© Thomas Mailund 2022

T. Mailund, *Beginning Data Science in R 4*

https://doi.org/10.1007/978-1-4842-8155-0_6

6. Supervised Learning

Thomas Mailund[1]

(1) Aarhus, Denmark

This chapter and the next concern the mathematical modelling of data that is the essential core of data science. We can call this statistics, or we can call it machine learning. At its heart, it is the same thing. It is all about extracting information out of data.

Machine Learning

Machine learning is the discipline of developing and applying models and algorithms for learning from data. Traditional algorithms implement fixed rules for solving particular problems like sorting numbers or finding the shortest route between two cities. To develop algorithms like that, you need a deep understanding of the problem you are trying to solve—a thorough understanding that you can rarely obtain unless the problem is particularly simple or you have abstracted away all the unusual cases. Far more often, you can collect examples of good or bad solutions to the problem you want to solve without being able to explain precisely why a given solution is good or bad. Or you can obtain data that provides examples of relationships between data you are interested in without necessarily understanding the underlying reasons for these relationships.

This is where machine learning can help. Machine learning concerns learning from data. You do not explicitly develop an algorithm for solving a particular problem. Instead, you use a generic learning algorithm that you feed examples of solutions to and let it learn how to solve the problem from those examples.

This might sound very abstract, but most statistical modelling is indeed examples of this. Take, for example, a linear model $y = \alpha x + \beta + \epsilon$ where ϵ is the stochastic noise (usually assumed to be normal distributed). When you want to model a linear relationship between x and y, you don't figure out α and β from the first principle. You can write an algorithm for sorting numbers without having studied the numbers beforehand, but you cannot usually figure out what the linear relationship is between y and x without looking at data. When you fit the linear model, you are doing machine learning. (Well, I suppose if you do it by hand, it isn't machine learning, but you are not likely to fit linear models by hand that often.) People typically do not call simple models like linear regression machine learning, but that is mostly because the term "machine learning" is much younger than these models. Linear regression is as much machine learning as neural networks are.

Supervised Learning

Supervised learning is used when you have variables you want to predict using other variables—situations like linear regression where you have some input variables, for example, x, and you want a model that predicts output (or response) variables, $y = f(x)$.

Unsupervised learning, the topic for the next chapter, is instead concerned with discovering patterns in data when you don't necessarily know what kind of questions you are interested in learning–when you don't have x and y values and want to know how they are related, but instead have a collection of data, and you want to discover what patterns there are in the data.

For the simplest case of supervised learning, we have one response variable, y, and one input variable, x, and we want to figure out a function, f, mapping input to output, that is, such that $y = f(x)$. What we have to work with is example data of matching x and y. Let us write that as vectors $\mathrm{x} = (x_{\mathbf{1}}, \ldots, x_{\mathbf{n}})$ and $\mathrm{y} = (y_{\mathbf{1}}, \ldots, y_{\mathbf{n}})$ where we want to figure out a function f such that $y_{\boldsymbol{i}} = f(x_{\boldsymbol{i}})$.

We will typically accept that there might be some noise in our observations, so f doesn't map perfectly from x to y. Therefore, we can change the setup slightly and assume that the data we have is x = $(x_1, \dots, x_n)$ and t = $(t_1, \dots, t_n)$, where t is target values and where $t_i = y_i + \epsilon_i$, $y_i = f(x_i)$, and ϵi is the error in the observation t_i.

How we model the error ϵ_i and the function f are choices that are up to us. It is only modelling, after all, and we can do whatever we want. Not all models are equally good, of course, so we need to be a little careful with what we choose and how we evaluate if the choice is good or bad, but in principle, we can do anything.

The way most machine learning works is that an algorithm, implicitly or explicitly, defines a class of parameterized functions $f(-;\theta)$, each mapping input to output $f(-;\theta) : x \mapsto f(x;\theta) = y(\theta)$ (now the value we get for the output depends on the parameters of the function, θ), and the learning consists of choosing parameters θ such that we minimize the errors, that is, such that $f(x_i;\theta)$ is as close to ti as we can get. We want to get close for all our data points, or at least get close on average, so if we let y(θ) denote the vector $(y(\theta)_1, \dots, y(\theta)_n) = (f(x_1;\theta), \dots, f(x_n;\theta))$, we want to minimize the distance from y(θ) to t, $||\mathrm{y}(\theta) - \mathrm{t}||$, for some distance measure $||\cdot||$.

Regression vs. Classification

There are two main types of supervised learning: regression and classification. Regression is used when the output variable we try to target is a number. Classification is used when we try to target some categorical variables.

Take linear regression, $y = \alpha x + \beta$ (or $t = \alpha x + \beta + \epsilon$). It is regression because the variable we are trying to target is a number. The parameterized class of functions, f_θ, are all lines. If we let $\theta = (\theta_1, \theta_0)$ and $\alpha = \theta_1$, $\beta = \theta_0$, then $y(\theta) = f(x;\theta) = \theta_1 x + \theta_0$. Fitting a linear model consists of finding the best θ, where best is defined as the θ that gets y(θ) closest to t. The distance measure used in linear regression is the squared Euclidean distance $\mid \mathrm{y}(\theta) - \mathrm{t} \mid 2 = \sum_{i=1}^{n} (yi(\theta) - ti)2$.

The reason it is the squared distance instead of just the distance is mostly mathematical convenience—it is easier to maximize θ that way—but also related to us interpreting the error term ϵ as normal distributed. Whenever you fit data in linear regression, you are minimizing this distance; you are finding the parameters θ that best fit the data in the sense of minimizing the distance from y(θ) to t:

$$\hat{\theta} = \underset{\theta_1 \quad ,\theta_0}{\operatorname{argmin}} \sum_{i=1}^{n} (\theta_1 x_i + \theta_0 - t_i)^2.$$

For an example of classification, let us assume that the targets t_i are binary, encoded as 0 and 1, but that the input variables x_i are still real numbers. A common way of defining the mapping function $f(-;\theta)$ is to let it map x to the unit interval [0, 1] and interpret the resulting $y(\theta)$ as the probability that t is 1. In a classification setting, you would then predict 0 if $f(x; \theta) < 0.5$ and predict 1 if $f(x; \theta) > 0.5$ (and have some strategy for dealing with $f(x; \theta) = 0.5$). In linear classification, the function f_θ could look like this:

$$f(x; \theta) = \sigma(\theta_1 x + \theta_0)$$

where σ is a sigmoid function (a function mapping $\mathbb{R} \mapsto [0, 1]$ that is "S-shaped"). A common choice of σ is the logistic function $\sigma : z \mapsto \frac{1}{1+e-z}$, in which case we call the fitting of $f(-;\theta)$ logistic regression.

Whether we are doing regression or classification, and whether we have linear models or not, we are simply trying to find parameters θ such that our predictions y(θ) are as close to our targets t as possible. The details that differ between different machine learning methods are how the class of prediction functions $f(-;\theta)$ is defined, what kind of parameters θ we have, and how we measure the distance between y(θ) and t. There are a lot of different choices here and a lot of different machine learning algorithms. Many of them are already implemented in R, however, so we rarely will have to implement our own. We just need to find the right package that implements the learning algorithms we need.

Inference vs. Prediction

A question always worth considering when we fit parameters of a model is this: Do we care about the model parameters or do we just want to make a function that is good at predicting?

If you were taught statistics the same way I was, your introduction to linear regression was mostly focused on the model parameters. You inferred the parameters θ_1 and θ_0 mostly to figure out if $\theta_1 \neq 0$, that is, to find out if there was a (linear) relationship between x and y or not. When we fit our function to data to learn about the parameters, we say we are doing inference, and we are inferring the parameters.

This focus on model parameters makes sense in many situations. In a linear model, the coefficient θ_1 tells us if there is a significant correlation between x and y, meaning we are statistically relatively certain that the correlation exists, and whether it is substantial, meaning that θ_1 is large enough to care about in practical situations.

When we care about model parameters, we usually want to know more than just the best-fitting parameters, $\hat{\theta}$. We want to know how certain we are that the "true parameters" are close to our estimated parameters. This usually means estimating not just the best parameters but also confidence intervals or posterior distributions of parameters. How easy it is to estimate these depends very much on the models and algorithms used.

I put "true parameters" in quotes earlier, where I talked about how close estimates were to the "true parameters," for a good reason. True parameters only exist if the data you are analyzing were simulated from a function $f\theta$ where some true θ exist. When you are estimating parameters, $\hat{\theta}$, you are looking for the best choice of parameters assuming that the data were generated by a function f_θ . Outside of statistics textbooks, there is no reason to think that your data was generated from a function in the class of functions you consider. Unless we are trying to model causal relationships—modelling how we think the world actually works as forces of nature—that is usually not an underlying assumption of model fitting. A lot of the theory we have for doing statistics on inferred parameters does assume that we have the right class of functions, and that is where you get confidence intervals and such from. In practice, data does not come from these sorts of functions, so treat the results you get from theory with some skepticism.

We can get more empirical distributions of parameters directly from data if we have a lot of data—which we usually do have when doing data science—using sampling methods. I will briefly return to that later in this chapter.

We don't always care about the model parameters, though. For linear regression, it is easy to interpret what the parameters mean, but in many machine learning models, the parameters aren't that interpretable—and we don't really care about them. All we care about is if the model we have fitted is good at predicting the target values. Evaluating how well we expect a function to be able to predict is also something that we sometimes have theoretical results regarding, but as for parameter estimation, we shouldn't really trust these too much. It is much better to use the actual data to estimate this, and as for getting empirical distributions of model parameters, it is something we return to later.

Whether you care about model parameters or not depends on your application and quite often on how you think your model relates to reality.

Specifying Models

The general pattern for specifying models in R is using what is called a "formula," which is a type of objects built into the language. The simplest form is `y ~ x` which we should interpret as saying $y = f(x)$. Implicitly, there is assumed some class of functions indexed with model parameters, $f(-;\theta)$, and which class of functions we are working with depends on which R functions we use.

Linear Regression

If we take a simple linear regression, $f_{\mathbf{\theta}}(x) = \theta_1 x + \theta_0$, we need the function `lm()`.

For an example, we can use the built-in data set `cars`, which just contains two variables, speed and breaking distance, where we can consider speed the *x* value and breaking distance the *y* value:

```
cars |> head()
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

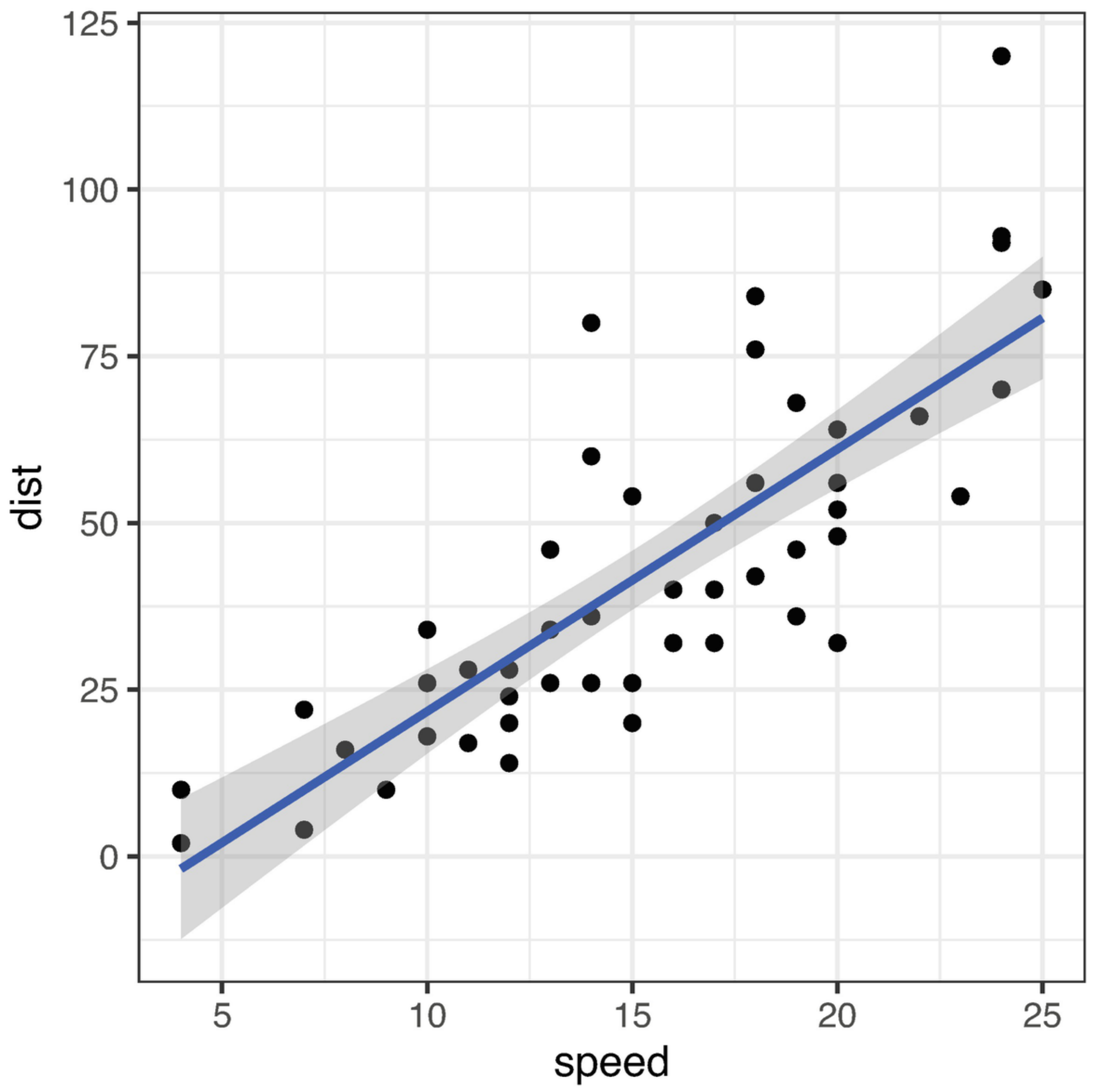

Figure 6-1 Plot of breaking distance vs. speed for cars

If we plot the data set (see Figure 6-1), we see that there is a very clear linear relationship between speed and distance:

