

11

Improving Model Performance

When a sports team falls short of meeting its goal – whether it is to obtain an Olympic gold medal, a league championship, or a world record time – it must begin a process of searching for improvements to avoid a similar fate in the future. Imagine that you're the coach of such a team. 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 place a greater emphasis on teamwork, which could utilize the athletes' strengths and weaknesses more smartly.

Now imagine that you're the coach tasked with finding a world champion machine learning algorithm – perhaps to enter a competition, such as those posted on the Kaggle website (<http://www.kaggle.com/competitions>), to win the million dollar Netflix Prize (<http://www.netflixprize.com/>), or simply to improve the bottom line for your business. Where do you begin?

Although the context of the competition may differ, many strategies one might use to improve a sports team's 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 in this book so far to introduce a set of techniques for improving the predictive performance of machine learners. You will learn:

- How to fine-tune the performance of machine learning models by searching for the optimal set of training conditions
- Methods for combining models into groups that use teamwork to tackle the most challenging problems
- Cutting edge techniques for getting the maximum level of performance out of machine learners


Not all of these methods will be successful on every problem. Yet if you look at the winning entries to machine learning competitions, you're likely to find at least one of them has been employed. To remain 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, you may not need 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 it may have elements of random chance, making it difficult to define the signal within the noise.

Developing models that perform extremely well on such 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 72 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 0.20, which suggested that the model was actually performing somewhat poorly. In this section, we'll revisit the credit scoring model to see if we can improve the results.

 To follow along with the examples, download the `credit.csv` file from the Packt Publishing's website and save it to your R working directory. Load the file into a data frame using the 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 trials from the default of 1 up to the values of 10 and 100, we were able to increase the model's accuracy. This process of adjusting the model fit options is called **parameter tuning**.

Parameter tuning is not limited to decision trees. For instance, we tuned k-nearest neighbor models when we searched for the best value of k , and used a number of options for neural networks and support vector machines, such as adjusting the number of nodes, hidden layers, or choosing different kernel functions. Most machine learning algorithms allow you to adjust at least one parameter, and the most sophisticated models offer a large number of ways to tweak the model fit to your liking. Although this allows the model to be tailored closely to the data, 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 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 to train 150 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 150 choices.

Automated parameter tuning requires you to consider three questions:

- What type of machine learning model (and/or specific implementation of the algorithm) 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?

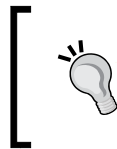
To answer the first question involves finding a well-suited match between the machine learning task and one of the 150 models. Obviously, this requires an understanding of the breadth and depth of machine learning models. This book provides the background needed for the former, while additional practice will help with the latter. Additionally, it can help to work through a process of elimination: nearly half of the models can be eliminated depending on whether the task is classification or regression; 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 result 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 each of 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.


 For a complete list of the 150 models and corresponding tuning parameters covered by `caret`, refer to the table provided by package author *Max Kuhn* at: <http://caret.r-forge.r-project.org/modelList.html>

Model	Learning task	Method name	Parameters
k-Nearest Neighbors	Classification	knn	k
Naïve 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

The goal of automatic tuning is to search a set of candidate models comprising a matrix, or **grid**, of possible combinations of parameters. Because it is impractical to search every conceivable parameter value, only a subset of possibilities is used to construct the grid. By default, `caret` searches at most three values for each of p parameters, which means that 3^p candidate models will be tested. For example, by default, the automatic tuning of k-nearest neighbors will compare $3^1 = 3$ candidate models, for instance, one each of $k=5$, $k=7$, and $k=9$. Similarly, tuning a decision tree could result in a comparison of up to 27 different candidate models, comprising the grid of $3^3 = 27$ possible 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 $3 \times 2 \times 2 = 12$.



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

The third and final step in automatic model tuning involves choosing an approach to identify the best model among the candidates. This uses the methods discussed in *Chapter 10, Evaluating Model Performance* such as the choice of resampling strategy to create 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 area under the ROC curve (AUC) can also be used if desired.

