4 Exploring data with graphs

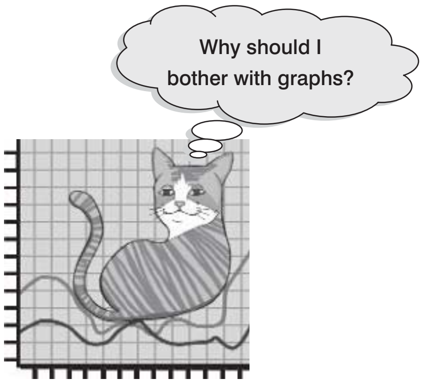
FIGURE 4.1
Explorer Field borrows a bike and gets ready to ride it recklessly around a caravan site

4.1. What will this chapter tell me? ①

As I got a bit older I used to love exploring. At school they would teach you about maps and how important it was to know where you were going and what you were doing. I used to have a more relaxed view of exploration and there is a little bit of a theme of me wandering off to whatever looked most exciting at the time. I got lost at a holiday camp once when I was about 3 or 4. I remember nothing about this but apparently my parents were frantically running around trying to find me while I was happily entertaining myself (probably by throwing myself head first out of a tree or something). My older brother, who was supposed to be watching me, got a bit of flak for that but he was probably working out equations to bend time and space at the time. He did that a lot when he was 7. The careless explorer in me hasn't really gone away: in new cities I tend to just wander off and hope for the best, and usually get lost and fortunately usually don't die (although I tested my luck once by wandering through part of New Orleans where apparently tourists get mugged a lot – it seemed fine to me). When exploring data you can't afford not to have a map; to explore data in the way that the 6-year-old me used to explore the world is to spin around 8000 times while drunk and then run along the edge of a cliff. Wright (2003) quotes Rosenthal who said that researchers should 'make friends with their data'. This wasn't meant to imply that people who use statistics may as well befriend their data because the data are the only friend they'll have; instead Rosenthal meant that researchers often rush their analysis. Wright makes the analogy of a fine wine: you should savour the bouquet and delicate flavours to truly enjoy the experience. That's perhaps overstating the joys of data analysis, but rushing your analysis is, I suppose, a bit like gulping down a bottle of wine: the outcome is messy and incoherent! To negotiate your way around your data you need a map. Maps of data are called graphs, and it is into this tranquil and tropical ocean that we now dive (with a compass and ample supply of oxygen, obviously).

4.2. The art of presenting data ①

4.2.1. Why do we need graphs ①



Graphs are a really useful way to look at your data before you get to the nitty-gritty of actually analysing them. You might wonder why you should bother drawing graphs – after all, you are probably drooling like a rabid dog to get into the statistics and to discover the answer to your really interesting research question. Graphs are just a waste of your precious time, right? Data analysis is a bit like Internet dating (actually it's not, but bear with me): you can scan through the vital statistics and find a perfect match (good IQ, tall, physically fit, likes arty French films, etc.) and you'll think you have found the perfect answer to your question. However, if you haven't looked at a picture, then you don't really know how to interpret this information – your perfect match might turn out to be Rimibald the Poisonous, King of the Colorado River Toads, who has genetically combined himself with a human to further his plan to start up a lucrative rodent farm (they like to eat small rodents).[1] Data analysis is much the same: inspect your data with a picture, see how it looks and only then think about interpreting the more vital statistics.

4.2.2. What makes a good graph? ①

Before we get down to the nitty-gritty of how to draw graphs in **R**, I want to begin by talking about some general issues when presenting data. **R** (and other packages) make it very easy to produce very snazzy-looking graphs, and you may find yourself losing

[1] On the plus side, he would have a long sticky tongue and if you smoke his venom (which, incidentally, can kill a dog) you'll hallucinate (if you're lucky, you'd hallucinate that he wasn't a Colorado river toad–human hybrid).

consciousness at the excitement of colouring your graph bright pink (really, it's amazing how excited my undergraduate psychology students get at the prospect of bright pink graphs – personally I'm not a fan of pink). Much as pink graphs might send a twinge of delight down your spine, I want to urge you to remember why you're doing the graph – it's not to make yourself (or others) purr with delight at the pinkness of your graph, it's to present information (dull, perhaps, but true).

Tufte (2001) wrote an excellent book about how data should be presented. He points out that graphs should, among other things:

- Show the data.
- Induce the reader to think about the data being presented (rather than some other aspect of the graph, like how pink it is).
- Avoid distorting the data.
- Present many numbers with minimum ink.
- Make large data sets (assuming you have one) coherent.
- Encourage the reader to compare different pieces of data.
- Reveal data.

However, graphs often don't do these things (see Wainer, 1984, for some examples).

Let's look at an example of a bad graph. When searching around for the worst example of a graph that I have ever seen, it turned out that I didn't need to look any further than myself – it's in the first edition of the SPSS version of this book (Field, 2000). Overexcited by SPSS's ability to put all sorts of useless crap on graphs (like 3-D effects, fill effects and so on – Tufte calls these **chartjunk**), I literally went into some weird orgasmic state and produced an absolute abomination (I'm surprised Tufte didn't kill himself just so he could turn in his grave at the sight of it). The only consolation was that because the book was published in black and white, it's not pink! The graph is reproduced in Figure 4.2 (you

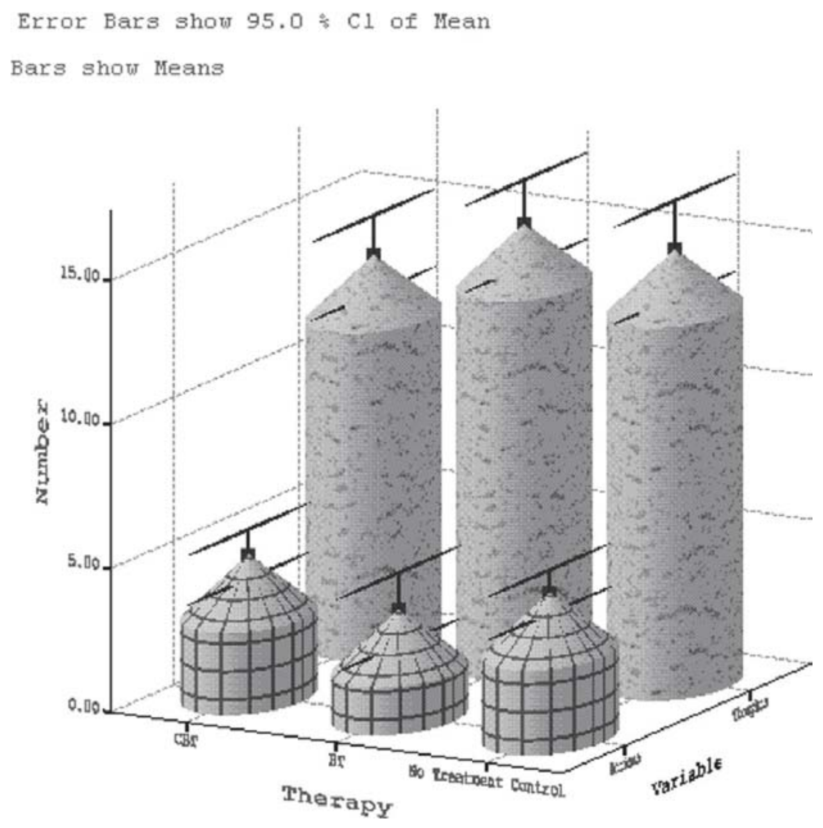


FIGURE 4.2
A cringingly bad example of a graph from the first edition of the SPSS version of this book

should compare this to the more sober version in this edition, Figure 16.4). What's wrong with this graph?

- × The bars have a 3-D effect: Never use 3-D plots for a graph plotting two variables: it obscures the data.[2] In particular it makes it hard to see the values of the bars because of the 3-D effect. This graph is a great example because the 3-D effect makes the error bars almost impossible to read.
- × Patterns: The bars also have patterns, which, although very pretty, merely distract the eye from what matters (namely the data). These are completely unnecessary.
- × Cylindrical bars: What's that all about, eh? Again, they muddy the data and distract the eye from what is important.
- × Badly labelled *y*-axis: 'number' of what? Delusions? Fish? Cabbage-eating sea lizards from the eighth dimension? Idiots who don't know how to draw graphs?

Now take a look at the alternative version of this graph (Figure 4.3). Can you see what improvements have been made?

- ✓ A 2-D plot: The completely unnecessary third dimension is gone, making it much easier to compare the values across therapies and thoughts/behaviours.
- ✓ The *y*-axis has a more informative label: We now know that it was the number of obsessive thoughts or actions per day that was being measured.
- ✓ Distractions: There are fewer distractions like patterns, cylindrical bars and the like!
- ✓ Minimum ink: I've got rid of superfluous ink by getting rid of the axis lines and by using lines on the bars rather than grid lines to indicate values on the *y*-axis. Tufte would be pleased.[3]

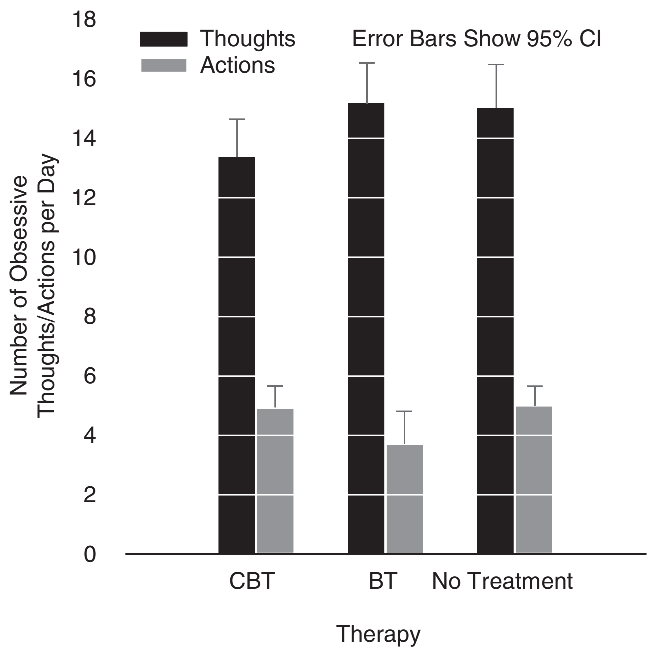


FIGURE 4.3 Figure 4.2 drawn properly

[2] If you do 3-D plots when you're plotting only two variables then a bearded statistician will come to your house, lock you in a room and make you write I μυστ νοτ δο 3–Δ γραπησ 75,172 times on the blackboard. Really, they will.

[3] Although he probably over-prescribes this advice: grid lines are more often than not very useful for interpreting the data.

4.2.3. Lies, damned lies, and … erm … graphs ①

Governments lie with statistics, but scientists shouldn't. How you present your data makes a huge difference to the message conveyed to the audience. As a big fan of cheese, I'm often curious about whether the urban myth that it gives you nightmares is true. Shee (1964) reported the case of a man who had nightmares about his workmates: 'He dreamt of one, terribly mutilated, hanging from a meat-hook.[4] Another he dreamt of falling into a bottomless abyss. When cheese was withdrawn from his diet the nightmares ceased.' This would not be good news if you were the minister for cheese in your country.

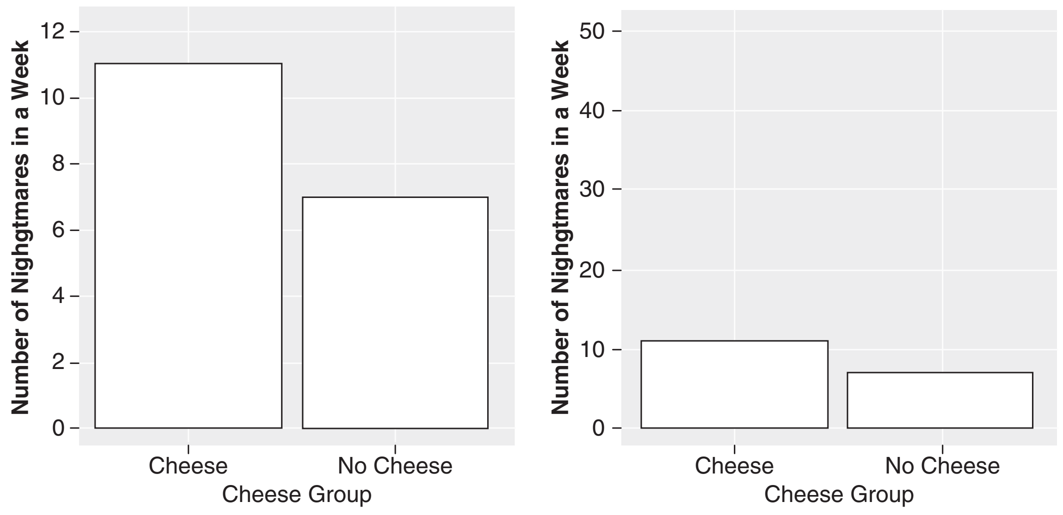


FIGURE 4.4 Two graphs about cheese

Figure 4.4 shows two graphs that, believe it or not, display exactly the same data: the number of nightmares had after eating cheese. The left-hand panel shows how the graph should probably be scaled. The *y*-axis reflects the maximum of the scale, and this creates the correct impression: that people have more nightmares about colleagues hanging from meat-hooks if they eat cheese before bed. However, as minister for cheese, you want people to think the opposite; all you have to do is rescale the graph (by extending the *y*-axis way beyond the average number of nightmares) and there suddenly seems to be little difference. Tempting as it is, don't do this (unless, of course, you plan to be a politician at some point in your life).

CRAMMING SAM'S TIPS Graphs ①

- ✓ The vertical axis of a graph is known as the *y*-axis of the graph.
- ✓ The horizontal axis of a graph is known as the *x*-axis of the graph.

If you want to draw a good graph follow the cult of Tufte:

- ✓ Don't create false impressions of what the data actually show (likewise, don't hide effects!) by scaling the *y*-axis in some weird way.
- ✓ Abolish chartjunk: Don't use patterns, 3-D effects, shadows, pictures of spleens, photos of your Uncle Fred or anything else.
- ✓ Avoid excess ink: don't include features unless they are necessary to interpret or understand the data.

[4] I have similar dreams, but that has more to do with some of my workmates than cheese!

4.3. Packages used in this chapter ①

The basic version of **R** comes with a *plot()* function, which can create a wide variety of graphs (type *?plot* in the command line for details) and the *lattice()* package is also helpful. However, throughout this chapter I use Hadley Wickham's **ggplot2** package (Wickham, 2009). I have chosen to focus on this package because I like it. I wouldn't take it out for a romantic meal, but I do get genuinely quite excited by some of the stuff it can do. Just to be very clear about this, it's a very different kind of excitement than that evoked by a romantic meal with my wife.

The *ggplot2* package excites me because it is a wonderfully versatile tool. It takes a bit of time to master it (I still haven't really got to grips with the finer points of it), but once you have, it gives you an extremely flexible framework for displaying and annotating data. The second great thing about *ggplot2* is it is based on Tufte's recommendations about displaying data and Wilkinson's grammar of graphics (Wilkinson, 2005). Therefore, with basically no editing we can create Tufte-pleasing graphs. You can install *ggplot2* by executing the following command:

```
install.packages("ggplot2")
```

You then need to activate it by executing the command:

```
library(ggplot2)
```

4.4. Introducing ggplot2 ①

There are two ways to plot graphs with *ggplot2*: (1) do a quick plot using the *qplot()* function; and (2) build a plot layer by layer using the *ggplot()* function. Undoubtedly the **qplot()** function will get you started quicker; however, the **ggplot()** function offers greater versatility so that is the function that I will use throughout the chapter. I like a challenge.

There are several concepts to grasp that help you to understand how *ggplot2* builds graphs. Personally, I find some of the terminology a bit confusing so I apologize if occasionally I use different terms than those you might find in the *ggplot2* documentation.

4.4.1. The anatomy of a plot ①

A graph is made up of a series of layers. You can think of a layer as a plastic transparency with something printed on it. That 'something' could be text, data points, lines, bars, pictures of chickens, or pretty much whatever you like. To make a final image, these transparencies are placed on top of each other. Figure 4.5 illustrates this process: imagine you begin with a transparent sheet that has the axes of the graph drawn on it. On a second transparent sheet you have bars representing different mean scores. On a third transparency you have drawn error bars associated with each of the means. To make the final graph, you put these three layers together: you start with the axes, lay the bars on top of that, and finally lay the error bars on top of that. The end result is an error bar graph. You can extend the idea of layers beyond the figure: you could imagine having a layer that contains labels for the axes, or a title, and again, you simply lay these on top of the existing image to add more features to the graph.

As can be seen in Figure 4.5, each layer contains visual objects such as bars, data points, text and so on. Visual elements are known as *geoms* (short for 'geometric objects') in *ggplot2*. Therefore, when we define a layer, we have to tell **R** what geom we want displayed on that layer (do we want a bar, line dot, etc.?). These geoms also have aesthetic properties that determine what they look like and where they are plotted (do we want red bars or green ones? do we want our data point to be a triangle or a square? etc.). These aesthetics (*aes()* for short) control the appearance of graph elements (for example, their colour, size, style and location). Aesthetics can be defined in general for the whole plot, or individually for a specific layer. We'll come back to this point in due course.

