Decision Trees & Random Forests

Albert Mata

25/12/2018

Contents

Divide and Conquer - Classification Using Decision Trees **2**
Understanding decision trees . . . 2
Divide and conquer . . . 2
The C5.0 decision tree algorithm . . . 3
Example – identifying risky bank loans using C5.0 decision trees . . . 4
Step 1 - collecting data . . . 4
Step 2 - exploring and preparing the data . . . 5
Step 3 - training a model on the data . . . 6
Step 4 - evaluating model performance . . . 10
Step 5 - improving model performance . . . 12

Improving Model Performance **15**
Tuning stock models for better performance . . . 15
Using caret for automated parameter tuning . . . 15
Boosting . . . 19
Random forests . . . 19

References **22**

Divide and Conquer - Classification Using Decision Trees

Decision trees is a machine learning method that makes complex decisions from sets of simple choices and present its knowledge in the form of logical structures that can be understood with no statistical knowledge. This makes this model particularly useful for business strategy and process improvement.

Understanding decision trees

Decision tree learners are powerful classifiers that utilize a **tree structure** to model the relationship among the features and the potential outcomes. A tree classifier uses a structure of branching decisions, which channel examples into a final predicted class value. This structure begins at the **root node**, then has some **decision nodes** that require choices to be made and split the data across **branches** indicating potential outcomes (usually yes or no) of a decision, and when a final decision can be made the tree is terminated by **leaf nodes** (or **terminal nodes**) denoting the result of the series of decisions.

A great benefit of decision tree algorithms is that the flowchart-like tree structure can be output in a human-readable format, showing how and why the model works or doesn't work well for a particular task. This also makes decision trees particularly appropiate for applications in which the classification mechanism needs to be transparent i.e. for legal reasons. Some potential uses include:

- Credit scoring models in which the criteria that causes an applicant to be rejected need to be clearly documented and free from bias.
- Marketing studies of customer behavior such as satisfaction or churn, which will be shared with management or advertising agencies.
- Diagnosis of medical conditions based on laboratory measurements, symptoms, or the rate of disease progression.

In general, decision trees are perhaps the single most widely used machine learning technique, and can be applied to model almost any type of data, often with excellent out-of-the-box applications. However, they have some tendency to overfit data and they are not an ideal fit when the data has a large number of nominal features with many levels or a large number of numeric features, as this may result in a very large number of decisions and an overly complex tree.

Divide and conquer

Decision trees are built using a heuristic called **recursive partitioning** (or **divide and conquer**). This approach splits the data into subsets (ideally, starting with the feature most predictive of the target class), which are then split repeatedly into even smaller subsets (choosing the best candidate remaining feature each time to create new decision nodes) until the process stops when the algorithm determines the data

within the subsets are sufficiently homogeneous or another stopping criterion has been met, i.e. in a case that:

- All (or nearly all) of the examples at the node have the same class.
- There are no remaining features to distinguish among the examples.
- The tree has grown to a predefined size limit.

It's always possible to continue to divide and conquer the data by splitting it based on specific ranges for each feature until each observation resides correctly classified in its own tiny partition. But this would surely lead to some overfit. A good rule of thumb is to stop the process when more than 80 percent of the examples in each group are from a single class.

The C5.0 decision tree algorithm

C5.0 algorithm is one of the most well-known implementations of decision trees and has become the industry standard to produce decision trees, because it does well for most types of problems directly out of the box.

Strengths:

- An all-purpose classifier that does well on most problems.
- Highly automatic learning process, which can handle numeric or nominal features, as well as missing data.
- Excludes unimportant features.
- Can be used on both small and large datasets.
- Results in a model that can be interpreted without a mathematical background (for relatively small trees).
- More efficient than other complex models.

Weaknesses:

- Decision tree models are often biased toward splits on features having a large number of levels.
- It is easy to overfit or underfit the model.
- Can have trouble modeling some relationships due to reliance on axis-parallel splits.
- Small changes in the training data can result in large changes to decision logic.
- Large trees can be difficult to interpret and the decisions they make seem counterintuitive.

The first challenge that a decision tree faces is to identify which feature to split upon. The degree to which a subset of examples contains only a single class is known as **purity** (and any subset composed of only a single class is called **pure**). C5.0 uses **entropy** to measure purity in order to identify the best decision tree splitting candidate. The decision tree hopes to find splits that reduce entropy, ultimately increasing homogeneity within the groups. Entropy is typically measured in **bits** and the minimum value indicates that the sample is completely homogeneous, while the maximum value indicates that the data are as diverse as possible.

