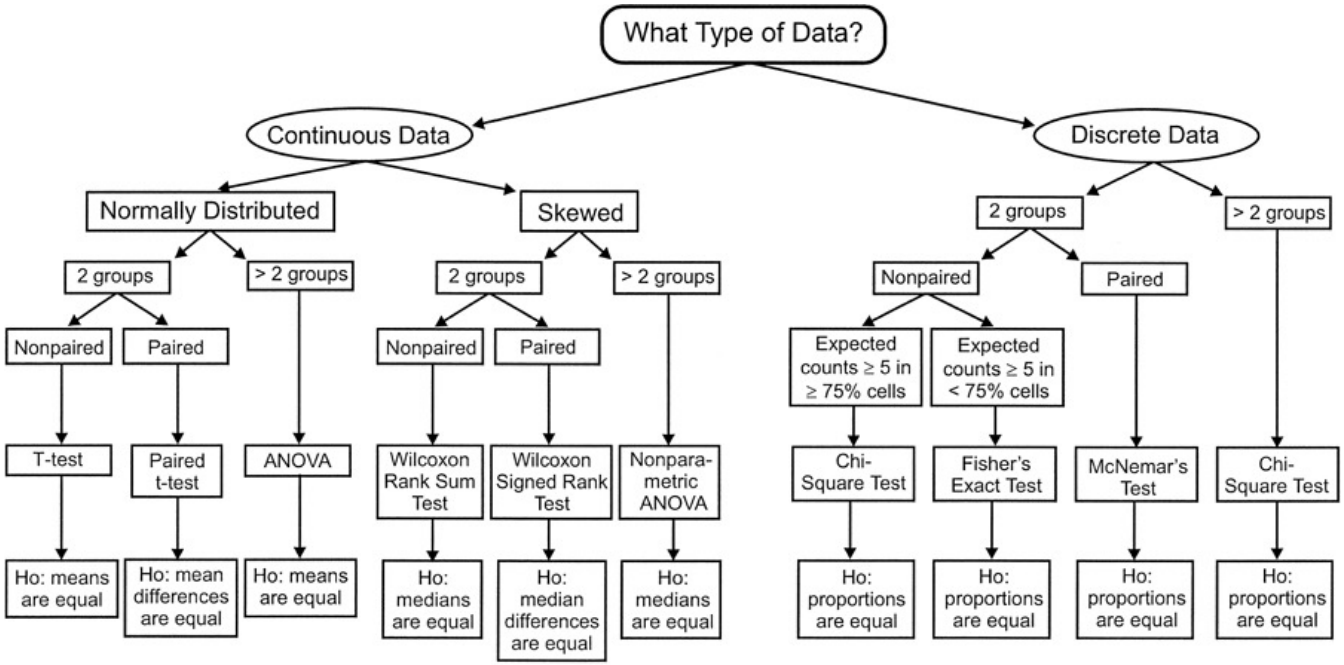
Interpretable Models

- But people in every walk of life have long been using interpretable models for differentiating between classes of objects and phenomena:



Interpretable Models

- But people in every walk of life have long been using interpretable models for differentiating between classes of objects and phenomena:



Source: Waning B, Montagne M: *Pharmacoepidemiology: Principles and Practice*: http://www.accesspharmacy.com

Decision Trees

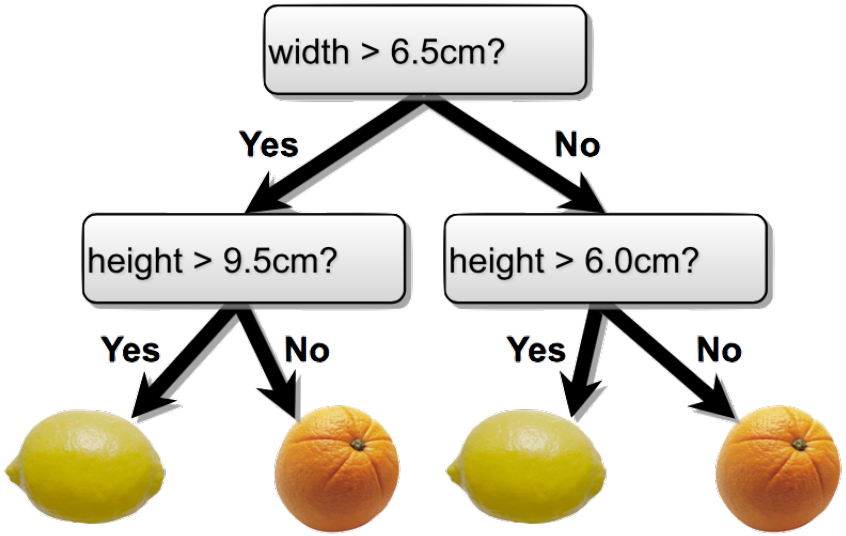
- It turns out that the simple flow charts in our examples can be formulated as mathematical models for classification and these models have the properties we desire; they are:

1. interpretable by humans
2. have sufficiently complex decision boundaries
3. the decision boundaries are locally linear, each component of the decision boundary is simple to describe mathematically.

The Geometry of Flow Charts

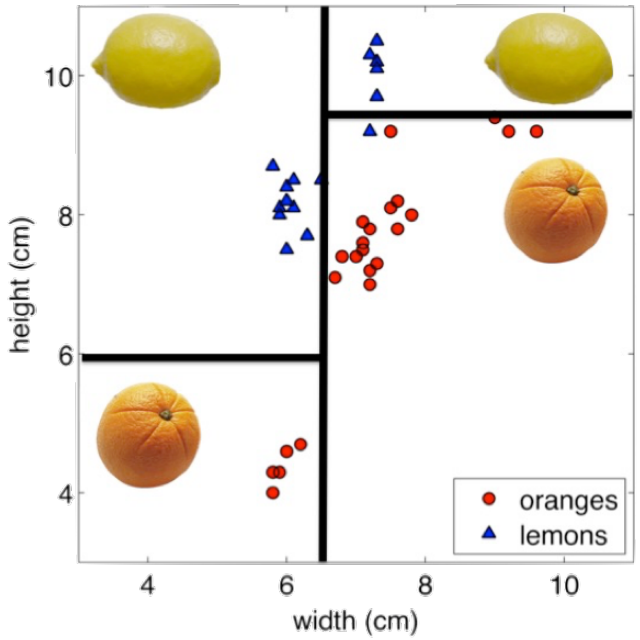
- Flow charts whose graph is a tree (connected and no cycles) represents a model called a ***decision tree***.

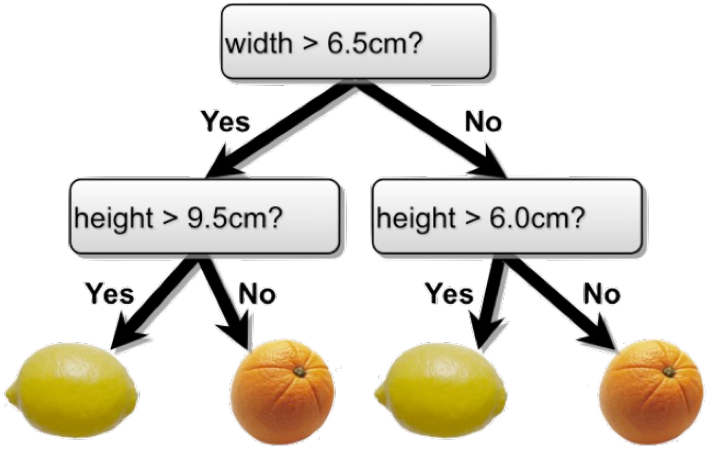
- Formally, a ***decision tree model*** is one in which the final outcome of the model is based on a series of comparisons of the values of predictors against threshold values.



The Geometry of Flow Charts

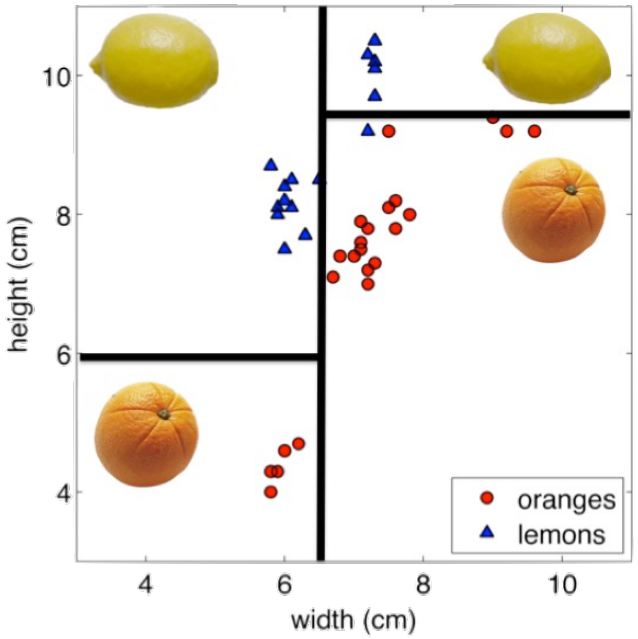
- Every flow chart tree corresponds to a partition of the feature space by *axis aligned lines* or (hyper) planes. Conversely, every such partition can be written as a flow chart tree.

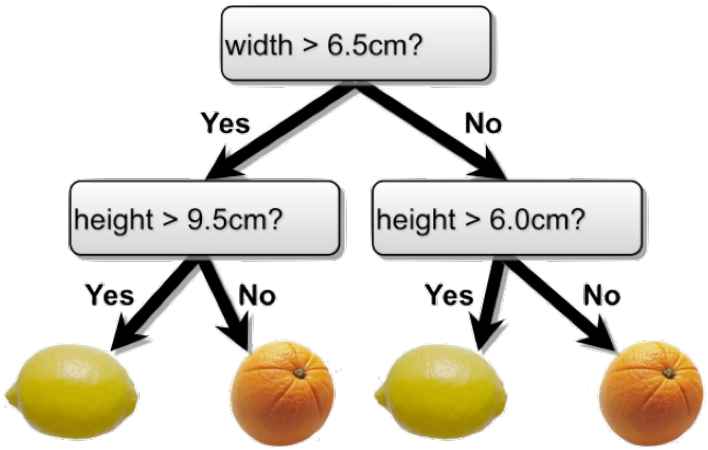




The Geometry of Flow Charts

- Each comparison and branching represents splitting a region in the feature space on a single feature. Typically, at each iteration, we split once along one dimension (one predictor)





Learning the Model

- Learning the smallest ‘optimal’ decision tree for any given set of data is NP complete for numerous simple definitions of ‘optimal’. Instead, we will seek a reasonable model using a *greedy algorithm*.

1. Start with an empty decision tree (undivided feature space)

2. Choose the ‘optimal’ predictor on which to split and choose the ‘optimal’ threshold value for splitting.

3. Recurse on on each new node until ***stopping condition*** is met

- We need to define the splitting criterion and stopping condition.

Numerical vs Categorical Attributes

Example

Supposed the feature we want to split on is **color**, and the values are: Red, Blue and Yellow. If we encode the categories numerically as:

Red = 0, Blue = 1, Yellow = 2

Then the possible non-trivial splits on **color** are

{{Red}, {Blue, Yellow}} {{Red, Blue},{Yellow}}

But if we encode the categories numerically as:

Red = 2, Blue = 0, Yellow = 1

The possible splits are

{{Blue}, {Yellow, Red}} {{Blue,Yellow}, {Red}}

Depending on the encoding, the splits we can optimize over can be different!

Optimality of Splitting

- While there is no ‘correct’ way to define an optimal split, there are some common sensical guidelines for every splitting criterion:
- the regions in the feature space should grow progressively more pure with the number of splits. That is, we should see each region ‘specialize’ towards a single class.
- the fitness metric of a split should take a differentiable form (making optimization possible)
- we shouldn’t end up with empty regions - regions containing no training points.

Gini Index

- Suppose we have J number of predictors, N number of training points and K classes.
- Suppose we select the j-the predictor and split a region containing N number of training points along the threshold $t_j \in \mathbb{R}$.
- We can assess the quality of this split by measuring the purity of each newly created region, R_1, R_2. This metric is called the ***Gini Index***:

$$\text{Gini}(i|j, t_j) = 1 - \sum_k p(k|R_i)^2$$

Gini Index

Example

	Class 1	Class 2	$\text{Gini}(i\|j,t_j)$
R_1	0	6	$1-(6/6^2+0/6^2)=0$
R_2	5	8	$1-[(5/13)^2+(8/13)^2]=80/169$

- We can now try to find the predictor j and the threshold t_j that minimizes the average Gini Index over the two regions, weighted by the population of the regions:

$$\min_{j,t_j}\left\{\frac{N_1}{N}\text{Gini}(1|j,t_j)+\frac{N_2}{N}\text{Gini}(2|j,t_j)\right\}$$

- where N_i is the number of training points inside region R_i.

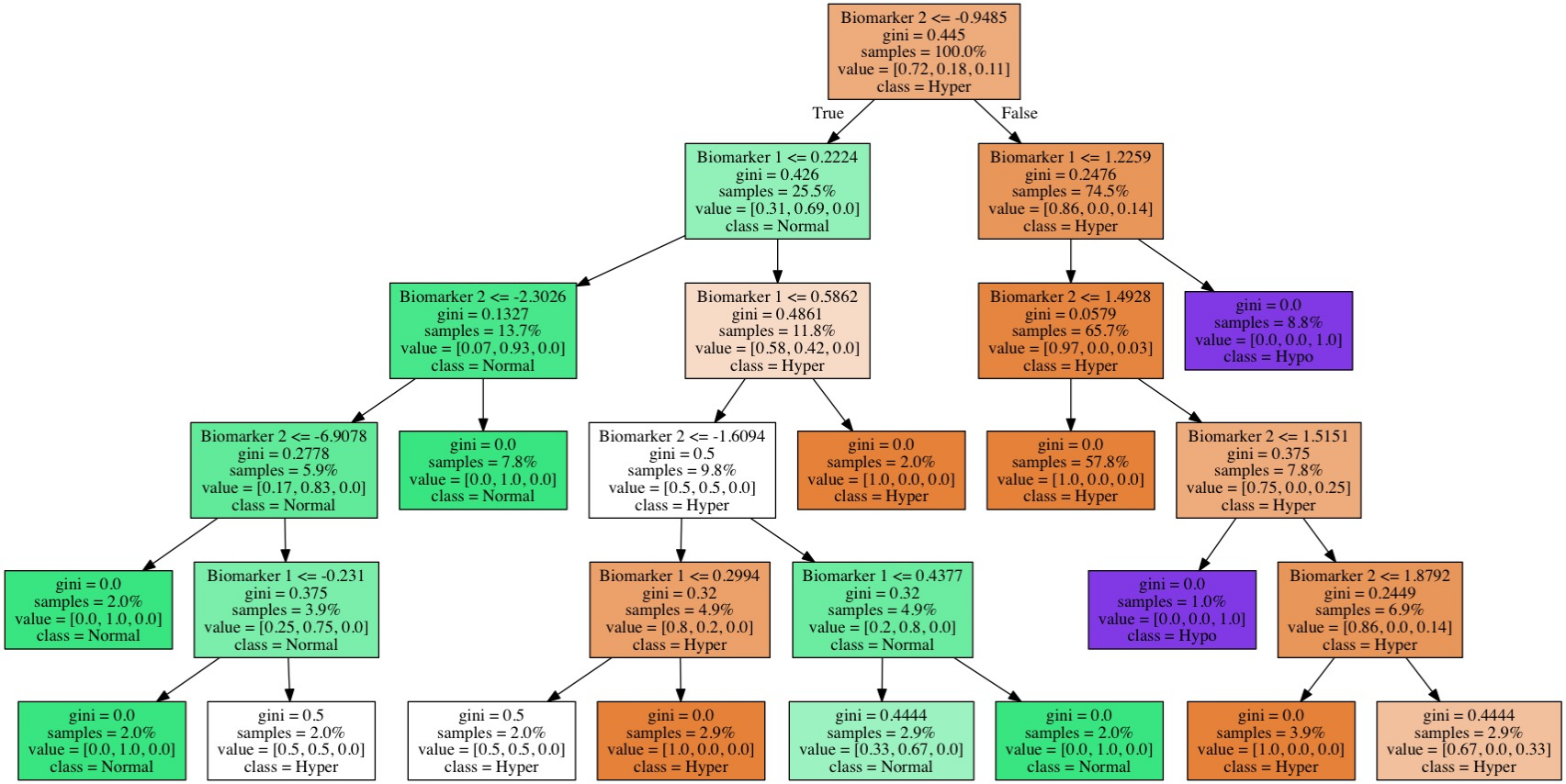
Stopping Conditions

- More restrictive stopping conditions:
- Don't split a region if the class distribution of the training points inside the region are independent of the predictors
- Compute the gain in purity, information or reduction in entropy of splitting a region *R* into R_1 and R_2*:*
  - $Gain(R) = \Delta(R) = m(R) - \frac{N_1}{N} m(R_1) - \frac{N_2}{N} m(R_2)$
- where *m* is a metric like the Gini Index or entropy. Don't split if the gain is less than some pre-defined threshold.

