Tree-Based Methods

Szu-Chi Chung

Department of Applied Mathematics, National Sun Yat-sen University

Tree-based Methods

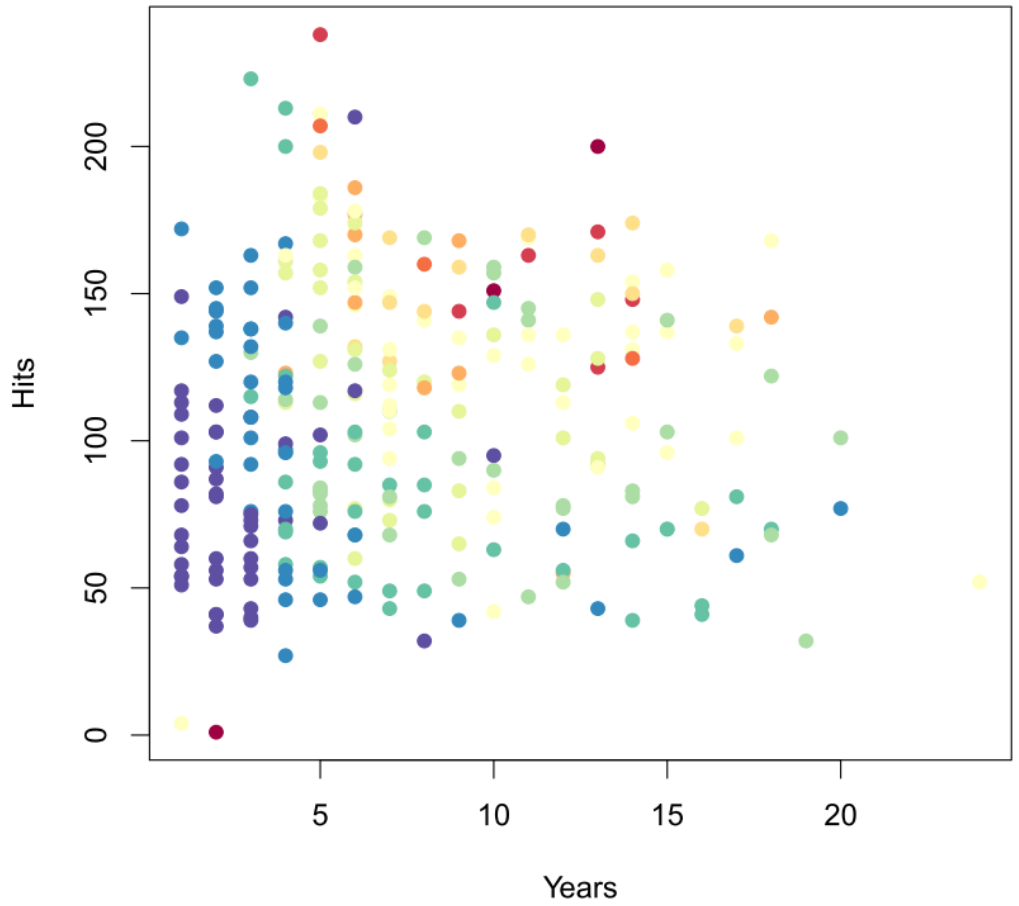
- Here we describe tree-based methods for regression and classification
  - Firstly *stratifying* or *segmenting* the predictor space into a number of simple regions
  - Use the mean or the mode response value for the training observations in the region to which it belongs for inference
- Since the set of splitting rules used to segment the predictor space can be summarized in a tree, these types of approaches are known as decision-tree methods

Pros and Cons

- ✓ Tree-based methods are simple and useful for interpretation
- x However, they typically are not competitive with the best supervised learning approaches in terms of prediction accuracy
- Hence we also discuss
  - bagging
  - random forests
  - boosting
- These methods grow multiple trees which are then combined to yield a single *consensus prediction*
  - Combining a large number of trees can often result in dramatic improvements in prediction accuracy, at the expense of some loss interpretation

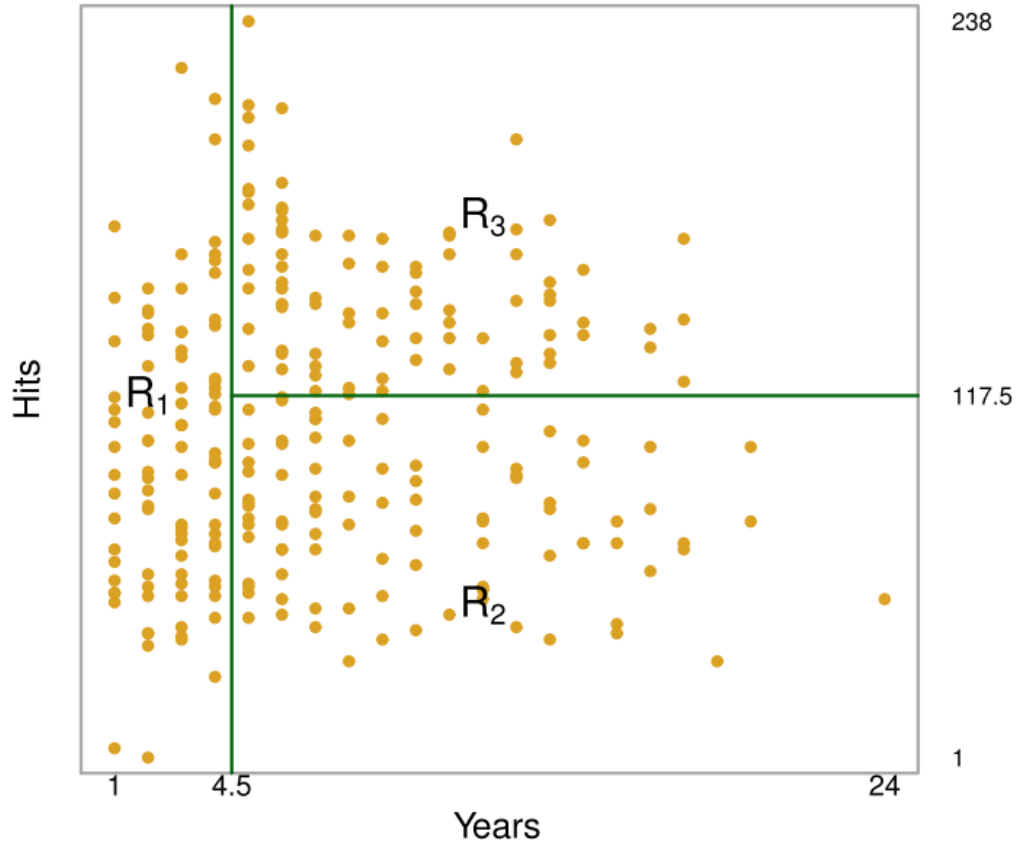
Baseball salary data: how would you stratify it?

- We first remove missing data and *log-transform* `Salary` in the Hitters dataset
  - `Salary` is color-coded from low (blue, green) to high (yellow, red)



A taste of decision tree

- Overall, the tree stratifies or segments the players into three regions of predictor space:
- $R_1 = \{ X \mid Years < 4.5 \}$
- $R_2 = \{ X \mid Years \geq 4.5, Hits < 117.5 \}$
- $R_3 = \{ X \mid Years \geq 4.5, Hits \geq 117.5 \}$



Decision tree for these data

Details of previous figure

- For the Hitters data, a regression tree for predicting the log `salary` of a baseball player, based on the number of `years` that he has played in the major leagues and the number of `hits` that he made in the previous year
  - At a given *internal node*, the label (of the form $X_j < t_k$) indicates the left-hand branch emanating from that split, and the right-hand branch corresponds to $X_j \geq t_k$. For instance, the split at the top of the tree results in two large branches. The left-hand branch corresponds to `Years`< 4.5, and the right-hand branch corresponds to `Years` ≥ 4.5
  - The tree has two internal nodes and three *terminal nodes*, or *leaves*. The number in each leaf is the *mean* of the response for the observations that fall there

Details of previous figure

- In keeping with the tree analogy, the regions R_1, R_2, and R_3 are also known as *terminal nodes*
  - Decision trees are typically drawn upside down, in the sense that the leaves are at the bottom of the tree
  - The points along the tree where the predictor space is split are referred to as *internal nodes*
  - In the hitters tree, the two internal nodes are indicated by the text $Years < 4.5$ and $Hits < 117.5$

Interpretation

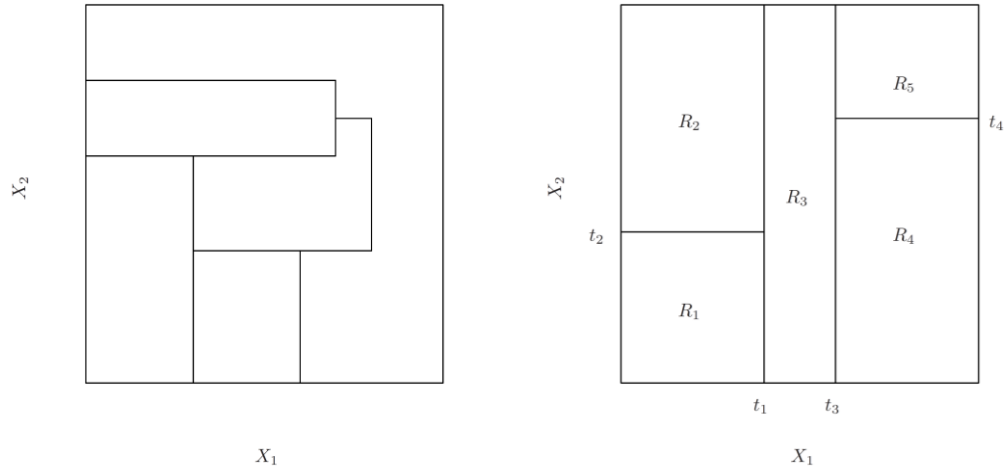
1. `Years` is the most important factor in determining `Salary`, and players with less experience earn lower salaries than more experienced players
2. *Given* that a player is less experienced, the number of `Hits` that he made in the previous year seems to play little role in his `Salary`
3. But among players who have been in the major leagues for five or more years, the number of `Hits` made in the previous year does affect `Salary`, and players who made more `Hits` last year tend to have higher salaries

- Surely an over-simplification, but compared to a regression model, it is easy to display, interpret and explain

Details of the tree-building process

1. We divide the predictor space — that is, the set of possible values for $X_1, X_2, \ldots, X_p$— into J distinct and non-overlapping regions, $R_1, R_2, \ldots, R_J$
2. For every observation that falls into the region R_j , we make the same prediction, which is simply the mean of the response values for the training observations in R_j

- How do we construct the regions $R_1, R_2, \ldots, R_J$?
  - In theory, the regions could have any shape!
  - However, we choose to divide the predictor space into high-dimensional *rectangles*, or *boxes*, for simplicity and for ease of interpretation of the resulting predictive model



Details of the tree-building process – Step 1

- The goal is to find boxes $R_1, R_2, \ldots, R_J$ that minimize the RSS, given by

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

  $\hat{y}_{R_j}$ is the mean response for the training observations within the jth box

- Unfortunately, it is still *computationally infeasible* to consider every possible partition of the feature space into J boxes
  1. We take a top-down, *greedy* approach that is known as *recursive binary splitting*
  2. The approach is *top-down* because it begins at the top of the tree and then successively splits the predictor space
  3. It is *greedy* because, at each step, the best split is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step

Details of the tree-building process – Step 1

- We first select the predictor X_j and the cutpoint s such that splitting the predictor space into the regions $\{X|X_j < s\}$ and $\{X|X_j \geq s\}$ leads to the greatest possible reduction in RSS (choosing j and s)

$$\sum_{i:x_i \epsilon R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \epsilon R_2(j,s)} (y_i - \hat{y}_{R_2})^2$$

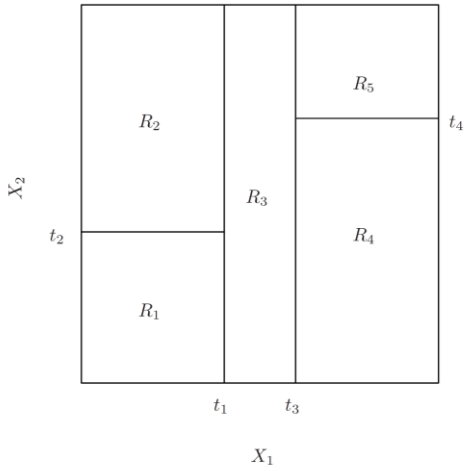
- Next, we look for the best predictor and best cutpoint in order to split the data *further* so as to minimize the RSS within each of the resulting regions
  - Instead of splitting the entire predictor space, we split *one of the two* previously identified regions. We now have three regions
  - Again, we look to split one of these three regions further, so as to minimize the RSS
  - The process continues *until a stopping criterion* is reached; for instance, we may continue until no region contains more than five observations

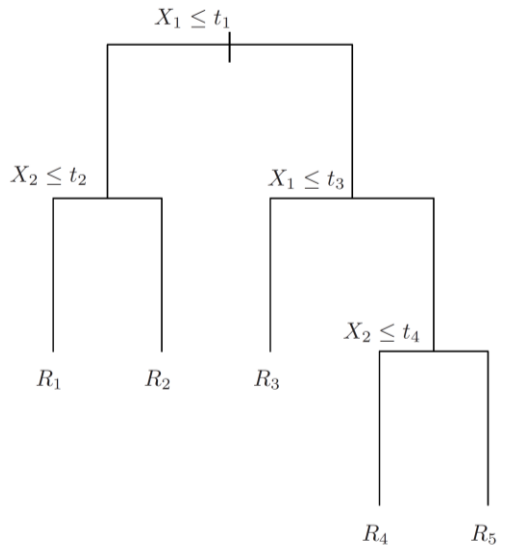
Details of the tree-building process – Step 1

