11

Improving Model Performance

When a sports team falls short of meeting its goal—whether the goal is to obtain an Olympic gold medal, a league championship, or a world record time—it must search for possible improvements. Imagine that you're the team's coach. How would you spend your practice sessions? Perhaps you'd direct the athletes to train harder or train differently in order to maximize every bit of their potential. Or, you might emphasize better teamwork, utilizing the athletes' strengths and weaknesses more smartly.

Now imagine that you're training a world champion machine learning algorithm. Perhaps you hope to compete in data mining competitions such as those posted on Kaggle (`http://www.kaggle.com/`). Maybe you simply need to improve business results. Where do you begin? Although the context differs, the strategies one uses to improve a sports team performance can also be used to improve the performance of statistical learners.

As the coach, it is your job to find the combination of training techniques and teamwork skills that allow you to meet your performance goals. This chapter builds upon the material covered throughout this book to introduce a set of techniques for improving the predictive performance of machine learners. You will learn:

- How to automate model performance tuning by systematically searching for the optimal set of training conditions
- Methods for combining models into groups that use teamwork to tackle tough learning tasks
- How to apply a variant of decision trees that has quickly become popular due to its impressive performance

None of these methods will be successful for every problem. Yet looking at the winning entries to machine learning competitions, you'll likely find at least one of them has been employed. To be competitive, you too will need to add these skills to your repertoire.

Tuning stock models for better performance

Some learning problems are well suited to the stock models presented in previous chapters. In such cases, it may not be necessary to spend much time iterating and refining the model; it may perform well enough as it is. On the other hand, some problems are inherently more difficult. The underlying concepts to be learned may be extremely complex, requiring an understanding of many subtle relationships, or the problem may be affected by random variation, making it difficult to define the signal within the noise.

Developing models that perform extremely well on difficult problems is every bit an art as it is a science. Sometimes a bit of intuition is helpful when trying to identify areas where performance can be improved. In other cases, finding improvements will require a brute-force, trial-and-error approach. Of course, the process of searching numerous possible improvements can be aided by the use of automated programs.

In *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, we attempted a difficult problem: identifying loans that were likely to enter into default. Although we were able to use performance tuning methods to obtain a respectable classification accuracy of about 82 percent, upon a more careful examination in *Chapter 10, Evaluating Model Performance*, we realized that the high accuracy was a bit misleading. In spite of the reasonable accuracy, the kappa statistic was only about 0.28, which suggested that the model was actually performing somewhat poorly. In this section, we'll revisit the credit scoring model to see whether we can improve the results.

To follow along with the examples, download the `credit.csv` file from the Packt Publishing website and save it to your R working directory. Load the file into R using the following command: `credit <- read.csv("credit.csv")`.

You will recall that we first used a stock C5.0 decision tree to build the classifier for the credit data. We then attempted to improve its performance by adjusting the `trials` parameter to increase the number of boosting iterations. By increasing the number of iterations from the default of one up to the value of 10, we were able to increase the model's accuracy. This process of adjusting the model options to identify the best fit is called **parameter tuning**.

Parameter tuning is not limited to decision trees. For instance, we tuned k-NN models when we searched for the best value of *k*. We also tuned neural networks and support vector machines as we adjusted the number of nodes, the number of hidden layers, or chose different kernel functions. Most machine learning algorithms allow the adjustment of at least one parameter, and the most sophisticated models offer a large number of ways to tweak the model fit. Although this allows the model to be tailored closely to the learning task, the complexity of all the possible options can be daunting. A more systematic approach is warranted.

Using caret for automated parameter tuning

Rather than choosing arbitrary values for each of the model's parameters—a task that is not only tedious but also somewhat unscientific—it is better to conduct a search through many possible parameter values to find the best combination.

The `caret` package, which we used extensively in *Chapter 10, Evaluating Model Performance*, provides tools to assist with automated parameter tuning. The core functionality is provided by a `train()` function that serves as a standardized interface for over 200 different machine learning models for both classification and regression tasks. By using this function, it is possible to automate the search for optimal models using a choice of evaluation methods and metrics.

Do not feel overwhelmed by the large number of models—we've already covered many of them in earlier chapters. Others are simple variants or extensions of the base concepts. Given what you've learned so far, you should be confident that you have the ability to understand all of the available methods.

Automated parameter tuning requires you to consider three questions:

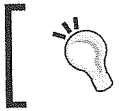
- What type of machine learning model (and specific implementation) should be trained on the data?
- Which model parameters can be adjusted, and how extensively should they be tuned to find the optimal settings?
- What criteria should be used to evaluate the models to find the best candidate?

Answering the first question involves finding a match between the machine learning task and one of the 200+ models available to the `caret` package. Obviously, this requires an understanding of the breadth and depth of machine learning models. It can also help to work through a process of elimination.

Nearly half of the models can be eliminated depending on whether the task is classification or numeric prediction; others can be excluded based on the format of the data or the need to avoid black box models, and so on. In any case, there's also no reason you can't try several approaches and compare the best results of each.

Addressing the second question is a matter largely dictated by the choice of model, since each algorithm utilizes a unique set of parameters. The available tuning parameters for the predictive models covered in this book are listed in the following table. Keep in mind that although some models have additional options not shown, only those listed in the table are supported by `caret` for automatic tuning.

