R FOR ACTUARIES

and Data Scientists with Applications to Insurance

BRIAN A. FANNIN, ACAS, CSPA

[
Reserved for
digital
supplement
access
key code.
]

Scratch off to reveal digital
supplement access key code.

ACTEX Learning

9 781647 563165

BRIAN A. FANNIN, ACAS, CSPA

R FOR ACTUARIES AND DATA SCIENTISTS
WITH APPLICATIONS TO INSURANCE

Table of Contents

III Heterogeneous Data

V   Practical considerations

2

Probability functions

We are going to dive into the deep end and start simulating random variables. We will brush over a few important topics that will be covered in depth later. For now, if you do not understand something, don't worry. Just type in the code and focus on *what* we are doing, and we will get to *how* to do it. As you work through the examples, be sure to type them into a script which you will save. Remember that you can execute each line by pressing CTRL+ENTER in RStudio. Watch the changes which take place in the Environment tab.

We will start by simulating a block of claims. First, we will establish a placeholder for the average claim size and the number of claims we will simulate. We will tackle random frequency later.

```
average_claim_size <- 5e3
num_claims <- 10e3
```

In the statements above, I am using something which looks like an arrow, placed to the right of some text. The <- symbol will *assign* a *value* to an *object*. Let's talk about what each of those words means:

- assign: When we *assign* something, we are creating space in your computer's memory. This space may contain more or less anything and it will change when you tell it to change.
- value: The *value* is what is being stored in the memory that you created during *assignment*. As noted above, the *value* may change. It only makes sense to talk about the *value* at a specific point in time.
- object: An *object* may be thought of as a container for a value. This container has a name and the name is quite important. The name allows us to reason about objects as something other than space in a computer's memory. Using a name like average_claim_size anchors it to a real-world concept. This allows us to think about the object without worrying about its value, similar to the way that symbols in algebra allow us to think about equations without needing to worry about the underlying numbers.

> `100 PRINT`
> `110 GOTO 100`
>
> What is meaningful about assignment in R, versus other languages is all of the code that we *did not* type. In other languages, we would have had to specify what kind of information the object would contain and possibly what parts of the program could access the object's value. In VBA, that would look something like:
> ```
> Public average_claim_size as Double
> average_claim_size = 5000
> ```

Financial values are often quite large and stated in thousands. Scientific notation will help us quickly type out values in a way that makes them easy to read. See if you can detect the difference between the code below and the lines above.

```
average_claim_size <- 5000
num_claims <- 1000
```

I will strongly recommend that you only use scientific notation in increments of 3. This is consistent with how we tend to talk about numbers, i.e. "ten *million*", "one hundred *thousand*", "six *billion*", etc.

Let's simulate some claims. The exponential distribution only has one parameter, so it's very easy to use. Type and execute the line below.

```
claims <- rexp(n = num_claims, rate = 1 / average_claim_size)
```

We are once again creating an *object*. This time our object is called `claims` and we assigning it a *value*. This time, we did not type out the value by hand. Instead, we invoked a *function* called `rexp()`. The name "rexp" is short for "random exponential". You are undoubtedly familiar with functions, either through programming in other languages, or from entering them in a spreadsheet. In R, a function may be recognized whenever we see some text, followed by a left parentheses, a comma-separated list of text and then a closing parentheses. The comma-separated list will be referred to as *arguments* or *parameters*. Not every function will have parameters. We will talk more about functions in Chapter 4.

In the line above, each argument has some text, followed by an equal sign and then more text. The text to the left of the equal sign is the argument's *name* and the text to the right is the argument's *value*. Most arguments to a function are *named*, to make it easy to understand their purpose[1]. R permits us to be very explicit about the names of a function's arguments, which means that we can invoke the function with the arguments in a different order, as follows:

---

[1]We will see an exception to this in Chapter 4.

```
claims <- rexp(rate = 1 / average_claim_size, n = num_claims)
```

You may have noticed some changes in the value of the `claims` object displayed in the Environment tab. This is because `rexp()` is performing a random simulation, so the value returned will change each time that we run the function, even if all of the inputs are the same. If we don't want this to happen, we can insist on a specific starting point for the chain of simulations, by using the function `set.seed()`. Whenever we want to reproduce random results, we simply call `set.seed()` again with the same argument, which we will call the *random number seed*, or simply *seed*.

```
set.seed(1234)
claims <- rexp(rate = 1 / average_claim_size, n = num_claims)
set.seed(1234)
claims <- rexp(n = num_claims, rate = 1 / average_claim_size)
```

This time, you should not have seen any changes to the `claims` object. In fact, you should get **exactly the same** numbers that I do! We have begun to create a program which is **reproducibly random** and this is a very good thing. This means that I can conduct research based on random simulations and someone else can check my work. We are guaranteed to get the same values even when using simulation.

Back to function arguments. Every argument has a name, but we are not required to use them. If I do not supply argument names, then R will assume that they have been passed in the same order as is defined in the function. If you want to know what that definition is, simply lookup the help for that function as described in Chapter 1. You can also glean some information by using the function `formals()`. This is a function which uses a function as an argument, i.e. `formals(rexp)`. Go ahead and give it a try. It exists, but I rarely use it.

When you look up the help, you will also notice that some arguments show a value assigned in the "Usage" section of the help file, as shown in Figure 2.1. This is a default argument, which means that if I like a value of 1, I am not required to pass in any value at all.

```
claims <- rexp(n = num_claims)
```

That's not what I want in this case. However, let's go ahead and call the function with the arguments unnamed, in their expected order, and not assuming any defaults. We will also set the seed so that your results will look the same as mine.
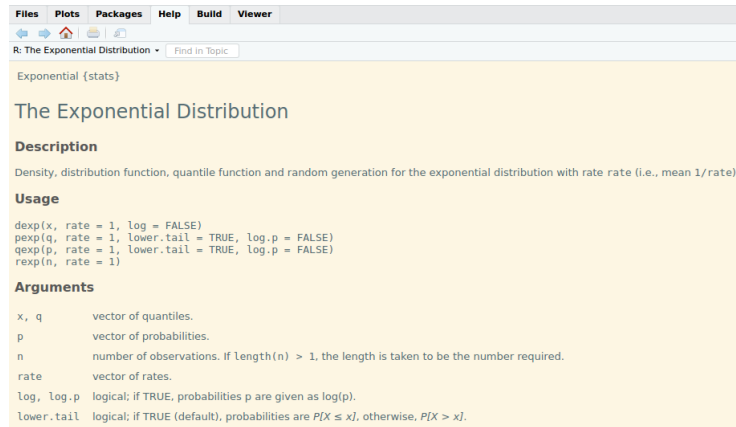
Figure 2.1: The help page for rexp

```
set.seed(1234)
claims <- rexp(num_claims, 1 / average_claim_size)
```

We now have ten thousand simulations of an exponential distribution. Do you want one million? That only takes three lines of code (or two if you don't reset the seed).

```
num_claims <- 1e6
set.seed(1234)
claims <- rexp(num_claims, 1 / average_claim_size)
```

A quick aside to talk about the parameterization. I am accustomed to thinking of the exponential parameter as being equal to the mean. R does not define it this way. Happily, the help file will make this clear, but you want to make sure that your understanding of a function definition is aligned with R. In this case, we can make an easy transformation.

But how do we know that this simulated data represents what we want? We will start with two quick checks.

```
summary(claims)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>       0    1440    3468    5001    6943   71016
```

The summary() function will return the min, max, median, mean and 1st and 3rd quartiles of a set of values. Quickly note that summary() is another function which takes a set of values as an argument. Also, rather than returning just one value, it

returns several. The mean 5001.318 is very close to our value of 5000, so the random sample appears to be sensible.

We may also like to check things visually. `hist()` is a function which won't return any value at all. Instead, it will create a plot, which displays in the Plots tab.

```
hist(claims)
```

Histogram of claims
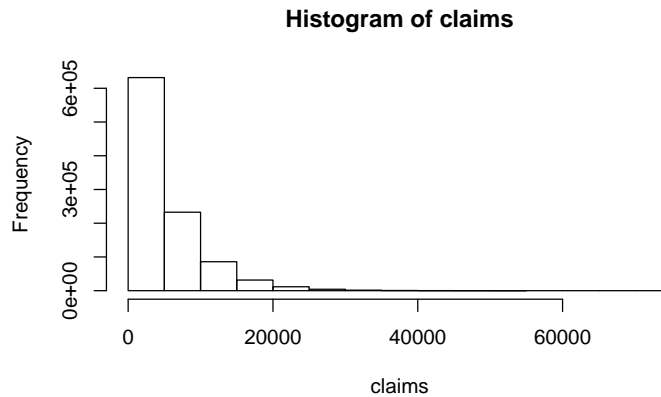
The picture is not publication quality, but it tells us all that we need to know for now. The data definitely seem exponentially distributed.

We can get a slightly better look at the results by applying a log transform before creating the histogram.

```
hist(log(claims))
```

Histogram of log(claims)
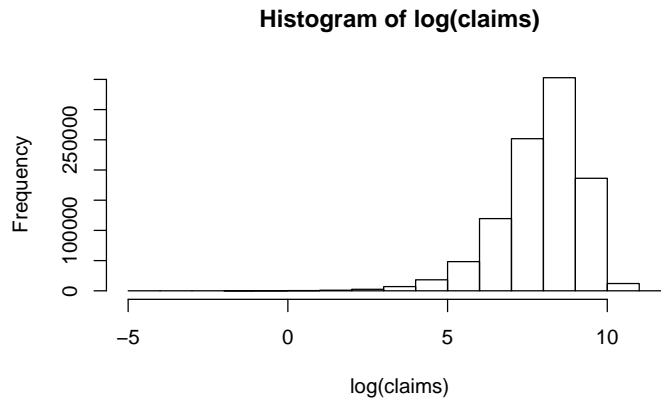
2.1   Exploring the exponential distribution

You have just finished your first stochastic simulation in R. It may have been fairly simple, but complicated is not that far away. Before we go any further, let's revisit the help page for `rexp()`. You will have noticed that there are several other functions listed there. These are `dexp()`, `pexp()` and `qexp()`. All of them will provide values related to the exponential distribution. Let's talk about `dexp()` first. Our exploration of this function will lead us to creating a function of our own.

`dexp()` will generate values from the density function. Recall that this is $f(x) = rate * e^{-rate*x}$. I prefer to think in terms of θ, which would correspond to the value we have given to average claim size. That density function is $f(x) = \frac{e^{x/\theta}}{\theta}$. For a continuous distribution, individual values of the density function are not very interesting. Here are results for one, five and ten thousand.

```
dexp(1e3, 1 / average_claim_size)
#> [1] 0.000164
dexp(5e3, 1 / average_claim_size)
#> [1] 7.36e-05
dexp(10e3, 1 / average_claim_size)
#> [1] 2.71e-05
```

As we can see, everything is less than .0002. Note that calling the function three times feels cumbersome. We had to type a lot of the same information over again, but the only thing which changed was the *x* value for the function. Just as the `summary()` function accepts more than value as an input, we may also pass more than one *x* value for `dexp()`. We do this by wrapping the three values in a very useful function called, simply, `c()`. This is short for *combine*.

```
dexp(c(1e3, 5e3, 10e3), 1 / average_claim_size)
#> [1] 1.64e-04 7.36e-05 2.71e-05
```

We can observe two things here. First, multiple function arguments produce multiple function outputs. Second, we can immediately pass the results from one function as an input to another function. `c()` was evaluated and its results were passed into `dexp()`. This is probably something you've done often when coding a function for a spreadsheet. Both of these properties will be very useful to us throughout this text.

Even though the individual results for `dexp()` are not that interesting, it would be great to use the density function to draw a picture, just as we did with the call to `hist()`. There are several strategies for this. For now, we will take a very straightforward one, in

which we create multiple inputs — as we just did in the last code block — and then use them to generate multiple outputs. Finally, we will create a simple X-Y scatter plots of those two sets of values.