```
cars |> ggplot(aes(x = speed, y = dist)) +
   geom_point() +
   geom_smooth(formula = y ~ x, method = "lm")
```

In this plot, I used the method `"lm"` for the smoothed statistics to see the fit. By default, the `geom_smooth()` function would have given us a loess curve, but since we are interested in linear fits, we tell it to use the `lm` method. By default, `geom_smooth()` will also plot the uncertainty of the fit. This is the gray area in the plot. This is the area where the line is likely to be (assuming that the data is generated by a linear model). Do not confuse this with where data points are likely to be, though. If target values are given by $t = \theta_1 x + \theta_0 + \epsilon$ where ϵ has a very large variance, then even if we knew θ_1 and θ_0 with high certainty, we still wouldn't be able to predict with high accuracy where any individual point would fall. There is a difference between prediction accuracy and inference accuracy. We might know model parameters with very high accuracy without being able to predict very well. We might also be able to predict very well without knowing all model parameters that well. If a given model parameter has little influence on where target variables fall, then the training data gives us little information about that parameter. This usually doesn't happen unless the model is more complicated than it needs to be, though, since we often want to remove parameters that do not affect the data.

To actually fit the data and get information about the fit, we use the `lm()` function with the model specification, `dist ~ speed`, and we can use the `summary()` function to see information about the fit:[1]

```
cars %>% lm(dist ~ speed, data = .) %>% summary()
##
## Call:
## lm(formula = dist ~ speed, data = .)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123
## speed         3.9324     0.4155   9.464 1.49e-12
##
## (Intercept) *
## speed       ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

or we can use the `coefficients()` function to get the point estimates and the `confint()` function to get the confidence intervals for the parameters:

```
cars %>% lm(dist ~ speed, data = .) %>% coefficients()
## (Intercept)       speed
##  -17.579095    3.932409
cars %>% lm(dist ~ speed, data = .) %>% confint()
##                  2.5 %    97.5 %
## (Intercept) -31.167850 -3.990340
## speed         3.096964  4.767853
```

Here, (Intercept) is θ_0 and speed is θ_1.

To illustrate the fitting procedure and really drive the point home, we can explicitly draw models with different parameters, that is, draw lines with different choices of θ. To simplify matters, I am going to set $\theta_0 = 0$. Then I can plot the lines $y = \theta_1 x$ for different choices of θ_1 and visually see the fit; see Figure 6-2.

```
predict_dist <- function(speed, theta_1)
    data.frame(speed = speed,
    dist = theta_1 * speed,
    theta = as.factor(theta_1))
cars %>% ggplot(aes(x = speed, y = dist, colour = theta)) +
    geom_point(colour = "black") +
    geom_line(data = predict_dist(cars$speed, 2)) +
    geom_line(data = predict_dist(cars$speed, 3)) +
    geom_line(data = predict_dist(cars$speed, 4)) +
    scale_color_discrete(name=expression(theta[1]))
```

In this plot, I want to color the lines according to their θ_1 parameter, but since the `cars` data frame doesn't have a `theta` column, I'm in a bit of a pickle. I specify that `theta` should determine the color anyway, but when I plot the points, I overwrite this and say that these should be plotted black; then it doesn't matter that I didn't have `theta` values. In the `geom_line()` calls, where I plot the lines, I do have a theta, and that will determine the colors for the lines. The lines are plotted according to their theta value which I set in the `predict_dist()` function.

Each of the lines shows a choice of model. Given an input value x, they all produce an output value $y(\theta) = f(x; \theta)$. So we can fix θ and consider the mapping $x \mapsto \theta_1 x$. This is the function we use when predicting the output for a given value of *x*. If we fix *x* instead, we can also see it as a function of θ: $\theta_1 \mapsto \theta_1 x$. This is what we use when we fit parameters to the data, because if we keep our data set fixed, this mapping defines an error function, that is, a function that given parameters gives us a measure of how far our predicted values are from our target values. If, as before, our input values and target values are vectors x and t, then the error function is

$$Ex, t\ \ (\theta i) = \sum_{i=1}^{n} (\theta 1 xi - ti)\, 2$$

and we can plot the errors against different choices of θ_1 (Figure 6-3). Where this function is minimized, we find our best estimate for θ_1:

```
# Get the error value for the specific theta
fitting_error <- Vectorize(function(theta)
   sum((theta * cars$speed - cars$dist)**2)
)
# Plot the errors for a range of thetas
tibble(theta = seq(0, 5, length.out = 50)) |> # set the theta
values
   mutate(errors = fitting_error(theta)) |> # add the errors
   ggplot(aes(x = theta, y = errors)) +
   geom_line() +
   xlab(expression(theta[1])) + ylab(expression(E(theta[1])))
```

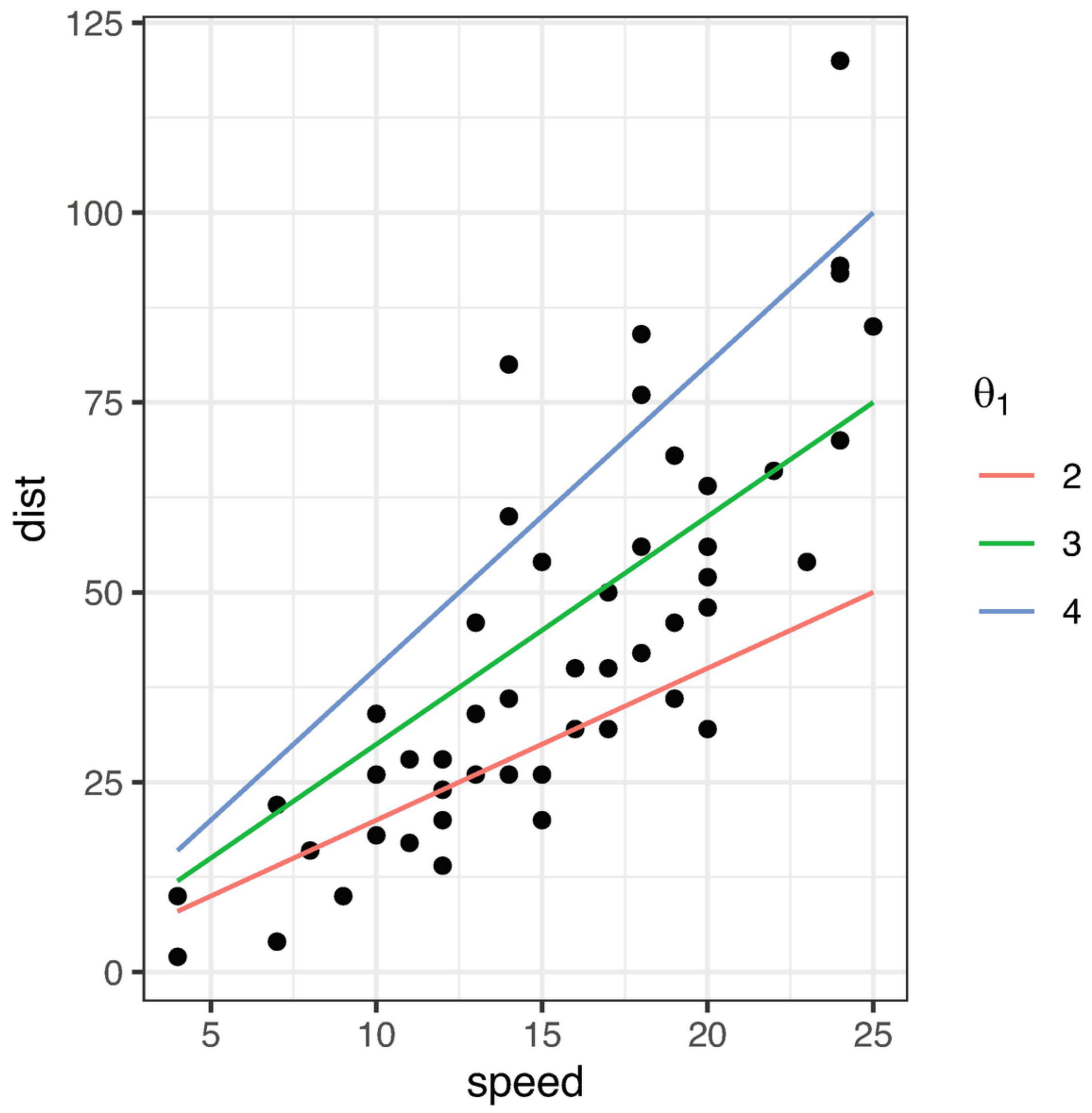

Figure 6-2 Prediction lines for different choices of parameters

To wrap up this example, we can also plot and fit the best model where $\theta_0 = 0$. The formula needed to remove the intercept is of the form "`y ~ x - 1`". It is the "`- 1`" that removes the intercept:

```
cars %>% lm(dist ~ speed - 1, data = .) %>% coefficients()
##    speed
## 2.909132
```

We can also plot this regression line, together with the confidence interval for where it lies, using `geom_smooth()`. See Figure **6-4**. Here, though, we need to use the formula `y ~ x - 1` rather than `dist ~ speed - 1`. This is because the `geom_smooth()` function works on the `ggplot2` layers that have `x` and `y` coordinates and not the data in the data frame as such. We map the `speed` variable to the `x`-axis and the `dist` variable to the `y` variable in the aesthetics, but it is `x` and `y` that `geom_smooth()` works on:

```
cars |> ggplot(aes(x = speed, y = dist)) +
   geom_point() +
   geom_smooth(method = "lm", formula = y ~ x - 1)
```

Logistic Regression (Classification, Really)

Using other statistical models works the same way. We specify the class of functions, $f_{\boldsymbol{\theta}}$, using a formula and use a function to fit its parameters. Consider binary classification and logistic regression.