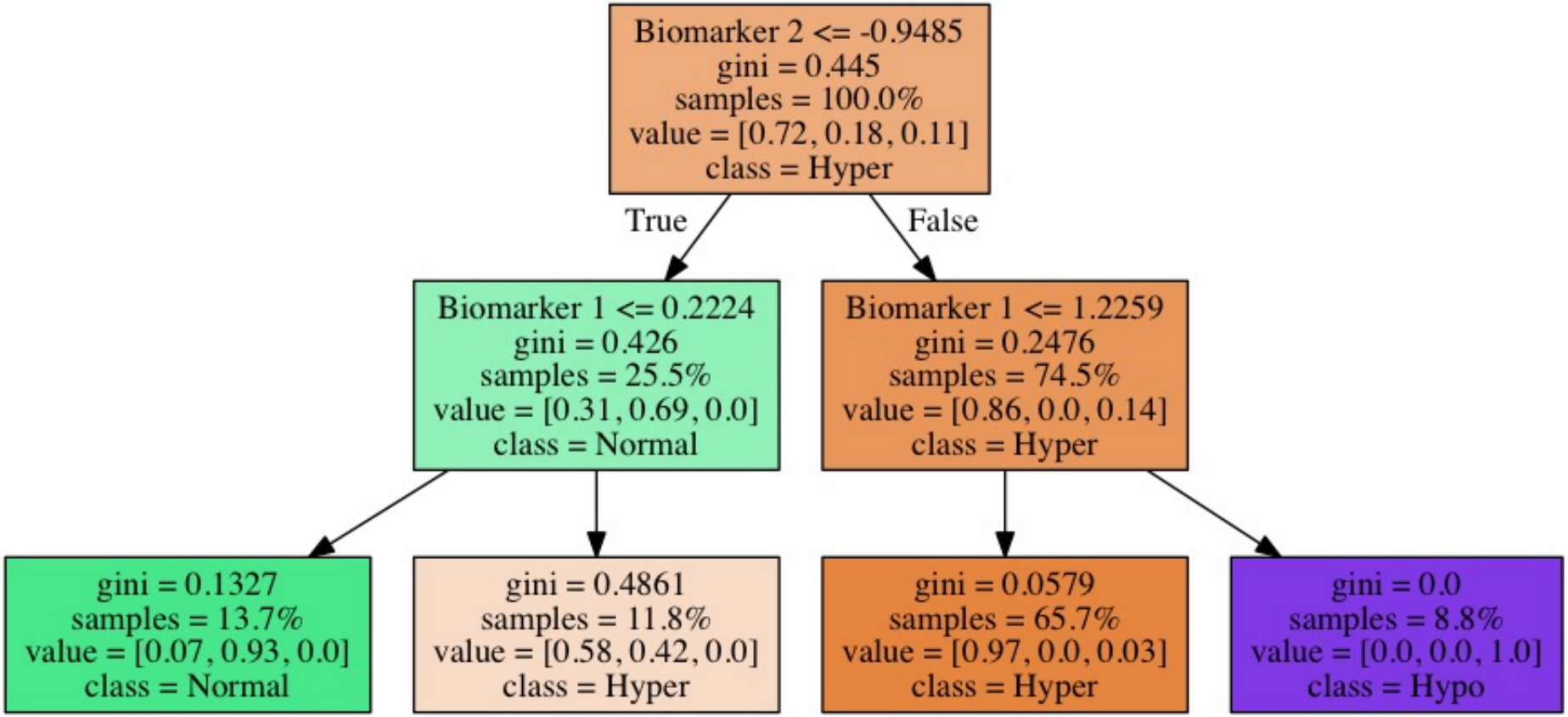
Alternative to Using Stopping Conditions

- What is the major issue with pre-specifying a stopping condition?
    - you may stop too early or stop too late.
- How can we fix this issue?
    - choose several stopping criterion (set minimal Gain(R) at various levels) and cross-validate which is the best.
- What is an alternative approach to this issue?
    - Don’t stop. Instead prune back!

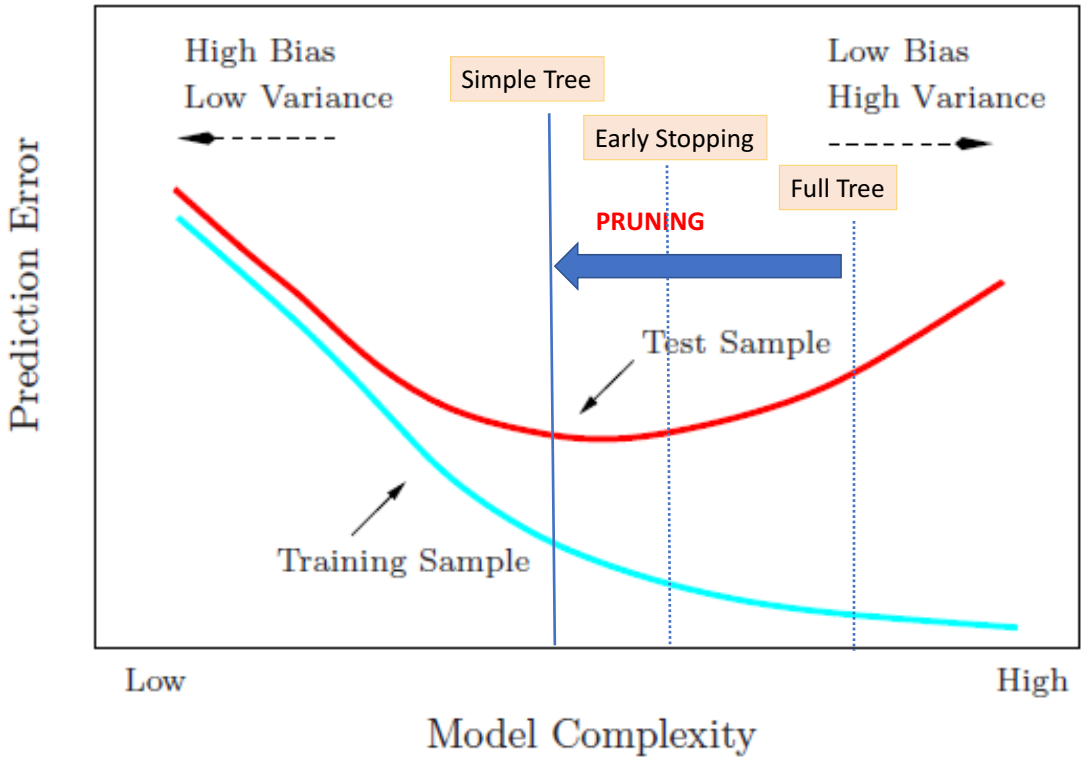
Motivation for Pruning



Motivation for Pruning



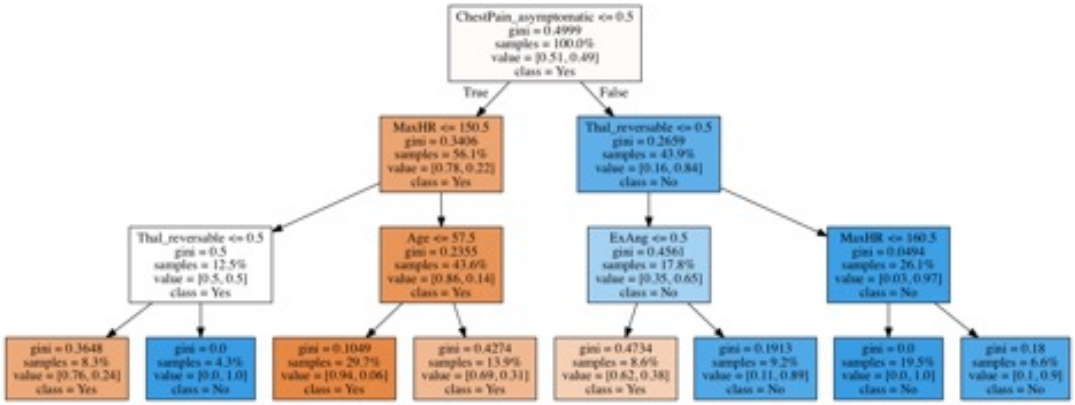
Motivation for Pruning



Pruning

- Rather than preventing a complex tree from growing, we can obtain a simpler tree by ‘pruning’ a complex one.
- There are many method of pruning, a common one is ***cost complexity pruning***, whereby we select from an array of smaller subtrees of the full model that optimizes a balance of performance and efficiency.
- That is, we measure
- $C(T) = Error(T) + \alpha|T|$
- where *T* is a decision (sub) tree, $|T|$ is the number of leaves in the tree and α is the parameter for penalizing model complexity.

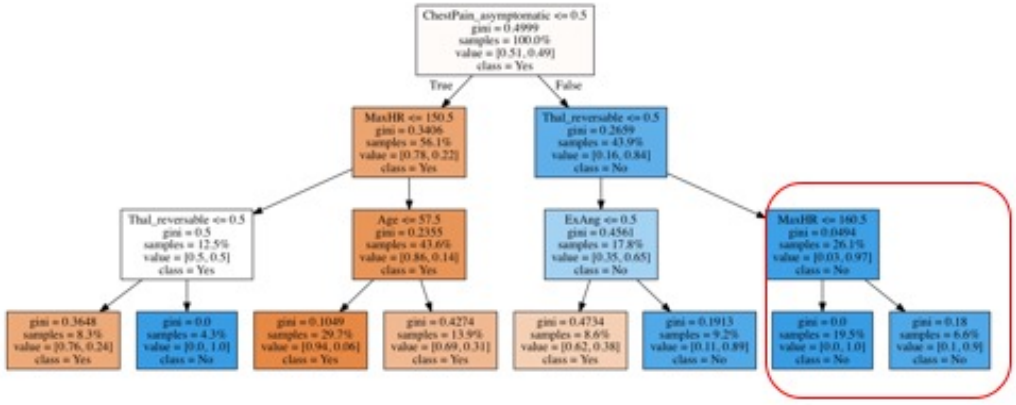
Pruning



$\alpha = 0.2$

Tree	Error	Num Leaves	Total

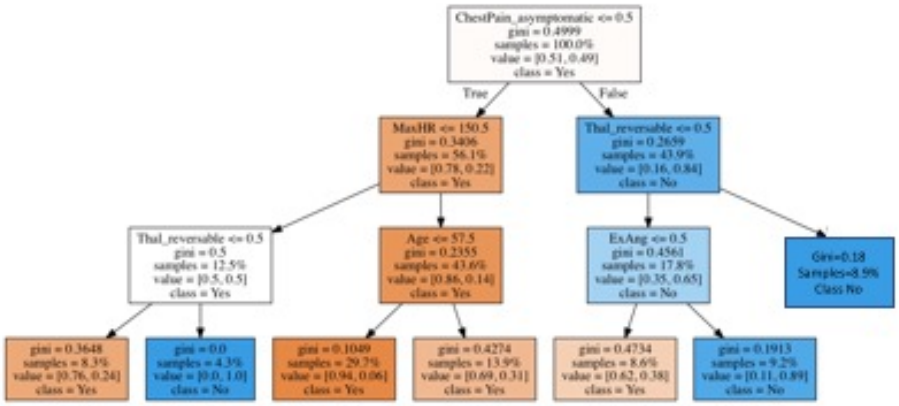
Pruning



$\alpha = 0.2$

Tree	Error	Num Leaves	Total
T	0.32	8	1.92

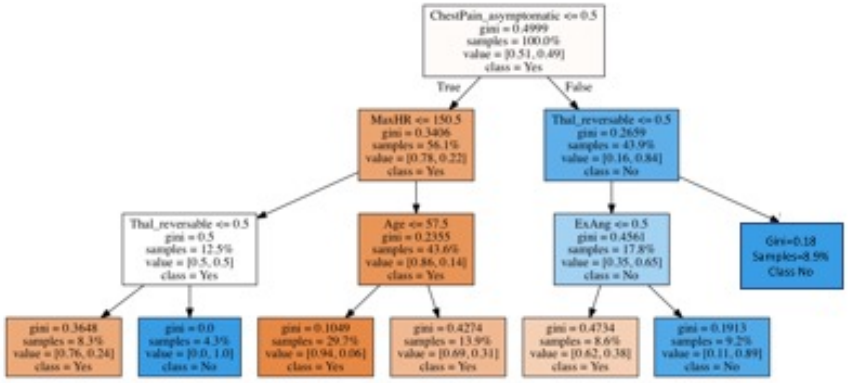
Pruning



$\alpha = 0.2$

Tree	Error	Num Leaves	Total
T	0.32	8	1.92
Tsmall	0.33	7	1.73

Pruning



$\alpha = 0.2$

Tree	Error	Num Leaves	Total
T	0.32	8	1.92
Tsmall	0.33	7	1.73

Smaller tree has larger error but less cost complexity score

Pruning

- $C(T) = Error(T) + \alpha|T|$

1. Fix α.

2. Find best tree for a given α and based on cost complexity *C*.

3. Find best α using CV (what should be the error measure?)

Pruning

- The pruning algorithm:

1. Start with a full tree T_0 (each leaf node is pure)
2. Replace a subtree in T_0 with a leaf node to obtain a pruned tree T_1. This subtree should be selected to minimize

$$\frac{Error(T_0) - Error(T_1)}{|T_0| - |T_1|}$$

3. Iterate this pruning process to obtain $T_0, T_1, \ldots, T_L$ where T_L is the tree containing just the root of T_0
4. Select the optimal tree T_i by cross validation.