The first step will require us to think a bit about how to construct the values for the x-axis. Have another look at the histogram we created earlier. Note that the axis marks do not go further than three thousand, so we will use this as our upper bound. We also know that the exponential distribution has a lower bound at x = 0. An easy way to get an evenly spaced set of values between those bounds is the seq() function. Here seq() is short for *sequence*. We will pass in the lower and upper bounds as named arguments, and a value of 5 thousand in the length.out parameter as the number of x values we would like to have returned.

```
x_values <- seq(from = 0, to = 3e3, length.out = 5e3)
```

Calculation of the y-values is similar to the call we made earlier when we passed in three different values for x. Now, instead of using the c() function, we can simply pass in the x_values *object* which we created.

```
y_values <- dexp(x_values, rate = 1 / average_claim_size)
```

If you look in the Environment tab, you can see that the objects x_values and y_values appear and each has 5 thousand values. (We can infer this by noting the numbers in brackets, [1:5000].) We are now ready to create a picture of the distribution. Whereas earlier we used the hist() function, now we will use the plot() function. As its name would suggest, this is a fairly generic function. Its default behavior is to construct a scatter plot based on two sets of values.

```
plot(x_values, y_values)
```
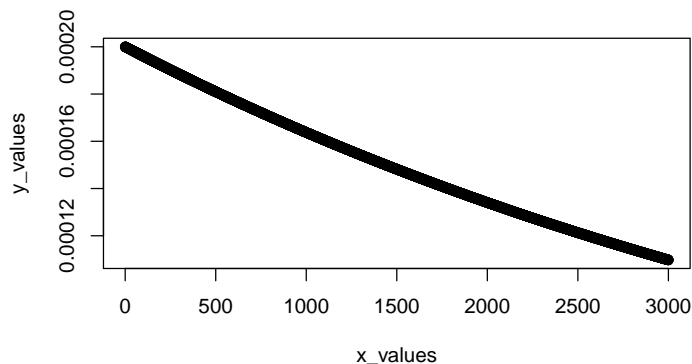
This looks quite a lot like a straight line and not as much like an exponential distribution as I would like to see. Perhaps this is because we have set the upper bound too low? We can quickly generate a new plot with a wider range of x values. We will simply repeat the three steps we carried out earlier, but with a different upper bound for the x values. To save a bit of typing, I will not repeat the names of the function arguments. Remember that I can do this, so long as I'm passing the arguments in in the proper order.

```r
x_values <- seq(0, 10e3, length.out = 5e3)
y_values <- dexp(x_values, 1 / average_claim_size)
plot(x_values, y_values)
```
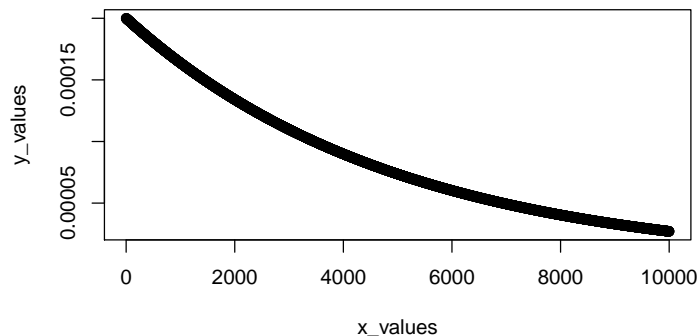

2.2   Writing your first function

That looks a little bit better. Notice that we had to repeat the same line of code to recreate the y_values. If you are familiar with the way that spreadsheets automatically recalculate, this may seem cumbersome. However, this is wholly consistent with the way that R works. A *value* is *assigned* to an *object*. Objects do not take **references** to other objects as they do in a spreadsheet. This is OK. If multiple objects need to be updated at the same time, we simply have to ensure that we do so. One of the most fundamental ways that we do this is by creating *functions*.

We have already seen some functions, but now we would like to create one of our own. A function is like a recipe, where we provide a set of ingredients and the function will follow the recipe to produce a result. The outcome may be different, depending on the ingredients, but the steps that are followed will be the same every time.

```r
plot_exponential <- function(mean, upper_x){
  x_values <- seq(0, upper_x, length.out = 500)
  y_values <- dexp(x_values, 1 / mean)
```

```
    plot(x_values, y_values)
}
```

Notice that we are using the same $<-$ symbol to *assign* a *value* to the *object* named `plot_exponential`[2]. This is because a function is also an object. For clarity, in this text, I will refer to functions by their name followed by parentheses, like `plot()` or `seq()`. The parentheses serve as a reminder that the object is a function. This is one of the only times where I will not do that. This is to emphasize the idea that a function is an *object* in the same way that things like `average_claim_size`, `x_values` and others are objects.

Even though it is an *object*, the *value* of a function is different from the objects we have encountered so far. The value for a function is an *expression*, which is basically a set of instructions. In this case, the instructions are simply the three lines that we executed earlier to display a probability density plot.

Let's see this in action. I will use an upper bound that is higher than one we have used so far. You should experiment with various values for the upper bound, including the ones we have already used. If the plots do not look the same, there is probably an error in your code which needs to be fixed.

```
plot_exponential(average_claim_size, 50e3)
```
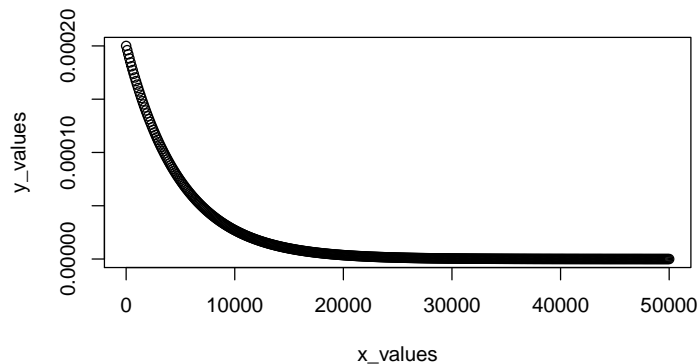

Defining a function means that we no longer have to worry about whether we have updated the y values. They will be changed based on the arguments that we provide to the function. As an additional benefit, I am not limited to the average claim size that we started with. Look again at our function definition and notice that we named the first parameter "mean", rather than "average_claim_size". This was deliberate. The

---

[2]Calling our function with an upper bound of 50 thousand gives a picture that reminds me of the exponential distributions I've seen in textbooks.

exponential distribution can model many different things; "wait time" should be an item
which actuaries think of. We can easily call our function with arguments as shown below.

```r
plot_exponential(10, 50)
```
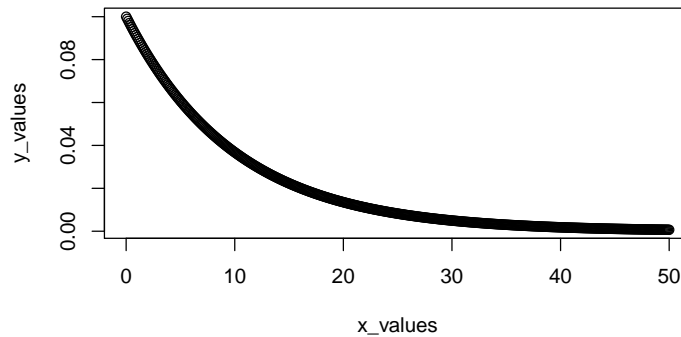

Spend a bit of time calling this function with various arguments and observing the
results. R is meant to be a great, big sandbox for you to play in. Start getting your
hands dirty!

2.3   More probability functions

Having gotten a deeper feel for the density function of the exponential distribution, let's
return to the other functions that we saw listed on the help page. First is the `pexp()`
function which will give us the cumulative distribution. This has some immediate
applications. In our case, we know that the average claim size is five thousand. If claims
follow an exponential distribution, what is the probability that a claim will be more than
twice its mean? With `pexp()` this is easy to calculate. The cumulative distribution will
give us the probability that a value is less than or equal to X, so the probability that a
value is higher than X is simply one minus that value. Or . . .

```r
1 - pexp(2 * average_claim_size, 1 / average_claim_size)
#> [1] 0.135
```

That's just under 14%. If I want to know probabilities that a claim is greater than one,
two or three times the mean, I can use the `c()` function as before.

```r
1 - pexp(c(1, 2, 3) * average_claim_size, 1 / average_claim_size)
#> [1] 0.3679 0.1353 0.0498
```

Again, the output of `c()` is immediately available as an input to `pexp()`. Note that whereas `c(1, 2, 3)` generates 3 values, multiplication by the scalar `average_claim_size` will also generate 3 values. R will automatically replicate the value for `average_claim_size` so that there are 3 values to multiply against the set of integers 1, 2, and 3. We will have more to say about this in Chapter 5.

The method of generating integers feels like it should be easier. Surely there will be many times when I simply want a set of integers. It was easy when I only needed three, but what if I had needed three hundred? Earlier, we saw that the `seq()` function could accomplish that. Should we use it for a sequence of integers?

The answer is yes and no. We can use the `:` operator to generate a sequence of integers. The `seq()` function could do this as well, but `:` is more common. It will simply produce integers between its first and second values.

```
1 - pexp(1:3 * average_claim_size, 1 / average_claim_size)
#> [1] 0.3679 0.1353 0.0498
```

There is about a five percent chance that any given claim will be three times the mean. That feels significant. Should we buy per occurrence excess reinsurance? The broker is on the phone and wants to know what we think of a treaty which attaches at 25 thousand for a limit of 50 thousand. How much loss would we cede under a treaty like that?

```
pexp(c(25e3, 75e3), 1 / average_claim_size)
#> [1] 0.993 1.000
```

This would cover less than 1% of our losses. Granted, there is some risk management value in doing this, but we'd like to offer a different structure to the broker. What if we wanted to cede all losses between the 95th and 99th percentile? What would be the attachment and limit of such a treaty? To answer that, we will need the `qexp()` function.

```
qexp(c(.95, .99), 1 / average_claim_size)
#> [1] 14979 23026
```

No actuary will give numbers like that to a broker. To make them easier to communicate, let's round them off to the nearest thousand. The `round()` function will handle this for us. We will just need to store the results in an interim object first. Observe how using a negative value for the `digits` argument will round to the *left* of the decimal point. Passing in a value of negative three will round to the nearest thousand.

```
treaty_limits <- qexp(c(.95, .99), 1 / average_claim_size)
round(treaty_limits, digits = -3)
#> [1] 15000 23000
```

The size of the layer may be a little unorthodox, but we can tweak that if we like. The important point is that we can easily get answers anywhere on the distribution function.

At this point, you have mastered four functions which will generate meaningful values for the exponential distribution. They are:

- `rexp()` — This will simulate values which come from an exponential distribution
- `dexp()` — This will calculate values of the density function for an exponential distribution
- `pexp()` — This will give us the cumulative distribution
- `qexp()` — The inverse of `dexp()`. Input a cumulative probability and get back a value for X

Four functions is a very great start, but I have some good news for you. You now know more than four functions. You actually know more than a hundred. **EVERY** probability function in R will follow this naming convention. Some of the parameters will obviously be different, but 'r', 'p', 'd' and 'q' are all the same. Let's look at a few.

The Poisson is another function with only one parameter. For a Poisson with $\lambda = 5$, what is the probability of observing a value equal to 2 or greater than 6? What value lies at the 95th percentile? We can answer all three questions in just 3 lines of code.

```
dpois(2, 5)
#> [1] 0.0842
ppois(6, 5)
#> [1] 0.762
qpois(0.95, 5)
#> [1] 9
```

This works just fine, but I'm bothered by typing in 5 more than once. It increases the chance of operational risk and it's less expressive. I've been using R for years, but I'd have to look at the help to remember whether lambda is the first or second parameter. The code below is preferred.

```
lambda <- 5
dpois(2, lambda)
#> [1] 0.0842
```

```r
ppois(6, lambda)
#> [1] 0.762
qpois(0.95, lambda)
#> [1] 9
```

The random number generator works the same way it did for the exponential distribution. We will again nest the functions to generate a histogram.

```r
hist(rpois(5e3, lambda))
```

Histogram of rpois(5000, lambda)
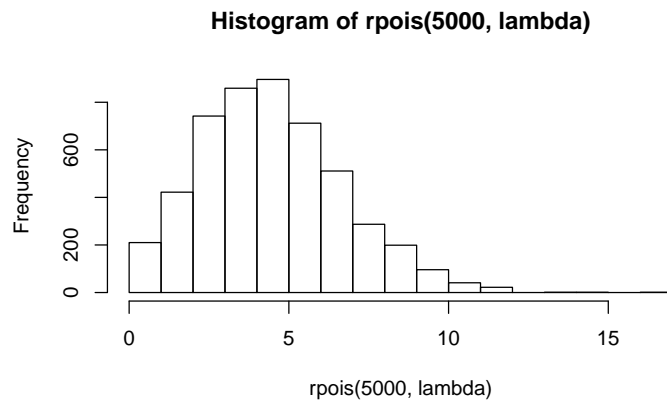
Function nesting is a common practice, used in many programming languages and spreadsheets. That's why I'm using it in this introductory chapter, but in Chapter 4, I will switch to the pipe operator, written as %>%. This is my preferred approach, but either will work.