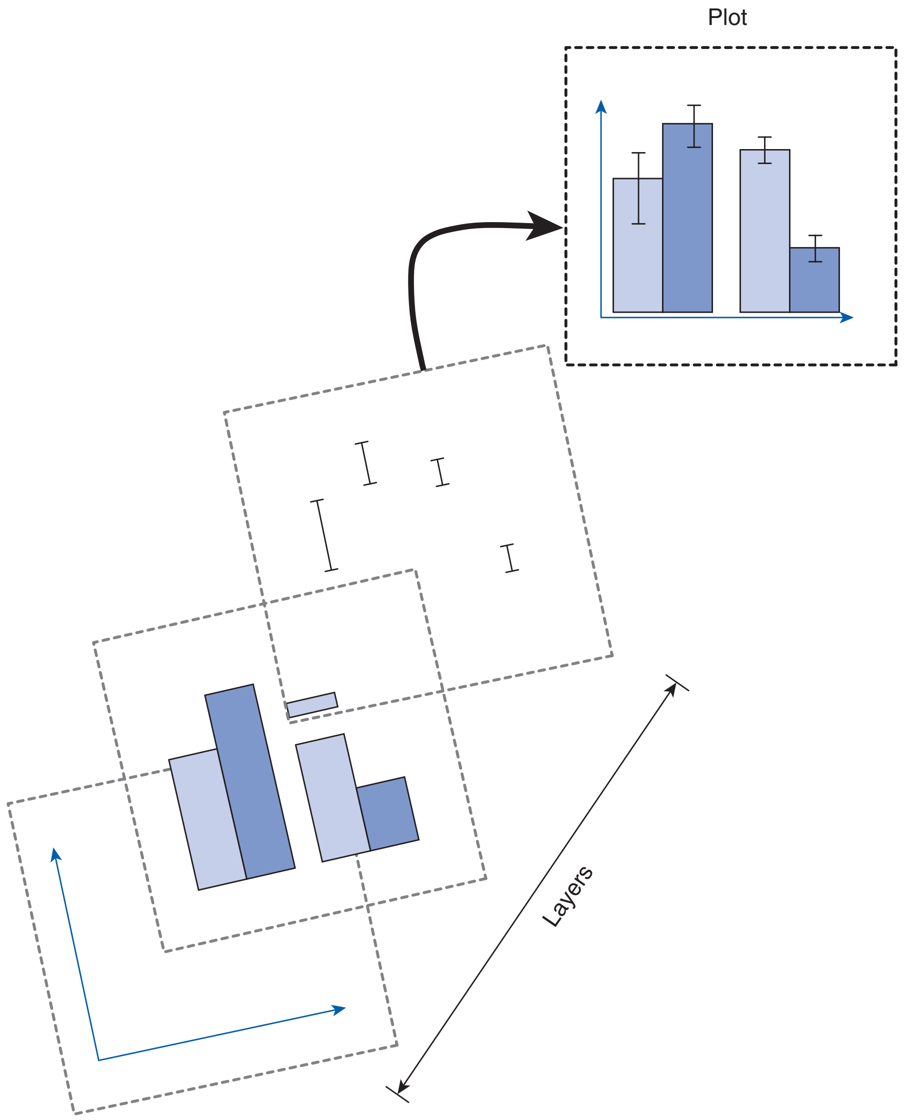


FIGURE 4.5 In *ggplot2* a plot is made up of layers

To recap, the finished plot is made up of layers, each layer contains some geometric element (such as bars, points, lines, text) known as a geom, and the appearance and location of these geoms (e.g., size, colour, shape used) is controlled by the aesthetic properties (*aes()*). These aesthetics can be set for all layers of the plot (i.e., defined in the plot as a whole) or can be set individually for each geom in a plot (Figure 4.6). We will learn more about geoms and aesthetics in the following sections.

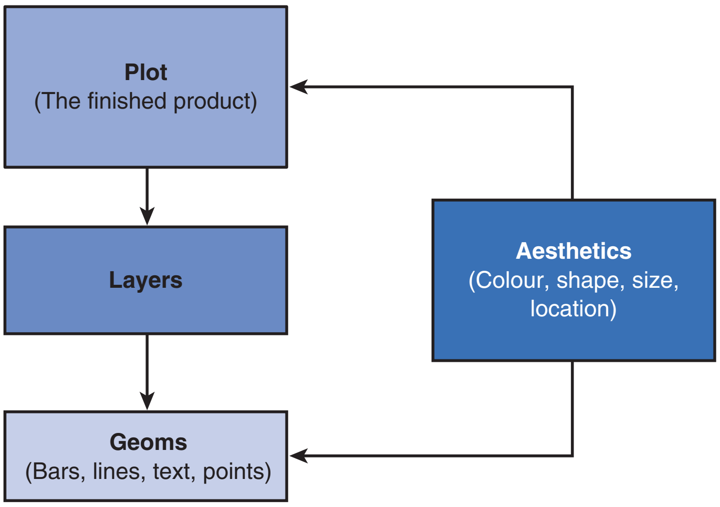


FIGURE 4.6 The anatomy of a graph

4.3.2. Geometric objects (geoms) ①

There are a variety of geom functions that determine what kind of geometric object is printed on a layer. Here is a list of a few of the more common ones that you might use (for a full list see the *ggplot2* website http://had.co.nz/ggplot2/):

- *geom_bar()*: creates a layer with bars representing different statistical properties.
- *geom_point()*: creates a layer showing the data points (as you would see on a scatterplot).
- *geom_line()*: creates a layer that connects data points with a straight line.
- *geom_smooth()*: creates a layer that contains a 'smoother' (i.e., a line that summarizes the data as a whole rather than connecting individual data points).
- *geom_histogram()*: creates a layer with a histogram on it.
- *geom_boxplot()*: creates a layer with a box–whisker diagram.
- *geom_text()*: creates a layer with text on it.
- *geom_density()*: creates a layer with a density plot on it.
- *geom_errorbar()*: creates a layer with error bars displayed on it.
- *geom_hline()* and *geom_vline()*: create a layer with a user-defined horizontal or vertical line, respectively.

Notice that each geom is followed by '()', which means that it can accept aesthetics that specify how the layer looks. Some of these aesthetics are required and others are optional. For example, if you want to use the text geom then you have to specify the text that you want to print and the position at which you want to print it (using *x* and *y* coordinates), but you do not have to specify its colour.

In terms of required aesthetics, the bare minimum is that each geom needs you to specify the variable or variables that the geom represents. It should be self-evident that *ggplot2* can't create the geom without knowing what it is you want to plot! Optional aesthetics take on default values but you can override these defaults by specifying a value. These are attributes of the geom such as the colour of the geom, the colour to fill the geom, the type

Table 4.1 Aesthetic properties associated with some commonly used geoms

	Required	***Optional***
geom_bar()	x: the variable to plot on the *x*-axis	colour size fill linetype weight alpha
geom_point()	x: the variable to plot on the *x*-axis y: the variable to plot on the *y*-axis	shape colour size fill alpha
geom_line()	x: the variable to plot on the *x*-axis y: the variable to plot on the *y*-axis	colour size linetype alpha
geom_smooth()	x: the variable to plot on the *x*-axis y: the variable to plot on the *y*-axis	colour size fill linetype weight alpha
geom_histogram()	x: the variable to plot on the *x*-axis	colour size fill linetype weight alpha
geom_boxplot()	x: the variable to plot ymin: lower limit of 'whisker' ymax: upper limit of 'whisker' lower: lower limit of the 'box' upper: upper limit of the 'box' middle: the median	colour size fill weight alpha
geom_text()	x: the horizontal coordinate of where the text should be placed y: the vertical coordinate of where the text should be placed label: the text to be printed all of these can be single values or variables containing coordinates and labels for multiple items	colour size angle hjust (horizontal adjustment) vjust (vertical adjustment) alpha
geom_density()	x: the variable to plot on the *x*-axis y: the variable to plot on the *y*-axis	colour size fill linetype weight alpha

Table 4.1 *(Continued)*

	Required	*Optional*
geom_errorbar()	x: the variable to plot ymin, ymax: lower and upper value of error bar	colour size linetype width alpha

geom_hline(),
geom_vline()
	yintercept = value
	xintercept = value
	(where value is the position on the x- or y-axis where you want the vertical/horizontal line)
		colour
		size
		linetype
		alpha

of line to use (solid, dashed, etc.), the shape of the data point (triangle, square, etc.), the size of the geom, and the transparency of the geom (known as alpha). Table 4.1 lists some common geoms and their required and optional aesthetic properties. Note that many of these aesthetics are common across geoms: for example, alpha, colour, linetype and fill can be specified for most of geoms listed in the table.

4.4.3. Aesthetics ①

We have already seen that aesthetics control the appearance of elements within a geom or layer. As already mentioned, you can specify aesthetics for the plot as a whole (such as the variables to be plotted, the colour, shape, etc.) and these instructions will filter down to any geoms in the plot. However, you can also specify aesthetics for individual geoms/layers and these instructions will override those of the plot as a whole. It is efficient, therefore, to specify things like the data to be plotted when you create the plot (because most of the time you won't want to plot different data for different geoms) but to specify idiosyncratic features of the geom's appearance within the geom itself. Hopefully, this process will become clear in the next section.

For now, we will simply look at *how* to specify aesthetics in a general sense. Figure 4.7 shows the ways in which aesthetics are specified. First, aesthetics can be set to a specific value (e.g., a colour such as red) or can be set to vary as a function of a variable (e.g., displaying data for different experimental groups in different colours). If you want to set an aesthetic to a specific value then you don't specify it within the *aes()* function, but if you want an aesthetic

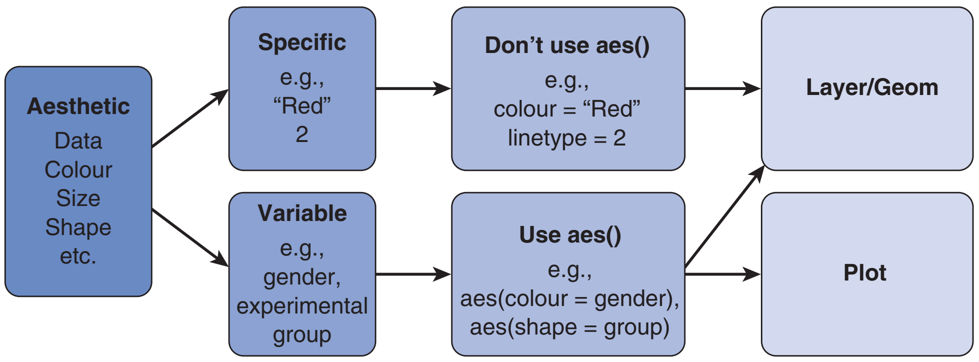


FIGURE 4.7 Specifying aesthetics in *ggplot2*

to vary then you need to place the instruction within *aes()*. Finally, you can set both specific and variable aesthetics at the layer or geom level of the plot, but you cannot set specific values at the plot level. In other words, if we want to set a specific value of an aesthetic we must do it within the *geom()* that we're using to create the particular layer of the plot.

Table 4.2 lists the main aesthetics and how to specify each one. There are, of course, others, and I can't cover the entire array of different aesthetics, but I hope to give you an idea of how to change some of the more common attributes that people typically want to change. For a comprehensive guide read Hadley Wickham's book (Wickham, 2009). It should be clear from Table 4.2 that most aesthetics are specified simply by writing the name of the aesthetic, followed by an equals sign, and then something that sets the value: this can be a variable (e.g., *colour = gender*, which would produce different coloured aesthetics for males and females) or a specific value (e.g., *colour = "Red"*).

Table 4.2 Specifying optional aesthetics

Aesthetic	***Option***	***Outcome***
Linetype	linetype = 1	Solid line (default)
	linetype = 2	Hashed
	linetype = 3	Dotted
	linetype = 4	Dot and hash
	linetype = 5	Long hash
	linetype = 6	Dot and long hash
Size	size = value	Replace 'value' with a value in mm (default size = 0.5). Larger values than 0.5 give you fatter lines/larger text/bigger points than the default whereas smaller values will produce thinner lines/smaller text and points than the default.
	e.g., size = 0.25	Produces lines/points/text of 0.25mm
Shape	shape = integer, shape = "x"	The integer is a value between 0 and 25, each of which specifies a particular shape. Some common examples are below. Alternatively, specify a single character in quotes to use that character (shape = "A" will plot each point as the letter A).
	shape = 0	Hollow square (15 is a filled square)
	shape = 1	Hollow circle (16 is a filled circle)
	shape = 2	Hollow triangle (17 is a filled triangle)
	shape = 3	'+'
	shape = 5	Hollow rhombus (18 for filled)
	shape = 6	Hollow inverted triangle
Colour	colour = "Name"	Simply type the name of a standard colour. For example, colour = "Red" will make the geom red.
	colour = "#RRGGBB"	Specify exact colours using the RRGGBB system. For example, colour = "#3366FF" produces a shade of blue, whereas colour = "#336633" produces a dark green.
Alpha	alpha(colour, value)	Colours can be made transparent by specifying alpha, which can range from 0 (fully transparent) to 1 (fully opaque). For example, alpha("Red", 0.5) will produce a half transparent red.

4.4.4. The anatomy of the ggplot() function ①

The command structure in *ggplot2* follows the anatomy of the plot described above in a very literal way. You begin by creating an object that specifies the plot. You can, at this stage set any aesthetic properties that you want to apply to all layers (and geoms) within the plot. Therefore, it is customary to define the variables that you want to plot at this top level. A general version of the command might look like this:

```
myGraph <- ggplot(myData, aes(variable for x axis, variable for y axis))
```

In this example, we have created a new graph object called *myGraph*, we have told *ggplot* to use the dataframe called *myData*, and we put the names of the variables to be plotted on the *x* (horizontal) and *y* (vertical) axis within the *aes()* function. Not to labour the point, but we could also set other aesthetic values at this top level. As a simple example, if we wanted our layers/geoms to display data from males and females in different colours then we could specify (assuming the variable **gender** defines whether a datum came from a man or a woman):

```
myGraph <- ggplot(myData, aes(variable for x axis, variable for y axis,
colour = gender))
```

In doing so any subsequent geom that we define will take on the aesthetic of producing different colours for males and females, assuming that this is a valid aesthetic for the particular geom (if not, the colour specification is ignored) and that we don't override it by defining a different colour aesthetic within the geom itself.

At this level you can also define options using the *opts()* function. The most common option to set at this level is a title:

```
+ opts(title = "Title")
```

Whatever text you put in the quotations will appear as your title exactly as you have typed it, so punctuate and capitalize appropriately.

So far we have created only the graph object: there are no graphical elements, and if you try to display *myGraph* you'll get an error. We need to add layers to the graph containing geoms or other elements such as labels. To add a layer we literally use the 'add' symbol (+). So, let's assume we want to add bars to the plot, we can execute this command:

```
myGraph + geom_bar()
```

This command takes the object *myGraph* that we have already created, and adds a layer containing bars to it. Now that there are graphical elements, *ggplot2* will print the graph to a window on your screen. If we want to also add points representing the data to this graph then we add '+ geom_point()' to the command and rerun it:

```
myGraph + geom_bar() + geom_point()
```

As you can see, every time you use a '+' you add a layer to the graph, so the above example now has two layers: bars and points. You can add any or all of the geoms that we have already described to build up your graph layer by layer. Whenever we specify a geom we can define an aesthetic for it that overrides any aesthetic setting for the plot as a whole. So, let's say we have defined a new graph as:

```
myGraph <- ggplot(myData, aes(variable for x axis, variable for y axis,
colour = gender))
```

but we want to add points that are blue (and do not vary by gender), then we can do this as:

```
myGraph + geom_point(colour = "Blue")
```

Note that because we've set a specific value we have not used the *aes()* function to set the colour. If we wanted our points to be blue triangles then we could simply add the shape command into the geom specification too:

```
myGraph + geom_point(shape = 17, colour =" "Blue")
```

We can also add a layer containing things other than geoms. For example, axis labels can be added by using the *labels()* function:

```
myGraph + geom_bar() + geom_point() + labels(x = "Text", y = "Text")
```

in which you replace the word "Text" (again keep the quotations) with the label that you want. You can also apply themes, faceting and options in a similar manner (see sections 4.4.6 and 4.10).

4.4.5. Stats and geoms ③

We have already encountered various geoms that map onto common plots used in research: *geom_histogram*, *geom_boxplot*, *geom_smooth*, *geom_bar* etc. (see Table 4.1). At face value it seems as though these geoms require you to generate the data necessary to plot them. For example, the boxplot geom requires that you tell it the minimum and maximum values of the box and the whiskers as well as the median. Similarly, the errorbar geom requires you to feed in the minimum and maximum values of the bars. Entering the values that the geom needs is certainly an option, but more often than not you'll want to just create plots directly from the raw data without having to faff about computing summary statistics. Luckily, *ggplot2* has some built-in functions called 'stats' that can be either used by a geom to get the necessary values to plot, or used directly to create visual elements on a layer of a plot.