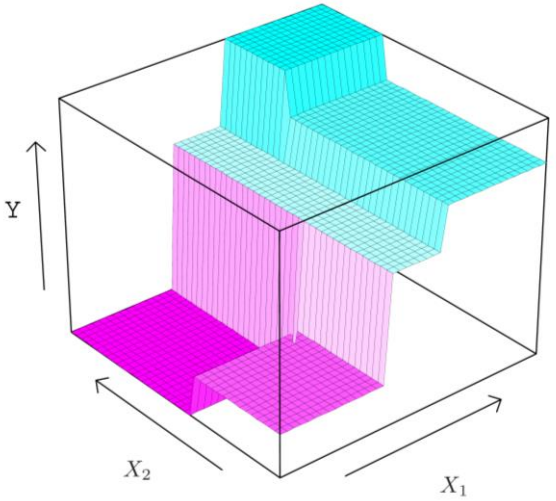
- If the predictor is *quantitative*, this means considering all possible thresholds for splitting
  - The threshold value is drawn from the sorted observation
- If the predictor is *categorical*, this means considering all ways to split the categories into two groups
  - We may rank the categories according to the average value of the target variable for observations in each category (label encoding) and split them like quantitative variables
  - If the target is qualitative, we may use this strategy (All combinations or one versus rest)

Predictions

- A five-region example of this approach is shown
- We predict the response for a given test observation using the mean of the training observations in the region to which that test observation belongs







Regularization

- The process described above may produce good predictions on the training set, but is likely to *overfit* the data, leading to poor test set performance
- A *smaller tree* with fewer splits (that is, fewer regions $R_1, R_2, \ldots, R_J$) might lead to lower variance and better interpretation at the cost of a little bias
  - A simple way to limit a tree's size is to directly regulate its depth, the size of its terminal nodes (training data belongs to them), or both
  - One possible alternative to the process described above is to grow the tree only so long as the decrease in the RSS due to each split exceeds some threshold
  - These strategy will result in smaller trees, but is too short-sighted: a seemingly worthless split early on in the tree might be followed by a very good split — that is, a split that leads to a large reduction in RSS later on

Pruning a tree

- A better strategy is to grow a very large tree T_0, and then prune it back in order to obtain a subtree
  - Again, considering all possible subtrees is not practical! *Cost complexity pruning* — also known as *weakest link pruning* — is used to do this
  - We consider a sequence of trees indexed by a nonnegative tuning parameter α. For each value of α, there corresponds a subtree $T \subset T_0$ such that

$$\sum_{m=1}^{|T|} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|$$

  is as small as possible
  - Here $|T|$ indicates the number of terminal nodes of the tree T, R_m is the rectangle (i.e. the subset of predictor space) corresponding to the mth terminal node, and $\hat{y}_{R_m}$ is the mean of the training observations in R_m

Choosing the best subtree

- The tuning parameter α controls a trade-off between the subtree's complexity and its fit to the training data
- Note that a similar formulation was used in order to control the complexity of a linear model when we discuss *lasso*!
- It turns out that as we increase α from zero, branches get pruned from the tree in a *nested and predictable fashion*!
  - We select an optimal value $\hat{\alpha}$ using cross-validation
  - We then return to the full data set and obtain the subtree corresponding to $\hat{\alpha}$

Algorithm 8.1 *Building a Regression Tree*

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.

2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.

3. Use K-fold cross-validation to choose α. That is, divide the training observations into K folds. For each $k = 1, \ldots, K$:

   (a) Repeat Steps 1 and 2 on all but the kth fold of the training data.

   (b) Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of α.

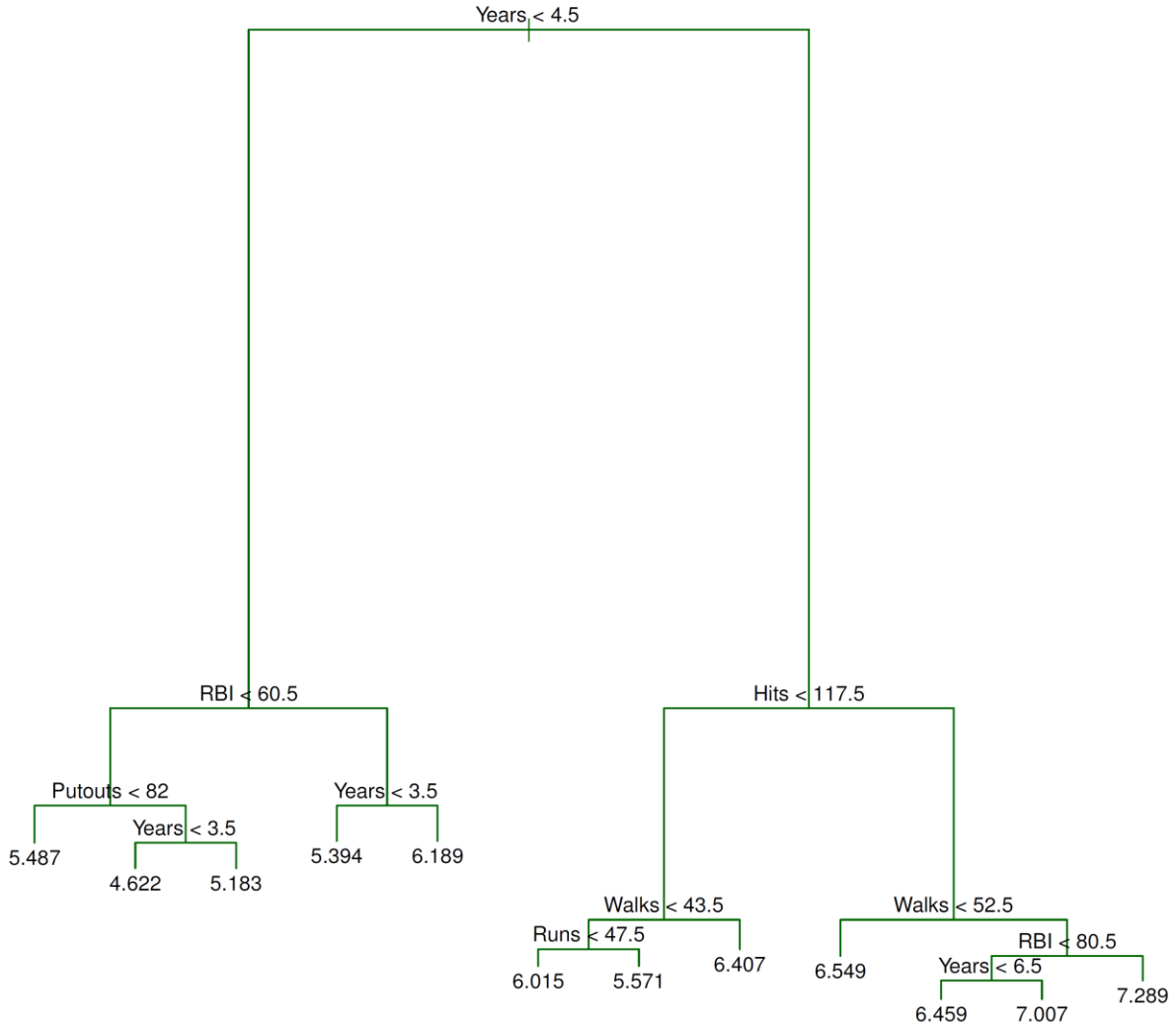
   Average the results for each value of α, and pick α to minimize the average error.

4. Return the subtree from Step 2 that corresponds to the chosen value of α.

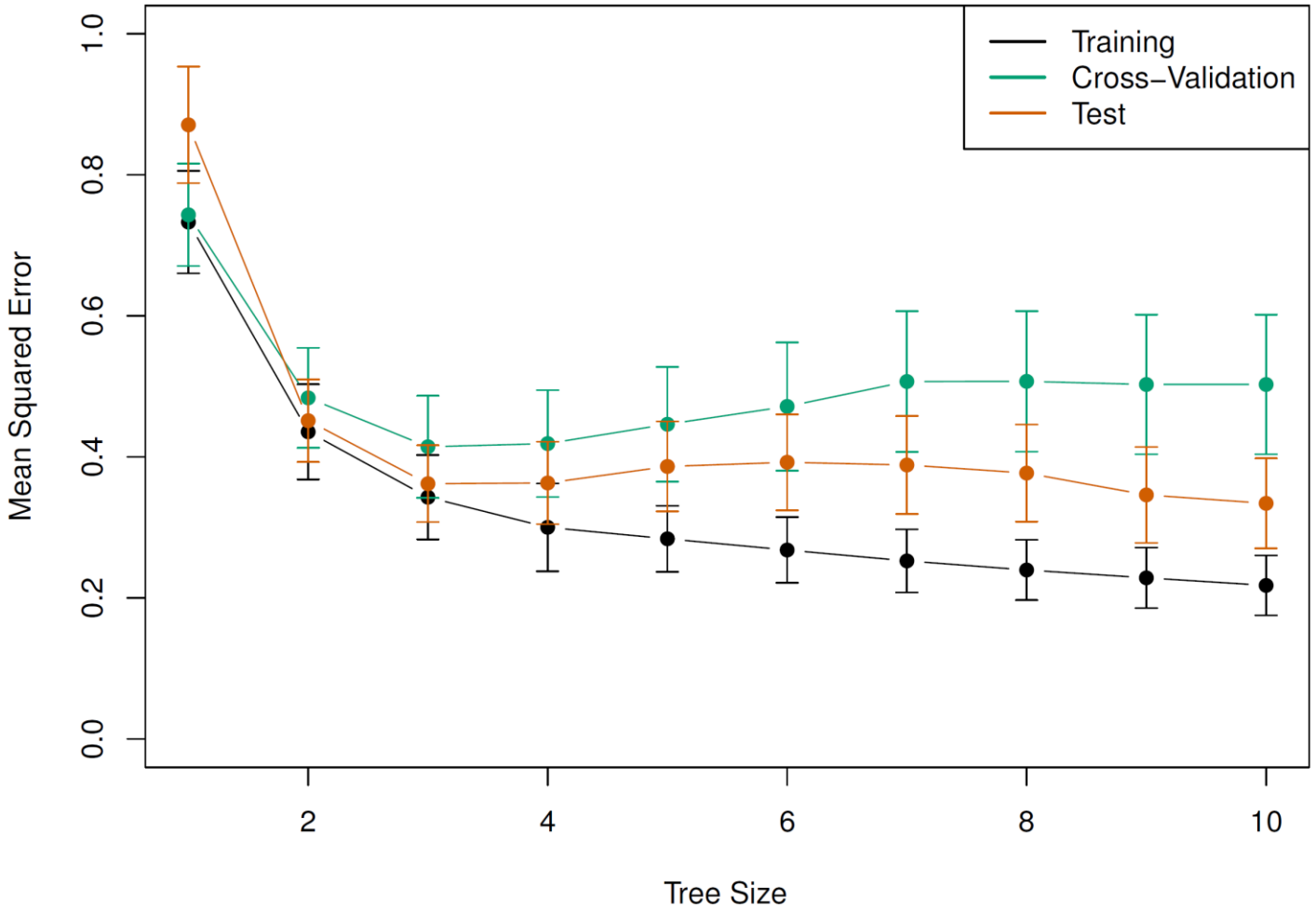
Baseball example revisit

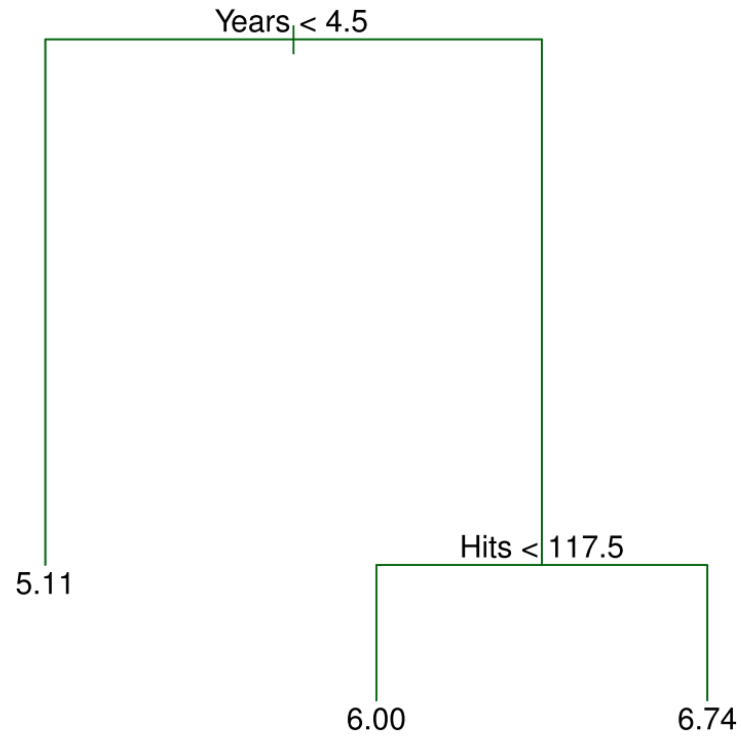
1. First, we randomly divided the data set in half, yielding 132 observations in the training set and 131 observations in the test set
2. We then built a large regression tree on the training data and varied α in order to create subtrees with different numbers of terminal nodes
3. Finally, we performed six-fold cross-validation in order to estimate the cross-validated MSE of the trees as a function of α

Baseball example revisit



Baseball example continued





Classification Trees

- Very similar to a regression tree, except that it is used to predict a qualitative response rather than a quantitative one
  - For a classification tree, we predict that each observation belongs to the most commonly occurring class of training observations in the region to which it belongs
    - The proportions among the training observations that fall into that region also matter!
  - We also use recursive binary splitting to grow a classification tree
  - In the classification setting, *RSS* cannot be used as a criterion for making the binary splits

Details of classification trees

- A natural alternative to RSS is the *classification error rate*
  - This is simply the fraction of the training observations in that region that do not belong to the most common class

$$E = \frac{1}{N_m} \sum_{i \in R_m} I(y_i \neq \hat{y}_i) = 1 - \max_k \hat{p}_{mk}$$

  Where $\hat{y}_i = arg \max_k \hat{p}_{mk}$ and $\hat{p}_{mk}$ represents the proportion of training observations in the mth region that are from the kth class

- However classification error is not sufficiently *sensitive for tree-growing*, and in practice two other measures are preferable

Gini index and cross-entropy

- The *Gini index* for a specific node is defined by

$$G = 1 - \sum_{i=1}^{K} \hat{p}_{mk}{}^2 = \sum_{i=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

  - The Gini index takes on a small value if all of the $\hat{p}_{mk}$'s are close to zero or one
  - For this reason, the Gini index is referred to as a measure of node *purity* — a small value indicates that a node contains predominantly observations from a single class

- An alternative to the Gini index is *cross-entropy*, given by

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$$

- It turns out that the Gini index and the cross-entropy are very similar numerically and both *differentiable*