Model	Learning Task	Method Name	Parameters
k-Nearest Neighbors	Classification	`knn`	`k`
Naive Bayes	Classification	`nb`	`fL, usekernel`
Decision Trees	Classification	`C5.0`	`model, trials, winnow`
OneR Rule Learner	Classification	`OneR`	None
RIPPER Rule Learner	Classification	`JRip`	`NumOpt`
Linear Regression	Regression	`lm`	None
Regression Trees	Regression	`rpart`	`cp`
Model Trees	Regression	`M5`	`pruned, smoothed, rules`
Neural Networks	Dual Use	`nnet`	`size, decay`
Support Vector Machines (Linear Kernel)	Dual Use	`svmLinear`	`C`
Support Vector Machines (Radial Basis Kernel)	Dual Use	`svmRadial`	`C, sigma`
Random Forests	Dual Use	`rf`	`mtry`



For a complete list of the models and corresponding tuning parameters covered by `caret`, refer to the table provided by package author Max Kuhn at `http://topepo.github.io/caret/modelList.html`

If you ever forget the tuning parameters for a particular model, the `modelLookup()` function can be used to find them. Simply supply the method name as illustrated for the C5.0 model:

```
> modelLookup("C5.0")
  model parameter                   label forReg forClass probModel
1  C5.0    trials # Boosting Iterations  FALSE     TRUE      TRUE
2  C5.0     model            Model Type  FALSE     TRUE      TRUE
3  C5.0    winnow                Winnow  FALSE     TRUE      TRUE
```

The goal of automatic tuning is to search a set of candidate models comprising a matrix, or **grid**, of parameter combinations. Because it is impractical to search every conceivable combination, only a subset of possibilities is used to construct the grid. By default, `caret` searches at most three values for each of the model's *p* parameters, which means that at most 3^p candidate models will be tested. For example, by default, the automatic tuning of k-NN will compare *3^1 = 3* candidate models with `k=5`, `k=7`, and `k=9`. Similarly, tuning a decision tree will result in a comparison of up to 27 different candidate models, comprising the grid of *3^3 = 27* combinations of `model`, `trials`, and `winnow` settings. In practice, however, only 12 models are actually tested. This is because the `model` and `winnow` parameters can only take two values (`tree` versus `rules` and `TRUE` versus `FALSE`, respectively), which makes the grid size *3*2*2 = 12*.

Since the default search grid may not be ideal for your learning problem, `caret` allows you to provide a custom search grid defined by a simple command that we will cover later.

The third and final step in automatic model tuning involves identifying the best model among the candidates. This uses the methods discussed in *Chapter 10, Evaluating Model Performance*, including the choice of resampling strategy for creating training and test datasets, and the use of model performance statistics to measure the predictive accuracy.

All of the resampling strategies and many of the performance statistics we've learned are supported by `caret`. These include statistics such as accuracy and kappa (for classifiers), and R-squared or RMSE (for numeric models). Cost-sensitive measures like sensitivity, specificity, and AUC can also be used if desired.

By default, `caret` will select the candidate model with the best value of the desired performance measure. Because this practice sometimes results in the selection of models that achieve minor performance improvements via large increases in model complexity, alternative model selection functions are provided.

Given the wide variety of options, it is helpful that many of the defaults are reasonable. For instance, `caret` will use prediction accuracy on a bootstrap sample to choose the best performer for classification models. Beginning with these default values, we can then tweak the `train()` function to design a wide variety of experiments.

Creating a simple tuned model

To illustrate the process of tuning a model, let's begin by observing what happens when we attempt to tune the credit scoring model using the `caret` package's default settings. From there, we will adjust the options to our liking.

The simplest way to tune a learner requires only that you specify a model type via the `method` parameter. Since we used C5.0 decision trees previously with the credit model, we'll continue our work by optimizing this learner. The basic `train()` command for tuning a C5.0 decision tree using the default settings is as follows:

```
> library(caret)
> RNGversion("3.5.2")
> set.seed(300)
> m <- train(default ~ ., data = credit, method = "C5.0")
```

First, the `set.seed()` function is used to initialize R's random number generator to a set starting position. You may recall that we used this function in several prior chapters. By setting the `seed` parameter (in this case to the arbitrary number `300`), the random numbers will follow a predefined sequence. This allows simulations that use random sampling to be repeated with identical results—a very helpful feature if you are sharing code or attempting to replicate a prior result.

Next, we define a tree as `default ~ .` using the R formula interface. This models loan default status (`yes` or `no`) using all of the other features in the `credit` data frame. The parameter `method = "C5.0"` tells `caret` to use the C5.0 decision tree algorithm.

After you've entered the preceding command, there may be a significant delay (depending upon your computer's capabilities) as the tuning process occurs. Even though this is a fairly small dataset, a substantial amount of calculation must occur. R must repeatedly generate random samples of data, build decision trees, compute performance statistics, and evaluate the result.

The result of the experiment is saved in an object named `m`. If you would like to examine the object's contents, the command `str(m)` will list all the associated data, but this can be substantial. Instead, simply type the name of the object for a condensed summary of the results. For instance, typing `m` yields the following output (note that labels have been added for clarity):

```
1000 samples
  16 predictor
   2 classes: 'no', 'yes'

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 1000, 1000, 1000, 1000, 1000, 1000, ...
Resampling results across tuning parameters:

  model  winnow  trials  Accuracy   Kappa
  rules  FALSE    1      0.6960037  0.2750983
  rules  FALSE   10      0.7147884  0.3181988
  rules  FALSE   20      0.7233793  0.3342634
  rules   TRUE    1      0.6849914  0.2513442
  rules   TRUE   10      0.7126357  0.3156326
  rules   TRUE   20      0.7225179  0.3342797
  tree   FALSE    1      0.6888248  0.2487963
  tree   FALSE   10      0.7310421  0.3148572
  tree   FALSE   20      0.7362375  0.3271043
  tree    TRUE    1      0.6814831  0.2317101
  tree    TRUE   10      0.7285510  0.3093354
  tree    TRUE   20      0.7324992  0.3200752

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were trials = 20, model = tree
and winnow = FALSE.
```

The labels highlight four main components in the output:

1. **A brief description of the input dataset**: If you are familiar with your data and have applied the `train()` function correctly, this information should not be surprising.
2. **A report of the preprocessing and resampling methods applied**: Here we see that 25 bootstrap samples, each including 1,000 examples, were used to train the models.
3. **A list of the candidate models evaluated**: In this section, we can confirm that 12 different models were tested based on the combinations of three C5.0 tuning parameters: `model`, `trials`, and `winnow`. The average accuracy and kappa statistics for each candidate model are also shown.
4. **The choice of best model**: As the footnote describes, the model with the largest accuracy was selected. This was the C5.0 model that used a decision tree with 20 trials and the setting `winnow = FALSE`.