Table 4.3 shows a selection of stats that geoms use to generate plots. I have focused only on the stats that will actually be used in this book, but there are others (for a full list see http://had.co.nz/ggplot2/). Mostly, these stats work behind the scenes: a geom uses them without you knowing about it. However, it's worth knowing about them because they enable you to adjust the properties of a plot. For example, imagine we want to plot a histogram, we can set up our plot object (*myHistogram*) as:

```
myHistogram <- ggplot(myData, aes(variable))
```

which has been defined as plotting the variable called **variable** from the dataframe *myData*. As we saw in the previous section, if we want a histogram, then we simply add a layer to the plot using the histogram geom:

```
myHistogram + geom_histogram()
```

That's it: a histogram will magically appear. However, behind the scenes the histogram geom is using the *bin* stat to generate the necessary data (i.e., to bin the data). We could get exactly the same histogram by writing:

```
myHistogram + geom_histogram(aes(y = ..count..))
```

The *aes(y = ..count..)* is simply telling *geom_histogram* to set the *y*-axis to be the *count* output variable from the *bin* stat, which *geom_histogram* will do by default. As we can see from Table 4.3, there are other variables we could use though. Let's say we wanted our histogram to show the density rather than the count. Then we can't rely on the defaults and we would have to specify that *geom_histogram* plots the *density* output variable from the *bin* stat on the *y*-axis:

```
myHistogram + geom_histogram(aes(y = ..density..))
```

Table 4.3 Some of the built-in 'stats' in *ggplot2*

Stat	***Function***	***Output Variables***	***Useful Parameters***	***Associated Geom***
bin	Bins data	**count**: number of points in bin **density**: density of points in bin, scaled to integrate to 1 **ncount**: count, scaled to maximum of 1 **ndensity**: density, scaled to maximum of 1	**binwidth**: bin width **breaks**: override bin width with specific breaks to use **width**: width of bars	histogram
boxplot	Computes the data necessary to plot a boxplot	**width**: width of boxplot **ymin**: lower whisker **lower**: lower hinge, 25% quantile **middle**: median **upper**: upper hinge, 75% quantile **ymax**: upper whisker		boxplot
density	Density estimation	**density**: density estimate **count**: density × number of points **scaled**: density estimate, scaled to maximum of 1		density
qq	Compute data for Q-Q plots	**sample**: sample quantiles **theoretical**: theoretical quantiles	quantiles	point
smooth	Create a smoother plot	**y**: predicted value **ymin**: lower pointwise CI around the mean **ymax**: upper pointwise CI around the mean **se**: standard error	**method**: e.g., lm, glm, gam, loess **formula**: formula for smoothing **se**: display CI (true by default) **level**: level of CI to use (0.95 by default)	smooth
summary	Summarize data		**fun.y**: determines the function to plot on the y-axis (e.g., fun.y = mean)	bar, errorbar, pointrange, linerange

Similarly, by default, *geom_histogram* uses a bin width of the range of scores divided by 30. We can use the parameters of the *bin* stat to override this default:

```
myHistogram + geom_histogram(aes(y = ..count..), binwidth = 0.4)
```

As such, it is helpful to have in mind the relationship between geoms and stats when plotting graphs. As we go through the chapter you will see how stats can be used to control what is produced by a geom, but also how stats can be used directly to make a layer of a plot.

4.4.6. Avoiding overplotting ②

Plots can become cluttered or unclear because (1) there is too much data to present in a single plot, and (2) data on a plot overlap. There are several positioning tools in *ggplot2* that help us to overcome these problems. The first is a position adjustment, defined very simply as:

```
position = "x"
```

in which *x* is one of five words:

- *dodge*: positions objects so that there is no overlap at the side.
- *stack* and *fill*: positions objects so that they are stacked. The *stack* instruction stacks objects in front of each other such that the largest object is at the back and smallest at the front. The *fill* instruction stacks objects on top of one other (to make up stacks of equal height that are partitioned by the stacking variable).
- *identity*: no position adjustment.
- *jitter*: adds a random offset to objects so that they don't overlap.

Another useful tool for avoiding overplotting is faceting, which basically means splitting a plot into subgroups. There are two ways to do this. The first is to produce a grid that splits the data displayed by the plot by combinations of other variables. This is achieved using *facet_grid()*. The second way is to split the data displayed by the plot by a single variable either as a long ribbon of individual graphs, or to wrap the ribbon onto the next line after a certain number of plots such that a grid is formed. This is achieved using *facet_wrap()*.

Figure 4.8 shows the differences between *facet_grid()* and *facet_wrap()* using a concrete example. Social networking sites such as Facebook offer an unusual opportunity to carefully manage your self-presentation to others (i.e., do you want to appear to be cool when in fact you write statistics books, appear attractive when you have huge pustules all over your face, fashionable when you wear 1980s heavy metal band t-shirts and so on).

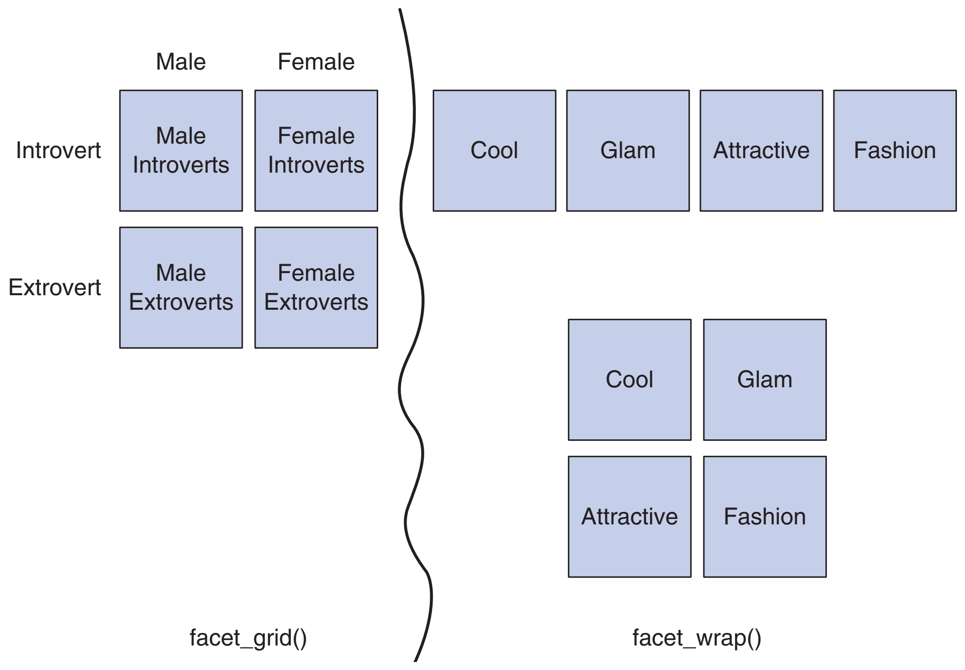


FIGURE 4.8 The difference between *facet_grid() and facet_wrap()*

A study was done that examined the relationship between narcissism and other people's ratings of your profile picture on Facebook (Ong et al., 2011). The pictures were rated on each of four dimensions: coolness, glamour, fashionableness and attractiveness. In addition, each person was measuresd on introversion/extroversion and also their gender recorded. Let's say we wanted to plot the relationship between narcissism and the profile picture ratings. We would have a lot of data because we have different types of rating, males and females and introverts and extroverts. We could use *facet_grid()* to produce plots of narcissism vs. photo rating for each combination of gender and extroversion. We'd end up with a grid of four plots (Figure 4.8). Alternatively, we could use *facet_wrap()* to split the plots by the type of rating (cool, glamorous, fashionable, attractive). Depending on how we set up the command, this would give us a ribbon of four plots (one for each type of rating) or we could wrap the plots to create a grid formation (Figure 4.8).

To use faceting in a plot, we add one of the following commands:

```
+ facet_wrap( ~ y, nrow = integer, ncol = integer)

+ facet_grid(x ~ y)
```

In these commands, *x* and *y* are the variables by which you want to facet, and for facet_wrap *nrow* and *ncol* are optional instructions to control how the graphs are wrapped: they enable you to specify (as an integer) the number of rows or columns that you would like. For example, if we wanted to facet by the variables **gender** and **extroversion**, we would add this command:

```
facet_grid(gender ~ extroversion)
```

If we wanted to draw different graphs for the four kinds of rating (**Rating_Type**), we could add:

```
+ facet_wrap( ~ Rating_Type)
```

This would give us an arrangement of graphs of one row and four columns (Figure 4.8); if we wanted to arrange these in a 2 by 2 grid (Figure 4.8) then we simply specify that we want two columns:

```
+ facet_wrap( ~ Rating_Type, ncol = 2)
```

or, indeed, two rows:

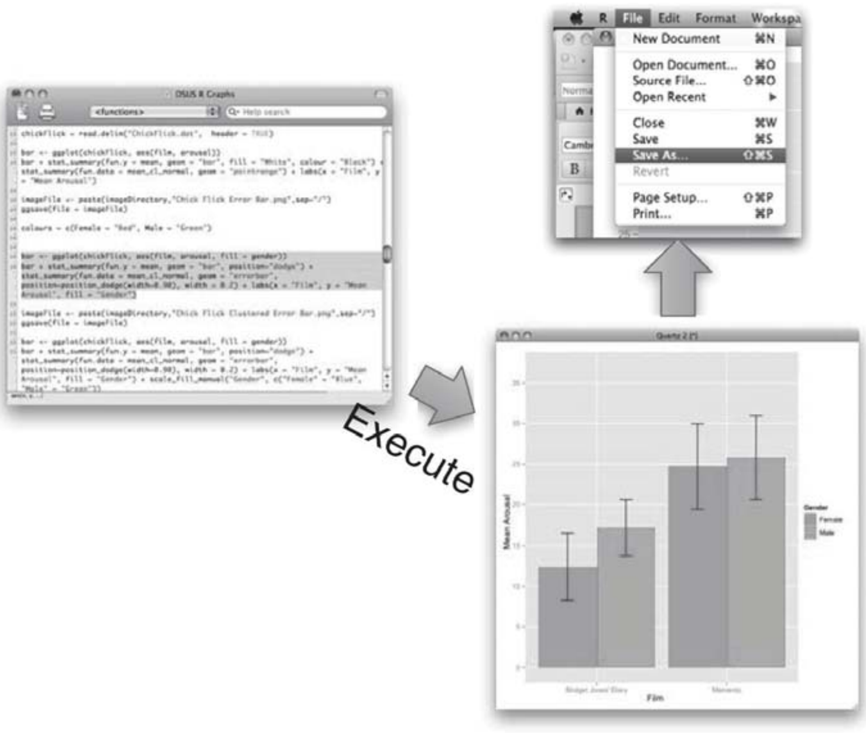
```
+ facet_wrap( ~ Rating_Type, nrow = 2)
```

4.4.7. Saving graphs ①

Having created the graph of your dreams, you'll probably want to save it somewhere. There are lots of options here. The simplest (but least useful in my view) is to use the *File* menu to save the plot as a pdf file. Figure 4.9 shows the stages in creating and saving a graph. Like anything in **R**, you first write a set of instructions to generate the graph. You select and execute these instructions. Having done this your graph appears in a new window. Click inside this window to make it active, then go to the **File⇒Save As** menu to open a standard dialog box to save the file in a location of your choice.

Personally, I prefer to use the *ggsave()* function, which is a versatile exporting function that can export as PostScript (.eps/.ps), tex (pictex), pdf, jpeg, tiff, png, bmp, svg and wmf (in Windows only). In its basic form, the structure of the function is very simple:

```
ggsave(filename)
```



```
chickFlick = read.delim("ChickFlick.dat",  header = TRUE)

bar <- ggplot(chickFlick, aes(film, arousal))
bar + stat_summary(fun.y = mean, geom = "bar", fill = "White", colour = "Black") +
stat_summary(fun.data = mean_cl_normal, geom = "pointrange") + labs(x = "Film", y
= "Mean Arousal")

imageFile <- paste(imageDirectory,"Chick Flick Error Bar.png",sep="/")
ggsave(file = imageFile)

colours = c(Female = "Red", Male = "Green")


bar <- ggplot(chickFlick, aes(film, arousal, fill = gender))
bar + stat_summary(fun.y = mean, geom = "bar", position="dodge") +
stat_summary(fun.data = mean_cl_normal, geom = "errorbar",
position=position_dodge(width=0.90), width = 0.2) + labs(x = "Film", y = "Mean
Arousal", fill = "Gender")

imageFile <- paste(imageDirectory,"Chick Flick Clustered Error Bar.png",sep="/")
ggsave(file = imageFile)

bar <- ggplot(chickFlick, aes(film, arousal, fill = gender))
bar + stat_summary(fun.y = mean, geom = "bar", position="dodge") +
stat_summary(fun.data = mean_cl_normal, geom = "errorbar",
position=position_dodge(width=0.90), width = 0.2) + labs(x = "Film", y = "Mean
Arousal", fill = "Gender") + scale_fill_manual("Gender", c("Female" = "Blue",
"Male" = "Green"))
```

FIGURE 4.9
Saving a graph manually

Here *filename* should be a text string that defines where you want to save the plot and the filename you want to use. The function automatically determines the format to which you want to export from the file extension that you put in the filename, so:

```
ggsave("Outlier Amazon.png")
```

will export as a png file, whereas

```
ggsave("Outlier Amazon.tiff")
```

will export as a tiff file. In the above examples I have specified only a filename, and these files will, therefore be saved in the current working directory (see section 3.4.4). You can, however, use a text string that defines an exact location, or create an object containing the file location that is then passed into the *ggsave()* function (see R's Souls' Tip 4.1). There are several other options you can specify, but mostly the defaults are fine. However, sometimes you might want to export to a specific size, and this can be done by defining the width and height of the image in inches: thus

```
ggsave("Outlier Amazon.tiff", width = 2, height = 2)
```

should save a tiff file that is 2 inches wide by 2 inches high.

4.4.8. Putting it all together: a quick tutorial ②

We have covered an enormous amount of ground in a short time, and have still only scratched the surface of what can be done with *ggplot2*. Also, we haven't actually plotted anything yet! In this section we will do a quick tutorial in which we put into practice various things that we have discussed in this chapter to give you some concrete experience of using *ggplot2* and to illustrate how some of the basic functionality the package works.

R's Souls' Tip 4.1 Saving graphs ③

By default *ggsave()* saves plots in your working directory (which hopefully you have set as something sensible). I find it useful sometimes to set up a specific location for saving images and to feed this into the *ggsave()* function. For example, executing:

```
imageDirectory<-file.path(Sys.getenv("HOME"), "Documents", "Academic", "Books",
"Discovering Statistics", "DSUR I Images")
```

uses the *file.path()* and *Sys.getenv()* functions to create an object *imageDirectory* which is a text string defining a folder called 'DSUR I Images', which is in a folder called 'Discovering Statistics' in a folder called 'Books' in a folder called 'Academic' which is in my main 'Documents' folder. On my computer (an iMac) this command sets *imageDirectory* to be:

```
"/Users/andyfield/Documents/Academic/Books/Discovering Statistics/DSUR I Images"
```

Sys.getenv("HOME") is a quick way to get the filepath of your home directory (in my case */Users/andyfield/*), and we use the *file.path()* function to paste the specified folder names together in an intelligent way based on the operating system that you use. Because I use a Mac it has connected the folders using an '/', but if I used Windows it would have used '\\' instead (because this is the symbol Windows uses to denote folders).

Having defined this location, we can use it to create a file path for a new image:

```
imageFile <- file.path(imageDirectory,"Graph.png")
ggsave(imageFile)
```

This produces a text string called *imageFile*, which is the filepath we have just defined (*imageDirectory*) with the filename that we want (*Graph.png*) added to it. We can reuse this code for a new graph by just changing the filename specified in *imageFile*:

```
imageFile <- file.path(imageDirectory,"Outlier Amazon.png")
ggsave(imageFile)
```

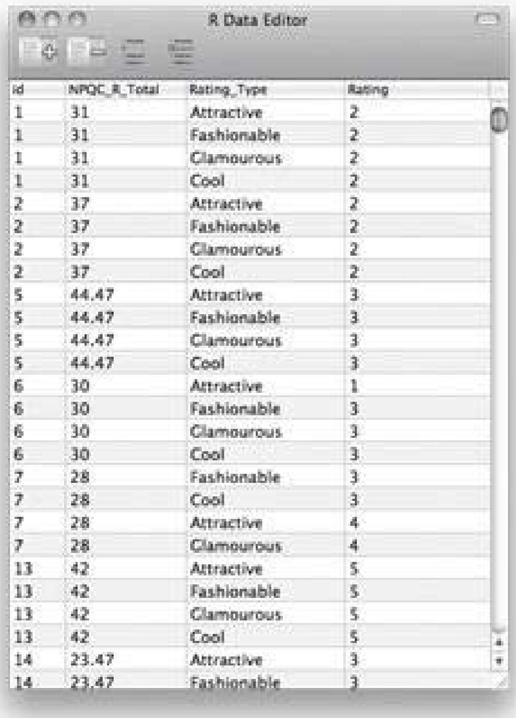
Earlier in the chapter we mentioned a study that looked at ratings of Facebook profile pictures (rated on coolness, fashion, attractiveness and glamour) and predicting them from how highly the person posting the picture scores on narcissism (Ong et al., 2011). The data are in the file **FacebookNarcissism.dat.**

First set your working directory to be the location of the data file (see section 3.4.4). Then create a dataframe called *facebookData* by executing the following command:

```
facebookData <- read.delim("FacebookNarcissism.dat", header = TRUE)
```

Figure 4.10 shows the contents of the dataframe. There are four variables:

1. **id**: a number indicating from which participant the profile photo came.
2. **NPQC_R_Total**: the total score on the narcissism questionnaire.
3. **Rating_Type**: whether the rating was for coolness, glamour, fashion or attractiveness (stored as strings of text).
4. **Rating**: the rating given (on a scale from 1 to 5).



id	NPQC_R_Total	Rating_Type	Rating
1	31	Attractive	2
1	31	Fashionable	2
1	31	Glamourous	2
1	31	Cool	2
2	37	Attractive	2
2	37	Fashionable	2
2	37	Glamourous	2
2	37	Cool	2
5	44.47	Attractive	3
5	44.47	Fashionable	3
5	44.47	Glamourous	3
5	44.47	Cool	3
6	30	Attractive	1
6	30	Fashionable	3
6	30	Glamourous	3
6	30	Cool	3
7	28	Fashionable	3
7	28	Cool	3
7	28	Attractive	4
7	28	Glamourous	4
13	42	Attractive	5
13	42	Fashionable	5
13	42	Glamourous	5
13	42	Cool	5
14	23.47	Attractive	3
14	23.47	Fashionable	3

FIGURE 4.10 The *facebookData* dataframe

First we need to create the plot object, which I have called, for want of a more original idea, *graph*. Remember that we initiate this object using the *ggplot()* function, which takes the following form:

```
graph <- ggplot(myData, aes(variable for x axis, variable for y axis))
```

To begin with, let's plot the relationship between narcissism (**NPQC_R_Total**) and the profile ratings generally (**Rating**). As such, we want **NPQC_R_Total** plotted on the *x*-axis and **Rating** on the *y*-axis. The dataframe containing these variables is called *facebookData* so we type and execute this command:

```
graph <- ggplot(facebookData, aes(NPQC_R_Total, Rating))
```

This command simply creates an object based on the *facebookData* dataframe and specifies the aesthetic mapping of variables to the *x*- and *y*-axes. Note that these mappings are contained within the *aes()* function. When you execute this command nothing will happen: we have created the object, but there is nothing to print.

