Tree-Based Methods

Arnab Maity

NCSU Statistics ~ 5240 SAS Hall ~ amaity[at]ncsu.edu

Contents

Introduction 2

Regression Trees 4

Early stopping 7

Tree Pruning 7

Classification Trees 11

Advantages and Disadvantages of Trees 15

Bagging 16

Random Forests 19

Boosting 20

Regression trees 21

XGBoost 26

Classification tree 27

Stacking 31

Bayesian additive regression trees 35

Summary and Discussion 38

Regression Model Trees 39

Rule based approach 39

Introduction

Tree-based methods partition the feature space into a set of regions, and then fit a simple model, such as a constant function, in each region. They are conceptually simple yet powerful methods, and useful for interpretation.

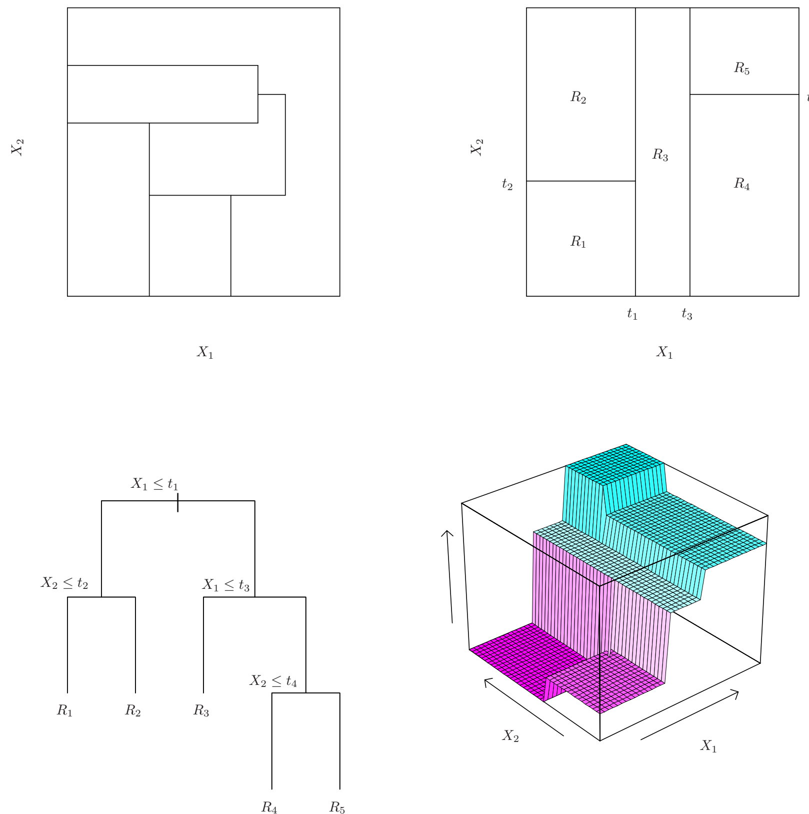


Figure 1: Top Left: A partition of two-dimensional feature space that could not result from recursive binary splitting. Top Right: The output of recursive binary splitting on a two-dimensional example. Bottom Left: A tree corresponding to the partition in the top right panel. Bottom Right: A perspective plot of the prediction surface corresponding to that tree. Figure and caption taken from *Introduction to Statistical Learning*.

Suppose we have a regression problem with continuous response Y and two predictors $\mathbf{X} = (X_1, X_2)$. Suppose $E(Y|\mathbf{X}) = f(\mathbf{X})$. The top left panel of Figure 1 shows a partition of the feature space by lines that are parallel to the coordinate axes. In each partition element we can model $f(\mathbf{X})$ with a different constant. Note that there are 5 regions, say, $R_1, \ldots, R_5$, in the top left panel of Figure 1. Since we are fitting a constant function in each region, we are modeling our regression function at any $\mathbf{X}$ as

$$f(\mathbf{X}) = \sum_{m=1}^{5} c_m I(\boldsymbol{x} \in R_m),$$

where $c_1, \ldots, c_5$ are unknown constants. Since $f(\mathbf{X})$ is constant in

each region, $\widehat{c}_m$ is simply average of Y values over the corresponding region.

However, although each partitioning line has a simple description like $X_1 = c$, some of the resulting regions are complicated to describe, and interpret. 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, see top right panel of Figure 1.

In general, we first split the space into two regions, and model the response by the mean of Y in each region. We choose the variable and split-point to achieve the best fit.[1] Then one or both of these regions are split into two more regions, and this process is continued, until some stopping criterion is applied. For example, in the top right panel of 1, we first split at whole space into regions $X_1 \le t_1$ and $X_1 > t_1$. Then the region $X_1 \le t_1$ is split into two according to $X_2 \le t_2$ and $X_2 > t_2$. Similarly, and the region $X_1 > t_1$ is split into two: $X_1 \le t_3$ and $X_1 > t_3$. Finally, the region $X_1 > t_3$ is split again at $X_2 = t_4$. The corresponding regression model predicts Y with a constant c_m in region R_m can be written as before:

[1] We will describe what "best fit" means in this context in the next section.

$$\widehat{f}(\mathbf{X}) = \sum_{m=1}^{5} \widehat{c}_m I(\mathbf{X} \in R_m),$$

where $\widehat{c}_m = average(Y_i|\mathbf{X}_i \in R_m)$. As an example, the bottom right panel of Figure 1 is a perspective plot of the regression surface from this model.

The same model described above can be represented by a binary tree, see the bottom left panel of Figure 1. 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 $R_1, \ldots, R_5$. This is the reason we call such methods *decision tree* methods. Such trees can be used for classfication problems as well. Figure 2 shows a basic (classification) tree and the corresponding terminology used for any tree.

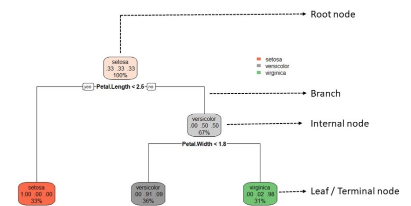


Figure 2: Terminology ralated to a decision tree.

A key advantage of the binary tree is its interpretability. The feature space partition is fully described by a single tree. When there are more than two inputs, it is difficult to draw partitions like that in the top right panel of Figure 1, but the binary tree representation works in the same way. For example, a binary tree for the involving more than two predictors is shown in Figure 3.

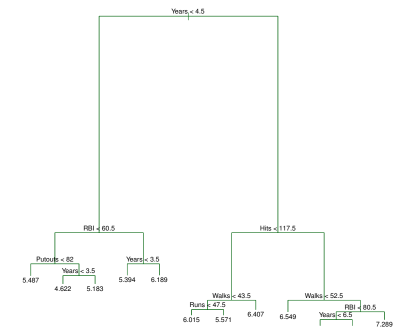


Figure 3: A binary tree for the involving more than two predictors. Figure taken from *Introduction to Statistical Learning*.

Regression Trees

Suppose we have p predictors, $\mathbf{X} = (X_1, \ldots, X_p)$ and a continuous response Y. Basic regression trees aim to partition the data into smaller regions that are more homogeneous with respect to the response. Recall that, as we are growing the tree, we only split a region into two using only one predictor at a split point. To achieve outcome homogeneity, regression trees need to determine:

- The predictor X_j to split on and split point s, and
- The depth or complexity of the tree.

There are many techniques for constructing regression trees. Perhaps the most utilized method is the classification and regression tree (CART) methodology.[2] We first discuss the CART algorithm.

[2] Breiman, L., Friedman, J., Olshen, R. and Stone, C. (1984). Classification and Regression Trees, Wadsworth, New York.

Suppose first that we have partitioned the data into M regions $R_1, R_2, \ldots, R_M$. Thus $\widehat{c}_m = average(Y_i|\mathbf{X}_i \in R_m)$. Therefore, for this configuration of regions, we have the residual sum of squares

$$RSS = \sum_{m=1}^{M} \sum_{i:\mathbf{X}_i \in R_m} (Y_i - \widehat{c}_m)^2.$$

In tree-based methods, we construct the regions dynamically from the data. Thus one might try to regions $R_1, \ldots, R_M$ 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 M regions. For this reason, we take a *top-down*, *greedy* approach that is known as *recursive binary splitting*.

