Tree Based Methods

Arnab Maity (modified by Justin Post)

# Brief Recap

We’ve looked at many models for doing supervised learning. Remember we have two types of supervised learning tasks:

* Regression tasks
* Classification tasks

These models we’ve dealt with often have a version that works for the continuous type response and the categorical type response. We can have the goal of:

* making inference
* focusing on predictive ability
* or both of those

We of course have to evaluate the regression and classification models differently due to the nature of the response variables. Common metrics:

* RMSE, MSE, MAE, , many ones specific for MLR models
* (mis)classification rate, sensitivity, specificity, log-loss, etc.

We have more structured, parametric, models and less-structured, non-parametric, models. The parametric models can often be used with less observations and are generally easier to interpret but may not be flexible enough to capture patterns in the data that non-parametric models can find.

When prediction is our goal, we saw that we really care about the ability of the model to predict on a test set. As such, we’ve talked a ton about training/test splits, CV, bootstrapping, and more! In addition, we often have *tuning parameters* that we use CV or bootstrapping to find appropriate values for.

In this section we look at a non-parametric approach to modeling using what are called **trees**. These apply to both the regression and classification task and are often used as part of an ensemble learning method (via bootstrapping or other model averaging) where we combine many models together to improve prediction accuracy.

## Concepts of a Tree Model

Tree based methods are very flexible. They attempt to **split up the predictor space into regions**. On each region, a different prediction can then be made. Adjacent regions need not have predictions close to each other!

With our two different types of supervised learning situations we can create either a regression tree or classification tree!

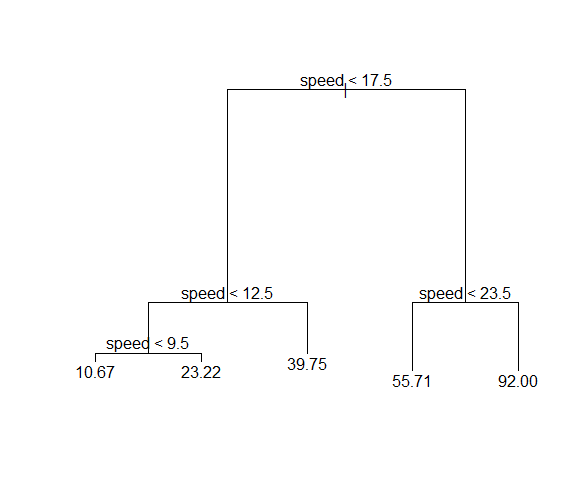
* *Classification* tree if the goal is to classify (predict) group membership
  + Often use **most prevalent class** in region as the prediction
* *Regression* tree if the goal is to predict a continuous response
  + Often use **mean of observations** in region as the prediction

These models are very easy for people to interpret! For instance, consider the tree below relating a predictor (speed) to stopping distance (dist).

library(tree) #rpart is also often used

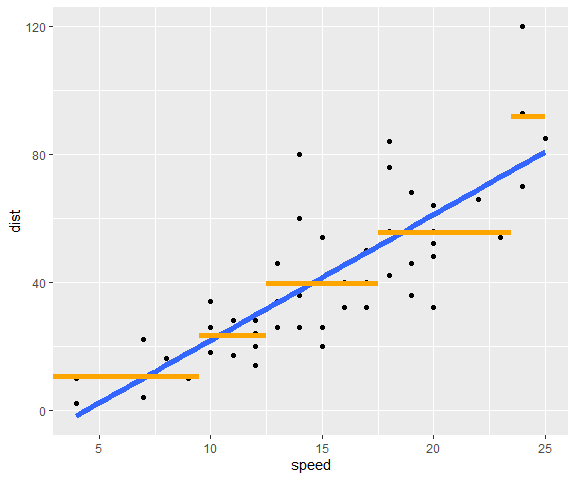
Warning: package 'tree' was built under R version 4.1.3

fitTree <- tree(dist ~ speed, data = cars) #default splitting is deviance  
plot(fitTree)  
text(fitTree)



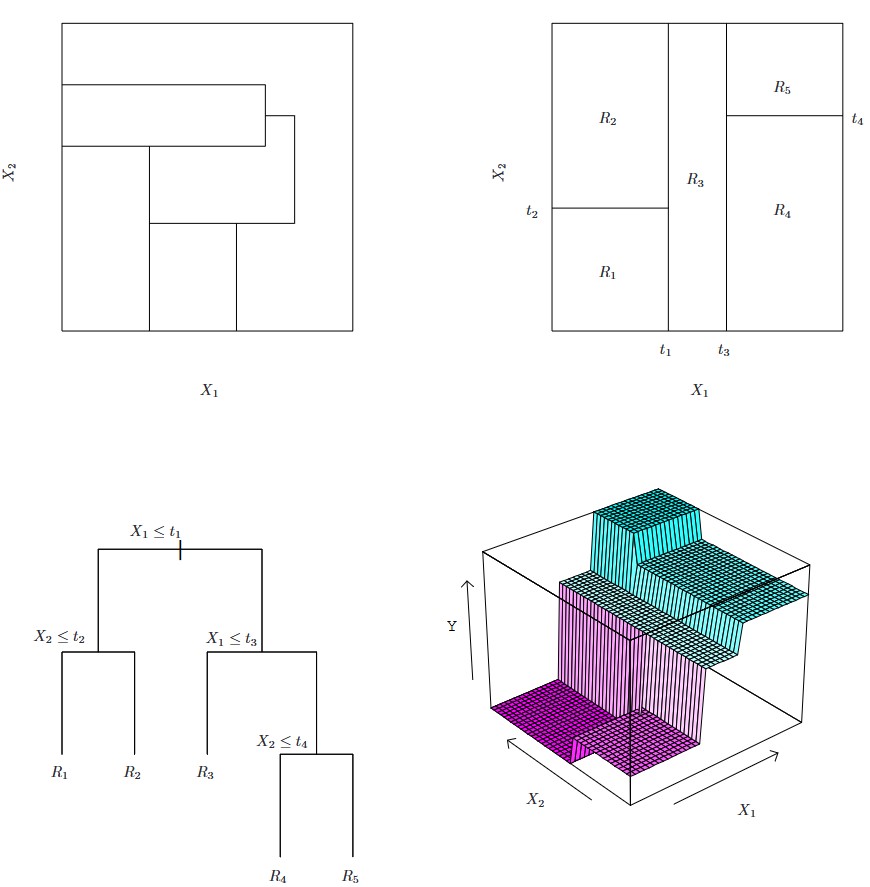
We can compare this to the simple linear regression fit to see the increased flexibility of a regression tree model.

ggplot(cars, aes(x = speed, y = dist)) +   
 geom\_point() +  
 geom\_smooth(method = "lm", se = FALSE, size = 2) +   
 geom\_segment(x = 0, xend = 9.5, y = 10.67, yend = 10.67, col = "Orange", size = 2) +  
 geom\_segment(x = 9.5, xend = 12.5, y = 23.22, yend = 23.22, col = "Orange", size = 2) +  
 geom\_segment(x = 12.5, xend = 17.5, y = 39.75, yend = 39.75, col = "Orange", size = 2) +  
 geom\_segment(x = 17.5, xend = 23.5, y = 55.71, yend = 55.71, col = "Orange", size = 2) +  
 geom\_segment(x = 23.5, xend = max(cars$speed), y = 92, yend = 92, col = "Orange", size = 2)



Suppose we have two predictors. The same idea can be used here. We can partition the predictor space into regions and have different predictions in each region.

Consider this figure from the ISLR book:



Four plots are shown in a two by two grid. Top left: A partition of two-dimensional feature space that could **not** result from recursive binary splitting as the resulting regions are not all rectangles. Top Right: The output of recursive binary splitting on a two-dimensional example. Bottom Left: A tree corresponding to the partition in the top right panel. Bottom Right: A perspective plot of the prediction surface corresponding to that tree. Figure and caption taken from .

The top right panel shows a partition of the feature space by lines that are parallel to the coordinate axes. In each partition element we can model with a different constant.

We could create a region such as that in the top left panel, but this would be difficult to interpret. 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.

To obtain our rectangles, we generally

* first split the space into two regions, and model the response by the mean of in each region.
  + We choose the variable and split-point to achieve the best ‘fit’.
* then one or both of these regions are split into two more regions
* this process is continued, until some stopping criterion is applied
  + For example, in the top right panel,
    - we first split at whole space into regions and
    - then the region is split into two according to and
    - similarly the region is split into two: and
    - finally, the region is split again at
* The corresponding regression model predicts with a constant.

Denote the 5 regions as . Since we are fitting a constant function in each region, we are modeling our regression function at any as

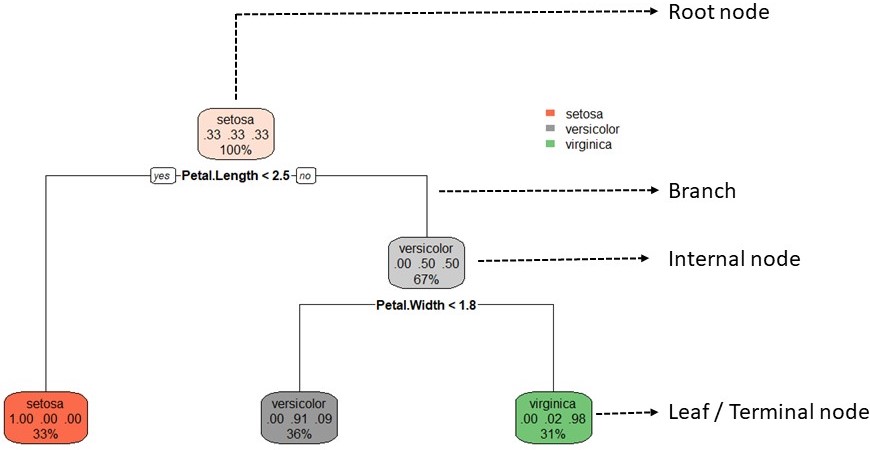
where are unknown constants. The sample mean is the constant that optimizes the MSE over that region. The bottom right panel is a perspective plot of the regression surface from this model.

The bottom left panel of the figure shows a the binary tree we’ve created.

* The top of the tree represents the full dataset.
* Then the branches represent the splitting at each step as we keep splitting the data into region.
* Observations satisfying the condition at each junction are assigned to the left branch, and the others to the right branch.
* The *terminal nodes*, called *leaves* of the tree correspond to the regions .
* This is the reason we call such methods *decision tree* methods.

Such trees can be used for classification problems as well.

The figure below shows a basic (classification) tree and the corresponding terminology used for any tree.



Terminology ralated to a decision tree.

## How Is a Regression Tree Fit?

Recall: Once we’ve chosen our model form, we need to fit the model to data. Generally, we can write the fitting process as the minimization of some loss function over the training data. How do we pick our splits of the predictor space in this case?

There are many techniques for constructing regression trees. Perhaps the most utilized method is the classification and regression tree (CART) methodology. (Breiman, L., Friedman, J., Olshen, R. and Stone, C. (1984). Classification and Regression Trees, Wadsworth, New York.)

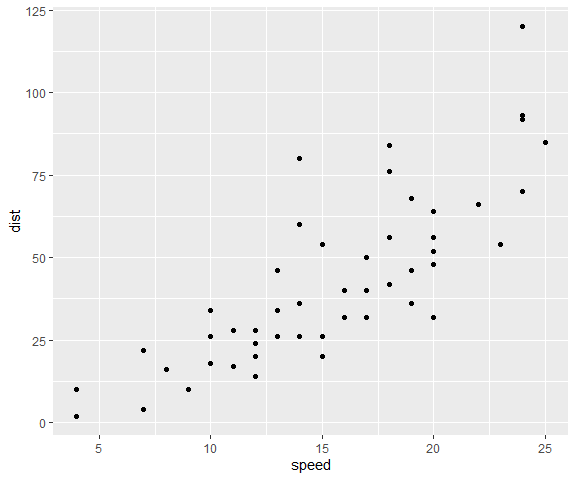
### One Predictor Concepts

We first discuss the CART algorithm with one predictor.

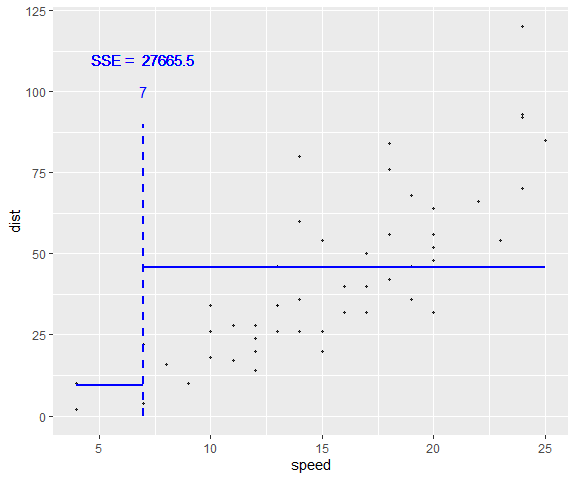
* Fit using recursive binary splitting - a greedy algorithm
* For every possible value of each predictor, we find the squared error loss based on splitting our data around that point. We then try to minimize that
  + Consider having one variable . For a given observed value, call it , we can think of having two regions (recall is read as ‘given’):
  + We seek the value of that minimize the equation
  + Written more mathematically, we could say we want minimize

Let’s visualize this idea! Consider that basic cars data set that has a response of dist (stopping distance) and a predictor of speed. Let’s find the value of the loss functions for different splits of our speed variable.

ggplot(cars, aes(x = speed, y = dist)) +   
 geom\_point()

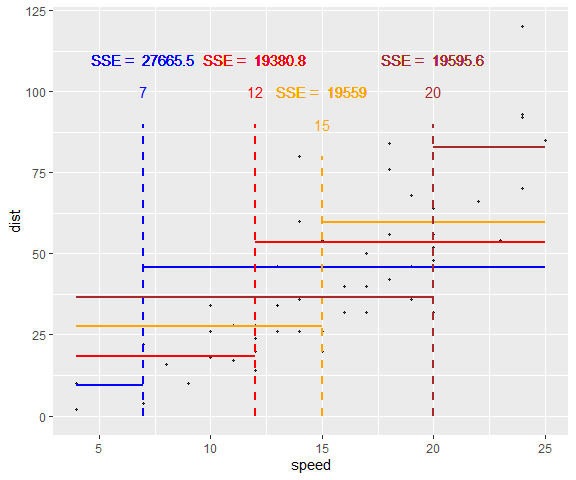


Let’s first try a split at speed = 7. The sum of squared errors based on this split is 2.766546^{4}.