If we want to see something then we need to take our object (*graph*) and add some visual elements. Let's start with something simple and add dots for each data point. This is done using the *geom_point()* function. If you execute the following command you'll see the graph in the top left panel of Figure 4.11 appear in a window on your screen:

```
graph + geom_point()
```

If we don't like the circles then we can change the shape of the points by specifying this for the geom. For example, executing:

```
graph + geom_point(shape = 17)
```

will change the dots to triangles (top right panel of Figure 4.11). By changing the number assigned to *shape* to other values you will see different shaped points (see section 4.4.3). If we want to change the size of the dots rather than the shape, this is easily done too by specifying a value (in mm) that you want to use for the 'size' aesthetic. Executing:

```
graph + geom_point(size = 6)
```

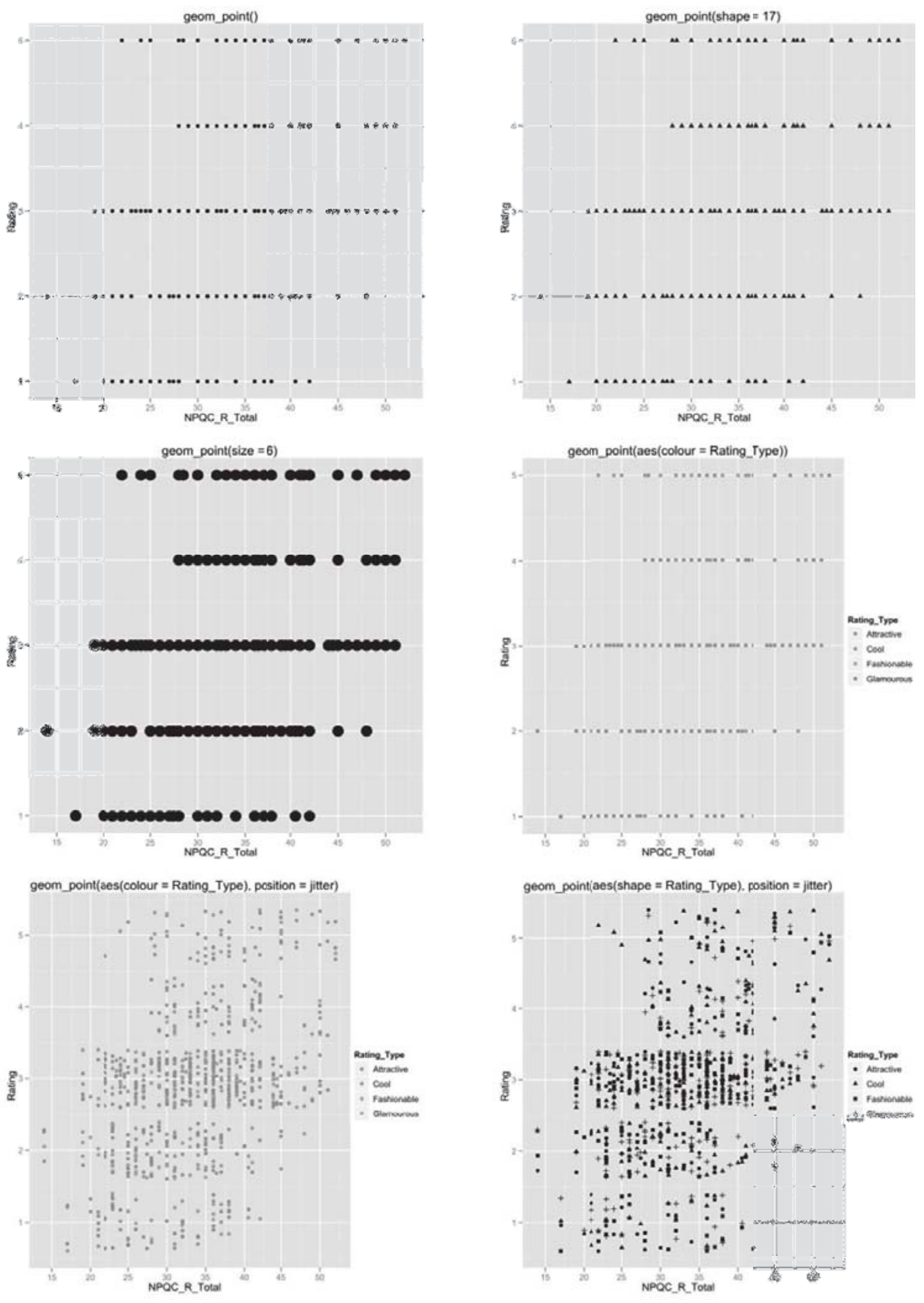


FIGURE 4.11 Different aesthetics for the point geom

creates the graph in the middle left panel of Figure 4.11. Note that the default shape has been used (because we haven't specified otherwise), but the size is larger than by default. At this stage we don't know whether a rating represented coolness, attractiveness or whatever. It would be nice if we could differentiate different ratings, perhaps by plotting them in different colours. We can do this by setting the colour aesthetic to be the variable **Rating_Type**. Executing this command:

```
graph + geom_point(aes(colour = Rating_Type))
```

creates the graph in the middle right panel of Figure 4.11, in which, onscreen, different types of ratings are now presented in different colours.[5]

[5] Note that here we set the colour aesthetic by enclosing it in aes() whereas in the previous examples we did not. This is because we're setting the value of colour based on a variable, rather than a single value.

We potentially have a problem of overplotting because there were a limited number of responses that people could give (notice that the data points fall along horizontal lines that represent each of the five possible ratings). To avoid this overplotting we could use the position option to add jitter:

```
graph + geom_point(aes(colour = Rating_Type), position = "jitter")
```

Notice that the command is the same as before; we have just added *position = "jitter"*. The results are shown in the bottom left panel of Figure 4.11; the dots are no longer in horizontal lines because a random value has been added to them to spread them around the actual value. It should be clear that many of the data points were sitting on top of each other in the previous plot.

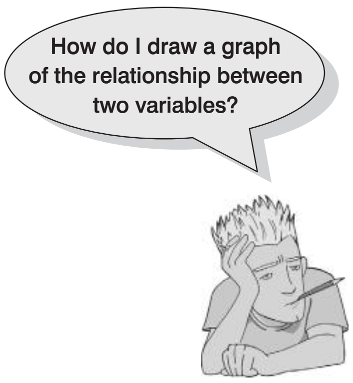
Finally, if we wanted to differentiate rating types by their shape rather than using a colour, we could change the colour aesthetic to be the shape aesthetic:

```
graph + geom_point(aes(shape = Rating_Type), position = "jitter")
```

Note how we have literally just changed *colour = Rating_Type* to *shape = Rating_Type*. The resulting graph in the bottom right panel of Figure 4.11 is the same as before except that the different types of ratings are now displayed using different shapes rather than different colours.

This very rapid tutorial has hopefully demonstrated how geoms and aesthetics work together to create graphs. As we now turn to look at specific kinds of graphs, you should hopefully have everything you need to make sense of how these graphs are created.

4.5. Graphing relationships: the scatterplot ①


How do I draw a graph of the relationship between two variables?

Sometimes we need to look at the relationships between variables. A **scatterplot** is a graph that plots each person's score on one variable against their score on another. A scatterplot tells us several things about the data, such as whether there seems to be a relationship between the variables, what kind of relationship it is and whether any cases are markedly different from the others. We saw earlier that a case that differs substantially from the general trend of the data is known as an *outlier* and such cases can severely bias statistical procedures (see Jane Superbrain Box 4.1 and section 7.7.1.1 for more detail). We can use a scatterplot to show us if any cases look like outliers.

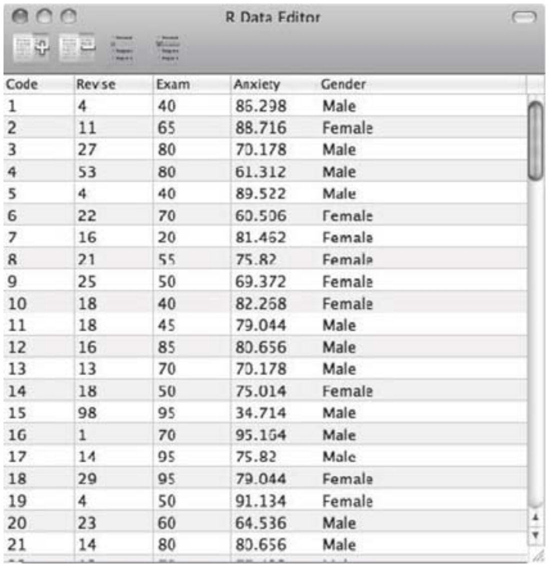
4.5.1. Simple scatterplot ①

This type of scatterplot is for looking at just two variables. For example, a psychologist was interested in the effects of exam stress on exam performance. So, she devised and validated a questionnaire to assess state anxiety relating to exams (called the Exam Anxiety Questionnaire, or EAQ). This scale produced a measure of anxiety scored out of 100. Anxiety was measured before an exam, and the percentage mark of each student on the exam was used to assess the exam performance. The first thing that the psychologist should do is draw a scatterplot of the two variables. Her data are in the file **ExamAnxiety.dat** and you should load this file into a dataframe called *examData* by executing:

```
examData <- read.delim("Exam Anxiety.dat", header = TRUE)
```

Figure 4.12 shows the contents of the dataframe. There are five variables:

1. **Code**: a number indicating from which participant the scores came.
2. **Revise**: the total hours spent revising.
3. **Exam**: mark on the exam as a percentage.
4. **Anxiety**: the score on the EAQ.
5. **Gender**: whether the participant was male or female (stored as strings of text).



Code	Revse	Exam	Anxiety	Gender
1	4	40	86.298	Male
2	11	65	88.716	Female
3	27	80	70.178	Male
4	53	80	61.312	Male
5	4	40	89.522	Male
6	22	70	60.506	Female
7	16	20	81.462	Female
8	21	55	75.82	Female
9	25	50	69.372	Female
10	18	40	82.268	Female
11	18	45	79.044	Male
12	16	85	80.656	Male
13	13	70	70.178	Male
14	18	50	75.014	Female
15	98	95	34.714	Male
16	1	70	95.164	Male
17	14	95	75.82	Male
18	29	95	79.044	Female
19	4	50	91.134	Female
20	23	60	64.536	Male
21	14	80	80.656	Male

FIGURE 4.12
The *examData* dataframe

First we need to create the plot object, which I have called *scatter*. Remember that we initiate this object using the *ggplot()* function. The contents of this function specify the dataframe to be used (*examData*) and any aesthetics that apply to the whole plot. I've said before that one aesthetic that is usually defined at this level is the variables that we want to plot. To begin with, let's plot the relationship between exam anxiety (**Anxiety**) and exam performance (**Exam**). We want **Anxiety** plotted on the *x*-axis and **Exam** on the *y*-axis. Therefore, to specify these variables as an aesthetic we type *aes(Anxiety, Exam)*. Therefore, the final command that we execute is:

```
scatter <- ggplot(examData, aes(Anxiety, Exam))
```

This command creates an object based on the *examData* dataframe and specifies the aesthetic mapping of variables to the *x*- and *y*-axes. When you execute this command nothing will happen: we have created the object, but there is nothing to print.

If we want to see something then we need to take our object (*scatter*) and add a layer containing visual elements. For a scatterplot we essentially want to add dots, which is done using the *geom_point()* function.

```
scatter + geom_point()
```

If we want to add some nice labels to our axes then we can also add a layer with these on using *labs()*:

```
scatter + geom_point() + labs(x = "Exam Anxiety", y = "Exam
Performance %")
```

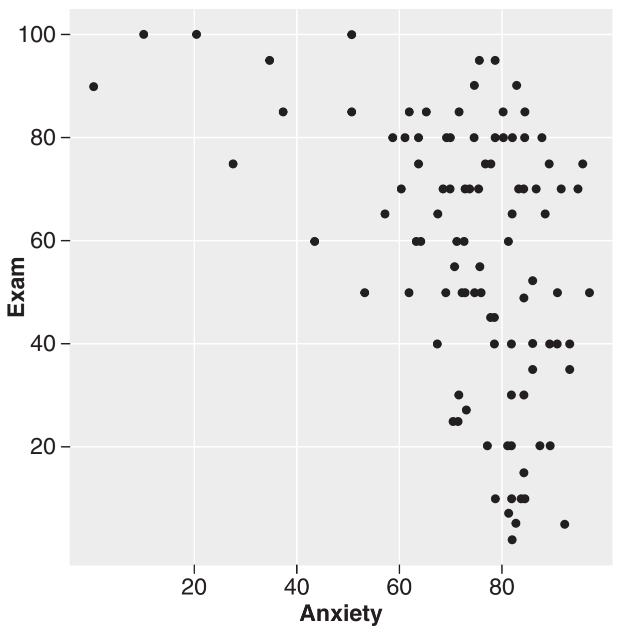
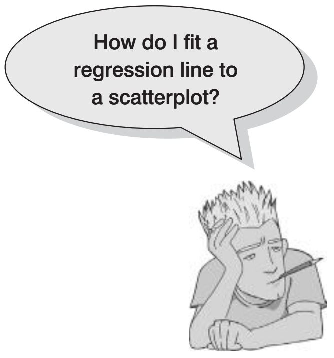


FIGURE 4.13 Scatterplot of exam anxiety and exam performance

If you execute this command you'll see the graph in Figure 4.13. The scatterplot tells us that the majority of students suffered from high levels of anxiety (there are very few cases that had anxiety levels below 60). Also, there are no obvious outliers in that most points seem to fall within the vicinity of other points. There also seems to be some general trend in the data, such that low levels of anxiety are almost always associated with high examination marks (and high anxiety is associated with a lot of variability in exam marks). Another noticeable trend in these data is that there were no cases having low anxiety and low exam performance – in fact, most of the data are clustered in the upper region of the anxiety scale.

4.5.2. Adding a funky line ①

You often see scatterplots that have a line superimposed over the top that summarizes the relationship between variables (this is called a **regression line** and we will discover more about it in Chapter 7). The scatterplot you have just produced won't have a funky line on it yet, but don't get too depressed because I'm going to show you how to add this line now.



In *ggplot2* terminology a regression line is known as a 'smoother' because it smooths out the lumps and bumps of the raw data into a line that summarizes the relationship. *The geom_smooth()* function provides the functionality to add lines (curved or straight) to summarize the pattern within your data.

To add a smoother to our existing scatterplot, we would simply add the *geom_smooth()* function and execute it:

```
scatter + geom_point() + geom_smooth() + labs(x = "Exam Anxiety",
y = "Exam Performance %")
```

Note that the command is exactly the same as before except that we have added a smoother in a new layer by typing + *geom_smooth()*. The resulting graph is shown in Figure 4.14. Note that the scatterplot now has a curved line (a 'smoother') summarizing the relationship between exam anxiety and exam performance. The shaded area around the line is the 95% confidence interval around the line. We'll see in due course how to remove this shaded error or to recolour it.

The smoothed line in Figure 4.14 is very pretty, but often we want to fit a straight line (or linear model) instead of a curved one. To do this, we need to change the 'method'

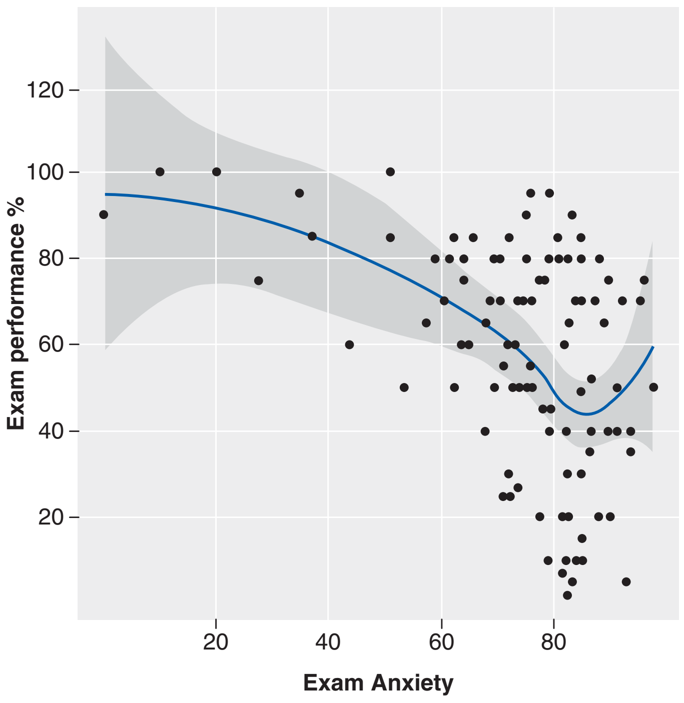


FIGURE 4.14 Scatterplot of exam anxiety against exam performance with a smoother added

associated with the smooth geom. In Table 4.3 we saw several methods that could be used for the smooth geom: *lm* fits a linear model (i.e., a straight line) and you could use *rlm* for a robust linear model (i.e., less affected by outliers).[6] So, to add a straight line (rather than curved) we change *geom_smooth()* to include this instruction:

```
+ geom_smooth(method = "lm")
```

We can also change the appearance of the line: by default it is blue, but if we wanted a red line then we can simply define this aesthetic within the geom:

```
+ geom_smooth(method = "lm", colour = "Red")
```

Putting this together with the code for the simple scatterplot, we would execute:

```
scatter <- ggplot(examData, aes(Anxiety, Exam))
scatter + geom_point() + geom_smooth(method = "lm", colour = "Red")+ labs(x
= "Exam Anxiety", y = "Exam Performance %")
```

The resulting scatterplot is shown in Figure 4.15. Note that it looks the same as Figure 4.13 and Figure 4.14 except that a red (because we specified the colour as red) regression line has been added.[7] As with our curved line, the regression line is surrounded by the 95% confidence interval (the grey area). We can switch this off by simply adding *se* = *F* (which is short for 'standard error = False') to the *geom_smooth()* function:

```
+ geom_smooth(method = "lm", se = F)
```

We can also change the colour and transparency of the confidence interval using the *fill* and *alpha* aesthetics, respectively. For example, if we want the confidence interval to be blue like the line itself, and we want it fairly transparent we could specify:

```
geom_smooth(method = "lm", alpha = 0.1, fill = "Blue")
```

[6] You must have the *MASS* package loaded to use this method.

