None of these methods will be successful for every problem. Yet, looking at the winning entries to machine learning competitions, you'll likely find that 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 the 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 it 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 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 1 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 or 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 175 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 the 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 well-suited match between the machine learning task and one of the 175 models. 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 here 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 *p* parameters. This means that at most *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 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, which 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*, such as 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 such as sensitivity, specificity, and area under the ROC curve (AUC) can also be used, if desired.

By default, `caret` will select the candidate model with the largest value of the desired performance measure. As 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, `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 you to only 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 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 (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 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 `str(m)` command 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 (note that labels have been added for clarity):

```
1    1000 samples
       16 predictor
        2 classes: 'no', 'yes'

2    No pre-processing
     Resampling: Bootstrapped (25 reps)

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

3    Resampling results across tuning parameters:

       model  winnow  trials  Accuracy   Kappa      Accuracy SD  Kappa SD
       rules  FALSE    1      0.6847204  0.2578421  0.02558775   0.05622302
       rules  FALSE   10      0.7112829  0.3094601  0.02087257   0.04585890
       rules  FALSE   20      0.7221976  0.3260145  0.01977334   0.04512083
       rules   TRUE    1      0.6888432  0.2549192  0.02683844   0.05695277
       rules   TRUE   10      0.7113716  0.3038075  0.01947701   0.04484956
       rules   TRUE   20      0.7233222  0.3266866  0.01843672   0.03714053
       tree   FALSE    1      0.6769653  0.2285102  0.03027647   0.07001131
       tree   FALSE   10      0.7222552  0.2880662  0.02061900   0.05601918
       tree   FALSE   20      0.7297858  0.3067404  0.02007556   0.05616826
       tree    TRUE    1      0.6771020  0.2219533  0.02703456   0.05955907
       tree    TRUE   10      0.7173312  0.2777136  0.01700633   0.04358591
       tree    TRUE   20      0.7285714  0.3058474  0.01497973   0.04145128

4    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 and standard deviation of the accuracy and kappa statistics for each candidate model are also shown.
4. **The choice of the best model**: As the footnote describes, the model with the largest accuracy was selected. This was the model that used a decision tree with 20 trials and the setting `winnow = FALSE`.