The probability functions included in the `stats` package (which automatically loads with every R session) are given in Table 2.1. In the tables, note that the asterisk indicates a placeholder for one of the four probability functions:

1. `r` - Random number generation
2. `d` - Probability density or probability mass function
3. `p` - Cumulative distribution
4. `q` - Inverse cumulative distribution

Table 2.1: Probability functions in R

Distribution	Abbreviation
beta	`*beta()`
binomial	`*binom()`
Cauchy	`*cauchy()`
chi-squared	`*chisq()`
exponential	`*vexp()`
Fisher F	`*f()`
gamma	`*gamma()`
geometric	`*geom()`
hypergeometric	`*hyper()`
logistic	`*logis()`
lognormal	`*lnorm()`
negative binomial	`*nbinom()`
normal	`*norm()`
Poisson	`*pois()`
Student t	`*t()`
uniform	`*unif()`
Weibull	`*weibull()`

The `actuar` package extends this list with the functions shown in Table 2.2.

Table 2.2: More probability functions from actuar

Distribution	Abbreviation
transformed beta	*trbeta()
Burr	*burr()
loglogistic	*llogis()
paralogistic	*paralogis()
generalized Pareto	*genpareto()
Pareto	*pareto()
inverse Burr	*invburr()
inverse Pareto	*invpareto()
inverse paralogistic	*invparalogis()
transformed gamma	*trgamma()
inverse transformed gamma	*invtrgamma()
inverse gamma	*invgamma()
inverse Weibull	*invweibull()
inverse exponential	*invexp()
loggamma	*lgamma()
gumbel	*gumbel()
inverse Gaussian	*invgauss()
single parameter Pareto	*pareto1()
generalized beta	*genbeta()

The Tweedie distribution

In addition to the army of probability functions we have available to us in Tables 2.1 and 2.2, there is an additional distribution, which we will use. The Tweedie [Tweedie, 1984] has found increasing use in actuarial applications, as seen in [Taylor, 2009] and [Frees et al., 2011]. A member of the exponential dispersion family, it is closely related to generalized linear models which we will explore in Chapter 15. The function may also be considered a compound Poisson-Gamma process, wherein we observe N Poisson distributed variables, each of which follows a gamma distribution. This is a classic collective risk model, in which the claim frequency and claim severity of a policyholder or portfolio are modeled separately.

Following [Meyers, 2009], we may express the parameters of the Tweedie as follows. The expected claim frequency is given by λ, the total amount of expected loss is given by μ. Recall that the expected value of a gamma is equal to $\alpha * \theta$. This is the expected value of a single claim.

$$p = \frac{\alpha + 2}{\alpha + 1} \tag{2.1}$$

$$\mu = \lambda * \alpha * \theta \tag{2.2}$$

$$\phi = \frac{\lambda^{1-p} * (\alpha * \theta)^{2-p}}{2 - p} \tag{2.3}$$

With these equations, we can easily calculate the parameters for a Tweedie. In the code below, think of the claim amounts being expressed in thousands of dollars.

```
lambda <- 2
alpha <- 50
theta <- 0.2

p <- (alpha + 2) / (alpha + 1)
mu <- lambda * alpha * theta
phi <- lambda ^ (1 - p) * (alpha * theta) ^ (2 - p)
phi <- phi / (2 - p)
```

Notice that we break the calculation of `phi` into two steps. There are a lot of parentheses going on there and it's easy to something out of order. Breaking the logic into smaller steps makes the code easier to read and debug.

To use the Tweedie distribution, you will need the `tweedie` package. We will load this in and simulate some values. Notice the apparent redundancy of associating the `mu` parameter of the function with the `mu` object we just created. When processing a fraction, R will handle these two labels separately and correctly.

```r
library(tweedie)

set.seed(1234)
tweeds <- rtweedie(1e3, mu = mu, phi = phi, power = p)
summary(tweeds)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>     0.0     9.7    19.3    20.4    29.5    77.8
```

When creating the histogram, I will break the bins using a sequence of integers from 0 through 80 using the `breaks` parameter. I know the exact sequence to use in this case based on the "Max." output from the call to `summary()`.

```r
hist(tweeds, breaks = 0:78)
```

Histogram of tweeds
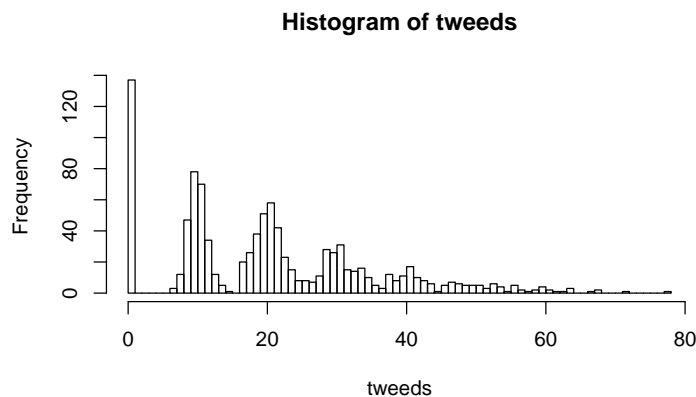
Figure 2.2: Simulated values from the Tweedie distribution

In Figure 2.2, notice the probability mass at zero. Also, notice the spikes near multiples of 10. The expected claim amount is equal to $\alpha * \theta$. For our example, this is 10. So, the first cluster represents those policyholders who experience only one claim, each of which will have a value around 10. For policyholders with two claims, the expected value will be 20, but the variability around that amount will increase.

2.4   A more complicated simulation

When we started, we used a fixed value for the number of claims. Now, let's fix the number of policyholders and generate a random number of claims for each of them. We will use the Poisson distribution for this. Again, we will reset the random number seed to ensure that the results you see are the same as the ones I see.

```r
num_policies <- 1e3
set.seed(1234)
num_claims <- rpois(num_policies, 2)
```

The `sample()` function will generate a random sample of discrete items. Below, we randomize the names of the months. If we would like more than 12 values, we may increase the `size` parameter. When we do this, we will need to set the `replace` parameter to TRUE in order to sample with replacement.

```r
set.seed(1234)
sample(month.name)
#>  [1] "December"  "October"        "June"         "May"     "April"
#>  [6]     "July"  "January"  "September"  "February"   "August"
#> [11] "November"      "March"
sample(month.name, size = 20, replace = TRUE)
#>  [1]   "October"     "June"     "April" "August"     "April" "April"
#>  [7]       "May" "August"     "April" "August"     "March" "April"
#> [13]   "October"       "May" "February" "August" "November" "April"
#> [19] "December"     "March"
```

To create a weighted sample, we may alter the `prob` argument. Below, we will flip a biased coin, where "tails" are twice as likely as heads. Notice that I don't have to normalize the probability amounts. We may simply enter relative sizes.

```r
sample(c('Heads', 'Tails'), size = 20, prob = c(1, 2),
  replace = TRUE)
#>  [1] "Heads" "Tails" "Tails" "Heads" "Tails" "Tails" "Heads" "Tails"
#>  [9] "Tails" "Tails" "Tails" "Heads" "Tails" "Heads" "Heads" "Tails"
#> [17] "Tails" "Tails" "Tails" "Heads"
```

In future chapters we will extend this simulation to generate data which shares characteristics of insurance claims. It will include information on policy effective and expiration dates and premium. You already have most of the tools you will need to do this. However, there are some foundational language and data elements that we will need first. That's coming up next.

Chapter 2 Exercises

1. Change our function `plot_exponental()` so that it does not take a fixed upper bound as an argument. Instead, have the upper bound be a function of:

   - the mean
   - the distribution function

2. Given a portfolio of 10,000 policies, each of which has a probability of 0.98 of going loss free:

   - What is the probability that 200 or more policies will have at least one claim?
   - What is the probability that *exactly* 200 will have at least one claim?
   - What is the probability that only 10 policies will experience a loss?
   - Create 10,000 simulated policy periods. What is the maximum and minimum number of policies to experience a claim? What is the median of the simulated output?

3. Carry on with the assumptions from above. Additionally assume that at the end of the policy period, 400 policyholders have experienced a claim.

   - What is the probability of that happening?
   - If the loss-free probability were 0.97 what is the probability of that happening?
   - Create a vector which shows the probability of the event that 250 policyholders experience a claim. Assume loss-free probabilities in the range of 0.90 to 0.99 in increments of 0.005.

4. Assume a lognormal distribution with a mean of 10,000 and a CV of 30%.

   - For that distribution, what is the probability of seeing a claim greater than $10,000? greater than 50,000? greater than 100,000?
   - Generate two samples from that distribution. The first should have 100 elements and the second should have 1,000.
   - What are the mean, standard deviation and CV of each sample?

5. Repeat the exercise above for a gamma distribution.
6. Simulate one hundred values from a Tweedie distribution with α and θ parameters of 50 and 0.2 respectively. For λ, use values of 2, 5, 10, 50 and 100. Plot a histogram of the results.
7. Create a function to plot the results for the previous question, as was done for the exponential distribution.

17

Tree-based models

The models we have looked at so far will perform very well when there is a linear relationship between a response and a set of predictors. They can even go beyond that. By transforming the response and/or the predictors, we can capture a wide range of scenarios. However, there is one kind of scenario which they do not handle very well. Run the code in the block below and then have a look at Figure 17.1.

```
library(tidyverse)
library(magrittr)

sims <- 250
set.seed(1234)
tbl_tree <- tibble(
    x_1 = runif(sims, 0, 10)
  , x_2 = runif(sims, 0, 10)
  , y = case_when(
      x_1 < 5 & x_2 < 4 ~ rnorm(sims, 7, 4)
    , x_1 < 5 ~ rnorm(sims, -8, 4)
    , x_1 >= 5 & x_2 < 2 ~ rnorm(sims, 50, 10)
    , TRUE ~ rnorm(sims)
  )
)
```

17.1   Continuous response

Figure 17.1 shows observations plotted in the two-dimensional predictor space with each point showing variation in y by shape and color. We can see that the lower right corner has large values of y and the upper left corner has smaller ones.

```
tbl_tree %>%
  ggplot(aes(x_1, x_2)) +
  geom_point(aes(size = y, color = y))
```
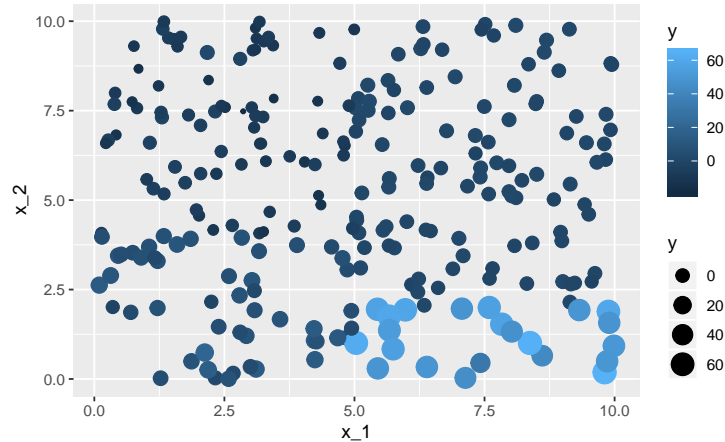
Figure 17.1: How could we model this with a linear model?