For regression, 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 j-th predictor, and split point s, we have two regions: $R_1(j,s) = \{\mathbf{X}|X_j \leq s\}$ and $R_2(j,s) = \{\mathbf{X}|X_j > s\}$, and the corresponding sum pf squares

$$RSS(j,s) = \sum_{i:\mathbf{X}_i \in R_1} (Y_i - \widehat{c}_1)^2 + \sum_{i:\mathbf{X}_i \in R_2} (Y_i - \widehat{c}_2)^2.$$

Note that we have indexed the regions by j and s since they depend on the splitting variable X_j and the split point s. Thus the resulting RSS is also indexed by (j,s). Now we find the best value of j and s that minimize $RSS(j,s)$. Then within each of the two regions, we apply the same method and search for the predictor and split point that best reduces RSS, and so on. For each splitting variable, the determination of the split point s can be done very quickly and hence by scanning through all of the inputs, determination of the best pair (j,s), for each region, is feasible.

Let us now go though the process of building a regression tree. Consider the `Hitters` data set in the `ISLR2` library 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. Figure 4 shows plots of `Salary` vs `Years` and `Hits`.

```
library(ISLR2)
dim(Hitters)
```

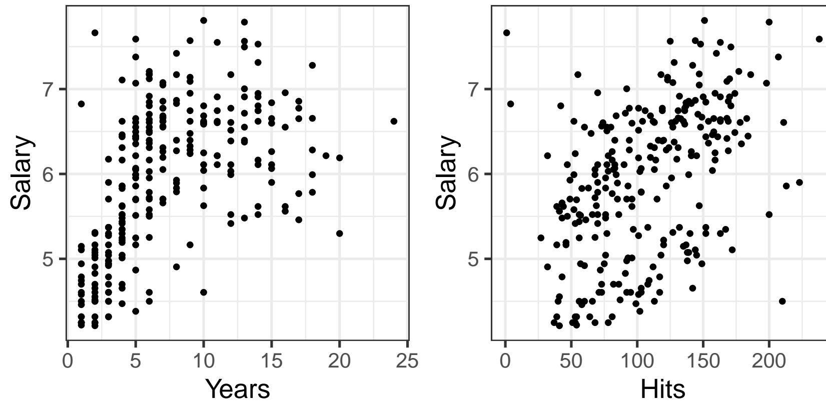
```
## [1] 322  20
```

```
Hitters <- na.omit(Hitters)
Hitters$Salary <- log(Hitters$Salary)
dim(Hitters)
```

```
## [1] 263  20
```

```
head(Hitters)
```

```
##                    AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun
## -Alan Ashby          315   81     7   24  38    39    14   3449   835     69
## -Alvin Davis         479  130    18   66  72    76     3   1624   457     63
## -Andre Dawson        496  141    20   65  78    37    11   5628  1575    225
## -Andres Galarraga    321   87    10   39  42    30     2    396   101     12
## -Alfredo Griffin     594  169     4   74  51    35    11   4408  1133     19
## -Al Newman           185   37     1   23   8    21     2    214    42      1
##                    CRuns CRBI CWalks League Division PutOuts Assists Errors
## -Alan Ashby          321  414    375      N        W     632      43     10
## -Alvin Davis         224  266    263      A        W     880      82     14
## -Andre Dawson        828  838    354      N        E     200      11      3
## -Andres Galarraga     48   46     33      N        E     805      40      4
## -Alfredo Griffin     501  336    194      A        W     282     421     25
## -Al Newman            30    9     24      N        E      76     127      7
##                      Salary NewLeague
## -Alan Ashby        6.163315         N
## -Alvin Davis       6.173786         A
## -Andre Dawson      6.214608         N
## -Andres Galarraga  4.516339         N
## -Alfredo Griffin   6.620073         A
## -Al Newman         4.248495         A
```



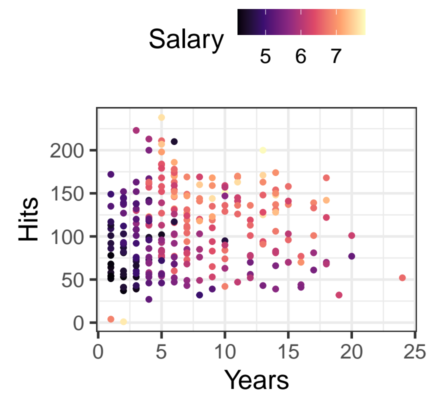


Figure 4: Salary vs Years and Hits in the Hitters data.

Now we determine the variable and the corresponding split point. Fig. 5 shows the RSS for the continuum of splits for `Years`. The optimal split point for this variable is 4.5. 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. In our example, the `Years` variable was chosen to be the best, and the resulting tree is in Figure 6. 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.

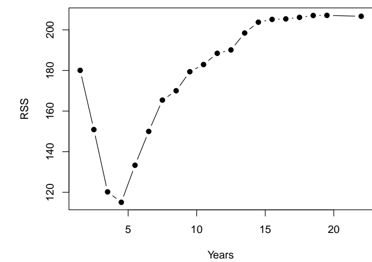


Figure 5: RSS for different split points for Years. The minimum occurs at 4.5.

Next, we split each of the regions into two using the same algorithm as above. The resulting three is shown in Figure 7. 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.

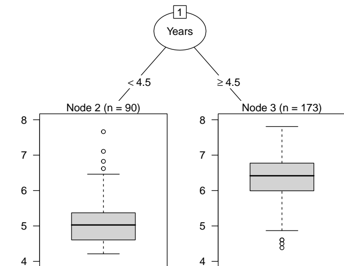


Figure 6: Splitting the initial data into two regions.

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.

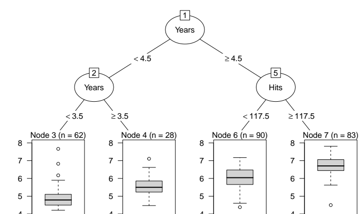


Figure 7: Splitting the regions in Figure 6 further.

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 extreeme 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, T_0, stopping the splitting process only when some minimum node size (say 5) is reached. Then *prune* this tree using *cost-complexity pruning*[3] to obtain a subtree. We define a *subtree* T, a subset of T_0, to be any tree that can be obtained by pruning T_0, that is, collapsing any number of its internal nodes. We index terminal nodes by m, with node m representing region R_m. Let $|T|$ denote the number of terminal nodes in T. We consider a sequence of trees indexed by a nonnegative tuning parameter α, such that,

$$RSS(\alpha) = \sum_{m=1}^{|T|} \sum_{\mathbf{X}_i \in R_m} (Y_i - \widehat{c}_m)^2 + \alpha\, |T|$$

is minimized. The tuning parameter α is called the *complexity parameter*, and it controls a trade-off between the subtree's complexity and its fit to the training data. When $\alpha = 0$, then the subtree T will simply equal T_0. However, as α increases, there a tree with many terminal nodes will have larger $RSS(\alpha)$, and so the quantity will tend to be minimized for a smaller subtree. It turns out that as we increase α 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 α is easy.[4] We can select a value of α using holdout or cross-validation. Breiman et al. (1984) also propose using the one-standard-error rule on the optimization criteria for identifying the simplest tree: find the smallest tree that is within one standard error of the tree with smallest absolute error. Another approach to select

[3] Also known as weakest link pruning.

[4] Breiman et al. (1984). Classification and Regression Trees. Chapman and Hall, New York.

the tree size is to choose with the numerically smallest error. We then return to the full data set and obtain the subtree corresponding to α.

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, and the function of the same name. 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[5] First we grow a large tree and look at the optimal subtrees for each value of the complexity parameter (denoted by `cp` in rpart).

[5] See `?rpart.control` for details. Also see `https://cran.r-project.org/web/packages/rpart/index.html` for an introduction to rpart functionality.

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

Here `xval=10` specifies that we are using 10-fold CV to estimate the prediction error corresponding to each value of the complexity parameter. Also, `minbucket = 2` indicates that the minimum number of observations in a terminal node must be 2. Finally, `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 α. Instead, it uses the following formula: for a subtree T,

$$RSS(cp) = \sum_{m=1}^{|T|} \sum_{\mathbf{X}_i \in R_m} (Y_i - \widehat{c}_m)^2 + cp\,|T|\,RSS(T_1),$$

where T_1 is a tree with no splits. Thus `cp` is a scaled, unit less, version of α. 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 R^2 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 $1 - R^2$.

```
printcp(T0)
```

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

```
##
## Regression tree:
## rpart(formula = 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
##
```

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

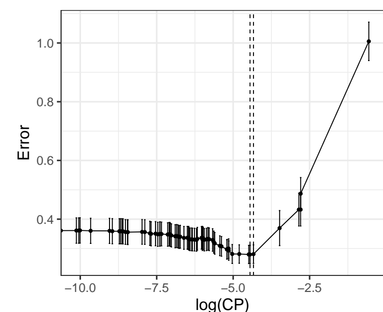


Figure 8: Cross-validated errors vs log of complexity parameter values.

```
##           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)
```

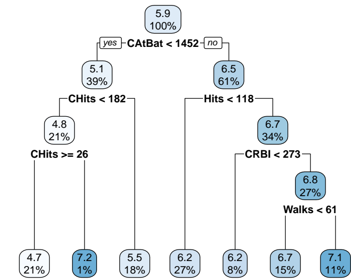
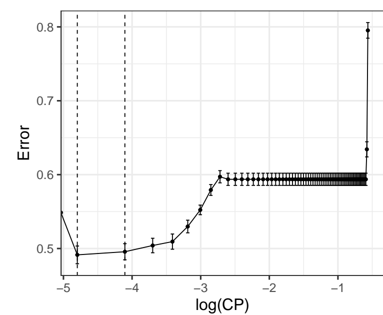


Figure 9: Final pruned tree for the Hitters data.