Gini index and cross-entropy

- In order to evaluate the purity of a split (rather than that of a node), we use the *weighted Gini index* or *weighted cross-entropy*
  - Consider a split of node which creates children R_m^L and R_m^R
  - Let the fraction of training observations going to R_m^L be N_L and the fraction going to R_m^R be N_R. The weighted loss (whether with the Gini index or the cross-entropy) is defined as

$$L = N_L G(R_m^L) + N_R G(R_m^R)$$

or

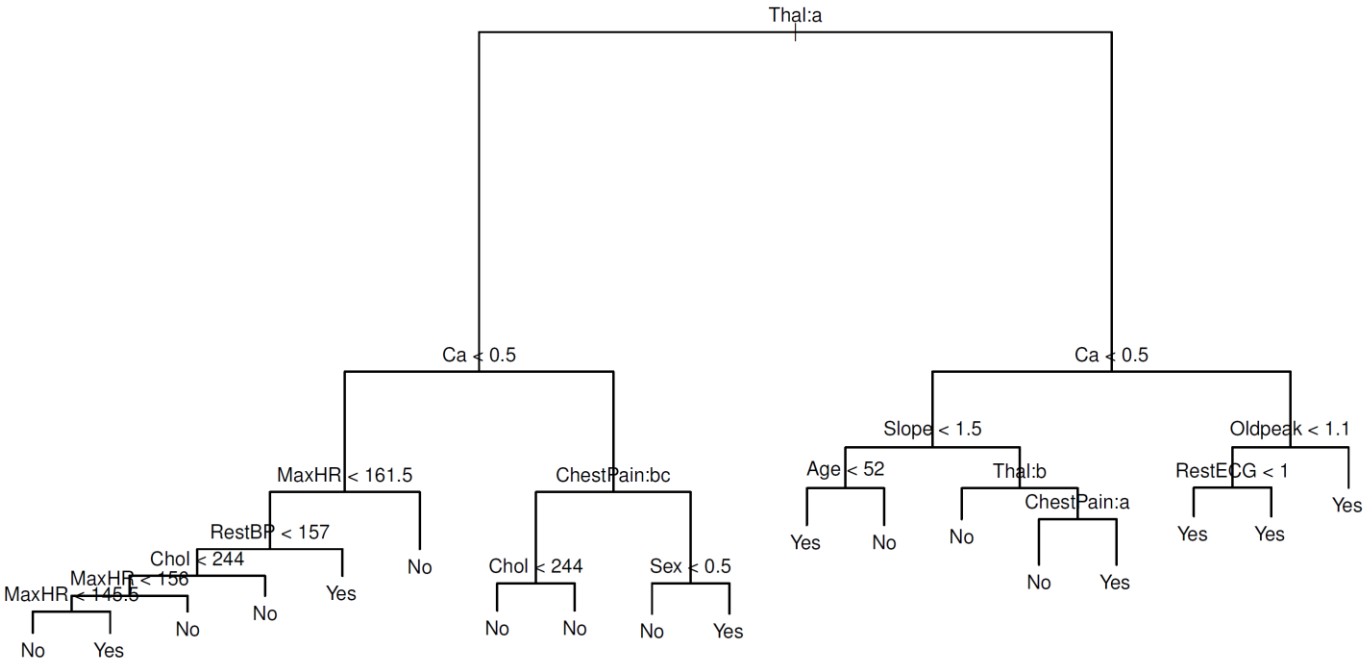
$$L = N_L C(R_m^L) + N_R C(R_m^R)$$

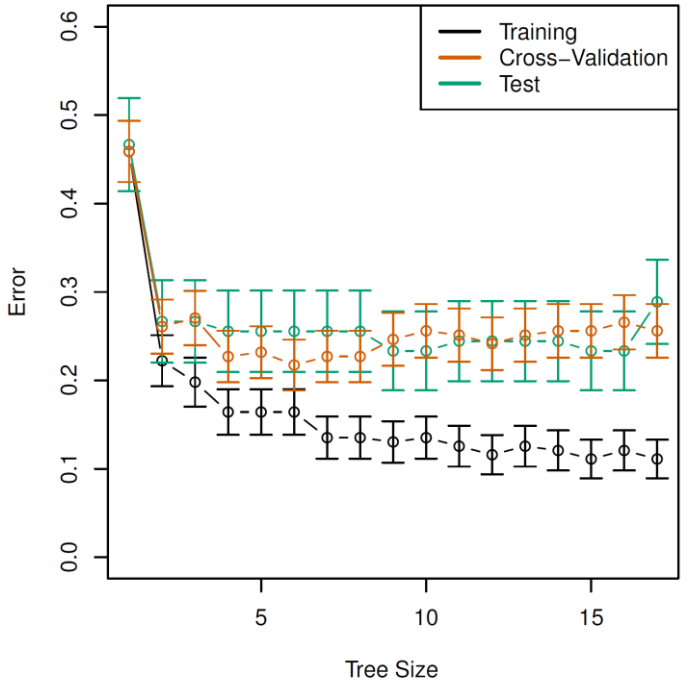
Example: Heart data

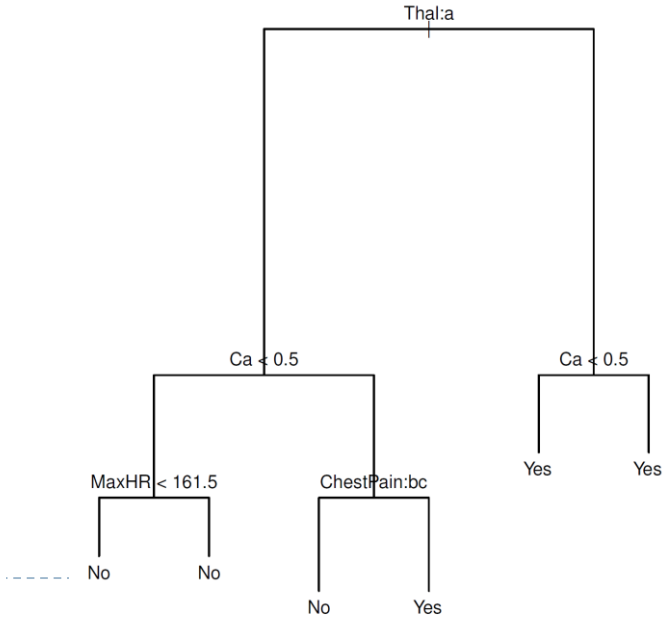
- These data contain a binary outcome `HD` for 303 patients who presented with chest pain
- An outcome value of `Yes` indicates the presence of heart disease based on an angiographic test, while `No` means no heart disease. There are 13 predictors including `Age`, `Sex`, `Chol` (a cholesterol measurement), and other heart and lung function measurements
- Cross-validation yields a tree with six terminal nodes. See next figure

- There are some qualitative predictors
- Some of the splits yield two terminal nodes that have the same predicted value
  - Though the split $RestECG < 1$ does not reduce the classification error, it improves the Gini index and the entropy, which are more sensitive to node purity

(weighted by the observation in each subtree)





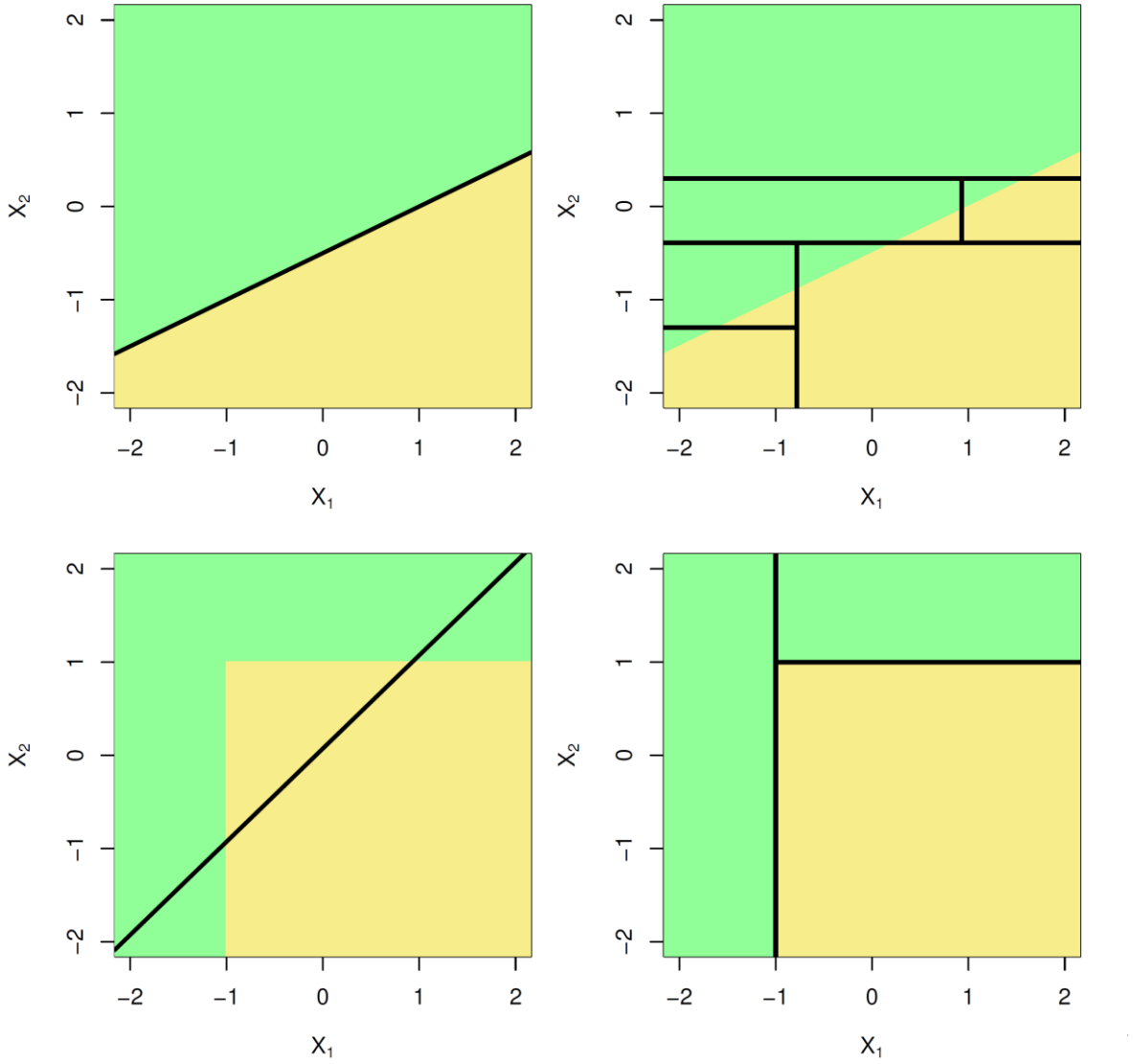


Trees Versus Linear Models

- Regression tree assume a model of a form

$$f(X) = \sum_{m=1}^{M} c_m 1_{(X \in R_m)}$$

- Top Row: True linear boundary; Bottom row: true non-linear boundary
- Left column: linear model; Right column: tree-based model



Advantages and Disadvantages of Trees

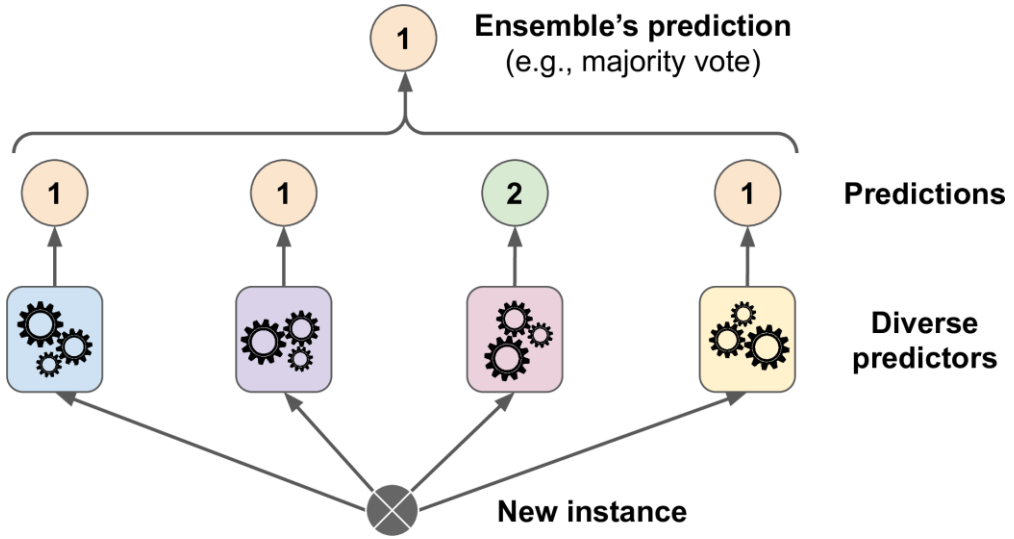
- Pros
  - ✓ Trees are very easy to explain to people. In fact, they are even easier to explain than linear regression!
  - ✓ Some people believe that decision trees more closely mirror human decision-making than do the regression and classification approaches seen in previous chapters
  - ✓ Trees can be displayed graphically, and are easily interpreted even by a non-expert (especially if they are small)
  - ✓ Trees can easily handle qualitative predictors without the need to create dummy variables
- Cons
  - x Unfortunately, trees generally do not have the same level of predictive accuracy as some of the other regression and classification approaches seen in this book
- However, by *aggregating* many decision trees, the predictive performance of trees can be substantially improved. We introduce these concepts next

Why and when to use ensemble learning?