By default, when choosing the best model, `caret` will select the model with the largest value of the desired performance measure. Because this practice sometimes results in the selection of models that achieve marginal 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, it 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 learn how 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)
> 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 have 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 like `train()`, which 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, the R formula interface is used to define a tree as `default ~ .`. 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 (dependent 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 is repeatedly generating random samples of data, building decision trees, computing performance statistics, and evaluating the result.

The result of the experiment is saved in an object, which we named `m`. If you would like a peek at the object's contents, the command `str(m)` will list all the associated data—but this can be quite overwhelming. Instead, simply type the name of the object for a condensed summary of the results. For instance, typing `m` yields the following output:

1

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

2

No pre-processing
Resampling: Bootstrap (25 reps)

Summary of sample sizes: 1000, 1000, 1000, 1000, 1000, 1000, ...

3

Resampling results across tuning parameters:

model	trials	winnow	Accuracy	Kappa	Accuracy SD	Kappa SD
rules	1	FALSE	0.685	0.258	0.0256	0.0562
rules	1	TRUE	0.689	0.255	0.0268	0.057
rules	10	FALSE	0.711	0.309	0.0209	0.0459
rules	10	TRUE	0.711	0.304	0.0195	0.0448
rules	20	FALSE	0.722	0.326	0.0198	0.0451
rules	20	TRUE	0.723	0.327	0.0184	0.0371
tree	1	FALSE	0.677	0.229	0.0303	0.07
tree	1	TRUE	0.677	0.222	0.027	0.0596
tree	10	FALSE	0.722	0.288	0.0206	0.056
tree	10	TRUE	0.717	0.278	0.017	0.0436
tree	20	FALSE	0.73	0.307	0.0201	0.0562
tree	20	TRUE	0.729	0.306	0.015	0.0415

4

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

The summary includes four main components:

1. **A brief description of the input dataset:** If you are familiar with your data and have applied the `train()` function correctly, none of this information should come as a surprise.
2. **A report of preprocessing and resampling methods applied:** Here we see that 25 bootstrap samples, each including 1000 examples, were used to train the models.
3. **A list of candidate models evaluated:** In this section, we can confirm that 12 different models were tested, based on combinations of three C5.0 tuning parameters: `model`, `trials`, and `winnow`. The average and standard deviation (labeled `SD`) of the accuracy and kappa statistics for each candidate model are also shown.
4. **The choice of best model:** As noted, the model with the largest accuracy value (0.73) was chosen as the best. This was the model that used a `model = tree`, `trials = 20`, and `winnow = FALSE`.

The `train()` function uses the tuning parameters from the best model (as indicated by #4 previously) to build a model on the full input dataset, which is stored in the `m` object as `m$finalModel`. In most cases, you will not need to work directly with the `finalModel` sub-object. Instead, using the `predict()` function with the `m` object will generate predictions as expected, while also providing added functionality that will be described shortly. For example, to apply the best model to make predictions on the training data, you would use the following commands:

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

The resulting vector of predictions works just as we have done many times before:

```
> table(p, credit$default)
```

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

Of the 1000 examples used for training the final model, only two were misclassified. Keep in mind that this is the resubstitution error and should not be viewed as indicative of performance on unseen data. The bootstrap estimate of 73 percent (shown in the summary output) is a more realistic estimate of future performance.

As mentioned previously, there are additional benefits of using `predict()`, directly on `train()` objects rather than using the stored `finalModel` directly or training a new model using the optimized parameters.

First, any data preprocessing steps that the `train()` function applied to the data will be similarly applied to the data used for generating predictions. This includes transformations like centering and scaling (that is, when using k-nearest neighbors), missing value imputation, and others. This ensures that the data preparation steps used for developing the model remain in place when the model is deployed.

