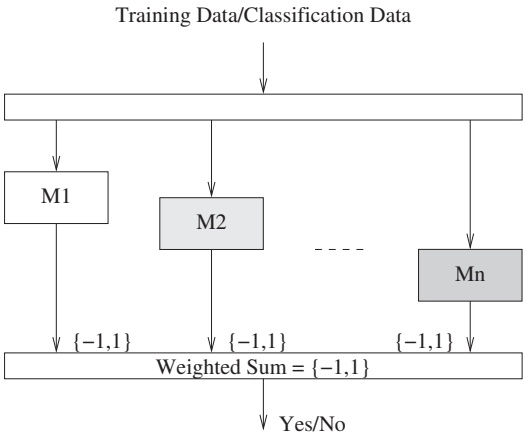
# Chapter 13

# Boosting

The Boosting meta-algorithm is an efficient, simple, and easy-to-use approach to building models. The popular variant called AdaBoost (an abbreviation for adaptive boosting) has been described as the "best off-the-shelf classifier in the world" (attributed to Leo Breiman by Hastie et al. (2001, p. 302)).




Boosting algorithms build multiple models from a dataset by using some other learning algorithm that need not be a particularly good learner. Boosting associates weights with observations in the dataset and increases (boosts) the weights for those observations that are hard to model accurately. A sequence of models is constructed, and after each model is constructed the weights are modified to give more weight to those observations that are harder to classify. In fact, the weights of such observations generally oscillate up and down from one model to the next. The final model is then an additive model constructed from the sequence of models, each model's output being weighted by some score. There is little tuning required and little is assumed about the learner used, except that it should be a weak learner! We note that boosting can fail to perform if there is insufficient data or if the weak models are overly complex. Boosting is also susceptible to noise.

Boosting algorithms are therefore similar to random forests in that an ensemble of models is built and then combined to deliver a better model

G. Williams, *Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery*, Use R, DOI 10.1007/978-1-4419-9890-3_13, © Springer Science+Business Media, LLC 2011

than any of the constituent models. The basic distinguishing characteristic of the boosting approach is that the trees are built one after another, with refinement being based on the previously built models. The concept is that after building one model any observations that are incorrectly classified by that model are boosted. A boosted observation is essentially given more prominence in the dataset, making the single observation overrepresented. This has the effect that the next model is more likely to correctly classify that observation. If not, then that observation will again be boosted.

In common with random forests, the boosting algorithms tend to be meta-algorithms. Any type of modelling approach might be used as the learning algorithm, but the decision tree algorithm is the usual approach.

## 13.1 Knowledge Representation

The boosting algorithm is commonly presented in terms of decision trees as their primary form for the representation of knowledge. The key extension to the knowledge representation is in the way that we combine the decisions that are made by the individual "experts" or models. For boosting, a weighted score is used, with each of the models in the ensemble having a weight corresponding to the quality of its expertise (e.g., the measured accuracy of the individual tree).

## 13.2 Algorithm

As a meta-learner, boosting employs any simple learning algorithm to build multiple models. Boosting often relies on the use of a weak learning algorithm—essentially any weak learner can be used. An ensemble of weak learners can lead to a strong model.

A weak learning algorithm is one that is only slightly better than random guessing in terms of error rates (i.e., the model gets the decision wrong just less than 50% of the time). An early example was a decision tree of depth 1 (having a single split point and thus often referred to as a *decision stump*). Each weak model is slightly better than random but as an ensemble delivers considerable accuracy.

The algorithm begins quite simply by building a "weak" initial model from the training dataset. Then, any observations in the training data that the model incorrectly classifies will have their importance within the

algorithm boosted. This is done by assigning all observations a weight—all observations might start, for example, with a weight of 1. Weights are boosted through a formula so that those that are wrongly classified by the model will have a weight greater than 1 for the building of the next decision stump.

A new model is built with these boosted observations. We can think of them as the problematic observations. The algorithm needs to take into account the weights of the observations in building a model. Consequently, the model builder effectively tries harder each iteration to correctly classify these "difficult" observations.

The process of building a model and then boosting observations incorrectly classified is repeated until a newly generated model performs no better than random. The result is then an ensemble of models to be used to make a decision on new data. The decision is arrived at by combining the "expertises" of each model in such a way that the more accurate models carry more weight.

We can illustrate the process abstractly with a simple example. Suppose we have ten observations. Each observation will get an initial weight of, let's say, $\frac{1}{10}$, or 0.1. We build a decision tree that incorrectly classifies four observations (e.g., observations 7, 8, 9, and 10). We can calculate the sum of the weights of the misclassified observations as 0.4 (and generally we denote this as $\epsilon$). This is a measure of the accuracy (actually the inaccuracy) of the model.