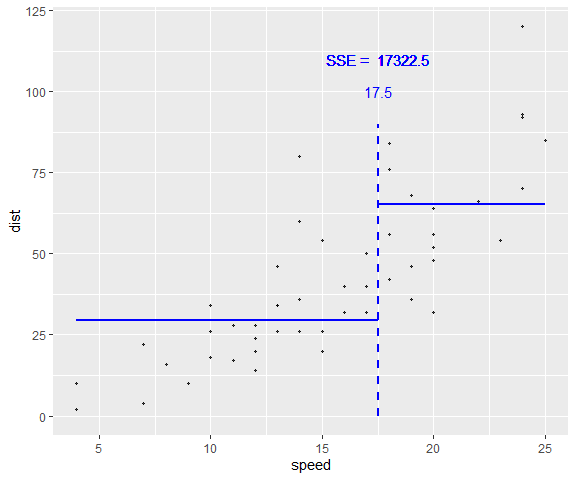


Again, this is found by taking all the points in the first region, finding the residual (from the mean, represented by the blue line here), squaring those, and summing the values. Then we repeat for the 2nd region. The sum of those two values is then the sum of squared errors (SSE) if we were to use this split.

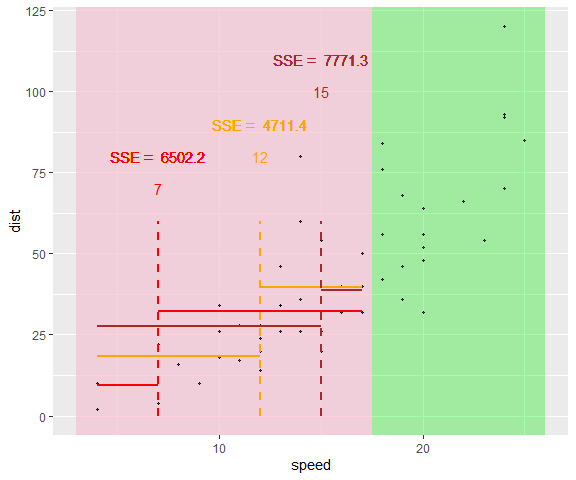
Is that the smallest it could be? Likely not! Let’s try some other splits and see what SSE they give.



* We would try this for all possible splits (across each predictor) and choose the split that minimizes the sum of squared errors as our first split. It turns out that speed = 17.5 is the optimal splitting point for this data set.



* Next, we’d go down the first branch of that split to that ‘node’. This node has all the observations corresponding to that branch. Now we repeat this process there!



* Here the best split on the lower portion is 12.5.
* Likewise, we go down the second branch to the other node and repeat the process.
* Generally, we grow a `large’ tree (many nodes)
* Trees can then be **pruned** back so as to not overfit the data (pruned back using some criterion like cost-complexity pruning)
* Generally, we can choose number of nodes/splits using the **training/test set or cross-validation**!

### Multiple Predictors

Suppose we have predictors, and a continuous response . We need to determine:

* The predictor to split on and split point
* The depth or complexity of the tree

Suppose first that we have partitioned the data into regions .

Define

The residual sum of squares is

As mentioned above, we construct the regions dynamically from the data. We might try to find regions that minimize the RSS above. Unfortunately, if the regions could be any of any shape, it is computationally infeasible to consider every possible partition of the feature space into regions.

For this reason, we take a *top-down*, *greedy* approach that is known as *recursive binary splitting*.

We begin with the entire data set, and search every distinct value of every predictor to find the predictor and split point that partitions the data into two groups such that the overall sums of squares error are minimized.

Formally, for the -th predictor and split point ,

* we have two regions: and
* the corresponding sum of squares is
* Note that we have indexed the regions by and since they depend on the splitting variable and the split point .
* Now we find the best value of and that minimize .
* Then within each of the two regions, we apply the same method and search for the predictor and split point that best reduces RSS.
* For each splitting variable, the determination of the split point can be done very quickly making it feasible to scan through all of the inputs and determine the optimal and at a particular step of the algorithm.

#### Example Tree Fit

Consider the Hitters data set in the ISLR2 library. Here we try to predict a baseball player’s Salary (1987 annual salary on opening day in thousands of dollars) based on Years (the number of years that he has played in the major leagues) and Hits (the number of hits that he made in the previous year).

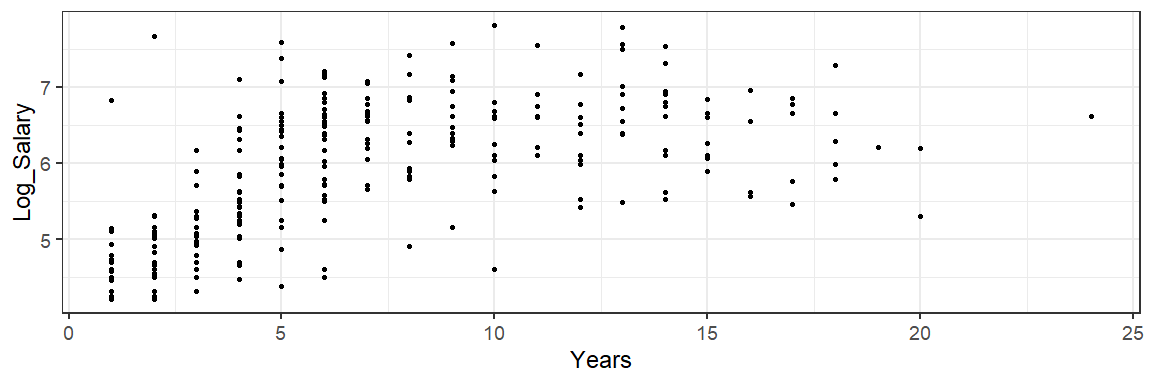
Examination of the data reveals that there are some missing (NA) values in the Salary variable. We first remove the missing salary values, and log-transform Salary so that its distribution has more of a typical bell-shape. The figures below show plots of Log\_Salary vs Years and Hits.

library(ISLR2)  
dim(Hitters)

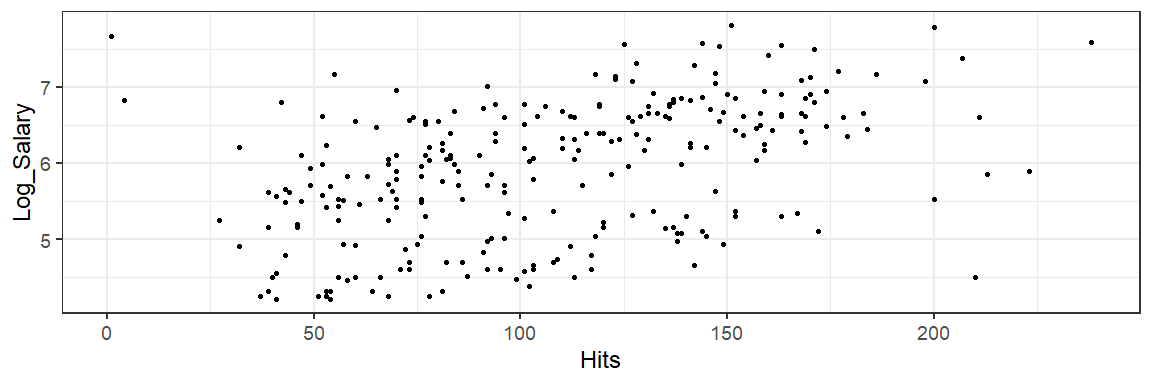
[1] 322 20

Hitters <- na.omit(Hitters) |>  
 as\_tibble() |>  
 mutate(Log\_Salary = log(Salary)) |>  
 select(-Salary)  
Hitters\_sub <- Hitters |>  
 select(Log\_Salary, Years, Hits)  
Hitters\_sub[1:5, ] |>  
 kable()

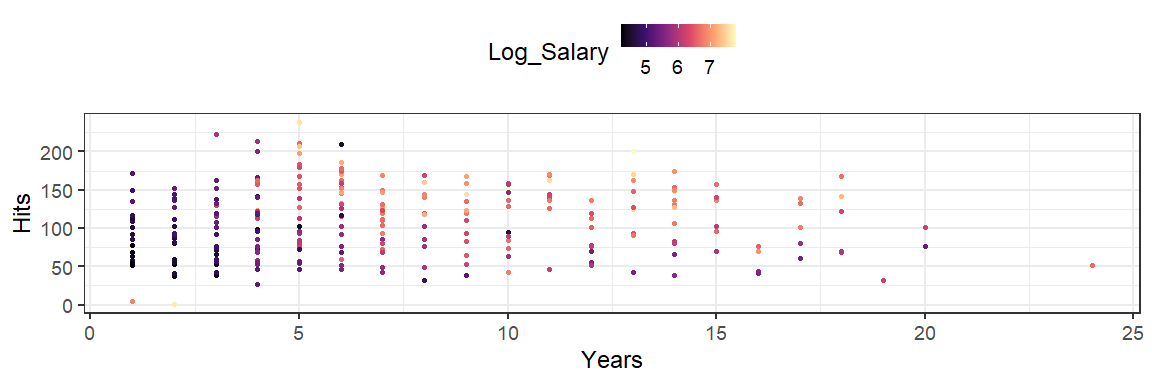
| Log\_Salary | Years | Hits |
| --- | --- | --- |
| 6.163315 | 14 | 81 |
| 6.173786 | 3 | 130 |
| 6.214608 | 11 | 141 |
| 4.516339 | 2 | 87 |
| 6.620073 | 11 | 169 |



Salary vs Years and Hits in the Hitters data.



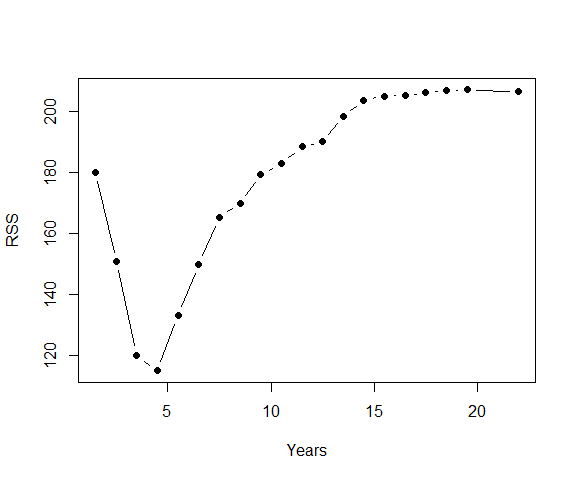
Salary vs Years and Hits in the Hitters data.



Salary vs Years and Hits in the Hitters data.

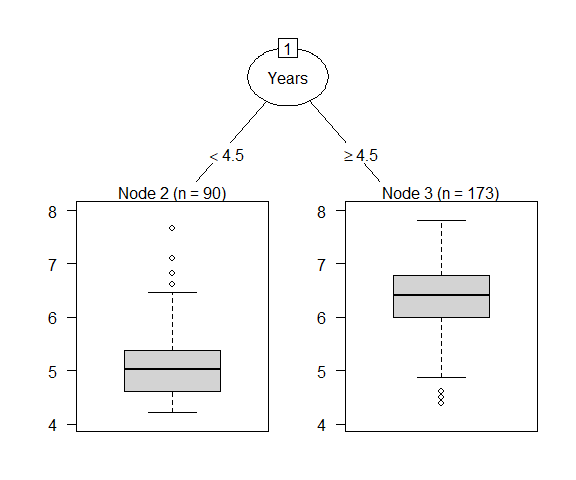
Now we determine the variable and the corresponding split point. The figure below shows the RSS for the continuum of splits for Years.

* The optimal split point for this variable is .
* The RSS associated with this split is compared to the optimal values for all of the other predictors (just Hits in this case) and the split corresponding to the absolute minimum error is used to form the two regions.



RSS for different split points for Years. The minimum occurs at 4.5.

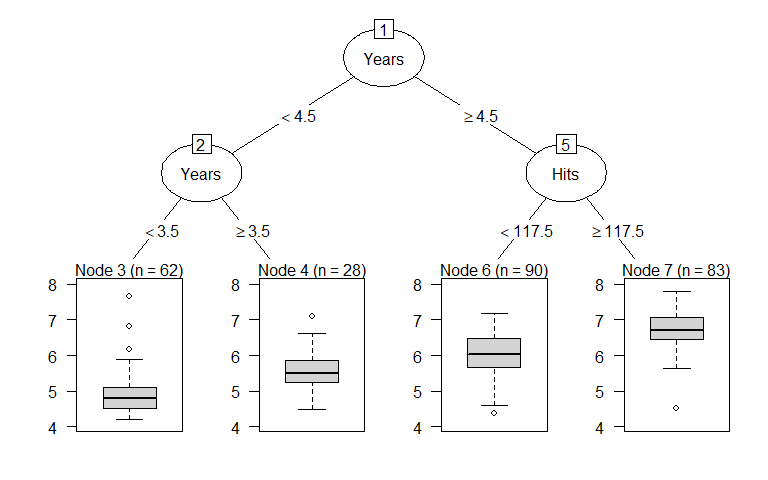
In our example, the Years variable was chosen to be the best, and the resulting tree shown below.



Splitting the initial data into two regions.

If stop building the tree at this point, all sample with values Years less than 4.5 would be predicted to be 5.11 (the average of the salary for these samples) and samples above the splits all have a predicted value of 6.35.