Figure 17.2 shows y plotted against x_1 and x_2 separately. In each case, there is a region below $y = 25$ which looks like it would fit with an intercept only model, but the variance of the response is heteroskedastic. Moreover, this heteroskedasticity has — at least for x_1 — two distinct regions, with no smooth transition between them. Finally, the values for $y > 25$ are white noise for two distinct regions of x_1 and x_2.

```
tbl_tree %>%
  pivot_longer(-y, names_to = 'variable') %>%
  ggplot(aes(value, y)) +
  geom_point() +
  facet_wrap(~variable)
```
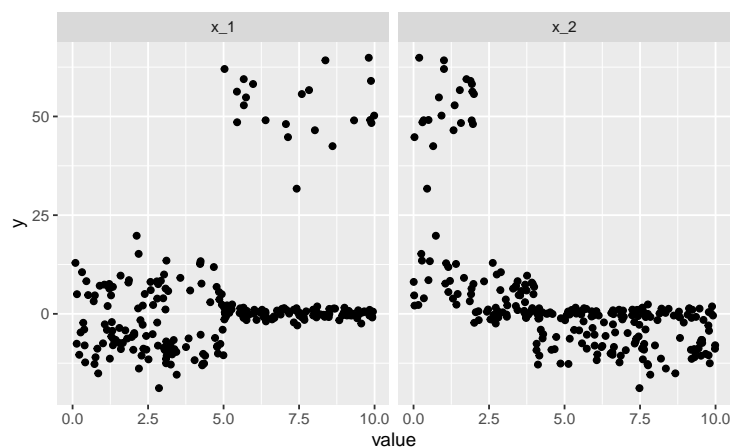

Figure 17.2: The response against each predictor

Any attempt at capturing this dynamic with a linear model will force us to splice the predictor space before we fit. Go ahead and try it (no, really!), but we will find this is a cumbersome process and that any model we develop will not tell us much more than we discovered via the process of splicing x_1 and x_2. So why not focus on all of the benefit that we get from splicing the predictor space? This is — among other things — exactly what a decision tree does.

We know from construction that our sample data is an amalgam of four distinct responses, each of which have the same mean. Another way of phrasing that is to say that we could divide Figure 17.1 into four rectangular regions such that the homogeneity of each one is maximized. That's the 2-D interpretation. In one dimension, we would divide the predictor into distinct line segments. In three dimensions, the regions would be right rectangular prisms and anything higher of higher dimension is simply a hyperrectangle.

So, that's the geometry. To gauge homogeneity, we will take the mean response within the region as our prediction. We will then use the sum of squared errors (SSE) between the predictor and the actual response as our homogeneity measure. The homogeneity of our model is simply the sum of the SSE of each region. Note that in talking about the metric in this way we are saying that homogeneity is *maximized* when total model SSE is *minimized*. We can capture this with the slightly hacky Equation (17.1).

$$SSE_{model} = \sum_{j}^{allregions} \sum_{i \in region_j} (y_i - \hat{y}_j)^2 \qquad (17.1)$$

In Equation (17.1), note that the estimate, $\hat{y}_j$ for region j is simply the average of the response in that region as shown in Equation (17.2).

$$\hat{y}_j = \sum_{i \in region_j} y_i / N_j \qquad (17.2)$$

Our model is constructed step by step, one predictor and one region at a time. At each stage, we take each predictor and explore all of the potential split points for that predictor. At each point, we calculate the SSE which would result by splitting at that point. Note that we only need to know about the SSE for the two new potential regions in question, not the overall SSE shown in Equation (17.1). Why? Because we will only split if the two subregions improve the SSE for their overall region. All of the areas are disjoint, so we are free to divide one, knowing that it will not have an impact on any of the other regions. The code below will do this.

```
calc_sse_split <- function(split_point, predictor, response) {

  predict_left <- mean(response[predictor <= split_point])
  predict_right <- mean(response[predictor > split_point])

  ifelse(
      predictor <= split_point
    , response - predict_left
    , response - predict_right
  ) %>%
  raise_to_power(2) %>%
  sum()
}
```

We can experiment with this for a few easy examples. You should try these out by hand to check that the results make sense.

```
calc_sse_split(2, 1:5, c(1,2,1,3,4))
#> [1] 5.17
calc_sse_split(3, 1:5, c(1,2,1,3,4))
#> [1] 1.17
```

Note that our function is not vectorized for split_point. The code below will generate a single answer along with several warnings.

```
calc_sse_split(2:3, 1:5, c(1,2,1,3,4))
#> Warning in predictor <= split_point: longer object length is not
#> a multiple of shorter object length
#> Warning in predictor > split_point: longer object length is not
#> a multiple of shorter object length
#> Warning in predictor <= split_point: longer object length is not
#> a multiple of shorter object length
#> [1] 5.17
```

The fact that split_point is not vectorized makes a bit of sense. We want to be able to pass in a vector of any potential length and it will not necessarily be the same size as the predictors and responses. If it's of a different size than the predictor we will have some confusing — and frankly unnecessary — overhead to work out the if statements. The map_dbl() function from purrr renders all of this moot. Recall that map_*() will call a function for each element in its first argument.

```
map_dbl(2:3, calc_sse_split, 1:5, c(1,2,1,3,4))
#> [1] 5.17 1.17
```

With that function in place, we can measure the SSE at various split points for each of
our predictors.

```
tbl_tree <- tbl_tree %>%
  mutate(
      sse_x1 = map_dbl(x_1, calc_sse_split, x_1, y)
    , sse_x2 = map_dbl(x_2, calc_sse_split, x_2, y)
  )
```

As we see in Figure 17.3, the SSE decreases along the x-axis until about $x = 5$, where it
begins to increase. The line is very noisy, but we are not surprised by this. y has a fair
bit of variation in it.

```
tbl_tree %>%
  ggplot(aes(x_1, sse_x1)) +
  geom_line() +
  scale_x_continuous(breaks = 1:10)
```
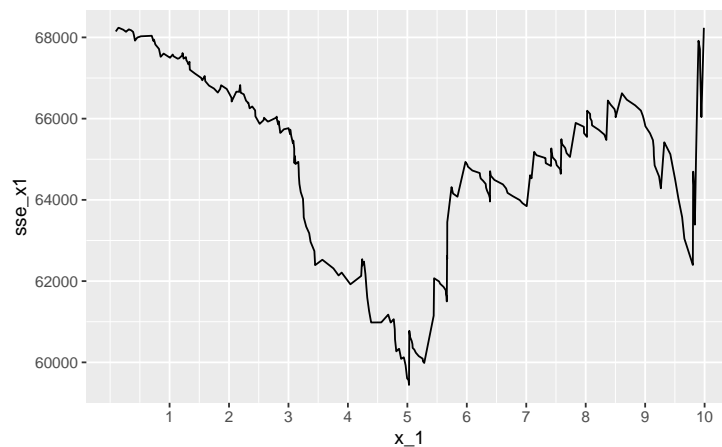

Figure 17.3: SSE against x_1

We will see something similar when we observe x_2. However, let's plot both predictors at the same time.

```
tbl_hach_predict %>%
  ggplot(aes(index, value, color = as.factor(state))) +
  geom_line(aes(linetype = model)) +
  geom_point(aes(shape = model)) +
  facet_wrap(~ state, scales = 'free_y') +
  geom_smooth(method = 'lm', formula = y ~ x, se = FALSE) +
  geom_smooth(
      method ='lm'
    , formula = y ~ 1
    , se = FALSE
    , linetype = 'dashed'
  )
```
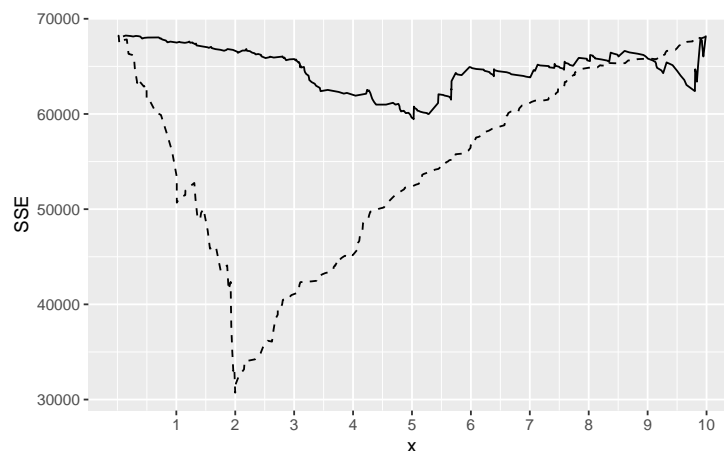

Figure 17.4: SSE for both predictors

In Figure 17.4, the dip at $x_1 = 5$ is still there, but the scale of change in x_2 is much more significant. Splitting my data here is the most efficient way to improve my prediction. So what happens next? At this point, we split the data in two: one set contains every observation having $x_2 \leq 2$ and the other every observation with $x_2 > 2$. We then split those two data sets into 3 or 4 and so on and so on.

We refer to each data set as a *node*. The first is known as the *root* or sometimes *trunk* node. At the user's discretion, we may opt to halt the process if the number of observations within a node drops below a certain population. This node will be referred to as a *leaf* or *terminal* node.

You will be happy to know that you will not need to program this recursive algorithm. (Although, give it a shot. It's good practice and will help you appreciate the design considerations that go into more widely used packages.) Two of the more popular packages for fitting a decision tree are `rpart` and `tree`. They are broadly similar, but we will focus on `rpart`, because it gives us a bit more control over the fit (which will come in handy later). The function to fit a model looks very similar to `lm()` and `glm()`. It will take a `formula` object and a data frame as inputs.

```
library(rpart)

fit_tree <- rpart(
    formula = y ~ x_1 + x_2
  , data = tbl_tree
)
```

The prediction function is also similar to `lm()` and `glm()`. Simply pass in the fit object and we will get back a vector of predictions. `predict()` has a few options for the `type` argument which are relevant when the response is categorical.

```
tbl_tree <- tbl_tree %>%
  mutate(
    predict_tree = predict(fit_tree)
  )
```

You will rarely hear the word 'residual' used in connection with decision trees, but it's straightforward to generate one. When we visualize it, we see that the decision tree has partitioned our data such that we have five distinct predictions for y. We know, from construction, that there are only four in the data[1].

```
tbl_tree %>%
  mutate(residual = y - predict_tree) %>%
  ggplot(aes(predict_tree, residual)) +
  geom_point()
```

---

[1]Re-examine the code at the beginning of the chapter if you are not sure how many different values for the response to expect.
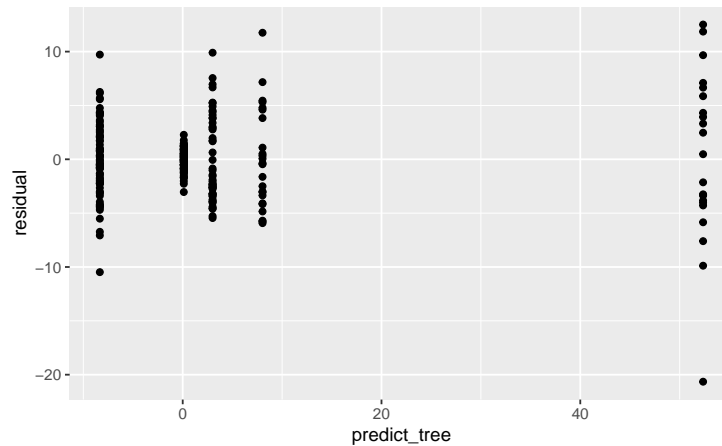
Figure 17.5: Five distinct predictions

To get a sense for what's behind this, we can plot the tree that we have produced. The display is spartan, but will suit our purposes. Note that you will almost always call text() immediately after calling plot()[2]. The plot is shown in Figure 17.6.

```
plot(fit_tree)
text(fit_tree)
```
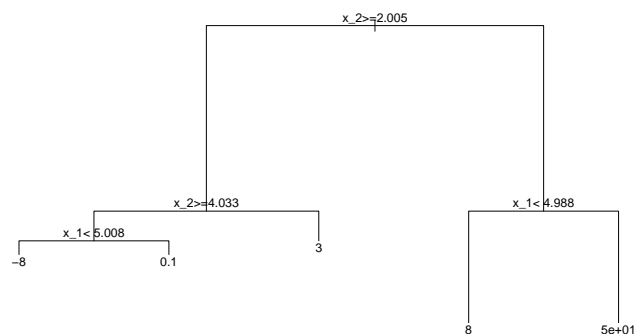

Figure 17.6: Where our decision tree predicts

----

[2]I don't know who would be satisfied with an unlabeled decision tree either.

So, why do we have five leaf nodes rather than 4? The answer is that we split on $x_2 >= 2$ first. If we had split on $x_1 < 5$, it would have consolidated the nodes shown at 3 and 8 in our model. This is a good illustration of the fact that a decision tree is a *greedy* algorithm. If the algorithm had looked one or two steps ahead, it would have recognized that SSE could be reduced further by splitting on x_1 rather than x_2 in the first step. Largely for computational reasons it does not do this. Greediness is not often an issue with decision trees. However, it illustrates that you must pay attention to the output of your model and check it against what you know about your data. Later, we will see a technique that may alleviate the effects of greediness.