[7] You'll notice that the figure doesn't have a red line but what you see on your screen does, that's because this book isn't printed in colour which makes it tricky for us to show you the colourful delights of **R**. In general, use the figures in the book as a guide only and read the text with reference to what you actually see on your screen.

FIGURE 4.15 A simple scatterplot with a regression line added

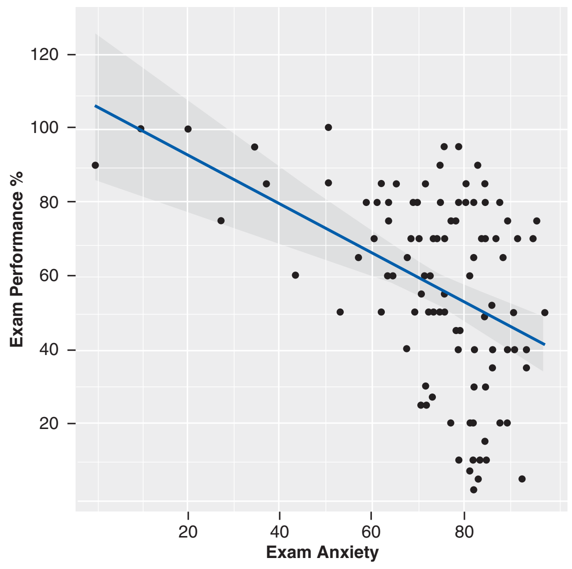
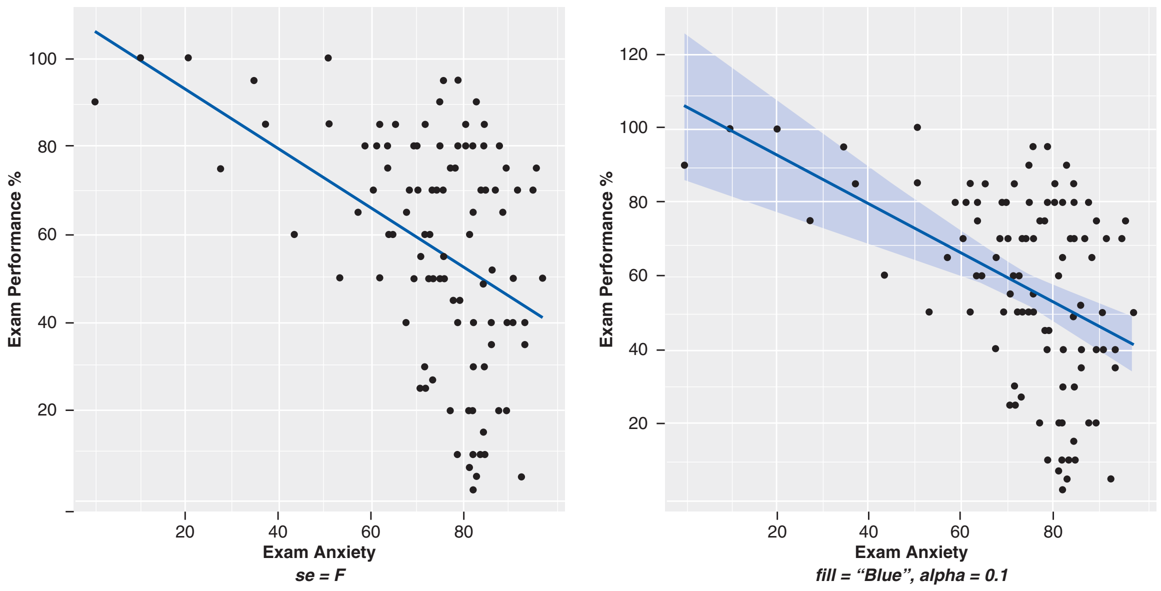


FIGURE 4.16 Manipulating the appearance of the confidence interval around the regression line



Note that transparency can take a value from 0 (fully transparent) to 1 (fully opaque) and so we have set a fairly transparent colour by using 0.1 (after all we want to see the data points underneath). The impact of these changes can be seen in Figure 4.16.

4.5.3. Grouped scatterplot ①

What if we want to see whether male and female students had different reactions to exam anxiety? To do this, we need to set **Gender** as an aesthetic. This is fairly straightforward. First, we define gender as a colour aesthetic when we initiate the plot object:

```
scatter <- ggplot(examData, aes(Anxiety, Exam, colour = Gender))
```

Note that this command is exactly the same as the previous example, except that we have added 'colour = Gender' so that any geoms we define will be coloured differently for men and women. Therefore, if we then execute:

```
scatter + geom_point + geom_smooth(method = "lm")
```

we would have a scatterplot with different coloured dots and regression lines for men and women. It's as simple as that. However, our lines would have confidence intervals and both intervals would be shaded grey, so we could be a little more sophisticated and add some instructions into *geom_smooth()* that tells it to also colour the confidence intervals according to the **Gender** variable:

```
scatter + geom_point() + geom_smooth(method = "lm", aes(fill = Gender), alpha
= 0.1)
```

Note that we have used *fill* to specify that the confidence intervals are coloured according to **Gender** (note that because we are specifying a variable rather than a single colour we have to place this option within *aes()*). As before, we have also manually set the transparency of the confidence intervals to be 0.1.

As ever, let's add some labels to the graph:

```
+ labs(x = "Exam Anxiety", y = "Exam Performance %", colour = "Gender")
```

Note that by specifying a label for 'colour' I am setting the label that will be used on the legend of the graph. The finished command to be executed will be:

```
scatter + geom_point() + geom_smooth(method = "lm", aes(fill = Gender), alpha
= 0.1) + labs(x = "Exam Anxiety", y = "Exam Performance %", colour = "Gender")
```

Figure 4.17 shows the resulting scatterplot. The regression lines tell us that the relationship between exam anxiety and exam performance was slightly stronger in males (the line is steeper) indicating that men's exam performance was more adversely affected by anxiety than women's exam anxiety. (Whether this difference is significant is another issue – see section 6.7.1.)

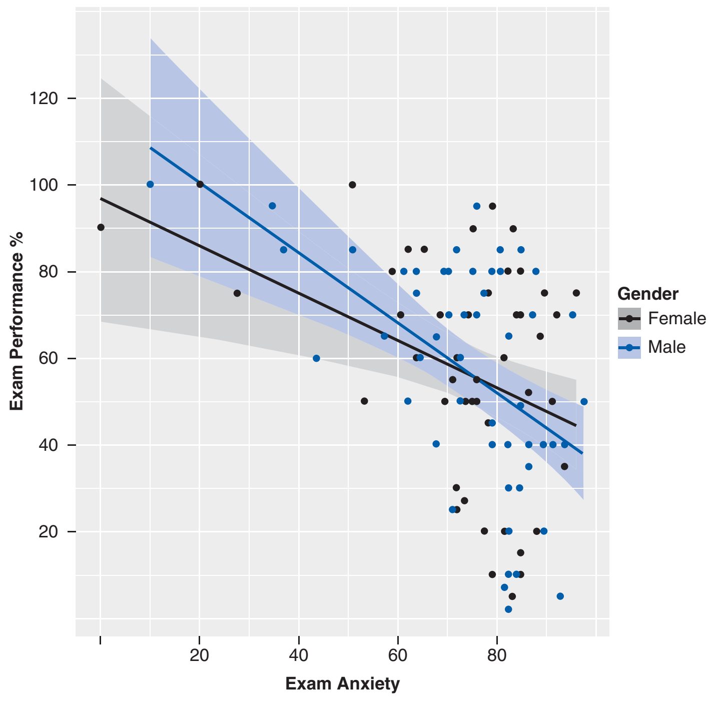


FIGURE 4.17 Scatterplot of exam anxiety and exam performance split by gender

SELF-TEST

✓ Go back to the Facebook narcissism data from the earlier tutorial. Plot a graph that shows the pattern in the data using only a line.
✓ Plot different coloured lines for the different types of rating (cool, fashionable, attractive, glamorous).
✓ Add a layer displaying the raw data as points.
✓ Add labels to the axes.

4.6. Histograms: a good way to spot obvious problems ①

In this section we'll look at how we can use frequency distributions to screen our data.[8] We'll use an example to illustrate what to do. A biologist was worried about the potential health effects of music festivals. So, one year she went to the Download Music Festival[9] (for those of you outside the UK, you can pretend it is Roskilde Festival, Ozzfest, Lollopalooza, Wacken or something) and measured the hygiene of 810 concert-goers over the three days of the festival. In theory each person was measured on each day but because it was difficult to track people down, there were some missing data on days 2 and 3. Hygiene was measured using a standardized technique (don't worry, it *wasn't* licking the person's armpit) that results in a score ranging between 0 (you smell like a corpse that's been left to rot up a skunk's arse) and 4 (you smell of sweet roses on a fresh spring day). Now I know from bitter experience that sanitation is not always great at these places (the Reading Festival seems particularly bad) and so this researcher predicted that personal hygiene would go down dramatically over the three days of the festival. The data file, **DownloadFestival.dat**, can be found on the companion website. We encountered histograms (frequency distributions) in Chapter 1; we will now learn how to create one in **R** using these data.

SELF-TEST

✓ What does a histogram show?

Load the data into a dataframe (which I've called *festivalData*); if you need to refresh your memory on data files and dataframes see section 3.5. Assuming you have set the working directory to be where the data file is stored, you can create the dataframe by executing this command:

```
festivalData <- read.delim("DownloadFestival.dat", header = TRUE)
```

Now we need to create the plot object and define any aesthetics that apply to the plot as a whole. I have called the object *festivalHistogram*, and have created it using the *ggplot()*

[8] An alternative way to graph the distribution is a density plot, which we'll discuss later.

[9] http://www.downloadfestival.co.uk

function. The contents of this function specify the dataframe to be used (*festivalData*) and any aesthetics that apply to the whole plot. I've said before that one aesthetic that is usually defined at this level is the variables that we want to plot. To begin with let's plot the hygiene scores for day 1, which are in the variable **day1**. Therefore, to specify this variable as an aesthetic we type *aes(day1)*. I have also decided to turn the legend off so I have added *opts(legend.position = "none")* to do this (see R's Souls' Tip 4.2):

```
festivalHistogram <- ggplot(festivalData, aes(day1)) + opts(legend.position
= "none")
```

Remember that having executed the above command we have an object but no graphical layers, so we will see nothing. To add the graphical layer we need to add the histogram geom to our existing plot:

```
festivalHistogram + geom_histogram()
```

Executing this command will create a graph in a new window. If you are happy using the default options then this is all there is to it; sit back and admire your efforts. However, we can tidy the graph up a bit. First, we could change the bin width. I would normally play around with different bin widths to get a feel for the distribution. To save time, let's just change it to 0.4. We can do this by inserting a command within the histogram geom:

```
+ geom_histogram(binwidth = 0.4)
```

We should also provide more informative labels for our axes using the *labs()* function:

```
+ labs(x = "Hygiene (Day 1 of Festival)", y = "Frequency")
```

As you can see, I have simply typed in the labels I want (within quotation marks) for the horizontal (*x*) and vertical (*y*) axes. Making these two changes leaves us with this command, which we must execute to see the graph:

```
festivalHistogram + geom_histogram(binwidth = 0.4) + labs(x = "Hygiene (Day
1 of Festival)", y = "Frequency")
```

The resulting histogram is shown in Figure 4.18. The first thing that should leap out at you is that there appears to be one case that is very different than the others. All of the

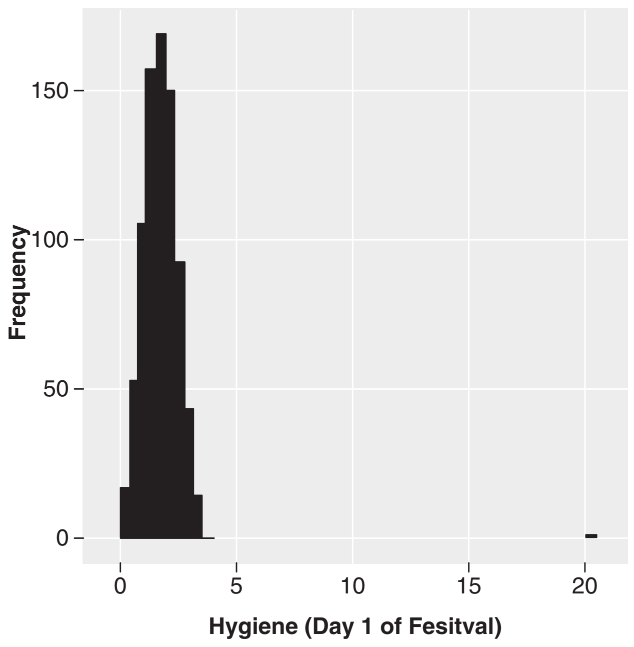


FIGURE 4.18 Histogram of the day 1 Download Festival hygiene scores

scores appear to be squashed up at one end of the distribution because they are all less than 5 (yielding a very pointy distribution) except for one, which has a value of 20. This is an **outlier**: a score very different than the rest (Jane Superbrain Box 4.1). Outliers bias the mean and inflate the standard deviation (you should have discovered this from the self-test tasks in Chapters 1 and 2) and screening data is an important way to detect them. You can look for outliers in two ways: (1) graph the data with a histogram (as we have done here) or a boxplot (as we will do in the next section); or (2) look at *z*-scores (this is quite complicated, but if you want to know see Jane Superbrain Box 4.2).

The outlier shown on the histogram is particularly odd because it has a score of 20, which is above the top of our scale (remember our hygiene scale ranged only from 0 to 4) and so it must be a mistake (or the person had obsessive compulsive disorder and had washed themselves into a state of extreme cleanliness).

R's Souls' Tip 4.2 Removing legends ③

By default *ggplot2* produces a legend on the right-hand side of the plot. Mostly this legend is a useful thing to have. However, there are occasions when you might like it to go away. This is achieved using the *opts()* function either when you set up the plot object, or when you add layers to the plot. To remove the legend just add:

```
+ opts(legend.position="none")
```

For example, either

```
myGraph <- ggplot(myData, aes(variable for x axis, variable for y axis)) + opts(legend.
position="none")
```

or

```
myGraph <- ggplot(myData, aes(variable for x axis, variable for y axis))
myGraph + geom_point() + opts(legend.position="none")
```

will produce a graph without a figure legend.

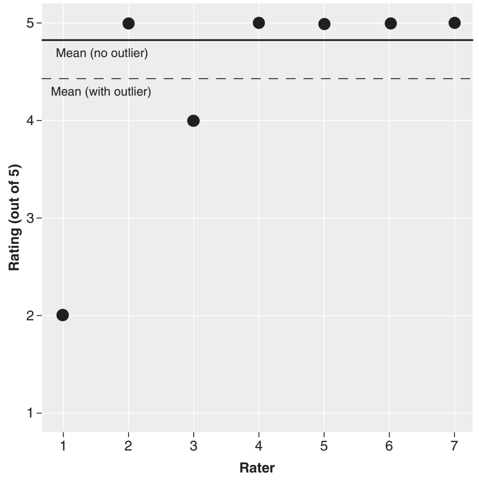
4.7. Boxplots (box–whisker diagrams) ①

Boxplots or box–whisker diagrams are really useful ways to display your data. At the centre of the plot is the median, which is surrounded by a box the top and bottom of which are the limits within which the middle 50% of observations fall (the interquartile range). Sticking out of the top and bottom of the box are two whiskers that extend to one and a half times the interquartile range. First, we will plot some using *ggplot2* and then we'll look at what they tell us in more detail. In the data file of hygiene scores we also have information about the gender of the concert-goer. Let's plot this information as well. To make our boxplot of the day 1 hygiene scores for males and females, we will need to set the variable **Gender** as an aesthetic. The simplest way to do this is just to specify **Gender** as the variable to be plotted on the *x*-axis, and the hygiene scores (**day1**) to be the variable plotted on the *y*-axis. As such, when we initiate

JANE SUPERBRAIN 4.1

What is an outlier? ①

An outlier is a score very different from the rest of the data. When we analyse data we have to be aware of such values because they bias the model we fit to the data. A good example of this bias can be seen by looking at the mean. When I published my first book (the first edition of the SPSS version of this book), I was quite young, I was very excited and I wanted everyone in the world to love my new creation and me. Consequently, I obsessively checked the book's ratings on Amazon.co.uk. These ratings can range from 1 to 5 stars. Back in 2002, my first book had seven ratings (in the order given) of 2, 5, 4, 5, 5, 5, and 5. All but one of these ratings are fairly similar (mainly 5 and 4) but the first rating was quite different from the rest – it was a rating of 2 (a mean and horrible rating). The graph plots seven reviewers on the horizontal axis and their ratings on the vertical axis and there is also a horizontal line that represents the mean rating (4.43 as it happens). It should be clear that all of the scores except one lie close to this line. The score of 2 is very different and lies some way below the mean. This score is an example of an outlier – a weird and unusual person (sorry, I mean score) that deviates from the rest of humanity (I mean, data set). The dashed horizontal line represents the mean of the scores when the outlier is not included (4.83). This line is higher than the original mean, indicating that by ignoring this score the mean increases (it increases by 0.4). This example shows how a single score, from some mean-spirited badger turd, can bias the mean; in this case the first rating (of 2) drags the average down. In practical terms this had a bigger implication because Amazon rounded off to half numbers, so that single score made a difference between the average rating reported by Amazon as a generally glowing 5 stars and the less impressive 4.5 stars. (Nowadays Amazon sensibly produces histograms of the ratings and has a better rounding system.) Although I am consumed with bitterness about this whole affair, it has at least given me a great example of an outlier! (Data for this example were taken from http://www.amazon.co.uk/ in about 2002.)



our plot object rather than set a single variable as an aesthetic as we did for the histogram (*aes(day1)*), we set **Gender** and **day1** as variables (*aes(Gender, day1)*). Having initiated the plot object (I've called it *festivalBoxplot*), we can simply add the boxplot geom as a layer (*+ geom_boxplot()*) and add some axis labels with the *labs()* function as we did when we created a histogram. To see the graph we therefore simply execute these two lines of code:

```
festivalBoxplot <- ggplot(festivalData, aes(gender, day1))
festivalBoxplot + geom_boxplot() + labs(x = "Gender", y = "Hygiene (Day 1 of
Festival)")
```

The resulting boxplot is shown in Figure 4.19. It shows a separate boxplot for the men and women in the data. Note that the outlier that we detected in the histogram is shown up as a point on the boxplot (we can also tell that this case was a female). An outlier is an extreme score, so the easiest way to find it is to sort the data:

```
festivalData<-festivalData[order(festivalData$day1),]
```

SELF-TEST

✓ Remove the outlier and replot the histogram.