- **Note:** you might wonder where we are computing the cost-complexity $C(T_l)$. One can prove that this process is equivalent to explicitly optimizing *C* at each step.

Decision Trees for Regression

Adaptations for Regression

- With just two modifications, we can use a decision tree model for regression:

1. The three splitting criteria we've examined each promoted splits that were pure - new regions increasingly specialized in a single class.

    A. **For classification**, purity of the regions is a good indicator the performance of the model.
    B. **For regression**, we want to select a splitting criterion that promotes splits that improves the predictive accuracy of the model as measured by, say, the MSE.

2. For regression with output in $\mathbb{R}$, we want to label each region in the model with a real number - typically the average of the output values of the training points contained in the region.

Learning Regression Trees

- The learning algorithms for decision trees in regression tasks is:
  1. Start with an empty decision tree (undivided feature space)
  2. Choose a predictor j on which to split and choose a threshold value t_j for splitting such that the weighted average MSE of the new regions as smallest possible:

$$\underset{j,t_j}{\operatorname{argmin}} \left\{ \frac{N_1}{N} \operatorname{MSE}(R_1) + \frac{N_2}{N} \operatorname{MSE}(R_2) \right\}$$

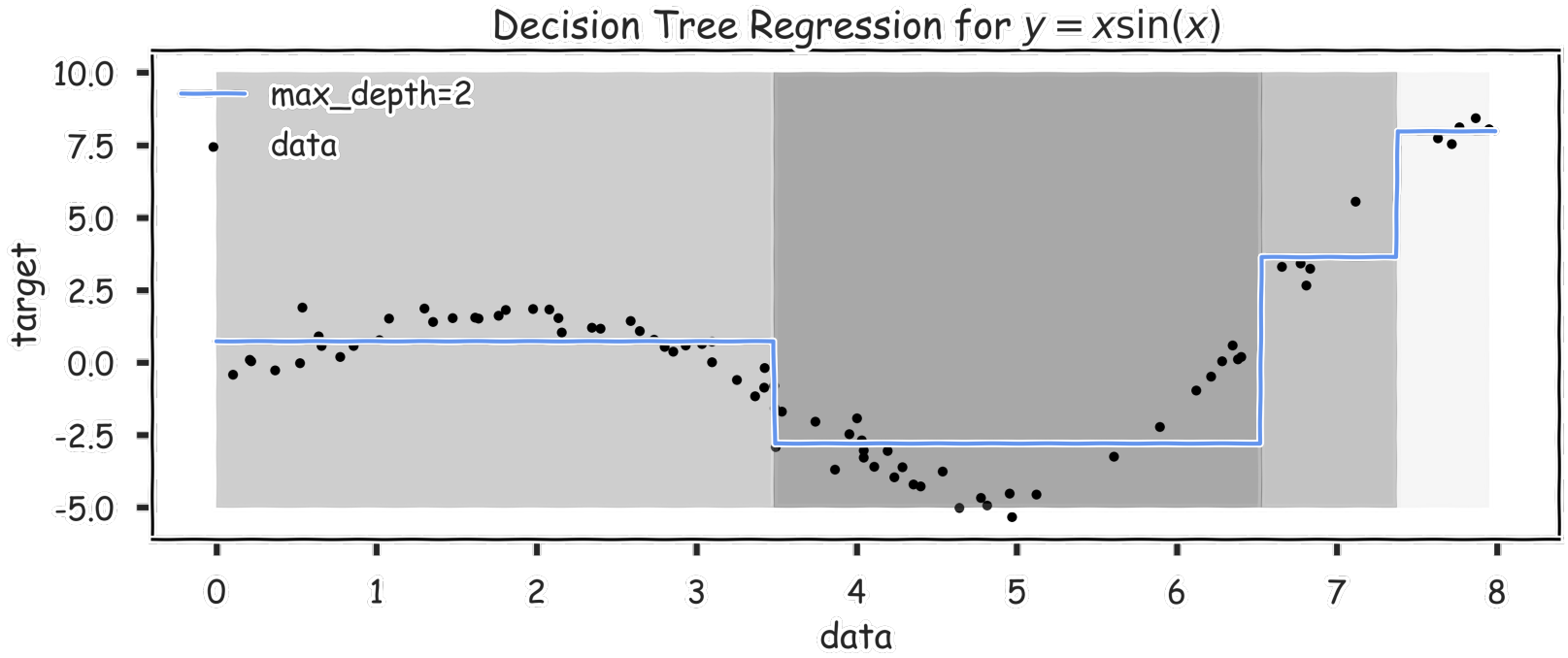
     or equivalently,

$$\underset{j,t_j}{\operatorname{argmin}} \left\{ \frac{N_1}{N} \operatorname{Var}(y|x \in R_1) + \frac{N_2}{N} \operatorname{Var}(y|x \in R_2) \right\}$$

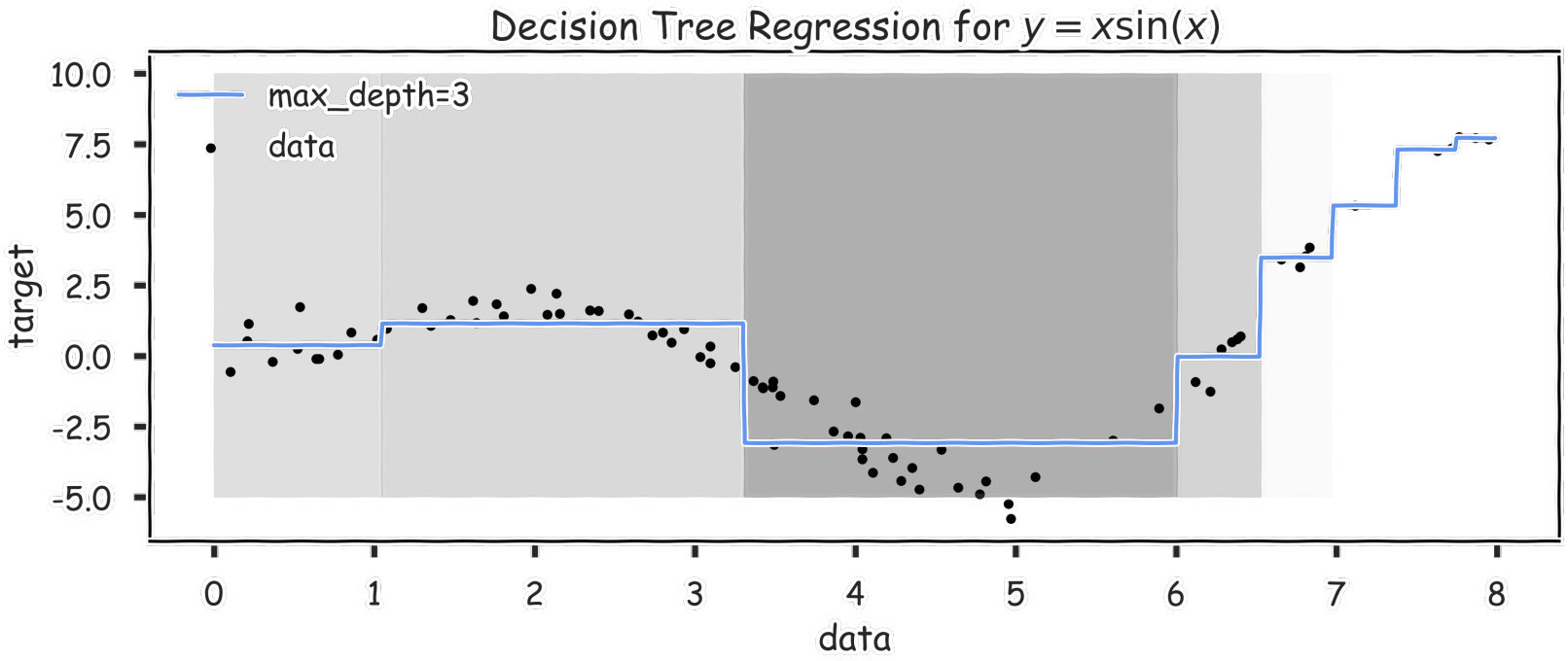
     where N_i is the number of training points in R_i and N is the number of points in R.

  3. Recurse on on each new node until ***stopping condition*** is met

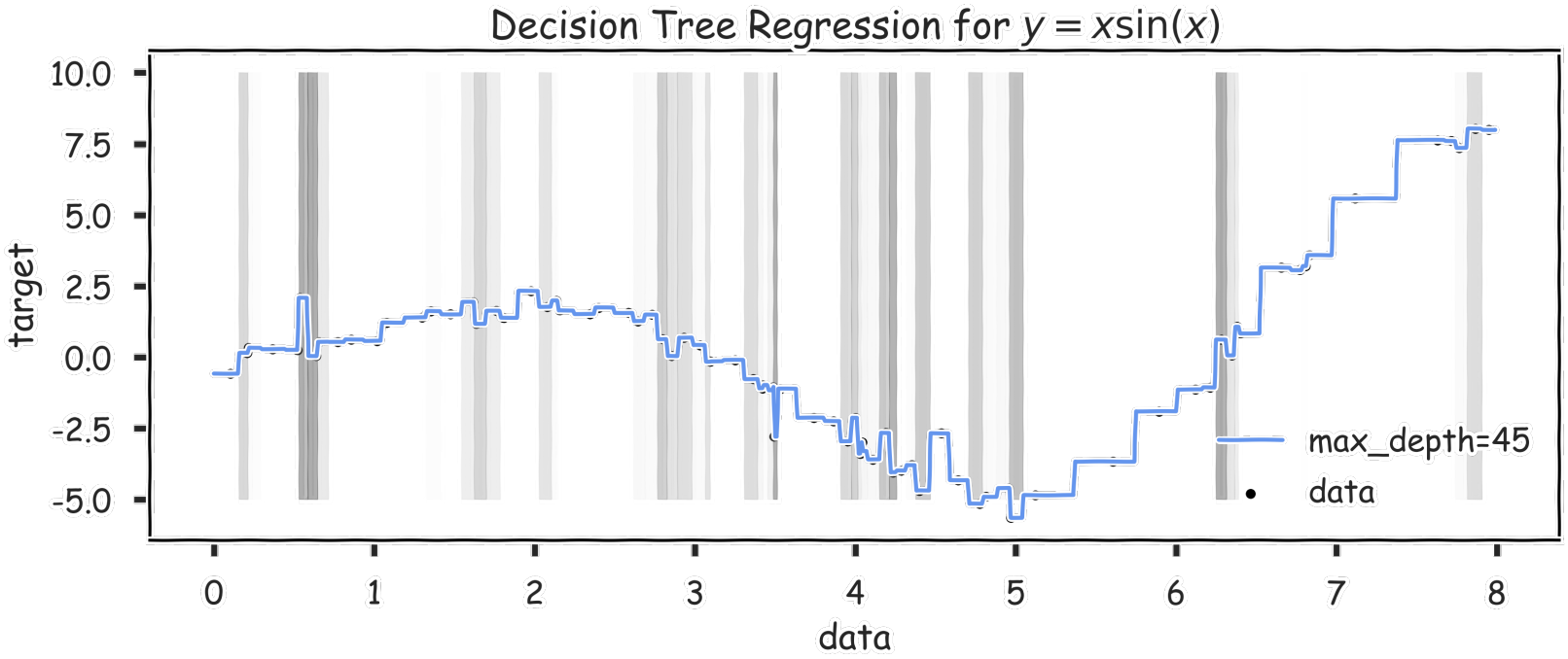
Regression Trees Prediction (grey scale represents MSE)



Regression Trees Prediction (grey scale represents MSE)



Regression Trees Prediction (grey scale represents MSE)



Stopping Conditions

- Most of the stopping conditions, like maximum depth or minimum number of points in region, we saw last time can still be applied.
- In the place of purity gain, we can instead compute accuracy gain for splitting a region R
- $$\text{Gain}(R) = \Delta(R) = MSE(R) - \frac{N_1}{N} MSE(R_1) - \frac{N_2}{N} MSE(R_2)$$
- and stop the tree when the gain is less than some pre-defined threshold.

Bagging
(aka Bootstrap Aggregating)

Bagging

- One way to adjust for the high variance of the output of an experiment is to perform the experiment multiple times and then average the results.
- The same idea can be applied to high variance models:
    1. **(Bootstrap)** we generate multiple samples of training data, via bootstrapping. We train a full decision tree on each sample of data.
    2. **(Aggregate)** for a given input, we output the averaged outputs of all the models for that input.
- For classification, we return the class that is outputted by the plurality of the models. For regression we return the average of the outputs for each tree.
- This method is called ***Bagging*** (Breiman, 1996), short for, of course, Bootstrap Aggregating.

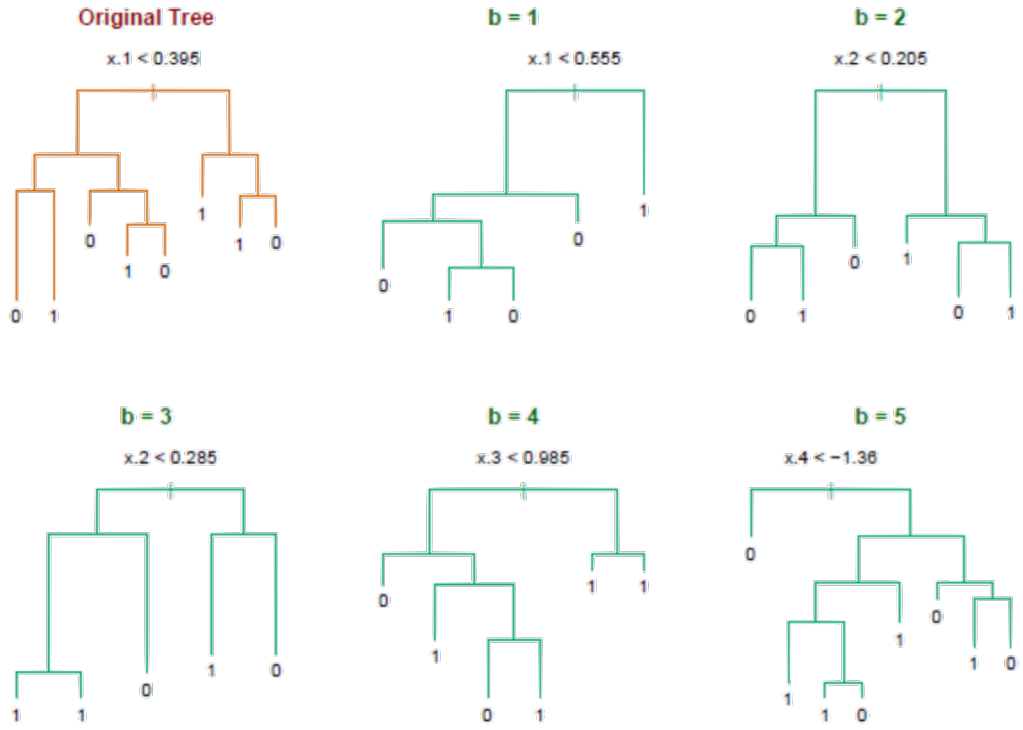
Bagging

- Note that bagging enjoys the benefits of:

    1. High expressiveness - by using full trees each model is able to approximate complex functions and decision boundaries.