17.2   Categorical responses

The algorithm we just implemented presumed that we have a continuous response. It is straightforward to use the same approach for a categorical response; we simply need to change how we measure homogeneity. Take a look at the vectors in the code snippet below. For the numeric vectors, try to imagine which one has a higher SSE. Now have a think about which character vector is more homogeneous. You may find it a bit easier to form that guess when looking at the character data.

```
nums_1 <- c(1, 1, 2, 3, 3, 3)
nums_2 <- c(1, 2, 3, 3, 3, 3)

cats_1 <- c('jaguar', 'jaguar', 'tiger', 'lynx', 'lynx', 'lynx')
cats_2 <- c('jaguar', 'tiger', 'lynx', 'lynx', 'lynx', 'lynx')
```

Having done that, let's look at three different mathematical definitions of homogeneity for categorical data. In each of the equations which follow, $\hat{p}_k$ measures — for a given region j — the percentage of observations in category k, as in Equation (17.3). Also, for each expression, a lower value indicates greater homogeneity.

$$\hat{p}_k = \sum_{i \in k} / \sum_{i \in j} = N_k/N_j \tag{17.3}$$

1. The *classification error rate* reflects the portion of observations which are not in the plurality class. This is measured simply as the complement to the portion of observations in that class, as in Equation (17.4). We can see that we achieve a lower value for this measure when one particular class dominates, that is, when its probability is close to 1.

$$CER = 1 - \max_k (\hat{p}_k) \tag{17.4}$$

2. The *Gini index* for a single category is equal to the probability of that category times the complement of probability. Again, if one class is predominant, it will drive the overall measure lower because of the presence of $1 - \hat{p}_k$. For a region with only two classes, the Gini index is maximized when $\hat{p}_k = 0.5$. In this instance, it is just as likely as not that an observation is a member of either class.

$$Gini = \sum_k \hat{p}_k (1 - \hat{p}_k) \tag{17.5}$$

3. The *entropy* is structurally very similar to the Gini measure. However, instead of multiplying by the complement of probability, we multiply by the log of the category's probability. Because the log will generate a negative number — $\hat{p}_k < 0$ — we negate the sum. The concept of entropy comes to us from information theory. Consequently, you will often see the term "information" or "information gain"[3] used to refer to the quantity defined in Equation (17.6).

$$Entropy = -\sum_k \hat{p}_k * log(\hat{p}_k) \tag{17.6}$$

Let's see how each of those measures responds to the simple categorical variables we created earlier. First, we will write a very simple function to compute the probabilities for each class.

```
class_probs <- function(x) {
  table(x) / length(x)
}

cats_1 %>% class_probs()
#> x
#> jaguar    lynx   tiger
#>  0.333  0.500   0.167
cats_2 %>% class_probs()
#> x
#> jaguar    lynx   tiger
#>  0.167  0.667   0.167
```

That done, we can explore the classification error rate:

---

[3]Note that "information gain" will more specifically refer to the *difference* in information before and after a split.

```r
calc_class_error <- function(x) {
  max_prob <- x %>%
    class_probs() %>%
    max()

  1 - max_prob
}

cats_1 %>% calc_class_error()
#> [1] 0.5
cats_2 %>% calc_class_error()
#> [1] 0.333
```

The Gini index:

```r
calc_gini <- function(x) {
  probs <- x %>%
    class_probs()

  (1 - probs) %>%
    multiply_by(probs) %>%
    sum()
}

cats_1 %>% calc_gini()
#> [1] 0.611
cats_2 %>% calc_gini()
#> [1] 0.5
```

And the entropy:

```r
calc_entropy <- function(x) {
  probs <- x %>%
    class_probs()

  probs %>%
    log() %>%
    multiply_by(probs) %>%
    sum() %>%
```

```
      multiply_by(-1)
}

cats_1 %>% calc_entropy()
#> [1] 1.01
cats_2 %>% calc_entropy()
#> [1] 0.868
```

No matter what measure we use, we see that `cats_2` is always the more homogeneous of the two vectors. Hopefully that squares with your intuition. A natural question arises as to which measure is more useful in practice. [James et al., 2017] states that the classification error rate is not always the must useful at partitioning our data. However, there is scant guidance as to the circumstances when Gini is to be preferred to entropy. [Raileanu and Stoffel, 2004] have conducted research which suggests that there is little reason to prefer one to the other. In the examples which follow, we will use the default in `rpart()`, which is Gini.

Let's add a categorical column to our sample data. We will simply examine the sample values of *y* and use `case_when()` to categorize them. We will also calculate the Gini and the entropy for the categorical response.

```
tbl_tree <- tbl_tree %>%
  mutate(
      y_cat = case_when(
        y > 25 ~ 'tiger'
      , y < 0 ~ 'lynx'
      , y > 3 ~ 'jaguar'
      , TRUE ~ 'bobcat'
    )
  )

calc_entropy(tbl_tree$y_cat)
#> [1] 1.21
calc_gini(tbl_tree$y_cat)
#> [1] 0.658
```

Just as we did with SSE, we will create a function to determine the homogeneity which results after a split. In this case, though, we will make the `measure` argument a function which we pass in. This allows us to use any of the three measures which we defined earlier. Notice that the value across the entire response is a weighted average of the two

sub-vectors, after the split. The weight is taken as the number of observations in each of the sub-vectors.

```r
calc_split_categorical <- function(
    split_point
  , predictor
  , response
  , measure) {

  index_left <- predictor <= split_point
  index_right <- predictor > split_point

  measure_left <- response[index_left] %>%
    measure()
  measure_right <- response[index_right] %>%
    measure()

  (sum(index_left) * measure_left +
    sum(index_right) * measure_right) %>%
    divide_by(length(predictor))

}
```

Now, let's measure the value of all the potential split points for each of our predictors. We will get a warning in the classification error rate when we try to split at the maximum value of x_1 and x_2. We can ignore this, but your code should be a bit more robust.

```r
tbl_tree <- tbl_tree %>%
  mutate(
    class_error_x1 = map_dbl(
      x_1
      , calc_split_categorical
      , x_1
      , y_cat
      , calc_class_error
    )
  , entropy_x1 = map_dbl(
      x_1
      , calc_split_categorical
      , x_1
```

```
      , y_cat
      , calc_entropy
    )
  , gini_x1 = map_dbl(
        x_1
      , calc_split_categorical
      , x_1
      , y_cat
      , calc_gini
    )
  , class_error_x2 = map_dbl(
        x_2
      , calc_split_categorical
      , x_2
      , y_cat
      , calc_class_error
    )
  , entropy_x2 = map_dbl(
        x_2
      , calc_split_categorical
      , x_2
      , y_cat
      , calc_entropy
    )
  , gini_x2 = map_dbl(
        x_2
      , calc_split_categorical
      , x_2
      , y_cat
      , calc_gini
    )
  )
#> Warning in max(.): no non-missing arguments to max;
#> returning -Inf
```

As we did with a continuous response, we may compare the two predictors on the basis of one of our measures to see which one we would split first. The results are displayed in Figure 17.7.

```
tbl_tree %>%
  ggplot() +
  geom_line(aes(x_1, entropy_x1)) +
  geom_line(aes(x_2, entropy_x2), linetype = 'dashed') +
  labs(x = 'x', y = 'entropy')
```
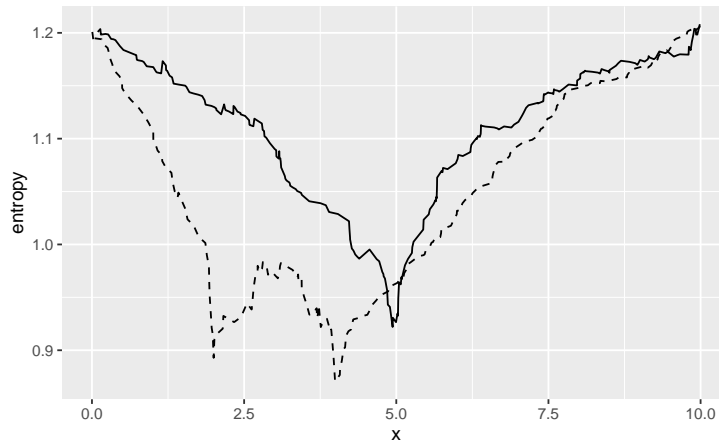

Figure 17.7: x_1 and x_2 for the categorical response

On the basis of entropy, we would again split at x_2 first. Would another measure have lead us to a different conclusion? Let's compare all three measures for x_2. First, we will construct a data frame which gathers the three measures and forms a column to identify them. That column will make the plotting easier[4]. Figure 17.8 shows us that the conclusion is more or less the same.

```
tbl_x2_measures <- tbl_tree %>%
  select(x_2, gini_x2, entropy_x2, class_error_x2) %>%
  pivot_longer(-x_2, names_to = 'measure') %>%
  mutate(measure = gsub('_x2', '', measure))

tbl_x2_measures %>%
  ggplot(aes(x_2, value)) +
  geom_line() +
  facet_wrap(~measure, scales = 'free')
```

---

[4]You could also do the `pivot_longer()` and `mutate()` before passing the result into `ggplot()`. I'm splitting them into distinct steps in hopes that it makes the exposition smoother.
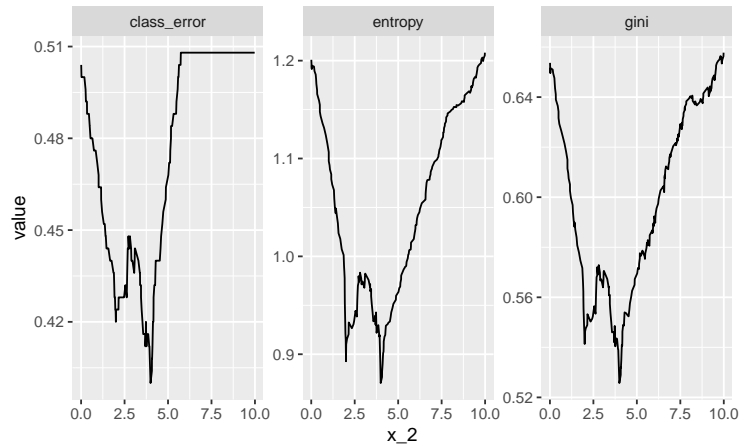
Figure 17.8: How the different measures compare for x_2

For a complete tree, we may again use `rpart()`, with exactly the same function interface. The default measure is Gini. If we like, we may choose to use entropy by passing in an option to the `parms` argument. This is shown in the second call in the code block below.

```
fit_rpart_gini <- rpart(
    data = tbl_tree
  , formula = y_cat ˜ x_1 + x_2
)

fit_rpart_info <- rpart(
    data = tbl_tree
  , formula = y_cat ˜ x_1 + x_2
  , parms = list(split = 'information')
)
```

For a categorical variable, the prediction will be the category which appears most often in that region. However, `rpart` provides several other options as well. The default will return a matrix showing the probability for every categorical response. For example:

```
predict(fit_rpart_gini) %>%
  head(2)
#>    bobcat jaguar  lynx tiger
#> 1   0.013      0 0.987     0
#> 2   0.760      0 0.240     0
```

To return the prediction as a vector which matches the response, specify `type = 'class'`.

```
predict(fit_rpart_gini, type = 'class') %>%
  head(2)
#>     1      2
#>   lynx bobcat
#> Levels: bobcat jaguar lynx tiger

tbl_tree <- tbl_tree %>%
  mutate(predict_cat = predict(fit_rpart_gini, type = 'class'))
```

Because categorical responses are non-numeric, the concept of a residual does not really apply. Even if we were to map predictions and observations to a set of integers, it would not be terribly enlightening. For each observation, the prediction either matches exactly, or it does not match at all. In other words, every "residual" is either 1 or 0.

Instead, we summarize the results for the model as a whole. We can capture this as a single statistic, called the *accuracy*, which is simply the number of correct predictions divided by the sample size. Though helpful, it is worthwhile to inspect *where* the model is and is not accurate. Results by class are easy to obtain and are shown in Table 17.1.

```
tbl_cat_results <- tbl_tree %>%
  mutate(
      true_prediction = (predict_cat == y_cat)
    , false_prediction = !true_prediction
  ) %>%
  group_by(y_cat) %>%
  summarise(
      true_prediction_rate = sum(true_prediction) / n()
    , false_prediction_rate = sum(false_prediction) / n()
  )
```

Category	True Prediction Rate	False Prediction Rate
bobcat	71.9%	28.1%
jaguar	100.0%	0.0%
lynx	82.1%	17.9%
tiger	100.0%	0.0%

Table 17.1: Prediction accuracy by class

The case where there are only two potential responses — analogous to a logistic regression — has been well studied and has several diagnostic tools. We will talk about some of them further in Chapter 19.