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

```
library(caret)
set.seed(1001)
hit_tree <- train(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)
```

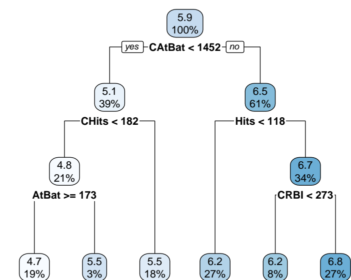


Figure 10: Pruned tree for Hitters data using caret.

Overall, the following steps are used to choose α 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 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 k-th fold, as a function of α.

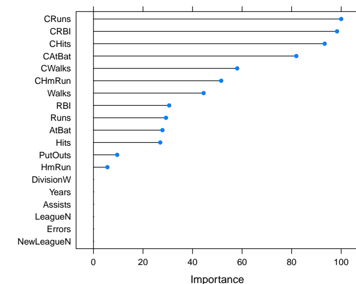
   Average the results for each value of α, and pick α to minimize the average error.
4. Return the subtree from Step 2 that corresponds to the chosen value of α.

The CART methodology can also handle 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.

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.[6] 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.

[6] Breiman et al. (1984)

```
plot(varImp(hit_tree))
```



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

In this setting, the outcome variable is a categorical variable taking possible values $1, \ldots, K$. In this case, we predict that each observation belongs to the most commonly occurring class of training observations in the region to which it belongs. 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 regresion trees, the only changes needed in the tree algorithm are the criteria for splitting nodes and pruning the tree.[7]

[7] Squared-error criterion might not be suited for a classification problem.

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 m, representing a region R_m, let $\widehat{p}_{mk}$ be the proportion of training observations in R_m that are from the k-th class. We classify the observations in node m to the majority class in node m, that is, the value of k that maximizes $\widehat{p}_{mk}$. Based on this observation, we can look at three different measures of node impurity:

- Misclassification error: $1 - \max_k \widehat{p}_{mk}$,
- Gini index: $\sum_{k=1}^{K} \widehat{p}_{mk}(1 - \widehat{p}_{mk})$,
- Cross-entropy or deviance: $-\sum_{k=1}^{K} \widehat{p}_{mk} log(\widehat{p}_{mk})$.

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 K classes. Gini index takes on a small value if all of the $\widehat{p}_{mk}$ are close to zero or one - a small value indicates that a node contains predominantly observations from a single class. The third measure, Cross-entropy, will take on a value near zero if all of the $\widehat{p}_{mk}$ are close to zero or one. Therefore, like the Gini index, the entropy will take on a small value if the m-th node is pure. In fact, it turns out that the Gini index and the entropy are quite similar numerically. Thus, when building a classification tree we would use either Gini or cross-entropy criteria.

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.

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$AHD <- as.factor(heart$AHD)
dim(heart)
```

```
## [1] 297  14
```

```
head(heart)
```

```
##   Age Sex    ChestPain RestBP Chol Fbs RestECG MaxHR ExAng Oldpeak Slope Ca
## 1  63   1      typical    145  233   1       2   150     0     2.3     3  0
## 2  67   1 asymptomatic    160  286   0       2   108     1     1.5     2  3
## 3  67   1 asymptomatic    120  229   0       2   129     1     2.6     2  2
## 4  37   1   nonanginal    130  250   0       0   187     0     3.5     3  0
## 5  41   0   nontypical    130  204   0       2   172     0     1.4     1  0
## 6  56   1   nontypical    120  236   0       0   178     0     0.8     1  0
##         Thal AHD
## 1      fixed  No
## 2     normal Yes
## 3 reversable Yes
## 4     normal  No
## 5     normal  No
## 6     normal  No
```

The textbook demonstrates classification trees using the `tree` library. Here, as before, we will use `rpart` for demonstration. We proceed in a similar way as we did for regression tree with the only change is the use of `method='class'` and `parms = list(split = "information")`. The first command specifies the type of problem (classification), and second command sets the splitting critarion 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 $1 - NIR$, 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[8] so that the first node has an error of 1. Using 1-SE rule, we chose the tree with three splits, and corresponding tree is shown in Figure 11.

[8] 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.

```
cp <- heart_rpart$cptable
heart_final <- prune(heart_rpart, cp = cp[3,1])
rpart.plot(heart_final)
```

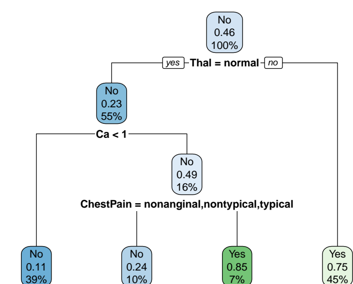


Figure 11: Pruned tree using 1-SE rule for the heart data.

```
# 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, see Figure 12.

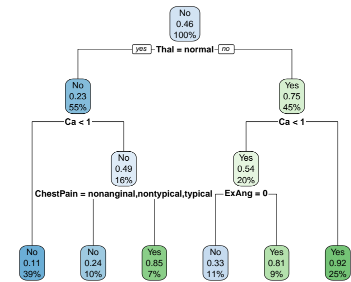


Figure 12: Pruned tree using minimum cp for the heart data.

```
heart_final <- prune(heart_rpart, cp = cp[4,1])
rpart.plot(heart_final)
```

```
# 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 75% data coming from `Yes` class. After splitting, 92% 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 orginal node, and if a test observation falls into the region, then its response value is probably `Yes`, but we are much less certain.

Advantages and Disadvantages of Trees

Decision trees for regression and classification have a number of advantages over the more classical approaches that we have discussed before. Trees are very easy to explain to people, perhaps 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. For example, the tree representation is also popular among medical scientists, perhaps because it mimics the way that a doctor thinks. The tree stratifies the population into strata of high and low outcome, on the basis of patient characteristics. One main advantage of trees is that they can be displayed graphically, and are easily interpreted even by a non-expert - this is especially true for small trees. Also, trees

can easily handle categorical predictors without the need to create dummy variables.

Tree, however, suffer from two main disadvantages. Trees generally do not have the same level of predictive performance as some of the other regression and classification approaches we have seen before. Additionally, trees can be very non-robust; a small change in the data can cause a large change in the final estimated tree. However, by *aggregating* many decision trees, using methods like *bagging*, *random forests*, and *boosting*, the predictive performance of trees can be substantially improved.

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 for classification (can be also applied for regression), 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 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

We discussed the bootstrap as a way of assessing the accuracy of a parameter estimate or a prediction, or to estimate test error of a learning method. Here we use the bootstrap to improve the prediction based on a tree model. *Bootstrap aggregation* or *Bagging* is a general approach that uses bootstrapping in conjunction with *any* regression or classification model to construct an ensemble.

Consider the regression problem where we fit a model to our training data $(Y_1, \mathbf{X}_1), \dots, (Y_n, \mathbf{X}_n)$, and obtaining the prediction $\widehat{f}(\boldsymbol{x})$ at an input $\boldsymbol{x}$. Bagging averages this prediction over a collection of bootstrap samples, thereby reducing its variance. For each bootstrap sample, $b = 1, \dots, B$ of the original data, we fit our model, giving

prediction $\widehat{f}^{*b}(\boldsymbol{x})$. The bagging estimate is defined by

$$\widehat{f}_{\text{bag}}(\boldsymbol{x}) = \frac{1}{B}\sum_{b=1}^{B}\widehat{f}^{*b}(\boldsymbol{x}).$$

Note that this approach can be taken for any regression model, e.g., smoothing splines.

For a regression tree, where $\widehat{f}(\boldsymbol{x})$ denotes the tree's prediction at input vector $\boldsymbol{x}$. Each bootstrap tree will typically involve different features than the original, and might have a different number of terminal nodes. The bagged estimate is the average prediction at $\boldsymbol{x}$ from these B trees. These trees are grown deep, and are not pruned. Hence each individual tree has high variance, but low bias. Averaging these B trees reduces the variance. Bagging has been demonstrated to give impressive improvements in accuracy by combining together hundreds or even thousands of trees into a single procedure.

For a classification tree, each of the B trees will predict a class for the new observation $\boldsymbol{x}$. The bagged classifier selects the class with the most "votes" from the B trees.

There are several packages in R to perform bagging, such as `randomForest` and `ipred`. We ca also use `caret` for this purpose as well. Let us demonstrate bagging using the `randomForest()` function in the package with the same name. We will use the `Hitters` data as used above.