    2. Low variance - averaging the prediction of all the models reduces the variance in the final prediction, assuming that we choose a sufficiently large number of trees.

Bagging


Original Tree
x.1 < 0.395
b = 1
x.1 < 0.555
b = 2
x.2 < 0.205
b = 3
x.2 < 0.285
b = 4
x.3 < 0.985
b = 5
x.4 < −1.36

Bagging

- **Question:** Do you see any problems?

- Still some overfitting if the trees are too large
- If trees are too shallow it can still underfit
- Interpretability
- The **major drawback** of bagging (and other ***ensemble methods*** that we will study) is that the averaged model is no longer easily interpretable - i.e. one can no longer trace the 'logic' of an output through a series of decisions based on predictor values!

Bagging

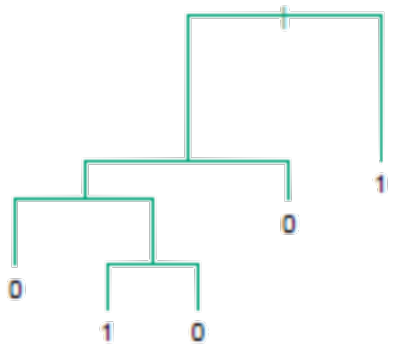
Original Data

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
X_4	y_4
X_5	y_5
⋮	⋮
X_n	y_n

Bootstrap Sample 1

X	Y
X_4	y_4
X_{14}	y_{14}
X_1	y_1
X_2	y_2
X_{35}	y_{35}
⋮	⋮
X_k	y_k

Decision Tree 1



Used and unused data

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
X_4	y_4
X_5	y_5
⋮	⋮
X_n	y_n

Bagging

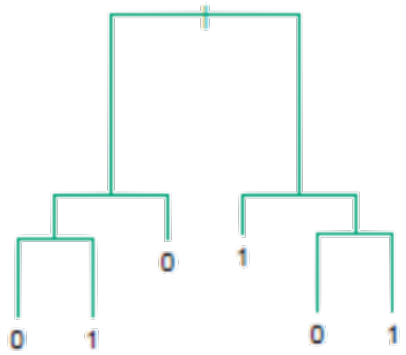
Original Data

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
X_4	y_4
X_5	y_5
⋮	⋮
X_n	y_n

Bootstrap Sample 2

X	Y
X_5	y_5
X_3	y_3
X_{12}	y_{12}
X_{43}	y_{43}
X_1	y_1
⋮	⋮
X_k	y_k

Decision Tree 2



Used and unused data

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
X_4	y_4
X_5	y_5
⋮	⋮
X_n	y_n

Bagging

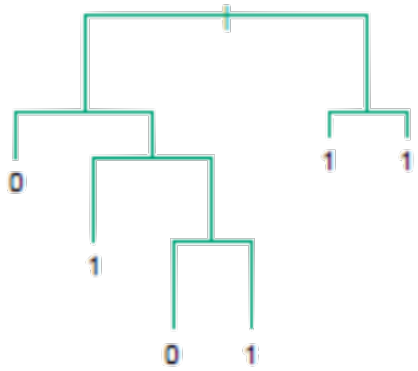
Original Data

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
X_4	y_4
X_5	y_5
⋮	⋮
X_n	y_n

Bootstrap Sample 3

X	Y
X_9	y_9
X_4	y_4
X_1	y_1
X_1	y_1
X_{65}	y_{65}
⋮	⋮
X_k	y_k

Decision Tree 3



Used and unused data

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
X_4	y_4
X_5	y_5
⋮	⋮
X_n	y_n

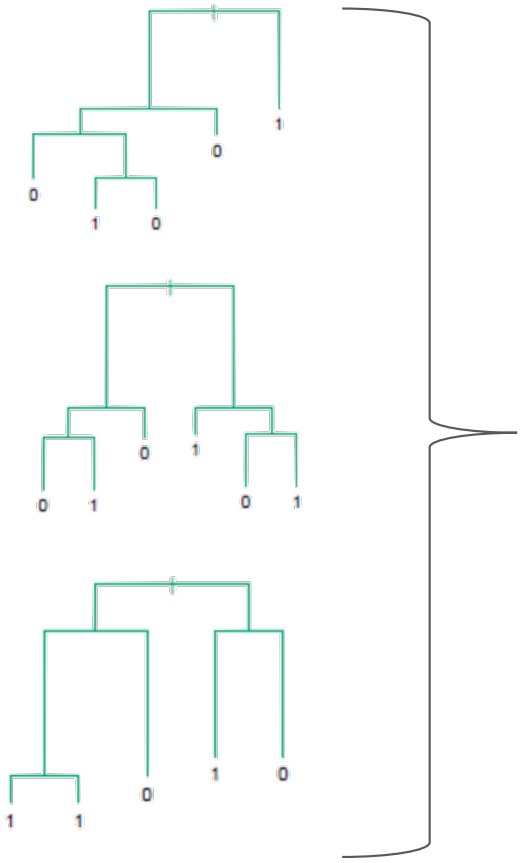
Point-wise out-of-bag error

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
⋮	⋮
X_i	y_i
⋮	⋮
X_n	y_n

Point-wise out of bag error

B Trees that did not see $\{X_i, y_i\}$

X	Y
X_1	y_1
X_2	y_2
X_3	y_3
⋮	⋮
X_i	y_i
⋮	⋮
X_n	y_n



Classification

$$\hat{y}_{i,pw} = \text{majority}(\hat{y}_i)$$

$$e_i = \mathbb{I}(\hat{y}_{i,pw} = y_i)$$

Regression

$$\hat{y}_{i,pw} = \sum_{j \in B} \hat{y}_{i,j}$$

$$e_i = \left(y_i - \hat{y}_{i.pw}\right)^2$$

OOB Error

- We average the point-wise out-of-bag error over the full training set.

Classification

$$Error_{OOB} = \sum_{i}^{n} e_i = \sum_{i}^{n} \mathbb{I}(\hat{y}_{i,pw} = \mathrm{y_i})$$

Regression

$$Error_{OOB} = \sum_{i}^{n} e_i = \sum_{i}^{n} \left(y_i - \hat{y}_{i,pw}\right)^2$$

Out-of-Bag Error

- Bagging is an example of an ***ensemble method***, a method of building a single model by training and aggregating multiple models.
- With ensemble methods, we get a new metric for assessing the predictive performance of the model, the ***out-of-bag error***.
- Given a training set and an ensemble of modeled each trained on a bootstrap sample, we compute the ***out-of-bag error*** of the averaged model by

    1. For each point in the training set, we average the predicted output for this point over the models whose bootstrap training set excludes this point. We compute the error or squared error of this averaged prediction. Call this the point-wise out-of-bag error.
    2. We average the point-wise out-of-bag error over the full training set.

Bagging

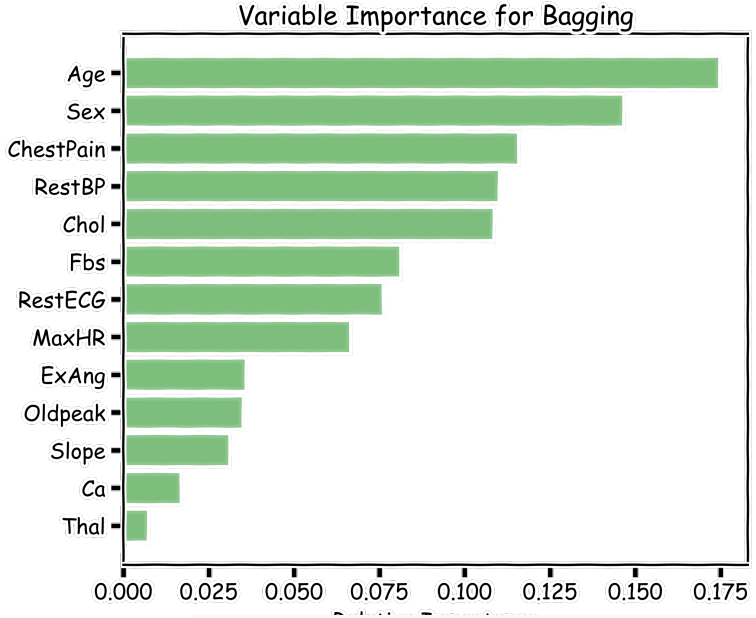
- **Question:** Do you see any problems?

- Still some overfitting if the trees are too large
- If trees are too shallow it can still underfits.
- interpretability
- The **major drawback** of bagging (and other ***ensemble methods*** that we will study) is that the averaged model is no longer easily interpretable - i.e. one can no longer trace the 'logic' of an output through a series of decisions based on predictor values!

Variable Importance for Bagging

- Bagging improves prediction accuracy at the expense of interpretability.

- Calculate the total amount that the RSS (for regression) or Gini index (for classification) is decreased due to splits over a given predictor, averaged over all B trees.



100 trees, max_depth=10

Bagging

- **Question:** Do you see any problems?

- Still some overfitting if the trees are too large
- If trees are too shallow it can still underfits.
- interpretability
- The **major drawback** of bagging (and other ***ensemble methods*** that we will study) is that the averaged model is no longer easily interpretable - i.e. one can no longer trace the 'logic' of an output through a series of decisions based on predictor values!

Improving on Bagging

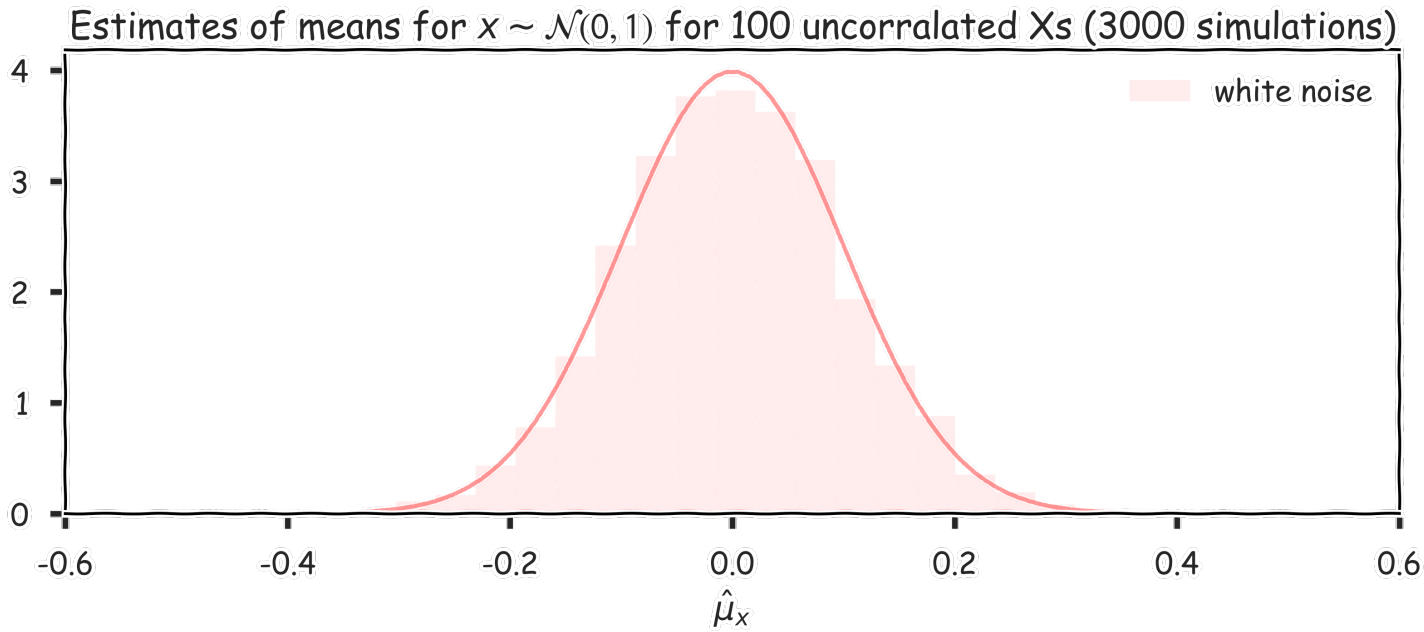
- In practice, the ensembles of trees in Bagging tend to be highly correlated.

- Suppose we have an extremely strong predictor, x_j , in the training set amongst moderate predictors. Then the greedy learning algorithm ensures that most of the models in the ensemble will choose to split on x_j in early iterations.

- That is, each tree in the ensemble is identically distributed, with the expected output of the averaged model the same as the expected output of any one of the trees.

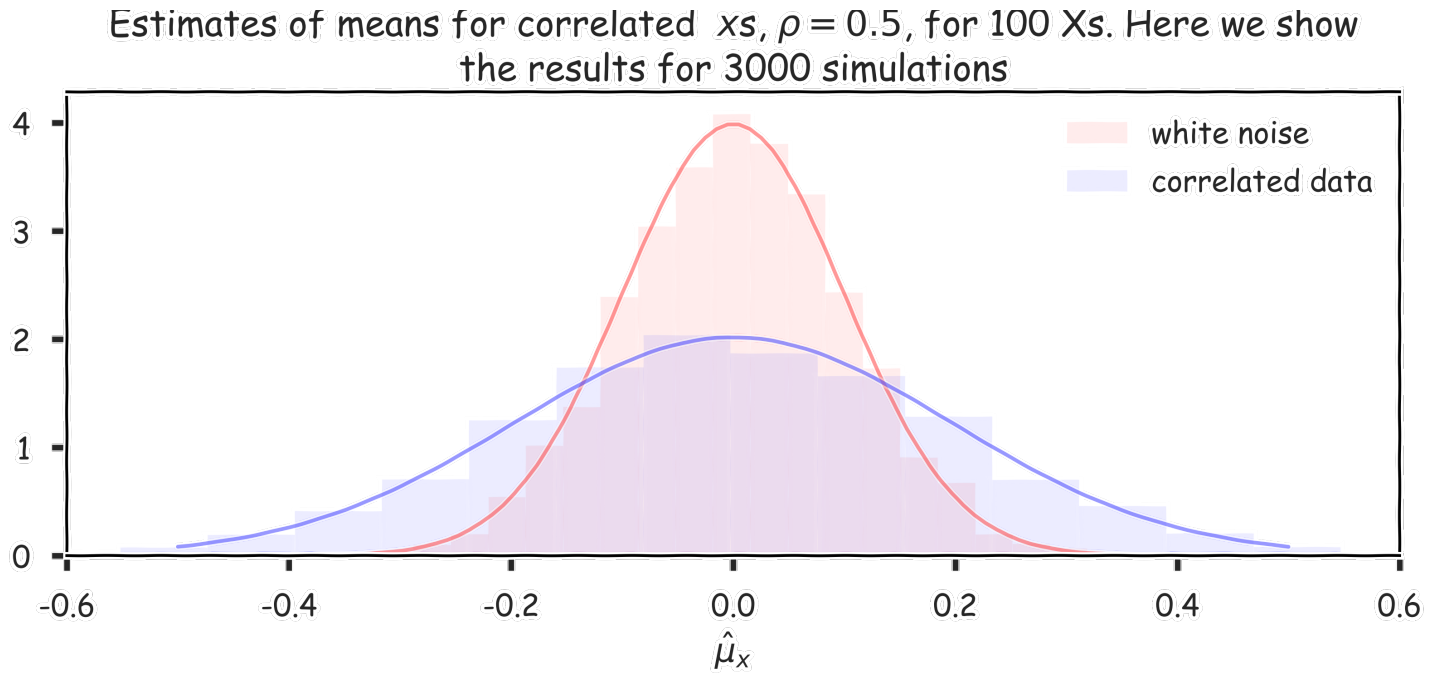
Improving on Bagging

- Recall, for B number of identically and independently distributed variable, X, with variance σ^2, the variance of the estimate of the mean is :
- $\text{var}(\hat{\mu}_x) = \frac{\sigma^2}{B}$



Improving on Bagging

- For B number of identically but not independently distributed variables with pairwise correlation ρ and variance σ^2, the variance of their mean is
- $\text{var}(\hat{\mu}_x) \propto \sigma^2\rho/B$



Bagging

- **Question:** Do you see any problems?