Next, we split each of the regions into two using the same algorithm as above. The resulting three is shown below.



Splitting the regions in Figure further.

At this point, the predictions for each region are 4.89 when Years less than 3.5, 5.58 when Years is between 3.5 and 4.5, 6 when Years is more than 4.5 and Hits is less than 117.5, and 6.74 when Years is more than 4.5 and Hits is more than 117.5.

We continue in this manner until we grow a large tree.

Note that one variable can be used multiple times times throughout the tree building process. Similarly, some of the variables might never be used at all.

Now the natural question is: how deep/complex should we grow the tree?

* Growing an overly complex tree will have the risk of overfitting our training data. This might result in poor test performance.
* On the other hand, growing a small tree might result in poor prediction. There are two primary approaches to find the “right size” of a regression tree: (1) early stopping, and (2) pruning.

### Early stopping

Using *early stopping*, we restrict tree growth explicitly using a pre-set criterion. Two of the most common approaches are as follows:

* restrict the tree depth to a certain level: we stop splitting after a certain depth, say, 5 levels.
  + This criterion results in shallow trees, resulting in less variance, but more bias.
* restrict the minimum number of observations allowed in any terminal node: stop splitting intermediate nodes which contain too few data points.
  + In the extreme case where each leaf contains only one oservation, we are essentially interpolating the training data. This results in overfitting and high variance. So restricting a minimum node size reduces variance.

These two approaches can be applied independently of each other, however, they do interact. Often, we use both of these criteria to build a tree.

### Tree Pruning

In this approach, we first grow a a very large, complex tree, , stopping the splitting process only when some minimum node size (say 5) is reached.

We then *prune* this tree using *cost-complexity pruning* to obtain a subtree.

* We define a *subtree* , a subset of , to be any tree that can be obtained by pruning , that is, collapsing any number of its internal nodes.
* We index terminal nodes by , with node representing region .
* Let denote the number of terminal nodes in . We consider a sequence of trees indexed by a nonnegative tuning parameter , such that,
* is minimized.
* The tuning parameter is called the *complexity parameter*, and it controls a trade-off between the subtree’s complexity and its fit to the training data.
  + When , then the subtree will simply equal .
  + As increases, there a tree with many terminal nodes will have larger , and so the quantity will tend to be minimized for a smaller subtree.
* It turns out that as we increase from zero, branches get pruned from the tree in a nested and predictable fashion, so obtaining the whole sequence of subtrees as a function of is easy!
* We can select a value of using a holdout set or cross-validation.

Similar to our previous discussion of selecting a tuning parameter, we often use the one-standard-error rule on the optimization criteria for identifying the simplest tree. That is, find the smallest tree that is within one standard error of the tree with smallest absolute error.

### Fitting Trees in R

There are several packages in R that are commonly used to build a regression (and classification) tree, such as rpart, party (uses a different splitting criterion called *conditional inference*), tree, and so on.

The textbook gives demonstration based on the tree package. Here we demonstrate the rpart library.

Let us use the Hitters data as before, but with all the predictors. The rpart fucntion has a few parameters that control the tree building (See ?rpart.control for details. Also see https://cran.r-project.org/web/packages/rpart/index.html for an introduction to rpart functionality.)

First we grow a large tree and look at the optimal subtrees for each value of the complexity parameter (denoted by cp in rpart).

set.seed(1001)  
T0 <- rpart(Log\_Salary ~ .,   
 data = Hitters,   
 control = rpart.control(xval = 10,   
 minbucket = 2,   
 cp = 0))

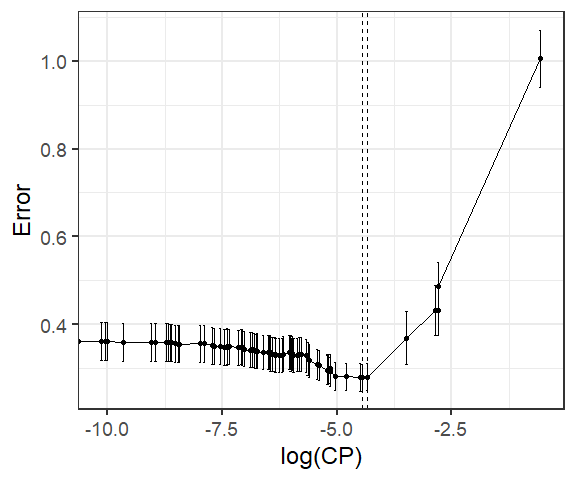
* xval=10 specifies that we are using 10-fold CV to estimate the prediction error corresponding to each value of the complexity parameter.
* minbucket = 2 indicates that the minimum number of observations in a terminal node must be 2.
* cp = 0 indicates that the threshold complexity parameter is zero, that is, we will use a grid on cp values all the way to zero.
  + If we set cp = 0.1 instead, only values down to 0.1 would be considered.
  + In rpart, the parameter cp is not exactly the same as . Instead, it uses the following formula: for a subtree ,
  + where is a tree with no splits. Thus cp is a scaled, unit less, version of .
  + A value of cp = 1 will always result in a tree with no splits.
  + For regression models the scaled cp has a very direct interpretation: if any split does not increase the overall of the model by at least cp then that split is decreed to be, a priori, not worth pursuing.
  + There are other criteria as well, that we do not explicitly set in this example.

Let us now look at the cross-validated prediction errors (xerror column below) and the corresponding standard error estimates (xstd column below). The relative error is .

printcp(T0)

To save space, we have shown only part of the output of the previous command.

Regression tree:  
rpart(formula = Log\_Salary ~ ., data = Hitters, control = rpart.control(xval = 10,   
 minbucket = 2, cp = 0))  
  
Variables actually used in tree construction:  
 [1] Assists AtBat CAtBat CHits CHmRun CRBI CRuns CWalks   
 [9] Division Errors Hits HmRun NewLeague PutOuts RBI Runs   
[17] Walks Years   
  
Root node error: 207.15/263 = 0.78766  
  
n= 263   
  
 CP nsplit rel error xerror xstd  
1 5.6894e-01 0 1.000000 1.00561 0.065469  
2 6.1288e-02 1 0.431062 0.48689 0.054917  
3 6.1195e-02 2 0.369774 0.43266 0.056019  
4 5.7784e-02 3 0.308579 0.43266 0.056019  
5 3.0786e-02 4 0.250795 0.36961 0.059732  
6 1.3097e-02 5 0.220008 0.28086 0.031432  
7 1.1701e-02 6 0.206912 0.27952 0.030926  
8 1.1215e-02 7 0.195211 0.27986 0.031201  
9 8.2164e-03 8 0.183996 0.28125 0.031467  
...  
...  
68 6.3797e-05 74 0.041751 0.36034 0.042937  
69 4.5067e-05 75 0.041687 0.36166 0.043262  
70 4.3197e-05 76 0.041642 0.36123 0.043238  
71 3.9989e-05 77 0.041599 0.36123 0.043238  
72 0.0000e+00 78 0.041559 0.36107 0.043257



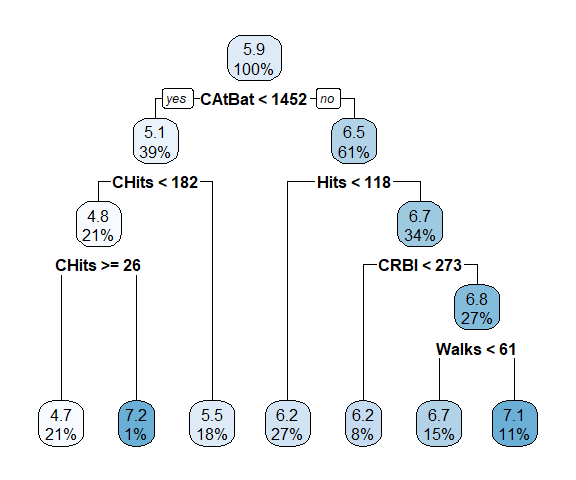
Cross-validated errors vs log of complexity parameter values.

The minimum error corresponds the following cp value. Here Upper and Lower correspond to Error plus/minus 1-SE.

CP Error Lower Upper  
7 0.01170077 0.2795239 0.2485983 0.3104495

Thus we can use either cp = 0.012, or use the 1-SE rule to chose cp = 0.013. Recall that larger cp implies smaller tree size. Now we can prune the tree using the chosen value of cp (using 1-SE rule) as follows.

final <- prune(T0, cp = 0.013)  
rpart.plot(final)

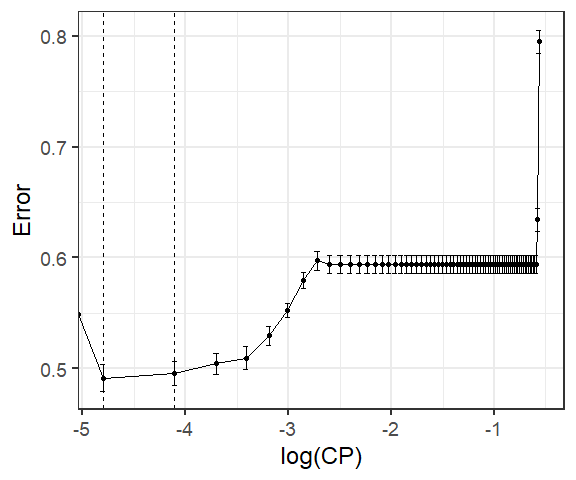


Final pruned tree for the Hitters data.

Alternatively, we can use caret, using the train() function with method = rpart.

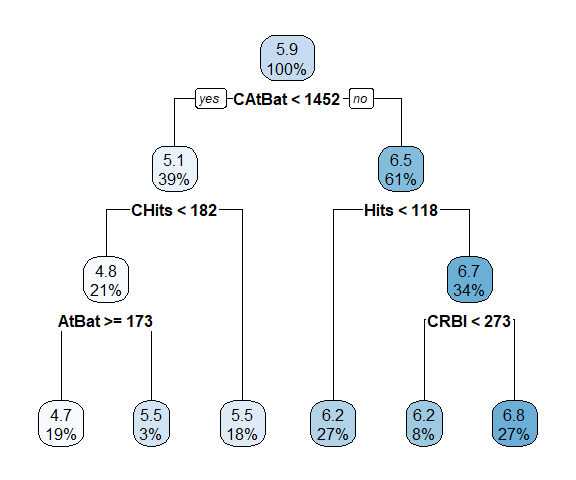
library(caret)  
set.seed(1001)  
hit\_tree <- train(Log\_Salary ~ ., data = Hitters,  
 method = "rpart",  
 tuneLength = 70,  
 trControl = trainControl(method = "cv",   
 number = 10))  
hit\_tree$bestTune

cp  
2 0.008245477



While the minimum error is obtained for cp = 0.008, we can again apply 1-SE rule, and obtain a larger cp value of 0.016.

final\_caret <- prune(hit\_tree$finalModel, cp = 0.016)  
rpart.plot(final\_caret)



Pruned tree for Hitters data using caret.

Overall, the following steps are used to choose using cross-validation, see the textbook, algorithm 8.1.

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 .
3. Use K-fold cross-validation to choose . That is, divide the training observations into folds. For each :
   1. Repeat Steps 1 and 2 on all but the kth fold of the training data.
   2. Evaluate the mean squared prediction error on the data in the left-out -th fold, as a function of .

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

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

### Missing Data

The CART methodology handles missing data! Missing data are ignored when building the tree.

For each split, one evaluates a variety of alternatives, called *surrogate splits*: a split whose results are similar to the original split actually used in the tree.

* If a surrogate split approximates the original split well, it can be used when the predictor data associated with the original split are not available.
* In practice, several surrogate splits may be saved for any particular split in the tree.

### Variable Importance

We can assess the relative importance of the the predictors to the outcome once we chose the final tree.

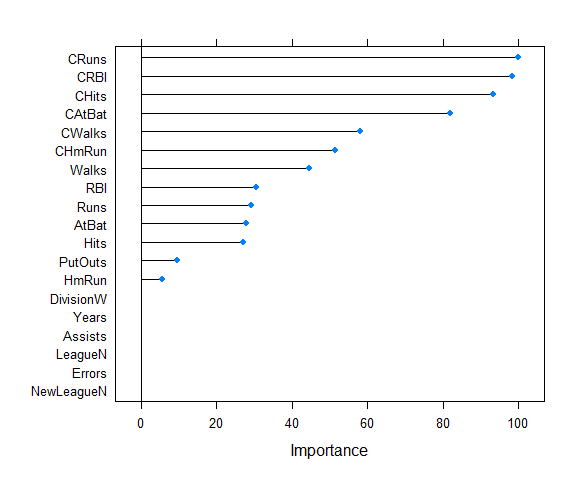
One way to compute an aggregate measure of importance is to keep track of the overall reduction in the optimization criteria for each predictor.

* In our example, we can tabulate the reduction RSS attributed to each variable.
* If a single variable could be used multiple times in a tree, the total reduction in RSS across all splits by a variable are summed up and used as the total feature importance.

When using caret, importance values are scaled so that the most important feature has a value of 100. The remaining features are then scored based on their relative reduction of RSS.

Also, since there may be candidate variables that are important but are not used in a split, the top competing variables are also tabulated at each split. In caret, the varImp() function can be used.

plot(varImp(hit\_tree))



### Categorical Predictors

So far we have only discussed the case where the predictors are continuous. However, splitting methods are also available for categorical data.

* For a binary predictor(say 0/1), splitting can be done based on whether the predictor takes value 0 or 1.
* For a categorical predictor with more than two levels, a split amounts to assigning some of the qualitative values to one branch and assigning the remaining to the other branch.

## Classification Trees

Classification trees are very similar to regression trees except, of course, our response is a categorical variable.

This means that we don’t use the same loss functions nor metrics, but we still split the predictor space up into regions. We then can make our prediction based on which bin an observation ends up in. Most often, we use the most prevalent class in a bin as our prediction.

In interpreting the results of a classification tree, we are often interested in both the class prediction corresponding to a particular terminal node region and the class proportions among the training observations that fall into that region.