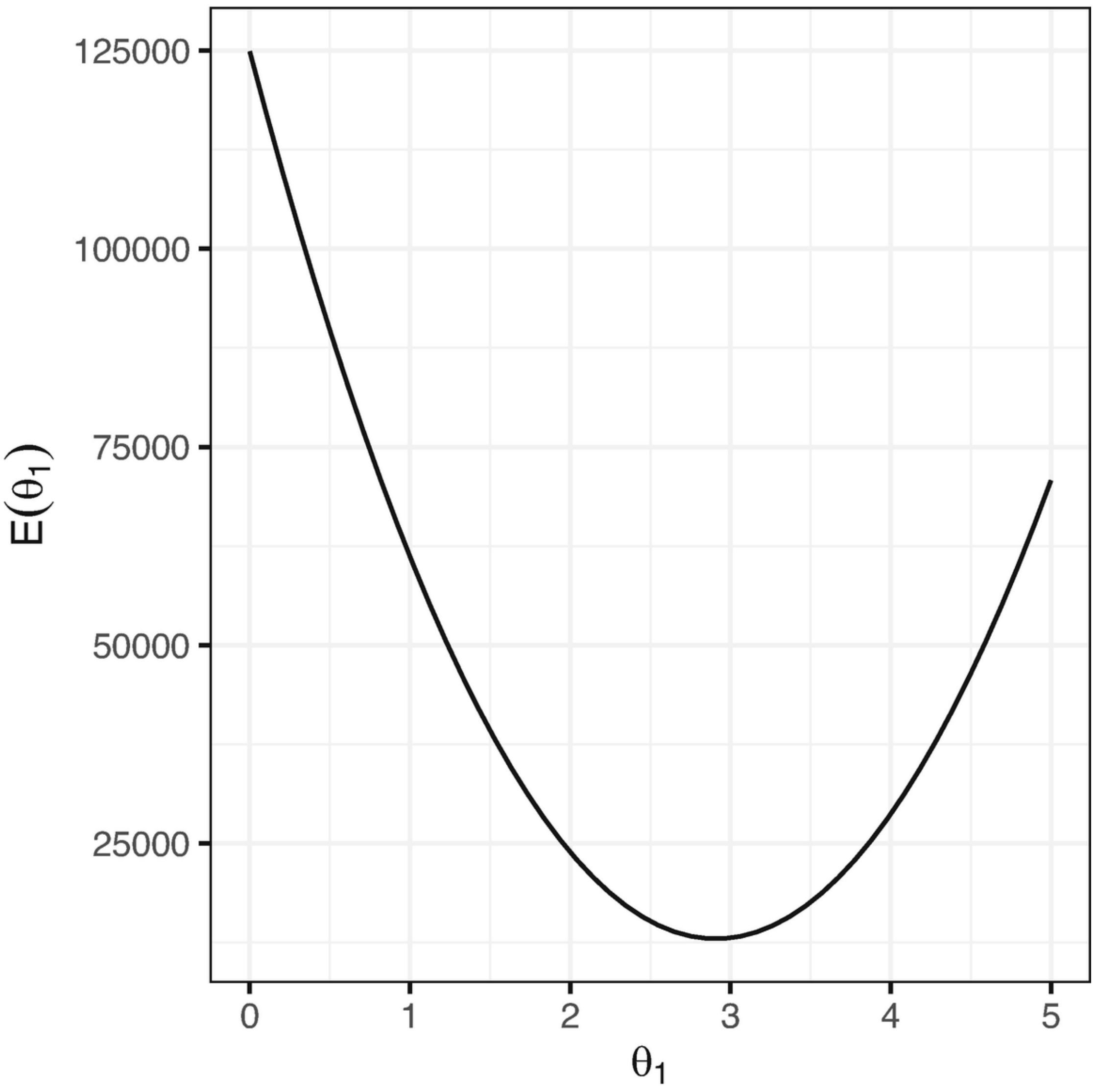

Figure 6-3 Error values for different choices of parameters

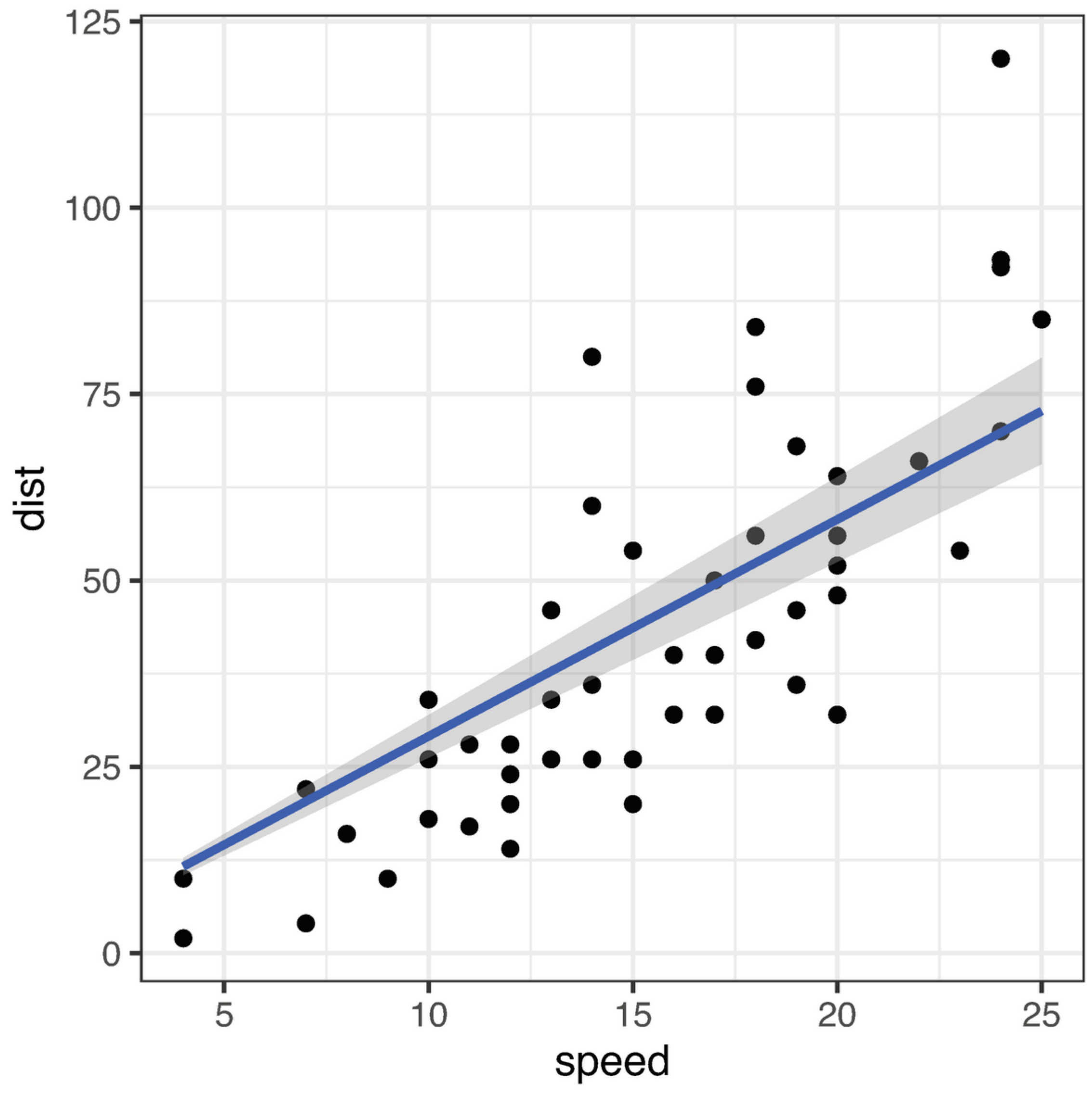

Figure 6-4 Best regression line going through (0,0)

Here, we can use the breast cancer data from the `mlbench` library that we also discussed in Chapter **3** and ask if the clump thickness has an effect on the risk of a tumor being malignant. That is, we want to see if we can predict the `Class` variable from the `Cl.thickness` variable:

```
library(mlbench)
data("BreastCancer")
BreastCancer |> head()
##        Id Cl.thickness Cell.size Cell.shape
## 1 1000025            5         1          1
## 2 1002945            5         4          4
## 3 1015425            3         1          1
## 4 1016277            6         8          8
## 5 1017023            4         1          1
## 6 1017122            8        10         10
##   Marg.adhesion Epith.c.size Bare.nuclei
## 1            1           2          1
## 2            5           7         10
## 3            1           2          2
## 4            1           3          4
## 5            3           2          1
## 6            8           7         10
##   Bl.cromatin Normal.nucleoli Mitoses     Class
## 1           3               1       1    benign
## 2           3               2       1    benign
## 3           3               1       1    benign
## 4           3               7       1    benign
## 5           3               1       1    benign
## 6           9               7       1 malignant
```

We can plot the data against the fit; see Figure 6-5. Since the malignant status is either 0 or 1, the points would overlap, but if we add a little jitter to the plot, we can still see them, and if we make them slightly transparent, we can see the density of the points.

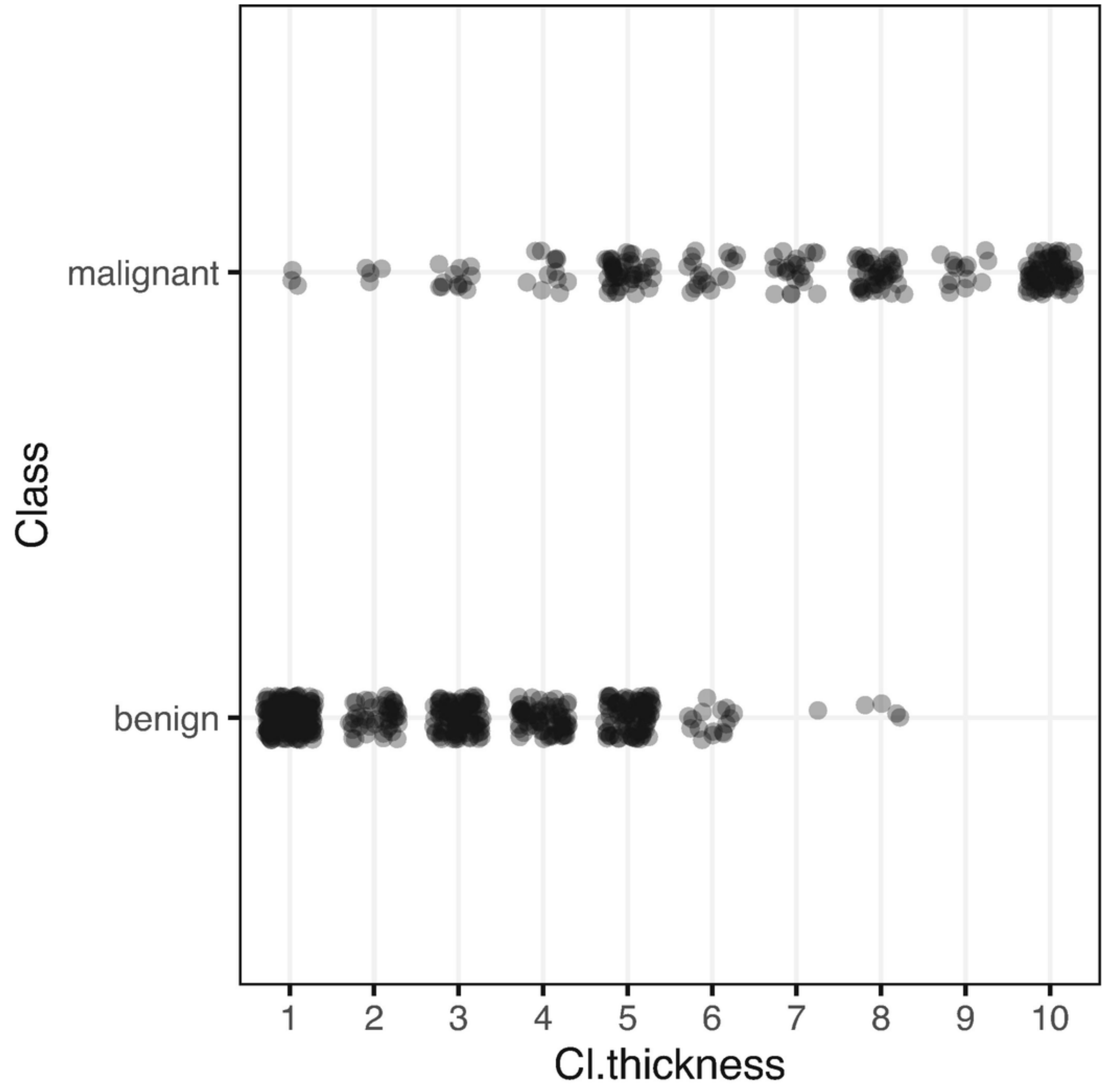

Figure 6-5 Breast cancer class vs. clump thickness

```
BreastCancer |>
   ggplot(aes(x = Cl.thickness, y = Class)) +
   geom_jitter(height = 0.05, width = 0.3, alpha = 0.4)
```

For classification, we still specify the prediction function $y = f(x)$ using the formula `y ~ x`. The outcome parameter for `y ~ x` is just binary now. To fit a logistic regression, we need to use the `glm()` function (generalized linear model) with the `family` set to `"binomial"`. This specifies that we use the logistic function to map from the linear space of x and θ to the unit interval. Aside from that, fitting and getting results are very similar.

We cannot directly fit the breast cancer data with logistic regression, though. There are two problems. The first is that the breast cancer data set considers the clump thickness ordered factors, but for logistic regression, we need the input variable to be numeric. While, generally, it is not advisable to directly translate categorical data into numeric data, judging from the plot it seems okay in this case.

Using the function `as.numeric()` will do this, but remember that this is a risky approach when working with factors! It actually would work for this data set, but we will use the safer approach of first translating the factor into strings and then into numbers.

The second problem is that the `glm()` function expects the response variable to be numerical, coding the classes like 0 or 1, while the `BreastCancer` data again encodes the classes as a factor. Generally, it varies a little from algorithm to algorithm whether a factor or a numerical encoding is expected for classification, so you always need to check the documentation for that, but in any case, it is simple enough to translate between the two representations.

We can translate the input variable to numerical values and the response variable to 0 and 1 and plot the data together with a fitted model; see Figure 6-6. For the `geom_smooth()` function, we specify that the method is `glm` and that the family is `binomial`. To specify the family, we need to pass this argument on to the smoothing method, and that is done by giving the parameter `method.args` a list of named parameters; here, we give it `list(family = "binomial)`:

```
BreastCancer |>
   mutate(Thickness =
               as.numeric(as.character(Cl.thickness))) |>
   mutate(Malignant = ifelse(Class != "benign", 1, 0)) |>
   ggplot(aes(x = Thickness, y = Malignant)) +
   geom_jitter(height = 0.05, width = 0.3, alpha = 0.4) +
   geom_smooth(method = "glm", formula = y ~ x,
               method.args = list(family = "binomial"))
```