- Still some overfitting if the trees are too large
- If trees are too shallow it can still underfits.
- interpretability
- The **major drawback** of bagging (and other ***ensemble methods*** that we will study) is that the averaged model is no longer easily interpretable - i.e. one can no longer trace the 'logic' of an output through a series of decisions based on predictor values!

Random Forests

Random Forests

- ***Random Forest*** is a modified form of bagging that creates ensembles of independent decision trees.

- To de-correlate the trees, we:

    1. train each tree on a separate bootstrap sample of the full training set (same as in bagging)

    2. for each tree, at each split, we ***randomly*** select a set of J' predictors from the full set of predictors.

- From amongst the J' predictors, we select the optimal predictor and the optimal corresponding threshold for the split.

Tuning Random Forests

- Random forest models have multiple hyper-parameters to tune:

    1. the number of predictors to randomly select at each split
    2. the total number of trees in the ensemble
    3. the minimum leaf node size

- In theory, each tree in the random forest is full, but in practice this can be computationally expensive (and added redundancies in the model), thus, imposing a minimum node size is not unusual.

Tuning Random Forests

- There are standard (default) values for each of random forest hyper-parameters recommended by long time practitioners, but generally these parameters should be tuned through **OOB** (making them data and problem dependent).

e.g. number of predictors to randomly select at each split:

- $\sqrt{N_j}$ for classification
- $\frac{N}{3}$ for regression

- Using out-of-bag errors, training and cross validation can be done in a single sequence - we cease training once the out-of-bag error stabilizes

Variable Importance for RF

- Same as with Bagging:
- Calculate the total amount that the RSS (for regression) or Gini index (for classification) is decreased due to splits over a given predictor, averaged over all B trees.

Variable Importance for RF

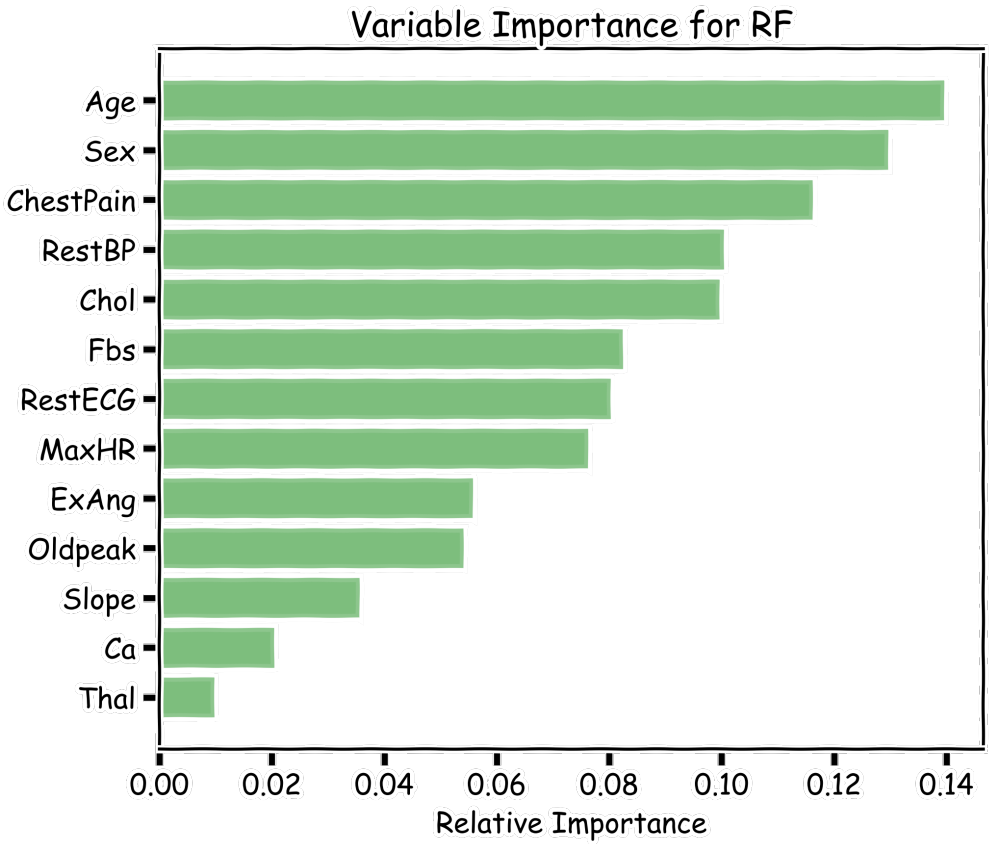
- **Alternative:**

- Record the prediction accuracy on the *oob* samples for each tree.

- Randomly permute the data for column *j* in the *oob* samples the record the accuracy again.

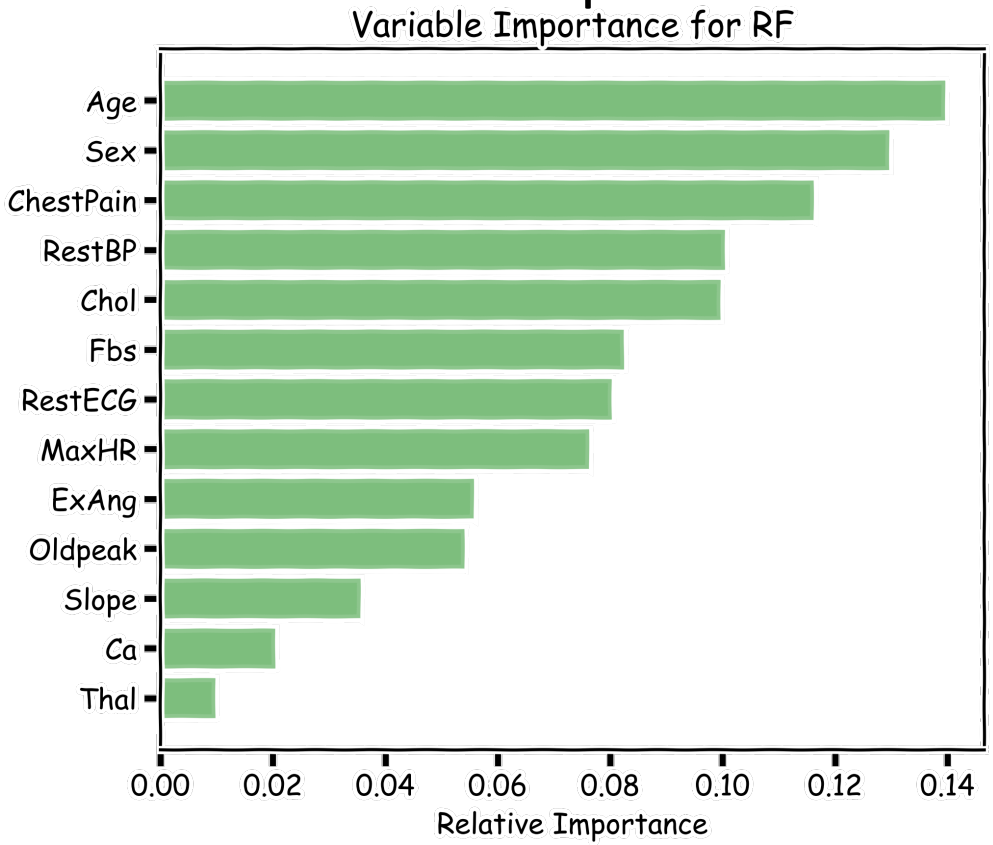
- The decrease in accuracy as a result of this permuting is averaged over all trees, and is used as a measure of the importance of variable *j* in the random forest.

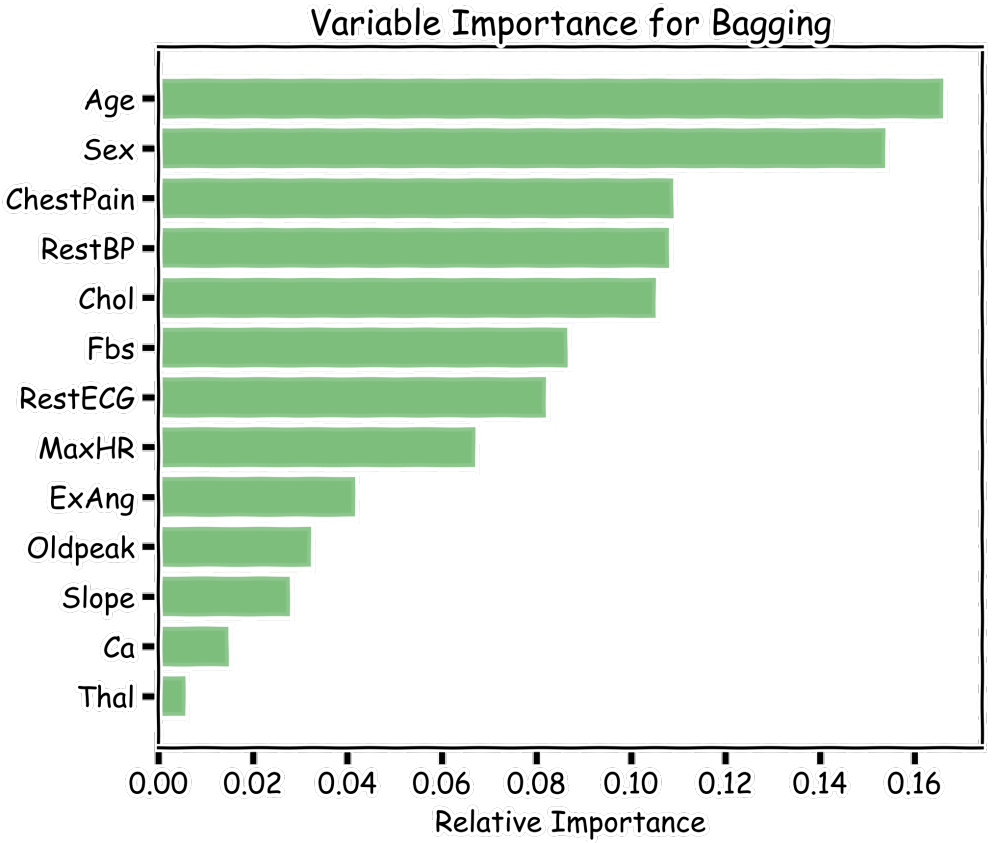
Variable Importance for RF



100 trees, max_depth=10

Variable Importance for RF





100 trees, max_depth=10

Final Thoughts on Random Forests

- When the number of predictors is large, but the number of relevant predictors is small, random forests can perform poorly.

- **Question**: Why?
- In each split, the chances of selected a relevant predictor will be low and hence most trees in the ensemble will be weak models.

Final Thoughts on Random Forests (cont.)

- Increasing the number of trees in the ensemble generally does not increase the risk of overfitting.
- Again, by decomposing the generalization error in terms of bias and variance, we see that increasing the number of trees produces a model that is at least as robust as a single tree.
- However, if the number of trees is too large, then the trees in the ensemble may become more correlated, increase the variance.

Boosting

Motivation for Boosting

- **Question:** Could we address the shortcomings of single decision trees models in some other way?

- For example, rather than performing variance reduction on complex trees, can we decrease the bias of simple trees - make them more expressive?

- A solution to this problem, making an expressive model from simple trees, is another class of ensemble methods called ***boosting***.

Gradient Boosting

- The key intuition behind boosting is that one can take an ensemble of simple models $\{T_h\}_{h \in H}$ and additively combine them into a single, more complex model.

- Each model T_h might be a poor fit for the data, but a linear combination of the ensemble

$$T = \sum_h \lambda_h T_H$$

- can be expressive/flexible.

- **Question:** But which models should we include in our ensemble? What should the coefficients or weights in the linear combination be?

Gradient Boosting: the algorithm

Gradient boosting is a method for iteratively building a complex regression model T by adding simple models. Each new simple model added to the ensemble compensates for the weaknesses of the current ensemble.

1. Fit a simple model $T^{(0)}$ on the training data

$$\{(x_1, y_1), \ldots, (x_N, y_N)\}$$

Set $T \leftarrow T^{(0)}$. Compute the residuals $\{r_1, \ldots, r_N\}$ for T.