After identifying the best model, the `train()` function uses its tuning parameters to build a model on the full input dataset, which is stored in the `m` list object as `m$finalModel`. In most cases, you will not need to work directly with the `finalModel` sub-object. Instead, simply use the `predict()` function with the `m` object as follows:

```
> p <- predict(m, credit)
```

The resulting vector of predictions works as expected, allowing us to create a confusion matrix that compares the predicted and actual values:

```
> table(p, credit)

p       no yes
  no   700   2
  yes    0 298
```

Of the 1,000 examples used for training the final model, only two were misclassified. However, it is very important to note that since the model was built on both the training and test data, this accuracy is optimistic and thus should not be viewed as indicative of performance on unseen data. The bootstrap estimate of 73 percent (shown in the `train()` model summary output) is a more realistic estimate of future performance.

In addition to the automatic parameter tuning, using the `caret` package's `train()` and `predict()` functions also offers a couple of benefits beyond the functions found in the stock packages.

First, any data preparation steps applied by the `train()` function will be similarly applied to the data used for generating predictions. This includes transformations like centering and scaling, as well as imputation of missing values. Allowing `caret` to handle the data preparation will ensure that the steps that contributed to the best model's performance will remain in place when the model is deployed.

Second, the `predict()` function provides a standardized interface for obtaining predicted class values and predicted class probabilities, even for model types that ordinarily would require additional steps to obtain this information. The predicted classes are provided by default:

```
> head(predict(m, credit))
[1] no  yes no  no  yes no
Levels: no yes
```

To obtain the estimated probabilities for each class, use the `type = "prob"` parameter:

```
> head(predict(m, credit, type = "prob"))
         no         yes
1 0.9606970 0.03930299
2 0.1388444 0.86115561
3 1.0000000 0.00000000
4 0.7720279 0.22797208
5 0.2948062 0.70519385
6 0.8583715 0.14162851
```

Even in cases where the underlying model refers to the prediction probabilities using a different string (for example, `"raw"` for a `naiveBayes` model), the `predict()` function will translate `type = "prob"` to the appropriate parameter setting automatically.

Customizing the tuning process

The decision tree we created previously demonstrates the `caret` package's ability to produce an optimized model with minimal intervention. The default settings allow optimized models to be created easily. However, it is also possible to change the default settings to something more specific to a learning task, which may assist with unlocking the upper echelon of performance.

Each step in the model selection process can be customized. To illustrate this flexibility, let's modify our work on the credit decision tree to mirror the process we used in *Chapter 10, Evaluating Model Performance*. If you recall, we had estimated the kappa statistic using 10-fold CV. We'll do the same here, using kappa to optimize the boosting parameter of the decision tree. Note that decision tree boosting was previously covered in *Chapter 5, Divide and Conquer - Classification Using Decision Trees and Rules*, and will also be covered in greater detail later in this chapter.

The `trainControl()` function is used to create a set of configuration options known as a **control object**. This object guides the `train()` function and allows for the selection of model evaluation criteria, such as the resampling strategy and the measure used for choosing the best model. Although this function can be used to modify nearly every aspect of a tuning experiment, we'll focus on two important parameters: `method` and `selectionFunction`.

If you're eager for more details, you can use the `?trainControl` command for a list of all the parameters.

For the `trainControl()` function, the `method` parameter is used to set the resampling method, such as holdout sampling or k-fold CV. The following table lists the possible `method` types, as well as any additional parameters for adjusting the sample size and number of iterations. Although the default options for these resampling methods follow popular convention, you may choose to adjust these depending upon the size of your dataset and the complexity of your model.

Resampling Method	Method Name	Additional Options and Default Values
Holdout sampling	`LGOCV`	`p = 0.75` (training data proportion)
k-fold CV	`cv`	`number = 10` (number of folds)
Repeated k-fold CV	`repeatedcv`	`number = 10` (number of folds) `repeats = 10` (number of iterations)
Bootstrap sampling	`boot`	`number = 25` (resampling iterations)
0.632 bootstrap	`boot632`	`number = 25` (resampling iterations)
Leave-one-out CV	`LOOCV`	None

The `selectionFunction` parameter is used to specify the function that will choose the optimal model among the various candidates. Three such functions are included. The `best` function simply chooses the candidate with the best value on the specified performance measure. This is used by default. The other two functions are used to choose the most parsimonious, or simplest, model that is within a certain threshold of the best model's performance. The `oneSE` function chooses the simplest candidate within one standard error of the best performance, and `tolerance` uses the simplest candidate within a user-specified percentage.

Some subjectivity is involved with the `caret` package's ranking of models by simplicity. For information on how models are ranked, see the help page for the selection functions by typing `?best` at the R command prompt.

To create a control object named `ctrl` that uses 10-fold CV and the `oneSE` selection function, use the following command (note that `number = 10` is included only for clarity; since this is the default value for `method = "cv"`, it could have been omitted):

```
> ctrl <- trainControl(method = "cv", number = 10,
                       selectionFunction = "oneSE")
```

We'll use the result of this function shortly.

In the meantime, the next step in defining our experiment is to create the grid of parameters to optimize. The grid must include a column named for each tuning parameter in the desired model. It must also include a row for each desired combination of parameter values. Since we are using a C5.0 decision tree, this means we'll need columns named `model`, `trials`, and `winnow`. For other machine learning models, refer to the table presented earlier in this chapter or use the `modelLookup()` function to lookup the parameters as described previously.

Rather than filling this data frame cell by cell—a tedious task if there are many possible combinations of parameter values—we can use the `expand.grid()` function, which creates data frames from the combinations of all values supplied. For example, suppose we would like to hold constant `model = "tree"` and `winnow = FALSE` while searching eight different values of trials. This can be created as:

```
> grid <- expand.grid(model = "tree",
                      trials = c(1, 5, 10, 15, 20, 25, 30, 35),
                      winnow = FALSE)
```

The resulting `grid` data frame contains *1*8*1 = 8* rows:

```
> grid
  model trials winnow
1  tree      1  FALSE
2  tree      5  FALSE
3  tree     10  FALSE
4  tree     15  FALSE
5  tree     20  FALSE
6  tree     25  FALSE
7  tree     30  FALSE
8  tree     35  FALSE
```

The `train()` function will build a candidate model for evaluation using each row's combination of model parameters.