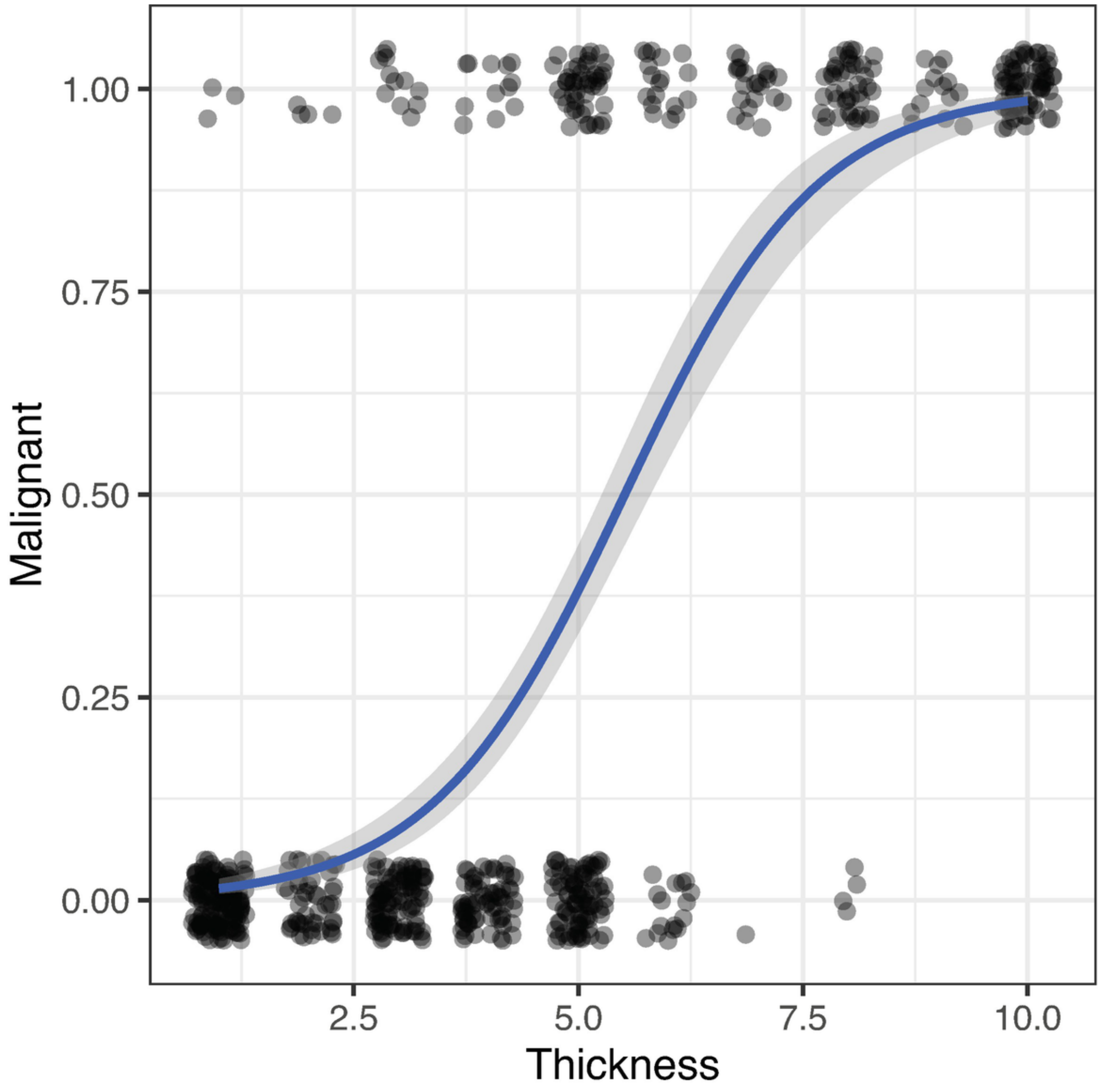

Figure 6-6 Logistic regression fit to breast cancer data

To get the fitted object, we use `glm()` like we used `lm()` for the linear regression:[2]

```
BreastCancer %>%
   mutate(Thickness =
              as.numeric(as.character(Cl.thickness))) %>%
   mutate(Malignant = ifelse(Class != "benign", 1, 0)) %>%
   glm(Malignant ~ Thickness,
       family = "binomial",
      data = .)
##
## Call: glm(formula = Malignant ~ Thickness, family =
"binomial", data = .)
##
## Coefficients:
## (Intercept)   Thickness
##     -5.1602      0.9355
##
## Degrees of Freedom: 698 Total (i.e. Null);   697 Residual
## Null Deviance:          900.5
## Residual Deviance: 464.1    AIC: 468.1
```

Model Matrices and Formula

Most statistical models and machine learning algorithms actually create a map not from a single value, $f(-;\theta) : x \mapsto y$, but from a vector, $f(-;\theta) : \mathrm{x} \mapsto y$. When we fit a line for single x and y values, we are actually also working with fitting a vector because we have both the x values and the intercept to fit. That is why the model has two parameters, $\theta_\mathbf{0}$ and $\theta_\mathbf{1}$. For each x value, we are really using the vector $(1, x)$ where the 1 is used to fit the intercept.

We shouldn't confuse this with the vector we have as input to the model fitting, though. If we have data (x, t) to fit, then we already have a vector for our input data. But what the linear model actually sees is a matrix for x, so let us call that X. This matrix, known as the model matrix, has a row per value in x, and it has two columns, one for the intercept and one for the x values:

$$X = \begin{bmatrix} 1 & x1 \\ 1 & x2 \\ 1 & x3 \\ \vdots & \vdots \\ 1 & xn \end{bmatrix}$$

We can see what model matrix R generates for a given data set and formula using the `model.matrix()` function. For the `cars` data, if we wish to fit `dist` vs. `speed`, we get this:

```
cars %>%
   model.matrix(dist ~ speed, data = .) %>%
   head(5)
##  (Intercept) speed
## 1           1     4
## 2           1     4
## 3           1     7
## 4           1     7
## 5           1     8
```

If we remove the intercept, we simply get this:

```
cars %>%
   model.matrix(dist ~ speed - 1, data = .) %>%
   head(5)
##   speed
## 1     4
## 2     4
## 3     7
## 4     7
## 5     8
```

Pretty much all learning algorithms work on a model matrix, so, in R, they are implemented to take a formula for specifying the model and then building the model matrix from that and the input data.

For linear regression, the map is a pretty simple one. If we let the parameters $\theta^T = (\theta_0, \theta_1)$, then it is just multiplying that with the model matrix, X:

$$X \cdot \theta = \begin{bmatrix} 1 & x1 \\ 1 & x2 \\ 1 & x3 \\ \vdots & \vdots \\ 1 & xn \end{bmatrix} \cdot \begin{bmatrix} \theta 0 \\ \theta 1 \end{bmatrix} = \begin{bmatrix} \theta 0 & + & \theta 1 x1 \\ \theta 0 & + & \theta 1 x2 \\ \theta 0 & + & \theta 1 x3 \\ & \vdots & \\ \theta 0 & + & \theta 1 xn \end{bmatrix}$$

This combination of formulas and model matrices is a powerful tool for specifying models. Since all the algorithms we use for fitting data work on model matrices anyway, there is no reason to hold back on how complex formulas to give them. The formulas will just be translated into model matrices anyhow, and they can all deal with them.

If you want to fit more than one parameter, no problem. You just write `y ~ x + z`, and the model matrix will have three columns:

$$X = \begin{bmatrix} 1 & x1 & z1 \\ 1 & x2 & z2 \\ 1 & x3 & z3 \\ \vdots & \vdots & \vdots \\ 1 & xn & zn \end{bmatrix}$$

Our model fitting functions are just as happy to fit this model matrix as the one we get from just a single variable.

So if we wanted to fit the breast cancer data to both cell thickness and cell size, we can do that just by adding both explanatory variables in the formula:

```
BreastCancer %>%
    mutate(Thickness =
              as.numeric(as.character(Cl.thickness)),
           CellSize =
              as.numeric(as.character(Cell.size))) %>%
    mutate(Malignant = ifelse(Class != "benign", 1, 0)) %>%
    model.matrix(Malignant ~ Thickness + CellSize,
                 data = .) %>%
    head(5)
##  (Intercept) Thickness CellSize
## 1          1         5        1
## 2          1         5        4
## 3          1         3        1
## 4          1         6        8
## 5          1         4        1
```

The generalized linear model fitting function will happily work with that:

```
BreastCancer %>%
    mutate(Thickness =
               as.numeric(as.character(Cl.thickness)),
           CellSize =
               as.numeric(as.character(Cell.size))) %>%
    mutate(Malignant = ifelse(Class != "benign", 1, 0)) %>%
    glm(Malignant ~ Thickness + CellSize, family =
"binomial", data = .)
##
## Call: glm(formula = Malignant ~ Thickness + CellSize,
family = "binomial",
##    data = .)
##
## Coefficients:
## (Intercept)    Thickness    CellSize
##     -7.1517       0.6174      1.1751
##
## Degrees of Freedom: 698 Total (i.e. Null);    696 Residual
## Null Deviance:         900.5
## Residual Deviance: 212.3    AIC: 218.3
```

Translating data into model matrices also works for factors; they are just represented as a binary vector for each level:

```
BreastCancer %>%
    mutate(Malignant = ifelse(Class != "benign", 1, 0)) %>%
    model.matrix(Malignant ~ Bare.nuclei, data = .) %>%
    head(5)
##  (Intercept) Bare.nuclei2 Bare.nuclei3
## 1           1            0            0
## 2           1            0            0
## 3           1            1            0
## 4           1            0            0
## 5           1            0            0
##  Bare.nuclei4 Bare.nuclei5 Bare.nuclei6
## 1            0            0            0
## 2            0            0            0
## 3            0            0            0
## 4            1            0            0
## 5            0            0            0
##  Bare.nuclei7 Bare.nuclei8 Bare.nuclei9
## 1            0            0            0
## 2            0            0            0
## 3            0            0            0
## 4            0            0            0
## 5            0            0            0
##  Bare.nuclei10
## 1             0
## 2             1
## 3             0
## 4             0
## 5             0
```

The translation for ordered factors gets a little more complicated, but R will happily do it for you:

```
BreastCancer %>%
   mutate(Malignant = ifelse(Class != "benign", 1, 0)) %>%
   model.matrix(Malignant ~ Cl.thickness, data = .) %>%
   head(5)
##  (Intercept) Cl.thickness.L Cl.thickness.Q
## 1           1    -0.05504819    -0.34815531
## 2           1    -0.05504819    -0.34815531
## 3           1    -0.27524094    -0.08703883
## 4           1     0.05504819    -0.34815531
## 5           1    -0.16514456    -0.26111648
##  Cl.thickness.C Cl.thickness^4 Cl.thickness^5
## 1     0.1295501     0.33658092    -0.21483446
## 2     0.1295501     0.33658092    -0.21483446
## 3     0.3778543    -0.31788198    -0.03580574
## 4    -0.1295501     0.33658092     0.21483446
## 5     0.3346710     0.05609682    -0.39386318
##  Cl.thickness^6 Cl.thickness^7 Cl.thickness^8
## 1    -0.3113996      0.3278724      0.2617852
## 2    -0.3113996      0.3278724      0.2617852
## 3     0.3892495     -0.5035184      0.3739788
## 4    -0.3113996     -0.3278724      0.2617852
## 5     0.2335497      0.2459043     -0.5235703
##  Cl.thickness^9
## 1    -0.5714300
## 2    -0.5714300
## 3    -0.1632657
## 4     0.5714300
## 5     0.3809534
```

If you want to include interactions between your parameters, you specify that using `*` instead of `+`:

```
BreastCancer %>%
   mutate(Thickness =
              as.numeric(as.character(Cl.thickness)),
          CellSize =
              as.numeric(as.character(Cell.size))) %>%
    mutate(Malignant = ifelse(Class != "benign", 1, 0)) %>%
    model.matrix(Malignant ~ Thickness * CellSize,
                 data = .) %>%
    head(5)
##   (Intercept) Thickness CellSize
## 1           1         5        1
## 2           1         5        4
## 3           1         3        1
## 4           1         6        8
## 5           1         4        1
##   Thickness:CellSize
## 1                  5
## 2                 20
## 3                  3
## 4                 48
## 5                  4
```

How interactions are modelled depends a little bit on whether your parameters are factors or numeric, but for numeric values, the model matrix will just contain a new column with the two values multiplied. For factors, you will get a new column for each level of the factor:

```
BreastCancer %>%
   mutate(Thickness =
            as.numeric(as.character(Cl.thickness))) %>%
   mutate(Malignant = ifelse(Class != "benign", 1, 0)) %>%
   model.matrix(Malignant ~ Thickness * Bare.nuclei, data =
.) %>%
   head(3)
##   (Intercept) Thickness Bare.nuclei2 Bare.nuclei3
## 1           1         5            0            0
## 2           1         5            0            0
## 3           1         3            1            0
##   Bare.nuclei4 Bare.nuclei5 Bare.nuclei6
## 1            0            0            0
## 2            0            0            0
## 3            0            0            0
##   Bare.nuclei7 Bare.nuclei8 Bare.nuclei9
## 1            0            0            0
## 2            0            0            0
## 3            0            0            0
##   Bare.nuclei10 Thickness:Bare.nuclei2
## 1             0                      0
## 2             1                      0
## 3             0                      3
##   Thickness:Bare.nuclei3 Thickness:Bare.nuclei4
## 1                      0                      0
## 2                      0                      0
## 3                      0                      0
##   Thickness:Bare.nuclei5 Thickness:Bare.nuclei6
## 1                      0                      0
## 2                      0                      0
## 3                      0                      0
##   Thickness:Bare.nuclei7 Thickness:Bare.nuclei8
## 1                      0                      0
## 2                      0                      0
## 3                      0                      0
##   Thickness:Bare.nuclei9 Thickness:Bare.nuclei10
## 1                      0                       0
## 2                      0                       5
## 3                      0                       0
```

The interaction columns all have `:` in their name, and you can specify an interaction term directly by writing that in the model formula as well:

```
BreastCancer %>%
    mutate(Thickness =
             as.numeric(as.character(Cl.thickness))) %>%
    mutate(Malignant = ifelse(Class != "benign", 1, 0)) %>%
    model.matrix(Malignant ~ Thickness : Bare.nuclei, data =
.) %>%
    head(3)
##   (Intercept) Thickness:Bare.nuclei1
## 1           1                      5
## 2           1                      0
## 3           1                      0
##   Thickness:Bare.nuclei2 Thickness:Bare.nuclei3
## 1                      0                      0
## 2                      0                      0
## 3                      3                      0
##   Thickness:Bare.nuclei4 Thickness:Bare.nuclei5
## 1                      0                      0
## 2                      0                      0
## 3                      0                      0
##   Thickness:Bare.nuclei6 Thickness:Bare.nuclei7
## 1                      0                      0
## 2                      0                      0
## 3                      0                      0
##   Thickness:Bare.nuclei8 Thickness:Bare.nuclei9
## 1                      0                      0
## 2                      0                      0
## 3                      0                      0
##   Thickness:Bare.nuclei10
## 1                       0
## 2                       5
## 3                       0
```

If you want to use all the variables in your data except the response variable, you can even use the formula `y ~ .` where the `.` will give you all parameters in your data except `y`.