- Suppose you pose a complex question to thousands of random people and then aggregate their answers. In many cases, you will find aggregated answer is better than an expert's answer. This is called the *wisdom of the crowd*
  - If you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor
  - A group of predictors is called an *ensemble*; thus, this technique is called Ensemble Learning, and an Ensemble Learning algorithm is called an Ensemble method
    - You will often use Ensemble methods *near the end of a project*, once you have already built a few good predictors, to combine them into an even better predictor
  - In fact, the winning solutions in Machine Learning competitions often involve several Ensemble methods

Simple example - Voting classifier

- Suppose you train a few classifiers, each one achieving about 80% accuracy
  - A very simple way to create an even better classifier is to aggregate the predictions of each classifier and predict the class that gets the *most votes*. This majority-vote classifier is called a hard voting classifier
  - If you build an ensemble containing 1,000 classifiers that are *weak learners* and individually correct only 51% of the time. If you predict the majority voted class, you can hope for up to 75% accuracy!
    - Only true if all classifiers are perfectly independent and make uncorrelated errors!
    - One way to get diverse classifiers is to train them using *very different algorithms*



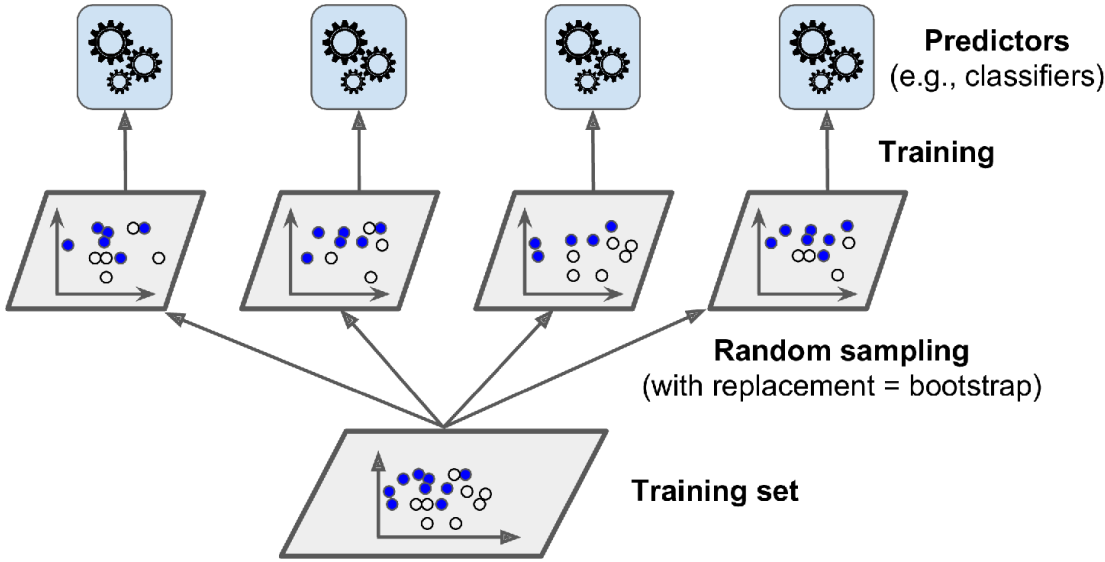
Ensemble method - Bagging

- Another approach is to use the same training algorithm for every predictor but train them on a different dataset
- Recall that given a set of n independent observations $Z_1, \ldots, Z_n$, each with variance σ^2, the variance of the mean $\bar{Z}$ of the observations is given by σ^2/n
- In other words, averaging a set of observations reduces variance. Of course, this is not practical because we generally do not have access to multiple training sets
  - *Bootstrap aggregation*, or *bagging*, is a general-purpose procedure for reducing the variance of a statistical learning method

Bagging— continued

- We can bootstrap, by taking repeated samples from the (single) training data set
  - In this approach, we generate B different bootstrapped training data sets
  - We then train our method on the bth bootstrapped training set in order to get $\hat{f}^{*b}(x)$, the prediction at a point x. We then average all the predictions to obtain

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x)$$

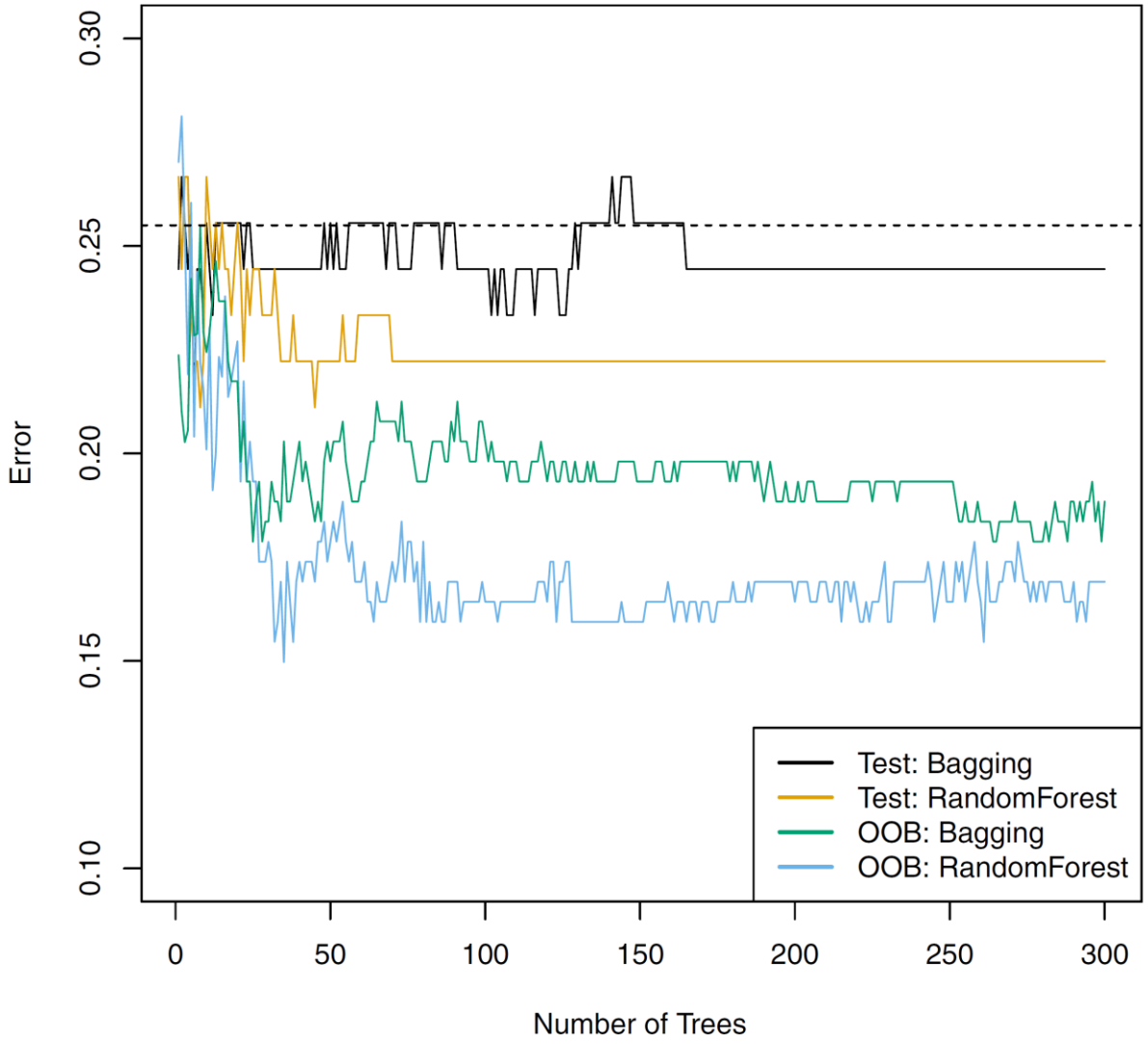


Bagging classification trees

- The above prescription applied to *regression trees*
  - These trees are grown deep, and are not pruned. Hence each individual tree has high variance, but low bias. For classification, we take a majority vote among the B predictions
  - It scales well because the predictors can all be trained in parallel and the predictions can be made in parallel, too
  - *Feature sampling* is also possible, which is called random subspaces methods
    - When combining both, it is called random patches

Bagging the heart data

- The number of trees B is not a critical parameter with bagging; using a very large value of B will not lead to overfitting
- In practice, we use a value of B *sufficiently large* that the error has settled down



Details of previous figure

- Bagging and random forest results for the Heart data.
  - The dashed line indicates the test error resulting from a single classification tree. The test error (black and orange) is shown as a function of B, the number of bootstrapped training sets used
  - Random forests were applied with $m = \sqrt{p}$
  - The green and blue traces show the OOB error, which in this case is considerably lower by chance

Out-of-Bag Error Estimation

- It turns out that there is a very straightforward way to estimate the test error of a bagged model
  - Recall that the key to bagging is that trees are repeatedly fit to bootstrapped subsets of the observations. One can show that, on average, each bagged tree makes use of around two-thirds of the observations (Exercise 2 of Chapter 5)
  - The remaining one-third of the observations not used to fit a given bagged tree are referred to as *out-of-bag (OOB)* observations
  - We can predict the response for the ith observation using each of the trees in which that observation was OOB. This will yield around $B/3$ predictions for the ith observation, which we average (or vote)
  - This estimate is essentially the LOO cross-validation error for bagging, if B is large (ESL, exercise 15.2)

Ensemble method - Random Forests

- Random forests provide an improvement over bagged trees by way of a small tweak that *decorrelates* the trees. This reduces the variance when we average
  - As in bagging, we build a number of decision trees on bootstrapped training samples
  - But when building these decision trees, each time a split in a tree is considered, a random selection of m predictors is chosen as split candidates from the full set of p predictors
    - The split is allowed to use only one of those m predictors!
    - A *fresh selection* of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$ — that is, the number of predictors considered at each split is approximately equal to the square root of the total number of predictors (4 out of the 13 for the Heart data)

Ensemble method - Random Forests

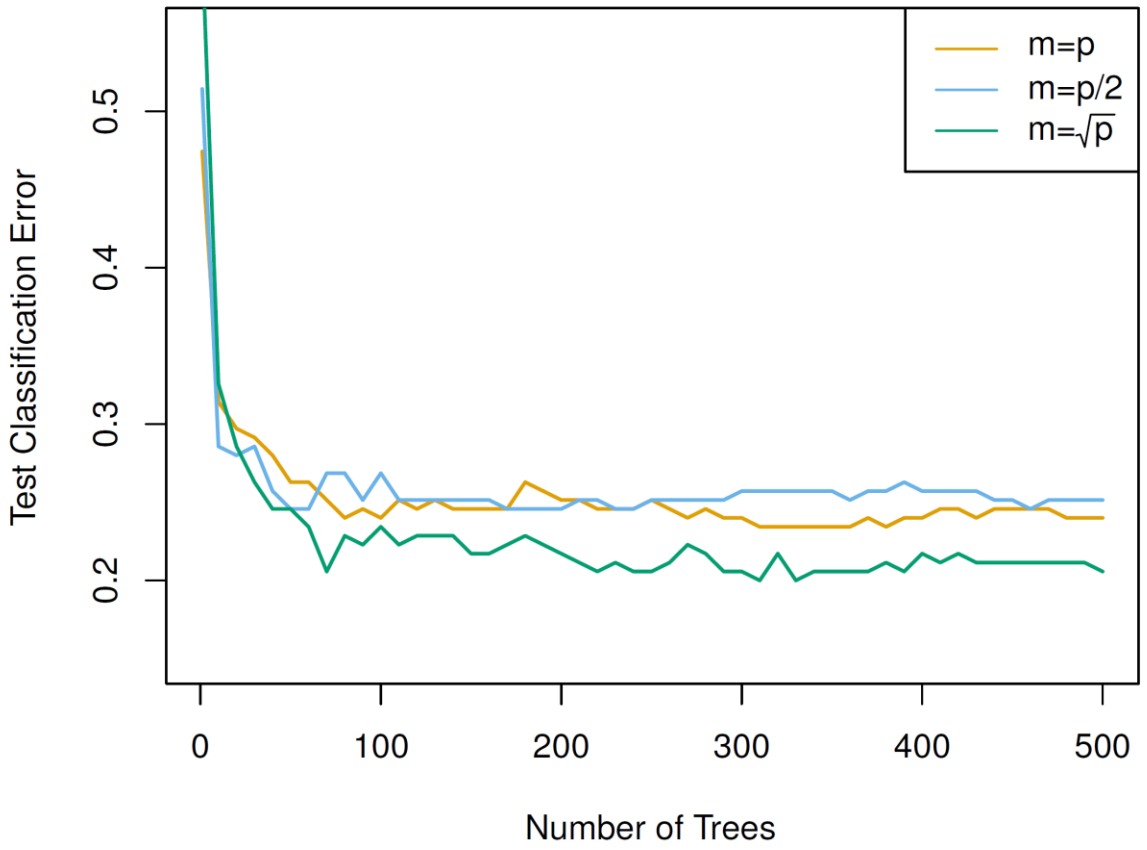
- Suppose that there is one *very strong predictor* in the data set. Then, in the collection of bagged trees, most or all of the trees will use this strong predictor in the top split
  - Consequently, all of the bagged trees will look quite similar to each other
  - Random forests overcome this problem by forcing each split to consider *only a subset* of the predictors
  - Using a small value of m in building a random forest will typically be helpful when we have a large number of strongly correlated predictors
  - C.f. Extra tree

Example: gene expression data

- We applied random forests to a high-dimensional biological data set consisting of expression measurements of 4,718 genes measured on tissue samples from 349 patients
  - There are around 20,000 genes in humans, and individual genes have different levels of activity, or expression, in particular cells, tissues, and biological conditions
  - Each of the patient samples has a qualitative label with 15 different levels: either normal or one of 14 different types of cancer
- Preprocessing
  - We use random forests to predict cancer type based on the 500 genes that have the largest variance in the training set
  - We randomly divided the observations into a training and a test set, and applied random forests to the training set for three different values of the number of splitting variables m

Results: gene expression data

- As with bagging, random forests will not overfit if we increase B, so in practice we use a value of B sufficiently large for the error rate to have settled down



Details of previous figure

- Results from random forests for the fifteen-class gene expression data set with $p = 500$ predictors
- The test error is displayed as a function of the number of trees. Each colored line corresponds to a different value of m, the number of predictors available for splitting at each interior tree node
- Random forests ($m < p$) lead to a slight improvement over bagging ($m = p$). Note that a single classification tree has an error rate of 45.7%