Compared to regression trees, the only changes needed in the tree algorithm are the criteria for splitting nodes and pruning the tree.

An ideal node would be the one with all observations are from the same class. Thus one alternative to RSS is to look at some measures of *node impurity*.

In a node , representing a region , let be the proportion of training observations in that are from the -th class. We classify the observations in node to the majority class in node , that is, the value of that maximizes . Based on this observation, we can look at three different measures of node impurity:

* Misclassification error:
* Gini index:
* Cross-entropy or deviance:

It turns out that classification error is not sufficiently sensitive for tree-growing, and in practice we often prefer one of the other two measures.

* Gini index is a measure of total variance across the classes.
  + Gini index takes on a small value if all of the are close to zero or one
  + A small value indicates that a node contains predominantly observations from a single class
* Cross-entropy, will take on a value near zero if all of the are close to zero or one.
  + Thus, entropy will take on a small value if the -th node is pure.

It turns out that the Gini index and the entropy are quite similar numerically!

Any of these three approaches might be used when pruning the tree, but the classification error rate is preferable if prediction accuracy of the final pruned tree is the goal.

### Classification Tree Example

Let us use the heart data to demonstrate classification trees.

* The outcome is AHD: 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.

# Read hear data  
heart <- read.csv("https://www.statlearning.com/s/Heart.csv", header = TRUE)  
# Remove the row numbers, and NAs  
heart <- heart[,-1]  
heart <- na.omit(heart)  
heart <- heart |>  
 mutate(AHD = as.factor(AHD))  
heart[1:5, 1:8] |>  
 kable()

| Age | Sex | ChestPain | RestBP | Chol | Fbs | RestECG | MaxHR |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 63 | 1 | typical | 145 | 233 | 1 | 2 | 150 |
| 67 | 1 | asymptomatic | 160 | 286 | 0 | 2 | 108 |
| 67 | 1 | asymptomatic | 120 | 229 | 0 | 2 | 129 |
| 37 | 1 | nonanginal | 130 | 250 | 0 | 0 | 187 |
| 41 | 0 | nontypical | 130 | 204 | 0 | 2 | 172 |

We proceed in a similar way as we did for regression tree with the only changes being:

* method='class' - specifies the type of problem (classification)
* parms = list(split = "information") sets the splitting criterion as cross-entropy

set.seed(1001)  
heart\_rpart <- rpart(AHD ~ .,   
 data = heart,  
 method='class',  
 parms = list(split = "information"),  
 control = rpart.control(xval = 10,   
 minbucket = 2,   
 cp = 0))  
printcp(heart\_rpart)

Classification tree:  
rpart(formula = AHD ~ ., data = heart, method = "class", parms = list(split = "information"),   
 control = rpart.control(xval = 10, minbucket = 2, cp = 0))  
  
Variables actually used in tree construction:  
 [1] Age Ca ChestPain Chol ExAng Fbs MaxHR   
 [8] Oldpeak RestBP RestECG Sex Thal   
  
Root node error: 137/297 = 0.46128  
  
n= 297   
  
 CP nsplit rel error xerror xstd  
1 0.4890511 0 1.00000 1.00000 0.062708  
2 0.0510949 1 0.51095 0.55474 0.054891  
3 0.0401460 3 0.40876 0.45985 0.051426  
4 0.0218978 5 0.32847 0.45255 0.051125  
5 0.0145985 7 0.28467 0.47445 0.052012  
6 0.0109489 9 0.25547 0.48175 0.052297  
7 0.0097324 12 0.21898 0.48905 0.052578  
8 0.0072993 15 0.18978 0.48905 0.052578  
9 0.0054745 17 0.17518 0.51095 0.053390  
10 0.0036496 27 0.11679 0.52555 0.053909  
11 0.0000000 29 0.10949 0.54745 0.054652

The Root node error corresponds to , that is, misclassification error is we simply assign everything to the majority class.

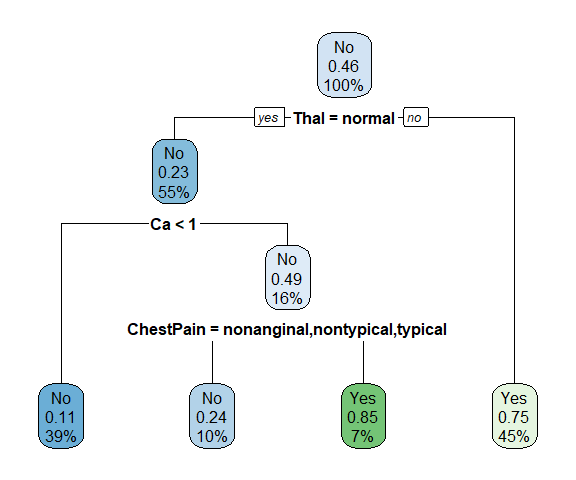
table(heart$AHD)/nrow(heart)

No Yes   
0.5387205 0.4612795

In the output above, for easier reading, the error columns have been scaled so that the first node has an error of 1. We can multiply the error rates in the table with Root node error to obtain the actual error rates. Here rel error corresponds to training error rate.

Using 1-SE rule, we chose the tree with three splits, and corresponding tree is shown below.

cp <- heart\_rpart$cptable  
heart\_final <- prune(heart\_rpart, cp = cp[3,1])  
rpart.plot(heart\_final)



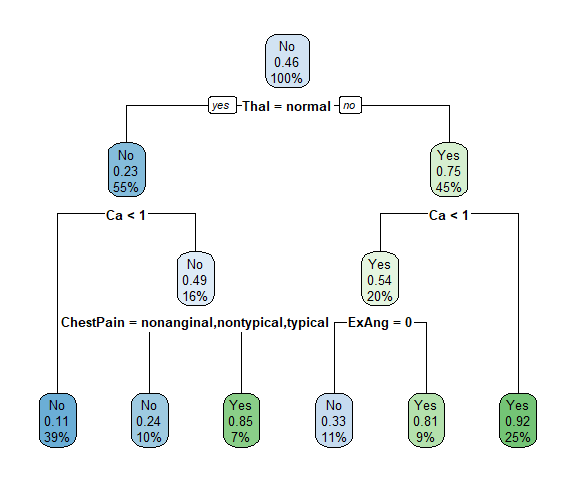
Pruned tree using 1-SE rule for the heart data.

# Training error rate  
pred <- predict(heart\_final, type = "class")  
klaR::errormatrix(true = heart$AHD, predicted = pred,   
 relative = TRUE)

predicted  
true No Yes -SUM-  
 No 0.7750000 0.2250000 0.2250000  
 Yes 0.1459854 0.8540146 0.1459854  
 -SUM- 0.3571429 0.6428571 0.1885522

If we use the cp value corresponding to the minimum CV error rate, we obtain the following tree:

heart\_final <- prune(heart\_rpart, cp = cp[4,1])  
rpart.plot(heart\_final)



Pruned tree using minimum cp for the heart data.

# Training error rate using min cp  
pred <- predict(heart\_final, type = "class")  
klaR::errormatrix(true = heart$AHD, predicted = pred,   
 relative = TRUE)

predicted  
true No Yes -SUM-  
 No 0.9125000 0.0875000 0.0875000  
 Yes 0.2262774 0.7737226 0.2262774  
 -SUM- 0.6888889 0.3111111 0.1515152

Consider the split Ca < 1 on the right side of the tree. We notice that regardless of the value of Ca, a response value of Yes is predicted for those observations. Then why was this node split in the first place? The reason is that by splitting this node, we get a leaf node (bottom right of the tree) which is much purer than the original node. Originally, the parent node has data coming from Yes class. After splitting, of the bottom right node are from the Yes class. Suppose that we have a test observation that belongs to the region given by that right-hand leaf. Then we can be pretty certain that its response value is Yes. In contrast, if we did not split the original node, and if a test observation falls into the region, then its response value is probably Yes, but we are much less certain.

## Recap and Pros & Cons

* Trees are a nonlinear model that can be more flexible than linear models. Pros:
* Simple to understand and easy to interpret output
* Predictors don’t need to be scaled. Unlike algorithms like the LASSO, having all the predictors on different scales makes no difference in the choosing of regions.
* No statistical assumptions necessary to get the fit (although this is true for least squares regression as well)
* Built in variable selection based on the algorithm!

Cons:

* No optimal algorithm for choosing splits exists.
  + We saw the use of a greedy algorithm to select our regions in the regression tree case. This is a greedy algorithm because it is only looking one step ahead to find the best split. There might be a split at this step that creates a great future split. However, we may never find it because we only ever look at the best thing we can do at the current split!
* Need to prune or use CV to determine the model.
  + With MLR models, CV isn’t used at all. However, here we really need to prune the tree and/or use CV to figure out the optimal size of the tree to build!
* Small changes in data can vastly change tree
  + The lack of ‘sharing’ information with nearby data points makes this algorithm more variable. Given a new data set from the same situation, the splits we get for the tree can differ quite a bit! That isn’t ideal as we’d like to have stable results across data sets collected on the same population.
  + Note: By *aggregating* many decision trees, using methods like *bagging*, *random forests*, and *boosting*, the predictive performance of trees can be substantially improved!

# Ensemble Tree Models

In general, bagging, random forests, and boosting are part of more general learning method called *ensemble learning*. The idea of ensemble learning is to build a prediction model by combining the strengths of a collection of simpler base models, called *weak learners*.

* Bagging and random forests are ensemble methods where a committee of trees each cast a vote for the predicted class.
* Boosting was initially proposed as a committee method as well, although unlike random forests, the committee of weak learners evolves over time, and the members cast a weighted vote.

There are other methods, for example *Stacking*, to combining the strengths of a number of fitted models. In fact one could characterize any dictionary method, such as regression splines, as an ensemble method, with the basis functions serving the role of weak learners.

In the context of the tree-based methods, we’ll discuss bagging, random forests, boosting, and Bayesian additive regression trees (BART). These are ensemble methods for which the simple building block is a regression or a classification tree.

## Bagging and Random Forests

One method to obtain many different trees from one data set is to use the bootstrap! Recall, we can do a non-parametric bootstrap by

* resampling using a sample size of from the data with replacement
  + we sometimes get the same observation multiple times
  + observations not included can act as a test set (Out-of-bag observations)

We can then fit a large tree (not pruned) on the bootstrap resample. Then we repeat many times!

Our prediction is then the average prediction (regression) or a majority vote (classification).

* These trees will have a lot of variance between them but each will have low bias.
* By averaging across many trees, we decrease the variance over an individual tree fit

**Bagging = Bootstrap Aggregation** is a general method. Applied to trees the algorithm is:

1. Create many bootstrap (re)samples,
2. Fit (large) tress to each (re)sample
   * Yields fitted trees:
3. For a given set of predictor values, find the prediction using each tree
   * Call prediction for a given set of values
4. Combine the predictions from the trees to create the final prediction!
   * For regression trees, we usually use the average of the predictions

+ For classification trees, usually use the \*\*majority vote\*\*   
  
$$\mbox{Use most common prediction made by all bootstrap trees}$$

**Random Forests** are bagged tree where, at each split of each tree, we consider a random subset of predictors rather than the entire set of predictors.

* If a really strong predictor exists, every bootstrap tree will probably use it for the first split (2nd split, etc.)
* Makes bagged trees predictions more correlated
* Correlation –> smaller reduction in variance from aggregation

By randomly selecting a subset of predictors, a good predictor or two won’t dominate the tree fits

* Rules of thumb exist for the number of predictors to use but we can just use CV or out-of-bag predictions to choose this number.

### Bagging and Random Forest Example

There are several packages in R to perform bagging, such as randomForest and ipred. We can also use caret for this purpose.

Let us demonstrate bagging using the caret package. We will use the Hitters data as used above.

library(caret)  
set.seed(1001)  
hit\_bagged <- train(Log\_Salary ~ .,   
 method = "rf",  
 data = Hitters,  
 tuneGrid = data.frame(mtry = 1:(ncol(Hitters)-1)),  
 trControl = trainControl(method = "oob", number = 3000))  
hit\_bagged

Random Forest   
  
263 samples  
 19 predictor  
  
No pre-processing  
Resampling results across tuning parameters:  
  
 mtry RMSE Rsquared   
 1 0.4618922 0.7291404  
 2 0.4268088 0.7687245  
 3 0.4222140 0.7736772  
 4 0.4236812 0.7721016  
 5 0.4225198 0.7733493  
 6 0.4266703 0.7688746  
 7 0.4272542 0.7682415  
 8 0.4242121 0.7715301  
 9 0.4264789 0.7690819  
 10 0.4311861 0.7639562  
 11 0.4288726 0.7664824  
 12 0.4307329 0.7644522  
 13 0.4290882 0.7662476  
 14 0.4359448 0.7587174  
 15 0.4326085 0.7623963  
 16 0.4328457 0.7621357  
 17 0.4298278 0.7654411  
 18 0.4356267 0.7590694  
 19 0.4346428 0.7601565  
  
RMSE was used to select the optimal model using the smallest value.  
The final value used for the model was mtry = 3.

hit\_bagged$bestTune

mtry  
3 3

* The argument mtry specifies , the number of predictors to consider at each split. As bagging is special case of random forests when , we’ve included the bagged tree model when doing our selection of mtry.