Using formulas and model matrices also means that we do not have to use our data raw. We can transform it before we give it to our learning algorithms. In general, we can transform our data using any function ϕ. It is traditionally called phi because we call what it produces features of our data, and the point of it is to pull out the relevant features of the data to give to the learning algorithm. It usually maps from vectors to vectors, so you can use it to transform each row in your raw data into the rows of the model matrix which we will then call Φ instead of X:

$$\Phi = \begin{bmatrix} - & \phi(x1) & - \\ - & \phi(x2) & - \\ - & \phi(x3) & - \\ & \cdots & \\ - & \phi(xn) & - \end{bmatrix}$$

If this sounds very abstract, perhaps it will help to see some examples. We go back to the `cars` data, but this time, we want to fit a polynomial to the data instead of a line. If d denotes breaking distance and s the speed, then we want to fit $d = \theta_0 + \theta_1 s + \theta_1 s^2 + \cdots + \theta_n s^n$. Let us just do $n = 2$, so we want to fit a second-degree polynomial. Don't be confused about the higher degrees of the polynomial, it is still a linear model. The linear in linear model refers to the θ parameters, not the data. We just need to map the single s parameter into a vector with the different polynomial degrees, so 1 for the intercept, s for the linear component, and s^2 for the squared component. So $\phi(s) = (1, s, s^2)$.

We can write that as a formula. There, we don't need to specify the intercept term explicitly—it will be included by default, and if we don't want it, we have to remove it with `-1` in the formula—but we need `speed`, and we need `speed^2`:

```
cars %>%
    model.matrix(dist ~ speed + speed^2, data = .) %>%
    head()
##   (Intercept) speed
## 1           1     4
## 2           1     4
## 3           1     7
## 4           1     7
## 5           1     8
## 6           1     9
```

Now this doesn't quite work—you can see that we only got the intercept and `speed`—and the reason is that multiplication is interpreted as interaction terms even if it is interaction with the parameter itself. And interaction with itself doesn't go into the model matrix because that would just be silly.

To avoid that problem, we need to tell R that the `speed^2` term should be interpreted just the way it is. We do that using the identity function, `I()`:

```
cars %>%
    model.matrix(dist ~ speed + I(speed^2), data = .) %>%
    head()
##   (Intercept) speed I(speed^2)
## 1           1     4         16
## 2           1     4         16
## 3           1     7         49
## 4           1     7         49
## 5           1     8         64
## 6           1     9         81
```

Now our model matrix has three columns, which is precisely what we want.

We can fit the polynomial using the linear model function like this:

```
cars %>% lm(dist ~ speed + I(speed^2), data = .) %>%
    summary()
##
## Call:
## lm(formula = dist ~ speed + I(speed^2), data = .)
##
## Residuals:
##     Min     1Q   Median      3Q     Max
## -28.720 -9.184  -3.188   4.628  45.152
##
## Coefficients:
##             Estimate  Std. Error  t value  Pr(>|t|)
## (Intercept)  2.47014    14.81716  0.167       0.868
## speed        0.91329     2.03422  0.449       0.656
## I(speed^2)   0.09996     0.06597  1.515       0.136
##
## Residual standard error: 15.18 on 47 degrees of freedom
## Multiple R-squared: 0.6673, Adjusted R-squared: 0.6532
## F-statistic: 47.14 on 2 and 47 DF, p-value: 5.852e-12
```

or we can plot it like this (see Figure 6-7):

```
cars %>% ggplot(aes(x = speed, y = dist)) +
    geom_point() +
    geom_smooth(method = "lm", formula = y ~ x + I(x^2))
```

This is a slightly better fitting model, but that wasn't the point. You can see how you can transform data in a formula to have different features to give to your fitting algorithms.

Validating Models

How did I know the polynomial fit was better than the linear fit? Well, theoretically a second-degree polynomial should always be a better fit than a line since a line is a special case of a polynomial. We just set θ_2 to zero. If the best-fitted polynomial doesn't have $\theta_2 = 0$, then that is because we can fit the data better if it is not.

The result of fitting the polynomial tells me, in the output from the `summary()` function, that the variables are not significant. It tells me that both from the linear and the squared component, though, so it isn't that useful. Clearly, the points are on a line, so it cannot be correct that there isn't a linear component. I cannot use the summary that much because it is only telling me that when I have both components, then neither of them is statistically significant. That doesn't mean much.

But should I even care, though? If I know that the more complex model always fits better, then shouldn't I just always use it? The problem with that idea is that while the most complex model will always fit the training data—the data I use for fitting the model—better, it will not necessarily generalize better. If I use a high enough degree polynomial—if I have a degree that is the same as the number of data points—I can fit the data perfectly. But it will be fitting both the systematic relationship between x and y and also the statistical errors in our targets t. It might be utterly useless for predicting point number $n + 1$.

What I really need to know is whether one or the other model is better at predicting the distance from the speed.

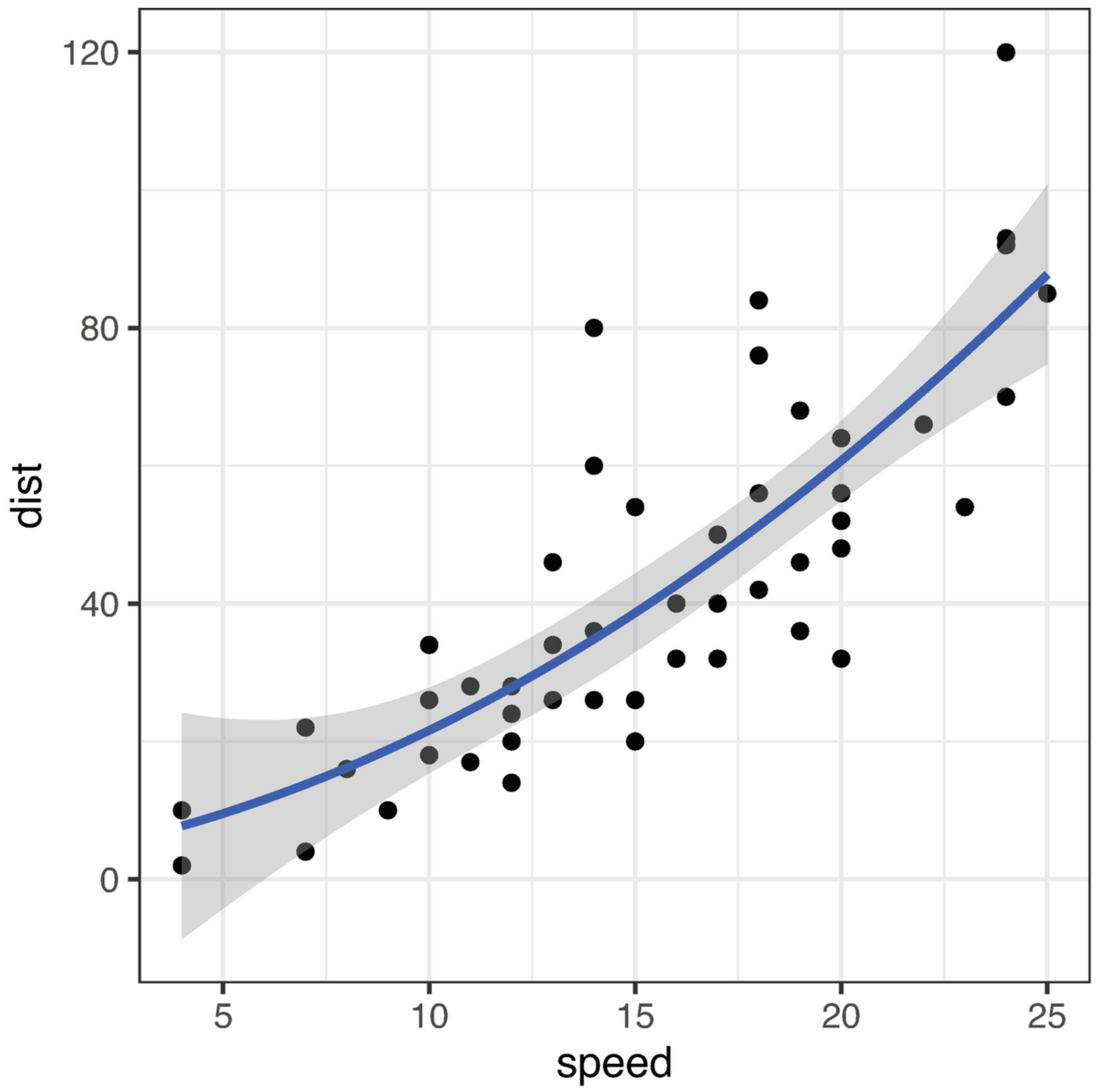

Figure 6-7 The cars data fitted to a second-degree polynomial

We can fit the two models and get their predictions using the `predict()` function. It takes the fitted model as the first argument and data to predict on as the second:

```
line <- cars %>% lm(dist ~ speed, data = .)
poly <- cars %>% lm(dist ~ speed + I(speed^2), data = .)
predict(line, cars) |> head()
##         1         2         3         4         5
## -1.849460 -1.849460  9.947766  9.947766 13.880175
##         6
## 17.812584
predict(poly, cars) |> head()
```

```
##         1         2         3         4         5 
##  7.722637  7.722637 13.761157 13.761157 16.173834 
##         6 
## 18.786430
```

Evaluating Regression Models

To compare the two models, we need a measure of how well they fit. Since both models are fitting the squared distances from predictions to targets, a fair measure would be looking at the mean squared error. The unit of that would be distance squared, though, so we usually use the square root of this mean distance to measure the quality of the predictions, which would give us the errors in the distance unit:

```
rmse <- function(x,t) sqrt(mean(sum((t - x)^2)))
rmse(predict(line, cars), cars$dist)
## [1] 106.5529
rmse(predict(poly, cars), cars$dist)
## [1] 104.0419
```

Now clearly the polynomial fits slightly better, which it should, based on theory, but there is a bit of a cheat here. We are looking at how the models work on the data we used to fit them. The more complex model will always be better at this. That is the problem we are dealing with. The more complex model might be overfitting the data and capturing the statistical noise we don't want it to capture. What we really want to know is how well the models generalize; how well do they work on data they haven't already seen and used to fit their parameters?

We have used all the data we have to fit the models. That is generally a good idea. You want to use all the data available to get the best-fitted model. But to compare models, we need to have data that isn't used in the fitting.

We can split the data into two sets, one we use for training and the other we use to test the models. There are 50 data points, so I can take the first 25 to train my models on and the next 25 to test them on:

```
training_data <- cars[1:25,]
test_data <- cars[26:50,]
line <- training_data %>% lm(dist ~ speed, data = .)
```

```
poly <- training_data %>% lm(dist ~ speed + I(speed^2), data
= .)
rmse(predict(line, test_data), test_data$dist)
## [1] 88.89189
rmse(predict(poly, test_data), test_data$dist)
## [1] 83.84263
```

The second-degree polynomial is still better, but I am also still cheating. There is more structure in my data set than just the speed and distances. The data frame is sorted according to the distance, so the training set has all the short distances and the test data all the long distances. They are not similar. That is not good.

In general, you cannot know if there is such structure in your data. In this particular case, it is easy to see because the structure is that obvious, but sometimes it is more subtle. So when you split your data into training and test data, you will want to sample data points randomly. That gets rid of the structure that is in the order of the data points.

We can use the `sample()` function to sample randomly zeros and ones:

```
sampled_cars <- cars |>
    mutate(training = sample(0:1, nrow(cars), replace =
TRUE))
sampled_cars |> head()
##   speed dist training
## 1     4    2        1
## 2     4   10        0
## 3     7    4        1
## 4     7   22        1
## 5     8   16        0
## 6     9   10        1
```

This doesn't give us 50/50 training and test data since which data point gets into each category will depend on the random samples, but it will be roughly half the data we get for training:

```
training_data <- sampled_cars |> filter(training == 1)
test_data <- sampled_cars |> filter(training == 0)
training_data |> head()
##   speed dist training
## 1     4    2        1
## 2     7    4        1
## 3      7   22         1
## 4      9   10         1
## 5     10   18         1
## 6     10   34         1
test_data |> head()
##   speed dist training
## 1      4   10        0
## 2      8   16        0
## 3     10   26        0
## 4     12   14        0
## 5     12   20        0
## 6     12   24        0
```

Now we can get a better estimate of how the functions are working:

```
line <- training_data %>% lm(dist ~ speed, data = .)
poly <- training_data %>% lm(dist ~ speed + I(speed^2), data
= .)
rmse(predict(line, test_data), test_data$dist)
## [1] 66.06671
rmse(predict(poly, test_data), test_data$dist)
## [1] 65.8426
```

Now, of course, the accuracy scores depend on the random sampling when we create the training and test data, so you might want to use more samples. We will return to that in the next section.

Once you have figured out what the best model is, you will still want to train it on all the data you have. Splitting the data is just a tool for evaluating how well different models work. For the final model you choose to work with, you will always want to fit it with all the data you have.

Evaluating Classification Models

If you want to do classification rather than regression, then the root mean square error is not the function to use to evaluate your model. With classification, you want to know how many data points are classified correctly and how many are not.