To use entropy to determine the optimal feature to split upon, the algorithm calculates the change in homogeneity that would result from a split on each possible feature, which is a measure known as **information gain** (difference between the entropy in the segment before the split and the entropy in the partitions resulting from the split). The higher the information gain, the better a feature is at creating homogeneous groups after a split on this feature. If the information gain is zero, there is no reduction in entropy for splitting on this feature. The maximum information gain is equal to the entropy prior to the split, as this would imply that the entropy after the split is zero, meaning that the split results in completely homogeneous groups.

A decision tree can continue to grow indefinitely, choosing splitting features and dividing the data until each example is perfectly classified or the algorithm runs out of features to split on. But this would result in an overfitted model. To avoid this, the process of **pruning** a decision tree involves reducing its size such that it generalizes better to unseen data:

- **Early stopping** or **pre-pruning** stops the tree from growing once it reaches a certain number of decisions or when the decision nodes contain only a small number of examples. This is very efficient, but important patterns can be missed.
- **Post-pruning** lets the tree grow and then prunes leaf nodes to reduce the size of the tree to a more appropriate level. This is oftem a more effective approach, as it's quite difficult to determine the optimal depth of a decision tree without growing it first.
- **Subtree raising** and **subtree replacement** is a post-pruning technique in which entire branches are moved further up the tree or replaced by simpler decisions.

Balancing overfitting and underfitting a decision tree is a bit of an art, but C5.0 algorithm makes it very easy to adjust the training options to see if it improves the performance on test data.

Example – identifying risky bank loans using C5.0 decision trees

Step 1 - collecting data

We will develop a simple credit approval model using C5.0 decision trees. The idea behind this is to identify factors that are predictive of higher risk of default. Therefore, we need to obtain data on a large number of past bank loans and whether the loan went into default, as well as information on the applicant. We'll use data donated by Hans Hofmann to the UCI Machine Learning Data Repository (Dheeru and Karra Taniskidou 2017).

The credit dataset includes 1,000 examples on loans, plus a set of numeric and nominal features indicating the characteristics of the loan and the loan applicant. A class variable indicates whether the loan went into default. We'll try to determine patterns that predict this outcome.

Step 2 - exploring and preparing the data

```
credit <- read.csv("credit.csv")
str(credit)
```

```
## 'data.frame':    1000 obs. of  17 variables:
##  $ checking_balance    : Factor w/ 4 levels "< 0 DM","> 200 DM",..: 1 3 4 1 1 4 4 3 4 3 ...
##  $ months_loan_duration: int  6 48 12 42 24 36 24 36 12 30 ...
##  $ credit_history      : Factor w/ 5 levels "critical","good",..: 1 2 1 2 4 2 2 2 2 1 ...
##  $ purpose             : Factor w/ 6 levels "business","car",..: 5 5 4 5 2 4 5 2 5 2 ...
##  $ amount              : int  1169 5951 2096 7882 4870 9055 2835 6948 3059 5234 ...
##  $ savings_balance     : Factor w/ 5 levels "< 100 DM","> 1000 DM",..: 5 1 1 1 1 5 4 1 2 1 ...
##  $ employment_duration : Factor w/ 5 levels "< 1 year","> 7 years",..: 2 3 4 4 3 3 2 3 4 5 ...
##  $ percent_of_income   : int  4 2 2 2 3 2 3 2 2 4 ...
##  $ years_at_residence  : int  4 2 3 4 4 4 4 2 4 2 ...
##  $ age                 : int  67 22 49 45 53 35 53 35 61 28 ...
##  $ other_credit        : Factor w/ 3 levels "bank","none",..: 2 2 2 2 2 2 2 2 2 2 ...
##  $ housing             : Factor w/ 3 levels "other","own",..: 2 2 2 1 1 1 2 3 2 2 ...
##  $ existing_loans_count: int  2 1 1 1 2 1 1 1 1 2 ...
##  $ job                 : Factor w/ 4 levels "management","skilled",..: 2 2 4 2 2 4 2 1 4 1 ...
##  $ dependents          : int  1 1 2 2 2 2 1 1 1 1 ...
##  $ phone               : Factor w/ 2 levels "no","yes": 2 1 1 1 1 2 1 2 1 1 ...
##  $ default             : Factor w/ 2 levels "no","yes": 1 2 1 1 2 1 1 1 1 2 ...
```