JANE SUPERBRAIN 4.2

Using z-scores to find outliers ③

To check for outliers we can look at *z*-scores. We saw in section 1.7.4 that *z*-scores are simply a way of standardizing a data set by expressing the scores in terms of a distribution with a mean of 0 and a standard deviation of 1. In doing so we can use benchmarks that we can apply to any data set (regardless of what its original mean and standard deviation were). To look for outliers we could convert our variable to *z*-scores and then count how many fall within certain important limits. If we take the absolute value (i.e., we ignore whether the *z*-score is positive or negative) then in a normal distribution we'd expect about 5% to have absolute values greater than 1.96 (we often use 2 for convenience), and 1% to have absolute values greater than 2.58, and none to be greater than about 3.29.

I have written a function that gets **R** to count them for you called *outlierSummary()*. To use the function you need to load the package associated with this book (see section 3.4.5), you then simply insert the name of the variable that you would like summarized into the function and execute it. For example, to count the number of *z*-scores with absolute values above our three cut-off values in the **day2** variable, we can execute:

```
outlierSummary(festivalData$day2)
```

```
Absolute z-score greater than 1.96 =  6.82 %
Absolute z-score greater than 2.58 =  2.27 %
Absolute z-score greater than 3.29 =  0.76 %
```

The output produced by this function is shown above. We would expect to see 5% (or less) with an absolute value greater than 1.96, 1% (or less) with an absolute value greater than 2.58, and we'd expect no cases above 3.29 (these cases are significant outliers). For hygiene scores on day 2 of the festival, 6.82% of *z*-scores had absolute values greater than 1.96. This is slightly more than the 5% we would expect in a normal distribution. Looking at values above 2.58, we would expect to find only 1%, but again here we have a higher value of 2.27%. Finally, we find that 0.76% of cases were above 3.29 (so 0.76% are significant outliers). This suggests that there may be slightly too many outliers in this variable and we might want to do something about them.

OLIVER TWISTED

Please, Sir, can I have some more … complicated stuff?

'Graphs are for laughs, and functions are full of fun' thinks Oliver as he pops a huge key up his nose and starts to wind the clockwork mechanism of his brain. We don't look at functions for another couple of chapters, which is why I've skipped over the details of how the *outlierSummary()* function works. If, like Oliver, you like to wind up your brain, the additional material for this chapter, on the companion website, explains how I wrote the function. If that doesn't quench your thirst for knowledge then you're a grain of salt.

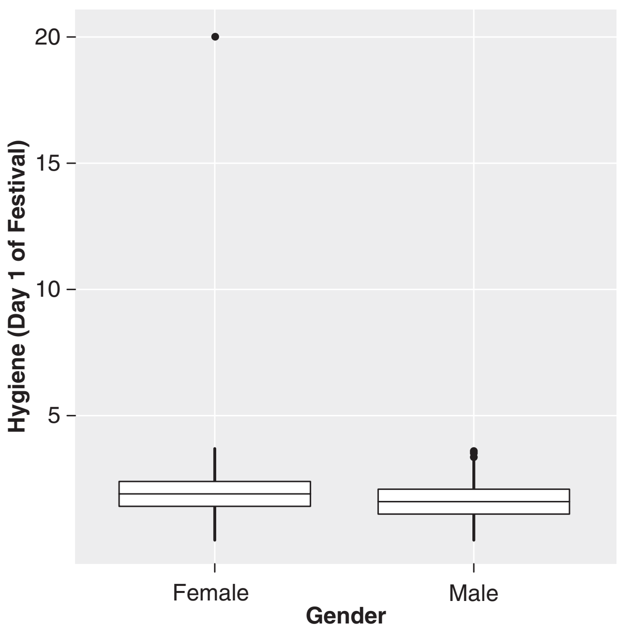


FIGURE 4.19 Boxplot of hygiene scores on day 1 of the Download Festival split by gender

This command takes *festivalData* and sorts it by the variable **day1**. All we have to do now is to look at the last case (i.e., the largest value of **day1**) and change it. The offending case turns out to be a score of 20.02, which is probably a mistyping of 2.02. We'd have to go back to the raw data and check. We'll assume we've checked the raw data and it should be 2.02, and that we've used R Commander's data editor (see section 3.6 or the online materials for this chapter) to replace the value 20.02 with the value 2.02 before we continue this example.

SELF-TEST

✓ Now we have removed the outlier in the data, try replotting the boxplot. The resulting graph should look like Figure 4.20.

Figure 4.20 shows the boxplots for the hygiene scores on day 1 after the outlier has been corrected. Let's look now in more detail about what the boxplot represents. First, it shows us the lowest score (the lowest point of the bottom whisker, or a dot below it) and the highest (the highest point of the top whisker of each plot, or a dot above it). Comparing the males and females we can see they both had similar low scores (0, or very smelly) but the women had a slightly higher top score (i.e., the most fragrant female was more hygienic than the cleanest male).

The lowest edge of the white box is the lower quartile (see section 1.7.3); therefore, the distance between the bottom of the vertical line and the lowest edge of the white box is the range between which the lowest 25% of scores fall. This range is slightly larger for women than for men, which means that if we take the most unhygienic 25% females then there is more variability in their hygiene scores than the lowest 25% of males. The box (the white area) shows the interquartile range (see section 1.7.3): that is, 50% of the scores are bigger than the lowest part of the white area but smaller than the top part of the white area. These boxes are of similar size in the males and females.

The top edge of the white box shows the value of the upper quartile (see section 1.7.3); therefore, the distance between the top edge of the white box and the top of the vertical line shows the range between which the top 25% of scores fall. In the middle of the white box is a line that represents the value of the median (see section 1.7.2). The median for females is higher than for males, which tells us that the middle female scored higher, or was more hygienic, than the middle male.

Boxplots show us the range of scores, the range between which the middle 50% of scores fall, and the median, the upper quartile and lower quartile score. Like histograms, they also tell us whether the distribution is symmetrical or skewed. If the whiskers are the same length then the distribution is symmetrical (the range of the top and bottom 25% of scores is the same); however, if the top or bottom whisker is much longer than the opposite whisker then the distribution is asymmetrical (the range of the top and bottom 25% of scores is different). Finally, you'll notice some dots above the male boxplot. These are cases that are deemed to be outliers. In Chapter 5 we'll see what can be done about these outliers.

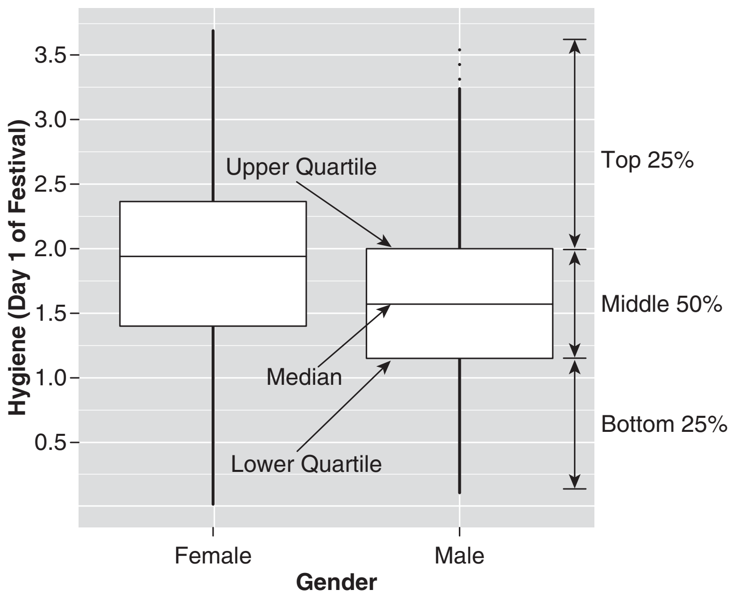


FIGURE 4.20 Boxplot of hygiene scores on day 1 of the Download Festival split by gender, after the outlier has been corrected

SELF-TEST

✓ Produce boxplots for the day 2 and day 3 hygiene scores and interpret them.

4.8. Density plots ①

Density plots are rather similar to histograms except that they smooth the distribution into a line (rather than bars). We can produce a density plot in exactly the same way as a histogram, except using the density geom: *geom_density()*. Assuming you have removed the outlier for the festival data set,[10] initiate the plot (which I have called *density*) in the same way as for the histogram:

```
density <- ggplot(festivalData, aes(day1))
```

[10] If you haven't there is a data file with it removed and you can load this into a dataframe called *festivalData* by executing:

```
festivalData <- read.delim("DownloadFestival(No Outlier).dat", header = TRUE)
```

Then, to get the plot simply add the *density_geom()* function:

```
density + geom_density()
```

We can also add some labels by including:

```
+ labs(x = "Hygiene (Day 1 of Festival)", y = "Density Estimate")
```

in the command. The resulting plot is shown in Figure 4.21.

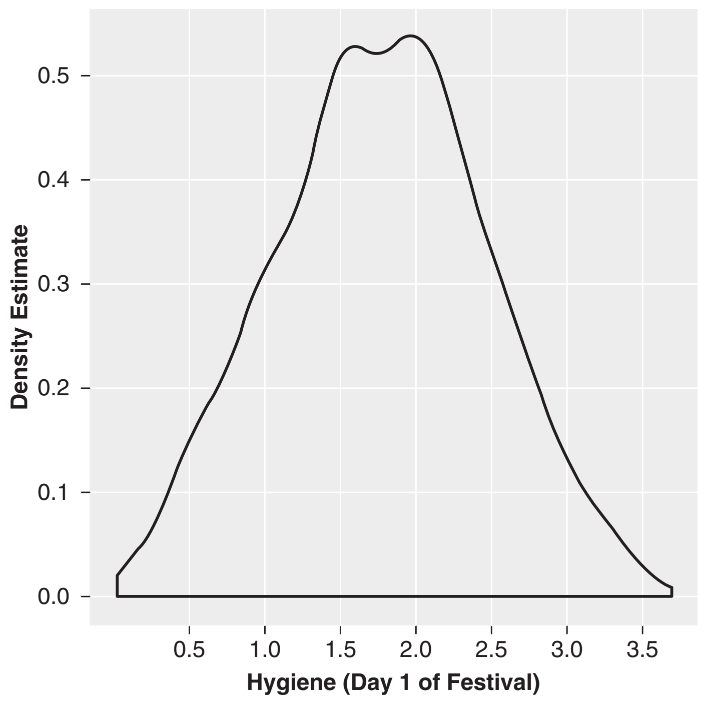


FIGURE 4.21
A density plot of the Download Festival data

4.9. Graphing means ③

4.9.1. Bar charts and error bars ②

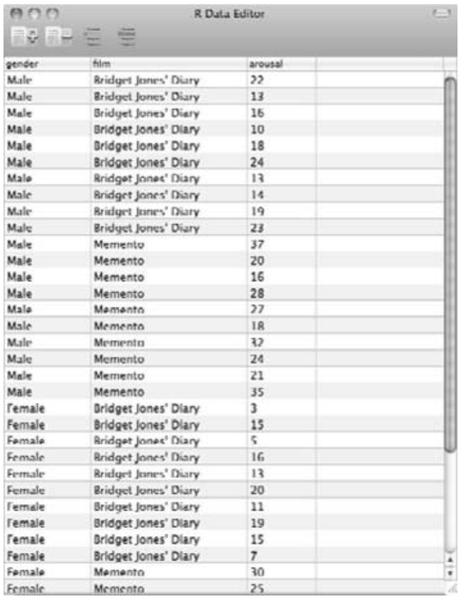
Bar charts are a common way for people to display means. The *ggplot2* package does not differentiate between research designs, so you plot bar charts in the same way regardless of whether you have an independent, repeated-measures or mixed design. Imagine that a film company director was interested in whether there was really such a thing as a 'chick flick' (a film that typically appeals to women more than men). He took 20 men and 20 women and showed half of each sample a film that was supposed to be a 'chick flick' (*Bridget Jones's Diary*), and the other half of each sample a film that didn't fall into the category of 'chick flick' (*Memento*, a brilliant film by the way). In all cases he measured their physiological arousal as an indicator of how much they enjoyed the film. The data are in a file called **ChickFlick.dat** on the companion website. Load this file into a dataframe called *chickFlick* by executing this command (I'm assuming you have set the working directory to be where the data file is stored):

```
chickFlick <- read.delim("ChickFlick.dat",  header = TRUE)
```

Figure 4.22 shows the data. Note there are three variables:

- **gender**: specifies the gender of the participant as text.
- **film**: specifies the film watched as text.
- **arousal**: is their arousal score.

Each row in the data file represents a different person.



gender	film	arousal
Male	Bridget Jones' Diary	22
Male	Bridget Jones' Diary	13
Male	Bridget Jones' Diary	16
Male	Bridget Jones' Diary	10
Male	Bridget Jones' Diary	18
Male	Bridget Jones' Diary	24
Male	Bridget Jones' Diary	13
Male	Bridget Jones' Diary	14
Male	Bridget Jones' Diary	19
Male	Bridget Jones' Diary	23
Male	Memento	37
Male	Memento	20
Male	Memento	16
Male	Memento	28
Male	Memento	27
Male	Memento	18
Male	Memento	32
Male	Memento	24
Male	Memento	21
Male	Memento	35
Female	Bridget Jones' Diary	3
Female	Bridget Jones' Diary	15
Female	Bridget Jones' Diary	5
Female	Bridget Jones' Diary	16
Female	Bridget Jones' Diary	13
Female	Bridget Jones' Diary	20
Female	Bridget Jones' Diary	11
Female	Bridget Jones' Diary	19
Female	Bridget Jones' Diary	15
Female	Bridget Jones' Diary	7
Female	Memento	30
Female	Memento	25

FIGURE 4.22 The **ChickFlick.dat** data

4.9.1.1 Bar charts for one independent variable ②

To begin with, let's just plot the mean arousal score (*y*-axis) for each film (*x*-axis). We can set this up by first creating the plot object and defining any aesthetics that apply to the plot as a whole. I have called the object *bar*, and have created it using the *ggplot()* function. The function specifies the dataframe to be used (*chickFlick*) and has set **film** to be plotted on the *x*-axis, and **arousal** to be plotted on the *y*-axis:

```
bar <- ggplot(chickFlick, aes(film, arousal))
```

This is where things get a little bit tricky; because we want to plot a summary of the data (the mean) rather than the raw scores themselves, we have to use a stat (section 4.4.5) to do this for us. Actually, we already used a stat when we plotted the boxplot in an earlier section, but we didn't notice because the boxplot geom sneaks off when we're not looking and uses the *bin* stat without us having to really do anything. However, if we want means then we have no choice but to dive head first into the pit of razors that is a *stat*. Specifically we are going to use *stat_summary()*.

The *stat_summary()* function takes the following general form:

```
stat_summary(function = x, geom = y)
```

Functions can be specified either for individual points (*fun.y*) or for the data as a whole (*fun.data*) and are set to be common statistical functions such as 'mean', 'median' and so on. As you might expect, the *geom* option is a way of telling the stat which geom to use to represent the function, and this can take on values such as 'errorbar', 'bar' and 'pointrange' (see Table 4.3). The *stat_summary()* function takes advantage of several built-in functions

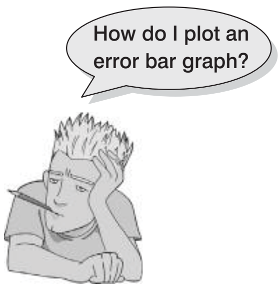
Table 4.4 Using *stat_summary()* to create graphs

Option	*Plots*	*Common geom*
fun.y = mean	The mean	geom = "bar"
fun.y = median	The median	geom = "bar"
fun.data = mean_cl_normal()	95% confidence intervals assuming normality	geom = "errorbar" geom = "pointrange"
fun.data = mean_cl_boot()	95% confidence intervals based on a bootstrap (i.e., not assuming normality)	geom = "errorbar" geom = "pointrange"
mean_sdl()	Sample mean and standard deviation	geom = "errorbar" geom = "pointrange"
fun.data = median_hilow()	Median and upper and lower quantiles	geom = "pointrange"

from the *Hmisc* package, which should automatically be installed. Table 4.4 summarizes these functions and how they are specified within the *stat_summary()* function.