2. Fit a simple model, $T^{(1)}$, to the current ***residuals***, i.e. train using

$$\{(x_1, r_1), \ldots, (x_N, r_N)\}$$

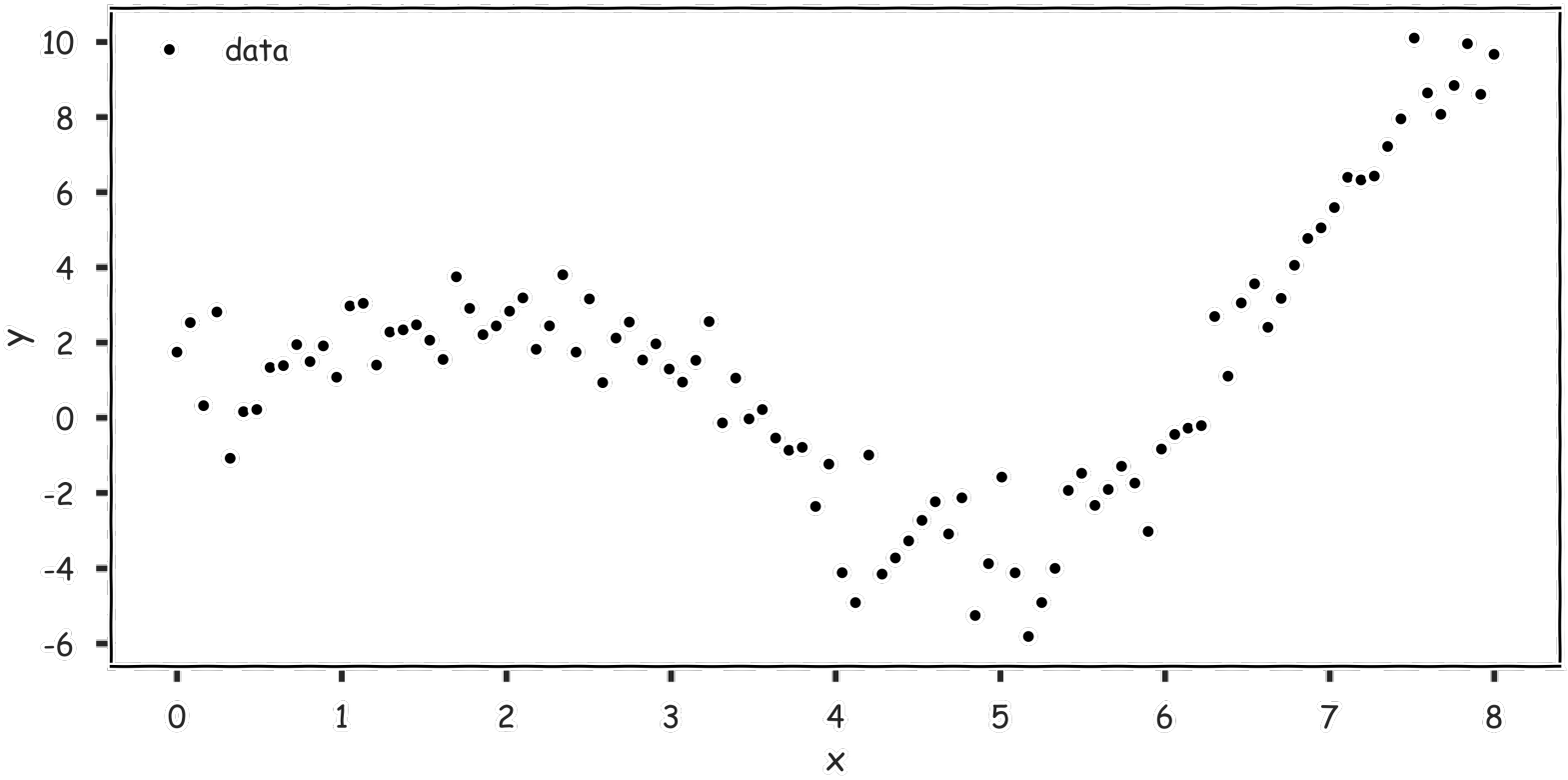
3. Set $T \leftarrow T + \lambda T^{(1)}$

4. Compute residuals, set $r_n \leftarrow r_n - \lambda T^i(x_n),\ n = 1, \ldots, N$

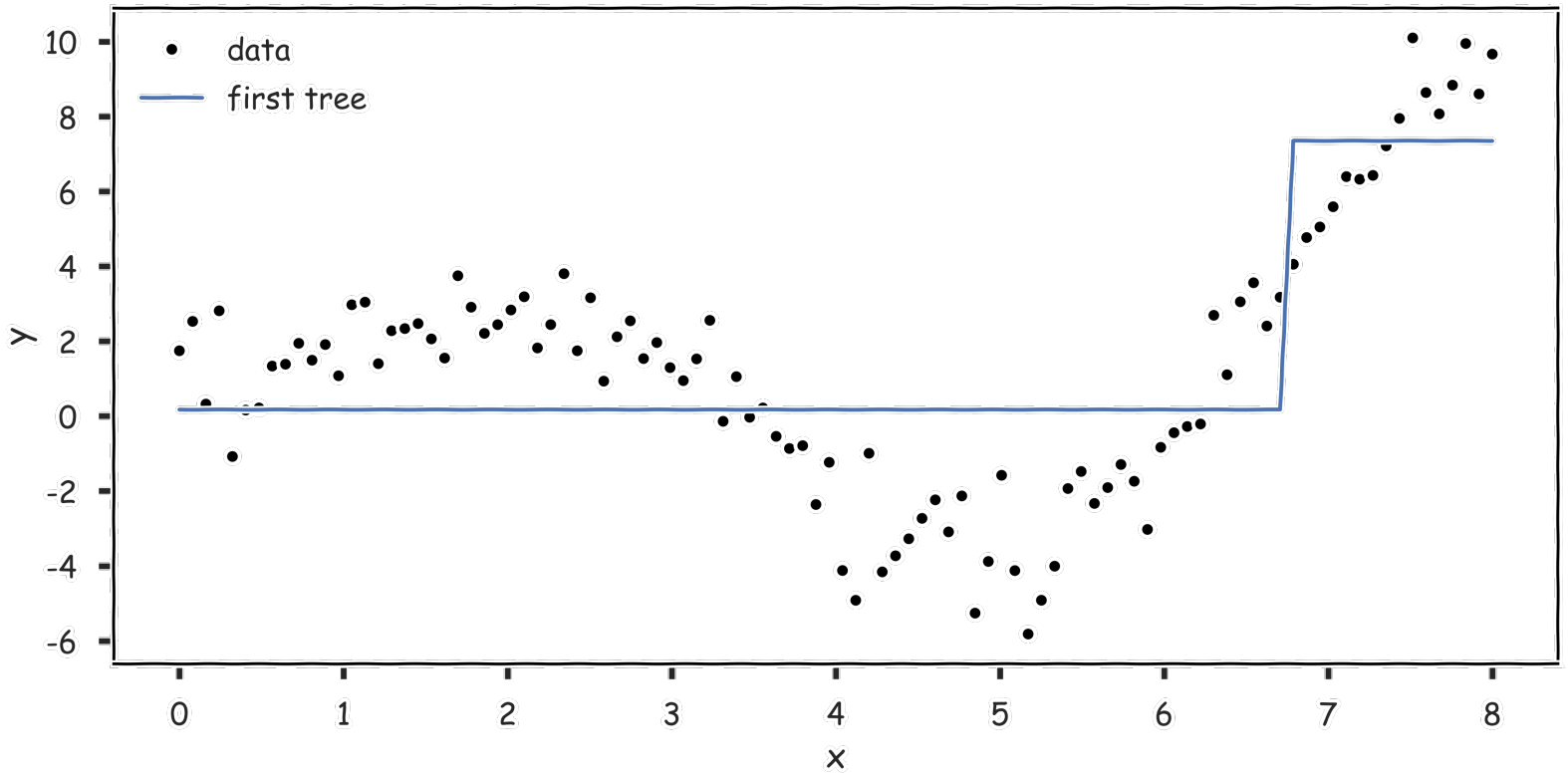
5. Repeat steps 2-4 until ***stopping*** condition met.

where λ is a constant called the ***learning rate***.