Now we can predict a new observation by using the predict() function.

newx <- Hitters[1,]  
newx

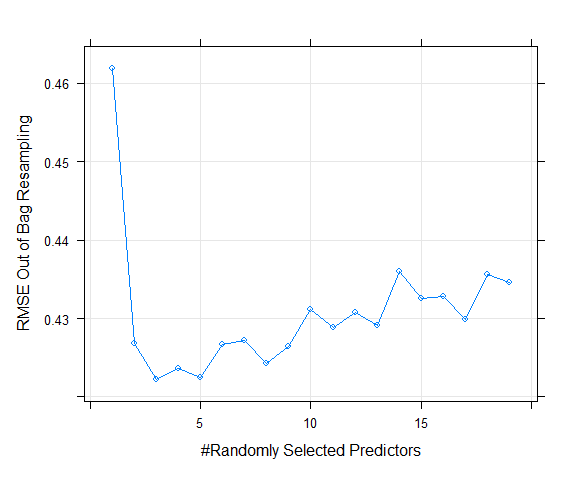
# A tibble: 1 x 20  
 AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun CRuns CRBI  
 <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int>  
1 315 81 7 24 38 39 14 3449 835 69 321 414  
# i 8 more variables: CWalks <int>, League <fct>, Division <fct>,  
# PutOuts <int>, Assists <int>, Errors <int>, NewLeague <fct>,  
# Log\_Salary <dbl>

pred <- predict(hit\_bagged, newdata = newx)  
pred

1   
6.157066

On average, each bagged tree/model uses about two-thirds of the training observations. out-of-bag (OOB) observations refer to the remaining one-third of the observations not included in the bootstrap sample.

* We can estimate the test error of a bagged model using these out-of-bag (OOB) observations.
* We can predict the response for the -th observation using each of the trees in which that observation was OOB.
  + This will yield around predictions for the -th observation.
  + In order to obtain a single prediction for the -th observation, we can average these predicted responses (for regression model) or can take a majority vote (for classification models).



Estimated test MSE for different number of aggregated trees.

The figure above shows the OOB estimate of test MSE. Notice that the more trees the better.

* As we add more trees, we are averaging over more high variance decision trees.
* We see a dramatic reduction in variance early but eventually the reduction in error will slow down.
* In our example, after around 100 trees, we do not get much benefit by averaging more trees.

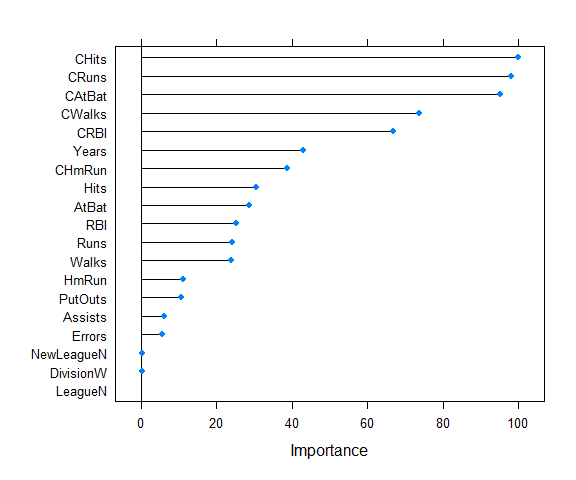
A disadvantage of bagging is that the resulting model is often difficult or impossible to interpret, as we are averaging many trees rather than looking at a single tree.

* We can still compute variable importance scores.

varImp(hit\_bagged)

rf variable importance  
  
 Overall  
CHits 100.0000  
CRuns 98.1913  
CAtBat 95.3215  
CWalks 73.6520  
CRBI 66.7259  
Years 42.8923  
CHmRun 38.8556  
Hits 30.5811  
AtBat 28.7212  
RBI 25.0925  
Runs 24.2266  
Walks 23.8598  
HmRun 11.2224  
PutOuts 10.5394  
Assists 6.1819  
Errors 5.6473  
NewLeagueN 0.3727  
DivisionW 0.1767  
LeagueN 0.0000

plot(varImp(hit\_bagged))



* This measure is based upon the mean decrease of accuracy in predictions on the out of bag samples when a given variable is permuted.

## Boosting

Boosting is another way to improve predictions from a decision tree. These models were originally developed for classification problems and were later extended to the regression setting. Like bagging, boosting is a general approach that can be applied to many statistical learning methods for regression or classification.

* In bagging, we create multiple copies of the original training data set using the bootstrap, fit a separate decision tree to each copy, and then combine all of the trees in order to create a single predictive model. 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.

In a way, boosting addressed the bias-variance-tradeoff by starting with a *weak model*, for example, a decision tree with only a few splits, and sequentially improves its performance by continuing to build new trees.

* Each new tree attempts to fix the biggest mistakes in the previous tree in the sequence.
* For example, each new tree in the sequence will focus on the training data where the previous tree had the largest prediction errors.

Here are the important components of boosting:

* *The base learners*: Technically, we can use boosting on many classification and regression models. Many boosting applications allow the user to “plug in” various classes of weak learners at their disposal. In practice however, boosted algorithms almost always use decision trees as the base-learner.
* *Training weak models*: We can call a model weak is its performance is only slightly better than random guessing. The idea behind boosting is that each model in the sequence slightly improves upon the performance of the previous one by focusing on the rows of the training data where the previous tree had the largest errors or residuals. With regards to decision trees, shallow trees (trees with relatively few splits) represent a weak learner. In boosting, trees with 1 - 6 splits are most common.
* *Sequential training with respect to errors*: Boosted trees are grown sequentially; each tree is grown using information from previously grown trees to improve performance. For example, in a regression problem, each tree is fitted to the previous tree’s residuals, and added back to the algorithm.

### Boosting Regression Trees

Let us start with boosting a regression tree. The following algorithm performs boosting for regression:

1. Set and set the residuals for all in the training set.
2. For , repeat:
   1. Fit a tree with splits ( terminal nodes) to the training data .
   2. Update by adding in a shrunken version of the new tree:
   3. Update the residuals,
3. Output the boosted model,

Unlike fitting a single large decision tree to the data, the boosting approach learns slowly. We fit a tree using the current residuals, rather than the outcome, , as the response. We then add this new decision tree into the fitted function in order to update the residuals. The size of each of these trees is determined by the parameter in the algorithm.

By fitting small trees to the residuals, we slowly improve in areas where it does not perform well. The shrinkage parameter slows the process down even further, allowing more and different shaped trees to attack the residuals. Note that in boosting, unlike in bagging, the construction of each tree depends strongly on the trees that have already been grown.

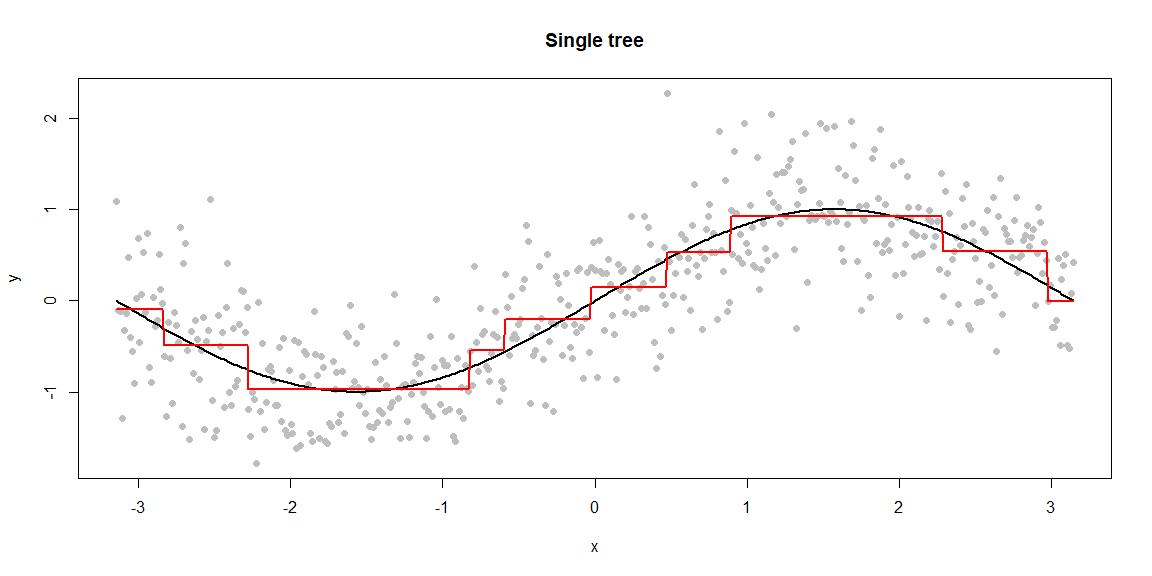
Boosting for regression has three tuning parameters:

1. *The number of trees* : Unlike bagging and random forests, boosting can overfit if is too large, although this overfitting tends to occur slowly if at all. We can use cross-validation to select .
2. *The shrinkage parameter* : This is a small positive number. This controls the rate at which boosting learns. Typical values are or but the right choice can depend on the problem. Very small can require using a very large value of in order to achieve good performance.
3. *The number of splits in each tree*: This controls the complexity of the boosted ensemble. Often works well, in which case each tree is a stump, consisting of a single split. In this case, the boosted ensemble is fitting an additive model, since each term involves only a single variable. More generally is the *interaction depth*, and controls the interaction order of the boosted model, since splits can involve at most variables.

Let us look at a simple example of a regression problem with one predictor. The truth comes from the following model:

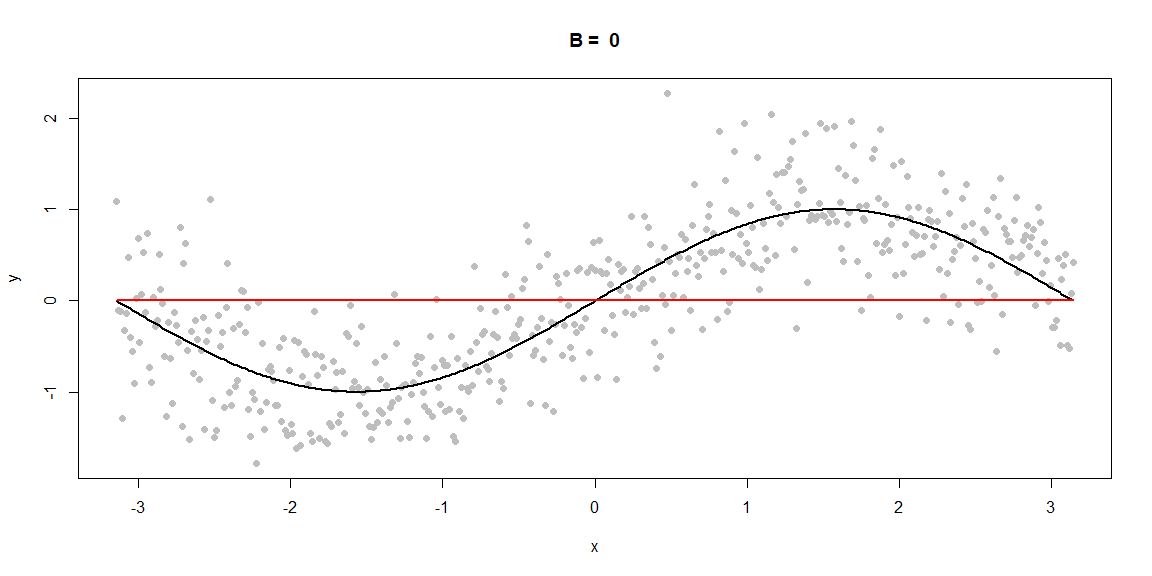
where takes values between and , and .

The figures below shows how gradient boosting proceeds with estimating the true underlying function, .



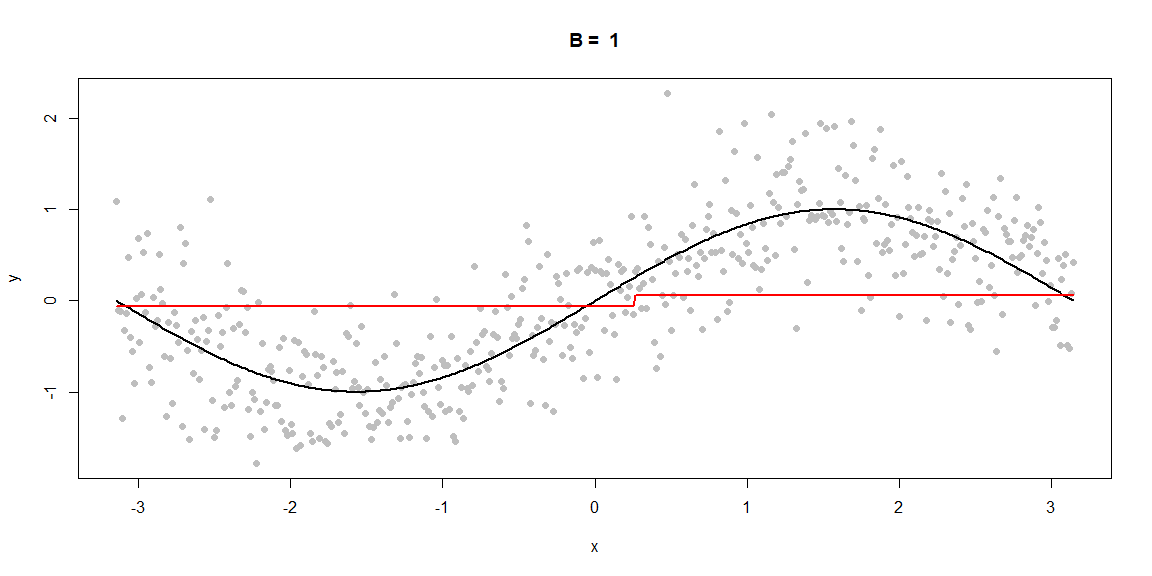
Demonstration of GBM using simulated data and different number of B.

* The first plot (“Single tree”) shows the estimated function based on a single tree pruned using cost-complexity pruning.