We see the expected 1,000 observations and 17 features, which are a combination of factor and integer data types. Let's have a look at a couple of features that seem likely to predict a default:

```
table(credit$checking_balance)
```

```
##
##     < 0 DM   > 200 DM 1 - 200 DM    unknown
##        274         63        269        394
```

```
table(credit$savings_balance)
```

```
##
##      < 100 DM     > 1000 DM  100 - 500 DM 500 - 1000 DM       unknown
##           603            48           103            63           183
```

The `default` vector indicates whether the loan applicant was unable to meet the agreed payment terms and went into default:

```
table(credit$default)
```

```
##
##  no yes
## 700 300
```

We need to partition the data into a training set with 90 percent of the examples and a testing set with the remaining 10 percent. But this time the data is not already randomly sorted, so we need to perform random sampling:

```
set.seed(123)
train_sample <- sample(1000, 900)
str(train_sample)
```

```
##  int [1:900] 288 788 409 881 937 46 525 887 548 453 ...
```

```
credit_train <- credit[train_sample, ]
credit_test  <- credit[-train_sample, ]
```

Step 3 - training a model on the data

We will use the C5.0 algorithm in the `C50` package to train our decision tree model. We can use `?C5.0Control` to display the help page for details on how to finely-tune the algorithm. For the first iteration of our credit approval model, we'll use the default C5.0 configuration:

```
library(C50)
credit_model <- C5.0(credit_train[-17], credit_train$default)
credit_model
```

```
##
## Call:
## C5.0.default(x = credit_train[-17], y = credit_train$default)
##
## Classification Tree
## Number of samples: 900
## Number of predictors: 16
##
## Tree size: 57
##
## Non-standard options: attempt to group attributes
```

That **Tree size: 57** indicates that the tree is 57 decisions deep. If we want, we can see their details (all the branches) using `summary(credit_model)`. For each branch, the numbers in parentheses indicate the