Ensemble method - Boosting

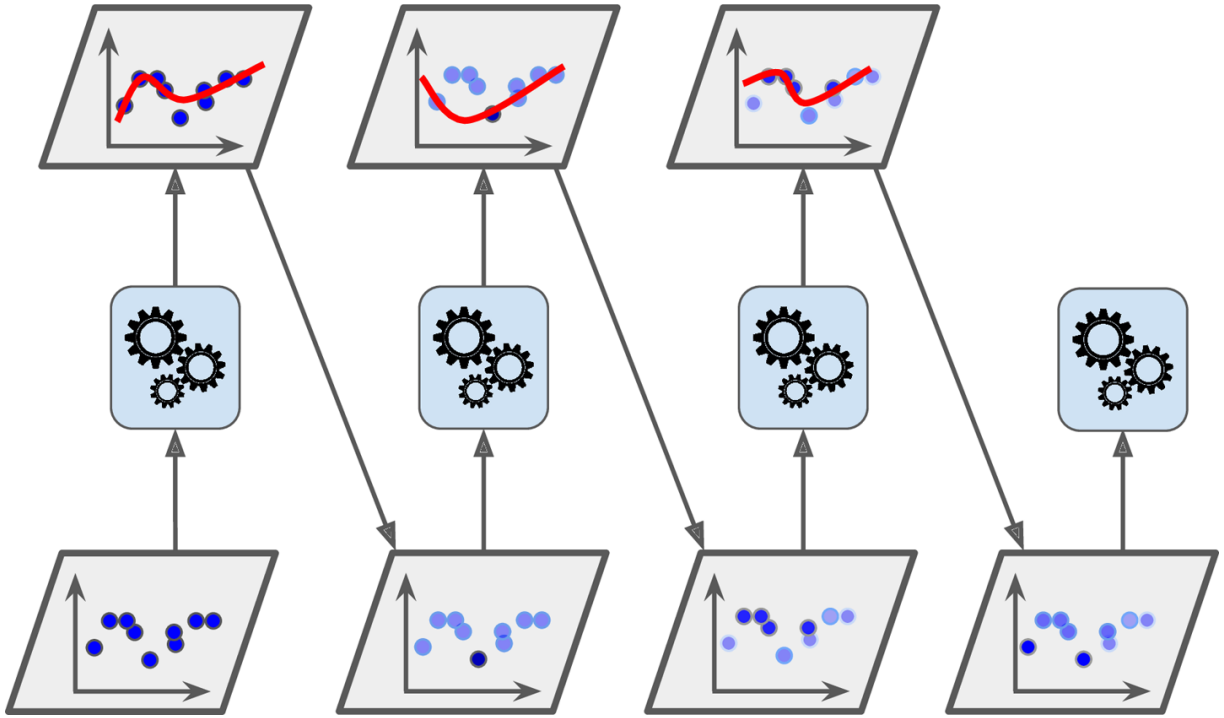
- Like bagging, *boosting* is a general approach that can be applied to many statistical learning methods for regression or classification

- Recall that bagging involves creating multiple copies of the original training data set using the bootstrap, fitting a separate decision tree to each copy, and then combining all of the trees in order to create a single predictive model
  - Notably, each tree is built on a bootstrap data set, *independent* of the other trees

- Boosting works in a similar way, except that the trees are grown *sequentially:* each tree is grown using information from previously grown trees
  - Boosting does not involve bootstrap sampling; instead, each tree is fit on a *modified version* of the original data set

Ensemble method - AdaBoost

- One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted
  - This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost
  - The algorithm increases the relative weight of *misclassified* training instances. Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on

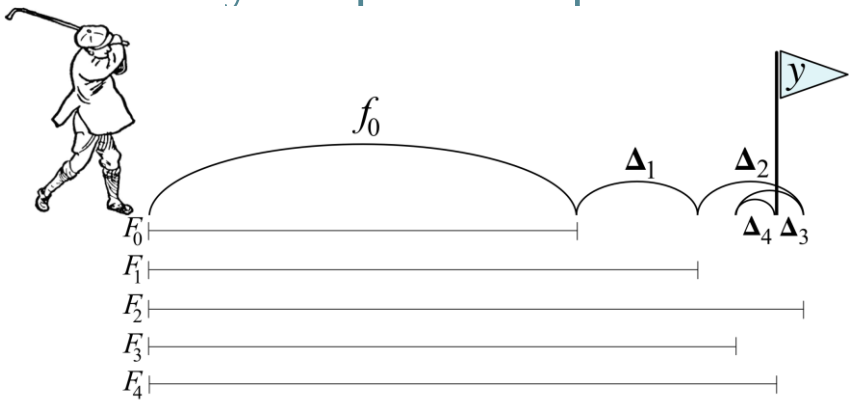


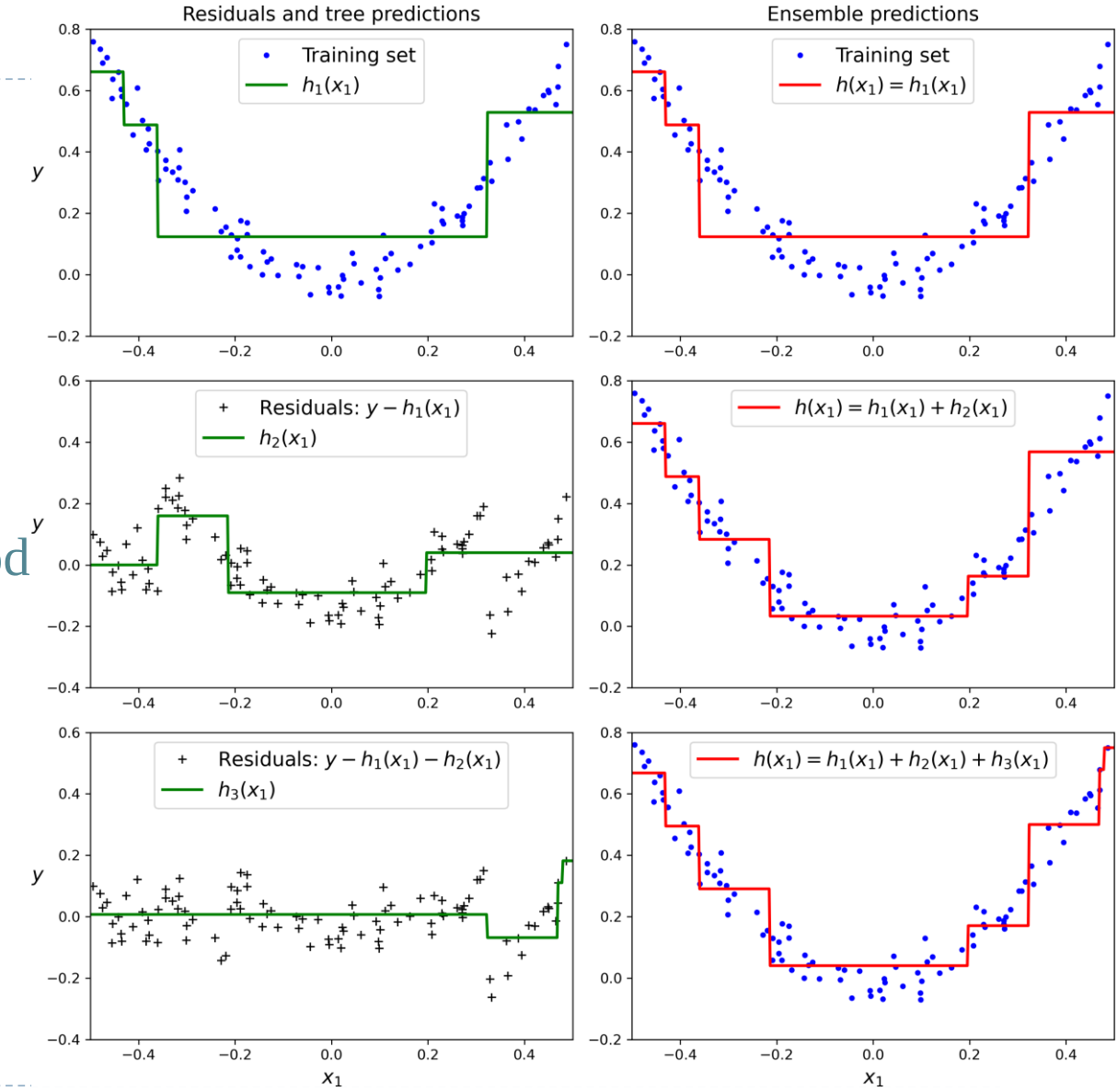
$$\alpha_j = \eta \log \frac{1 - e_j}{e_j} \qquad w_{j+1}(i) = \begin{cases} w_j(i) \times e^{-\alpha_j} \ if h_j(x_i) = y_i \\ w_j(i) \times e^{\alpha_j} \ if h_j(x_i) \neq y_i \end{cases}$$

$$\hat{y} = argmax_k \sum_{j=1, h_j(x)=k}^{B} \alpha_j$$

Gradient Boosting

- A very popular boosting algorithm is *Gradient Boosting*
  - Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor
  - Instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to *fit the new predictor to the residual errors* made by the previous predictor





https://sefiks.com/2018/10/04/a-step-by-step-gradient-boosting-decision-tree-example/

Gradient Boosting algorithm for regression trees (GBDT/GBRT)

Algorithm 8.2 *Boosting for Regression Trees*

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.

2. For $b = 1, 2, \ldots, B$, repeat:

   (a) Fit a tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X, r).

   (b) Update $\hat{f}$ by adding in a shrunken version of the new tree:

   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \tag{8.10}$$

   (c) Update the residuals,

   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \tag{8.11}$$

3. Output the boosted model,

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \tag{8.12}$$

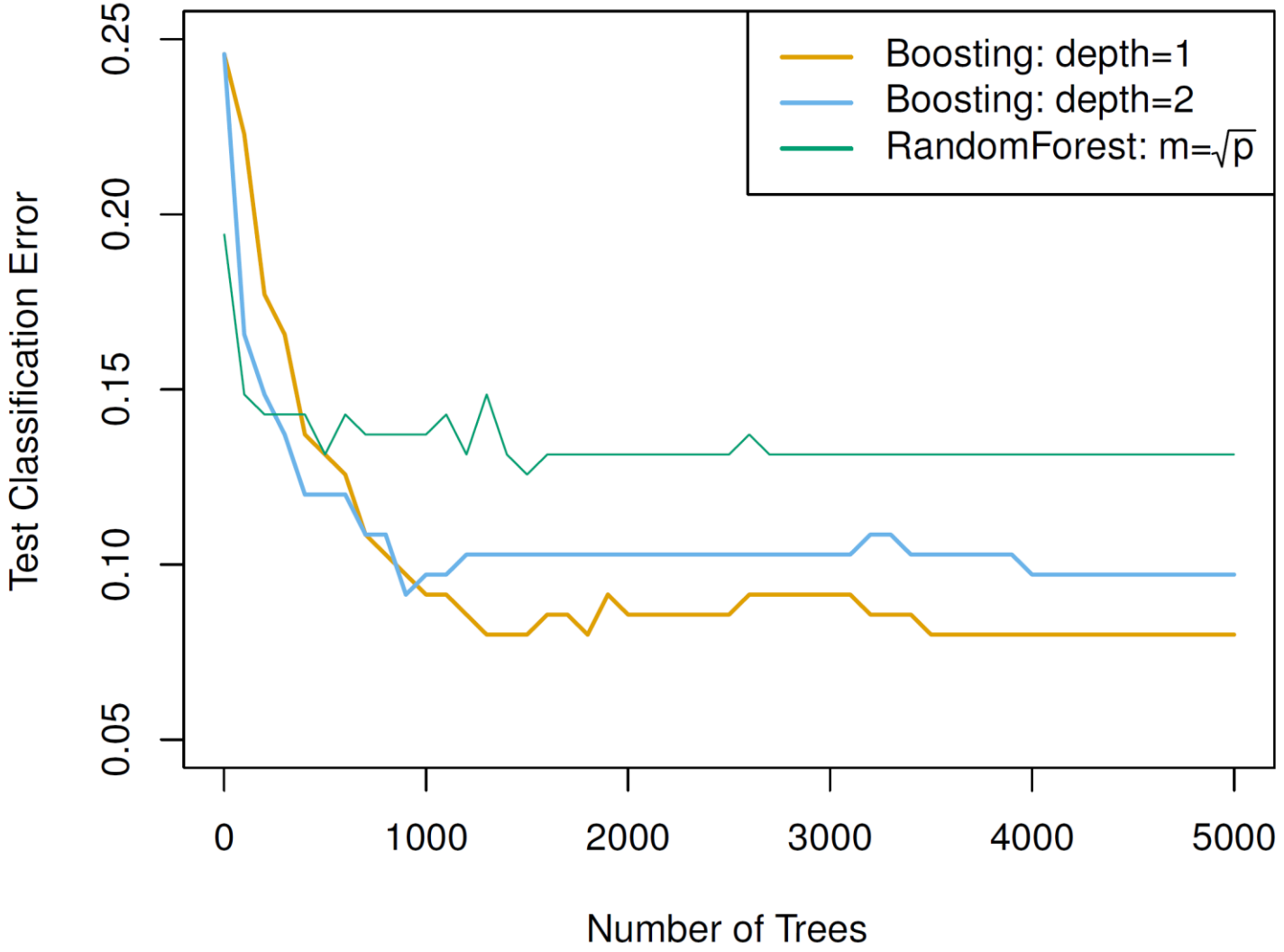
What is the idea behind this procedure?