```
library(randomForest)
set.seed(1001)
hit_bagged <- randomForest(Salary ~ .,
                           data = Hitters,
                           mtry = ncol(Hitters)-1,
                           importance = TRUE)
print(hit_bagged)
```

```
##
## Call:
##  randomForest(formula = Salary ~ ., data = Hitters, mtry = ncol(Hitters) -      1, importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 19
##
##           Mean of squared residuals: 0.1913966
##                     % Var explained: 75.7
```

As we will see shortly, bagging is a special case of *random forests*, where instead of scanning through all the predictors, only a random

sample of m predictors is chosen when considering a split. The argument `mtry` specifies m. Thus bagging is special case of random forests when $m = p$. The argument `mtry` indicates that all 19 predictors should be considered for each split of the tree—in other words, that bagging should be done. Now we can predict a new observation by using the `predict()` function.

```
newx <- Hitters[1,]
pred <- predict(hit_bagged, newdata = newx)
pred
```

```
## -Alan Ashby
##    6.169975
```

We can estimate the test error of a bagged model using out-of-bag (OOB) observations.[9] 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. Thus the OOB observations can serve as a test set. We can predict the response for the i-th observation using each of the trees in which that observation was OOB. This will yield around $B/3$ predictions for the i-th observation. In order to obtain a single prediction for the i-th observation, we can average these predicted responses (for regression model) or can take a majority vote (for classification models). This leads to a single OOB prediction for the i-th observation. An OOB prediction can be obtained in this way for each of the n observations. Now we can estimate the overall OOB MSE or misclassification rate.

[9] Recall our discussion about OOB samples in the chapter about data splitting methods.

Figure 13 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.

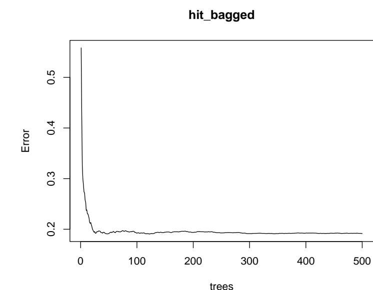


Figure 13: Estimated test MSE for different number of aggregated trees.

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. Using the `importance()` function, we can view the importance of each variable. Two measures of variable importance are reported. The first is based upon the mean decrease of accuracy in predictions on the out of bag samples when a given variable is permuted. The second is a measure of the total decrease in node impurity that results from splits over that variable, averaged over all trees. In the case of regression trees, the node impurity is measured by the training RSS, and for classification trees by the deviance. We

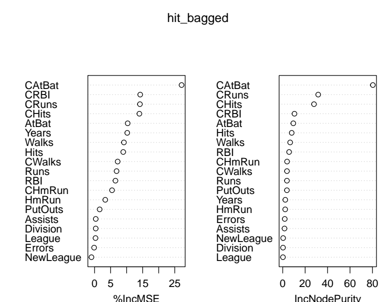


Figure 14: Plots of importance measures.

can use the `varImpPlot()` function to plot these importance metrics.

Random Forests

Random forests provide an improvement over bagging by decorrelating the trees. Consider the situation where there is one very strong predictor in the data set, along with a number of other moderately strong predictors. Then most or all of the trees in the collection of bagged trees will use the strong predictor in the top split. Thus, all of the bagged trees will look quite similar to each other, and the predictions from the bagged trees will be highly correlated. Averaging many such highly correlated predictions does not lead to much reduction in variance. In particular, this means that bagging will not lead to a substantial reduction in variance over a single tree in this setting.

Random forests overcome this problem by forcing each split to consider only a subset of the predictors. 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 sample of m predictors is chosen as split candidates from the full set of p predictors.[10] The split is allowed to use only one of those m predictors. For each split, a new sample 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. However, we can treat m as a tuning parameter if needed.

[10] Therefore bagging is a special case of random forest when $m = p$.

By adopting such a strategy, on average $(p - m)/p$ of the splits will not even consider the strong predictor, and so other predictors will have more of a chance. This if how random forest decorrelates the trees, and thereby making the average of the resulting trees less variable and hence more reliable.

We use the `Hitters` data again to demonstrate random forest. We effectively use the same command as bagging, but with a smaller value for `mtry`.

```
set.seed(1001)
hit_rf <- randomForest(Salary ~ .,
                           data = Hitters,
                           mtry = 5,
                           importance = TRUE)
print(hit_rf)
```

```
##
## Call:
```

```
##  randomForest(formula = Salary ~ ., data = Hitters, mtry = 5,      importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 0.1766532
##                     % Var explained: 77.57
```

All the diagnostics we presented in the bagging section apply here as well, including variable importance plot.

Let us look at a classification example using the `hearts` data.

```
set.seed(1001)
heart_rf <- randomForest(AHD ~ ., data = heart,
                         mtry  = 4,
                         importance = TRUE)
print(heart_rf)
```

```
##
## Call:
##  randomForest(formula = AHD ~ ., data = heart, mtry = 4, importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
##         OOB estimate of  error rate: 19.19%
## Confusion matrix:
##      No Yes class.error
## No  136  24   0.1500000
## Yes  33 104   0.2408759
```

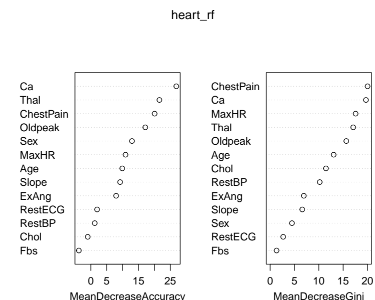


Figure 15: Variable importance plot of the hearts data.

Variable importance plots are shown in Figure 15. We can access the OOB estimates of the errors rate as follows – see Figure 16.