As an example, we can take the breast cancer data and fit a model:

```
formatted_data <- BreastCancer |>
```

```
    mutate(Thickness =
              as.numeric(as.character(Cl.thickness)),
           CellSize =
              as.numeric(as.character(Cell.size))) %>%
    mutate(Malignant = ifelse(Class != "benign", 1, 0))
fitted_model <- formatted_data %>%
    glm(Malignant ~ Thickness + CellSize,
        family = "binomial",
        data = .)
```

To get its prediction, we can again use `predict()`, but we will see that for this particular model, the predictions are probabilities of a tumor being malignant:

```
predict(fitted_model, formatted_data, type = "response") |>
head()
##          1          2          3          4
## 0.05266571 0.65374326 0.01591478 0.99740926
##          5          6
## 0.02911157 0.99992795
```

We would need to translate that into actual predictions. The natural choice here is to split the probabilities at 50%. If we are more certain that a tumor is malignant than benign, we will classify it as malignant:

```
classify <- function(probability) ifelse(probability < 0.5,
0, 1) classified_malignant <- classify(predict(fitted_model,
formatted_data))
```

Where you want to put the threshold of how to classify depends on your data and the consequences of the classification. In a clinical situation, maybe you want to examine further a tumor with less than 50% probability that it is malignant, or maybe you don't want to tell patients that a tumor might be malignant if it is only 50% probable. The classification should take into account how sure you are about the classification, and that depends a lot on the situation you are in. Of course, you don't want to bet against the best knowledge you have, so I am not suggesting that you should classify everything below probability 75% as the "false" class, for instance. The only thing you gain from this is making worse predictions than you could. But sometimes you want to leave some data unpredicted. So here you can use the probabilities the model predicts to leave some data points as `NA`. How you want to use that your prediction gives you probabilities instead of just classes—assuming it does, it depends on the algorithm used for classifying—is up to you and the situation you are analyzing.

Confusion Matrix

In any case, if we just put the classification threshold at 50/50, then we can compare the predicted classification against the actual classification using the `table()` function:

```
table(formatted_data$Malignant, classified_malignant)
##    classified_malignant
##       0   1
##   0 447  11
##   1  31 210
```

This table, contrasting predictions against true classes, is known as the confusion matrix. The rows count how many zeros and ones we see in the `formatted_data$Malignant` argument and the columns how many zeros and ones we see in the `classified_malignant` argument. So the first row is where the data says the tumors are not malignant, and the second row is where the data says that the tumors are malignant. The first column is where the predictions say the tumors are not malignant, while the second column is where the predictions say that they are.

This, of course, depends on the order of the arguments to `table()`; it doesn't know which argument contains the data classes and which contains the model predictions. It can be a little hard to remember which dimension, rows or columns, are the predictions, but you can provide a parameter, `dnn` (dimnames names), to make the table remember it for you:

```
table(formatted_data$Malignant, classified_malignant,
      dnn = c("Data", "Predictions"))
##     Predictions
## Data   0   1
##    0 447  11
##    1  31 210
```

The correct predictions are on the diagonal, and the off-diagonal values are where our model predicts incorrectly.

The first row is where the data says that tumors are not malignant. The first element, where the model predicts that the tumor is benign, and the data agrees, is called the true negatives. The element to the right of it, where the model says a tumor is malignant but the data says it is not, is called the false positives.

The second row is where the data says that tumors are malignant. The first column is where the prediction says that it isn't a malignant tumor, and this is called the false negatives. The second column is the cases where both the model and the data say that the tumor is malignant. That is the true positives.

The terms positives and negatives are a bit tricky here. I managed to sneak them past you by having the classes called zeros and ones which you already associate with true and false and positive and negative and by having a data set where it was more natural to think of malignant tumors as being the ones we want to predict.

The classes do not have to be zeros and ones. That was just easier in this particular model where I had to translate the classes into zeros and ones for the logistic classification anyway. But really, the classes are `"benign"` and `"malignant"`:

```
classify <- function(probability)
   ifelse(probability < 0.5, "benign", "malignant")
classified <- classify(predict(fitted_model, formatted_data))
table(formatted_data$Class, classified,
      dnn=c("Data", "Predictions"))
##          Predictions
## Data       benign malignant
## benign        447        11
## malignant      31       210
```

What is positive and what is negative now depends on whether we want to predict malignant or benign tumors. Of course, we really want to predict both well, but the terminology considers one class true and the other false.

The terms carry over into several of the terms used in classification described in the following where the classes and predictions are not so explicitly stated. In the confusion matrix, we can always see exactly what the true classes are and what the predicted classes are, but once we start summarizing it in various ways, this information is no longer explicitly available. The summaries still will often depend on which class we consider "positive" and which we consider "negative," though.

Since which class is which really is arbitrary, so it is always worth a thought deciding which you want to call which and definitely something you want to make explicit in any documentation of your analysis.

Accuracy

The simplest measure of how well a classification is doing is the accuracy. It measures how many classes it gets right out of the total, so it is the diagonal values of the confusion matrix divided by the total:

```
confusion_matrix <- table(formatted_data$Class, classified,
                          dnn=c("Data", "Predictions"))
accuracy <- sum(diag(confusion_matrix)) /
sum(confusion_matrix)
accuracy
## [1] 0.9399142
```

This measure of the classification accuracy is pretty simple to understand, but you have to be careful in what you consider a good accuracy. Of course, "good" is a subjective term, so let us get technical and think in terms of "better than chance." That means that your baseline for what you consider "good" is randomly guessing. This, at least, is not subjective.

It is still something you have to consider a bit carefully, though. Because what does randomly guessing mean? We naturally think of a random guess as one that chooses either class with the same 50/50 probability. If the data has the same number of observations for each of the two classes, then that would be a good strategy and would get the average accuracy of 0.5. So better than chance would, in that case, be better than 0.5. The data doesn't have to have the same number of instances for each class. The breast cancer data does not. The breast cancer data has more benign tumors than malignant tumors:

```
table(BreastCancer$Class)
##
##    benign malignant
##       458       241
```

Here, you would be better off guessing more benign than malignant. If you had to guess and already knew that you were more than twice as likely to have a benign than a malignant tumor, you would always guess benign:

```
tbl <- table(BreastCancer$Class)
tbl["benign"] / sum(tbl)
##    benign
## 0.6552217
```

Always guessing "benign" is a lot better than 50/50. Of course, it is arguable whether this is guessing, but it is a strategy for guessing, and you want your model to do better than this simple strategy.

Always guessing the most frequent class—assuming that the frequency of the classes in the data set is a representative for the frequency in new data as well (which is a strong assumption)—is the best strategy for guessing.

If you actually want to see "random" guessing, you can get an estimate of this by simply permuting the classes in the data. The function `sample()` can do this:

```
table(BreastCancer$Class, sample(BreastCancer$Class))
##
##           benign malignant
## benign       291       167
## malignant    167        74
```

This gives you an estimate for random guessing, but since it is random, you would want to get more than one to get a feeling for how much it varies with the guess:

```
accuracy <- function(confusion_matrix)
   sum(diag(confusion_matrix))/sum(confusion_matrix)
sample_table <- function()
   table(BreastCancer$Class, sample(BreastCancer$Class))
replicate(8, sample_table() |> accuracy())
## [1] 0.5450644 0.5336195 0.5879828 0.5565093
## [5] 0.5622318 0.5565093 0.5364807 0.5278970
```

As you can see, even random permutations do better than 50/50—but the better guess is still just the most frequent class, and at the very least, you would want to beat that.

Sensitivity and Specificity

We want a classifier to have a high accuracy, but accuracy isn't everything. The costs in real life of misclassifying often have different consequences when you classify something like a benign tumor as malignant from when you classify a malignant tumor as benign. In a clinical setting, you have to weight the false positives against the false negatives and the consequences they have. You are interested in more than pure accuracy.

We usually use two measures of the predictions of a classifier that takes that into account: the specificity and the sensitivity of the model. The first measure captures how often the model predicts a negative case correctly. In the breast cancer data, this is how often, when the model predicts a tumor as benign, it actually is:

```
(specificity <- confusion_matrix[1,1] /
   (confusion_matrix[1,1] + confusion_matrix[1,2]))
## [1] 0.9759825
```

The sensitivity does the same thing but for the positives. It captures how well, when the data has the positive class, your model predicts this correctly:

```
(sensitivity <- confusion_matrix[2,2]/
   (confusion_matrix[2,1] + confusion_matrix[2,2]))
## [1] 0.8713693
```

If your accuracy is 100%, then both of these will also be 100%. But there is usually a trade-off between the two. Using the "best guessing" strategy of always picking the most frequent class will set one of the two to 100% but at the cost of the other. In the breast cancer data, the best guess is always benign, the negative case, and always guessing benign will give us a specificity of 100%.

This strategy can always achieve 100% for one of the two measures but at the cost of setting the other to 0%. If you only ever guess at one class, you are perfect when the data is actually from that class, but you are always wrong when the data is from the other class.

Because of this, we are never interested in optimizing either measure alone. That is trivial. We want to optimize both. We might consider specificity more important than sensitivity or vice versa, but even if we want one to be 100%, we also want the other to be as good as we can get it.

To evaluate how much better than chance we are doing, we can again compare to random permutations. This tells us how well we are doing compared to random guesses for both:

```
specificity <- function(confusion_matrix)
   confusion_matrix[1,1] /
   (confusion_matrix[1,1]+confusion_matrix[1,2])
sensitivity <- function(confusion_matrix)
   confusion_matrix[2,2] /
   (confusion_matrix[2,1]+confusion_matrix[2,2])
prediction_summary <- function(confusion_matrix)
   c("accuracy" = accuracy(confusion_matrix),
     "specificity" = specificity(confusion_matrix),
     "sensitivity" = sensitivity(confusion_matrix))
random_prediction_summary <- function()
   prediction_summary(
     table(BreastCancer$Class, sample(BreastCancer$Class))
)
replicate(3, random_prediction_summary())
##                  [,1]      [,2]      [,3]
## accuracy  0.5536481 0.5536481 0.5278970
## specificity 0.6593886 0.6593886 0.6397380
## sensitivity 0.3526971 0.3526971 0.3153527
```

Other Measures

The specificity is also known as the true negative rate since it measures how many of the negative classifications are true. Similarly, the sensitivity is known as the true positive rate. There are analogue measures for getting things wrong. The false negative rate is the analogue of the true negative rate, but instead of dividing the true negatives by all the negatives, it divides the false negatives by all the negatives. The false positive rate similarly divides the false positives by all the positives. Having these two measures together with sensitivity and specificity is not really adding much. The true negative rate is just one minus the false negative rate and similar for the true positive rate and false positive rate. They just focus on when the model gets things wrong instead of when it gets things right.

All four measures split the confusing matrix into the two rows. They look at when the data says the class is true and when the data says the class is false. We can also look at the columns instead and consider when the predictions are true and when the predictions are false.

When we look at the column where the predictions are false—for the breast cancer when the tumors are predicted as benign—we have the false omission rate, which is the false negatives divided by all the predicted negatives:

```
confusion_matrix[2,1] / sum(confusion_matrix[,1])
## [1] 0.06485356
```

The negative predictive value is instead the true negatives divided by the predicted negatives:

```
confusion_matrix[1,1] / sum(confusion_matrix[,1])
## [1] 0.9351464
```

These two will always sum to one, so we are really only interested in one of them, but which we choose is determined by which we find more important.

For the predicted positives, we have the positive predictive values and false discovery rate:

```
confusion_matrix[2,2] / sum(confusion_matrix[,2])
## [1] 0.9502262
confusion_matrix[1,2] / sum(confusion_matrix[,2])
## [1] 0.04977376
```

The false discovery rate, usually abbreviated FDR, is the one most frequently used. It is closely related to the threshold used on p-values (the significance thresholds) in classical hypothesis testing. Remember that if you have a 5% significance threshold in classical hypothesis testing, it means that when the null hypothesis is true, you will predict it is false 5% of the time. This means that your false discovery rate is 5%.

The classical approach is to pick an acceptable false discovery rate; by convention, this is 5%, but there is nothing magical about that number—it is simply convention—and then that threshold determines how extreme a test statistic has to be before we switch from predicting a negative to predicting a positive. This approach entirely ignores the cases where the data is from the positive class. It has its uses, but not for classification where you have data from both the positive class and the negative class, so we will not consider it more here. You will have seen it in statistics classes, and you can learn more about it in any statistics textbook.

More Than Two Classes

All of the above considers a situation where we have two classes, one we call positive and one we call negative. This is a common case, which is the reason we have so many measures for dealing with it, but it is not the only case. Quite often, we need to classify data into more than two classes.

The only measure you can reuse there is the accuracy. The accuracy is always the sum along the diagonal divided by the total number of observations. Accuracy still isn't everything in those cases. Some classes are perhaps more important to get right than others—or just harder to get right than others—so you have to use a lot of sound judgment when evaluating a classification. There are just fewer rules of thumb to use here, so you are more left to your own judgment.

Sampling Approaches

To validate classifiers, I suggested splitting the data into a training data set and a test data set. I also mentioned that there might be hidden structures in your data set, so you always want to make this split a random split of the data.

Generally, there are a lot of benefits you can get out of randomly splitting your data or randomly subsampling from your data. We have mostly considered prediction in this chapter, where splitting the data into training and a test data lets us evaluate how well a model does at predicting on unseen data. But randomly splitting or subsampling from data is also very useful for inference. When we do inference, we can typically get confidence intervals for model parameters, but these are based on theoretical results that assume that the data is from some (usually) simple distribution. Data is generally not. If you want to know how a parameter is distributed from the empirical distribution of the data, you will want to subsample and see what distribution you get.

Random Permutations of Your Data

With the `cars` data, we split the observations into two equally sized data sets. Since this data is ordered by the stopping distance, splitting it into the first half and the second half makes the data sets different in distributions.

The simplest approach to avoiding this problem is to reorder your data randomly before you split it. Using the `sample()` function, we can get a random permutation of any input vector—we saw that earlier—and we can exploit this to get a random order of your data set.

Using `sample(1:n)`, we get a random permutation of the numbers from 1 to *n*. We can select rows in a data frame by giving it a vector of indices for the rows. Combining these two observations, we can get a random order of `cars` observations this way:

```
permuted_cars <- cars[sample(1:nrow(cars)),]
permuted_cars |> head(3)
##    speed dist
## 9     10   34
## 1      4    2
## 48    24   93
```

The numbers to the left of the data frame are the original row numbers (it really is the row names, but it is the same in this case).