The $\epsilon$ is transformed into a measure used to update the weights and to provide a weight for the model when it forms part of the ensemble. This transformed value is $\alpha$ and is often something like $0.5 * \log(\frac{1-\epsilon}{\epsilon})$. The new weights for the misclassified observations can then be recalculated as $e^{\alpha}$ times the old weight. In our example, $\alpha = 0.2027$ (i.e., $(0.5 * \log(\frac{1-0.4}{0.4}))$ and so the new weights for observations 7, 8, 9, and 10 become $0.1 * e^{\alpha}$, or 0.1225.

The tree builder is called upon again, noting that some observations are effectively multiplied to have more representation in the algorithm. Thus a different tree is likely to be built that is more likely to correctly classify the observations that have higher weights (i.e., have more representation in the training dataset).

This new model will again have errors. Suppose this time that the model incorrectly classifies observations 1 and 8. Their current weights are 0.1 and 0.1225, respectively. Thus, the new $\epsilon$ is $0.1 + 0.1225$, or 0.2225. The new $\alpha$ is then 0.6257. This is the weight that we give to

this model when included in the ensemble. It is again used to modify the weights of the incorrectly classified observations, so that observation 1 gets a weight of $0.1 * e^{\alpha}$, or 0.1869 and observation 8's weight becomes $0.1225 * e^{\alpha}$, or 0.229. So we can see that observation 8 has the highest weight now since it seems to be quite a problematic observation. The process continues until the individual tree that is built has an error rate of greater than 50%.

To deploy the individual models as an ensemble, each tree is used to classify a new observation. Each tree will provide a probability that it will rain tomorrow (a number between 0 and 1). For each tree, this is multiplied by the weight ($\alpha$) associated with that tree. The final result is then calculated as the average of these predictions.

Actual implementations of the boosting algorithm use variations to the simple approach we have presented here. Variations are found in the formulas for updating the weights and for weighting the individual models. However, the overall concept remains the same.

## 13.3 Tutorial Example

### Building a Model Using Rattle

The Boost option of the Model tab will build an ensemble of decision trees using the approach of boosting misclassified observations. The individual decision trees are built using **rpart**. The results of building the model are shown in the Textview area. Using the *weather* dataset (loaded by default in Rattle if we click Execute on starting up Rattle) we will see the Textview populated as in Figure 13.1.

The Textview begins with the usual summary of the underlying function call to build the model:

```
Summary of the Ada Boost model:

Call:
ada(RainTomorrow ~ .,
    data = crs$dataset[crs$train, c(crs$input,
    crs$target)], control = rpart.control(maxdepth = 30,,
    minsplit = 20, xval = 10),
    iter = 50)
```