17.3   Bagging

The example we have used so far is very simple. For something more complex, we will simulate some data using the caret package, about which we will have more to say in Chapter 18. This will have several potential predictors, as well as some "noise" columns.

```r
library(caret)

set.seed(1234)
tbl_tree_complex <- twoClassSim(
    sims
  , intercept = -5
  , linearVars = 2
  , noiseVars = 2
  , corrVars = 3
  , corrValue = 0.1)
```

Play around with that data for a bit. It's purely simulated, but make some plots and see if any structure suggests itself.

All done? Now we are going to fit a decision tree, but instead of fitting it to the entire data set, we will construct a model using only half of our sample. We will then use that model to try and predict the other half.

```r
test_index <- sample(nrow(tbl_tree_complex), sims / 2)
tbl_tree_complex_train <- tbl_tree_complex[test_index, ]
tbl_tree_complex_test <- tbl_tree_complex[-test_index, ]

fit_tree_train <- rpart(
    Class ~ .
  , data = tbl_tree_complex_train
)
```

We will create a short utility function to compute the accuracy of our model. Note the use of `enquo()` so that we may flexibly pass in columns from our data frame within a call to `mutate()`. We first saw this in Chapter 10.

```r
test_tree <- function(tbl_in, actual, predicted) {
  actual <- enquo(actual)
  predicted <- enquo(predicted)

  tbl_in %>%
    mutate(accurate = !!predicted == !!actual) %>%
    summarise(pct_accurate = sum(accurate) / n()) %>%
    pull(pct_accurate)
}
```

Finally, we will compare the accuracy of our model prediction on the training set to the prediction on the test set.

```r
tbl_tree_complex_train %>%
  mutate(predicted_class = predict(
      fit_tree_train
    , newdata = .
    , type = 'class'
  )) %>%
  test_tree(Class, predicted_class)
#> [1] 0.904

tbl_tree_complex_test %>%
  mutate(predicted_class = predict(
      fit_tree_train
    , newdata = .
    , type = 'class'
  )) %>%
  test_tree(Class, predicted_class)
#> [1] 0.76
```

Although the model performs rather well on the data used to calibrate it, the performance drops for out-of-sample data. This is surprising, because we know that both data sets were formed from the same stochastic process. This is not a good thing. Most of what actuaries do relates to out-of-sample data, that is, estimates about things which will happen in the future. Predicting the past does not do us much good!

This is a common challenge for a decision tree model. In this case, the decision tree is too specific. Rather than capturing the broad stochastic process and variable relationships in play, it has simply "memorized" the data used to fit it. One strategy to overcome this is to "prune" the tree. To do so, one "cuts back" leaf nodes based on some criteria. Another commonly used criteria is to penalize the measure of model fit by the model complexity as determined by the number of leaf nodes.

Another way to address poor fit for out-of-sample data is to meet it head on. We failed to predict the test data because we did not use it in our model. So, let's use it. We could fit each sample and then weight the results in our prediction. Below, we are using the default version of `predict()` which returns a matrix of probabilities for each observation for each class. We fetch only the first column, which will be the probability for *Class*1.

```r
fit_tree_test <- rpart(
    Class ~ .
  , data = tbl_tree_complex_test
)

tbl_tree_predict <- tbl_tree_complex %>%
  mutate(
      predict_train = predict(fit_tree_train, newdata = .)[, 1]
    , predict_test = predict(fit_tree_test, newdata = .)[, 1]
    , predict_mean_prob = (predict_train + predict_test) / 2
    , predict_mean_class = ifelse(
          predict_mean_prob > 0.5
        , 'Class1'
        , 'Class2'
    )
  )
```

With that, we can test the combination of two models on the training set, the test set and all of our data[5].

```r
# training set
tbl_tree_predict %>%
  slice(-test_index) %>%
  test_tree(Class, predict_mean_class)
#> [1] 0.76
```

[5]In this instance, we are using the terms "training" and "test" set rather loosely, in order to facilitate a comparison with our earlier example. We will get more precise definitions for these terms in Chapter 19.

```
# test set
tbl_tree_predict %>%
  slice(test_index) %>%
  test_tree(Class, predict_mean_class)
#> [1] 0.904

# everything
tbl_tree_predict %>%
  test_tree(Class, predict_mean_class)
#> [1] 0.832
```

The performance for the training set went down, but the performance on the test set went up. The overall performance was somewhere in between. Taking the average of "memorizing" two sets of data has some appeal. To get even better results, we can scale up from two subsamples to many. We bootstrap a sample, fit a model and aggregate the results. If this is the first time you've heard the word "bootstrap" in a statistical context, here's the definition in a nutshell: 1) take multiple samples — with replacement — from a sample, 2) calculate some statistical measure, 3) average across all of the measures. You can find more detail in [James et al., 2017].

This form of "bootstrap aggregation" is better known as "bagging". One implementation is found in the ipred package.

```
library(ipred)
fit_bag <- bagging(
    Class ~ .
  , data = tbl_tree_complex
  , nbagg = 10
)
```

Bagging is our first example of an *ensemble model*. An ensemble model works from the idea that a collection of many weak models will aggregate to a single model which performs well. To me, this calls to mind the old cliche that "the whole is greater than the sum of its parts".

Because we are bootstrapping, we should be aware that the predictions themselves are random. We may minimize the variability by running many trials. See the results below for two models run with only ten samples each. The predictions are different between them.

```r
set.seed(1234)
bagging(
    Class ~ .
  , data = tbl_tree_complex
  , nbagg = 10
) %>%
  predict() %>%
  head()
#> [1] Class2 Class2 Class1 Class2 Class1 Class2
#> Levels: Class1 Class2

set.seed(5678)
bagging(
    Class ~ .
  , data = tbl_tree_complex
  , nbagg = 10
) %>%
  predict() %>%
  head()
#> [1] Class2 Class2 Class2 Class2 Class2 Class1
#> Levels: Class1 Class2
```

One great benefit about bagging is that we automatically generate an estimate for how a model will perform on out of sample data. Whenever we fit a decision tree on a subsample, we can test its performance on all of the data which we did not use. That data is referred to as "out of bag" or *OOB*. To capture the model error, we must override the default for coob (calculate out of bag) by passing in coob = TRUE.

```r
fit_bag_oob <- bagging(
    Class ~ .
  , data = tbl_tree_complex
  , coob = TRUE
)

fit_bag_oob$err
#> [1] 0.192
```

The complement of estimated error is the accuracy figure we calculated earlier with our training and test sets. Using 25 trees (the default), we have 0.808. Using fewer trees will decrease this amount, while increasing the number of trees might increase it. Why might? Because there is a ceiling on just how accurate our models can be. In general, there will always be some random variation which will cause our predictions to fail for some observations.

Bagging is also a departure from simple decision trees in that there is no longer a simple interpretation. Rather than following a sequence of logical propositions, we are computing a prediction on the basis of the averages of many such sequences. In other words, what does the prediction have to do with the various predictors?

This — and another improvement — is something we get from random forests.

17.4   Random Forests

With bagging, each time we are testing the same set of potential predictors on a random subset of our original data. What's wrong with this approach? In principal, not much. However, consider this: if one, or a few, of the predictors works very well, then it will work well in every bootstrapped tree. This will erode the independence between the individual trees. Moreover, it could obscure the ability of other predictors to perform well for certain areas of the response.

That brings us to random forests. Like bagging, a random forest will use a bootstrapped subsample to fit a single decision tree. However, at *each* node it will use only a subset of the potential predictors. This means that each tree in a random forest model will have less in common with all of the others.

```
library(randomForest)

set.seed(1234)
fit_forest <- randomForest(
    formula = Class ~ .
  , data = tbl_tree_complex
)
```

How many trees are enough? The default is 500. We can gain a sense for how much each additional tree improves our estimate by using the "out of bag" error concept, which we first referenced in our discussion of bagging. The default plot function for the `randomForest` package will display this, but I like the implementation from the `ggRandomForests` package a bit better. That package has a number of methods which

will extract information from a random forest. You must then convert it to a `ggplot2` object with a call to `plot()`. Once that's done, we may use all of the `ggplot2` features that we know and love.

```
library(ggRandomForests)

fit_forest %>%
  ggRandomForests::gg_error() %>%
  plot() +
  geom_vline(xintercept = 70)
```
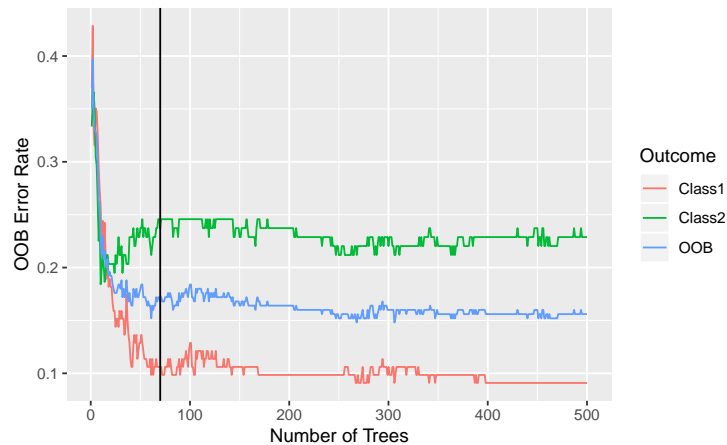

Figure 17.9: OOB error improvement for a random forest model

As we see in Figure 17.9, the OOB error rate shows little improvement after 70 trees have been used.

Earlier, we said that bagging does not provide us with an instant view of the relationship between predictors and a response in the same way that a single decision tree does. However, by examining all of the different trees used in the fit, we can get a feel for this. The *importance* of a variable measures the average improvement in fit across all of the models which were constructed. Again, remember that a variable is not necessarily tested at each node.

The `randomForest` package uses two different measures for "improvement". The first examines the difference between the OOB error of the model and a measure of OOB where each of the predictors has been randomly rearranged. That quantity is averaged across all trees. The idea here is that when those two measures are similar, the model fit

does not depend on a particular predictor. The second measure simply computes the average of the total decrease in node heterogeneity for each predictor.

To retrieve both, we must override the default argument for `importance` when fitting the model as follows:

```
fit_forest <- randomForest(
    formula = Class ~ .
  , data = tbl_tree_complex
  , importance = TRUE
)
```

The `importance()` function returns values as a matrix, an example of which is shown in Table 17.2.

```
importance(fit_forest)
```

We may also plot the results with the `varImpPlot()` function, as in Figure 17.10.

```
varImpPlot(fit_forest, main = NULL, n.var = 10)
```

	Class1	Class2	MeanDecreaseAccuracy	MeanDecreaseGini
TwoFactor1	27.217	35.672	38.582	27.07
TwoFactor2	28.737	37.425	40.349	29.97
Linear1	-1.120	-1.652	-1.815	5.13
Linear2	6.479	6.756	8.730	10.03
Nonlinear1	1.530	5.282	4.638	8.36
Nonlinear2	-2.071	0.554	-1.047	6.74
Nonlinear3	-2.019	2.119	-0.168	6.38
Noise1	-1.664	-2.524	-2.938	4.82
Noise2	1.178	-1.880	-0.244	5.85
Corr1	0.521	-1.226	-0.427	6.55
Corr2	0.751	0.177	0.624	7.32
Corr3	-0.376	1.859	0.999	5.91

Table 17.2: Variable importance information
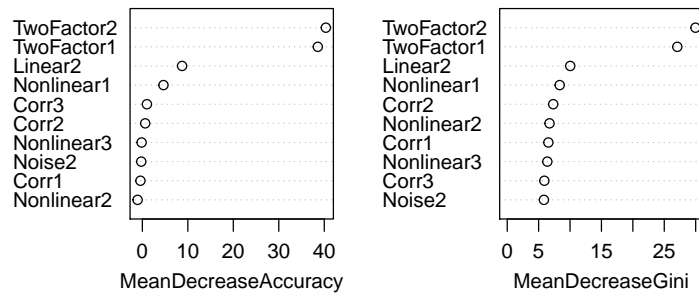
Figure 17.10: Visual display of variable importance

In each instance, we see that the variables `TwoFactor1` and `TwoFactor2` tend to be most useful in making a prediction.