If we want to add the mean, displayed as bars, we can simply add this as a layer to 'bar' using the *stat_summary()* function:

```
bar + stat_summary(fun.y = mean, geom = "bar", fill = "White", colour
= "Black"
```



As shown in Table 4.4, *fun.y = mean* computes the mean for us, *geom = "bar"* displays these values as bars, *fill = "White"* makes the bars white (the default is dark grey and you can replace with a different colour if you like), and *colour = "Black"* makes the outline of the bars black.

If we want to add error bars to create an **error bar chart**, we can again add these as a layer using *stat_summary()*:

```
+ stat_summary(fun.data = mean_cl_normal, geom = "pointrange")
```

This command adds a standard 95% confidence interval in the form of the pointrange geom. Again, if you like you could change the colour of the pointrange geom by setting its colour as described in Table 4.2.

Finally, let's add some nice labels to the graph using *lab()*:

```
+ labs(x = "Film", y = "Mean Arousal")
```

To sum up, if we put all of these commands together we can create the graph by executing the following command:

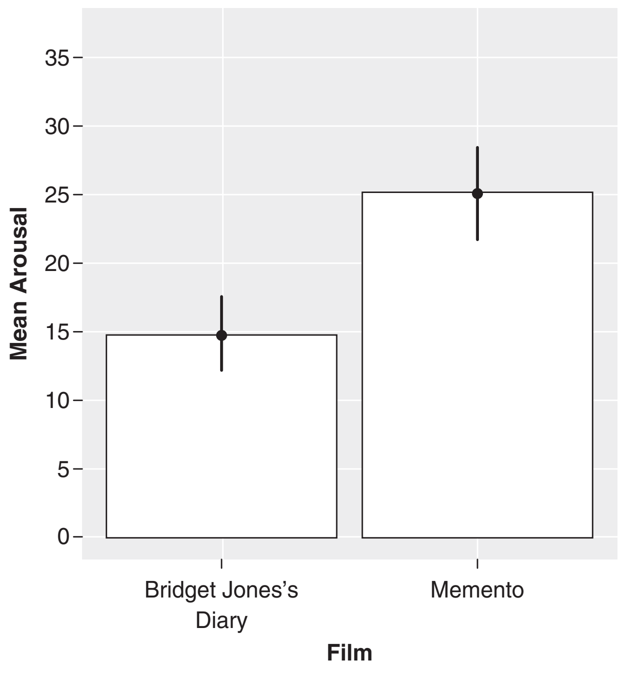
```
bar + stat_summary(fun.y = mean, geom = "bar", fill = "White", colour =
"Black") + stat_summary(fun.data = mean_cl_normal, geom = "pointrange") +
labs(x = "Film", y = "Mean Arousal")
```

Figure 4.23 shows the resulting bar chart. This graph displays the means (and the 95% confidence interval of those means) and shows us that on average, people were more aroused by *Memento* than they were by *Bridget Jones's Diary*. However, we originally wanted to look for gender effects, so we need to add this variable into the mix.

SELF-TEST

- ✓ Change the geom for the error bar to 'errorbar' and change its colour to red. Replot the graph.
- ✓ Plot the graph again but with bootstrapped confidence intervals.

FIGURE 4.23
Bar chart of the mean arousal for each of the two films



4.9.1.2. Bar charts for several independent variables ②

If we want to factor in gender we could do this in several ways. First we could set an aesthetic (such as colour) to represent the different genders, but we could also use faceting to create separate plots for men and women. We could also do both. Let's first look at separating men and women on the same graph. This takes a bit of work, but if we build up the code bit by bit the process should become clear.

First, as always we set up our plot object (again I've called it *bar*). This command is the same as before, except that we have set the *fill* aesthetic to be the variable **gender**. This means that any geom specified subsequently will be filled with different colours for men and women.

```
bar <- ggplot(chickFlick, aes(film, arousal, fill = gender))
```

If we want to add the mean, displayed as bars, we can simply add this as a layer to *bar* using the *stat_summary()* function as we did before, but with one important difference: we have to specify *position = "dodge"* (see section 4.4.6) so that the male and female bars are forced to stand side-by-side, rather than behind each other.

```
bar + stat_summary(fun.y = mean, geom = "bar", position="dodge")
```

As before, *fun.y = mean* computes the mean for us, *geom = "bar"* displays these values as bars.

If we want to add error bars we can again add these as a layer using *stat_summary()*:

```
+ stat_summary(fun.data = mean_cl_normal, geom = "errorbar", position = posi-
tion_dodge(width=0.90), width = 0.2)
```

This command is a bit more complicated than before. Note we have changed the geom to *errorbar*; by default these bars will be as wide as the bars displaying the mean, which looks a bit nasty, so I have reduced their width with *width = 0.2*, which should make them 20% of the width of the bar (which looks nice in my opinion). The other part of the command is that we have again had to use the *dodge* position to make sure that the error bars stand side-by-side). In this case *position = position_dodge(width=0.90)* does the trick, but you might have to play around with the values of *width* to get what you want.

Finally, let's add some nice labels to the graph using *lab()*:

```
+ labs(x = "Film", y = "Mean Arousal", fill = "Gender")
```

Notice that as well as specifying titles for each axis, I have specified a title for *fill*. This will give a title to the legend on the graph (if we omit this option the legend will be given the variable name as a title, which might be OK for you if you are less anally retentive than I am).

To sum up, if we put all of these commands together we can create the graph by executing the following command:

```
bar + stat_summary(fun.y = mean, geom = "bar", position="dodge") + stat_
summary(fun.data = mean_cl_normal, geom = "errorbar", position = position_
dodge(width = 0.90), width = 0.2) + labs(x = "Film", y = "Mean Arousal", fill
= "Gender")
```

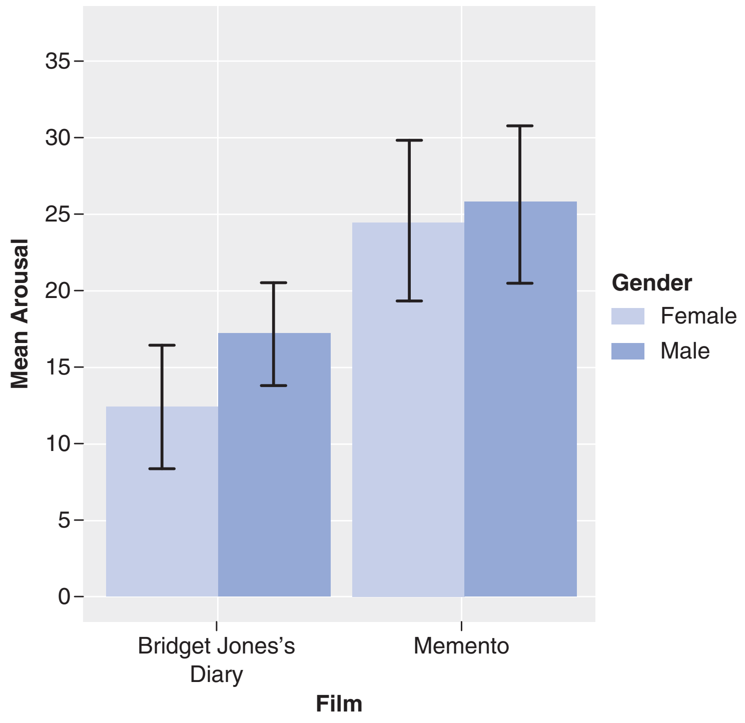


FIGURE 4.24 Bar chart of the mean arousal for each of the two films

Figure 4.24 shows the resulting bar chart. It looks pretty good, I think. It is possible to customise the colours that are used to fill the bars also (see R's Souls' Tip 4.3). Like the simple bar chart, this graph tells us that arousal was overall higher for *Memento* than for *Bridget Jones's Diary*, but it also splits this information by gender. The mean arousal for *Bridget Jones's Diary* shows that males were actually more aroused during this film than females. This indicates they enjoyed the film more than the women did. Contrast this with *Memento*, for which arousal levels are comparable in males and females. On the face of it, this contradicts the idea of a 'chick flick': it actually seems that men enjoy chick flicks more than the so-called 'chicks' do (probably because it's the only help we get to understand the complex workings of the female mind☺).

The second way to express gender would be to use this variable as a facet so that we display different plots for males and females:

```
bar <- ggplot(chickFlick, aes(film, arousal, fill = film))
```

Executing the above command sets up the graph in the same way as before. Note, however, that we do not need to use 'fill = gender' because we do not want to vary the colour by gender. (You can omit the fill command altogether, but I have set it so that the bars representing the different films are filled with different colours.) We set up the bar in the same way as before, except that we do not need to set the position to dodge because we are no longer plotting different bars for men and women on the same graph:

```
bar + stat_summary(fun.y = mean, geom = "bar")
```

We set up the error bar in the same way as before, except again we don't need to include a dodge:

```
+ stat_summary(fun.data = mean_cl_normal, geom = "errorbar", width = 0.2)
```

To get different plots for men and women we use the facet option and specify **gender** as the variable by which to facet:

```
+ facet_wrap( ~ gender)
```

We add labels as we did before:

```
+ labs(x = "Film", y = "Mean Arousal")
```

I've added an option to get rid of the graph legend as well (see R's Souls' Tip 4.2). I've included this option because we specified different colours for the different films so *ggplot* will create a legend; however, the labels on the *x*-axis will tell us to which film each bar relates so we don't need a colour legend as well):

```
+ opts(legend.position = "none")
```

The resulting graph is shown in Figure 4.25; compare this with Figure 4.24 and note how by using **gender** as a facet rather than an aesthetic results in different panels for men and women. The graphs show the same pattern of results though: men and women differ little in responses to *Memento*, but men showed more arousal to *Bridget Jones's Diary*.

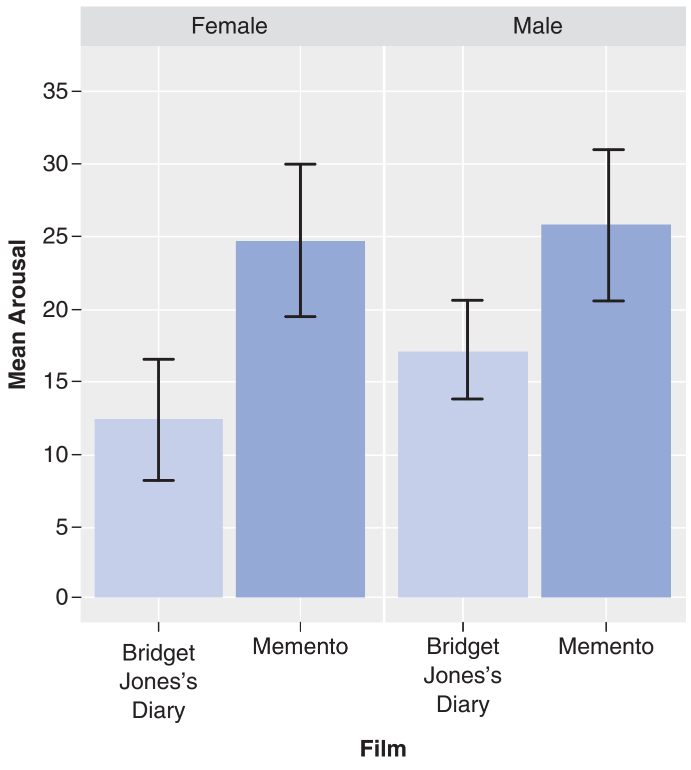


FIGURE 4.25 The mean arousal (and 95% confidence interval) for two different films displayed as different graphs for men and women using *facet_wrap()*

R's Souls' Tip 4.3 Custom colours ②

If you want to override the default fill colours, you can do this using the *scale_fill_manual()* function. For our chick flick data, for example, if we wanted blue bars for females and green for males then we can add the following command:

```
+ scale_fill_manual("Gender", c("Female" = "Blue", "Male" = "Green"))
```

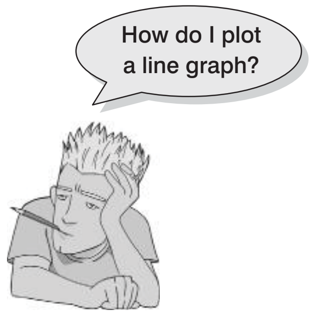
Alternatively, you can use very specific colours by specifying colours using the RRGGBB system. For example, the following produces very specifically coloured blue and green bars:

```
+ scale_fill_manual("Gender", c("Female" = "#3366FF", "Male" = "#336633"))
```

Try adding these commands to the end of the command we used to generate Figure 4.24 and see the effect it has on the bar colours. Then experiment with other colours.

4.9.2. Line graphs ②

4.9.2.1. Line graphs of a single independent variable ②

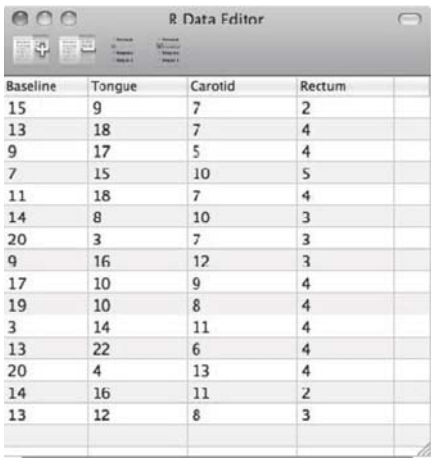


Hiccups can be a serious problem: Charles Osborne apparently got a case of hiccups while slaughtering a hog (well, who wouldn't?) that lasted 67 years. People have many methods for stopping hiccups (a surprise, holding your breath), but actually medical science has put its collective mind to the task too. The official treatment methods include tongue-pulling manoeuvres, massage of the carotid artery, and, believe it or not, digital rectal massage (Fesmire, 1988). I don't know the details of what the digital rectal massage involved, but I can probably imagine. Let's say we wanted to put digital rectal massage to the test (as a cure for hiccups, I mean). We took 15 hiccup sufferers, and during a bout of hiccups administered each of the three procedures (in random order and at intervals of 5 minutes) after taking a baseline of how many hiccups they had per minute. We counted the number of hiccups in the minute after each procedure. Load the file **Hiccups.dat** from the companion website into a dataframe called *hiccupsData* by executing (again assuming you have set your working directory to be where the file is located):

```
hiccupsData <- read.delim("Hiccups.dat",  header = TRUE)
```

Figure 4.26 shows the data. Note there are four variables:

- **Baseline**: specifies the number of hiccups at baseline.
- **Tongue**: specifies the number of hiccups after tongue pulling.
- **Carotid**: specifies the number of hiccups after carotid artery massage.
- **Rectum**: specifies the number of hiccups after digital rectal massage.



Baseline	Tongue	Carotid	Rectum
15	9	7	2
13	18	7	4
9	17	5	4
7	15	10	5
11	18	7	4
14	8	10	3
20	3	7	3
9	16	12	3
17	10	9	4
19	10	8	4
3	14	11	4
13	22	6	4
20	4	13	4
14	16	11	2
13	12	8	3

FIGURE 4.26
The Hiccups.dat data

Each row in the data file represents a different person, so these data are laid out as a repeated-measures design, with each column representing a different treatment condition and every person undergoing each treatment.

These data are in the wrong format for *ggplot2* to use. We need all of the scores stacked up in a single column and then another variable that specifies the type of intervention.

SELF-TEST

✓ Thinking back to Chapter 3, use the *stack()* function to restructure the data into long format.

We can rearrange the data as follows (see section 3.9.4):

```
hiccups<-stack(hiccupsData)
names(hiccups)<-c("Hiccups","Intervention")
```

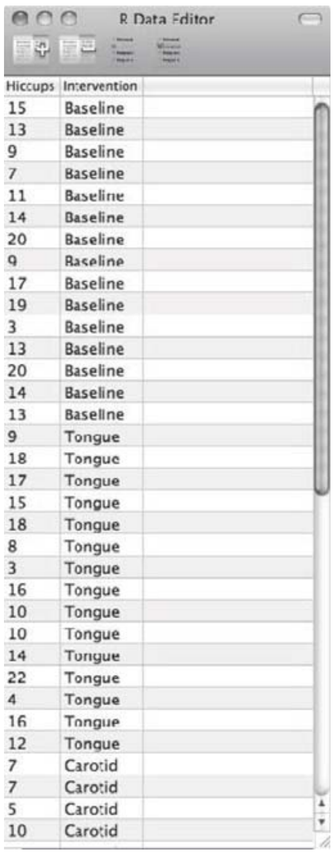
Executing these commands creates a new dataframe called *hiccups*, which has the number of hiccups in one column alongside a new variable containing the original variable name associated with each score (i.e., the column headings) in the other column (Figure 4.27). The *names()* function just assigns names to these new variables in the order that they appear in the dataframe. To plot a categorical variable in *ggplot()* it needs to be recognized as a factor, so we also need to create new variable in the *hiccups* dataframe called **Intervention_Factor**, which is just the **Intervention** variable converted into a factor:

```
hiccups$Intervention_Factor <- factor(hiccups$Intervention, levels =
hiccups$Intervention)
```

We are now ready to plot the graph. As always we first create the plot object and define the variables that we want to plot as aesthetics:

```
line <- ggplot(hiccups, aes(Intervention_Factor, Hiccups))
```

I have called the object *line*, and have created it using the *ggplot()* function. The function specifies the dataframe to be used (*hiccups*) and has set **Intervention_Factor** to be plotted on the *x*-axis, and **Hiccups** to be plotted on the *y*-axis.