```
oob_error <- heart_rf$err.rate[,1]
plot(oob_error, type = "l")
```

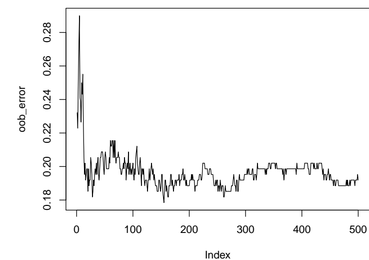


Figure 16: OOB error rates for different number of trees.

Boosting

Boosting is another way to improve predictions from a decision tree. Boosting is one of the most powerful learning ideas introduced in the last twenty years. 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 and then 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.

Regression trees

Let us start with boosting a regression tree. The we will move to classification. Historically, however, boosting for classification (algorithms such as AdaBoost.M1) appeared first. Friedman's ability to see boosting's statistical framework yielded a simple, elegant, and highly adaptable algorithm for different kinds of problems.[11]. Friedman

[11] Friedman (2001) Greedy Function Approximation: A Gradient Boosting Machine. Annals of Statistics, 29(5), 1189–1232.

named this method *gradient boosting machines* (GBM) which encompassed both classification and regression. The basic principles of gradient boosting are as follows: given a loss function (e.g., squared error for regression) and a weak learner (e.g., regression trees), the algorithm finds an *additive model* that minimizes the loss function. The algorithm is initialized with the best guess of the response (e.g., the mean of the response in regression, or even zero). The gradient (e.g., residuals) is calculated, and a model is then fit to the residuals to minimize the loss function. The current model is added to the previous model, and the procedure continues for a number of iterations that the user specified.

The following algorithm performs boosting for regression:

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

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

   (a) Fit a tree $\widehat{f}_b$ with d splits ($d + 1$ terminal nodes) to the training data $(\mathbf{X}_i, r_i), i = 1, \ldots, n$.

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

   $$\widehat{f}(\boldsymbol{x}) \leftarrow \widehat{f}(\boldsymbol{x}) + \lambda \widehat{f}_b(\boldsymbol{x}).$$

   (c) Update the residuals,

   $$r_i \leftarrow r_i - \lambda \widehat{f}_b(\mathbf{X}_i).$$

3. Output the boosted model,

$$\widehat{f}(\boldsymbol{x}) = \sum_{i=1}^{B} \lambda \widehat{f}_b(\boldsymbol{x}).$$

Unlike fitting a single large decision tree to the data, which amounts to fitting the data hard and potentially overfitting, the boosting approach instead learns slowly. We fit a tree using the current residuals, rather than the outcome, Y, 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 d in the algorithm. By fitting small trees to the residuals, we slowly improve $\widehat{f}$ 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.[12] Note that in boosting, unlike in bagging, the construction of each tree depends strongly on the trees that have already been grown.

[12] In general, statistical learning approaches that learn slowly tend to perform well.

Boosting for regression has three tuning parameters:

1. *The number of trees B*: Unlike bagging and random forests, boosting can overfit if B is too large, although this overfitting tends to occur slowly if at all. We use cross-validation to select B.

2. *The shrinkage parameter λ*: This is a small positive number. This controls the rate at which boosting learns. Typical values are 0.01 or 0.001. However, and the right choice can depend on the problem. Very small λ can require using a very large value of B in order to achieve good performance.

3. *The number d of splits in each tree*: This which controls the complexity of the boosted ensemble. Often $d = 1$ 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 d is the *interaction depth*, and controls the interaction order of the boosted model, since d splits can involve at most d variables.

Let us look at a simple example of a regression problem with one predictor:

$$Y_i = sin(X_i) + \epsilon_i,$$

where X_i takes values between $-\pi$ and π, and $\epsilon_i \sim (0, 0.25)$. Figure 17 shows how gradient boosting proceeds with estimate the true underlying function. The upper left panel ("Single tree") shows the estimated function based on a single tree pruned using cost-complexity pruning. The second plot of the first row ($B = 0$) shows the initialization for gradient boosting, that is, $\widehat{f}(x) = 0$. Thereafter, B 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. 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., $B = 1000$), the procedure shows signs of overfitting. Thus it is important to tune the parameter 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*[13]. 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, known as the *bagging fraction*. This becomes another tuning parameter for the model. It

[13] Friedman (2002) Stochastic Gradient Boosting. Computational Statistics & Data Analysis 38 (4). Elsevier: 367-78.

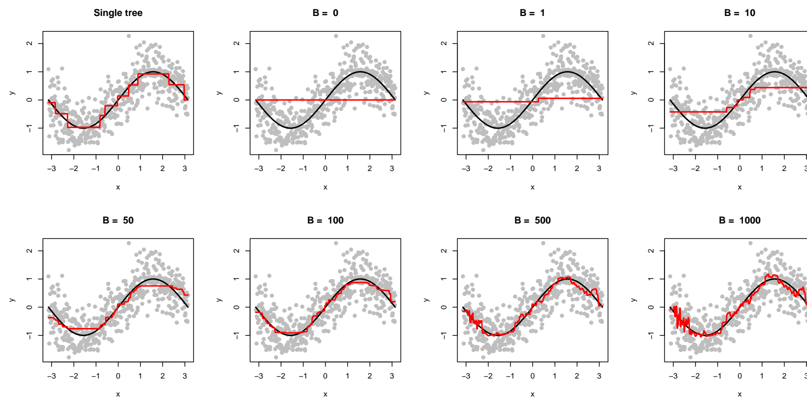


Figure 17: Demonstration of GBM using simulated data and different number of B.

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 0.5. 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, but typical values range from 0.5 to 0.8. 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.

In R, the most widely used package for boosting regression trees via stochastic gradient boosting machines is `gbm`.[14]

[14] `gbm` has two training functions: `gbm()` and `gbm.fit()`. The primary difference is that `gbm()` uses the formula interface to specify your model whereas `gbm.fit()` requires the separated `x` and `y` matrices. `gbm.fit()` is more efficient and recommended for advanced users.

```
library(gbm)
set.seed(1001)
hit_gbm1 <- gbm(
  formula = 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), $\lambda = 0.1$ (shrinkage = 0.1), $d = 3$ (interaction.depth = 3), and tree control parameter is the minimum number of observations in a node to be 10 (n.minobsinnode = 10). We also specified use of 10-fold CV (cv.folds = 10) 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 = 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.
```

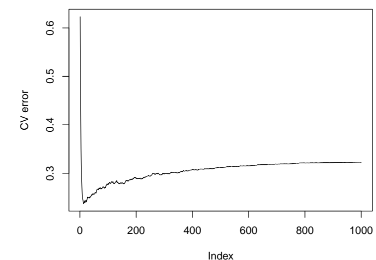


Figure 18: CV error vs number of trees in GBM

```
# 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")
```

```
## [1] 13
```

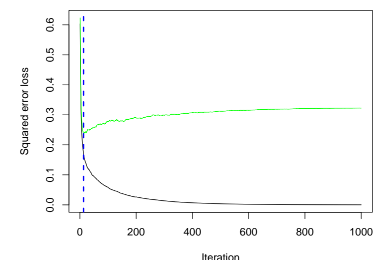


Figure 19: Training (black) and test errors (green) for GBM fit on 'Hitters' data.

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

```
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(Salary ~ .,
                   data = Hitters,
                   method = "gbm",
                   trControl = trainControl(method = "cv",
                                            number = 5),
                   tuneGrid = gr,
                   verbose = FALSE)

plot(caret_gbm)
```

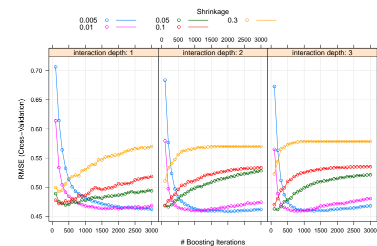


Figure 20: Tuning gradient boosting machine using caret.

```
# best parameters
best <- which.min(caret_gbm$results$RMSE)
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, and also using `h2o` package.

Although we discussed the most popular GBM algorithms, there are alternative algorithms such as `LightGBM`[15] and `CatBoost`[16]. 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.

[15] 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.

[16] Dorogush, Anna Veronika, Vasily Ershov, and Andrey Gulin. 2018. CatBoost: Gradient Boosting with Categorical Features Support. arXiv Preprint arXiv:1810.11363.

Classification tree

Several boosting algorithms appeared in early 1900's, such as, Schapire (1990)[17] and Freund (1995)[18] that implement the original theory of boosting a classification tree. Freund and Schapire (1996) and later Freund and Schapire (1996)[19] finally provided the first practical implementation of boosting theory in their famous AdaBoost/AdaBoost.M1 algorithm.

[17] Schapire R (1990). The Strength of Weak Learnability. Machine Learning, 45, 197–227.

[18] Freund Y (1995) Boosting a Weak Learning Algorithm by Majority. Information and Computation, 121, 256–285.

[19] Freund, Y. and Schapire, R. (1997). A decision-theoretic generalization of online learning and an application to boosting, Journal of Computer and System Sciences 55: 119-139.

Consider a two-class problem, with the output variable Y coded as -1 and 1. Given a vector of predictor variables $\mathbf{X}$, a classifier, $G(\mathbf{X})$ produces a prediction taking one of the two values -1 and 1. 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, $G_1(\mathbf{X}), \ldots, G_M(\mathbf{X})$. The predictions from all of them are then combined through a weighted majority vote to produce the final prediction:

$$G(\mathbf{X}) = sign\left(\alpha_1 G_1(\mathbf{X}) + \ldots + \alpha_M G_M(\mathbf{X})\right).$$

Here $\alpha_1, \ldots, \alpha_M$ are computed by the boosting algorithm, and weight the contribution of each respective weak classifier. 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 $w_1, \ldots, w_n$ to each of the training observations $(\mathbf{X}_i, Y_i), i = 1, \ldots, n$. Initially, we take all of the weights to be $w_i = 1/n$. Thus the first step simply trains the classifier on the data in the usual manner. Then for iteration $m = 2, \ldots, M$, the algorithm modifies the weights

so that the misclassified observation in step $m-1$ 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 is below.

1. Initialize the observation weights $w_i = 1/n, i = 1, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier to the training data using weights w_i, and obtain predictions $\widehat{G}_m(\mathbf{X}_1), \ldots, \widehat{G}_m(\mathbf{X}_n)$.
   (b) Compute the misclassification error rate
   $$err_m = \sum_{i=1}^{n} w_i I(Y_i \neq \widehat{G}_m(\mathbf{X}_i)) / \sum_{i=1}^{n} w_i.$$
   (c) Compute $\alpha_m = log[(1 - err_m)/err_m]$.
   (d) Set $w_i \leftarrow w_i \, exp[\alpha_m \, I(Y_i \neq G_m(\mathbf{X}_i))], i = 1, \ldots, N$.
3. Output $G(\mathbf{X}) = sign\left(\alpha_1 G_1(\mathbf{X}) + \ldots + \alpha_M G_M(\mathbf{X})\right)$. for any new observation $\mathbf{X}$.