Given this search grid and the control list created previously, we are ready to run a thoroughly customized `train()` experiment. As before, we'll set the random seed to the arbitrary number `300` in order to ensure repeatable results. But this time, we'll pass our control object and tuning grid while adding a parameter `metric = "Kappa"`, indicating the statistic to be used by the model evaluation function—in this case, `"oneSE"`. The full command is as follows:

```
> RNGversion("3.5.2")
> set.seed(300)
```

```
> m <- train(default ~ ., data = credit, method = "C5.0",
             metric = "Kappa",
             trControl = ctrl,
             tuneGrid = grid)
```

This results in an object that we can view by typing its name:

```
> m

1000 samples
  16 predictor
   2 classes: 'no', 'yes'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
Resampling results across tuning parameters:

  trials  Accuracy  Kappa
   1      0.735     0.3243679
   5      0.722     0.2941429
  10      0.725     0.2954364
  15      0.731     0.3141866
  20      0.737     0.3245897
  25      0.726     0.2972530
  30      0.735     0.3233492
  35      0.736     0.3193931

Tuning parameter 'model' was held constant at a value of tree
Tuning parameter 'winnow' was held constant at  a value of FALSE
Kappa was used to select the optimal model using  the one SE rule.
The final values used for the model were trials = 1, model = tree
and winnow = FALSE.
```

Although much of the output is similar to the automatically tuned model, there are a few differences of note. Because 10-fold CV was used, the sample size to build each candidate model was reduced to 900 rather than the 1,000 used in the bootstrap. As we requested, eight candidate models were tested. Additionally, because `model` and `winnow` were held constant, their values are no longer shown in the results; instead, they are listed as a footnote.

The best model here differs quite significantly from the prior experiment. Before, the best model used `trials = 20`, whereas here, it used `trials = 1`. This change is due to the fact that we used the `oneSE` rule rather than the `best` rule to select the optimal model. Even though the 35-trial model offers the best raw performance according to kappa, the single-trial model offers nearly the same performance with a much simpler algorithm.

Due to the large number of configuration parameters, `caret` can seem overwhelming at first. Don't let this deter you—there is no easier way to test the performance of models using 10-fold CV. Instead, think of the experiment as defined by two parts: a `trainControl()` object that dictates the testing criteria, and a tuning grid that determines what model parameters to evaluate. Supply these to the `train()` function and with a bit of computing time, your experiment will be complete!

Improving model performance with meta-learning

As an alternative to increasing the performance of a single model, it is possible to combine several models to form a powerful team. Just as the best sports teams have players with complementary rather than overlapping skillsets, some of the best machine learning algorithms utilize teams of complementary models. Since a model brings a unique bias to a learning task, it may readily learn one subset of examples, but have trouble with another. Therefore, by intelligently using the talents of several diverse team members, it is possible to create a strong team of multiple weak learners.

This technique of combining and managing the predictions of multiple models falls into a wider set of **meta-learning** methods, which are techniques that involve learning how to learn. This includes anything from simple algorithms that gradually improve performance by iterating over design decisions—for instance, the automated parameter tuning used earlier in this chapter—to highly complex algorithms that use concepts borrowed from evolutionary biology and genetics for self-modifying and adapting to learning tasks.

For the remainder of this chapter, we'll focus on meta-learning only as it pertains to modeling a relationship between the predictions of several models and the desired outcome. The teamwork-based techniques covered here are quite powerful and are used quite often to build more effective classifiers.

Understanding ensembles

Suppose you were a contestant on a television trivia show that allowed you to choose a panel of five friends to assist you with answering the final question for the million-dollar prize. Most people would try to stack the panel with a diverse set of subject matter experts. A panel containing professors of literature, science, history, and art, along with a current pop-culture expert would be safely well rounded. Given their breadth of knowledge, it would be unlikely that a question would stump the group.

The meta-learning approach that utilizes a similar principle of creating a varied team of experts is known as an **ensemble**. All ensemble methods are based on the idea that by combining multiple weaker learners, a stronger learner is created. The various ensemble methods can be distinguished, in large part, by the answers to two questions:

- How are the weak learning models chosen and/or constructed?
- How are the weak learners' predictions combined to make a single final prediction?

When answering these questions, it can be helpful to imagine the ensemble in terms of the following process diagram; nearly all ensemble approaches follow this pattern:

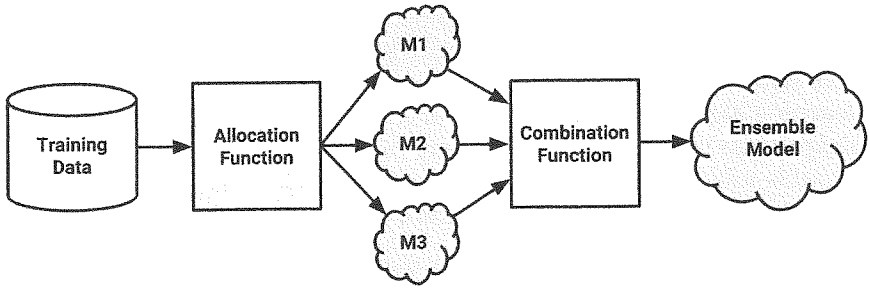


Figure 11.1: Ensembles combine multiple weaker models into a single stronger model

First, input training data is used to build a number of models. The **allocation function** dictates how much of the training data each model receives. Do they each receive the full training dataset or merely a sample? Do they each receive every feature or a subset?

Although the ideal ensemble includes a diverse set of models, the allocation function can increase diversity by artificially varying the input data to bias the resulting learners, even if they are the same type. For instance, in an ensemble of decision trees, the allocation function might use bootstrap sampling to construct unique training datasets for each tree, or it may pass each one a different subset of features. On the other hand, if the ensemble already includes a diverse set of algorithms—such as a neural network, a decision tree, and a k-NN classifier—then the allocation function might pass the data on to each algorithm relatively unchanged.