- Unlike fitting a single large decision tree to the data, which amounts to fitting the data and potentially overfitting, the boosting approach instead *learns slowly*
  - Given the current model, we fit a decision tree to the *residuals (gradient)* from the model. We then add this new decision tree into the fitted function in order to update the residuals
  - Each of these trees can be *rather small*, with just a few terminal nodes, determined by the parameter d in the algorithm
  - By fitting small trees to the residuals, we slowly improve $\hat{f}$ in areas where it does not perform well. The shrinkage parameter λ slows the process down even further, allowing more and different shaped trees to attack the residuals

Gradient Boosting

- Gradient Boosting has three tuning parameters
  - The number of trees B (Boosting can overfit therefore choose this by CV)
  - The shrinkage parameter λ (Typical values are 0.01 or 0.001)
  - The number d of splits in each tree (Often $d = 1$ works well, in which case each tree is a stump, consisting of a single split which leads to an additive model) is the *interaction depth*
- Gradient Boosting for classification is similar in spirit to boosting for regression, but is a bit more complex (ESL ch10)
- The Python package XGboost (gradient boosted models) handles a variety of regression and classification problems

Gene expression data continued

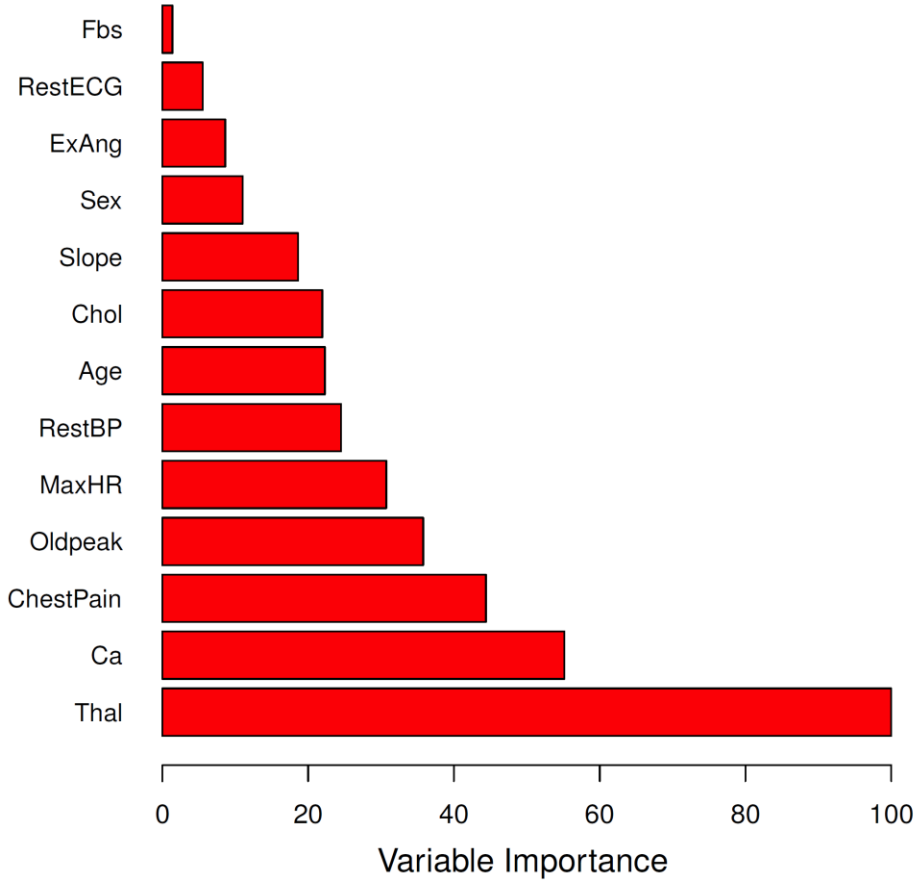


Details of previous figure

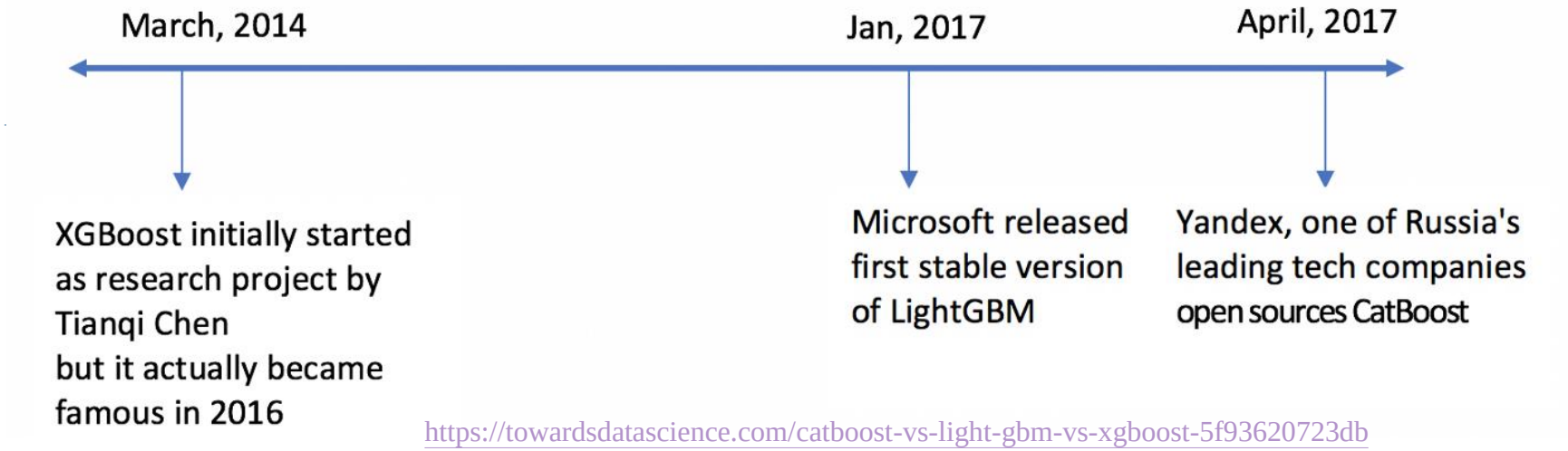
- Results from performing boosting and random forests on the fifteen-class gene expression data set in order to predict cancer versus normal (Binary classification)
  - The test error is displayed as a function of the number of trees. For the two boosted models, $\lambda = 0.01$. Depth-1 trees slightly outperform depth-2 trees, and both outperform the random forest
  - The test error rate for a single tree is 24%
  - In boosting, because the growth of a particular tree takes into account the other trees that have already been grown, smaller trees are typically sufficient. Using smaller trees can aid in interpretability as well; for instance, using stumps leads to an additive model (Exercise 2)

Variable importance measure

- For bagged/boosting regression trees, we record the *total amount that the RSS is decreased* due to splits over a given predictor, averaged over all *B* trees. A large value indicates an important predictor
- Similarly, for bagged/boosting classification trees, we add up the total amount that the *Gini index is decreased* by splits over a given predictor, averaged over all *B* trees



XGBoost



https://towardsdatascience.com/catboost-vs-light-gbm-vs-xgboost-5f93620723db

- XGBoost, short for *Extreme Gradient Boosting*, is a form of gradient boosting included *built-in regularization* and impressive gains in speed
  - The need for faster algorithms is evident when dealing with big data
- XGBoost or Gradient Boosting Decision Tree (GBDT)
  - Don't need to perform scaling (Only the relative size matters)
  - Don't need encoding
  - When given a missing data point, XGBoost treats missing value as a feature and scores different split options and chooses the one with the best results

XGBoost

$$\left[\sum_{i=1}^{n} L(y_i, p_i^0 + O_{value})\right] + \frac{1}{2}\lambda O_{value}^2$$

$$L(y, p_i + O_{value}) \approx L(y, p_i) + \left[\frac{d}{dp_i} L(y, p_i)\right] O_{value} + \frac{1}{2}\left[\frac{d^2}{dp_i^2} L(y, p_i)\right] O_{value}^2$$

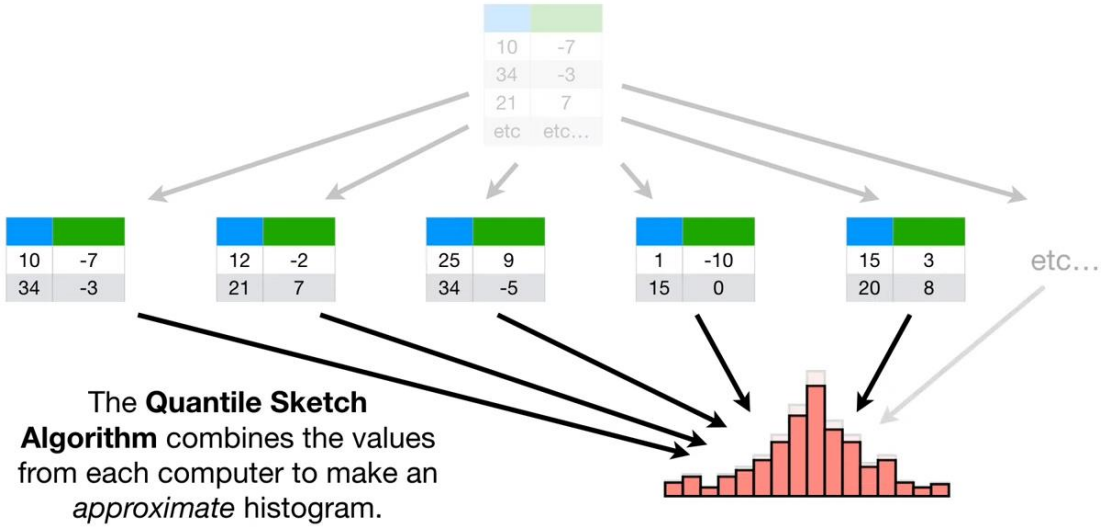
- Performance gain
  - XGBoost adds built-in regularization to achieve accuracy gains beyond gradient boosting. XGBoost is a regularized version of gradient boosting
    - For more information about the objective function, please refer to here or here
    - In addition to the regularization term, it used an approximation similar to Newton's Method which is more accurate than naïve gradient boosting. An in-depth discussion can be found here

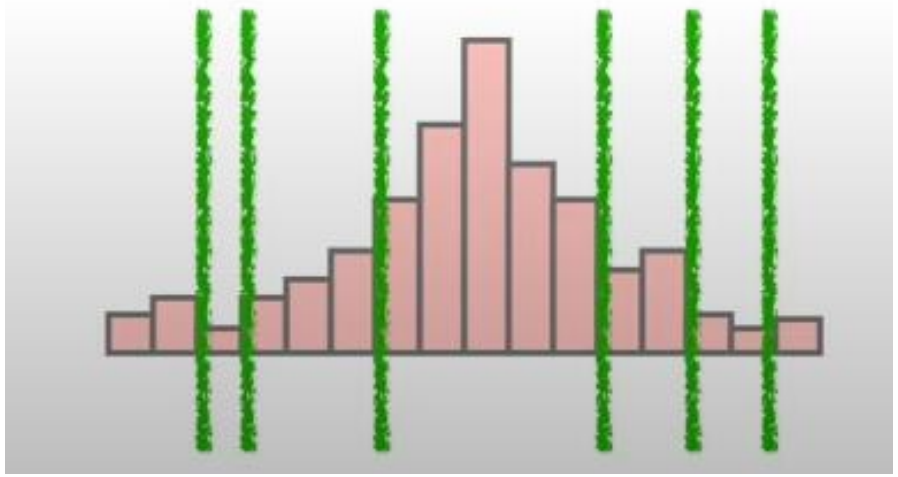
$$O_{value} = \frac{-(g_1 + g_2 + \cdots + g_n)}{(h_1 + h_2 + \cdots + h_n + \lambda)} \qquad \text{Score} = \frac{1}{2}\frac{(g_1 + g_2 + \cdots + g_n)^2}{(h_1 + h_2 + \cdots + h_n + \lambda)}$$

- Take a look at how to handle categorical variables and missing value
  - Encode categorical variable before entering the algorithm
  - Missing values can be automatically handled

XGBoost

- XGBoot presents
  - Parallel computing – Quantile sketch
  - Approximate split-finding algorithm on weighted quantile



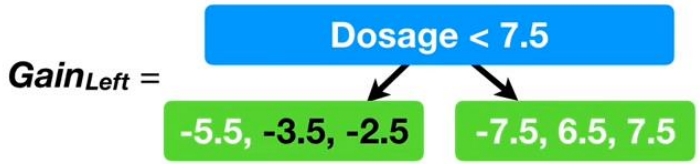


https://www.youtube.com/watch?v=oRrKeUCEbq8

XGBoost

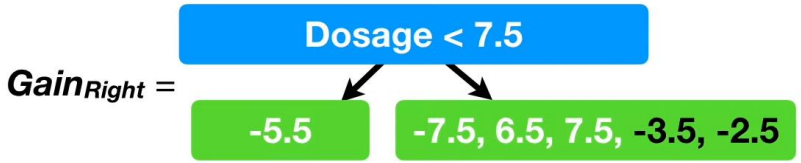
- XGBoot presents
  - Can handle sparse matrix (Sparsity-aware split finding)
  - Cache-aware access – improve cache performance (Puts gradient and hessian in it)
  - Block compression – compress data which is stored in hard disk and parallel reading
  - Random sampling on samples or features

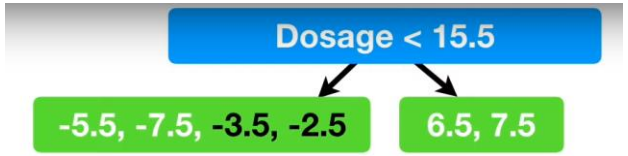
Dosage	Drug Effectiveness	Residuals
5	-5	-5.5
10	-7	-7.5
21	7	6.5
25	8	7.5



Dosage	Drug Effectiveness
???	???