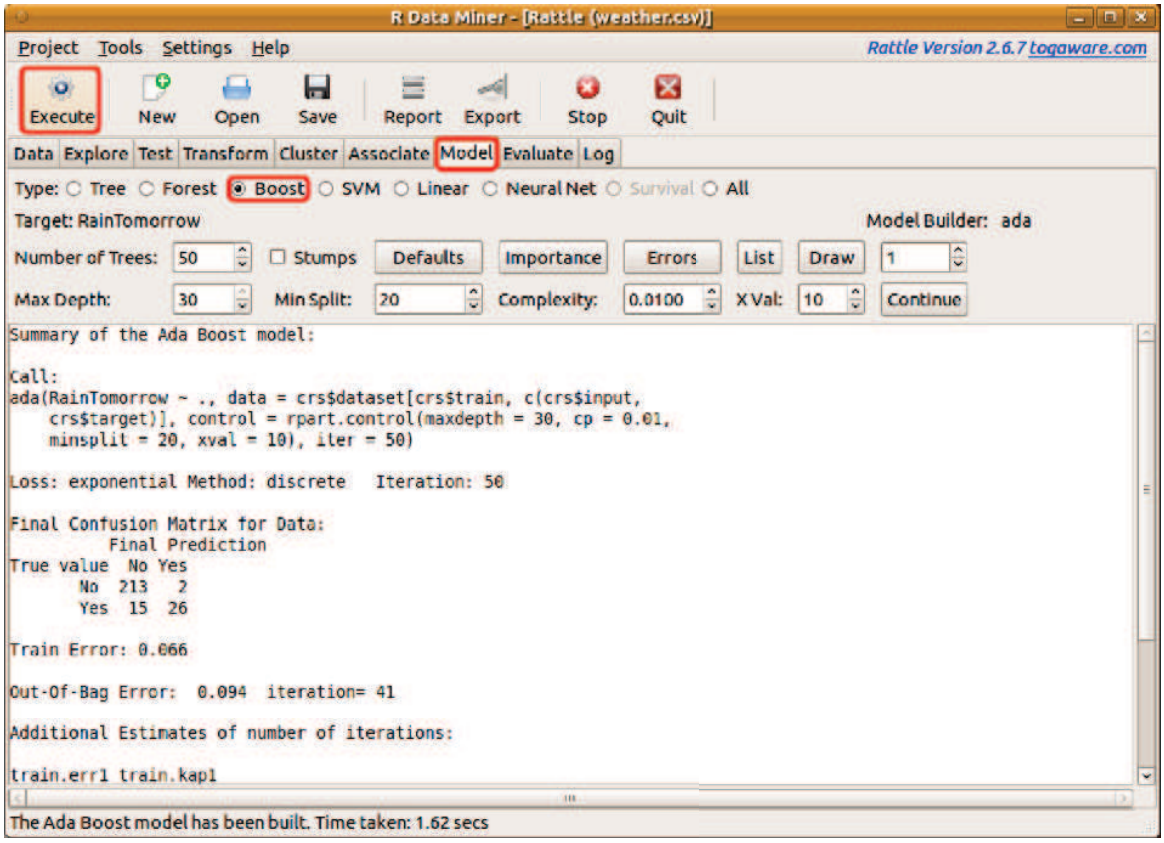



Figure 13.1: Building an AdaBoost predictive model.

The model is to predict `RainTomorrow` based on the remainder of the variables. The data consists of the dataset loaded into Rattle, retaining only the observations whose index is contained in the training list and including all but columns 1, 2, and 23. The `control=` argument is passed directly to `rpart()` and has the same meaning as for `rpart()` (see Chapter 11). The number of trees to build is specified by the `iter=` argument. The next line of information reports on some of the parameters used for building the model.

We won't go into the details of the `Loss` and `Method`. Briefly, though, the `Loss` is `exponential`, indicating that the algorithm is minimising a so called exponential loss function, and the `Method` used in the algorithm is `discrete` rather than `gentle` or `real`. The `Iteration:` item simply indicates the number of trees that were asked to be built.

### Performance Evaluation

A confusion matrix presents the performance of the model over the training data, and the following line in the Textview reports the training

dataset error.

```
Final Confusion Matrix for Data:
          Final Prediction
True value  No Yes
       No  213   2
       Yes  15  26
```

The out-of-bag error and the associated iteration are then reported. This is followed by suggestions of the number of iterations based on the training error and an error measure based on the Kappa statistic. The Kappa statistic adjusts for the situation where there are very different numbers of observations for each value of the target variable. Using these error estimates, the best number of iterations is suggested:

```
Out-Of-Bag Error:  0.094  iteration= 41

Additional Estimates of number of iterations:

train.err1 train.kap1
        47         47
```

The actual training and Kappa (adjusted) error rates are then recorded:

```
train.err train.kap
     0.07      0.28
```

### Time Taken

The `ada()` implementation takes longer than `randomForest()` because it is relying on using the inbuilt `rpart()` rather than specially written Fortran code as is the case for `randomForest()`.

```
Time taken: 1.62 secs
```

### Error Plot

Once a boosted model has been built, the Error button will display a plot of the decreasing error rate as more trees are added to the model. The plot annotates the curve with a series of five 1s simply to identify the curve. (Extended plots can include curves for test datasets.)