We can write a simple function for doing this for general data frames:

```
permute_rows <- function(df) df[sample(1:nrow(df)),]
```

Using this, we can add it to a data analysis pipeline where we would write

```
permuted_cars <- cars |> permute_rows()
```

Splitting the data into two sets, training and testing, is one approach to subsampling, but a general version of this is used in something called cross-validation. Here, the idea is to get more than one result out of the random permutation we use. If we use a single training/test split, we only get one estimate of how a model performs on a data set. Using more gives us an idea about the variance of this.

We can split a data set into *n* groups like this:

```
group_data <- function(df, n) {
   groups <- rep(1:n, each = nrow(df)/n)
   split(df, groups)
}
```

You don't need to understand the details of this function for now, but it is a good exercise to try to figure it out, so you are welcome to hit the documentation and see if you can work it out.

The result is a `list`, a data structure we haven't explored yet (but we will later in the book, when we do some more serious programming). It is necessary to use a list here since vectors or data frames cannot hold complex data, so if we combined the result in one of those data structures, they would just be merged back into a single data frame here.

As it is, we get something that contains *n* data structures that each have a data frame of the same form as the `cars` data:

```
grouped_cars <- cars |> permute_rows() |> group_data(5)
grouped_cars |> str()
## List of 5
##  $ 1:'data.frame':   10 obs. of  2 variables:
##   ..$ speed: num [1:10] 12 4 14 13 18 19 14 4 ...
##   ..$ dist : num [1:10] 28 2 60 26 76 46 26 10 ...
##  $ 2:'data.frame':   10 obs. of  2 variables:
##   ..$ speed: num [1:10] 15 13 12 18 20 18 12 15 ..
##   ..$ dist : num [1:10] 54 34 20 84 32 42 24 26 ..
##  $ 3:'data.frame':   10 obs. of  2 variables:
##   ..$ speed: num [1:10] 11 24 22 9 14 13 24 20 ...
##   ..$ dist : num [1:10] 28 70 66 10 36 46 120 48..
##  $ 4:'data.frame':   10 obs. of  2 variables:
##   ..$ speed: num [1:10] 12 10 25 17 17 23 19 11 ..
##   ..$ dist : num [1:10] 14 26 85 40 32 54 68 17 ..
##  $ 5:'data.frame':   10 obs. of  2 variables:
##   ..$ speed: num [1:10] 10 20 15 16 13 7 19 24 ...
##   ..$ dist : num [1:10] 18 52 20 32 34 22 36 92 ..
grouped_cars[[1]] # First sample
##    speed dist
## 15    12   28
## 1      4    2
## 22    14   60
## 16    13   26
## 34    18   76
## 37    19   46
## 20    14   26
## 2      4   10
## 5      8   16
## 42    20   56
```

All you really need to know for now is that to get an entry in a list, you need to use `[[]]` indexing instead of `[]` indexing.

If you use `[]`, you will also get the data, but the result will be a list with one element, which is not what you want:

```
grouped_cars[1]
## $`1`
##    speed dist
## 15    12   28
## 1      4    2
## 22    14   60
## 16    13   26
## 34    18   76
## 37    19   46
## 20    14   26
## 2      4   10
## 5      8   16
## 42    20   56
```

We can use the different groups to get estimates of the model parameters in the linear model for cars:

```
lm(dist ~ speed, data = grouped_cars[[1]])$coefficients
##  (Intercept)     speed
##   -10.006702  3.540214
```

With a bit of programming, we can get the estimates for each group:

```
get_coef <- function(df)
   lm(dist ~ speed, data = df)$coefficients
# Get estimates from first group
estimates <- get_coef(grouped_cars[[1]])
for (i in 2:length(grouped_cars)) {
   # Append the next group
   estimates <- rbind(estimates, get_coef(grouped_cars[[i]]))
}
Estimates
##           (Intercept)     speed
## estimates   -10.00670  3.540214
##             -33.89655  4.862069
##             -29.25116  4.744186
##             -11.82554  3.555755
##             -14.87639  3.494779
```

Right away, I will stress that this is not the best way to do this, but it shows you how it could be done. We will get to better approaches shortly. Still, you can see how splitting the data this way lets us get distributions for model parameters.

There are several reasons why this isn't the optimal way of coding this. The row names are ugly, but that is easy to fix. The way we combine the estimates in the data frame is inefficient—although it doesn't matter much with such a small data set—and later in the book, we will see why. The main reason, though, is that explicit loops like this make it hard to follow the data transformations since it isn't a pipeline of processing.

The package `purrr` lets us work on lists using pipelines. You import the package:

```
library(purrr)
```

and then you have access to the function `map_df()` that lets you apply a function to each element of the list:

```
estimates <- grouped_cars |> map_df(get_coef)
```

The `map_df` function maps (as the name suggests) across its input, applying the function to each element in the input list. The results of each function call are turned into rows in a data frame (where the `_df` part of the name comes from). This pipeline is essentially doing the same as the more explicit loop we wrote before; there is just less code to write. If you are used to imperative programming languages, this will look very succinct, but if you have experience in functional programming languages, it should look familiar.

Cross-Validation

A problem with splitting the data into many small groups is that we get a large variance in estimates. Instead of working with each little data set independently, we can remove one of the data sets and work on all the others. This will mean that our estimates are no longer independent, but the variance goes down. The idea of removing a subset of the data and then cycling through the groups evaluating a function for each group that is left out is called cross-validation. Well, it is called cross-validation when we use it to validate prediction, but it works equally well for inferring parameters.

If we already have the grouped data frames in a list, we can remove one element from the list using `[-i]` indexing—just as we can for vectors—and the result is a list containing all the other elements. We can then combine the elements in the list into a single data frame using the `do.call("rbind",.)` magical invocation.

So we can write a function that takes the grouped data frames and gives us another list of data frames that contains data where a single group is left out. One way to do this is listed as follows; that implementation uses the `bind_rows` function from the `dplyr` package (get it using `library(dplyr)`):

```
cross_validation_groups <- function(grouped_df) {
   remove_group <- function(group)
      # remove group "group" from the list
      grouped_df[-group] |>
      # merge the remaining groups into one data frame
      bind_rows()
   # Iterate over indices from 1 to number of groups
   seq_along(grouped_df) |>
      # get the data frame with this group removed
      map(remove_group)
}
```

This function is a little more spicy than those we have written before, but it doesn't use anything we haven't seen already. In the function, we write another helper function, `remove_group`. You can write functions inside other functions, and if you do, then the inner function can see the variables in the outer function. Our `remove_group` function can see the `grouped_df` data frame. We give it an argument, `group`, and it removes the group with that index using `-group` in the subscript `grouped_df[-group]`. Since `grouped_df` is a `list` of data frames, removing one of them still leaves us with a list of data frames, but we would rather have a single data frame. The `bind_rows` function merges the list into a single data frame containing all the data points we didn't remove.

With the function written, we can create the cross-validation groups. We use `seq_along(grouped_df)` to create all the numbers from one to the length of `grouped_df`—using this function is slightly safer than writing `1:len(grouped_df)` because it correctly handles empty lists, so you should get used to using it. We loop over all these numbers with a `map` function—this function is also from the `purrr` package and behaves like `map_df` except that it returns a list and not a data frame—and apply `remove_group` to each index. This results in a list of data frames, which is exactly what we want.

We could have combined this with the `group_data()` function, but I prefer to write functions that do one simple thing and combine them instead using pipelines. We can use this function and all the stuff we did earlier to get estimates using cross-validation:

```
cars |>
   permute_rows() |> # randomize for safety...
   group_data(5) |> # get us five groups
   cross_validation_groups() |> # then make five cross-
validation groups
   # For each cross-validation group, estimate the
cofficients and put
   # the results in a data frame
   map_df(
      # We need a lambda expression here because lm doesn't
take
      # the data frame as its first argument
      \(df) lm(dist ~ speed, data = df)$coefficients
   )
## # A tibble: 5 × 2
##   `(Intercept)` speed
##           <dbl> <dbl>
## 1         -18.0  3.92
## 2         -14.4  3.86
## 3         -20.1  4.15
## 4         -16.2  3.79
## 5         -19.5  3.96
```

Where cross-validation is typically used is when leaving out a subset of the data for testing and using the rest for training.

We can write a simple function for splitting the data this way, similar to the `cross_validation_groups()` function. It cannot return a list of data frames but needs to return a list of lists, each list containing a training data frame and a test data frame. It looks like this:

```
cross_validation_split <- function(grouped_df) {
   seq_along(grouped_df) |> map(
      \(group) list(
         # Test is the current group
         test = grouped_df[[group]],
         # Training is all the others
         training = grouped_df[-group] |> bind_rows()
      ))
}
```

The function follows the same pattern as the previous, I just haven't bothered with writing an inner function; instead I use a lambda expression (`\(group) ...`). It creates a list with two elements, `test` and `training`. In `test`, we put the current group—we subscript with `[[group]]` to get the actual data frame instead of a list that holds it—and in `training`, we put all the other groups. Here, we use `[-group]` to get a list of all the other elements—`[[-group]]` would not work for us—and then we use the `bind_rows()` function we saw earlier to merge the list into a single data frame.

Don't worry if you don't understand all the details of it. After reading later programming chapters, you will. Right now, I hope you just get the gist of it.

I will not show you the result. It is just long and not that pretty, but if you want to see it, you can type in

```
cars |>
   permute_rows() |>
   group_data(5)  |>
   cross_validation_split()
```

As we have seen, we can index into a list using `[[]]`. We can also use the `$name` indexing like we can for data frames, so if we have a list `lst` with a `training` data set and a `test` data set, we can get them as `lst$training` and `lst$test`.

```
prediction_accuracy <- function(test_and_training) {
   test_and_training |>
      map_dbl(
         \(tt) {
          # Fit the model using training data
          fit <- lm(dist ~ speed, data = tt$training)
          # Then make predictions on the test data
          predictions <- predict(fit, newdata = tt$test)
          # Get root mean square error of result
          rmse(predictions, tt$test$dist)
         }
      )
}
```

You should be able to understand most of this function even though we haven't covered much R programming yet, but if you do not, then don't worry.

You can then add this function to your data analysis pipeline to get the cross-validation accuracy for your different groups:

```
cars |>
   permute_rows() |>
   group_data(5) |>
   cross_validation_split() |>
   prediction_accuracy()
## [1] 56.62113 38.55348 33.52728 59.27442 48.77524
```

The prediction accuracy function isn't general. It is hardwired to use a linear model and the model `dist ~ speed`. It is possible to make a more general function, but that requires a lot more R programming skills, so we will leave the example here.

Selecting Random Training and Testing Data

In the example earlier where I split the data `cars` into training and test data using `sample(0:1, n, replacement = TRUE)`, I didn't permute the data and then deterministically split it afterward. Instead, I sampled training and test based on probabilities of picking any given row as training and test.

What I did was adding a column to the data frame where I randomly picked whether an observation should be used for the training or for the test data. Since it required first adding a new column and then selecting rows based on it, it doesn't work well as part of a data analysis pipeline. We can do better and slightly generalize the approach at the same time.

To do this, I shamelessly steal two functions from the documentation of the `purrr` package. They do the same thing as the grouping function I wrote earlier. If you do not quite follow the example, do not worry. But I suggest you try to read the documentation for any function you do not understand and at least try to work out what is going on. Follow it as far as you can, but don't sweat it if there are things you do not fully understand. After finishing the entire book, you can always return to the example.

The grouping function earlier defined groups by splitting the data into *n* equally sized groups. The first function here instead samples from groups specified by probabilities. It creates a vector naming the groups, just as I did before. It just names the groups based on named values in a probability vector and creates a group vector based on probabilities given by this vector:

```
random_group <- function(n, probs) {
   probs <- probs / sum(probs)
```

```
  g <- findInterval(seq(0, 1, length = n), c(0,
cumsum(probs)),
                     rightmost.closed = TRUE)
  names(probs)[sample(g)]
}
```

If we pull the function apart, we see that it first normalizes a probability vector. This just means that if we give it a vector that doesn't sum to one, it will still work. To use it, it makes the code easier to read if it already sums to one, but the function can deal with it, even if it doesn't.

The second line, which is where it is hardest to read, just splits the unit interval into *n* subintervals and assigns a group to each subinterval based on the probability vector. This means that the first chunk of the *n* intervals is assigned to the first group, the second chunk to the second group, and so on. It is not doing any sampling yet, it just partitions the unit interval into *n* subintervals and assigns each subinterval to a group.

The third line is where it is sampling. It now takes the *n* subintervals, permutes them, and returns the names of the probability vector each one falls into.

We can see it in action by calling it a few times. We give it a probability vector where we call the first probability "`training`" and the second "`test`":

```
random_group(8, c(training = 0.5, test = 0.5))
## [1] "training" "training" "training" "test"
## [5] "test"     "training" "test"     "test"
random_group(8, c(training = 0.5, test = 0.5))
## [1] "training" "test"    "training" "test"
## [5] "training" "test"    "training" "test"
```

We get different classes out when we sample, but each class is picked with 0.5 probability. We don't have to pick them 50/50, though; we can choose more training than test data, for example:

```
random_group(8, c(training = 0.8, test = 0.2))
## [1] "training" "training" "training" "training"
## [5] "test"     "training" "test"     "training"
```

The second function just uses this random grouping to split the data set. It works exactly like the cross-validation splitting we saw earlier:

```
partition <- function(df, n, probs) {
  replicate(n, split(df, random_group(nrow(df), probs)),
FALSE)
}
```

The function replicates the subsampling *n* times. Here, *n* is not the number of observations you have in the data frame, but a parameter to the function. It lets you pick how many subsamples of the data you want.