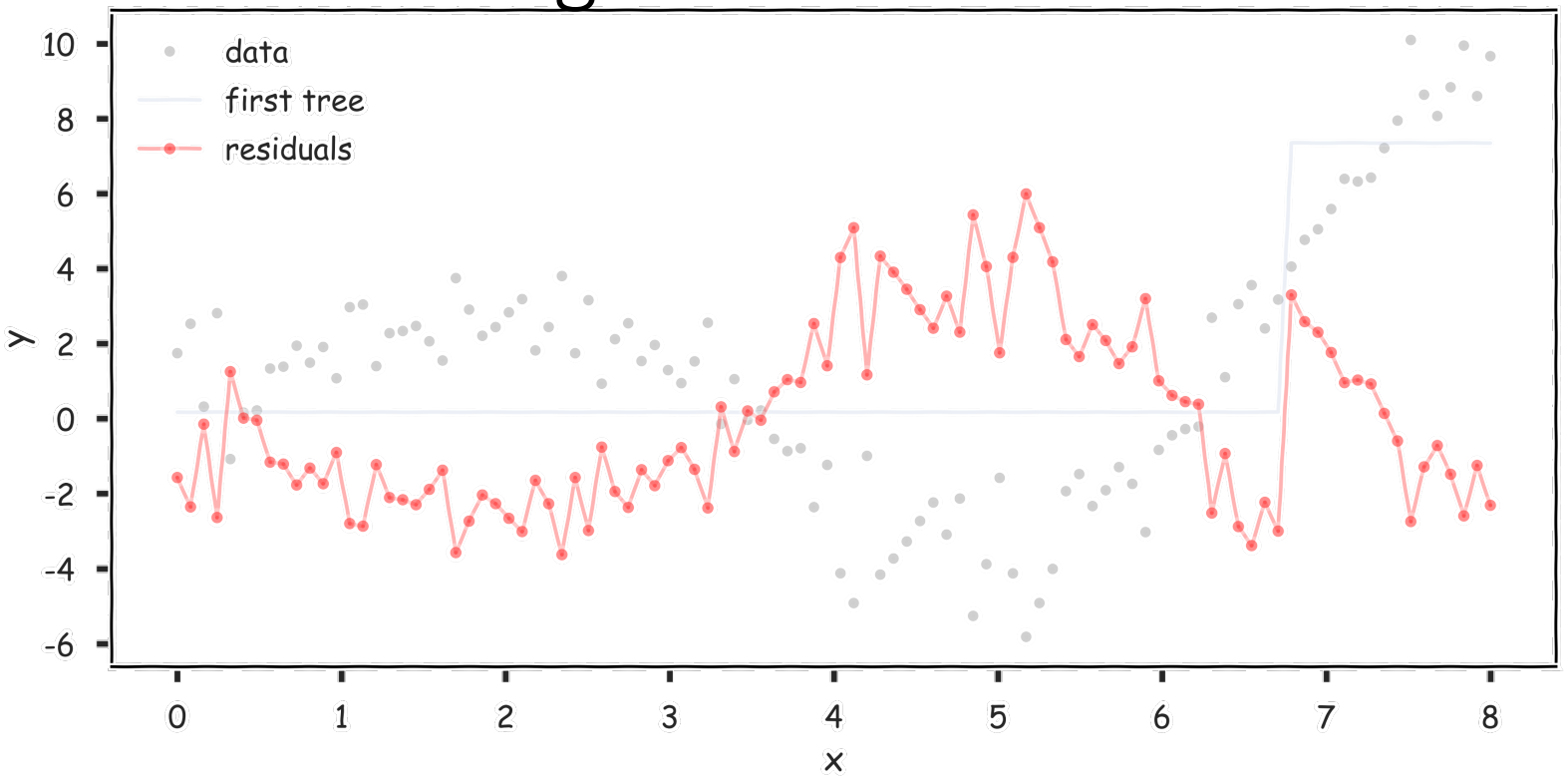
Gradient Boosting: illustration



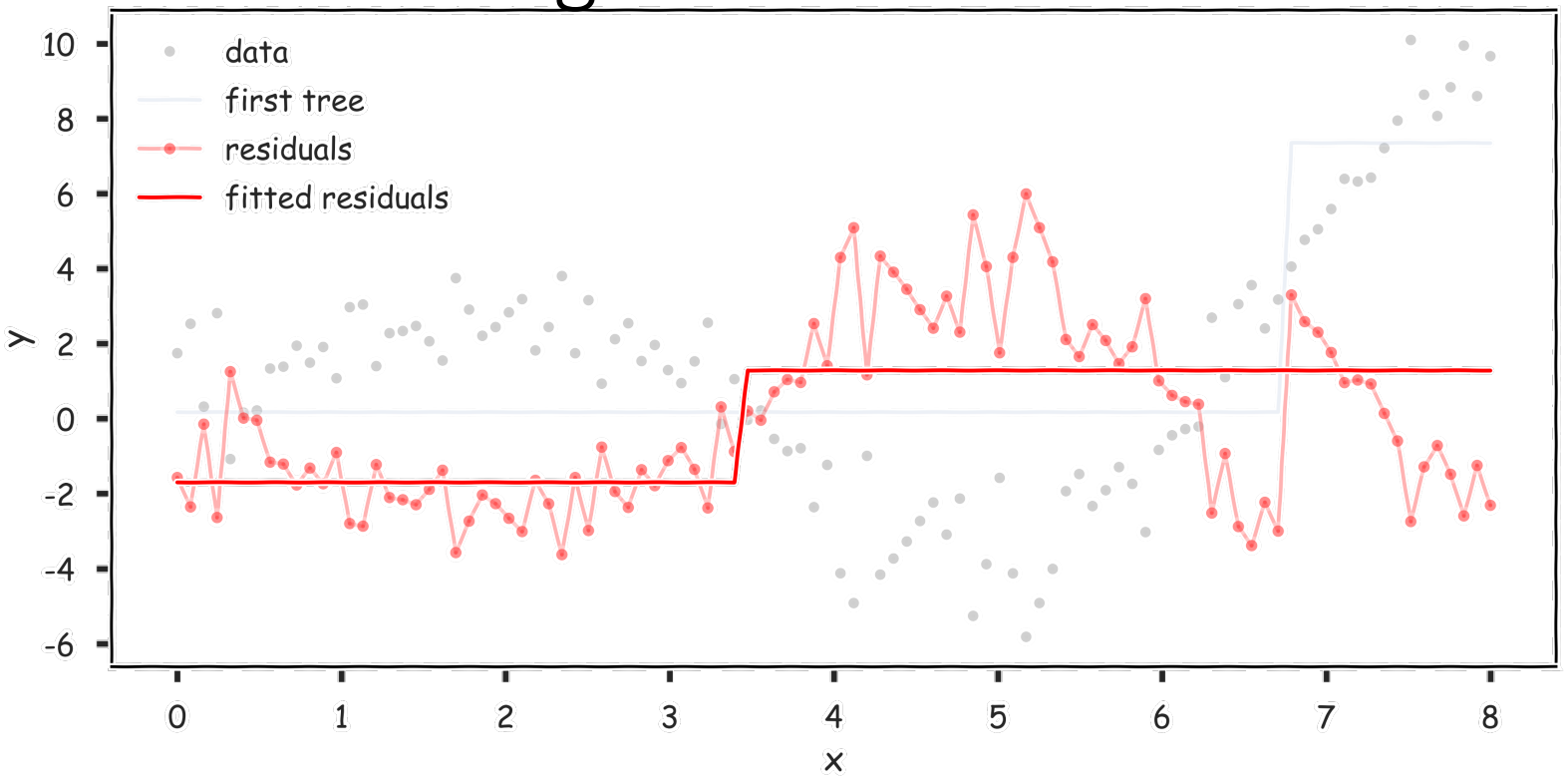
Gradient Boosting: illustration



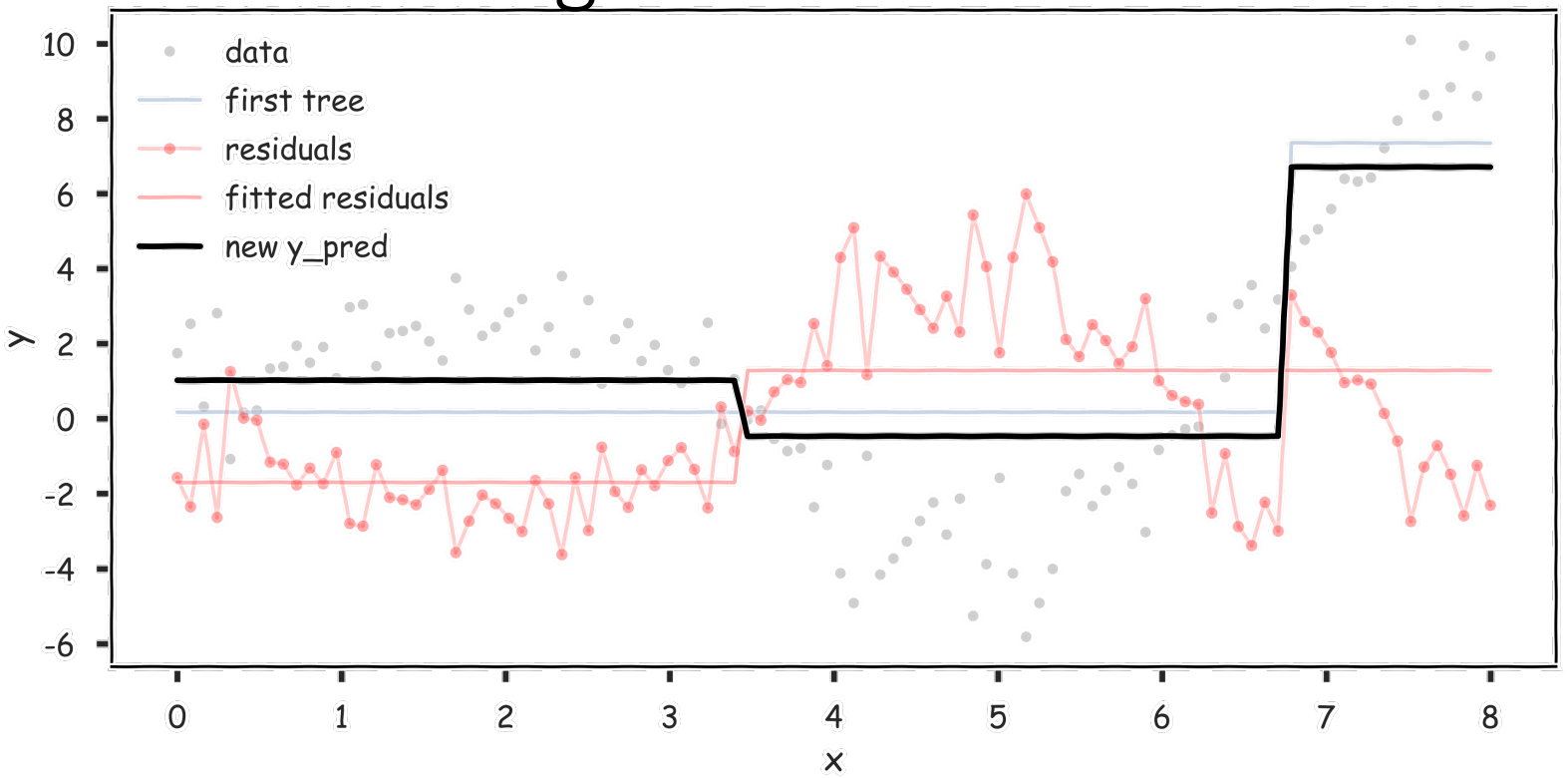
Gradient Boosting: illustration



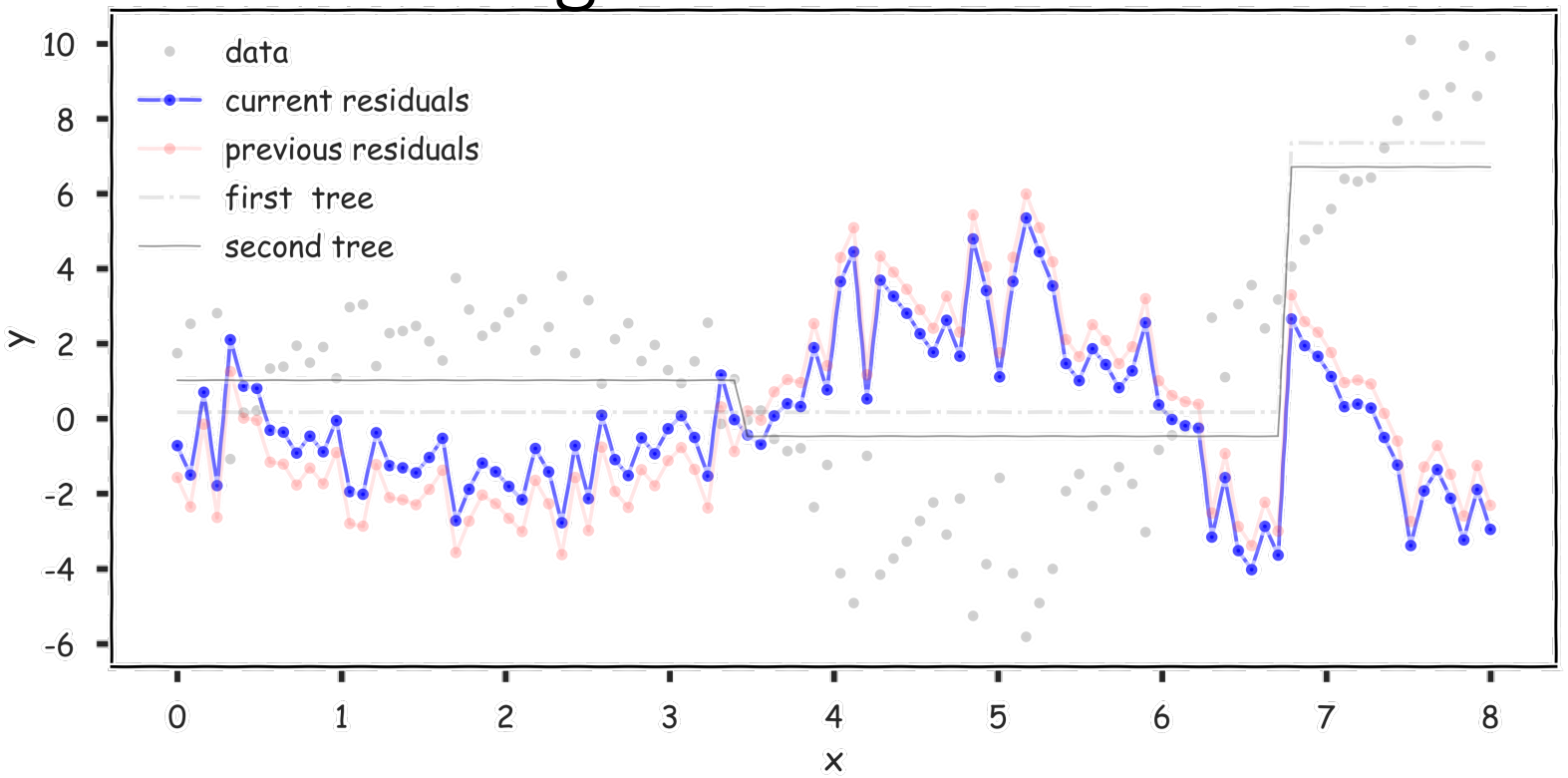
Gradient Boosting: illustration



Gradient Boosting: illustration



Gradient Boosting: illustration



Why Does Gradient Boosting Work?

- Intuitively, each simple model $T^{(i)}$ we add to our ensemble model T, models the errors of T.
- Thus, with each addition of $T^{(i)}$, the residual is reduced
- $r_n - \lambda T^{(i)}(x_n)$

- **Note** that gradient boosting has a tuning parameter, λ.

- If we want to easily reason about how to choose λ and investigate the effect of λ on the model T, we need a bit more mathematical formalism.
- In particular, how can we effectively descend through this optimization via an iterative algorithm?
- We need to formulate gradient boosting as a type of ***gradient descent***.

Review: A Brief Sketch of Gradient Descent

- In optimization, when we wish to minimize a function, called the ***objective function***, over a set of variables, we compute the partial derivatives of this function with respect to the variables.

- If the partial derivatives are sufficiently simple, one can analytically find a common root - i.e. a point at which all the partial derivatives vanish; this is called a ***stationary point.***

- If the objective function has the property of being ***convex***, then the stationary point is precisely the min.

Review: A Brief Sketch of Gradient Descent the Algorithm

- In practice, our objective functions are complicated and analytically find the stationary point is intractable.
- Instead, we use an iterative method called ***gradient descent***:

1. Initialize the variables at any value:

$$x = [x_1, ..., x_J]$$

2. Take the gradient of the objective function at the current variable values:

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_J}(x)\right]$$

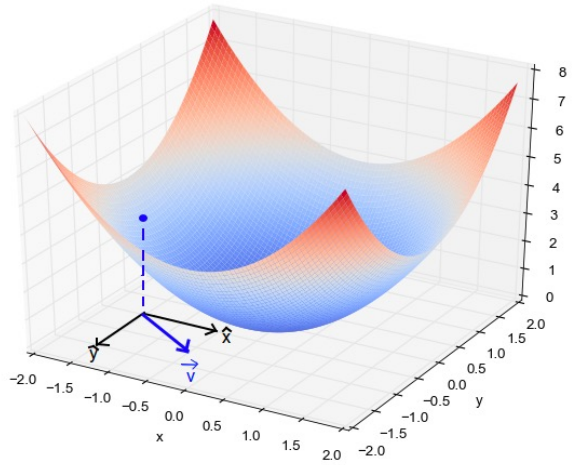
3. Adjust the variables values by some negative multiple of the gradient:

$$x \leftarrow x - \lambda \nabla f(x)$$

- The factor λ is often called the learning rate.

Why Does Gradient Descent Work?

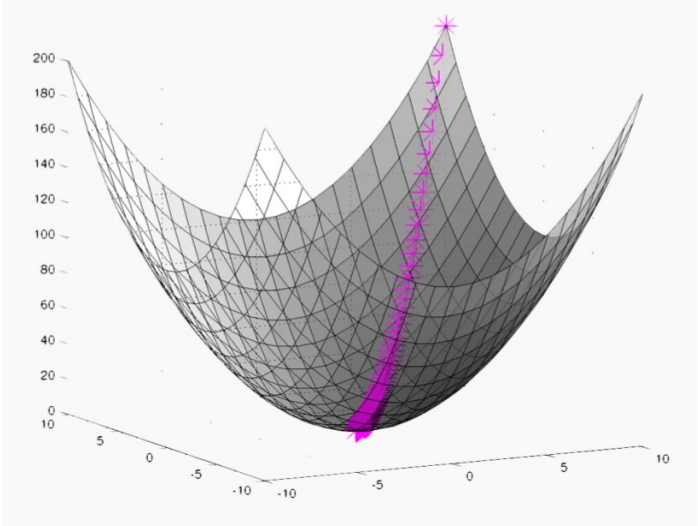
- **Claim:** If the function is convex, this iterative methods will eventually move x close enough to the minimum, for an appropriate choice of λ.
- **Why does this work?** Recall, that as a vector, the gradient at at point gives the direction for the greatest possible rate of increase.



Why Does Gradient Descent Work?

- Subtracting a λ multiple of the gradient from x, moves x in the ***opposite*** direction of the gradient (hence towards the steepest decline) by a step of size λ.

- If *f* is convex, and we keep taking steps descending on the graph of *f*, we will eventually reach the minimum.



Gradient Boosting as Gradient Descent

- Often in regression, our objective is to minimize the MSE

$$\text{MSE}(\hat{y}_1, \ldots, \hat{y}_N) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

- Treating this as an optimization problem, we can try to directly minimize the MSE with respect to the predictions

$$\begin{aligned} \nabla \text{MSE} &= \left[\frac{\partial \text{MSE}}{\partial \hat{y}_1}, \ldots, \frac{\partial \text{MSE}}{\partial \hat{y}_N} \right] \\ &= -2 \left[y_1 - \hat{y}_1, \ldots, y_N - \hat{y}_N \right] \\ &= -2 \left[r_1, \ldots, r_N \right] \end{aligned}$$

- The update step for gradient descent would look like

$$\hat{y}_n \leftarrow \hat{y}_n + \lambda r_n, \quad n = 1, \ldots, N$$

Gradient Boosting as Gradient Descent (cont.)

- There are two reasons why minimizing the MSE with respect to $\hat{y}_n$'s is not interesting:

- We know where the minimum MSE occurs: $\hat{y}_n = y_n$, for every *n*.
- Learning sequences of predictions, $\hat{y}_n^1, \ldots, \hat{y}_n^i, \ldots$, does not produce a model. The predictions in the sequences do not depend on the predictors!

Gradient Boosting as Gradient Descent (cont.)

- The solution is to change the update step in gradient descent. Instead of using the gradient - the residuals - we use an ***approximation*** of the gradient that depends on the predictors:

- $\hat{y} \leftarrow \hat{y}_n + \lambda\, \hat{r}_n(x_n), \quad n = 1, \ldots, N$
  In gradient boosting, we use a simple model to approximate the residuals, $\hat{r}_n(x_n)$, in each iteration.
- **Motto:** gradient boosting is a form of gradient descent with the MSE as the objective function.
- **Technical note:** note that gradient boosting is descending in a space of models or functions relating x_n to y_n!

Gradient Boosting as Gradient Descent (cont.)

- But why do we care that gradient boosting is gradient descent?

- By making this connection, we can import the massive amount of techniques for studying gradient descent to analyze gradient boosting.

-

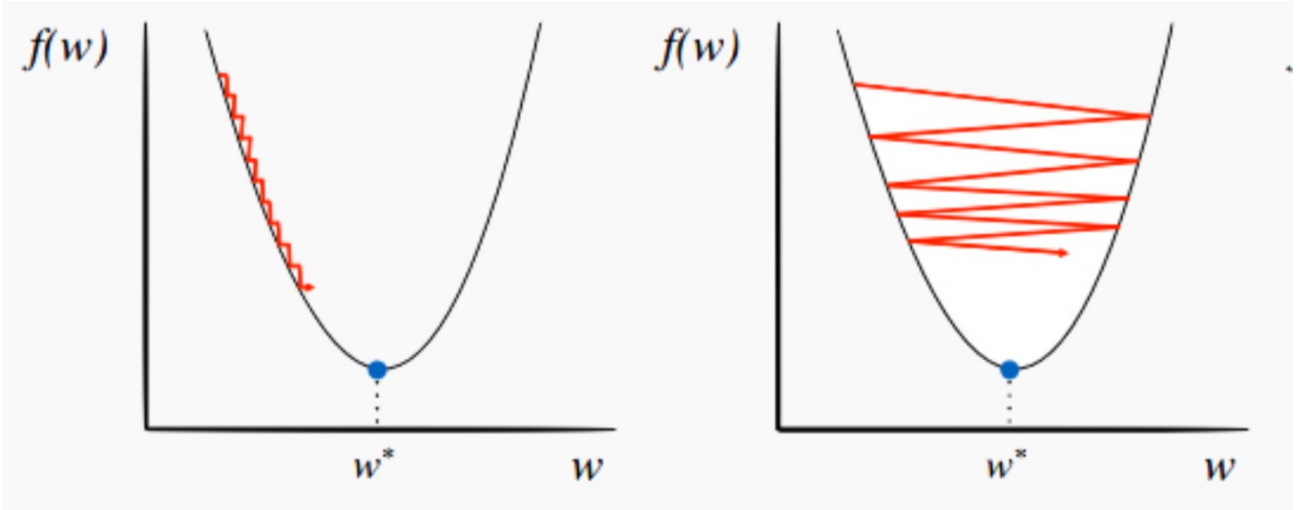
- **For example**, we can easily reason about how to choose the learning rate λ in gradient boosting.