After the ensemble's models are constructed, they can be used to generate a set of predictions, which must be managed in some way. The **combination function** governs how disagreements among the predictions are reconciled. For example, the ensemble might use a majority vote to determine the final prediction, or it could use a more complex strategy such as weighting each model's votes based on its prior performance.

Some ensembles even utilize another model to learn a combination function from various combinations of predictions. For example, suppose that when M_1 and M_2 both vote yes, the actual class value is usually no. In this case, the ensemble could learn to ignore the vote of M_1 and M_2 when they agree. This process of using the predictions of several models to train a final arbiter model is known as **stacking**.

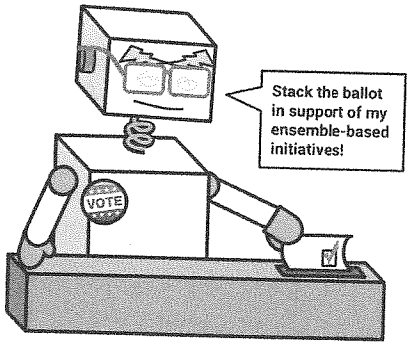


Figure 11.2: Stacking is a sophisticated ensemble that uses a learning algorithm to combine predictions

One of the benefits of using ensembles is that they may allow you to spend less time in pursuit of a single best model. Instead, you can train a number of reasonably strong candidates and combine them. Yet convenience isn't the only reason why ensemble-based methods continue to rack up wins in machine learning competitions; ensembles also offer a number of performance advantages over single models:

- **Better generalizability to future problems**: As the opinions of several learners are incorporated into a single final prediction, no single bias is able to dominate. This reduces the chance of overfitting to a learning task.
- **Improved performance on massive or miniscule datasets**: Many models run into memory or complexity limits when an extremely large set of features or examples are used, making it more efficient to train several small models than a single full model. Conversely, ensembles also do well on the smallest datasets because resampling methods like bootstrapping are inherently part of many ensemble designs. Perhaps most importantly, it is often possible to train an ensemble in parallel using distributed computing methods.
- **The ability to synthesize data from distinct domains**: Since there is no one-size-fits-all learning algorithm, the ensemble's ability to incorporate evidence from multiple types of learners is increasingly important as complex phenomena rely on data drawn from diverse domains.
- **A more nuanced understanding of difficult learning tasks**: Real-world phenomena are often extremely complex, with many interacting intricacies. Models that divide the task into smaller portions are likely to more accurately capture subtle patterns that a single global model might miss.

None of these benefits would be very helpful if you weren't able to easily apply ensemble methods in R, and there are many packages available to do just that. Let's take a look at several of the most popular ensemble methods and how they can be used to improve the performance of the credit model we've been working on.

Bagging

One of the first ensemble methods to gain widespread acceptance used a technique called **bootstrap aggregating**, or **bagging** for short. As described by Leo Breiman in 1994, bagging generates a number of training datasets by bootstrap sampling the original training data. These datasets are then used to generate a set of models using a single learning algorithm. The models' predictions are combined using voting (for classification) or averaging (for numeric prediction).

For additional information on bagging, refer to *Bagging predictors, Breiman, L, Machine Learning, 1996, Vol. 24, pp. 123-140.*

Although bagging is a relatively simple ensemble, it can perform quite well as long as it is used with relatively **unstable** learners, that is, those generating models that tend to change substantially when the input data changes only slightly. Unstable models are essential in order to ensure the ensemble's diversity in spite of only minor variations between the bootstrap training datasets. For this reason, bagging is often used with decision trees, which have the tendency to vary dramatically given minor changes in input data.

The `ipred` package offers a classic implementation of bagged decision trees. To train the model, the `bagging()` function works similarly to many of the models used previously. The `nbagg` parameter is used to control the number of decision trees voting in the ensemble (with a default value of 25). Depending on the difficulty of the learning task and the amount of training data, increasing this number may improve the model's performance, up to a limit. The downside is that this creates additional computational expense, and a large number of trees may take some time to train.

After installing the `ipred` package, we can create the ensemble as follows. We'll stick to the default value of 25 decision trees:

```
> library(ipred)
> RNGversion("3.5.2")
> set.seed(300)
> mybag <- bagging(default ~ ., data = credit, nbagg = 25)
```

The resulting model works as expected with the `predict()` function:

```
> credit_pred <- predict(mybag, credit)
> table(credit_pred, credit$default)

credit_pred  no yes
        no  699   2
        yes   1 298
```

Given the preceding results, the model seems to have fit the training data extremely well. To see how this translates into future performance, we can use the bagged trees with 10-fold CV using the `train()` function in the `caret` package. Note that the method name for the `ipred` bagged trees function is `treebag`:

```
> library(caret)
> RNGversion("3.5.2")
> set.seed(300)
> ctrl <- trainControl(method = "cv", number = 10)
> train(default ~ ., data = credit, method = "treebag",
        trControl = ctrl)

Bagged CART

1000 samples
  16 predictor
   2 classes: 'no', 'yes'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
Resampling results:

  Accuracy  Kappa
  0.746     0.3540389
```

The kappa statistic of 0.35 for this model suggests that the bagged tree model performs at least as well as the best C5.0 decision tree we tuned earlier in this chapter, which had a 0.32 kappa statistic. This illustrates the power of ensemble methods; a set of simple learners, working together, can outperform very sophisticated models.

Boosting

Another common ensemble-based method is called **boosting**, because it boosts the performance of weak learners to attain the performance of stronger learners. This method is based largely on the work of Robert Schapire and Yoav Freund, who have published extensively on the topic.

For additional information on boosting, refer to *Boosting: Foundations and Algorithms, Schapire, RE, Freund, Y, Cambridge, MA: The MIT Press, 2012.*

Similar to bagging, boosting uses ensembles of models trained on resampled data and a vote to determine the final prediction. There are two key distinctions. First, the resampled datasets in boosting are constructed specifically to generate complementary learners. Second, rather than giving each learner an equal vote, boosting gives each learner's vote a weight based on its past performance. Models that perform better have greater influence over the ensemble's final prediction.