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 n and so on. As with the regression setting, when trees are used as the base learner, we have two tuning parameters: tree depth (or interaction depth), d and number of iterations, M. We can also adapt gradient boosting for classification using a general loss function such as the Bernoulli distribution, where we model the odds.

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)
```

```
## [1] 761
```

```
summary(heart_ada)
```

```
##                 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
```

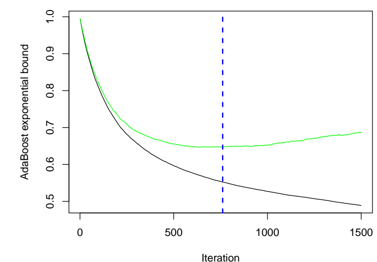


Figure 21: Test error estimated by CV for hearts data using adaboost loss.

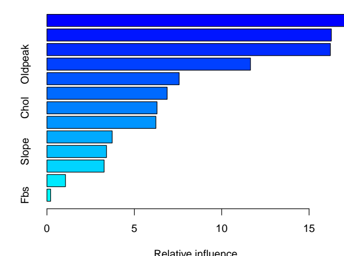


Figure 22: Relative influence of preecitors in reducing the loss function in hearts data using adaboost loss.

```
## 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)
```

```
## [1] 977
```

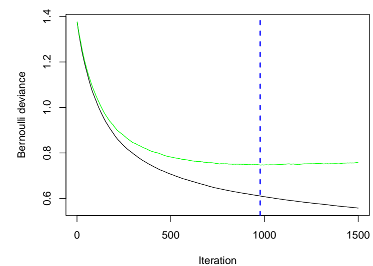


Figure 23: Test error estimated by CV for hearts data using bernoulli loss.

```
summary(heart_gbm)
```

```
##                  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
```

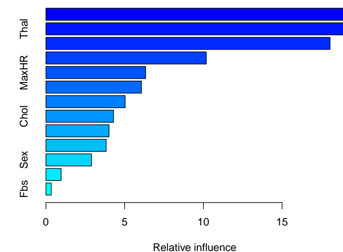


Figure 24: Relative incluence of preecitors in reducing the loss function in hearts data using bernoulli loss.

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.

Stacking

Stacked generalization or *Stacking*[20] involves training a new learning algorithm to combine the predictions of several base learners. First, we train the base learners using training data. Then a combiner, called the *meta model*, is trained to make a final prediction based on the predictions of the base learners. It is shown that such stacked ensembles tend to outperform any of the individual base learners. Although the original idea was due to Wolpert, Breiman[21] formalized stacking – the modern form of stacking that uses internal k-fold CV was Breiman's contribution. The theoretical background for stacking was developed in 2007, and the algorithm took on the name *Super Learner*[22]. The article illustrated that super learners will learn an optimal combination of the base learner predictions and will typically perform as well as or better than any of the individual models that make up the stacked ensemble.

[20] Wolpert, D. (1992). Stacked generalization, Neural Networks 5: 241-259.

[21] Breiman, Leo. 1996b. Stacked Regressions. Machine Learning 24 (1). Springer: 49-64.

[22] Van der Laan, Polley, and Hubbard (2007) Super Learner. Statistical Applications in Genetics and Molecular Biology 6 (1).

Unlike boosting, which combines several weak learners, stacking is designed to ensemble a diverse group of *strong learners*. Here, instead of a sequence of models, a *single* model is used to learn how to best combine the predictions from different base models in consideration. The super learner algorithm consists of three phases:

1. *Setting up the ensemble:*
    - Here we specify a list of L base learners with a specific set of model parameters.[23]
    - We also specify a *meta learning algorithm* – This can be any one of the algorithms discussed before but most often is some form of regularized regression.[24]

[23] Sometimes these are called *level-0 models*.

[24] Sometimes this model called *level-1 model*.

2. *Train the ensemble:*
    - Next we train each of the L base learners on the training set. Usually, we perform cross-validation[25] on each of the base learners and collect the cross-validated (out of fold) predictions from each learner and for each training observation. **The same folds must be used for each base learner**.
    - We form a new $n \times L$ feature matrix by collecting all n cross-validated predicted values from each of the L learners. The original response vector, $\mathbf{Y}$, and the new feature matrix are together called the *level-one* data.
    - Next we train the combiner/meta learning algorithm on the level-one data. Thus the "ensemble model" consists of the L base learning models and the meta learning model, which we can used to generate predictions on new data.

[25] Bootstrap based algorithms are also available

3. *Prediction for new data:*

   - We generate ensemble predictions by first generating predictions from the base learners, and then use those predictions as inputs for the meta learner to generate the ensemble prediction.

Stacking never does worse than selecting the single best base learner on the training data, but not necessarily the validation or test data. The biggest gains are usually produced when stacking base learners that have high variability, and uncorrelated, predicted values. The more similar the predicted values are between the base learners, the less advantage there is to combining them.

We demonstrate stacking using `h2o` package. However, there are many other available packages such as `SuperLearner`, `subsemble`, and `caretEnsemble`.

The first approach to stacking is to train individual base learner models separately and then stack them together. For example, let us look at the `Hitters` data.

```
Hitters <- na.omit(ISLR2::Hitters)
Hitters$Salary <- log(Hitters$Salary)
```

Some initial steps are needed to start `h2o`. We will also split the data to create a holdout set for testing purposes. This is step is done purely to estimate the test errors.

```
library(h2o)

# Create train/test sets
set.seed(1001)
split <- rsample::initial_split(Hitters, strata = "Salary", prop = 0.8)
Hitters_train <- rsample::training(split)
Hitters_test <- rsample::testing(split)

# Get response and feature names
Y <- "Salary"
X <- setdiff(names(Hitters_train), Y)
```

```
# Init h2o
h2o.init()
# Convert train/test sets to h2o format
Hitters_train <- as.h2o(Hitters_train)
Hitters_test <- as.h2o(Hitters_test)
```

Now we want to stack three base models: linear regression with lasso penalty, random forest, and regression gradient boosting. We

first run the base learners on the data using 5-fold CV. Recall, we must use the **same** folds for all the three models. In the code chunk below, `alpha = 1` indicates lasso penalty, `nfolds = 5` for 5-fold CV, `fold_assignment = "Modulo"` is to ensure that all models must use the same fold assignment, and `keep_cross_validation_predictions = TRUE` ensures that the cross-validated predictions from all of the models must be preserved.

```
nfolds <- 5
seed <- 123
# Lasso
hit_glm <- h2o.glm(
  x = X, y = Y, training_frame = Hitters_train,
  nfolds = nfolds, seed = seed,
  keep_cross_validation_predictions = TRUE,
  fold_assignment = "Modulo",
  alpha = 1, remove_collinear_columns = TRUE
  )
# Random forest
hit_rf <- h2o.randomForest(
  x = X, y = Y, training_frame = Hitters_train,
  nfolds = nfolds, seed = seed,
  keep_cross_validation_predictions = TRUE,
  fold_assignment = "Modulo",
  ntrees = 1000, mtries = 5, max_depth = 30,
  sample_rate = 0.8, stopping_rounds = 50,
  stopping_metric = "RMSE", stopping_tolerance = 0
  )
# GBM
hit_gbm <- h2o.gbm(
  x = X, y = Y, training_frame = Hitters_train,
  nfolds = nfolds, seed = seed,
  keep_cross_validation_predictions = TRUE,
  fold_assignment = "Modulo",
  ntrees = 5000, learn_rate = 0.01,
  max_depth = 7, min_rows = 5, sample_rate = 0.8,
  stopping_rounds = 50, stopping_metric = "RMSE",
  stopping_tolerance = 0
  )
```

The next step is to use `h2o.stackedEnsemble()` to stack the models above. There are several choice of meta models available, such as, GLM with non-negative weights, deep learning, random forest, standard GLM and so on. We will use GLM with non-negative weights (in this case a linear regression with non-negative coefficients) by

setting `metalearner_algorithm = "AUTO"`.

```
hit_ensemble <- h2o.stackedEnsemble(
  x = X, y = Y, training_frame = Hitters_train,
  model_id = "hitters_ensemble",
  base_models = list(hit_glm, hit_rf, hit_gbm),
  metalearner_algorithm = "AUTO"
)
```

Let us examine the test RMSE of the base models and the stacked model. We can use `h2o.performance()` function. Note that the output is not a standard list or data frame, and we require the `@` operator to extract its components.