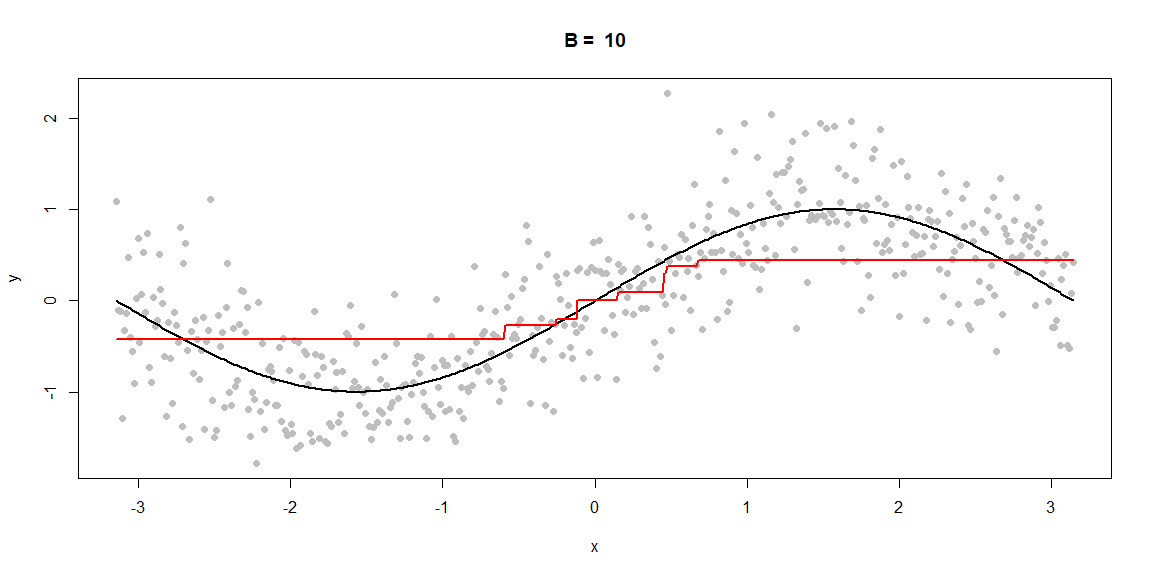
Demonstration of GBM using simulated data and different number of B.

* The second plot () shows the initialization for gradient boosting, that is, . Thereafter, indicates the number of trees grown sequentially. The first tree fit in the series is a single decision stump, i.e., a tree with a single split.

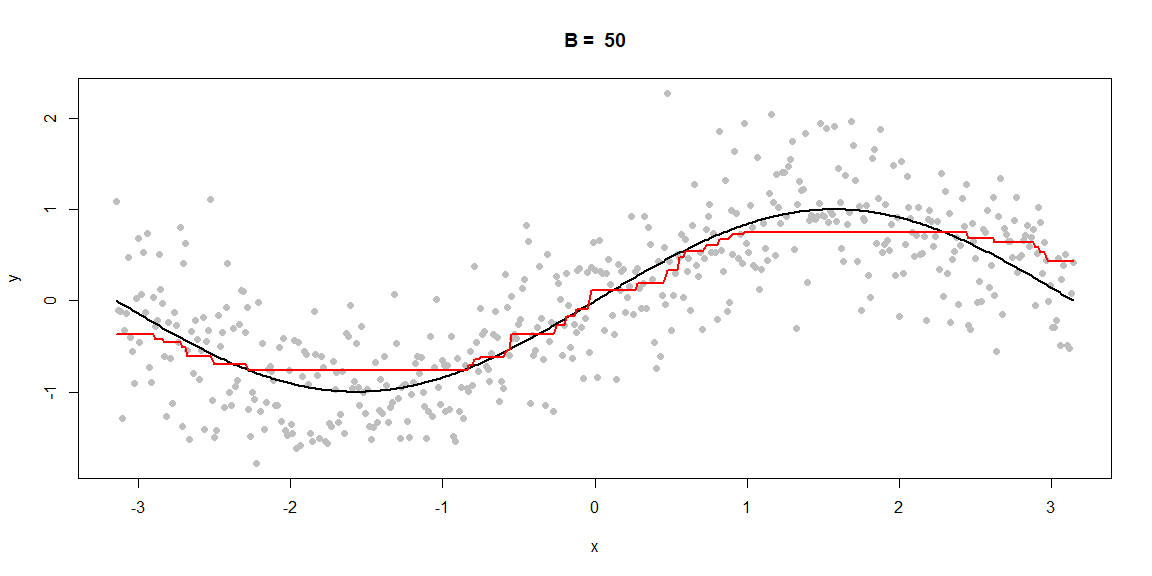


Demonstration of GBM using simulated data and different number of B.

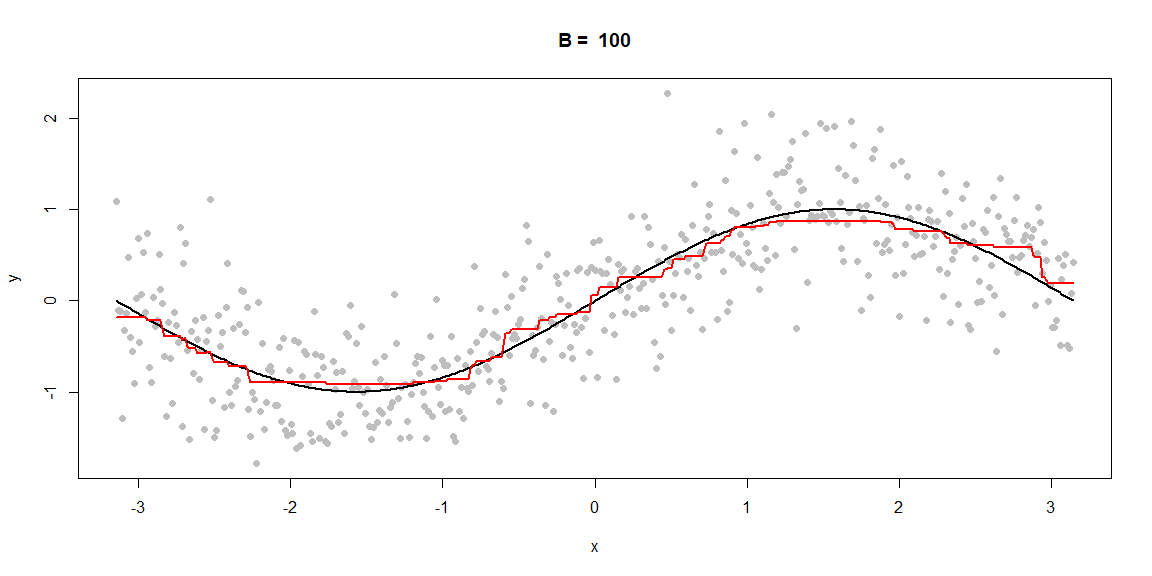
* After that, each successive decision stump is fit to the previous three’s residuals.
  + Initially there are large errors, but each additional decision stump in the sequence makes a small improvement in different areas across the feature space where errors still remain.
  + We also notice that after some point (e.g., ), the procedure shows signs of overfitting. Thus it is important to tune the parameter .



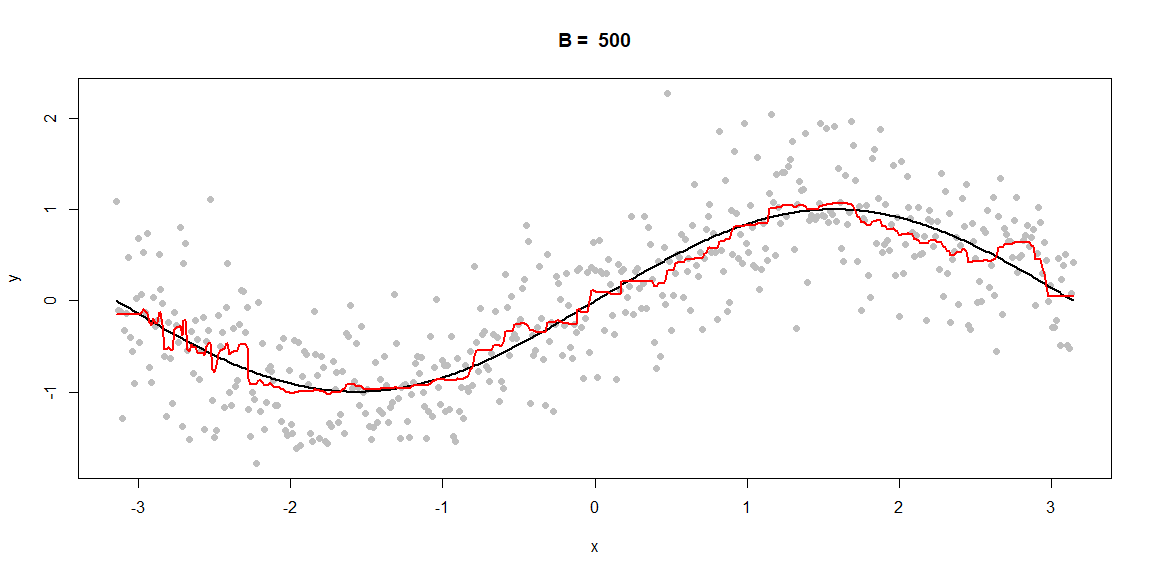
Demonstration of GBM using simulated data and different number of B.



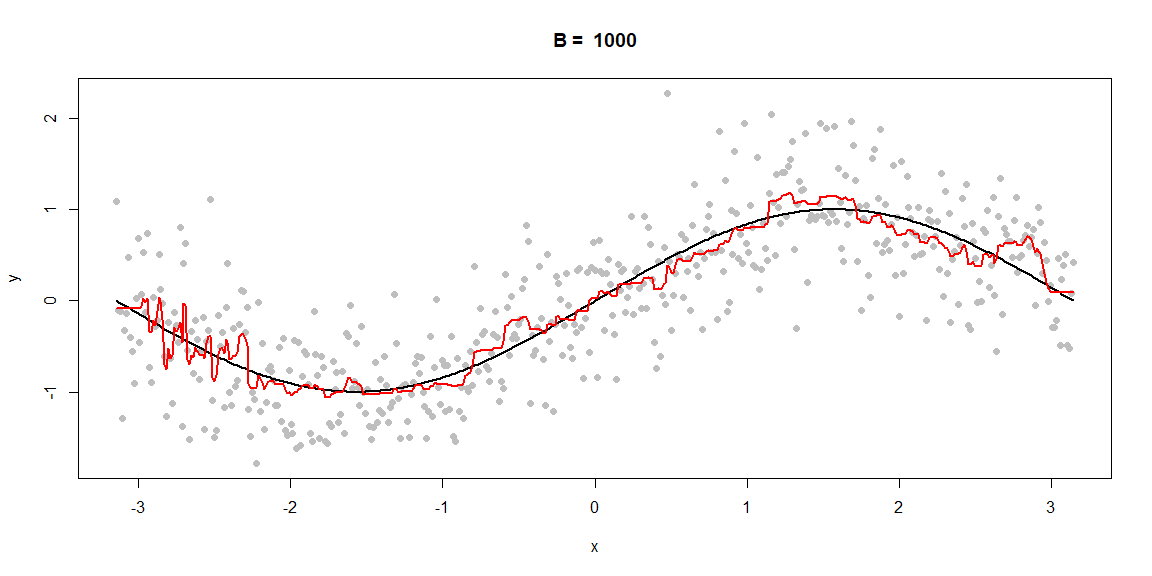
Demonstration of GBM using simulated data and different number of B.



Demonstration of GBM using simulated data and different number of B.



Demonstration of GBM using simulated data and different number of B.



Demonstration of GBM using simulated data and different number of B.

After Friedman published his gradient boosting machine, he updated the boosting machine algorithm with a random sampling scheme. Typically, such a random selection approach reduces the prediction variance. The new procedure is called *stochastic gradient boosting* (Friedman (2002) Stochastic Gradient Boosting. Computational Statistics & Data Analysis 38 (4). Elsevier: 367-78.).

To be specific, a subsample of the training data is drawn at random without replacement at each iteration, and used in place of the full training data.

Fitting the base learner and computing the model update for the current iteration is done only based on this subsample. The fraction of training data used is known as the *bagging fraction*.

It turns out that this simple modification improved the prediction accuracy of boosting while also reducing the required computational resources.

Friedman suggests using a bagging fraction of around . This value, however, can be tuned like any other parameter. There are a few variants of stochastic GBMs that can be used, often requiring additional hyperparameters:

* Subsample rows (traing observations) before creating each tree (available in gbm, h2o, and xgboost)
* Subsample columns (predictors) before creating each tree (available in h2o, and xgboost)
* Subsample columns (predictors) before considering each split in each tree (available in h2o and xgboost)

While the fraction of rows taken as a subsample (i.e., bagging fraction) is set at 0.5, typical values range from to . Subsampling of predictors and the impact to performance largely depends on the nature of the data and if strong multicollinearity or a lot of noisy features present in the data. When there are many relevant predictors, a lower values of predictor subsampling tends to perform well.

#### Fitting a Boosted Model in R

In R, the most widely used package for boosting regression trees via stochastic gradient boosting machines is gbm. gbm has two training functions:

* gbm() - uses the formula interface to specify your model
* gbm.fit() - requires the separated x and y matrices. This is more efficient and recommended for advanced users.

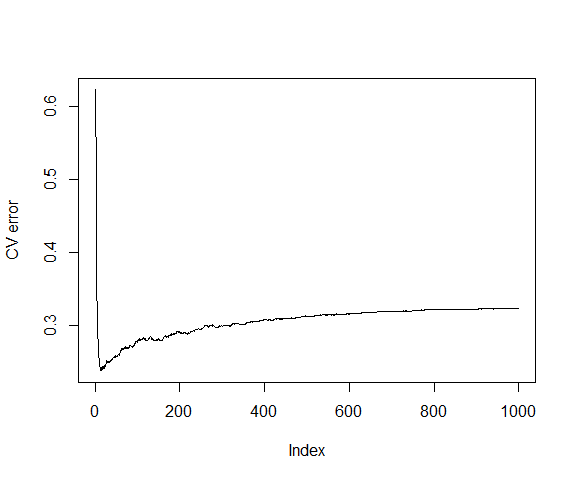
library(gbm)  
set.seed(1001)  
hit\_gbm1 <- gbm(  
 formula = Log\_Salary ~ .,  
 data = Hitters,  
 distribution = "gaussian",  
 n.trees = 1000,  
 shrinkage = 0.1\*2,  
 interaction.depth = 3,  
 n.minobsinnode = 10,  
 cv.folds = 5  
)

Here we have fit a gradient boosting model with

* the RSS loss (distribution = “gaussian”)
* 5000 sequentially generated trees (n.trees = 5000)
* (shrinkage = 0.1)
* (interaction.depth = 3)
* tree control parameter is the minimum number of observations in a node to be 10 (n.minobsinnode = 10)
* the use of 5-fold CV (cv.folds = 5) to estimate test error rate
* by default, the bagging fraction is taken to be 0.5 (bag.fraction = 0.5 is not specified as it is default)

print(hit\_gbm1)

gbm(formula = Log\_Salary ~ ., distribution = "gaussian", data = Hitters,   
 n.trees = 1000, interaction.depth = 3, n.minobsinnode = 10,   
 shrinkage = 0.1 \* 2, cv.folds = 5)  
A gradient boosted model with gaussian loss function.  
1000 iterations were performed.  
The best cross-validation iteration was 13.  
There were 19 predictors of which 16 had non-zero influence.