Variable importance is the start of the conversation about how to interpret an ensemble model. Additionally, it is useful to see roughly *where* a predictor is important. A partial dependency plot does this. For example, consider Figure 17.11.

```
partialPlot(
    fit_forest
  , pred.data = tbl_tree_complex
  , x.var = 'TwoFactor2')
```
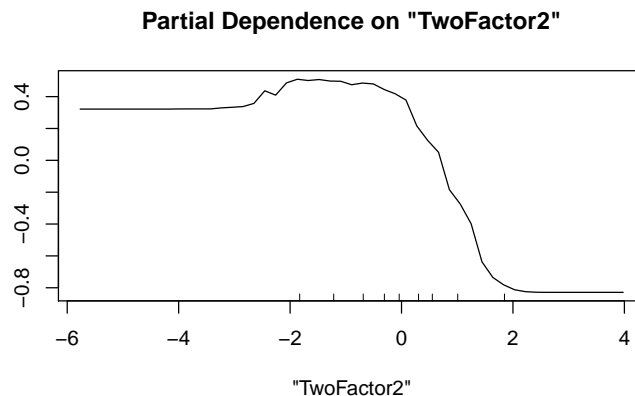

Figure 17.11: Partial dependence plot for our complicated data

The scale of the y-axis of the plot is not shown and its construction is a bit involved[6].

---

[6]For more detail, please refer to [Hastie et al., 2017].

We take a set of points along the range of the predictor which we are interested in[7]. At each point, we substitute that value for every cell for the predictor in our sample data frame. We then predict using our model. That is, we hold a single predictor constant and observe the central tendency of the prediction. For trees which use SSE, "central tendency" is simply the mean of the response. For a classifier, we must specify which class we are interested in observing. The "central tendency" is then the difference between the log probability for the class of interest and the average log probability across all classes. This is shown in Equation (17.7).

$$f(x) = \log p_k(x) - \frac{1}{K} \sum_{j=1}^{K} \log p_j(x) \tag{17.7}$$

The idea here is that there will be particular places along the predictor space where the predicted response changes. For example, cast your eye on Figure 17.12 (the code to produce it appears after this paragraph). We can see that the curve increases sharply at $x_1 = 5$. This jives with what we know about the data. For a real-world application, imagine being able to replace $x_1 = 5$ with "risk-free interest rates $> 3.5\%$", or "policyholder retention $< 80\%$". Risk-free interest rate or policyholder retention could be one of a dozen, or several dozen variables in a model. They will not get a coefficient as they would in a linear regression or a GLM, so their influence on the result is not as easy to describe. A partial dependence plot helps to bridge that gap.
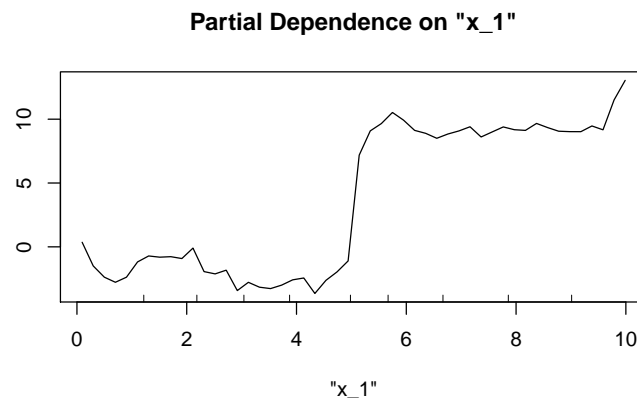
Partial Dependence on "x_1"

Figure 17.12: A partial dependency plot for our simple tree data

---

[7]The default will be the number of unique values, but not more than 51.

```
fit_forest_simple <- randomForest(
    y ~ x_1 + x_2
  , data = tbl_tree
)

partialPlot(
    fit_forest_simple
  , pred.data = tbl_tree %>% as.data.frame()
  , x.var = 'x_1'
  , n.pt = 50)
```

For more detail on how variable importance and partial dependency are calculated, the interested reader is referred to [Hastie et al., 2017].

17.5   Boosting

Boosting is another approach to ensemble models. However, rather than fitting many random decision trees and averaging across all of them, boosting works iteratively. The idea is that we begin with a very simple model, which we do not expect to perform well. We then observe where the fit is the worst and direct our next model to those areas of our data. This continues until the change in our prediction is very small, or a maximum number of iterations has been reached.

We can target the next model in the sequence in one of two ways. The first uses the difference between the current estimate and the actual observation (i.e. the residual) as the response term in a decision tree. The estimate is recursively adjusted by adding in the latest predicted value from the decision tree. This is implemented in the gbm package.

The second approach will continually adjust the *weight* which we assign to each observation in our models. Those observations associated with particularly poor predictions get more weight in the next iteration of the model. That's the idea behind the model found in the ada package.

gbm

We can think through this with our simple data set. In a moment, we will be adding many new columns, so let's create a fresh and streamlined version of our sample tree data to store it. We will also initialize our prediction and residual.

```r
tbl_tree_boost <- tbl_tree %>%
  select(y, x_1, x_2) %>%
  mutate(
      predict_0 = 0
    , resid_0 = y - predict_0
  )
```

Now let's construct a function which will perform the iterative prediction. We don't need to do this — in a moment we will have a look at an R package which does this for us. However, this is good practice. We will get a better feel for how the algorithm works and we will continue to accumulate experience in function design and data structure. And who knows? Maybe someone reading this will get inspired to design the next great machine learning algorithm!

```r
add_prediction <- function(
    tbl_in
  , n_iteration
  , predictors
  , max_depth = 1
  , shrinkage = 0.05) {

  # Ensure that the sequence has been initialized
  if (n_iteration < 1) {
    stop("n_iteration must be >= 1.")
  }

  # collapse will convert a vector of predictors into a single
  # character string with each element separated by `+`
  the_formula <- paste(predictors, collapse = '+') %>%
    paste0('resid_', n_iteration - 1, ' ~ ', .) %>%
    as.formula()

  fit <- rpart(
      the_formula
    , data = tbl_in
    , control = rpart.control(maxdepth = max_depth)
  )

  # This will pull the current and new prediction columns
```

```r
  pred_cols <- paste0("predict_", c(n_iteration - 1, n_iteration))

  # The new prediction is equal to the current +
  # the predicted residual
  tbl_in[[pred_cols[2]]] <- tbl_in[[pred_cols[1]]] +
    shrinkage * predict(fit)
  tbl_in[[paste0('resid_', n_iteration)]] <- tbl_in$y -
    tbl_in[[pred_cols[2]]]

  tbl_in
}
```

If you have read the code, with attention paid to the comments and you have typed it in yourself, you are good to go. Two items, however, warrant a bit of attention. The first is the shrinkage parameter. This is the *rate* at which our model evolves[8]. Setting it to a low value means that we will move very slowly towards the next model estimate. This is cautious, but means that we are less likely to respond to particularly significant bits of noise in our data. We will need more iterations to arrive at a solution, so bear that in mind.

The second item is the the `max_depth` parameter. As with a low learning rate, this is another way of keeping the model evolution process very simple and slow. `max_depth` will control the number of levels which we permit in the decision tree which we are using to model the residuals. In the extreme, we may limit ourselves to only one split point, giving us a "stump". In our function, `max_depth` is passed to the `control` parameter to the `rpart()` function. That parameter is constructed by the `rpart.control()` function. This allows us to tune a number of options like the number of observations required for a split to be considered (`minsplit`), or the minimum number of observations which must exist in a leaf node (`minbucket`).

And now we are ready to iterate! Attentive readers may be surprised to see a `for` loop after the discussion in Chapter 4. It is true that we should prefer functions like `map()`. However, recall that one of the cases where we *must* use a `for` loop is when we want to recursively perform an operation. That is what we are doing in this case.

```r
for (i_iteration in seq_len(50)) {
  tbl_tree_boost <- tbl_tree_boost %>%
    add_prediction(i_iteration, c('x_1', 'x_2'))
}
```

---

[8]In fact, some models will refer to this parameter as the "learning rate".

We visualize the results by plotting our prediction and the actual value. In Figure 17.13, we will show the actual observations as black, the final prediction as blue and the prediction after five iterations as green. We can see that after 50 iterations, the results form several sets where the final results are meaningfully distant from iteration ten. This reflects the careful evolution of our prediction based on the default shrinkage of 0.05.

```
tbl_tree_boost %>%
  pivot_longer(
      c(x_1, x_2)
    , names_to = 'pred_name'
    , values_to = 'pred_val'
  ) %>%
  ggplot(aes(pred_val)) +
  geom_point(aes(y = predict_50), color = 'blue') +
  geom_point(aes(y = predict_10), color = 'green') +
  geom_point(aes(y = y)) +
  facet_wrap(~ pred_name)
```
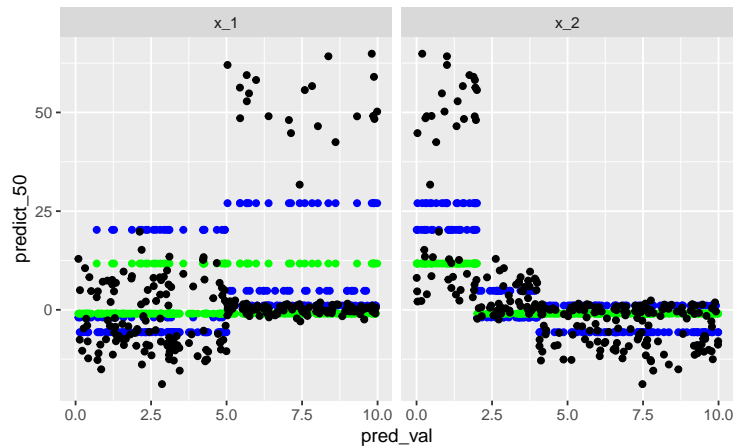

Figure 17.13: How our boosting algorithm performs

We can get a feel for how the learning rate influences the outcome by overriding the default shrinkage parameter with a value of 0.5. The code to do this is not shown. In Figure 17.14, we see that iteration 50 is quite a bit closer to iteration 10. This is because the algorithm is moving much faster. In addition, the straight lines which we saw in Figure 17.13 show a bit more variation.

Which parameter do we prefer? This is a question which we will answer in Chapter 19.

Figure 17.14: Our boosting algorithm with a higher learning rate

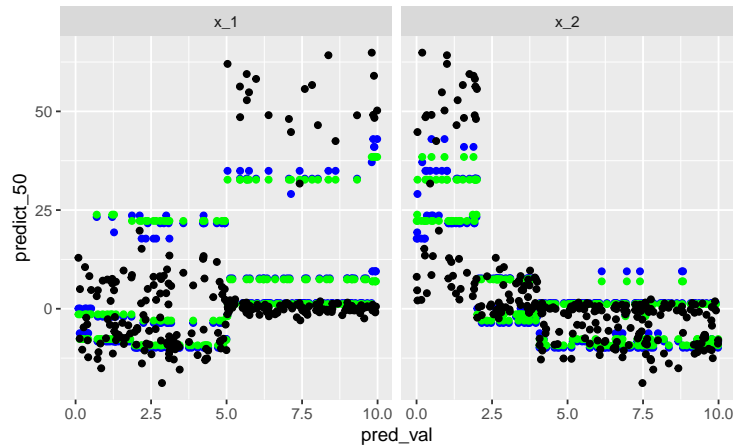The process above is implemented in the package gbm, which is an abbreviation for *gradient boosting machine*. We will apply it to our tbl_tree_complex data frame, but first, we need to convert our *Class* variable from a factor to an integer. Moreover, we must ensure that the values are either 0 or 1. This is a constraint imposed by the algorithm in gbm. The code below will do this.

```
tbl_tree_complex_gbm <- tbl_tree_complex %>%
  mutate(
      Class = as.integer(Class)
    , Class = Class - min(Class)
  )
```

When running the model, we must specify how many trees to use. Two hundred will do just fine here. In addition, it is a good idea to specify whether this is a classification or a regression problem. If we do not pass in something for the distribution argument, gbm() will make its best guess. In this case, it will assume "Bernoulli" because we have only two values for the response.

```
library(gbm)
fit_gbm <- gbm(
    Class ~ .
  , data = tbl_tree_complex_gbm
  , n.trees = 200
)
#> Distribution not specified, assuming bernoulli ...
```