Boosting will result in performance that is often better and certainly no worse than the best of the models in the ensemble. Since the models in the ensemble are built to be complementary, it is possible to increase ensemble performance to an arbitrary threshold simply by adding additional classifiers to the group, assuming that each additional classifier performs better than random chance. Given the obvious utility of this finding, boosting is thought to be one of the most significant discoveries in machine learning.

Although boosting can create a model that meets an arbitrarily low error rate, this may not always be reasonable in practice. For one, the performance gains are incrementally smaller as additional learners are gained, making some thresholds practically infeasible. Additionally, the pursuit of pure accuracy may result in the model being overfitted to the training data and not generalizable to unseen data.

A boosting algorithm called **AdaBoost**, or **adaptive boosting**, was proposed by Freund and Schapire in 1997. The algorithm is based on the idea of generating weak leaners that iteratively learn a larger portion of the difficult-to-classify examples in the training data by paying more attention (that is, giving more weight) to often misclassified examples.

Beginning from an unweighted dataset, the first classifier attempts to model the outcome. Examples that the classifier predicted correctly will be less likely to appear in the training dataset for the following classifier, and conversely, the difficult-to-classify examples will appear more frequently.

As additional rounds of weak learners are added, they are trained on data with successively more difficult examples. The process continues until the desired overall error rate is reached or performance no longer improves. At that point, each classifier's vote is weighted according to its accuracy on the training data on which it was built.

Though boosting principles can be applied to nearly any type of model, the principles are most commonly used with decision trees. We already used boosting in this way in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*, as a method to improve the performance of a C5.0 decision tree.

The **AdaBoost.M1** algorithm provides another tree-based implementation of AdaBoost for classification. The AdaBoost.M1 algorithm can be found in the `adabag` package.

For more information about the `adabag` package, refer to *adabag: An R Package for Classification with Boosting and Bagging, Alfaro, E, Gamez, M, Garcia, N, Journal of Statistical Software, 2013, Vol. 54, pp. 1-35.*

Let's create an AdaBoost.M1 classifier for the credit data. The general syntax for this algorithm is similar to other modeling techniques:

```
> RNGversion("3.5.2")
> set.seed(300)
> m_adaboost <- boosting(default ~ ., data = credit)
```

As usual, the `predict()` function is applied to the resulting object to make predictions:

```
> p_adaboost <- predict(m_adaboost, credit)
```

Departing from convention, rather than returning a vector of predictions, this returns an object with information about the model. The predictions are stored in a sub-object called `class`:

```
> head(p_adaboost$class)
[1] "no"  "yes" "no"  "no"  "yes" "no"
```

A confusion matrix can be found in the `confusion` sub-object:

```
> p_adaboost$confusion
                Observed Class
Predicted Class  no yes
             no  700   0
             yes   0 300
```

Did you notice that the AdaBoost model made no mistakes? Before you get your hopes up, remember that the preceding confusion matrix is based on the model's performance on the training data. Since boosting allows the error rate to be reduced to an arbitrarily low level, the learner simply continued until it made no more errors. This likely resulted in overfitting on the training dataset.

For a more accurate assessment of performance on unseen data, we need to use another evaluation method. The `adabag` package provides a simple function to use 10-fold CV:

```
> RNGversion("3.5.2")
> set.seed(300)
> adaboost_cv <- boosting.cv(default ~ ., data = credit)
```

Depending on your computer's capabilities, this may take some time to run, during which it will log each iteration to screen—on my recent MacBook Pro computer, it took about four minutes. After it completes, we can view a more reasonable confusion matrix:

```
> adaboost_cv$confusion
                Observed Class
Predicted Class  no yes
            no  594 151
            yes 106 149
```

We can find the kappa statistic using the `vcd` package as described in *Chapter 10, Evaluating Model Performance*:

```
> library(vcd)
> Kappa(adaboost_cv$confusion)
             value     ASE     z  Pr(>|z|)
Unweighted 0.3607 0.0323 11.17 5.914e-29
Weighted   0.3607 0.0323 11.17 5.914e-29
```

With a kappa of about 0.361, this is our best-performing credit scoring model yet. Let's see how it compares to one last ensemble method.

The AdaBoost.M1 algorithm can be tuned in `caret` by specifying `method = "AdaBoost.M1"`.

Random forests

Another ensemble-based method called **random forests** (or **decision tree forests**) focuses only on ensembles of decision trees. This method was championed by Leo Breiman and Adele Cutler, and combines the base principles of bagging with random feature selection to add additional diversity to the decision tree models. After the ensemble of trees (the forest) is generated, the model uses a vote to combine the trees' predictions.

For more detail on how random forests are constructed, refer to *Random Forests, Breiman, L, Machine Learning, 2001, Vol. 45, pp. 5-32.*

Random forests combine versatility and power into a single machine learning approach. Because the ensemble uses only a small, random portion of the full feature set, random forests can handle extremely large datasets, where the so-called "curse of dimensionality" might cause other models to fail. At the same time, its error rates for most learning tasks are on a par with nearly any other method.

Although the term "random forests" is trademarked by Breiman and Cutler, the term is sometimes used colloquially to refer to any type of decision tree ensemble. A pedant would use the more general term "decision tree forests" except when referring to the specific implementation by Breiman and Cutler.

It's worth noting that relative to other ensemble-based methods, random forests are quite competitive and offer key advantages. For instance, random forests tend to be easier to use and less prone to overfitting. The following table lists the general strengths and weaknesses of random forest models:

Strengths	Weaknesses
• An all-purpose model that performs well on most problems • Can handle noisy or missing data, as well as categorical or continuous features • Selects only the most important features • Can be used on data with an extremely large number of features or examples	• Unlike a decision tree, the model is not easily interpretable

Due to their power, versatility, and ease of use, random forests are one of the most popular machine learning methods. Later in this chapter, we'll compare a random forest model head-to-head against the boosted C5.0 tree.