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

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


To obtain the estimated probabilities for each class, add an additional parameter specifying `type = "prob"`:

```
> 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), `caret` automatically translates `type = "prob"` to the appropriate string behind the scenes.

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 strongly performing models to be created easily. However, without digging deeper, you may miss out on the upper echelon of performance. Or perhaps you want to change the default evaluation criteria to something more appropriate for your learning problem. Each step in the process can be customized to your learning task.

To illustrate this flexibility, let's modify our work on the credit decision tree explained previously to mirror the process we had used in *Chapter 10, Evaluating Model Performance*. If you recall from that chapter, we had estimated the kappa statistic using 10-fold cross-validation. We'll do the same here, using kappa to optimize the boosting parameter of the decision tree (boosting the accuracy of decision trees was previously covered in *Chapter 5, Divide and Conquer – Classification Using Decision Trees and Rules*).

The `trainControl()` function is used to create a set of configuration options known as a control object, which can be used with the `train()` function. These options allow for the management 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` help command for a list of all parameters.

]

For the `trainControl()` function, the `method` parameter is used to set the resampling method, such as holdout sampling or k-fold cross-validation. The following table lists the shortened name string `caret` uses to call the method, 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	<code>p = 0.75</code> (training data proportion)
k-fold cross-validation	<code>cv</code>	<code>number = 10</code> (number of folds)
Repeated k-fold cross-validation	<code>repeatedcv</code>	<code>number = 10</code> (number of folds) <code>repeats = 10</code> (number of iterations)
Bootstrap sampling	<code>boot</code>	<code>number = 25</code> (resampling iterations)
0.632 bootstrap	<code>boot632</code>	<code>number = 25</code> (resampling iterations)
Leave-one-out cross-validation	LOOCV	None

The `trainControl()` parameter `selectionFunction` can be used to choose a function that selects 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 (that is, 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 cross-validation 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 a grid of parameters to optimize. The grid must include a column for each parameter in the desired model, prefixed by a period. Since we are using a C5.0 decision tree, this means we'll need columns with the names `.model`, `.trials`, and `.winnow`. For other models, refer to the table presented earlier in this chapter. Each row in the data frame represents a particular combination of parameter values.

Rather than creating this data frame ourselves—a difficult 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 $1 \times 8 \times 1 = 8$ 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
```

Each row will be used to generate a candidate model for evaluation, built using that 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 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:

```
> 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 as before:

```
> m

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

No pre-processing
Resampling: Cross-validation (10 fold)

summary of sample sizes: 900, 900, 900, 900, 900, 900, ...

Resampling results across tuning parameters:

  trials  Accuracy  Kappa  Accuracy SD  Kappa SD
    1      0.724    0.312  0.0255         0.059
    5      0.713    0.292  0.0211         0.0602
   10      0.719    0.295  0.0311         0.0672
   15      0.721    0.301  0.0197         0.0511
   20      0.717    0.293  0.0279         0.0791
   25      0.728    0.315  0.0322         0.0937
   30      0.729    0.31  0.0277         0.0807
   35      0.741    0.339  0.0314         0.0935