Choosing a Learning Rate

- Under ideal conditions, gradient descent iteratively approximates and converges to the optimum.
- ***When do we terminate gradient descent?***
- We can limit the number of iterations in the descent. But for an arbitrary choice of maximum iterations, we cannot guarantee that we are sufficiently close to the optimum in the end.
- If the descent is stopped when the updates are sufficiently small (e.g. the residuals of *T* are small), we encounter a new problem: the algorithm may never terminate!
- Both problems have to do with the magnitude of the learning rate, λ.

Choosing a Learning Rate

- For a constant learning rate, λ, if λ is too small, it takes too many iterations to reach the optimum.



- If λ is too large, the algorithm may 'bounce' around the optimum and never get sufficiently close.

Choosing a Learning Rate

- Choosing λ:
- If λ is a constant, then it should be tuned through cross validation.
- For better results, use a variable λ. That is, let the value of λ depend on the gradient

$$\lambda = h(\|\nabla f(x)\|),$$

where $\|\nabla f(x)\|$ is the magnitude of the gradient, $\nabla f(x)$. So

- around the optimum, when the gradient is small, λ should be small
- far from the optimum, when the gradient is large, λ should be larger

Extreme Gradient Boosting (XGBoost)

XGBoost

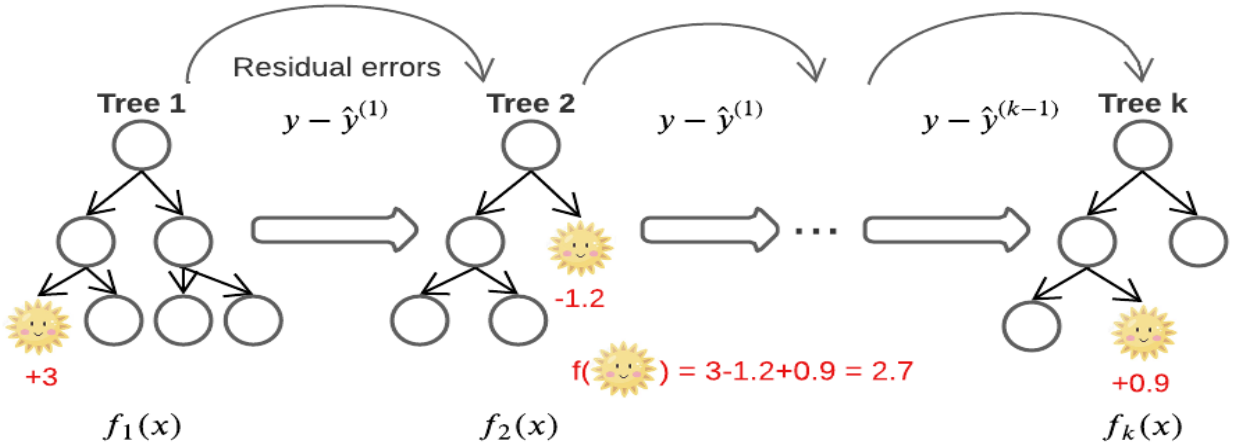
- XGBoost is a more **regularized form of Gradient Boosting**. XGBoost uses advanced regularization (L1 & L2), which improves model generalization capabilities.
- XGBoost delivers high performance as compared to Gradient Boosting. Its training is very fast and can be **parallelized** across clusters.

XGBoost

XGBoost is an ensemble tree method which uses K additive weak base learners (tress) to predict the output

$$\hat{y}_i = \phi(x_i) = \sum_{k=1}^{k} f_k(x_i), \; f_k \in \mathcal{F}$$

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$$



Regularized Objective Function

$$\mathcal{L}(\theta)^{(t)} = \sum_i l(\hat{y}_i^{(t)}, y_i) + \Omega(f_t)$$

$$= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

- Loss function
- Regularization term

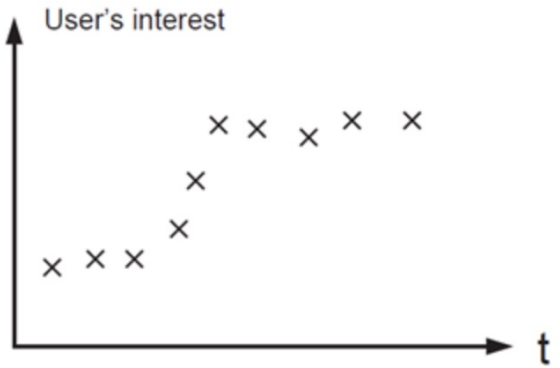
- For the regression case

$$\mathcal{L}(\theta)^{(t)} = \sum_{i=1}^{n} (y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)))^2 + \Omega(f_t)$$

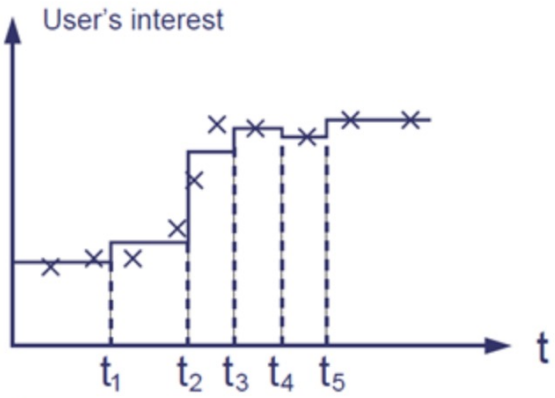
$$= \sum_{i=1}^{n} [2(\hat{y}_i^{(t-1)} - y_i) f_t(x_i) + f_t(x_i)^2] + \Omega(f_t) + const$$

Regularization

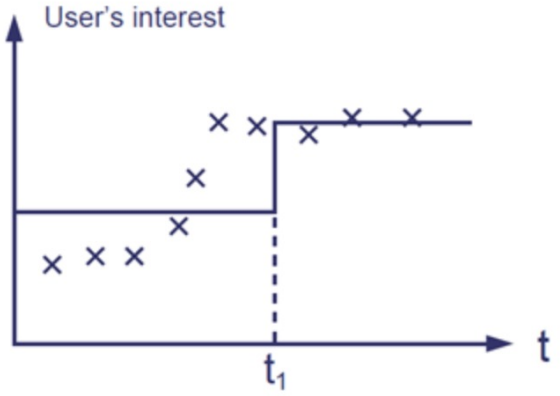
The **regularization term** is what people usually forget to add. The regularization term controls the complexity of the model, which helps us to avoid overfitting. This sounds a bit abstract, so let us consi following problem in the following picture. You are asked to *fit* visually a step function given the input points on the upper left corner of the image. Which solution among the three do you think is the best



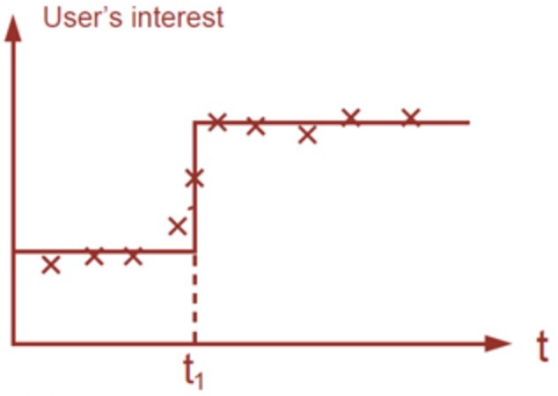
Observed user's interest on topic k against time t



☒ Too many splits, $\Omega(f)$ is high



☒ Wrong split point, $L(f)$ is high



☑ Good balance of $\Omega(f)$ and $L(f)$

Taylor Expansion of the Objective Function

- Taylor expansion

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

- Expand equation 1 at $\hat{y_i}^{(t-1)} - y_i$, then we have

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^{n} [l(y_i, \hat{y_i}^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

where the g_i and h_i are defined as

$$g_i = \partial_{\hat{y_i}^{(t-1)}} l(y_i, \hat{y_i}^{(t-1)})$$
$$h_i = \partial^2_{\hat{y_i}^{(t-1)}} l(y_i, \hat{y_i}^{(t-1)})$$

Objective Function

$\Omega(f) = \gamma T + \frac{1}{2}\lambda\|w\|^2$

- γ and λ are regularization parameters
- T number of leaves

$f_t(x) = w_{q(x)}$

- $f_t(x)$ regression trees
- $w \in R^T$ leaf scores
- $q : R^d \to \{1, 2, 3..., T\}$

- Removing the constant term and substituting

$$\tilde{\mathcal{L}}^{(t)} \simeq \sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)] + \gamma T + \frac{1}{2}\lambda\|w\|^2$$

$$= \sum_{j=1}^{T}[(\sum_{i\in I_j} g_i)w_j + \frac{1}{2}(\sum_{i\in I_j} h_i + \lambda)w_j^2] + \gamma T$$

Objective Function

- If we define $G_i = \sum_{i \in I_j} g_i$ and $H_i = \sum_{i \in I_j} h_i$, the objective function could be further compressed as

$$\tilde{\mathcal{L}}^{(t)} = \sum_{j=1}^{T} [G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2] + \lambda T \tag{2}$$

- The objective function only relies on the **first** and **second derivatives** of the loss function. Thus, the objective function can be **customized**.

Optimal Weights and Objective Function

- By setting $G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2$ equal to 0, we get

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

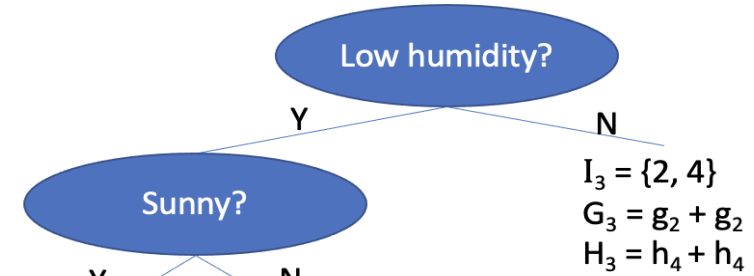
- Plug w_j^* back into equation 2 and the objective function is

$$\tilde{\mathcal{L}}^{(t)} = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \lambda T$$

Structure Score Calculation

$$\tilde{\mathcal{L}}^{(t)} = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j+\lambda} + \lambda T$$

Instance index		Gradient statistics
1		g1, h1
2		g2, h2
3		g3, h3
4		g4, h4



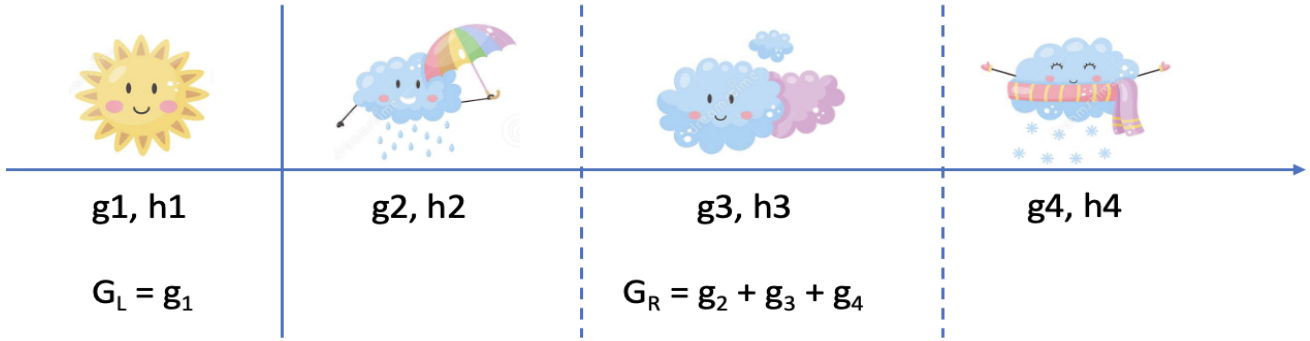
$$Obj = -\sum_j \frac{G_j^2}{H_j+\lambda} + 3\gamma$$

The smaller the score is, the better the structure is

Optimal Node Split

XGBoost finds the best node split using a exact greedy algorithm

$$\mathcal{L}_{splitgain} = \frac{1}{2}[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}] - \gamma$$



The larger the gain score is, the better the split is

XGBoost Advantages

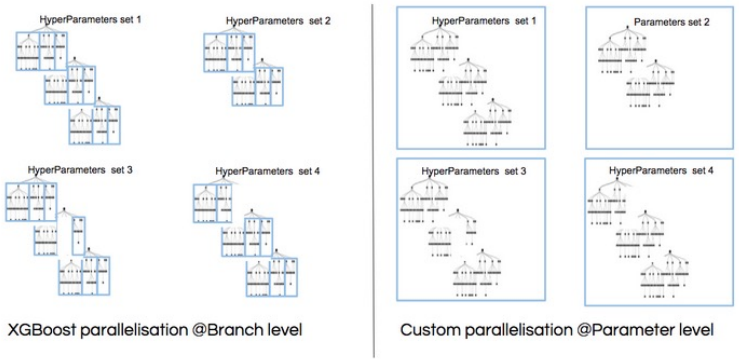
Regularization:

Standard GBM implementation has no regularization like XGBoost, therefore it also helps to reduce overfitting.
In fact, XGBoost is also known as a '**regularized boosting**' technique.

Parallel Processing:

XGBoost implements parallel processing and is **blazingly faster** as compared to GBM.
But hang on, we know that boosting is a sequential process so how can it be parallelized? We know that each tree can be built only after the previous one, so what stops us from making a tree using all cores?

In XGBoost, the parallelisation happens during the construction of each trees, at a very low level. Each independent branches of the tree are trained separately. Hyperparameters tuning requires many branches per tree and many trees per model and several models per hyperparameters value and many hyperparameters values to be tested…



Each blue square represent a core

2 parallelisation strategies for hyperparameter Tuning

XGBoost Advantages

High Flexibility

XGBoost allows users to define **custom optimization objectives and evaluation criteria**.

Handling Missing Values

XGBoost has an in-built routine to handle missing values.

The user is required to supply a different value than other observations and pass that as a parameter. XGBoost tries different things as it encounters a missing value on each node and learns which path to take for missing values in future.

Tree Pruning:

A GBM would stop splitting a node when it encounters a negative loss in the split. Thus it is more of a **greedy algorithm**.

XGBoost on the other hand make **splits up to the max_depth** specified and then start **pruning** the tree backwards and remove splits beyond which there is no positive gain.

Another advantage is that sometimes a split of negative loss say -2 may be followed by a split of positive loss +10. GBM would stop as it encounters -2. But XGBoost will go deeper and it will see a combined effect of +8 of the split and keep both.

XGBoost Advantages

Built-in Cross-Validation

XGBoost allows user to run a **cross-validation at each iteration** of the boosting process and thus it is easy to get the exact optimum number of boosting iterations in a single run.

This is unlike GBM where we have to run a grid-search and only a limited values can be tested.

Continue on Existing Model

User can start training an XGBoost model from its last iteration of previous run. This can be of significant advantage in certain specific applications.