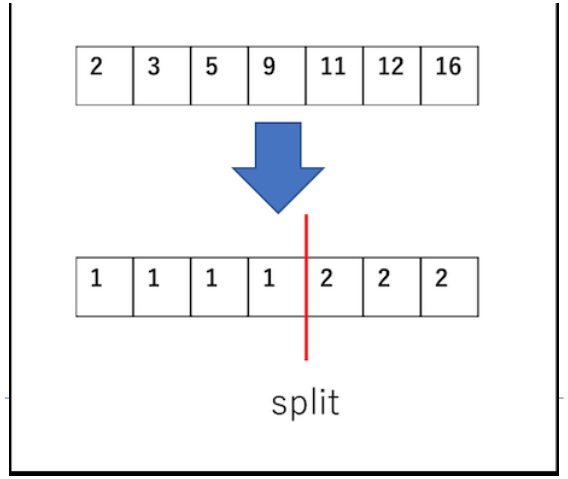
Dosage	Drug Effectiveness	Residuals
???	-3	-3.5
???	-2	-2.5





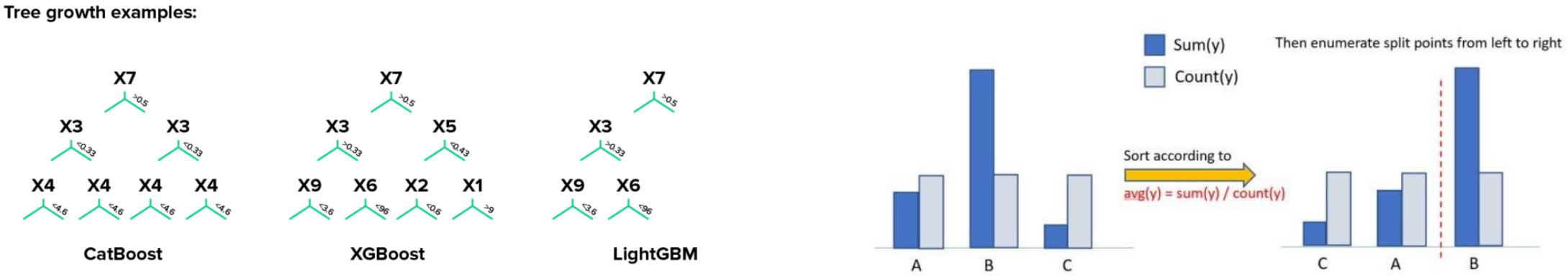
LightGBM

- For speed, use Histogram-based Gradient Boosting (HGB) and Exclusive Feature Bundling
  - It works by binning the input features, replacing them with integers. The number of bins is controlled defaults to 255 and cannot be set any higher than this. The way the bins are built ($O(n)$) removes the need for sorting ($O(nlog(n))$) the features when training each tree
    - The complexity of split a single node reduce from $O(n_{features} \times nlog(n))$ to $O(n_{features} \times n)$
  - Binning can enormously reduce the number of possible thresholds that the training algorithm needs to evaluate. Moreover, working with integers makes it possible to use faster and more *memory-efficient* data structures
  - Exclusive Feature Bundling algorithm, which can reduce the *number of features* by regrouping mutually exclusive features into a bundle (c.f. One hot vs label encoding)



LightGBM

- Optimization in accuracy
  - Gradient-based One-Side Sampling (GOSS), which adjust the sampling strategy
    - Keeps *all data instances with large gradients* and performs random sampling for data instances with small gradients. Data points with larger gradients have higher errors and would be important for finding the optimal split point
  - Leaf-wise (Best-first) tree growth instead of fixed ordered, see discussion here
  - *Optimal Split* for Categorical Features
    - Use a strategy similar to target encoding



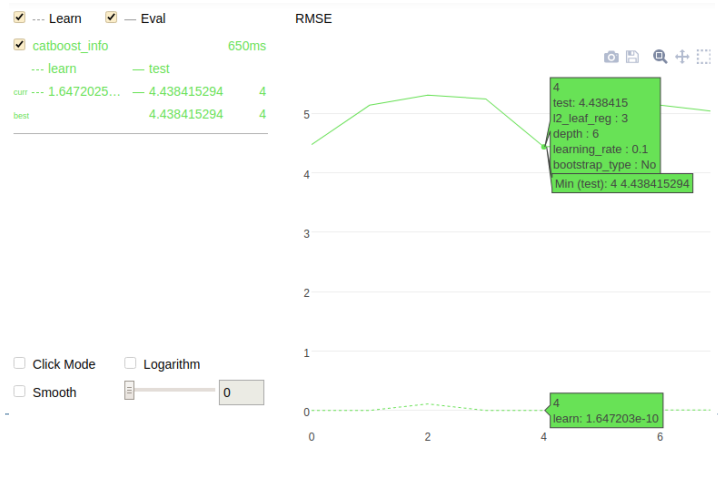
https://medium.com/riskified-technology/xgboost-lightgbm-or-catboost-which-boosting-algorithm-should-i-use-e7fda7bb36bc

CatBoost

- Symmetric trees
  - CatBoost builds symmetric (balanced) trees, unlike XGBoost and LightGBM. In every step, leaves from the previous tree are split using the same condition. The feature-split pair that accounts for the lowest loss is selected and used *for all the level's nodes*
    - This balanced tree architecture aids in efficient CPU implementation, decreases prediction time and controls overfitting as the structure serves as *regularization*
- Ordered boosting
  - When calculating the *gradient estimate* of a data instance, classic algorithms use the same data that the model was built with
  - CatBoost, on the other hand, uses the concept of ordered boosting to train the model on a subset of data while calculating residuals on another subset, thus preventing *overfitting*

CatBoost

- Sampling techniques
  - MVS can be considered a variant of the GOSS, and provide a lower variance for estimating the gradient
- CatBoost adds native supports for all kinds of features, be it numeric, categorical, or text and saves time and effort in preprocessing
  - Take a look at how to deal with categorical features here
- Visualization tools provided



Hyperparameters

1. For faster speed
   - Setting bagging fraction ratio to randomly choose instances
   - Use feature sub-sampling (random subspace) by setting the fraction of features
   - Use a smaller number of bins for Histogram-based Gradient Boosting
2. For better accuracy
   - Use a smaller learning rate with a larger number of iterations (number of estimators)
   - Use a larger number of bins for Histogram-based Gradient Boosting
   - Try different categorical encoding methods
3. Prevent overfitting
   - Use a larger value of the number of data in leaf to avoid splitting
   - Use a smaller number of depth to avoid growing deeper tree
   - Try to adjust regularization strength in the objective function
   - Use DART (Like dropout in neural network)

Hyperparameters

	XGBoost	LightGBM	CatBoost
Speed	*subsample* *colsample_bytree* *n_estimator*	*bagging_fraction* *feature_fraction* *num_iterations*	*subsample* *rsm* *iterations*
Control overfitting/accuracy	*learning_rate (0.01~0.2)* *max_depth* *min_child_weight*	*learning_rate* *max_depth, num_leaves* *min_data_in_leaf*	*learning_rate* *depth* *l2-leaf-reg*
Categorical variable	Experimental	*categorical_feature*	*cat_features* *one_hot_max_szie*

Conclusion

- In conclusion, ensemble learning is versatile, powerful, and fairly simple to use
  - Ensemble can help push your system's performance to its limits
  - Random Forests and GBDT are among the first models you should test on most Machine Learning tasks, and they particularly shine with heterogeneous tabular data. Moreover, as they require very little preprocessing, they're great to get a prototype up and running quickly
- About the choice of the framework
  - XGBoost has the largest community and provides sufficient support for production
  - LightGBM may be a better choice when considering the speed and accuracy
  - CatBoost is a choice when the dataset is small or when the categorical variables are important in the model

Appendix

The tree training algorithm

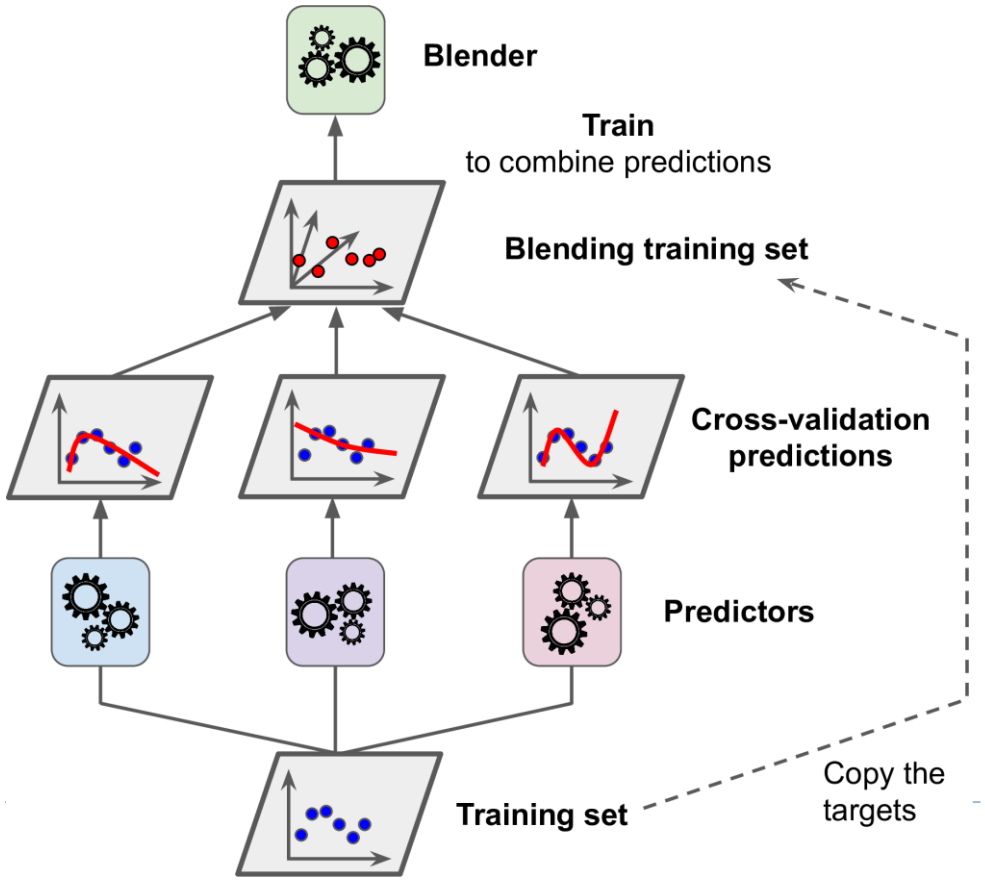
- ID3 (Iterative Dichotomiser 3) was developed in 1986 by Ross Quinlan. The algorithm creates a multiway tree, finding for each node (i.e. in a greedy manner) the categorical feature that will yield the largest information gain for categorical targets. Trees are grown to their maximum size and then a pruning step is usually applied to improve the ability of the tree to generalise to unseen data
- C4.5 is the successor to ID3 and removed the restriction that features must be categorical by dynamically defining a discrete attribute (based on numerical variables) that partitions the continuous attribute value into a discrete set of intervals. C4.5 converts the trained trees (i.e. the output of the ID3 algorithm) into sets of if-then rules. These accuracy of each rule is then evaluated to determine the order in which they should be applied. Pruning is done by removing a rule's precondition if the accuracy of the rule improves without it

The tree training algorithm

- C5.0 is Quinlan's latest version release under a proprietary license. It uses less memory and builds smaller rulesets than C4.5 while being more accurate
- CART (Classification and Regression Trees) is very similar to C4.5, but it differs in that it supports numerical target variables (regression) and does not compute rule sets. CART constructs binary trees using the feature and threshold that yield the largest information gain at each node
- scikit-learn uses an optimised version of the CART algorithm; however, scikit-learn implementation does not support categorical variables for now
  - Scikit-learn's default max $\_features = n\_features$

Other ensemble methods - Stacking

- Stacking is based on a simple idea: instead of using trivial functions (such as hard/soft voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation?
  - To train the blender (aggregator), you first need to build the blending training set
  - You can use cross-validation on every estimator in the ensemble to get out-of-sample predictions for each instance in the original training set
    - These can be used as the input features to train the blender, and the targets can be simply be copied from the original training set

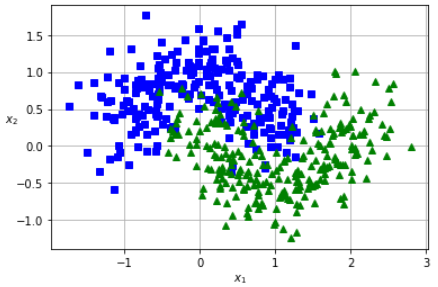


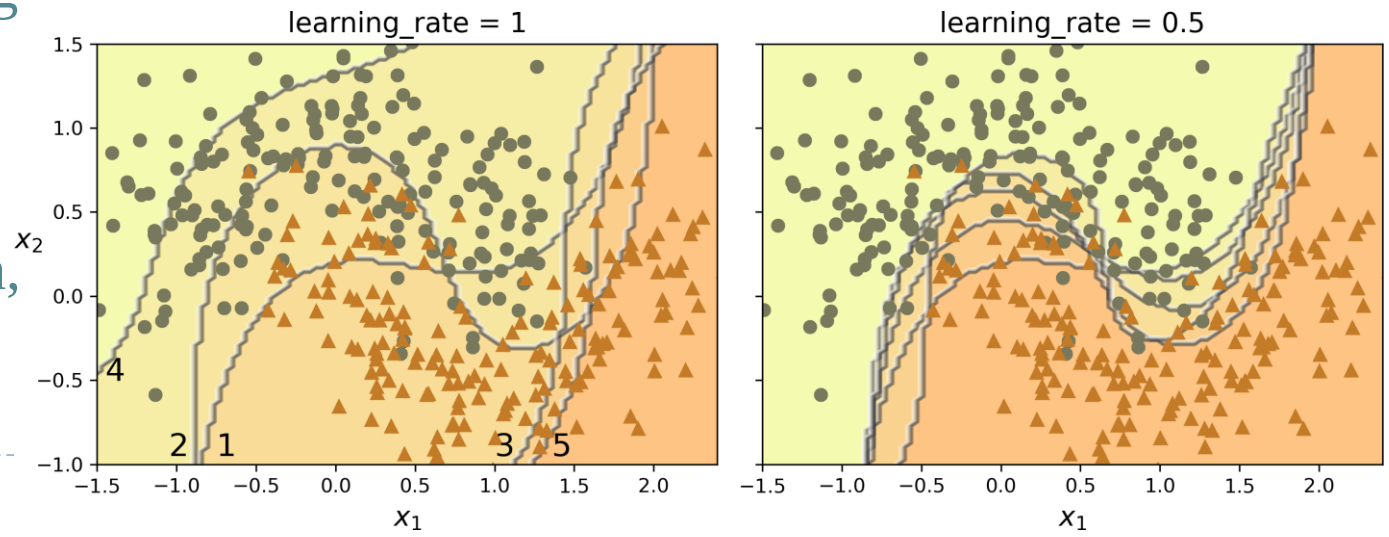
Other ensemble methods

- Extra-trees
  - In extremely randomized trees, randomness goes one step further in the way splits are computed
  - As in random forests, a random subset of candidate features is used, but instead of looking for the most discriminative thresholds, *thresholds are drawn at random* for each candidate feature and the best of these randomly-generated thresholds is picked as the splitting rule
  - This usually allows to reduce the variance of the model a bit more, at the expense of a slightly greater increase in bias

Other ensemble methods - AdaBoost

- The following shows the decision boundaries of five consecutive predictors on the moons dataset
  - The first classifier gets many instances wrong, so their weights get boosted. The second classifier therefore does a better job on these instances, and so on
  - The plot on the right represents the same sequence of predictors, except that the learning rate is halved
  - As you can see, this sequential learning technique has some similarities with Gradient Descent, except that instead of tweaking a single predictor's parameters to minimize a cost function, AdaBoost adds predictors to the ensemble, gradually making it better





ESL 10.9 – Boosting Trees

Setting	Loss Function	$-\partial L(y_i, f(x_i))/\partial f(x_i)$
Regression	$\frac{1}{2}[y_i - f(x_i)]^2$	$y_i - f(x_i)$
Regression	$\lvert y_i - f(x_i)\rvert$	$\text{sign}[y_i - f(x_i)]$
Regression	Huber	$y_i - f(x_i)$ for $\lvert y_i - f(x_i)\rvert \le \delta_m$ $\delta_m \text{sign}[y_i - f(x_i)]$ for $\lvert y_i - f(x_i)\rvert > \delta_m$ where $\delta_m = \alpha$th-quantile$\{\lvert y_i - f(x_i)\rvert\}$
Classification	Deviance	kth component: $I(y_i = \mathcal{G}_k) - p_k(x_i)$

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 $\hat{f}(x) = f_M(x)$.