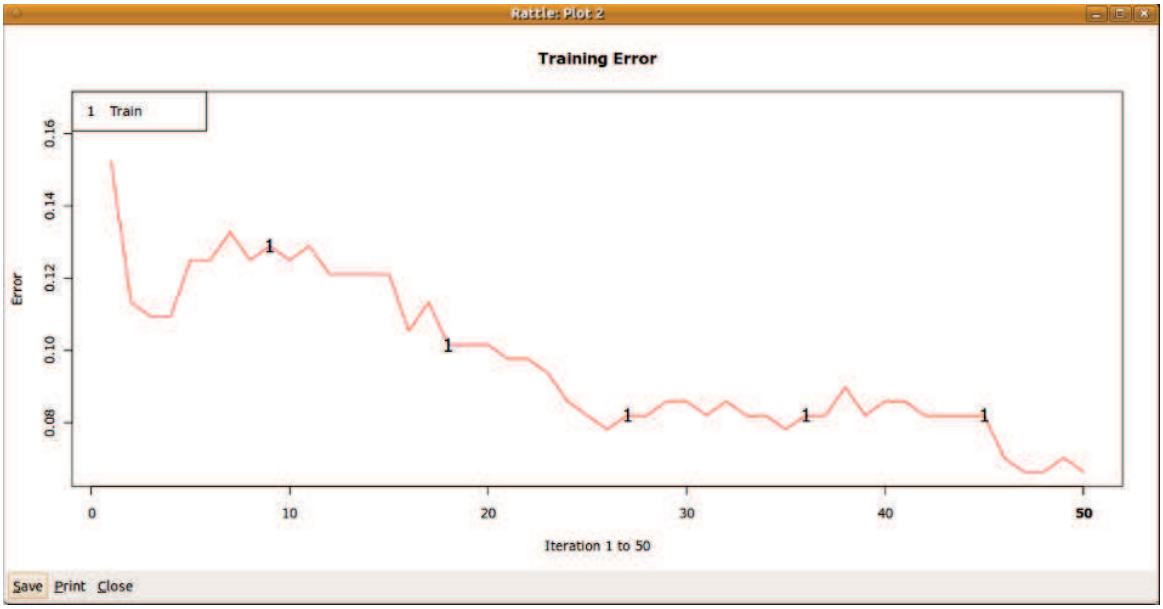



Figure 13.2: The error rate as more trees are added to the ensemble.

Figure 13.2 shows the decreasing error as more trees are added. The plot is typical of ensembles where the error rate drops off quite quickly early on and then flattens out as we proceed. We might decide, from the plot, a point at which we stop building further trees. Perhaps that is around 40 trees for our data.

### Variable Importance

A measure of the importance of variables is also provided by **ada** (Culp et al., 2010). Figure 13.3 shows the plot. The measure is a relative measure so that the order and distance between the scores are more relevant than the actual scores.

The measure calculates, for each tree, the improvement in accuracy that the variable chosen to split the dataset offers the model. This is then averaged over all trees in the ensemble.

Of the five most important variables, we notice that there are two categoric variables (`WindDir9am` and `WindDir3pm`). Because of the nature of how variables are chosen for a decision tree algorithm, there may well be a bias here in favour of categoric variables, so we might discount their importance. See Chapter 12, page 265 for a discussion of the issue.

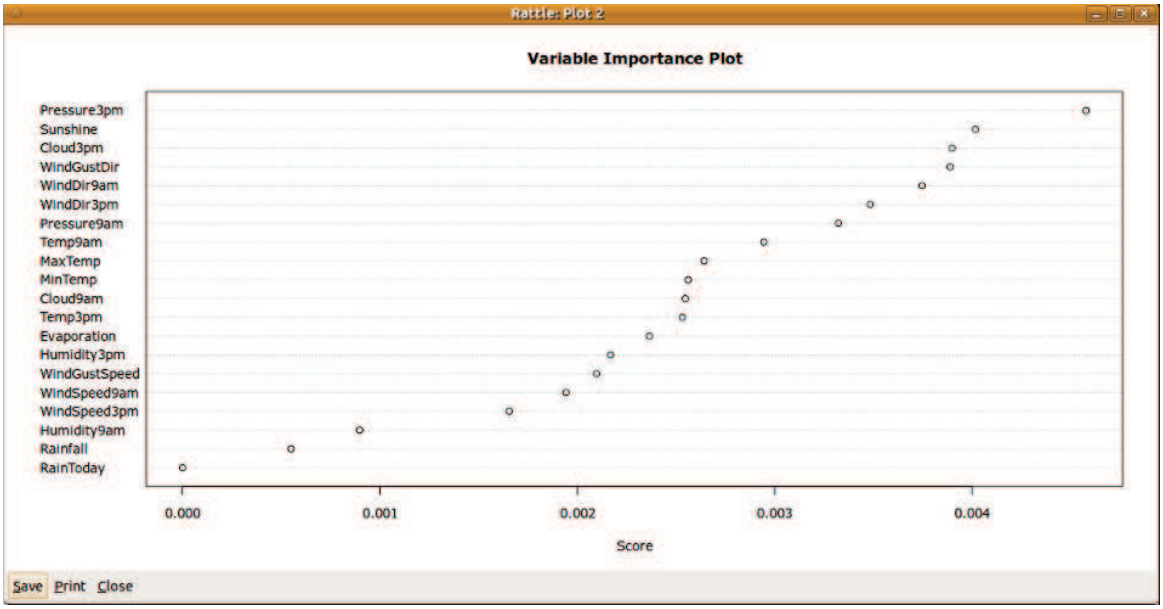



Figure 13.3: The variable importance plot for a boosted model.

### Tuning Options

A few basic tuning options for boosting are provided by the Rattle interface. The first option is the Number of Trees to build, which is set to 50 by default. The Max Depth, Min Split, and Complexity are as provided by the decision tree algorithm and are discussed in Section 13.4.

### Adding Trees

The Continue button allows further trees to be added to the model. This allows us to easily explore whether the addition of further trees will offer much improvement in the performance of the model, without starting the modelling over again.