```
# Function to extract rmse
get_rmse <- function(model){
  perf <- h2o.performance(model,
                          newdata = Hitters_test)
  return(perf@metrics$RMSE)
}
# model list
mod_list <- list(GLM = hit_glm, RF = hit_rf,
                 GBM = hit_gbm, STACK = hit_ensemble)
# apply function to model list
purrr::map_dbl(mod_list, get_rmse)
```

```
##       GLM        RF       GBM     STACK
## 0.4906867 0.2785704 0.2700380 0.2815641
```

It seems in this example, stacking did not provide any gain over the base learners random forest and gradient boosting on the test set. By comparing the predictions of the three model (see Figure 25), we see that they are quite correlated – random forest and gradient boosting predictions have a correlation of 96.4% and which GLM has correlation about 74% with the other two methods. Consequently, stacking provides less advantage in this situation since the base learners have highly correlated predictions. The biggest gains are usually produced when we are stacking base learners that have high variability, and uncorrelated, predicted values.

```
# Compare predictions
pred <- data.frame(
  GLM_pred = as.vector(h2o.getFrame(hit_glm@model$cross_validation_holdout_predictions_frame_id$name)),
  RF_pred = as.vector(h2o.getFrame(hit_rf@model$cross_validation_holdout_predictions_frame_id$name)),
  GBM_pred = as.vector(h2o.getFrame(hit_gbm@model$cross_validation_holdout_predictions_frame_id$name))
```

```
)
cor(pred)
```

```
##          GLM_pred   RF_pred  GBM_pred
## GLM_pred 1.0000000 0.7479203 0.7381802
## RF_pred  0.7479203 1.0000000 0.9622414
## GBM_pred 0.7381802 0.9622414 1.0000000
```

```
plot(pred)
```

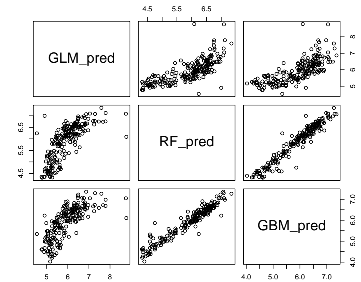


Figure 25: Comparison of predictions of the three base models.

An alternative ensemble approach focuses on stacking multiple models generated from the same base learner. Often we simply select the best performing tuning parameters in a grid search when we build a model. We can also apply the concept of stacking to this process. The process is similar to that shown above. As discussed before, stacking the grid search or even the best models in the grid search can provide significant performance gains when we see high variability across hyperparameter settings.

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. However, BART uses Bayesian methodology and builds upon earlier research on Bayesian methods for CART[26]. 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.

[26] Chipman HA, George EI, McCulloch RE (1998). Bayesian CART Model Search. Journal of the American Statistical Association, 93(443), 935-948.

Let us start with a continuous outcome and regression problem. To start with, let us introduce some notations:

$$
\begin{aligned}
K &= \text{number of regression trees,} \\
B &= \text{number of iterations for which the BART algorithm will be run,} \\
\widehat{f}_k^b(\boldsymbol{x}) &= \text{prediction at } \boldsymbol{x} \text{ for the } k\text{-th regression tree used in the } b\text{-th iteration,} \\
\widehat{f}^b(\boldsymbol{x}) &= \sum_{k=1}^{K} f_k^b(\boldsymbol{x}), \text{prediction of the } b\text{-th iteration.}
\end{aligned}
$$

In the first iteration of the BART algorithm, all trees are initialized to have a single root node, with $\widehat{f}_k^1(\boldsymbol{x}) = \sum_{i=1}^n Y_i/(nK)$, and consequently $\widehat{f}^b(\boldsymbol{x}) = \sum_{i=1}^n Y_i/n$. In subsequent iterations, BART updates each of the K trees, one at a time. In the b-th iteration, to update the k-th tree, we subtract from each response value the predictions from

all but the k-th tree, in order to obtain a partial residual for each observation,

$$r_i = Y_i - \sum_{j<k} \widehat{f}_j^b(\mathbf{X}_i) - \sum_{j>k} \widehat{f}_j^{b-1}(\mathbf{X}_i).$$

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 ones that improve the fit to the partial residual. The following figure illustrates examples of possible perturbations to a tree.

(a) Original (b) Same shape, different prediction (c) Pruning (d) Adding branches

The left panel (panel (a)) in the figure is the k-th tree in iteration $b-1$. The remaining panels are possible perturbations of this tree that can be chosen in iteration b as the k-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 (panel (b)). Another possibility is that the new tree is obtained from pruning the previous tree (panel (c)). Yet another option is that the new tree may have more terminal nodes than the old tree (panel (d).

After B iterations, we have a collection of prediction models, $\widehat{f}^b(\boldsymbol{x})$. 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 L iterations as burn-in, out final prediction for input $\boldsymbol{x}$ would be

$$\widehat{f}(\boldsymbol{x}) = \frac{1}{B-L} \sum_{b=L+1}^{B} \widehat{f}^b(\boldsymbol{x}).$$

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.[27] There are other packages such as `bartMachine`, which is also supported by `caret`.

[27] See `?wbart` for details.

```
library(BART)
set.seed(1001)
hit_bart <- wbart(x.train = Hitters[,-19], y.train = Hitters$Salary,
                  ntree = 200, ndpost = 1000, nskip = 200)
```

```
## *****Into main of wbart
## *****Data:
## data:n,p,np: 263, 22, 0
## y1,yn: 0.236093, 0.980534
## x1,x[n*p]: 315.000000, 0.000000
## *****Number of Trees: 200
## *****Number of Cut Points: 100 ... 1
## *****burn and ndpost: 200, 1000
## *****Prior:beta,alpha,tau,nu,lambda: 2.000000,0.950000,0.063565,3.000000,0.073304
## *****sigma: 0.613451
## *****w (weights): 1.000000 ... 1.000000
## *****Dirichlet:sparse,theta,omega,a,b,rho,augment: 0,0,1,0.5,1,22,0
## *****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: 4s
## check counts
```

```
## trcnt,tecnt,temecnt,treedrawscnt: 1000,0,0,1000
```

```
# Variable importance
sort(hit_bart$varcount.mean)
```

```
##      CRuns     CWalks     CHmRun     CAtBat       Hits      CHits      AtBat
##      6.524      7.744      7.762      8.839      9.117      9.222      9.285
##     Errors       CRBI       Runs    PutOuts    League2    Assists      Walks
##      9.810     10.037     10.070     10.324     10.634     10.773     10.893
##        RBI NewLeague1      HmRun  Division1 NewLeague2    League1  Division2
##     11.010     11.074     11.099     11.244     11.299     11.356     11.615
##      Years
##     14.593
```

We can also use `predict()` function to generate predictions for new data.

Summary and Discussion

In this section, we discussed regression and classification trees using the CART approach. Also, we discussed about 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 stacking, we again use only the original data, and do not draw any random samples. Unlike boosting, we combine several strong learners (base models) using a meta algorithm (meta model) by using the predictions from the base models as input to the meta model.

- 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.

There are other methods we have not covered in this chapter, but are also important. We briefly mention them in the following sections.

Regression Model Trees

One limitation of simple regression trees is that each terminal node uses the average of the training set outcomes in that node for prediction. Thus, these models may produce inaccurate predictions for samples whose true outcomes are extremely high or low. Quinlan[28] describes the *model tree* approach, called M5, which is similar to regression trees with the follows changes:

[28] Quinlan R (1992). Learning with Continuous Classes. Proceedings of the 5th Australian Joint Conference On Artificial Intelligence, pp. 343-348.

- The splitting criterion is different.
- The terminal nodes predict the outcome using a linear model as opposed to the simple average.
- When prediction of performed for a sample, it is often a combination of the predictions from different models along the same path through the tree.

Model trees also incorporate a type of *smoothing* to decrease the potential for over-fitting. The technique is based on the *recursive shrinking* methodology of Hastie and Pregibon (1990)[29]. When predicting, the new sample travels down the appropriate path of the tree, and moving from the bottom up, the linear models along that path are combined using a specific mathematical formula. This type of smoothing can have a significant positive effect on the model tree when the linear models across nodes are very different, which can happen due to small training set or multicollinearity.

[29] Hastie T, Pregibon D (1990). Shrinking Trees. Technical report, AT&T Bell Laboratories Technical Report.

Rule based approach

A rule is defined as a distinct path through a tree. For the tree, a new sample can only travel down a single path through the tree defined by these rules. The number of samples affected by a rule is called its *coverage*. In addition to the pruning algorithms described in the last section, the complexity of the model tree can be further reduced by either removing entire rules or removing some of the conditions that define the rule.

Consider the rule in Figure 9 that represents the second leaf node from left.

cAtBat < 1452 & CHits < 182 & cHits < 26

In the rule above, note that the `CHits` variable is used twice. This occurred because another path through the tree determined that modeling the data subset where `CHits` is between 26 and 182 was important. However, when viewed in isolation, the rule above is unnecessarily complex because of this redundancy. In general, it may be advantageous to remove some conditions in a rule because they do not contribute much to the model.

Quinlan[30] describes methodologies for simplifying the rules generated from classification trees – this algorithm is for tree is called C4.5, and the rule based method is called C4.5Rules. Similar techniques can be applied to model trees to create a more simplistic set of rules from an initial model tree. We also have C5.0, which is a advanced version of C4.5. C5.0 also has a rule based variant.

[30] Quinlan R (1993). C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers.

The model tree and rule based approaches are available in the `RWeka` package: `M5P()` and `M5Rules()` function for model trees and rule based models, and `J48()` function for C4.5 classifier. The C5.0 algorithm is available in `C50` package. Other RWeka functions for rules can be found on the help page `?Weka_classifier_rules`.