CV error vs number of trees in GBM

# Best number of trees with min CV error  
best <- which.min(hit\_gbm1$cv.error)  
best

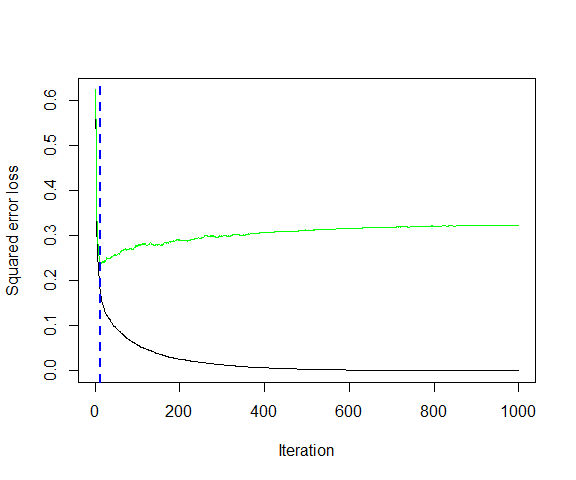
[1] 13

# get MSE  
hit\_gbm1$cv.error[best]

[1] 0.2372428

We can also use gbm.perf() function to plot the CV errors as well as training errors.

gbm.perf(hit\_gbm1, method = "cv")

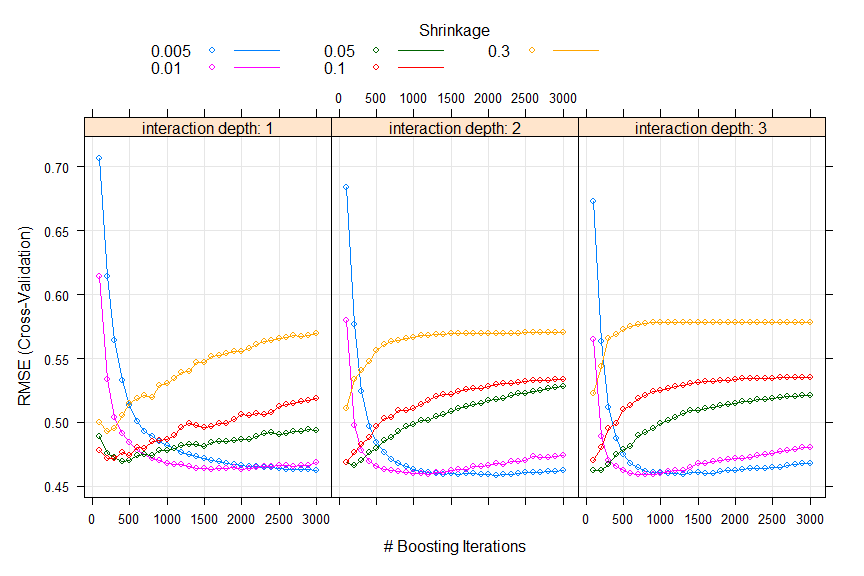


Training (black) and test errors (green) for GBM fit on Hitters data.

[1] 13

We can also tune for parameters either manually, or using caret, as we show below. Here shrinkage is the *learning rate* .

set.seed(1001)  
gr <- expand.grid(shrinkage = c(0.3, 0.1, 0.05, 0.01, 0.005),  
 interaction.depth = c(1,2,3),  
 n.trees = seq(100, 3000, by=100),  
 n.minobsinnode = 10)  
  
caret\_gbm <- train(Log\_Salary ~ .,  
 data = Hitters,  
 method = "gbm",  
 trControl = trainControl(method = "cv",   
 number = 5),  
 tuneGrid = gr,  
 verbose = FALSE)  
  
plot(caret\_gbm)



Tuning gradient boosting machine using caret.

# best parameters  
best <- which.min(caret\_gbm$results$RMSE)  
best

[1] 306

caret\_gbm$results[best,]

shrinkage interaction.depth n.minobsinnode n.trees RMSE Rsquared  
51 0.005 2 10 2100 0.4586201 0.7291873  
 MAE RMSESD RsquaredSD MAESD  
51 0.3214669 0.1068343 0.1475975 0.04424383

### XGBoost

Another efficient and flexible gradient boosting library is extreme gradient boosting (XGBoost). It is optimized for distributed computing and portable across multiple languages such as R, Python, Julia, Scala, Java, and C++. XGBoost also provides a few advantages over traditional boosting:

* Regularization: XGBoost offers additional regularization techniques that provides added protection against overfitting.
* Early stopping: XGBoost implements early stopping so that we can stop model assessment when additional trees offer no improvement.
* Loss functions: XGBoost allows users to define and optimize gradient boosting models using custom objective and evaluation criteria.
* Continue with existing model: A user can train an XGBoost model, save the results, and later on return to that model and continue building onto the results.
* Different base learners: XGBoost also provides boosted generalized linear models.

XGboost can be implemented multiple ways within R: using xgboost package, using caret as a meta engine, or using h2o package.

Although we discussed the most popular GBM algorithms, there are alternative algorithms such as LightGBM (Ke, Guolin, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A Highly Efficient Gradient Boosting Decision Tree. In Advances in Neural Information Processing Systems, 3146-54.) and CatBoost (Dorogush, Anna Veronika, Vasily Ershov, and Andrey Gulin. 2018. CatBoost: Gradient Boosting with Categorical Features Support. arXiv Preprint arXiv:1810.11363.). LightGBM is a gradient boosting framework that focuses on leaf-wise tree growth versus the traditional level-wise tree growth. As a tree is grown deeper, it focuses on extending a single branch versus growing multiple branches. CatBoost develops efficient methods for encoding categorical features during the gradient boosting process.

### Boosting Classification Trees

Consider a two-class problem, with the output variable coded as and . Given a vector of predictor variables , a classifier, produces a prediction taking one of the two values and .

Recall that a weak classifier is one whose error rate is only slightly better than random guessing. Boosting in classification works by sequentially applying the weak classification algorithm to repeatedly modified versions of the data, thereby producing a sequence of weak classifiers, .

The predictions from all of them are then combined through a weighted majority vote to produce the final prediction:

Here are computed by the boosting algorithm, and weight the contribution of each respective weak classifier. $sign()` takes on -1 for a negative input and 1 for a positive input (0 for a 0 input).

* Their effect is to give higher influence to the more accurate classifiers in the sequence.
* The data modifications at each boosting step consist of applying weights to each of the training observations .
* Initially, we take all of the weights to be . Thus the first step simply trains the classifier on the data in the usual manner.
* Then for iteration , the algorithm modifies the weights so that the misclassified observation in step have their weights increased whereas the weights are decreased for those that were classified correctly.
* The classification algorithm is reapplied to the weighted observations. Thus as we go through more iterations, observations that are difficult to classify correctly receive ever-increasing influence.
* Each successive classifier is thereby forced to concentrate on those training observations that are missed by previous ones in the sequence.

The details of the AdaBoost.M1 algorithm often used is:

1. Initialize the observation weights
2. For to :
   1. Fit a classifier to the training data using weights , and obtain predictions .
   2. Compute the misclassification error rate
   3. Compute .
   4. Set
3. Output for any new observation .

The AdaBoost.M1 algorithm is known as “Discrete AdaBoost” in Friedman et al. (2000), because the base classifier returns a discrete class label. - If the base classifier instead returns a real-valued prediction (e.g., a probability), AdaBoost can be modified appropriately, see “Real AdaBoost” in Friedman et al. (2000).

Notice that the algorithm above can take *any* base classifier, not just classification tree. However, decision trees are an ideal base learner for data mining applications of boosting due to various advantages such as their natural handling of mixed (numerical and categorical) data, handling of missing values, robustness to outliers and monotone transformations in *input space*, computational scalability for large , etc.

As with the regression setting, when trees are used as the base learner, we have two tuning parameters:

* = tree depth (or interaction depth)
* = the number of iterations

We can also adapt gradient boosting for classification using a general loss function such as the Bernoulli distribution, where we model the odds. #### Fitting a Boosted Classification Tree

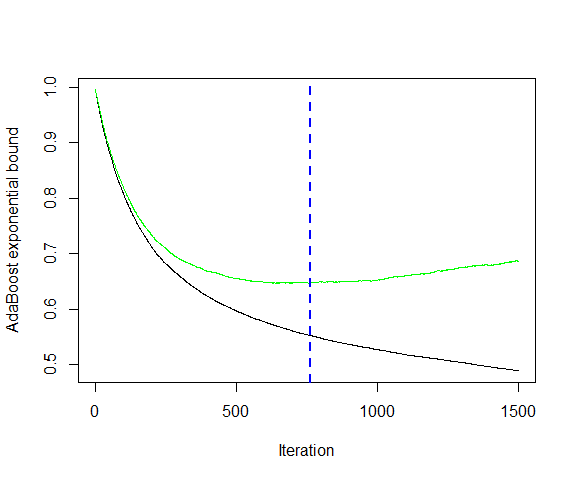
The primary boosted tree package in R is gbm, which implements stochastic gradient boosting.

* The primary difference between boosting regression and classification trees is the choice of the distribution of the data.
  + The gbm function can only accommodate two class problems and using distribution = "bernoulli" is an appropriate choice here.
  + Another option is distribution = "adaboost" to replicate the loss function used by that methodology.
  + One complication when using gbm for classification is that it expects that the outcome is coded as 0/1.

heart <- read.csv("https://www.statlearning.com/s/Heart.csv", header = TRUE)  
# Remove the row numbers, and NAs  
heart <- heart[,-1]  
heart <- na.omit(heart)  
heart$AHD <- ifelse(heart$AHD == "Yes", 1, 0)  
heart$ChestPain <- as.factor(heart$ChestPain)  
heart$Thal <- as.factor(heart$Thal)  
dim(heart)

[1] 297 14

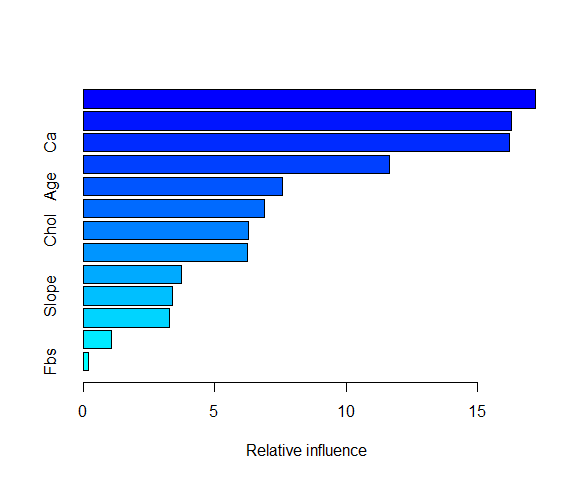
set.seed(1001)  
heart\_ada <- gbm(AHD ~ ., data = heart,  
 distribution = "adaboost",  
 interaction.depth = 1,  
 n.trees = 1500,  
 shrinkage = 0.01,  
 verbose = FALSE,  
 cv.folds = 5)  
gbm.perf(heart\_ada)



Test error estimated by CV for hearts data using adaboost loss.

[1] 761

summary(heart\_ada)

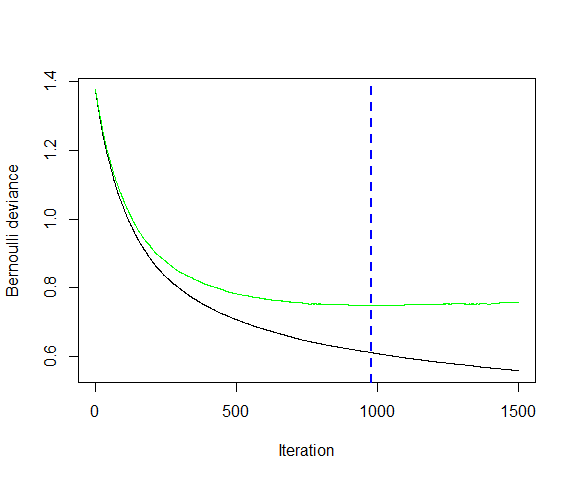


Relative incluence of preecitors in reducing the loss function in hearts data using adaboost loss.

var rel.inf  
ChestPain ChestPain 17.1945328  
Thal Thal 16.2746756  
Ca Ca 16.2185685  
Oldpeak Oldpeak 11.6460202  
Age Age 7.5651527  
MaxHR MaxHR 6.8791140  
Chol Chol 6.2987601  
RestBP RestBP 6.2336475  
Sex Sex 3.7340304  
Slope Slope 3.4090655  
ExAng ExAng 3.2718838  
RestECG RestECG 1.0615683  
Fbs Fbs 0.2129807

We can also use “bernoulli” loss and stochastic GBM.