To add further trees, increase the value specified in the Number of Trees text box and click the Continue button. This will pick up the model building from where it left off and build as many more trees as is needed to get up to the specified number of trees.

### R

The package **ada** provides `ada()`, which implements the boosting algorithm deployed by Rattle. The `ada()` function itself uses `rpart()` from **rpart** to build the decision trees. With the default settings, a very reasonable model can be built.

We will step through the simple process of building a boosted model. First, we create the dataset object, as usual. This will encapsulate the *weather* dataset from **rattle**, together with a collection of other useful data about the *weather* dataset. A training sample is also identified/

```
> library(rattle)
> weatherDS <- new.env()
> evalq({
    data <- weather
    nobs <- nrow(weather)
    vars <- -grep('^(Date|Location|RISK_)', names(data))
    form <- formula(RainTomorrow ~ .)
    target <- all.vars(form)[1]
    set.seed(42)
    train <- sample(nobs, 0.7*nobs)
  }, weatherDS)
```

We can now build the boosted model based on this dataset. Once again we create a container for the model, and include the above container for the dataset within this container.

```
> library(ada)
> weatherADA <- new.env(parent=weatherDS)
```

Within this new container we now build our model.

```
> evalq({
    control <- rpart.control(maxdepth=30,
                             cp=0.010000,
                             minsplit=20,
                             xval=10)
    model <- ada(formula=form,
                 data=data[train, vars],
                 control=control,
                 iter=50)
  }, weatherADA)
```

We can obtain a basic overview of the model simply by printing its value, as we do in the following code block (note that the results here may vary slightly between 32 bit and 64 bit implementations of R).

```
> weatherADA$model

Call:
ada(form, data = data[train, vars],
  control = control, iter = 50)

Loss: exponential Method: discrete   Iteration: 50

Final Confusion Matrix for Data:
          Final Prediction
True value  No Yes
       No  213   2
       Yes  16  25

Train Error: 0.07

Out-Of-Bag Error:  0.105  iteration= 38

Additional Estimates of number of iterations:

train.err1 train.kap1
        41         41
```

The `summary()` command provides a little more detail.

```
> summary(weatherADA$model)

Call:
ada(form, data = data[train, vars],
  control = control, iter = 50)

Loss: exponential Method: discrete   Iteration: 50

Training Results

Accuracy: 0.93 Kappa: 0.697
```

### Replicating AdaBoost Directly using `rpart()`

We can replicate the boosting process directly using `rpart()`. We will illustrate this as an example of a little more sophistication in R coding.

We will first load the *weather* dataset and extract the input variables ($x$) and the output variable ($y$). To simplify some of the mathematics we will map the predictions to $-1/1$ rather than $0/1$ (since then a model that predicts a value greater than 0 is a positive example and one below zero is a negative example). The data is encapsulated within a container called `weatherBRP`.

```
> library(rpart)
> weatherBRP <- new.env()
> evalq({
    data <- weather
    vars <- -grep('^(Date|Location|RISK_)', names(data))
    target <- "RainTomorrow"
    N <- nrow(data)
    M <- ncol(data) - length(vars)
    data$Target <- rep(-1, N)
    data$Target[data[target] == "Yes"] <- 1
    vars <- c(vars, -(ncol(data)-1)) # Remove old target
    form <- formula(Target ~ .)
    target <- all.vars(form)[1]
  }, weatherBRP)
```

The first few observations show the mapping from the original target, which has the values `No` and `Yes`, to the new numeric values $-1$ and 1.

```
> head(weatherBRP$data[c("RainTomorrow", "Target")])

  RainTomorrow Target
1          Yes      1
2          Yes      1
3          Yes      1
4          Yes      1
5           No     -1
6           No     -1
```

We can check the list of variables available (only checking the first few here), and note that we exclude four from our analysis:

```
> head(names(weatherBRP$data))

[1] "Date"        "Location"    "MinTemp"
[4] "MaxTemp"     "Rainfall"    "Evaporation"

> weatherBRP$vars

[1]  -1  -2 -23 -24
```

Now we can initialise the observation weights, which to start with are all the same $(1/N)$:

```
> evalq({
    w <- rep(1/N, N)
  }, weatherBRP)
> round(head(weatherBRP$w), 4)

[1] 0.0027 0.0027 0.0027 0.0027 0.0027 0.0027
```

Next we build the first model. The `rpart()` function, conveniently, has a weights argument, and we simply pass to it the calculated weights store in `w`. We also set up `rpart.control()` for building a decision tree stump. The control simply includes `maxdepth=`, setting it to 1 so that a single-level tree is built:

```
> evalq({
    control <- rpart.control(maxdepth=1)
    M1 <- rpart(formula=form,
                data=data[vars],
                weights=w/mean(w),
                control=control,
                method="class")
  }, weatherBRP)
```

We can then display the first model:

```
> weatherBRP$M1

n= 366

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 366 66 -1 (0.8197 0.1803)
  2) Humidity3pm< 71.5 339 46 -1 (0.8643 0.1357) *
  3) Humidity3pm>=71.5 27  7 1 (0.2593 0.7407) *
```

We see that the decision tree algorithm has chosen `Humidity3pm` on which to split the data, at a split point of 71.5. For $Humidity3pm < 71.5$ the decision is –1 with probability 0.86, and for $Humidity3pm \geq 71.5$ the decision is 1 with probability 0.75.