When predicting with gbm, we must also specify how many trees to use. The value must be less than or equal to the number of trees used when constructing the model. Again, the higher number would use the prediction from a higher iteration. Similar to glm(), the default does not return predictions on the scale of the response, so we must override the default type parameter. On the response scale, we do not get predicted classes. Instead we get probabilities which we must to convert to classes.

```
fit_gbm %>%
  predict(n.trees = 200, type = 'response') %>%
  summary()
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   0.004   0.079   0.360   0.464   0.884   0.996
```

AdaBoost

The *AdaBoost* algorithm is an example of the second type of boosting method, that is, one in which a simple decision tree has weights which constantly adjust to focus on areas of the predictor space which don't predict well.

```
library(ada)
fit_ada <- ada(
    Class ~ .
  , data = tbl_tree_complex
  , iter = 70
)
```

The plot() function for an AdaBoost model object shows the error on the set which was used to calibrate the model, as shown in Figure 17.15.
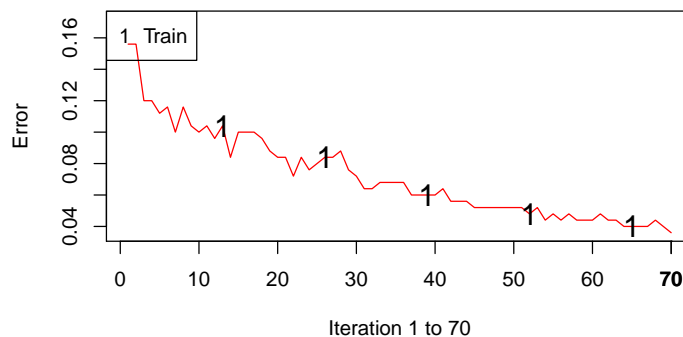
```
plot(fit_ada, tflag = FALSE)
```


Figure 17.15: Training error for AdaBoost

Of greater use will be a plot which shows the error for both training *and* test data sets. To take advantage of this, let's fit our model again, using only the training data set. We then add the test data. When we do, we will have to pass it in as two separate pieces: first the predictors, then the response. This is fairly typical and illustrates the use of two different conventions in defining models in R. The first uses a formula and a complete data frame to define the response and the predictors. This is the approach used by `lm()`, `glm()` and many others. The other paradigm is to pass in variables typically labeled `x` and `y`. This is used often for newer methods like tree-based models, neural networks and the like.

Why the difference? I can only speculate, but here's my take: machine learning methods do not need variable relationships like interaction terms, intercepts and the like. Further, many of them are ported from other programming environments like C++, which use matrices to represent data. Many methods — like `randomForest()` — support both function interfaces. Others — like `xgboost` — do not. Using one or the other is straightforward, just be aware that there are instances when the `formula` argument (which I very much prefer) will not be available.

```
fit_ada_both <- ada(
    Class ~ .
  , data = tbl_tree_complex_train) %>%
  addtest(
    test.x = tbl_tree_complex_test %>% select(-Class)
  , test.y = tbl_tree_complex_test$Class
  )

fit_ada_both %>%
  plot(test = TRUE, tflag = FALSE)
```
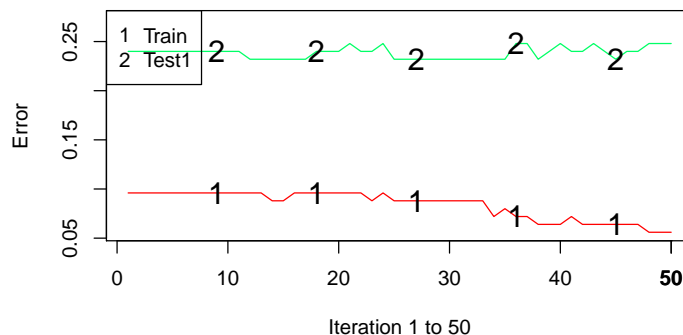

Figure 17.16: Training and test error for AdaBoost

OK, back to the error in our test data. As we see in Figure 17.16, the error rate for the training set continues to decline with each iteration. However, the error for the test set is a good deal higher and declines at a slower pace.

The benefit of ensemble methods

Ensemble methods have an underlying estimation structure which is opaque and they take longer to compute. So why use them? There are a few reasons:

1. They are often better at estimating results for out of sample data.
2. They may be tuned to be less "greedy" than a simple decision tree. They may not be as likely to memorize quirks in the sample data.
3. Random forest methods have an innate defense against correlation across predictors.

Data

Before we leave this chapter, I should pass on a couple quick notes about data.

Many of the tree-based methods — indeed many machine learning algorithms in general — expect character data to be coded as a factor. Some (like xgboost) expect all inputs to be numeric. This is not hard to arrange. The `mutate_if()` function will scan all of the columns of our data frame and test for a particular condition. The `is.*()` functions which we first learned about way back in Chapter 3 are great candidates. Whenever the condition is met, the second argument (a function) will be applied to the column. We can also apply this change to logical columns. This will ensure that none of the algorithms we are using will interpret a logical value as numeric. In other words, we want to ensure its treatment as a qualitative variable.

```
tbl_tree_complex %>%
  mutate_if(is.character, as.factor) %>%
  mutate_if(is.logical, as.factor)
```

Many algorithms will not tolerate missing data. We have a choice here: we can eliminate those observations from the model calibration, or we can replace `NA` with a value. The first option is not desirable as it erodes the power of our model. The second option — variable imputation — allows us to use all of our data. However, we must be cautious about imputing when the true values are unknown (or even unknowable). There are quite a few different algorithms to choose from. The `vtreat` package has very good

support for this. The techniques it uses are non-trivial, so I will suggest reading more about it in [Mount and Zumel, 2014].

An easier approach, which may be useful in situations where missing data is rare, is to use the `na.roughfix()` function. This will replace missing values with the mean or the mode, when the variable is continuous or categorical, respectively.

Below, we randomly insert some missing values in some of our linear predictors. Make sure you understand the use of the `mutate_at()` function and the `random_na()` function which we create.

```r
random_na <- function(x, prob = .02) {
  indices <- rbernoulli(length(x), prob)
  x[indices] <- NA
  x
}

tbl_tree_missing <- tbl_tree_complex %>%
  mutate_at(
      vars(starts_with('Linear'))
    , random_na
  )

fit <- randomForest(
    Class ~ .
  , data = tbl_tree_missing
)
#> Error in na.fail.default(structure(list(Class =
#> structure(c(2L, 1L, 2L, : missing values in object
```

Below, we impute values for cells where we are missing data. This will allow the `randomForest()` algorithm to run. You should compare the two data frames to ensure you understand how the imputation took place.

```r
tbl_tree_missing_fixed <- tbl_tree_missing %>%
  na.roughfix()

fit <- randomForest(
  Class ~ .
  , data = tbl_tree_missing_fixed
)
```

I will again emphasize that variable imputation is not something to be done lightly. There may be operational or market reasons which will account for missingness in data. For instance, is there rating information that was not collected historically? In this case, you should consider whether and how historical data points can contribute to a model. Rather than imputation, it could make more sense to replace the NAs with a signal value which will segment your data and may be directly modeled. Use of a signal is obviously much easier to do if that predictor in question is categorical.

We will have more to say about this in Chapter 18

The forest and the trees

Decision trees have some appealing features. For one, they appear analogous to the process a human goes through when processing information. For example, "Write this risk if revenues are greater than $X, but only if loss experience is less than $Y, but loss experience may be acceptable if they have recently implemented a risk management program". This makes fairly basic decision trees easy to explain to decision makers.

Another way in which they have some appeal is in their treatment of potential predictor variables. For linear models, we must specify which variables are to be included and the model will determine the optimum coefficient for each predictor. When a predictor may have no relation to the outcome, the analyst must decide whether to remove it from the model. The model must then be recalibrated with the remaining predictors, whose coefficient estimates will change. With a decision tree, at every node, we are only retaining those parameters which improve the fit[9].

Finally, as we stated at the top of this chapter, decision trees are invaluable when the response does not change in a linear way, that is when we have "rectangles" of responses.

However, decision trees come with a few potential downsides as well. For one, they may be explainable when there are only a few nodes and levels, but they may quickly become complex. Imagine a tree with several hundred nodes and a similar number of decision points, involving more than a dozen explanatory variables. This is impossible for a human to memorize or intuit

When the response behavior *is* linear relative to predictors, a decision tree will not perform as well. Let's look at an example. We will generate some simulated data, similar to the kinds of examples we explored in Chapter 14.

---

[9]Within the broad family of linear models, there is a technique called "regularization" which may also be used to drop parameters. Read more in [James et al., 2017].

```
tbl_linear <- tibble(
    x = runif(sims, 10, 20)
  , y = 100 + 1.5 * x + rnorm(sims)
)
```

And now, we will fit a decision-tree model to the data and form a predicted result.

```
fit_tree_linear <- rpart(y ~ x, data = tbl_linear)

tbl_linear <- tbl_linear %>%
  mutate(
    predict_tree = predict(fit_tree_linear, newdata = .)
  )
```

Finally, we will plot the prediction alongside the actual values.

```
tbl_linear %>%
  ggplot(aes(x, y)) +
  geom_point(aes(color = 'actual')) +
  geom_point(aes(y = predict_tree, color = 'predicted'))
```
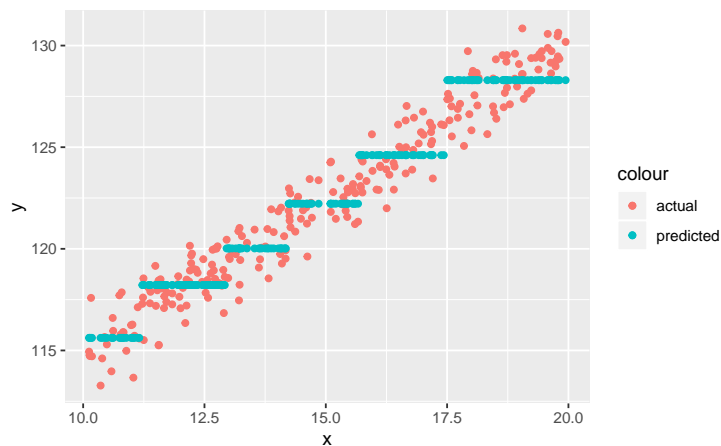

Figure 17.17: A decision tree fit to linear data

The result is a discrete set of predictions which forecast the same value for segments along the x-axis. Each segment will be unbiased, but the error will be more than we would expect from a linear model.

Figure 17.17 suggests another weakness in decision trees. What would our prediction be for $x < 10$ or $x > 20$? It would simply be whatever we have for those regions at the left and right side of our data. A linear model, on the other hand, would extrapolate. Extrapolation is something that a decision tree simply cannot do. To be clear, extrapolation beyond the range of observation is not something to be done lightly. However, in capital modeling, or reinsurance pricing, where we require an estimate of events more extreme than ones we have experienced, it is vital. This is a situation where a decision tree may not be a good choice.

Chapter 17 Exercises

1. Using the results from the `importance()` function, use `ggplot2` to create a plot comparable to the result given by `varImpPlot()`.
2. Using 80% of the records from the `tbl_frequency` data from Chapter 13, fit a single tree, random forest and gradient boosted machine for the number of claims.
3. Using your model from exercise 2, make predictions for the 20% of records you did not use in exercise 2. Form a confidence interval around this prediction. Feel free to use simulation, if that's easier.
4. Using the `tbl_medmal` data set from Chapter 13, construct a tree-based model to predict the cumulative paid loss for development lag 2. Use only the upper triangle, but feel free to use whatever predictors you like. Compare the prediction from your models to the actual observation in accident year 1997. How does this model compare with any of the ones you constructed in the exercises in Chapter 15?
5. Using the `tbl_term_life_face` data from Chapter 13, construct a single tree, random forest and gradient boosted machine for the face amount purchased. Use the `importance()` function to talk about the results. Which model do you think performs best and why?
6. Consider the `tbl_term_life_purchase` data from Chapter 13. Using `term_purchased` as the response variable, which single column will lead to minimal entropy?
7. Carry out a similar exercise for `tbl_severity` from Chapter 13, but this time find the three columns which are most effective in dividing the data. What implications does this have for designing a rating plan?