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 model = tree, trials = 1
and winnow = FALSE.
```

Although much of the output is similar to the previously tuned model, there are a few differences of note. Because 10-fold cross-validation was used, the sample size to build each candidate model was reduced to 900 rather than the 1000 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 trial. Before, the best model used `trials = 20` whereas here, the best used `trials = 1`. This seemingly odd finding 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 1-trial model offers nearly the same performance yet is a much simpler model. Not only are simple models more computationally efficient, simple models are preferable because they reduce the chance of overfitting the training data.

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. Because 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 within a wider set of **meta-learning** methods that broadly encompass any technique that involves learning how to learn. This might include anything from simple algorithms that gradually improve performance by automatically 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.

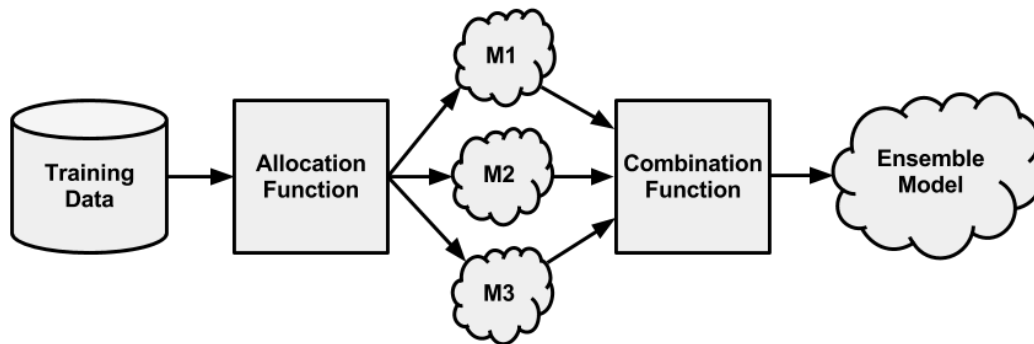
Understanding ensembles

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. For instance, a panel containing professors of literature, science, history, and art, along with a current pop-culture expert would be a safely well-rounded group. Given their breadth of knowledge, it would be unlikely to find a question that stumps the panel.

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. Using this simple principle, a large variety of algorithms has been developed distinguished largely by 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 process diagram as follows; nearly all ensemble approaches follow this pattern.



First, input training data is used to build a number of models. The **allocation function** dictates whether each model receives the full training dataset or merely a sample. Since the ideal ensemble includes a diverse set of models, the allocation function could increase diversity by artificially varying the input data to train a variety of learners. For instance, it might use bootstrap sampling to construct unique training datasets or pass on a different subset of features or examples to each model. On the other hand, if the ensemble already includes a diverse set of algorithms—such as a neural network, a decision tree, and a kNN classifier—then the allocation function might pass on the data relatively unchanged.

After the 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, when *M1* and *M2* both vote *yes* the actual class value is usually *no*, then the ensemble might ignore the votes of *M1* and *M2* and instead predict *no*. This process of using the predictions of several models to train a final arbiter model is known as **stacking**.

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:** Because 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. Additionally, it is often trivial to parallelize an ensemble using distributed computing methods. Conversely, ensembles also do well on the smallest datasets because resampling methods like bootstrapping are inherently part of many ensemble designs.
- **The ability to synthesize of data from distinct domains:** Since there is no one-size-fits-all learning algorithm—recall the No Free Lunch theorem—the ensemble's ability to incorporate evidence from multiple types of learners is increasingly important as Big Data continues to draw from disparate 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*, *Machine Learning*, Vol. 24, pp. 123-140, by L. Breiman (1996).

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 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 similar 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 comes at the expense of additional computational expense; 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)
> 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 via the `train()` function in the `caret` package. Note that the method name for the `ipred` bagged trees function is `treebag` as follows:

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



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

No pre-processing

Resampling: Cross-Validation (10 fold)

Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...

Resampling results

Accuracy	Kappa	Accuracy SD	Kappa SD
0.735	0.33	0.0344	0.0859

The kappa statistic of 0.33 for this model suggests that the bagged tree model performs on par with our best-tuned C5.0 decision tree.

To get beyond bags of decision trees, the `caret` package also provides a more general `bag()` function. It includes out-of-the-box support for a handful of models, though it can be adapted to more types with a bit of additional effort. The `bag()` function uses a control object to configure the bagging process. It requires the specification of three functions: one for fitting the model, one for making predictions, and one for aggregating the votes.

For example, suppose we wanted to create a bagged **support vector machine (SVM)** model, using the `ksvm()` function in the `kernlab` package we used in *Chapter 7, Black Box Methods – Neural Networks and Support Vector Machines*. The `bag()` function requires us to provide functionality for training the SVMs, making predictions, and counting votes.


Rather than writing these ourselves, the `caret` package's built-in `svmBag` list object supplies three functions we can use for this purpose:

```
> str(svmBag)
List of 3
 $ fit      :function (x, y, ...)
 $ pred     :function (object, x)
 $ aggregate: function (x, type = "class")
```

By looking at the `svmBag$fit` function, we see that it simply calls the `ksvm()` function from the `kernlab` package and returns the result:

```
> svmBag$fit
function (x, y, ...)
{
  library(kernlab)
  out <- ksvm(as.matrix(x), y, prob.model = is.factor(y), ...)
  out
}
<environment: namespace:caret>
```

The `pred` and `aggregate` functions for `svmBag` are also similarly straightforward. By studying these functions and creating your own in the same format, it is possible to use bagging with any machine learning algorithm you would like.

[ The `caret` package also includes example objects for bags of naive Bayes models (`nbBag`), decision trees (`ctreeBag`), and neural networks (`nnetBag`).]

Applying the three functions in the `svmBag` list, we can create a bagging control object:

```
> bagctrl <- bagControl(fit = svmBag$fit,
                        predict = svmBag$pred,
                        aggregate = svmBag$aggregate)
```

By using this with the `train()` function and the training control object (`ctrl`) defined earlier, we can evaluate the bagged SVM model as follows. Keep in mind that the `kernlab` package is required for this to work; you may need to install it if you have not done so previously.