We now need to find those observations that are incorrectly classified by the model. The R code here calls `predict()` to apply the model `M1` to the dataset from which it was built. From this result, we get the second column. This is the list of probabilities of each observation being in class 1. If this probability is above 0.5, then the result is 1, otherwise it is $-1$ (multiplying the logical value by 2 and then subtracting 1 achieves this since TRUE is regarded as 1 and FALSE as 0). The resulting class is then compared with the target, and `which()` returns the index of those observations for which the prediction differs from the actual class:

```
> evalq({
    ms <- which(((predict(M1)[,2]>0.5)*2)-1 !=
                data[target])
    names(ms) <- NULL
  }, weatherBRP)
```

The indexes of the first few of the 53 misclassified can be listed:

```
> evalq({
    cat(paste(length(ms),
              "observations incorrectly classified:\n"))
    head(ms)
  }, weatherBRP)

53 observations incorrectly classified:
[1]  1  2  3  4  9 17
```

We now calculate the model weight (based on the weighted error rate of this deicison tree) dividing by the total sum of weights to get a normalised value (so that sum(w) remains 1):

```
> evalq({e1 <- sum(w[ms])/sum(w); e1}, weatherBRP)

[1] 0.1448
```

The adjustment is then calculated:

```
> evalq({a1 <- log((1-e1)/e1); a1}, weatherBRP)

[1] 1.776
```

We then update the observation weights:

```
> round(head(weatherBRP$w[weatherBRP$ms]), 4)

[1] 0.0027 0.0027 0.0027 0.0027 0.0027 0.0027

> evalq({w[ms] <- w[ms]*exp(a1)}, weatherBRP)
> round(head(weatherBRP$w[weatherBRP$ms]), 4)

[1] 0.0161 0.0161 0.0161 0.0161 0.0161 0.0161
```

A second model can now be built:

```
> evalq({
    M2 <- rpart(formula=form,
                data=data[vars],
                weights=w/mean(w),
                control=control,
                method="class")
  }, weatherBRP)
```

This results in a simple decision tree involving the variable `Pressure3pm`:

```
> weatherBRP$M2

n= 366

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 366 170.50 -1 (0.5341 0.4659)
  2) Pressure3pm>=1016 206  29.96 -1 (0.8065 0.1935) *
  3) Pressure3pm< 1016 160  70.58 1 (0.3343 0.6657) *
```

Once again we identify the misclassified observations

```
> evalq({
    ms <- which(((predict(M2)[,2]>0.5)*2)-1 !=
                data[target])
    names(ms) <- NULL
  }, weatherBRP)
```

There are 118 of them:

```
> evalq({length(ms)}, weatherBRP)

[1] 118
```

The indexes of the first few can also be listed:

```
> evalq({head(ms)}, weatherBRP)

[1]  9 14 15 16 18 19
```

We again boost the misclassified observations, first calculating the weighted error rate of the decision tree:

```
> evalq({e2 <- sum(w[ms])/sum(w); e2}, weatherBRP)

[1] 0.2747
```

The adjustment is calculated:

```
> evalq({a2 <- log((1-e2)/e2); a2}, weatherBRP)

[1] 0.9709
```

The adjustments are then made to the weights of the individual observations that were misclassified:

```
> round(head(weatherBRP$w[weatherBRP$ms]), 4)

[1] 0.0161 0.0027 0.0027 0.0027 0.0027 0.0027

> evalq({w[ms] <- w[ms]*exp(a2)}, weatherBRP)
> round(head(weatherBRP$w[weatherBRP$ms]), 4)

[1] 0.0426 0.0072 0.0072 0.0072 0.0072 0.0072
```

A third (and for our purposes the last) model can then be built:

```
> evalq({
    M3 <- rpart(formula=form,
                data=data[vars],
                weights=w/mean(w),
                control=control,
                method="class")
    ms <- which(((predict(M3)[,2]>0.5)*2)-1 !=
                data[target])
    names(ms) <- NULL
  }, weatherBRP)
```

Again we identify the misclassified observations:

```
> evalq({length(ms)}, weatherBRP)

[1] 145
```