Encoding categorical variable – One hot encoding

- Categorical data can be extremely useful. However, in its original form, it is unrecognizable to most models. We can use different "encoding" techniques
  - *One hot encoding* convert it to dummy variables by produces one feature per category
    - In linear and logistic regression, one hot encoding causes problems with multicollinearity. In such cases, one dummy is omitted (its value can be inferred from the other values)
    - The number of categorical features should be small so that it can be effectively applied

Animal	Target	isCat	isDog	isHamster
Cat	1	1	0	0
Hamster	0	0	0	1
Cat	0	1	0	0
Dog	1	0	1	0
Hamster	0	0	0	1
Cat	1	1	0	0
Dog	0	0	1	0

Encoding categorical variable – Label encoding

- *Ordinal encoding* or *label encoding* will transform each categorical feature to one new feature of integers (0 to number of features-1)
  - This coding suggests an *ordering*. Furthermore, it implies that the difference between cat and dog is the same as between dog and hamster

Animal	Target	Animal_encoded
Cat	1	0
Hamster	0	2
Cat	0	0
Dog	1	1
Hamster	0	2
Cat	1	0
Dog	0	1

Encoding categorical variable – Target encoding

- *Target encoding* or *mean encoding* will replace a feature's categories with some number derived from the target
  - Group the data by each category and count the number of occurrences of each target. Calculate the average of the target given each specific category and add it to a new column
  - A target encoding derives numbers for the categories using the feature's most important property: its relationship with the target

Animal	Target	Animal_encoded
Cat	1	0.67
Hamster	0	0.50
Cat	0	0.67
Dog	0	0.00
Hamster	1	0.50
Cat	1	0.67
Dog	0	0.00

Encoding categorical variable – Target encoding

- When a category only occurs a few times in the dataset, any statistics calculated on its group are unlikely to be very accurate and may leak the target
  - To avoid target leak and overfitting, target encoding need to be trained on an independent "encoding" split. You can use cross-validation in practice

		Feature	Target
Fold-1	0	A	1
	1	B	0
	2	B	0
	3	B	1
Fold-2	4	B	1
	5	A	1
	6	B	0
Fold-3	7	A	0
	8	A	0
	9	B	0
Fold-4	10	A	1
	11	A	0
	12	B	1
	13	A	0
	14	A	1
Fold-5	15	B	0
	16	B	0
	17	B	0
	18	A	1
	19	A	1

Mean_A = 5/9= 0.556
Mean_B = 2/7= 0.285

	Feature	Target	Feature_Kfold_Target_Enc
0	A	1	0.555556
1	B	0	0.285714
2	B	0	0.285714
3	B	1	0.285714
4	B	1	0.250000
5	A	1	0.625000
6	B	0	0.250000
7	A	0	0.625000
8	A	0	0.714286
9	B	0	0.333333
10	A	1	0.714286
11	A	0	0.714286
12	B	1	0.250000
13	A	0	0.625000
14	A	1	0.625000
15	B	0	0.250000
16	B	0	0.375000
17	B	0	0.375000
18	A	1	0.500000
19	A	1	0.500000

Mean_A = (.5556+.625+.625+0.714286+0.714286+0.714286+0.625+0.625+0.5+0.5)/10
Mean_A = 0.61981

	Feature
0	B
1	B
2	B
3	A
4	A

	Feature	Feature_Kfold_Target_Enc
0	B	0.294048
1	B	0.294048
2	B	0.294048
3	A	0.619841
4	A	0.619841

https://medium.com/@pouryaayria/k-fold-target-encoding-dfe9a594874b

Ensemble method - Bayesian Additive Regression Trees (BART)

- BART is related to the bagging and boosting approaches: each tree is constructed in a random manner as in bagging and random forests, and each tree tries to capture signal not yet accounted for by the current model, as in boosting
  - The main novelty in BART is the way in which new trees are generated
  - Let K denote the number of regression trees, and B the number of iterations for which the BART algorithm will be run. The notation $\hat{f}_k^b(x)$ represents the *prediction* at x for the kth regression tree used in the bth iteration
  - At the end of each iteration, the K trees from that iteration will be summed $\hat{f}^b(x) = \sum_{k=1}^{K} \hat{f}_k^b(x)$ for $b = 1, \ldots, B$

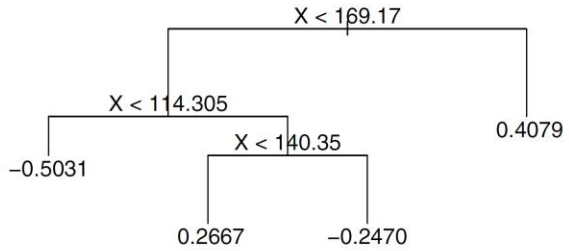
Bayesian Additive Regression Trees (BART)

- There are two components to this perturbation:
  1. We may change the structure of the tree by adding or pruning branches
  2. We may change the prediction in each terminal node of the tree
- Algorithm 8.3 can be viewed as a Markov chain Monte Carlo for fitting the BART model

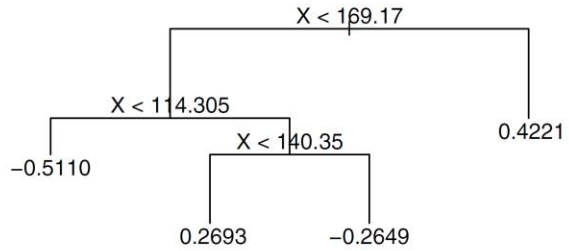
Algorithm 8.3 *Bayesian Additive Regression Trees*

1. Let $\hat{f}_1^1(x) = \hat{f}_2^1(x) = \cdots = \hat{f}_K^1(x) = \frac{1}{nK}\sum_{i=1}^n y_i$.
2. Compute $\hat{f}^1(x) = \sum_{k=1}^K \hat{f}_k^1(x) = \frac{1}{n}\sum_{i=1}^n y_i$.
3. For $b = 2, \ldots, B$:
   (a) For $k = 1, 2, \ldots, K$:
      i. For $i = 1, \ldots, n$, compute the current partial residual
      $$r_i = y_i - \sum_{k'<k} \hat{f}_{k'}^b(x_i) - \sum_{k'>k} \hat{f}_{k'}^{b-1}(x_i).$$
      ii. Fit a new tree, $\hat{f}_k^b(x)$, to r_i, by randomly perturbing the kth tree from the previous iteration, $\hat{f}_k^{b-1}(x)$. Perturbations that improve the fit are favored.
   (b) Compute $\hat{f}^b(x) = \sum_{k=1}^K \hat{f}_k^b(x)$.
4. Compute the mean after L burn-in samples,
$$\hat{f}(x) = \frac{1}{B-L}\sum_{b=L+1}^{B} \hat{f}^b(x).$$

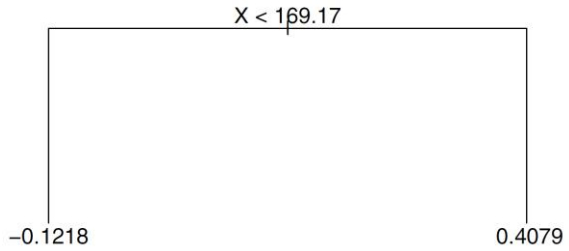
(a): $\hat{f}_k^{b-1}(X)$



(b): Possibility #1 for $\hat{f}_k^b(X)$



(c): Possibility #2 for $\hat{f}_k^b(X)$



(d): Possibility #3 for $\hat{f}_k^b(X)$

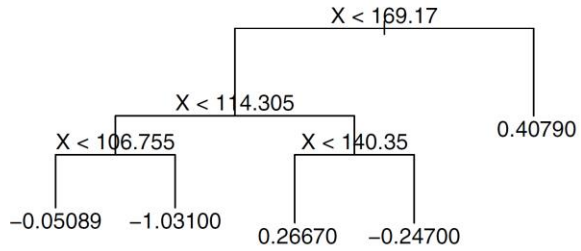


FIGURE 8.12. *A schematic of perturbed trees from the BART algorithm.* (a): *The kth tree at the* $(b-1)$*st iteration,* $\hat{f}_k^{b-1}(X)$*, is displayed. Panels (b)–(d) display three of many possibilities for* $\hat{f}_k^b(X)$*, given the form of* $\hat{f}_k^{b-1}(X)$*.* (b): *One possibility is that* $\hat{f}_k^b(X)$ *has the same structure as* $\hat{f}_k^{b-1}(X)$*, but with different predictions at the terminal nodes.* (c): *Another possibility is that* $\hat{f}_k^b(X)$ *results from pruning* $\hat{f}_k^{b-1}(X)$*.* (d): *Alternatively,* $\hat{f}_k^b(X)$ *may have more terminal nodes than* $\hat{f}_k^{b-1}(X)$*.*

Bayesian Additive Regression Trees (BART)

- We typically throw away the first few of these prediction models, since models obtained in the earlier iterations tend not to provide very good results
  - We can let L denote the number of burn-in iterations; for instance, we might take $L = 200$. Then, to obtain a single prediction, we simply take the average after the burn-in iterations, $\hat{f}(x) = \frac{1}{B-L}\sum_{b=L+1}^{B}\hat{f}^b(x)$
- A key element is that in Step 3(a)ii., we do not fit a fresh tree to the current partial residual: instead, we try to improve the fit to the current partial residual by slightly modifying the tree obtained in the previous iteration
- This guards against overfitting since it limits how "hard" we fit the data in each iteration. Furthermore, the individual trees are typically quite small. We limit the tree size in order to avoid overfitting the data, which would be more likely to occur if we grew very large trees

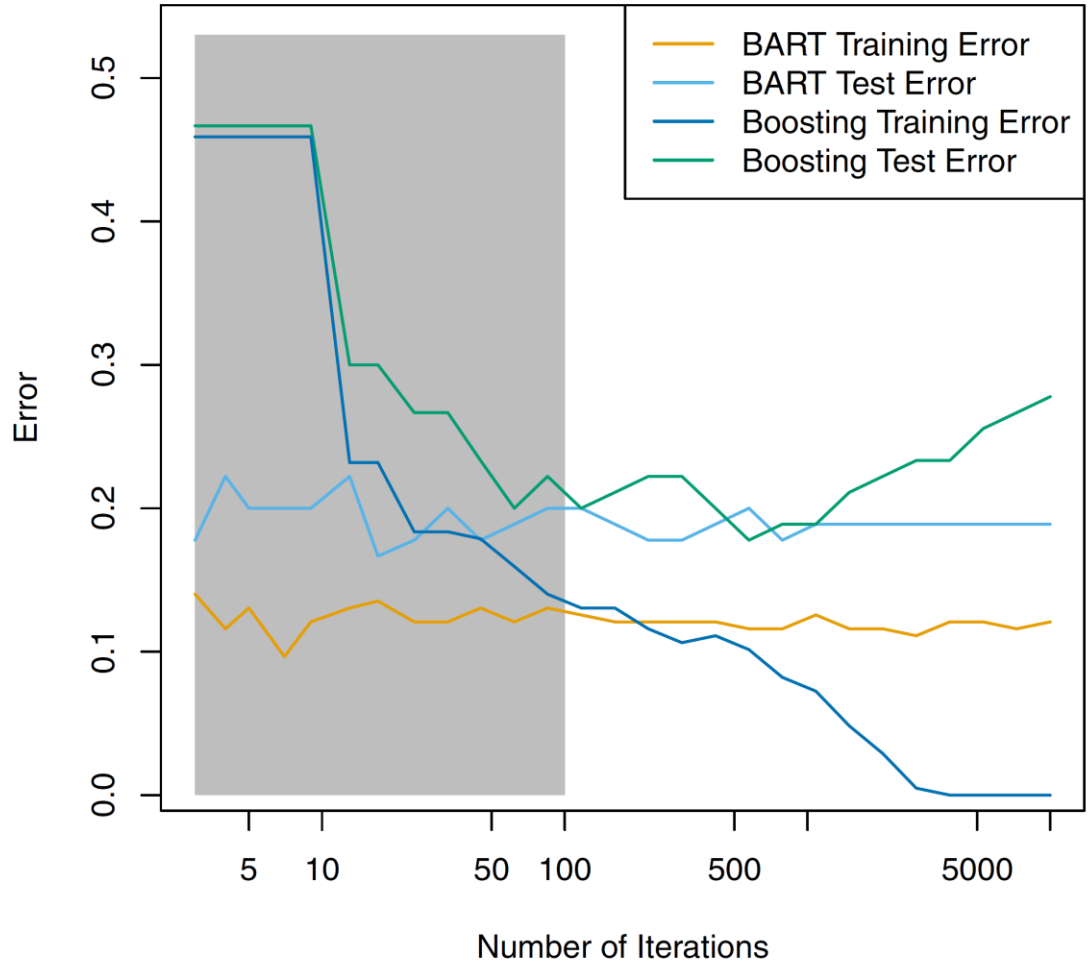


FIGURE 8.13. *BART and boosting results for the* `Heart` *data. Both training and test errors are displayed. After a burn-in period of* 100 *iterations (shown in gray), the error rates for BART settle down. Boosting begins to overfit after a few hundred iterations.*

Tuning parameters for BART

- When we apply BART, we must select the number of trees K, the number of iterations B, and the number of burn-in iterations L. We typically choose large values for B and K, and a moderate value for L
  - For instance, $K = 200$, $B = 1{,}000$, and $L = 100$ is a reasonable choice. BART has been shown to have very impressive out-of-box performance — that is, it performs well with minimal tuning

Reference

- ESL Chapter 8,9,10,15,16
- https://github.com/serengil/chefboost