```
> set.seed(300)
> svmbag <- train(default ~ ., data = credit, "bag",
                  trControl = ctrl, bagControl = bagctrl)

> svmbag
1000 samples
  16 predictors
  2 classes: 'no', 'yes'
```

No pre-processing

Resampling: Cross-Validation (10 fold)

Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...

Resampling results

Accuracy	Kappa	Accuracy SD	Kappa SD
0.728	0.293	0.0444	0.132

Tuning parameter 'vars' was held constant at a value of 35

Given that the kappa statistic is below 0.30, it seems that the bagged SVM model performs more poorly than the bagged decision tree model. It's worth pointing out that the standard deviation of the kappa statistic (labeled `Kappa SD`) is fairly large compared to the bagged decision tree model. This suggests that the performance varies substantially among the folds in the cross-validation. Such variation may imply that the performance could be improved further by upping the number of models in the ensemble.

Boosting

Another popular 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 *Rob Schapire* and *Yoav Freund*, who have published extensively on the topic.



For additional information on boosting, refer to: *Boosting – Foundations and Algorithms Understanding Rule Learners* by R. Schapire, and Y. Freund, (The MIT Press, 2012).

Given a number of classifiers, each with an error rate less than 50 percent; *Schapire* and *Freund* discovered that boosting will result in performance often quite better and certainly no worse than the best of these models. Essentially, this allows one to increase performance to an arbitrary threshold simply by adding more weak learners. Given the obvious utility of this finding, boosting is thought to be one of the most significant discoveries in machine learning.


Similar to bagging, boosting uses ensembles of models trained on resampled data and a vote to determine the final prediction. The key difference is that the resampled datasets in boosting are constructed specifically to generate complementary learners, and the vote is weighted based on each model's performance rather than giving each an equal vote.

A boosting algorithm called **AdaBoost**, or **adaptive boosting**, was proposed in 1997. The algorithm is based on the idea of generating weak learners 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 an alternative tree-based implementation of AdaBoost for classification. Due to its similarity to the boosted trees we created earlier, AdaBoost.M1 is not covered here.

 The AdaBoost.M1 algorithm can be found in the *adabag* R package. For more information refer to *adabag – an R package for classification with boosting and bagging*, *Journal of Statistical Software*, Vol 54(2), pp. 1-35, by E. Alfaro, M. Gamez, and N. Garcia (2013).

Random forests

Another ensemble-based method called **random forests** (or **decision tree forests**) focus 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*, *Machine Learning*, Vol. 45, pp. 5-32, by L. Breiman (2001).

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 par with nearly any other method.



Although the term "Random Forests" is trademarked by *Breiman* and *Cutler* (see <http://www.stat.berkeley.edu/~breiman/RandomForests/> for details), the term is used sometimes 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 algorithm by *Breiman* and *Cutler*.

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

Strengths	Weaknesses
<ul style="list-style-type: none"> • An all-purpose model that performs well on most problems • Can handle noisy or missing data; categorical or continuous features • Selects only the most important features • Can be used on data with an extremely large number of features or examples 	<ul style="list-style-type: none"> • Unlike a decision tree, the model is not easily interpretable • May require some work to tune the model to the data

Due to their power, versatility, and ease of use, random forests are quickly becoming one of the most popular machine learning methods. Later on 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*. An added benefit is that it is supported by `caret` for automated tuning. The syntax for training this model is as follows:

Random forest syntax
using the <code>randomForest()</code> function in the <code>randomForest</code> package
<p>Building the classifier:</p> <pre>m <- randomForest(train, class, ntree = 500, mtry = sqrt(p))</pre> <ul style="list-style-type: none">• <code>train</code> is a data frame containing training data• <code>class</code> is a factor vector with the class for each row in the training data• <code>ntree</code> is an integer specifying the number of trees to grow• <code>mtry</code> is an optional integer specifying the number of features to randomly select at each split (uses <code>sqrt(p)</code> by default, where <code>p</code> is the number of features in the data) <p>The function will return a random forest object that can be used to make predictions.</p> <p>Making predictions:</p> <pre>p <- predict(m, test, type = "response")</pre> <ul style="list-style-type: none">• <code>m</code> is a model trained by the <code>randomForest()</code> function• <code>test</code> is a data frame containing test data with the same features as the training data used to build the classifier• <code>type</code> is either <code>"response"</code>, <code>"prob"</code>, or <code>"votes"</code> and is used to indicate whether the predictions vector should contain the predicted class, the predicted probabilities, or a matrix of vote counts, respectively. <p>The function will return predictions according to the value of the <code>type</code> parameter.</p> <p>Example:</p> <pre>credit_model <- randomForest(credit_train, loan_default) credit_prediction <- predict(credit_model, credit_test)</pre>

As noted previously, 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). Whether or not these 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.

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 $\sqrt{16} = 4$ 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 (the `set.seed()` function ensures that the result can be repeated).