Calculate the error rate:

```
> evalq({e3 <- sum(w[ms])/sum(w); e3}, weatherBRP)

[1] 0.3341
```

Calculate the adjustment:

```
> evalq({a3 <- log((1-e3)/e3); a3}, weatherBRP)

[1] 0.6896
```

We can then finally adjust the weights (in case we decide to continue building further decision trees):

```
> round(head(weatherBRP$w[weatherBRP$ms]), 4)
[1] 0.0161 0.0161 0.0027 0.0027 0.0027 0.0027
> evalq({w[ms] <- w[ms]*exp(a3)}, weatherBRP)
> round(head(weatherBRP$w[weatherBRP$ms]), 4)
[1] 0.0322 0.0322 0.0054 0.0054 0.0054 0.0054
```

The final (combined or ensemble) model, if we choose to stop here, is then

$$\mathcal{M}(x) = 1.7759 * \mathcal{M}_1(x) + 0.9709 * \mathcal{M}_2(x) + 0.6896 * \mathcal{M}_3(x).$$

## 13.4 Tuning Parameters

A number of options are given by Rattle for boosting a decision tree model. We briefly review them here.

**Number of Trees** `iter=50`

The number of trees to build is specified by the `iter=` argument. The default is to build 50 trees.

**Bagging** `bag.frac=0.5`

Bagging is used to randomly sample the supplied dataset. The default is to select a random sample from the population of 50%.

## 13.5 Discussion

### References

Boosting originated with Freund and Schapire (1995). Building a collection of models into an ensemble can reduce misclassification error, bias, and variance (Bauer and Kohavi, 1999; Schapire et al., 1997). The original formulation of the algorithm adjusts all weights each iteration—weights are increased if the corresponding record is misclassified or decreased if it is correctly classified. The weights are then further normalised each iteration to ensure they continue to represent a distribution

(so that $\sum_{j=1}^{n} w_j = 1$). This can be simplified, as by Hastie et al. (2001), to increase only the weights of the misclassified observations.

A number of R packages implement boosting. We have covered **ada** here, and this is the package presently used by Rattle. **caTools** (Tuszynski, 2009) provides `LogitBoost()`, which is simple to use and an efficient implementation for very large datasets, using a carefully crafted implementation of decision stumps as the weak learners. **gbm** (Ridgeway, 2010) implements generalised boosted regression, providing a more widely applicable boosting algorithm. **mboost** (Hothorn et al., 2011) is another alternative offering model-based boosting. The variable importance measure implemented for `ada()` is described by Hastie et al. (2001, pp. 331–332).

### Alternating Decision Trees—Using Weka

An alternating decision tree (Freund and Mason, 1997), combines the simplicity of a single decision tree with the effectiveness of boosting. The knowledge representation combines tree stumps, a common model deployed in boosting, into a decision tree type structure.

A key characteristic of the tree representation is that the different branches are no longer mutually exclusive. The root node is a prediction node and has just a numeric score. The next layer of nodes are decision nodes and are essentially a collection of decision tree stumps. The next layer then consists of prediction nodes, and so on, alternating between prediction nodes and decision nodes.

A model is deployed by identifying the possibly multiple paths from the root node to the leaves, through the alternating decision tree, that correspond to the values for the variables of an observation to be classified. The observation's classification score (or measure of confidence) is the sum of the prediction values along the corresponding paths.

The alternating decision tree algorithm is implemented in the Weka data mining suite. Weka is available directly from R through **RWeka** (Hornik et al., 2009), which provides its comprehensive collection of data mining tools within the R framework. A simple example will illustrate the incredible power that this offers—using R as a unifying interface to an extensive collection of data mining tools.

We can build an alternating decision tree in R using **RWeka** after installing the appropriate Weka package:

```
> library(RWeka)
> WPM("refresh-cache")
> WPM("install-package", "alternatingDecisionTrees")
```

We use `make_Weka_classifier()` to turn a Weka object into an R function:

```
> WPM("load-package", "alternatingDecisionTrees")
> cpath <- "weka/classifiers/trees/ADTree"
> ADT <- make_Weka_classifier(cpath)
```

We can obtain some background information about the resulting function by printing the value of the resulting variable:

```
> ADT

An R interface to Weka class
'weka.classifiers.trees.ADTree', which has
information

  Class for generating an alternating decision
  tree. The basic algorithm is based on:

  [...]

Argument list:
  (formula, data, subset, na.action, control =
  Weka_control(),
  options = NULL)

Returns objects inheriting from classes:
  Weka_classifier
```