```
library(RWeka)
hit_m5 <- M5P(Salary ~ ., data = Hitters,
              control = Weka_control(M=10))
hit_m5
```

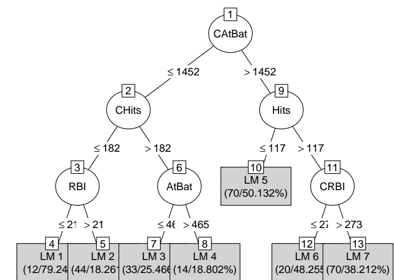
```
## M5 pruned model tree:
## (using smoothed linear models)
##
## CAtBat <= 1452 :
## |   CHits <= 182 :
## |   |   RBI <= 21.5 : LM1 (12/79.24%)
## |   |   RBI >  21.5 : LM2 (44/18.261%)
## |   CHits >  182 :
## |   |   AtBat <= 465 : LM3 (33/25.466%)
## |   |   AtBat >  465 : LM4 (14/18.802%)
## CAtBat >  1452 :
## |   Hits <= 117.5 : LM5 (70/50.132%)
## |   Hits >  117.5 :
## |   |   CRBI <= 273 : LM6 (20/48.255%)
## |   |   CRBI >  273 : LM7 (70/38.212%)
##
```

```
## LM num: 1
## Salary =
##  -0.0009 * AtBat
##  + 0.0017 * Hits
##  + 0.0014 * Runs
##  - 0.0047 * RBI
##  - 0.0267 * Walks
##  + 0.0067 * Years
##  + 0.0001 * CAtBat
##  - 0.0021 * CHits
##  + 0.0003 * CHmRun
##  + 0.007 * CRBI
##  - 0.0001 * CWalks
##  + 0.0003 * PutOuts
##  + 5.2312
##
## LM num: 2
## Salary =
##  -0.0009 * AtBat
##  + 0.0017 * Hits
##  + 0.0014 * Runs
##  - 0.0021 * RBI
##  + 0.0027 * Walks
##  + 0.0067 * Years
##  + 0.0001 * CAtBat
##  + 0.0005 * CHits
##  + 0.0003 * CHmRun
##  + 0.0059 * CRBI
##  - 0.0001 * CWalks
##  + 0.0002 * PutOuts
##  + 4.3843
##
## LM num: 3
## Salary =
##  -0.0016 * AtBat
##  + 0.0027 * Hits
##  + 0.0027 * Runs
##  + 0.0013 * Walks
##  + 0.0067 * Years
##  + 0.0003 * CAtBat
##  + 0.0001 * CHits
##  + 0.0003 * CHmRun
##  + 0.0029 * CRBI
##  - 0.0001 * CWalks
```

```
## + 0.0001 * PutOuts
## + 4.9154
##
## LM num: 4
## Salary =
## -0.002 * AtBat
## + 0.0051 * Hits
## + 0.0034 * Runs
## + 0.0013 * Walks
## + 0.0067 * Years
## + 0.0004 * CAtBat
## + 0.0001 * CHits
## + 0.0003 * CHmRun
## + 0.0005 * CRBI
## - 0.0001 * CWalks
## + 0.0001 * PutOuts
## + 4.767
##
## LM num: 5
## Salary =
## -0.0003 * AtBat
## + 0.0014 * Hits
## + 0.0078 * Walks
## - 0.0566 * Years
## + 0 * CAtBat
## + 0.0004 * CHits
## - 0.0001 * CHmRun
## + 0.0001 * CRBI
## - 0.0002 * CWalks
## + 0 * PutOuts
## - 0.0001 * Assists
## + 5.9318
##
## LM num: 6
## Salary =
## -0.0003 * AtBat
## + 0.0014 * Hits
## + 0.0036 * Walks
## - 0.0242 * Years
## + 0 * CAtBat
## + 0 * CHits
## - 0.0005 * CHmRun
## + 0.0007 * CRBI
## - 0.0001 * CWalks
```

```
##  + 0 * PutOuts
##  - 0.0007 * Assists
##  + 6.1865
##
## LM num: 7
## Salary =
##  -0.0003 * AtBat
##  + 0.0035 * Hits
##  - 0.0061 * HmRun
##  + 0.0057 * Walks
##  - 0.0638 * Years
##  + 0.0001 * CAtBat
##  + 0 * CHits
##  + 0.0014 * CHmRun
##  + 0.0003 * CRBI
##  - 0.0001 * CWalks
##  + 0 * PutOuts
##  - 0.0001 * Assists
##  + 6.1356
##
## Number of Rules : 7
```

```
plot(hit_m5)
```




```
hit_m5rules <- M5Rules(Salary ~ ., data = Hitters,
              control = Weka_control(M=10))
hit_m5rules
```

```
## M5 pruned model rules
## (using smoothed linear models) :
## Number of Rules : 8
##
## Rule: 1
## IF
##  CAtBat > 1452
##  Hits <= 117.5
## THEN
##
## Salary =
##  -0.0003 * AtBat
##  + 0.0014 * Hits
##  + 0.0078 * Walks
##  - 0.0566 * Years
##  + 0 * CAtBat
```

```
##  + 0.0004 * CHits
##  - 0.0001 * CHmRun
##  + 0.0001 * CRBI
##  - 0.0002 * CWalks
##  + 0 * PutOuts
##  - 0.0001 * Assists
##  + 5.9318 [70/50.132%]
##
## Rule: 2
## IF
##  CRuns > 213.5
##  CRBI > 273
## THEN
##
## Salary =
##  -0.0006 * AtBat
##  + 0.0046 * Hits
##  - 0.005 * HmRun
##  - 0.0009 * RBI
##  + 0.0049 * Walks
##  - 0.0549 * Years
##  + 0.0001 * CAtBat
##  - 0.0002 * CHits
##  + 0.0014 * CHmRun
##  + 0.0003 * CRuns
##  + 0.0004 * CRBI
##  - 0.0002 * CWalks
##  + 0.0001 * PutOuts
##  + 6.1099 [70/34.674%]
##
## Rule: 3
## IF
##  CAtBat <= 842.5
##  AtBat > 206.5
## THEN
##
## Salary =
##  -0.0006 * AtBat
##  + 0.0101 * Years
##  - 0.0003 * CAtBat
##  + 0.0088 * CRuns
##  + 0.0008 * CRBI
##  + 0.0014 * CWalks
##  + 0.0002 * PutOuts
```

```
##  + 0.0035 * Errors
##  + 4.3354 [51/25.436%]
##
## Rule: 4
## IF
##  CAtBat > 1283.5
## THEN
##
## Salary =
##  -0.0005 * AtBat
##  + 0.0023 * RBI
##  + 0.0015 * CRuns
##  + 5.7599 [32/71.137%]
##
## Rule: 5
## IF
##  CRuns > 61.5
##  AtBat <= 406
##  HmRun <= 8.5
## THEN
##
## Salary =
##  -0.0006 * AtBat
##  + 0.0033 * HmRun
##  + 0.0004 * CAtBat
##  + 5.1954 [15/33.291%]
##
## Rule: 6
## IF
##  AtBat <= 406
##  HmRun <= 7.5
## THEN
##
## Salary =
##  -0.006 * AtBat
##  + 0.0389 * HmRun
##  + 5.9503 [10/34.556%]
##
## Rule: 7
## IF
##  AtBat > 406
## THEN
##
## Salary =
```

```
##  -0.0008 * AtBat
##  + 5.6685 [9/40.874%]
##
## Rule: 8
##
## Salary =
##  + 5.7277 [6/100%]
```