number of examples meeting the criteria for that decision and the number incorrectly classified by the decision.

```
summary(credit_model)
```

```
##
## Call:
## C5.0.default(x = credit_train[-17], y = credit_train$default)
##
##
## C5.0 [Release 2.07 GPL Edition]      Tue Dec 25 10:29:50 2018
## -------------------------------
##
## Class specified by attribute 'outcome'
##
## Read 900 cases (17 attributes) from undefined.data
##
## Decision tree:
##
## checking_balance in {> 200 DM,unknown}: no (412/50)
## checking_balance in {< 0 DM,1 - 200 DM}:
## :...credit_history in {perfect,very good}: yes (59/18)
##     credit_history in {critical,good,poor}:
##     :...months_loan_duration <= 22:
##         :...credit_history = critical: no (72/14)
##         :   credit_history = poor:
##         :   :...dependents > 1: no (5)
##         :   :   dependents <= 1:
##         :   :   :...years_at_residence <= 3: yes (4/1)
##         :   :       years_at_residence > 3: no (5/1)
##         :   credit_history = good:
##         :   :...savings_balance in {> 1000 DM,500 - 1000 DM}: no (15/1)
##         :       savings_balance = 100 - 500 DM:
##         :       :...other_credit = bank: yes (3)
##         :       :   other_credit in {none,store}: no (9/2)
##         :       savings_balance = unknown:
##         :       :...other_credit = bank: yes (1)
##         :       :   other_credit in {none,store}: no (21/8)
##         :       savings_balance = < 100 DM:
##         :       :...purpose in {business,car0,renovations}: no (8/2)
##         :           purpose = education:
##         :           :...checking_balance = < 0 DM: yes (4)
```

```
##         :           :   checking_balance = 1 - 200 DM: no (1)
##         :           purpose = car:
##         :           :...employment_duration = > 7 years: yes (5)
##         :           :   employment_duration = unemployed: no (4/1)
##         :           :   employment_duration = < 1 year:
##         :           :   :...years_at_residence <= 2: yes (5)
##         :           :   :   years_at_residence > 2: no (3/1)
##         :           :   employment_duration = 1 - 4 years:
##         :           :   :...years_at_residence <= 2: yes (2)
##         :           :   :   years_at_residence > 2: no (6/1)
##         :           :   employment_duration = 4 - 7 years:
##         :           :   :...amount <= 1680: yes (2)
##         :           :       amount > 1680: no (3)
##         :           purpose = furniture/appliances:
##         :           :...job in {management,unskilled}: no (23/3)
##         :               job = unemployed: yes (1)
##         :               job = skilled:
##         :               :...months_loan_duration > 13: [S1]
##         :                   months_loan_duration <= 13:
##         :                   :...housing in {other,own}: no (23/4)
##         :                       housing = rent:
##         :                       :...percent_of_income <= 3: yes (3)
##         :                           percent_of_income > 3: no (2)
##         months_loan_duration > 22:
##         :...savings_balance = > 1000 DM: no (2)
##             savings_balance = 500 - 1000 DM: yes (4/1)
##             savings_balance = 100 - 500 DM:
##             :...credit_history in {critical,poor}: no (14/3)
##             :   credit_history = good:
##             :   :...other_credit = bank: no (1)
##             :       other_credit in {none,store}: yes (12/2)
##             savings_balance = unknown:
##             :...checking_balance = 1 - 200 DM: no (17)
##             :   checking_balance = < 0 DM:
##             :   :...credit_history = critical: no (1)
##             :       credit_history in {good,poor}: yes (12/3)
##             savings_balance = < 100 DM:
##             :...months_loan_duration > 47: yes (21/2)
##                 months_loan_duration <= 47:
##                 :...housing = other:
##                     :...percent_of_income <= 2: no (6)
```

```
##                     :   percent_of_income > 2: yes (9/3)
##                     housing = rent:
##                     :...other_credit = bank: no (1)
##                     :   other_credit in {none,store}: yes (16/3)
##                     housing = own:
##                     :...employment_duration = > 7 years: no (13/4)
##                         employment_duration = 4 - 7 years:
##                         :...job in {management,skilled,
##                         :   :       unemployed}: yes (9/1)
##                         :   job = unskilled: no (1)
##                         employment_duration = unemployed:
##                         :...years_at_residence <= 2: yes (4)
##                         :   years_at_residence > 2: no (3)
##                         employment_duration = 1 - 4 years:
##                         :...purpose in {business,car0,education}: yes (7/1)
##                         :   purpose in {furniture/appliances,
##                         :   :           renovations}: no (7)
##                         :   purpose = car:
##                         :   :...years_at_residence <= 3: yes (3)
##                         :       years_at_residence > 3: no (3)
##                         employment_duration = < 1 year:
##                         :...years_at_residence > 3: yes (5)
##                             years_at_residence <= 3:
##                             :...other_credit = bank: no (0)
##                                 other_credit = store: yes (1)
##                                 other_credit = none:
##                                 :...checking_balance = 1 - 200 DM: no (8/2)
##                                     checking_balance = < 0 DM:
##                                     :...job in {management,skilled,
##                                         :       unemployed}: yes (2)
##                                         job = unskilled: no (3/1)
##
## SubTree [S1]
##
## employment_duration in {< 1 year,4 - 7 years}: no (4)
## employment_duration in {> 7 years,1 - 4 years,unemployed}: yes (10)
##
##
## Evaluation on training data (900 cases):
##
##      Decision Tree
```

```
##    ----------------
##    Size      Errors
##
##      56  133(14.8%)   <<
##
##
##     (a)   (b)    <-classified as
##    ----  ----
##     598    35    (a): class no
##      98   169    (b): class yes
##
##
##  Attribute usage:
##
##  100.00% checking_balance
##   54.22% credit_history
##   47.67% months_loan_duration
##   38.11% savings_balance
##   14.33% purpose
##   14.33% housing
##   12.56% employment_duration
##    9.00% job
##    8.67% other_credit
##    6.33% years_at_residence
##    2.22% percent_of_income
##    1.56% dependents
##    0.56% amount
##
##
## Time: 0.0 secs
```

The **Evaluation on training data** section indicates that the model correctly classified all but 133 of the 900 training instances for an error rate of 14.8 percent. But decision trees are known for having a tendency to overfit the model to the training data. For this reason, the error rate reported on training data may be overly optimistic and it is especially important to evaluate decision trees on a test dataset.