Training random forests

Though there are several packages to create random forests in R, the `randomForest` package is perhaps the implementation most faithful to the specification by Breiman and Cutler, and is also supported by `caret` for automated tuning. The syntax for training this model is as follows:

Random forest syntax

using the `randomForest()` function in the `randomForest` package

Building the classifier:

```
m <- randomForest(train, class, ntree = 500, mtry = sqrt(p))
```

- `train` is a data frame containing training data
- `class` is a factor vector with the class for each row in the training data
- `ntree` is an integer specifying the number of trees to grow
- `mtry` is an optional integer specifying the number of features to randomly select at each split (uses `sqrt(p)` by default, where `p` is the number of features in the data)

The function will return a random forest object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "response")
```

- `m` is a model trained by the `randomForest()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier
- `type` is either `"response"`, `"prob"`, or `"votes"` and is used to indicate whether the predictions vector should contain the predicted class, the predicted probabilities, or a matrix of vote counts, respectively.

The function will return predictions according to the value of the `type` parameter.

Example:

```
credit_model <- randomForest(credit_train, loan_default)
credit_prediction <- predict(credit_model, credit_test)
```

By default, the `randomForest()` function creates an ensemble of 500 trees that consider `sqrt(p)` random features at each split, where `p` is the number of features in the training dataset and `sqrt()` refers to R's square root function. Whether or not these default parameters are appropriate depends on the nature of the learning task and training data. Generally, more complex learning problems and larger datasets (both more features as well as more examples) work better with a larger number of trees, though this needs to be balanced with the computational expense of training more trees.

The goal of using a large number of trees is to train enough that each feature has a chance to appear in several models. This is the basis of the `sqrt(p)` default value for the `mtry` parameter; using this value limits the features sufficiently such that substantial random variation occurs from tree to tree. For example, since the credit data has 16 features, each tree would be limited to splitting on four features at any time.

Let's see how the default `randomForest()` parameters work with the credit data. We'll train the model just as we have done with other learners. Again, the `set.seed()` function ensures that the result can be replicated:

```
> library(randomForest)
> RNGversion("3.5.2")
> set.seed(300)
> rf <- randomForest(default ~ ., data = credit)
```

To look at a summary of the model's performance, we can simply type the resulting object's name:

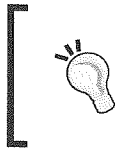
```
> rf

Call:
 randomForest(formula = default ~ ., data = credit)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4

        OOB estimate of  error rate: 23.3%
Confusion matrix:
     no yes class.error
no  638  62  0.08857143
yes 171 129  0.57000000
```

The output shows that the random forest included 500 trees and tried four variables at each split, as expected. At first glance, you might be alarmed at the seemingly poor performance according to the confusion matrix—the error rate of 23.3 percent is far worse than the resubstitution error of any of the other ensemble methods so far. However, this confusion matrix does not show resubstitution error. Instead, it reflects the **out-of-bag error rate** (listed in the output as `OOB estimate of error rate`), which, unlike resubstitution error, is an unbiased estimate of the test set error. This means that it should be a fairly reasonable estimate of future performance.

The out-of-bag estimate is computed during the construction of the random forest. Essentially, any example not selected for a single tree's bootstrap sample can be used to test the model's performance on unseen data. At the end of the forest construction, for each of the 1,000 examples in the dataset, the trees that did not use the example in training are allowed to make a prediction. These predictions are tallied for each example and a vote is taken to determine the single final prediction for the example. The total error rate of such predictions becomes the out-of-bag error rate.



In *Chapter 10, Evaluating Model Performance,* it was stated that any given example has a 63.2 percent chance of being included in a bootstrap sample. This implies that an average of 36.8 percent of the 500 trees in the random forest voted for each of the 1,000 examples in the out-of-bag estimate.

To calculate the kappa statistic on the out-of-bag predictions, we can use the function in the `vcd` package as follows. The code applies the `Kappa()` function to the first two rows and columns of the `confusion` object, which stores the confusion matrix of the out-of-bag predictions for the `rf` random forest model object:

```
> library(vcd)
> Kappa(rf$confusion[1:2,1:2])
            value     ASE     z  Pr(>|z|)
Unweighted 0.381 0.03215 11.85 2.197e-32
Weighted   0.381 0.03215 11.85 2.197e-32
```

With a kappa statistic of 0.381, the random forest is our best-performing model yet. It was higher than the best boosted C5.0 decision tree, which had a kappa of about 0.325, and also higher than the AdaBoost.M1 model, which had a kappa of about 0.361. Given this impressive initial result, we should attempt a more formal evaluation of its performance.

Evaluating random forest performance in a simulated competition

As mentioned previously, the `randomForest()` function is supported by `caret`, which allows us to optimize the model while at the same time calculating performance measures beyond the out-of-bag error rate. To make things interesting, let's compare an auto-tuned random forest to the best auto-tuned boosted C5.0 model we've developed. We'll treat this experiment as if we were hoping to identify a candidate model for submission to a machine learning competition.

We must first load `caret` and set our training control options. For the most accurate comparison of model performance, we'll use repeated 10-fold CV, or 10-fold CV repeated 10 times. This means that the models will take a much longer time to build and will be more computationally intensive to evaluate, but since this is our final comparison, we should be *very* sure that we're making the right choice; the winner of this showdown, selected via the "best" performance metric, will be our only entry into the machine learning competition.

Additionally, we'll add a few new options to the `trainControl()` function. First, we'll set the `savePredictions` and `classProbs` parameters to `TRUE`, which saves the holdout sample predictions and predicted probabilities for plotting the ROC curve later. Then, we'll also set the `summaryFunction` to `twoClassSummary`, which is a `caret` function that computes performance metrics like AUC. The full control object is defined as follows:

```
> library(caret)
> ctrl <- trainControl(method = "repeatedcv",
                       number = 10, repeats = 10,
                       selectionFunction = "best",
                       savePredictions = TRUE,
                       classProbs = TRUE,
                       summaryFunction = twoClassSummary)
```

Next, we'll set up the tuning grid for the random forest. The only tuning parameter for this model is `mtry`, which defines how many features are randomly selected at each split. By default, we know that the random forest will use `sqrt(16)`, or four features per tree. To be thorough, we'll also test values half of that, and twice that, as well as the full set of 16 features. Thus, we need to create a grid with values of `2`, `4`, `8`, and `16` as follows:

```
> grid_rf <- expand.grid(mtry = c(2, 4, 8, 16))
```

A random forest that considers the full set of features at each split is essentially the same as a bagged decision tree model.

We'll supply the resulting grid to the `train()` function with the `ctrl` object, and use the `"ROC"` metric to select the best model. This metric refers to the area under the ROC curve. The complete experiment can be run as follows:

```
> RNGversion("3.5.2"); set.seed(300)
> m_rf <- train(default ~ ., data = credit, method = "rf",
                metric = "ROC", trControl = ctrl,
                tuneGrid = grid_rf)
```

The preceding command may take some time to complete, as it has quite a bit of work to do—on my recent MacBook Pro computer, it took about seven minutes! When the random forests are finished training, we'll compare the best forest to the best boosted decision tree among trees with `10`, `25`, `50`, and `100` iterations using the following `caret` experiment:

```
> grid_c50 <- expand.grid(model = "tree",
                          trials = c(10, 25, 50, 100),
                          winnow = FALSE)
> RNGversion("3.5.2"); set.seed(300)
> m_c50 <- train(default ~ ., data = credit, method = "C5.0",
                 metric = "ROC", trControl = ctrl,
                 tuneGrid = grid_c50)
```

When the C5.0 decision tree finally completes, we can compare the two approaches side by side. For the random forest model, the results are:

```
> m_rf

Resampling results across tuning parameters:

  mtry  ROC        Sens       Spec
   2    0.7579643  0.9900000  0.09766667
   4    0.7695071  0.9377143  0.30166667
   8    0.7739714  0.9064286  0.38633333
  16    0.7747905  0.8921429  0.44100000
```

For the boosted C5.0 model, the results are:

```
> m_c50

Resampling results across tuning parameters:

  trials  ROC        Sens       Spec
   10     0.7399571  0.8555714  0.4346667
   25     0.7523238  0.8594286  0.4390000
   50     0.7559857  0.8635714  0.4436667
  100     0.7566286  0.8630000  0.4450000
```

Based on these head-to-head results, the random forest with `mtry = 16` appears to be our winner, as its best AUC of 0.775 is better than the AUC of 0.757 for the best boosted C5.0 model.

To visualize their performance, we can use the `pROC` package to plot the ROC curves. We'll supply the `roc()` function with the observed (`obs`) values of the loan default, as well as the estimated probability of `"yes"` for the loan default. Note that these were saved by `caret` because we requested them via the `trainControl()` function. We can then `plot()` the ROC curves as follows:

```
> library(pROC)
> roc_rf <- roc(m_rf$pred$obs, m_rf$pred$yes)
> roc_c50 <- roc(m_c50$pred$obs, m_c50$pred$yes)
> plot(roc_rf, col = "red", legacy.axes = TRUE)
> plot(roc_c50, col = "blue", add = TRUE)
```

As anticipated, the resulting curves show that the random forest with an AUC of 0.775 slightly outperforms the boosted C5.0 model with an AUC of 0.757. The random forest is the outermost curve in the following R plot:

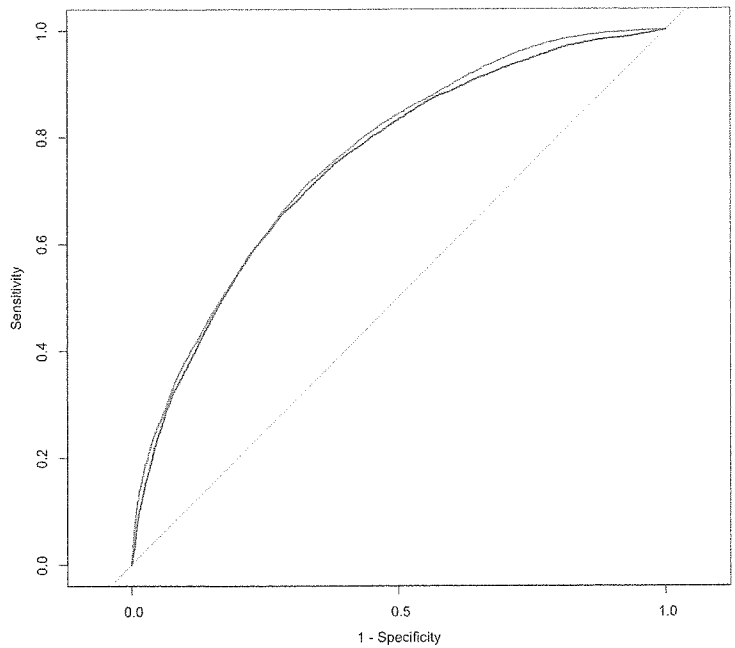


Figure 11.3: ROC curves comparing a random forest (outermost curve) to a boosted C5.0 decision tree on the loan default dataset

Based on our experiment, we would submit a random forest to the competition as our top-performing model. However, until it actually makes predictions on the competition test set, we have no way of knowing for sure whether our random forest will end up winning. Given our performance estimates, it's the safest bet of the models we evaluated, and with a bit of luck, perhaps we'll come away with the prize.

Summary

After reading this chapter, you should now know the base techniques that are used to win data mining and machine learning competitions. Automated tuning methods can assist with squeezing every bit of performance out of a single model. On the other hand, performance gains are also possible by creating groups of machine learning models that work together.

Although this chapter was designed to help you prepare competition-ready models, note that your fellow competitors have access to the same techniques. You won't be able to get away with stagnancy; therefore, continue to add proprietary methods to your bag of tricks. Perhaps you can bring unique subject-matter expertise to the table, or perhaps your strengths include an eye for detail in data preparation. In any case, practice makes perfect, so take advantage of open competitions to test, evaluate, and improve your own machine learning skillset.

In the next chapter – the last in this book – we'll take a bird's-eye look at ways to apply machine learning to some highly specialized and difficult domains using R. You'll gain the knowledge needed to apply machine learning to tasks at the cutting edge of the field, involving extremely large, challenging, or unusual datasets.