We can use it to pick four random partitions. Here, with training and test, select with 50/50 probability:

```
random_cars <- cars |> partition(4, c(training = 0.5, test =
0.5))
```

If you evaluate it on your computer and look at `random_cars`, you will see that resulting values are a lot longer now. This is because we are not looking at smaller data sets this time; we have as many observations as we did before (which is 50), but we have randomly partitioned them.

We can combine this `partition()` function with the accuracy prediction from before:

```
random_cars |> prediction_accuracy()
## [1] 62.76781 87.52504 92.00689 84.99749
```

Examples of Supervised Learning Packages

So far in this chapter, we have looked at classical statistical methods for regression (linear models) and classification (logistic regression), but there are many machine learning algorithms for both, and many are available as R packages.

They all work similarly to the classical algorithms. You give the algorithms a data set and a formula specifying the model matrix. From this, they do their magic. All the ideas presented in this chapter can be used together with them.

In the following, I go through a few packages, but there are many more. A Google search should help you find a package if there is a particular algorithm you are interested in applying.

I present their use with the same two data sets we have used earlier, the `cars` data where we aim at predicting the stopping distance from the speed and the `BreastCancer` where we try to predict the class from the cell thickness. For both these cases, the classical models—a linear model and a logistic regression—are more ideal solutions, and these models will not outcompete them, but for more complex data sets, they can usually be quite powerful.

Decision Trees

Decision trees work by building a tree from the input data, splitting on a parameter in each inner node according to a variable value. This can be splitting on whether a numerical value is above or below a certain threshold or which level a factor has.

Decision trees are implemented in the `rpart` package, and models are fitted just as linear models are:

```
library(rpart)
## Warning: package 'rpart' was built under R version
## 4.1.2
model <- cars %>% rpart(dist ~ speed, data = .)
rmse(predict(model, cars), cars$dist)
## [1] 117.1626
```

Building a classifying model works very similar. We do not need to translate the cell thickness into a numerical value, though; we can use the data frame as it is (but you can experiment with translating factors into numbers if you are interested in exploring this):

```
model <- BreastCancer %>%
   rpart(Class ~ Cl.thickness, data = .)
```

The predictions when we used the `glm()` function were probabilities for the tumor being malignant. The predictions made using the decision tree give you the probabilities both for being benign and malignant:

```
predict(model, BreastCancer) |> head()
##       benign malignant
## 1 0.82815356 0.1718464
## 2 0.82815356 0.1718464
## 3 0.82815356 0.1718464
## 4 0.82815356 0.1718464
## 5 0.82815356 0.1718464
## 6 0.03289474 0.9671053
```

To get a confusion matrix, we need to translate these probabilities into the corresponding classes. The output of `predict()` is not a data frame but a matrix, so we first convert it into a data frame using the function `as.data.frame()`, and then we use the `%$%` operator in the pipeline to get access to the columns by name in the next step:

```
predicted_class <-
   predict(model, BreastCancer) %>%
   as.data.frame() %$%
   ifelse(benign > 0.5, "benign", "malignant")
table(BreastCancer$Class, predicted_class)
##        predicted_class
##          benign malignant
## benign      453         5
## malignant    94       147
```

Another implementation of decision trees is the `ctree()` function from the `party` package:

```
library(party)
model <- cars %>% ctree(dist ~ speed, data = .)
rmse(predict(model, cars), cars$dist)
## [1] 117.1626
model <- BreastCancer %>%
   ctree(Class ~ Cl.thickness, data = .)
predict(model, BreastCancer) %>% head()
## [1] benign    benign    benign    benign
## [5] benign    malignant
## Levels: benign malignant
table(BreastCancer$Class, predict(model, BreastCancer))
##
##          benign malignant
##  benign    453         5
##  malignant  94         147
```

I like this package slightly more since it can make plots of the fitted models (see Figure 6-8):

```
cars %>% ctree(dist ~ speed, data = .) %>% plot()
```

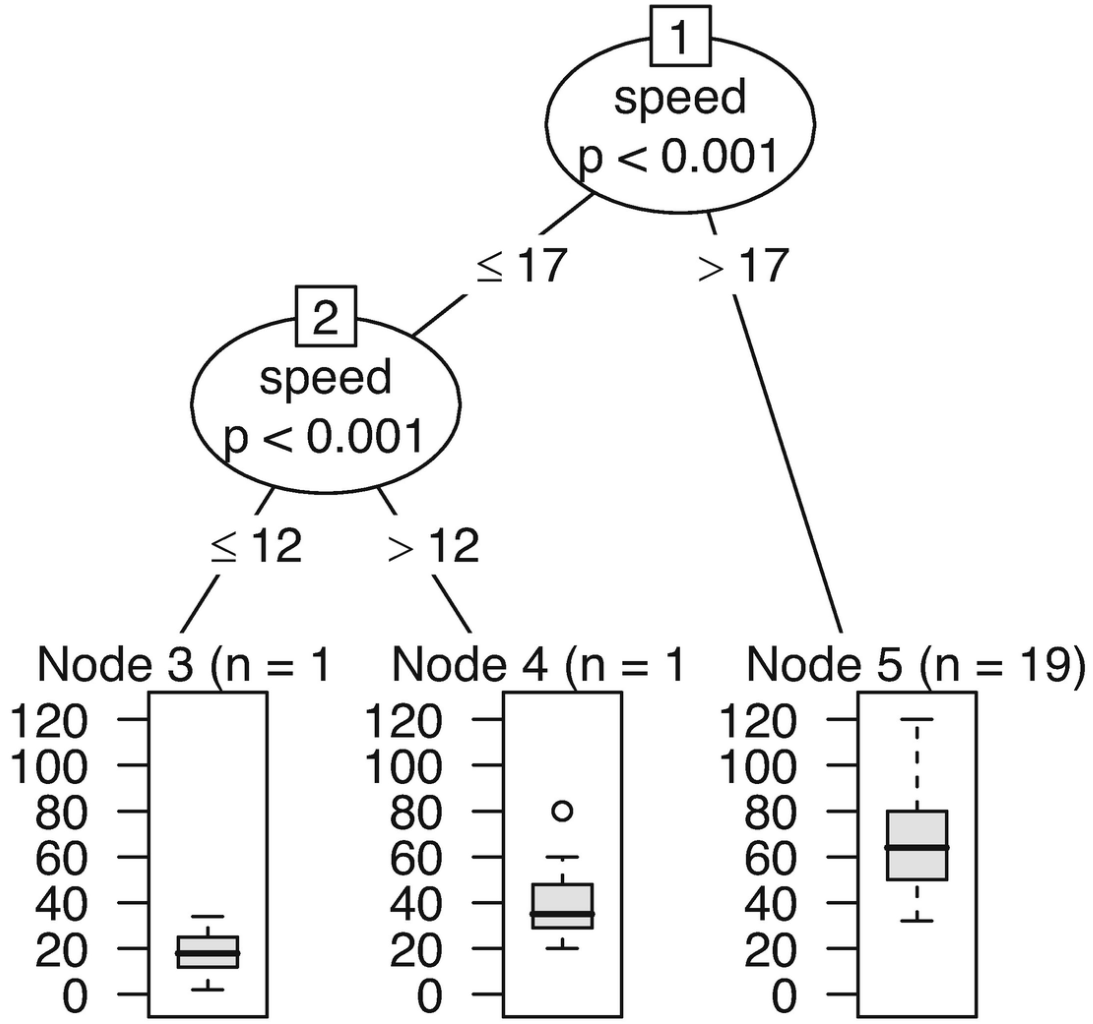

Figure 6-8 Plot of the cars decision tree

Random Forests

Random forests generalize decision trees by building several of them and combining them. They are implemented in the `randomForest` package:

```
library(randomForest)
model <- cars %>% randomForest(dist ~ speed, data = .)
rmse(predict(model, cars), cars$dist)
## [1] 83.7541
```

For classification, the predictions are the actual classes as a factor, so no translation is needed to get a confusion matrix:

```
model <- BreastCancer %>%
   randomForest(Class ~ Cl.thickness, data = .)
predict(model, BreastCancer) %>% head()
##          1         2         3         4         5
##     benign    benign    benign malignant    benign
##          6
##  malignant
##  Levels: benign malignant
table(BreastCancer$Class, predict(model, BreastCancer))
##
##           benign malignant
##  benign      437        21
##  malignant    76       165
```

Neural Networks

You can use a package called `nnet` to construct neural networks:

```
library(nnet)
```

You can use it for both classification and regression. We can see it in action on the `cars` data set:

```
model <- cars %>% nnet(dist ~ speed, data = ., size = 5)
## # weights:  16
## initial  value 123632.602158
## final  value 120655.000000
## converged
rmse(predict(model, cars), cars$dist)
## [1] 347.3543
```

The neural networks require a `size` parameter specifying how many nodes you want in the inner layer of the network. Here, I have just used five.

For classification, you use a similar call:

```
model <- BreastCancer %>%
   nnet(Class ~ Cl.thickness, data = ., size = 5)
## # weights: 56
## initial value 453.502123
## iter 10 value 226.317196
## iter 20 value 225.125028
## iter 30 value 225.099296
## iter 40 value 225.098355
## final value 225.098268
## converged
```

The output of the `predict()` function is probabilities for the tumor being malignant:

```
predict(model, BreastCancer) %>% head()
```

```
##        [,1]
## 1 0.3461460
## 2 0.3461460
## 3 0.1111139
## 4 0.5294021
## 5 0.1499858
## 6 0.9130386
```

We need to translate it into classes, and for this, we can use a lambda expression:

```
predicted_class <- predict(model, BreastCancer) %>%
   { ifelse(. < 0.5, "benign", "malignant") }
table(BreastCancer$Class, predicted_class)
##         predicted_class
##           benign malignant
##  benign      437        21
##  malignant    76       165
```

Support Vector Machines

Another popular method is support vector machines. These are implemented in the `ksvm()` function in the `kernlab` package:

```
library(kernlab)
model <- cars %>% ksvm(dist ~ speed, data = .)
rmse(predict(model, cars), cars$dist)
## [1] 92.41686
```

For classification, the output is again a factor we can use directly to get a confusion matrix:

```
model <- BreastCancer %>%
   ksvm(Class ~ Cl.thickness, data = .)
predict(model, BreastCancer) %>% head()
## [1] benign    benign    benign    malignant
## [5] benign    malignant
## Levels: benign malignant
table(BreastCancer$Class, predict(model, BreastCancer))
##
##           benign malignant
## benign       437        21
## malignant     76       165
```

Naive Bayes

Naive Bayes essentially assumes that each explanatory variable is independent of the others and uses the distribution of these for each category of data to construct the distribution of the response variable given the explanatory variables.

Naive Bayes is implemented in the `e1071` package:

```
library(e1071)
```

The package doesn't support regression analysis—after all, it needs to look at conditional distributions for each output variable value—but we can use it for classification. The function we need is `naiveBayes()`, and we can use the `predict()` output directly to get a confusion matrix:

```
model <- BreastCancer %>%
   naiveBayes(Class ~ Cl.thickness, data = .)
predict(model, BreastCancer) %>% head
## [1] benign    benign    benign    malignant
## [5] benign    malignant
## Levels: benign malignant
table(BreastCancer$Class, predict(model, BreastCancer))
##
##           benign malignant
## benign       437        21
## malignant     76       165
```

Exercises

Fitting Polynomials

Use the `cars` data to fit higher degree polynomials and use training and test data to explore how they generalize. At which degree do you get the better generalization?

Evaluating Different Classification Measures

Earlier, I wrote functions for computing the accuracy, specificity (true negative rate), and sensitivity (true positive rate) of a classification. Write similar functions for the other measures described before. Combine them in a `prediction_summary()` function like I did earlier.

Breast Cancer Classification

You have seen how to use the `glm()` function to predict the classes for the breast cancer data. Use it to make predictions for training and test data, randomly splitting the data in these two classes, and evaluate all the measures with your `prediction_summary()` function.

If you can, then try to make functions similar to the ones I used to split data and evaluate models for the `cars` data.

Leave-One-Out Cross-Validation (Slightly More Difficult)

The code I wrote earlier splits the data into *n* groups and constructs training and test data based on that. This is called *n*-fold cross-validation. There is another common approach to cross-validation called leave-one-out cross-validation. The idea here is to remove a single data observation and use that for testing and all the rest of the data for training.

This isn't used that much if you have a lot of data—leaving out a single data point will not change the trained model much if you have lots of data points anyway—but for smaller data sets, it can be useful.

Try to program a function for constructing subsampled training and test data for this strategy.

Decision Trees

Use the `BreastCancer` data to predict the tumor class, but try including more of the explanatory variables. Use cross-validation or sampling of training/test data to explore how it affects the prediction accuracy.

Random Forests

Use the `BreastCancer` data to predict the tumor class, but try including more of the explanatory variables. Use cross-validation or sampling of training/test data to explore how it affects the prediction accuracy.

Neural Networks

The `size` parameter for the `nnet` function specifies the complexity of the model. Test how the accuracy depends on this variable for classification on the `BreastCancer` data.

Earlier, we only used the cell thickness variable to predict the tumor class. Include the other explanatory variables and explore if having more information improves the prediction power.

Support Vector Machines

Use the `BreastCancer` data to predict the tumor class, but try including more of the explanatory variables. Use cross-validation or sampling of training/test data to explore how it affects the prediction accuracy.

Compare Classification Algorithms

Compare the logistic regression, the neural networks, the decision trees, the random forests, and the support vector machines in how well they classify tumors in the `BreastCancer` data. For each, take the best model you obtained in your experiments.

Footnotes

1 We need the `%>%` operator here because of where we want the `cars` data to go in the call to `lm`; we can't do that this easily with `|>`.

2 In this pipeline, we have switched to `%>%` again, because we need the left-hand side to go into the `data` argument in `glm`.