The function `WOW()`, standing for "Weka option wizard", will list the command line arguments that become available with the generated function, as seen in the following code block:

```
> WOW(ADT)

-B      Number of boosting iterations.  (Default =
        10)
        Number of arguments: 1.
-E      Expand nodes: -3(all), -2(weight),
        -1(z_pure), >=0 seed for random walk
        (Default = -3)
        Number of arguments: 1.
-D      Save the instance data with the model
```

Next we perform our usual model building. As always we first create a container for the model, making available the appropriate dataset container for use from within this new container:

```
> weatherADT <- new.env(parent=weatherDS)
```

The model is built as a simple call to `ADT`:

```
> evalq({
    model <- ADT(formula=form, data=data[train, vars])
  }, weatherADT)
```

The resulting alternating decision tree can then be displayed as we see in the following code block.

```
> weatherADT$model

Alternating decision tree:

: -0.794
|  (1)Pressure3pm < 1011.9: 0.743
|  (1)Pressure3pm >= 1011.9: -0.463
|  |  (3)Temp3pm < 14.75: -1.498
|  |  (3)Temp3pm >= 14.75: 0.165
|  (2)Sunshine < 8.85: 0.405
|  |  (4)WindSpeed9am < 6.5: 0.656
|  |  (4)WindSpeed9am >= 6.5: -0.26
|  |  |  (8)Sunshine < 6.55: 0.298
|  |  |  |  (9)Temp3pm < 18.75: -0.595
|  |  |  |  (9)Temp3pm >= 18.75: 0.771
|  |  |  (8)Sunshine >= 6.55: -0.931
|  (2)Sunshine >= 8.85: -0.76
|  |  (6)MaxTemp < 24.35: -1.214
|  |  (6)MaxTemp >= 24.35: 0.095
|  |  |  (7)Sunshine < 10.9: 0.663
|  |  |  (7)Sunshine >= 10.9: -0.723
|  (5)Pressure3pm < 1016.1: 0.295
|  (5)Pressure3pm >= 1016.1: -0.658
|  |  (10)MaxTemp < 19.55: 0.332
|  |  (10)MaxTemp >= 19.55: -1.099
Legend: -ve = No, +ve = Yes
Tree size (total number of nodes): 31
Leaves (number of predictor nodes): 21
```

We can then explore how well the model performs:

```
> evalq({
    predictions <- predict(model, data[-train, vars])
    table(predictions, data[-train, target],
          dnn=c("Predicted", "Actual"))
  }, weatherADT)

         Actual
Predicted No Yes
      No  72  11
      Yes 13  14
```

Compare this with the `ada()` generated model:

```
> evalq({
    predictions <- predict(model, data[-train, vars])
    table(predictions, data[-train, target],
          dnn=c("Predicted", "Actual"))
  }, weatherADA)

         Actual
Predicted No Yes
      No  78  10
      Yes  7  15
```

In this example, the `ada()` model performs better than the `ADT()` model.

## 13.6 Summary

Boosting is an efficient, simple, and easy-to-understand model building strategy that tends not to overfit our data, hence building good models. The popular variant called AdaBoost (an abbreviation for adaptive boosting) has been described as the "best off-the-shelf classifier in the world" (attributed to Leo Breiman by Hastie et al. (2001, p. 302)).

Boosting algorithms build multiple models from a dataset, using some other model builders, such as a decision tree builder or neural network, that need not be particularly good model builders. The basic idea of boosting is to associate a weight with each observation in the dataset. A series of models are built and the weights are increased (boosted) if a

model incorrectly classifies the observation. The weights of such observations generally oscillate up and down from one model to the next. The final model is then an additive model constructed from the sequence of models, each model's output weighted by some score. There is little tuning required and little is assumed about the model builder used, except that it should be relatively weak model. We note that boosting can fail to perform if there is insufficient data or if the weak models are overly complex. Boosting is also susceptible to noise.

## 13.7 Command Summary

This chapter has referenced the following R packages, commands, functions, and datasets:

| | | |
|---|---|---|
| `ada()` | function | Implementation of AdaBoost. |
| **ada** | package | Builds AdaBoost models. |
| **caTools** | package | Provides `LogitBoost()`. |
| **gbm** | package | Generalised boosted regression. |
| `LogitBoost()` | function | Alternative boosting algorithm. |
| `predict()` | function | Applies model to new dataset. |
| `randomForest()` | function | Implementation of random forests. |
| **rattle** | package | The *weather* dataset and GUI. |
| `rpart()` | function | Builds a decision tree model. |
| `rpart.control()` | function | Controls `ada()` passes to `rpart()`. |
| **rpart** | package | Builds decision tree models. |
| **RWeka** | package | Interface Weka software. |
| `summary()` | function | Summarise an ada model. |
| `which()` | function | Elements of a vector that are TRUE. |
| `WOW()` | function | The Weka option wizard. |