Step 4 - evaluating model performance

We use our decision tree to generate predictions and then compare them to the true values:

```
credit_pred <- predict(credit_model, credit_test)
library(gmodels)
CrossTable(credit_test$default, credit_pred,
           prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
           dnn = c('actual default', 'predicted default'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  100
##
##
##                 | predicted default
## actual default |        no |       yes | Row Total |
## ---------------|-----------|-----------|-----------|
##             no |        59 |         8 |        67 |
##                |     0.590 |     0.080 |           |
## ---------------|-----------|-----------|-----------|
##            yes |        19 |        14 |        33 |
##                |     0.190 |     0.140 |           |
## ---------------|-----------|-----------|-----------|
##   Column Total |        78 |        22 |       100 |
## ---------------|-----------|-----------|-----------|
##
##
```

Out of the 100 test loan application records, our model correctly predicted that 59 did not default and 14 did default, resulting in an accuracy of 73 percent and an error rate of 27 percent (instead of that 14.8 percent error rate on the training data). But the model only correctly predicted 14 of the 33 actual loan defaults (42 percent). And this type of error is a potentially very costly mistake, as it's where the bank actually loses money.

Step 5 - improving model performance

If the model had predicted "no default" for every test case, it would have been correct 67 percent of the time, which is not that far from 73 percent. So we should try to improve our model. There are a couple of simple ways to adjust the C5.0 algorithm that may help to improve the performance of our model, both overall and for the more costly type of mistakes: boosting the accuracy of decision trees, and making mistakes more costlier than others.

Boosting the accuracy of decision trees

Adaptive boosting is a process in which many decision trees are built and the trees vote on the best class for each example. Boosting is rooted in the notion that by combining a number of weak performing learners, you can create a team that is much stronger than any of the learners alone. Using a combination of several learners with complementary strengths and weaknesses can dramatically improve the accuracy of a classifier.

```
# trials = 10 is the de facto standard, as research suggests that
# this reduces error rates on test data by about 25 percent.
credit_boost10 <- C5.0(credit_train[-17], credit_train$default, trials = 10)
credit_boost10_pred <- predict(credit_boost10, credit_test)
CrossTable(credit_test$default, credit_boost10_pred,
           prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
           dnn = c('actual default', 'predicted default'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  100
##
##
##                 | predicted default
## actual default |        no |       yes | Row Total |
## ---------------|-----------|-----------|-----------|
##             no |        62 |         5 |        67 |
##                |     0.620 |     0.050 |           |
## ---------------|-----------|-----------|-----------|
##            yes |        13 |        20 |        33 |
```

```
##                |     0.130 |     0.200 |           |
## ---------------|-----------|-----------|-----------|
##   Column Total |        75 |        25 |       100 |
## ---------------|-----------|-----------|-----------|
##
##
```

So we've reduced the total error rate from 27 percent prior to boosting down to 18 percent in the boosted model. And regarding the defaults, first model predicted 14 out of 33 and boosted model predicts 20 out of 33. Still not great results, but certainly an improvement.

But then, if boosting can be added this easily, why not apply it by default to every decision tree? The reason is twofold. First, if building a decision tree once takes a great deal of computation time, building many trees may be computationally impractical. Secondly, if the training data is very noisy, then boosting might not result in an improvement at all. But if greater accuracy is needed, it's worth giving it a try.

Making mistakes more costlier than others

Giving a loan out to an applicant who is likely to default can be an expensive mistake. One solution to reduce the number of false negatives may be to reject a larger number of borderline applicants, under the assumption that the interest the bank would earn from a risky loan is far outweighed by the massive loss it would incur if the money is not paid back at all.