```
> library(randomForest)
> 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: 23.8%
```

Confusion matrix:

```
      no yes class.error
no  640  60  0.08571429
yes 178 122  0.59333333
```

As expected, the output notes that the random forest included 500 trees and tried 4 variables at each split. You might be alarmed at the seemingly poor resubstitution error according to the display confusion matrix—the error rate of 23.8 percent is far worse than any of the other ensemble methods so far. In fact, this confusion matrix is not resubstitution error at all. Instead, it reflects the **out-of-bag error rate** (labeled OOB estimate of error rate), which 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 as a way to test the model's performance on unseen data. At the end of the forest construction, the predictions for each example each time it was held out are tallied, and a vote is taken to determine the final prediction for the example. The total error rate of such predictions becomes the out-of-bag error rate.

Evaluating random forest performance

As mentioned previously, the `randomForest()` function is also 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 been working on. 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 cross-validation: 10 times 10-fold CV. While this means that the models will take a much longer time and be more computationally intensive to evaluate; since this is our final comparison, we should be *very* sure that we're making the right choice—the winner of this showdown will be our only entry into the machine learning competition.

```
> library(caret)
> ctrl <- trainControl(method = "repeatedcv",
                      number = 10, repeats = 10)
```

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 $\sqrt{16} = 4$ features. To be thorough, we'll also test values half of that, twice that, as well as the full set of 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 can supply the resulting grid to the `train()` function with the `ctrl` object as follows. We'll use the kappa metric to select the best model.

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

The preceding command may take some time to complete as it has quite a bit of work to do! When it finishes, we'll compare that to a boosted tree using 10, 20, 30, and 40 iterations:

```
> grid_c50 <- expand.grid(.model = "tree",
                        .trials = c(10, 20, 30, 40),
                        .winnow = "FALSE")
> set.seed(300)
> m_c50 <- train(default ~ ., data = credit, method = "C5.0",
               metric = "Kappa", 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	Accuracy	Kappa	Accuracy SD	Kappa SD
2	0.725	0.128	0.0169	0.0636
4	0.75	0.293	0.0299	0.0877
8	0.754	0.338	0.0311	0.0835
16	0.756	0.361	0.0338	0.0889

For the boosted C5.0 model the results are:

```
> m_c50
```

Resampling results across tuning parameters:

trials	Accuracy	Kappa	Accuracy SD	Kappa SD
10	0.732	0.322	0.0402	0.0952
20	0.734	0.327	0.0403	0.0971
30	0.738	0.334	0.0367	0.0894
40	0.739	0.334	0.0393	0.0975

With a kappa of 0.361, the random forest model with `mtry = 16` was the winner among these eight models. It was marginally higher than the best C5.0 decision tree, which had a kappa of 0.334. Based on these results, we would submit the random forest as our final model. Without actually evaluating the model on the competition data, we have no way of knowing for sure whether it will end up winning; but given our performance estimates, it's the safer bet. With a bit of luck, perhaps we'll come away with the prize.

Summary

After reading this chapter, you now know the base techniques that can be 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 keep in mind that your fellow competitors have access to the same techniques. You won't be able to get away with stagnancy; you have to keep working 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.