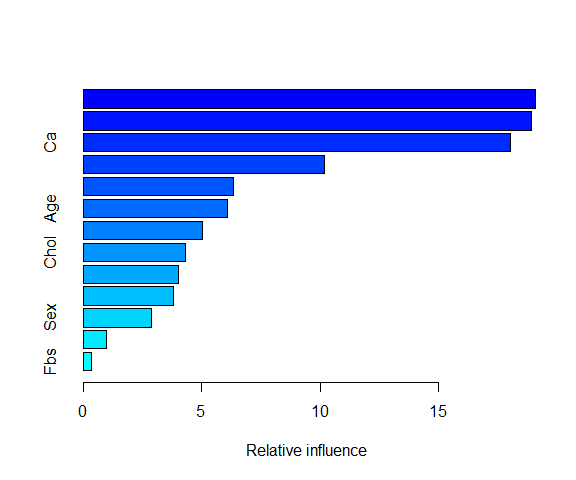
set.seed(1001)  
heart\_gbm <- gbm(AHD ~ ., data = heart,  
 distribution = "bernoulli",  
 interaction.depth = 1,  
 n.trees = 1500,  
 shrinkage = 0.01,  
 verbose = FALSE,  
 cv.folds = 5)  
gbm.perf(heart\_gbm)



Test error estimated by CV for hearts data using bernoulli loss.

[1] 977

summary(heart\_gbm)



Relative incluence of preecitors in reducing the loss function in hearts data using bernoulli loss.

var rel.inf  
ChestPain ChestPain 19.0881047  
Thal Thal 18.9221487  
Ca Ca 18.0450914  
Oldpeak Oldpeak 10.1864796  
MaxHR MaxHR 6.3337764  
Age Age 6.0669566  
RestBP RestBP 5.0320954  
Chol Chol 4.2931006  
Slope Slope 4.0085639  
ExAng ExAng 3.8265112  
Sex Sex 2.8913726  
RestECG RestECG 0.9674834  
Fbs Fbs 0.3383156

The original AdaBoost algorithm is available in the ada package. Another function for boosting trees is blackboost in the mboost package. This package also contains functions for boosting other types of models (such as logistic regression) as does the bst package.

## Bayesian additive regression trees

Like other ensemble methods discussed so far, Bayesian additive regression trees (BART) relies on a collection of trees to form a prediction.

* BART uses Bayesian methodology and builds upon earlier research on Bayesian methods for CART (Chipman HA, George EI, McCulloch RE (1998). Bayesian CART Model Search. Journal of the American Statistical Association, 93(443), 935-948.)
* In BART, 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 us start with a continuous outcome and regression problem. To start with, let us introduce some notations:

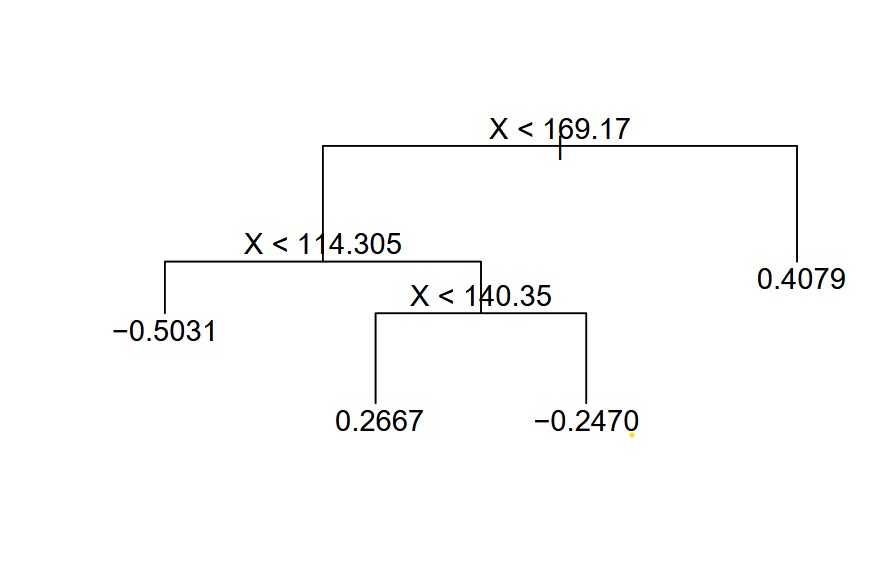
In the first iteration of the BART algorithm, all trees are initialized to have a single root node, with , and consequently .

In subsequent iterations, BART updates each of the K trees, one at a time.

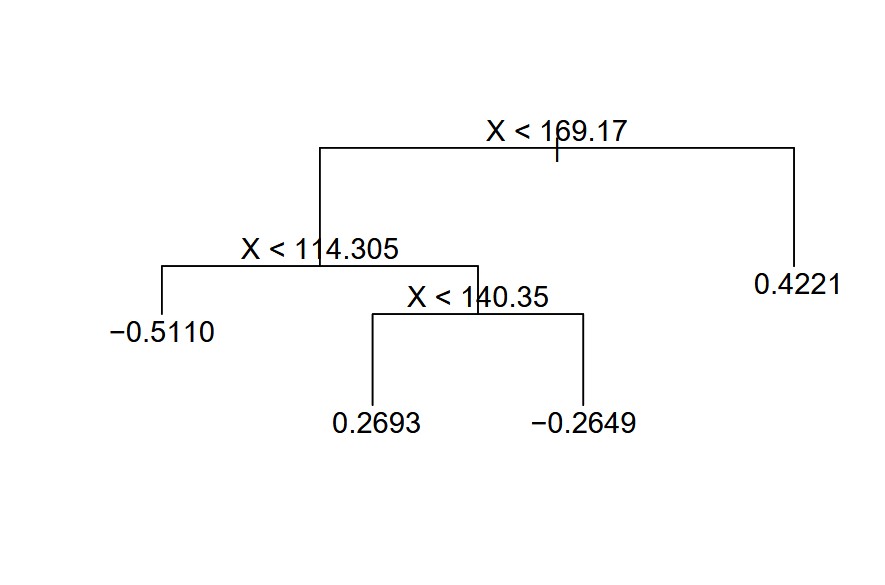
* In the -th iteration, to update the -th tree, we subtract from each response value the predictions from all but the -th tree, in order to obtain a partial residual for each observation,
* However, we do not fit a fresh tree to this partial residual. Instead, BART randomly chooses a *perturbation* to the tree from the previous iteration from a set of possible perturbations. There are two components to this perturbation:
* We may change the structure of the tree by adding or pruning branches.
* We may change the prediction in each terminal node of the tree.

We favor the perturbations that improve the fit to the partial residual.

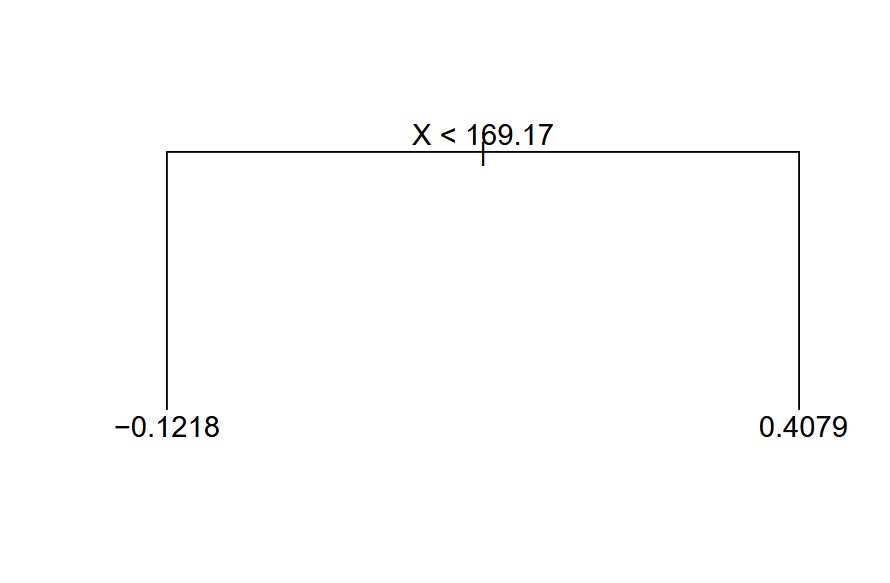
The following figures illustrates examples of possible perturbations to a tree.



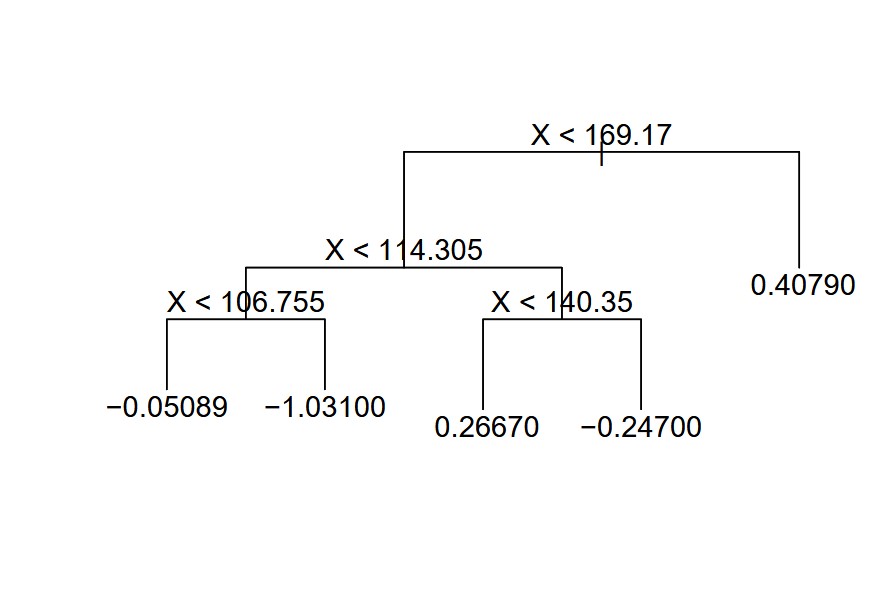
Original tree



Same shape but different prediction tree



Pruned tree



Tree with additional branches

The first plot in the figure is the -th tree in iteration . The remaining plots are possible perturbations of this tree that can be chosen in iteration as the -th tree.

* One possibility is that the new tree has the same structure as the previous tree, but with different predictions at the terminal nodes (second plot).
* Another possibility is that the new tree is obtained from pruning the previous tree (third plot).
* Yet another option is that the new tree may have more terminal nodes than the old tree (fourth plot).

After iterations, we have a collection of prediction models, . Typically, models obtained in the first few iterations tend to not perform well, we typically throw away the first few prediction models. In Bayesian literature, this is known as the burn-in period.

Then, to obtain a single prediction, we simply take the average (or other quantities such as percentiles, a measure of uncertainty in the final prediction) after the burn-in iterations.

Formally, if we throw away the first iterations as burn-in, out final prediction for input would be

A key element of the BART approach is that 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.

* Roughly speaking, 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.

The code chunk below shows BART fit to the Hitters data using the BART package. There are other packages such as bartMachine, which is also supported by caret.

library(BART)  
set.seed(1001)  
hit\_bart <- wbart(x.train = as.matrix(Hitters[,1:13]), y.train = Hitters$Log\_Salary,  
 ntree = 200, ndpost = 1000, nskip = 200)

\*\*\*\*\*Into main of wbart  
\*\*\*\*\*Data:  
data:n,p,np: 263, 13, 0  
y1,yn: 0.236093, 0.980534  
x1,x[n\*p]: 315.000000, 249.000000  
\*\*\*\*\*Number of Trees: 200  
\*\*\*\*\*Number of Cut Points: 100 ... 100  
\*\*\*\*\*burn and ndpost: 200, 1000  
\*\*\*\*\*Prior:beta,alpha,tau,nu,lambda: 2.000000,0.950000,0.063565,3.000000,0.076233  
\*\*\*\*\*sigma: 0.625585  
\*\*\*\*\*w (weights): 1.000000 ... 1.000000  
\*\*\*\*\*Dirichlet:sparse,theta,omega,a,b,rho,augment: 0,0,1,0.5,1,13,0  
\*\*\*\*\*nkeeptrain,nkeeptest,nkeeptestme,nkeeptreedraws: 1000,1000,1000,1000  
\*\*\*\*\*printevery: 100  
\*\*\*\*\*skiptr,skipte,skipteme,skiptreedraws: 1,1,1,1  
  
MCMC  
done 0 (out of 1200)  
done 100 (out of 1200)  
done 200 (out of 1200)  
done 300 (out of 1200)  
done 400 (out of 1200)  
done 500 (out of 1200)  
done 600 (out of 1200)  
done 700 (out of 1200)  
done 800 (out of 1200)  
done 900 (out of 1200)  
done 1000 (out of 1200)  
done 1100 (out of 1200)  
time: 3s  
check counts  
trcnt,tecnt,temecnt,treedrawscnt: 1000,0,0,1000

# Variable importance  
sort(hit\_bart$varcount.mean)

CRuns CHmRun CWalks CHits CAtBat Hits AtBat CRBI Walks HmRun RBI   
11.312 12.237 12.718 13.016 13.537 15.185 15.963 18.158 19.094 19.893 20.046   
 Runs Years   
20.413 21.645

We can use predict() function to generate predictions for new data (as with pretty much all of our models!)

# Summary and Discussion of Tree Based Models

In this section, we discussed regression and classification trees using the CART approach and several ensemble learning methods.

* In bagging, we grow the trees independently on bootstrap samples of the observations. These trees tend to be quite similar to each other and hence bagging can get caught in local optima. In other words, bagging can fail to thoroughly explore the model space.
* In random forests, we grow the trees independently on bootstrap samples of the observations with the added step that each split on each tree is performed using a random subset of the features. This is done to decorrelate the trees, and to obtain a more thorough exploration of model space relative to bagging.
* In boosting, we only use the original data, and do not draw any bootstrap samples. The trees are grown successively, using a slow learning approach, governed by the learning rate. Each new tree is fit to the signal that is left over from the earlier trees, and shrunken down before it is used.
* In BART, we once again only make use of the original data, and we grow the trees successively. However, each tree is perturbed in order to avoid local minima and achieve a more thorough exploration of the model space.