The C5.0 algorithm allows us to assign a penalty to different types of errors in order to discourage a tree from making more costly mistakes. These penalties are designated in a **cost matrix** which specifes how much costlier each error is, relative to any other prediction.

```
# First, we specify cost matrix dimensions and names.
matrix_dimensions <- list(c("no", "yes"), c("no", "yes"))
names(matrix_dimensions) <- c("predicted", "actual")
matrix_dimensions
```

```
## $predicted
## [1] "no"  "yes"
##
## $actual
## [1] "no"  "yes"
```

```
# Second, we assign the penalty for the various types of errors.
# No cost assigned for correct predictions.
# Cost of 4 assigned to false negatives.
# Cost of 1 assigned to false positives.
error_cost <- matrix(c(0, 1, 4, 0), nrow = 2, dimnames = matrix_dimensions)
error_cost
```

```
##          actual
## predicted no yes
##       no   0   4
##       yes  1   0
```

```
# Finally, we apply this cost matrix to our decision tree.
credit_cost <- C5.0(credit_train[-17], credit_train$default, costs = error_cost)
credit_cost_pred <- predict(credit_cost, credit_test)
CrossTable(credit_test$default, credit_cost_pred,
           prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
           dnn = c('actual default', 'predicted default'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  100
##
##
##                | predicted default
## actual default |        no |       yes | Row Total |
## ---------------|-----------|-----------|-----------|
##             no |        37 |        30 |        67 |
##                |     0.370 |     0.300 |           |
## ---------------|-----------|-----------|-----------|
##            yes |         7 |        26 |        33 |
##                |     0.070 |     0.260 |           |
## ---------------|-----------|-----------|-----------|
##   Column Total |        44 |        56 |       100 |
## ---------------|-----------|-----------|-----------|
##
##
```

This version makes more mistakes overall (37 percent error here versus 18 percent in the boosted case). But boosted model predicted 20 out of 33 defaults and this new model predicts 26 out of 33. This trade resulting in a reduction of false negatives at the expense of increasing false positives may be acceptable if our cost estimates were accurate.

Improving Model Performance

When we want to improve the predictive performance of machine learners, we consider things as:

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

Tuning stock models for better performance

Some learning problems are well-suited to the stock models presented so far. But some other problems are inherently more difficult, their underlying concepts to be learned may be extremely complex, requiring an understanding of many subtle relationships, or they may be affected by random variation, making it difficult to define the signal within the noise. Developing models that perform extremely well on these difficult problems is not straightforward.

The process of adjusting the model options to identify the best fit as we've done when we've added the `trials = 10` option when boosting the accuracy of decision trees is called **parameter tuning**. We did something similar when we tuned k-NN models searching for the best value of *k*, when we set number of nodes or hidden layers in ANN models, or when we chose different kernel functions in SVM models.

Most machine learning algorithms allow the adjustment of at least one parameter. And some offer a large number of ways to tweak the model fit. The complexity of all the possible options can be daunting, that's why we need a more systematic approach.

Using caret for automated parameter tuning

Rather than choosing arbitrary values for each of the model's parameters, it's better to conduct a search through many possible parameter values to find the best combination. The `caret` package provides tools to assist with automated parameter tuning. Its `train()` function serves as a standardized interface for over 175 different machine learning models for both classification and regression tasks and makes it possible to automate the search for optimal models using a choice of evaluation methods and metrics.

Automated parameter tuning requires us to consider three questions:

- **What type of machine learning model (and specific implementation) should be trained on the data?** The answer will depend on whether the task is classification or numeric prediction, the format of the data, the need to avoid black box models, etc.
- **Which model parameters can be adjusted, and how extensively should they be tuned to find the optimal settings?** This is largely dictated by the choice of model, as each algorithm utilizes a

unique set of parameters. The `modelLookup()` function shows information about the tuning parameters for a particular model. By default, `caret` searches at most three values for each parameter, and also lets us provide a custom search grid defined by a simple command.

```
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
```

- **What criteria should be used to evaluate the models to find the best candidate?** This uses methods 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. By default, `caret` will select the candidate model with the largest value of the desired performance measure, but alternative model selection functions are provided. In general, `caret`'s defaults are reasonable, so we can begin with them and then tweak the `train()` function to design a wide variety of experiments.

Creating a simple tuned model

The simplest way to tune a learner requires us to only specify a model type via the `method` parameter:

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

```
## C5.0
##
## 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.
```

After identifying the best model, the `train()` function uses its tuning parameters to build a model and store it in `m$finalModel`. But we can simply use `m` to make predictions:

```
head(predict(m, credit))
```

```
## [1] no  yes no  no  yes no
## Levels: no yes
```

```
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
```

Using the `train()` and `predict()` functions also offers a couple of benefits in addition to the automatic parameter tuning:

- Any data preparation steps (centering, scaling, imputation of missing values, etc.) applied by the `train()` function will be similarly applied to the data used for generating predictions, ensuring that the steps that contributed to the best model's performance will remain in place when the model is deployed.
- The `predict()` function provides a standardized interface for obtaining predicted class values and class probabilities, even for model types that ordinarily would require additional steps to obtain this information

Customizing the tuning process

It's possible to change `caret`'s default settings to something more specific to a learning task to try to achieve better performance. We use the `trainControl()` function to create a set of configuration options known as a **control object**, which guides the `train()` function. These options let us change the resampling strategy (`method`) or the measure used for choosing the best model (`selectionFunction`), among others.

For instance, we use the following command to create a control object that uses 10-fold cross-validation and the `oneSE` selection function:

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

The next step in defining our experiment is to create the grid of parameters to optimize. This grid must include a column named for each parameter in the desired model (prefixed by a period) and a row for each desired combination of parameter values (we can use `expand.grid()` to generate all combinations):

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

```
set.seed(300)
m <- train(default ~ ., data = credit, method = "C5.0",
           metric = "Kappa", trControl = ctrl, tuneGrid = grid)
m
```

```
## C5.0
##
## 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.
```

Boosting

Boosting is a common ensemble-based method that makes weak learners perform as stronger learners. It uses ensembles of models trained on resampled data and a vote to determine the final prediction. This resampled datasets are constructed specifically to generate complementary learners. Each learner's vote has a weight based on its past performance. Boosting will result in performance that is often quite better and certainly no worse than the best of the models in the ensemble.

Assuming that each classifier performs better than random chance, it's possible to increase ensemble performance simply by adding additional classifiers to the group. Though boosting principles can be applied to nearly any type of model, the principles are most commonly used with decision trees.

Random forests

Random forests (or **decision tree forests**) is another ensemble-based method that focuses only on ensembles of decision trees, combining the base principles of bootstrap aggregating (bagging) with random feature selection to add additional diversity to the decision tree models. As the ensemble uses only a small random portion of the full feature set, random forests can handle extremely large datasets. After the ensemble of trees (so, the forest) is generated, the model uses a vote to combine the trees' predictions.

Strengths:

- 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.

- Tend to be easier to use and less prone to overfitting than other ensemble-based methods.

Weaknesses:

- Unlike a decision tree, the model is not easily interpretable.
- May require some work to tune the model to the data.

We can use the `randomForest()` function in the `randomForest` package to create an ensemble of 500 trees that consider `sqrt(total_number_of_features)` random features at each split (these are values by default that can be changed). Limiting the number of random features allows for substantial random variation to occur from tree to tree.

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

In this case, the **out-of-bag error rate** is pretty accurate compared to what we will find when using this model with unseen data.

As a final exercise, let's compare an auto-tuned random forest to an auto-tuned boosted C5.0 model:

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

```
# Auto-tuned random forest.
set.seed(300)
grid_rf <- expand.grid(.mtry = c(2, 4, 8, 16))
m_rf <- train(default ~ ., data = credit, method = "rf",
              metric = "Kappa", trControl = ctrl, tuneGrid = grid_rf)
m_rf
```

```
## Random Forest
##
## 1000 samples
```

```
##   16 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 10 times)
## Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy  Kappa
##    2    0.7223    0.1157648
##    4    0.7469    0.2830979
##    8    0.7504    0.3287980
##   16    0.7568    0.3638677
##
## Kappa was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 16.
```

```
# Auto-tuned boosted C5.0 model.
set.seed(300)
grid_c50 <- expand.grid(.model = "tree",
                        .trials = c(10, 20, 30, 40),
                        .winnow = "FALSE")
m_c50 <- train(default ~ ., data = credit, method = "C5.0",
               metric = "Kappa", trControl = ctrl, tuneGrid = grid_c50)
m_c50
```

```
## C5.0
##
## 1000 samples
##   16 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 10 times)
## Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
## Resampling results across tuning parameters:
##
##   trials  Accuracy  Kappa
##   10      0.7293    0.3083621
##   20      0.7362    0.3285118
##   30      0.7349    0.3229257
##   40      0.7368    0.3251888
```

```
##
## 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 largest value.
## The final values used for the model were trials = 20, model = tree
##  and winnow = FALSE.
```

The random forest model with `mtry = 16` has accuracy = 0.76 and Kappa = 0.36. It's a bit better than the best C5.0 decision tree (accuracy = 0.74 and Kappa = 0.33).

References

Dheeru, Dua, and Efi Karra Taniskidou. 2017. "UCI Machine Learning Repository." University of California, Irvine, School of Information; Computer Sciences. http://archive.ics.uci.edu/ml.

Lantz, Brett. 2015. *Machine Learning with R*. Packt Publishing Ltd.