Hiccups	Intervention
15	Baseline
13	Baseline
9	Baseline
7	Baseline
11	Baseline
14	Baseline
20	Baseline
9	Baseline
17	Baseline
19	Baseline
3	Baseline
13	Baseline
20	Baseline
14	Baseline
13	Baseline
9	Tongue
18	Tongue
17	Tongue
15	Tongue
18	Tongue
8	Tongue
3	Tongue
16	Tongue
10	Tongue
10	Tongue
14	Tongue
22	Tongue
4	Tongue
16	Tongue
12	Tongue
7	Carotid
7	Carotid
5	Carotid
10	Carotid

FIGURE 4.27 The hiccups data in long format

Just as we did for our bar charts, we are going to use *stat_summary()* to create the mean values within each treatment condition. Therefore, as with the bar chart, we create a layer using *stat_summary()* and add this to the plot:

```
line + stat_summary(fun.y = mean, geom = "point")
```

Note that this command is exactly the same as for a bar chart, except that we have chosen the point geom rather than a bar. At the moment we have a plot with a symbol representing each group mean. If we want to connect these symbols with a line then we use *stat_summary()* again, we again specify *fun.y* to be the mean, but this time choose the *line* geom. To make the line display we also need to set an aesthetic of *group = 1*; this is because we are joining summary points (i.e., points that summarize a group) rather than individual data points. Therefore, we specify the line as:

```
+ stat_summary(fun.y = mean, geom = "line", aes(group = 1))
```

The above command will add a solid black line connecting the group means. Let's imagine we want this line to be blue, rather than black, and dashed rather than solid, we can simply add these aesthetics into the above command as follows:

```
+ stat_summary(fun.y = mean, geom = "line", aes(group = 1), colour = "Blue",
linetype = “dashed”)
```

Now let's add an error bar to each group mean. We can do this by adding another layer using *stat_summary()*. When we plotted an error bar on the bar chart we used a normal error bar, so this time let's add an error bar based on bootstrapping. We set the function for the data to be *mean_cl_boot* (*fun.data = mean_cl_boot*) – see Table 4.4 – and set the geom to be *errorbar* (you could use *pointrange* as we did for the bar chart if you prefer):

```
+ stat_summary(fun.data = mean_cl_boot, geom = "errorbar")
```

The default error bars are quite wide, so I recommend setting the width parameter to 0.2 to make them look nicer:

```
+ stat_summary(fun.data = mean_cl_boot, geom = "errorbar", width = 0.2)
```

You can, of course, also change the colour and other properties of the error bar in the usual way (e.g., by adding *colour* = *"Red"* to make them red). Finally, we will add some labels to the *x*- and *y*-axes using the *labs()* function:

```
+ labs(x = "Intervention", y = "Mean Number of Hiccups")
```

If we put all of these commands together, we can create the graph by executing the following command:

```
line <- ggplot(hiccups, aes(Intervention_Factor, Hiccups))

line + stat_summary(fun.y = mean, geom = "point") + stat_summary(fun.y =
mean, geom = "line", aes(group = 1),colour = "Blue", linetype = "dashed")
+ stat_summary(fun.data = mean_cl_boot, geom = "errorbar", width = 0.2) +
labs(x = "Intervention", y = "Mean Number of Hiccups")
```

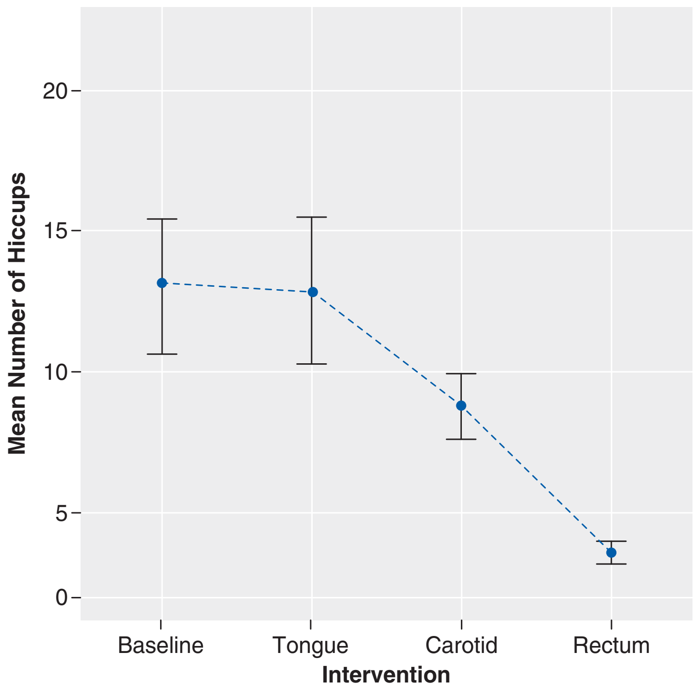


FIGURE 4.28 Line chart with error bars of the mean number of hiccups at baseline and after various interventions

The resulting graph in Figure 4.28 displays the mean number of hiccups at baseline and after the three interventions (and the confidence intervals of those means based on bootstrapping). As we will see in Chapter 9, the error bars on graphs of repeated-measures designs aren't corrected for the fact that the data points are dependent; I don't want to get into the reasons why here because I want to keep things simple, but if you're doing a graph of your own data then I would read section 9.2 before you do.

We can conclude that the amount of hiccups after tongue pulling was about the same as at baseline; however, carotid artery massage reduced hiccups, but not by as much as a good old fashioned digital rectal massage. The moral here is: if you have hiccups, find something digital and go amuse yourself for a few minutes.

4.9.2 2. Line graphs for several independent variables ②

We all like to text-message (especially students in my lectures who feel the need to text-message the person next to them to say 'Bloody hell, this guy is so boring I need to poke out my own eyes'). What will happen to the children, though? Not only will they develop super-sized thumbs, they might not learn correct written English. Imagine we conducted an experiment in which a group of 25 children was encouraged to send text messages on their mobile phones over a six-month period. A second group of 25 children was forbidden from sending text messages for the same period. To ensure that kids in this latter group didn't use their phones, this group was given armbands that administered painful shocks in the presence of radio waves (like those emitted from phones).[11] The outcome was a score on a grammatical test (as a percentage) that was measured both before and after the intervention. The first independent variable was, therefore, text message use (text messagers versus controls) and the second independent variable was the time at which grammatical ability was assessed (baseline or after 6 months). The data are in the file **Text Messages.dat**.

Load this file into a dataframe called *textData* by executing this command (I'm assuming you have set the working directory to be where the data file is stored):

```
textData <- read.delim("TextMessages.dat", header = TRUE)
```

Figure 4.29 shows the data. Note there are three variables:

- **Group**: specifies whether they were in the text message group or the control group.
- **Baseline**: grammar scores at baseline.
- **Six_months**: grammar scores after 6 months.

Each row in the data file represents a different person. These data are again in the wrong format for *ggplot2*. Instead of the current wide format, we need the data in long (i.e., molten) format (see section 3.9.4). This format will have the following variables:

- **Group**: specifies whether they were in the text message group or the control group.
- **Time**: specifies whether the score relates to baseline or 6 months.
- **Grammar_Score**: the grammar scores.

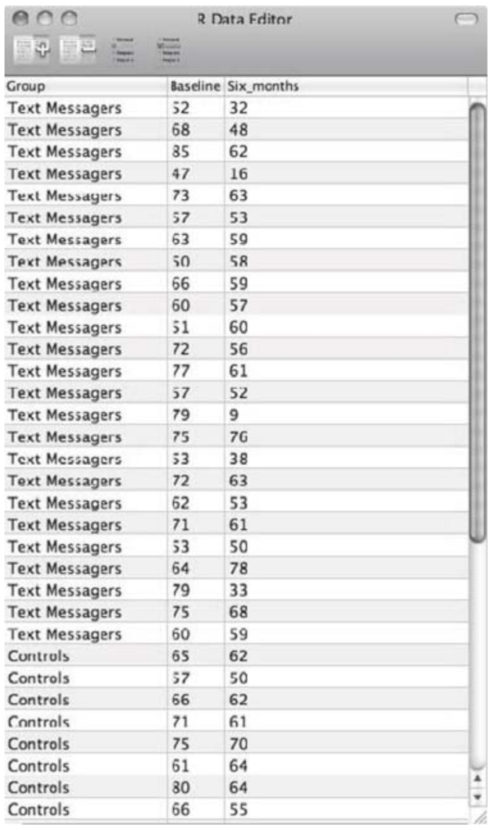
SELF-TEST

✓ Restructure the data to a new dataframe called *textMessages* that is in long format. Use the *factor()* function (see section 3.5.4.3) to convert the 'Time' variable to a factor with levels called 'Baseline' and '6 Months'.

Assuming that you have done the self-test, you should now have a dataframe called *textMessages* that is formatted correctly for *ggplot2*. As ever, we set up our plot object (I've called it *line*). This command is the same as before, except that we have set the 'fill' aesthetic to be the variable **Group**. This means that any geom specified subsequently will

[11] Although this punished them for any attempts to use a mobile phone, because other people's phones also emit microwaves, an unfortunate side effect was that these children acquired a pathological fear of anyone talking on a mobile phone.

FIGURE 4.29
The text message data before being reshaped



Group	Baseline	Six_months
Text Messagers	52	32
Text Messagers	68	48
Text Messagers	85	62
Text Messagers	47	16
Text Messagers	73	63
Text Messagers	57	53
Text Messagers	63	59
Text Messagers	50	58
Text Messagers	66	59
Text Messagers	60	57
Text Messagers	51	60
Text Messagers	72	56
Text Messagers	77	61
Text Messagers	57	52
Text Messagers	79	9
Text Messagers	75	76
Text Messagers	53	38
Text Messagers	72	63
Text Messagers	62	53
Text Messagers	71	61
Text Messagers	53	50
Text Messagers	64	78
Text Messagers	79	33
Text Messagers	75	68
Text Messagers	60	59
Controls	65	62
Controls	57	50
Controls	66	62
Controls	71	61
Controls	75	70
Controls	61	64
Controls	80	64
Controls	66	55

be filled with different colours for text messagers and the control group. Note that we have specified the data to be the *textMessages* dataframe, and for **Time** to be plotted on the *x*-axis and **Grammar_Score** on the *y*-axis.

```
line <- ggplot(textMessages, aes(Time, Grammar_Score, colour = Group))
```

If we want to add the means, displayed as symbols, we can add this as a layer to *line* using the *stat_summary()* function just as we did in the previous section:

```
line + stat_summary(fun.y = mean, geom = "point")
```

To add lines connecting the means we can add these as a layer using *stat_summary()* in exactly the same way as we did in the previous section. The main difference is that because in this example we have more than one group, rather than setting *aes(group = 1)* as we did before, we now set this aesthetic to be the variable (**Group**) that differentiates the different sets of means (*aes(group = Group)*):

```
+ stat_summary(fun.y = mean, geom = "line", aes(group = Group))
```

We can also add a layer containing error bars and a layer containing labels using the same commands as the previous example:

```
+ stat_summary(fun.data = mean_cl_boot, geom = "errorbar", width = 0.2) +
labs(x = "Time", y = "Mean Grammar Score", colour = "Group")
```

If we put all of these commands together we can create the graph by executing the following command:

```
line + stat_summary(fun.y = mean, geom = "point") + stat_summary(fun.y =
mean, geom = "line", aes(group = Group)) + stat_summary(fun.data = mean_cl_
boot, geom = "errorbar", width = 0.2) + labs(x = "Time", y = "Mean Grammar
Score", colour = "Group")
```

SELF-TEST

✓ Use what you have learnt to repeat the text message data plot but to also have different symbols for text messagers and controls and different types of lines.

Figure 4.30 shows the resulting chart. It shows that at baseline (before the intervention) the grammar scores were comparable in our two groups; however, after the intervention, the grammar scores were lower in the text messagers than in the controls. Also, if you look at the dark blue line you can see that text messagers' grammar scores have fallen over the 6 months; compare this to the controls (the red line on your screen, or black in the figure) whose grammar scores are fairly similar over time. We could, therefore, conclude that text messaging has a detrimental effect on children's understanding of English grammar and civilization will crumble, with Abaddon rising cackling from his bottomless pit to claim our wretched souls. Maybe.

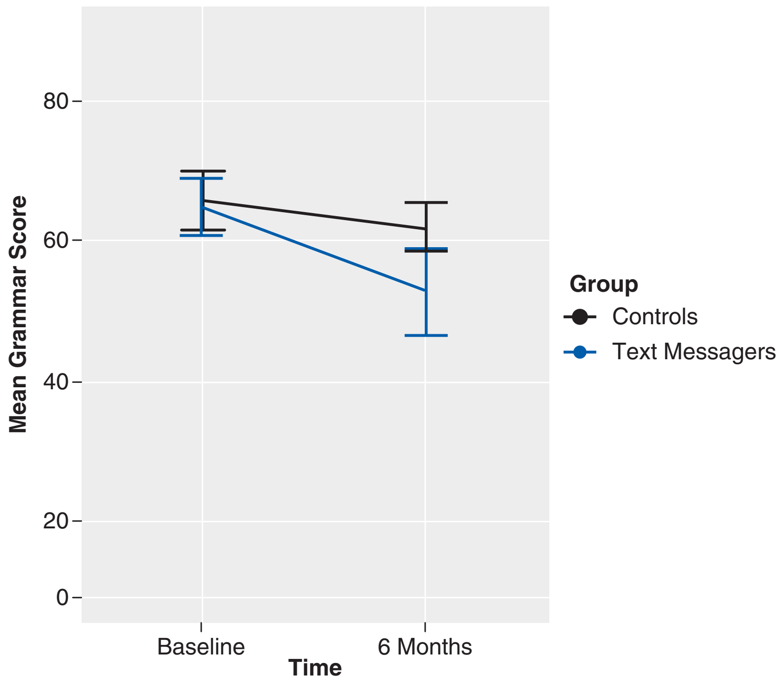


FIGURE 4.30 Error bar graph of the mean grammar score over 6 months in children who were allowed to text-message versus those who were forbidden

4.10. Themes and options ①

I mentioned earlier that *ggplot2* produces Tufte-friendly graphs. In fact, it has two built-in themes. The default is called *theme_grey()*, which follows Tufte's advice in that it uses grid lines to ease interpretation but makes them have low visual impact so that they do not distract the eye from the data. The second theme is a more traditional black and white theme called *theme_bw()*. The two themes are shown in Figure 4.31.

As well as these global themes, the *opts()* function allows you to control the look of specific parts of the plot. For example, you can define a title, set the properties of that title

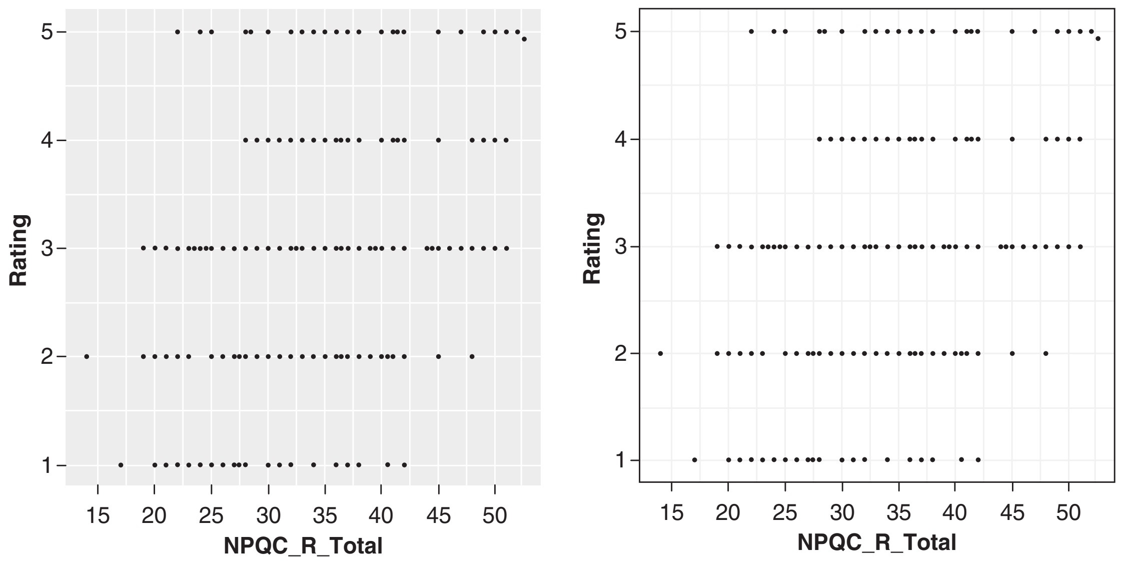


FIGURE 4.31 Default graph styles in *ggplot2*: the left panel shows *theme_grey()*, the right panel shows *theme_bw()*

(size, font, colour, etc.). You can also change the look of axes, grid lines, background panels and text. You apply theme and formatting instructions by adding a layer to the plot:

```
myGraph + geom_point() + opts()
```

Table 4.5 shows these themes, their aesthetic properties and the elements of the plot associated with them. The table makes clear that there are four types of theme that

Table 4.5 Summary of theme elements and their properties

Theme	***Properties***	***Elements***	***Element Description***
theme_text()	family face colour size hjust vjust angle lineheight	axis.text.x axis.text.y axis.title.x axis.title.y legend.text legend.title plot.title strip.text.x strip.text.y	*x*-axis label *y*-axis label Horizontal tick labels Vertical tick labels Legend labels Legend name Plot title Horizontal facet label text Vertical facet label text
theme_line()	colour size linetype	panel.grid.major panel.grid.minor	Major grid lines Minor grid lines
theme_segment()	colour size linetype	axis.line axis.ticks	Line along an axis Axis tick marks
theme_rect()	colour size linetype fill	legend.background legend.key panel.background panel.background plot.background strip.background	Background of legend Background under legend key Background of panel Border of panel Background of the entire plot Background of facet labels

determine the appearance of text (*theme_text*), lines (*theme_line*), axes (*theme_segment*) and rectangles (*theme_rect*). Each of these themes has properties that can be adjusted; so for all of them you can adjust size and colour, for text you can also adjust things like the font family and angle, for rectangles you can change the fill colour and so on. Different elements of a plot can be changed by adjusting the particular theme attached to that element. So, for example, if we wanted to change the colour of the major grid lines to blue, we would have to do this by setting the colour aesthetic of the *panel.grid.major* element using *theme_line()*. Aesthetic properties are set in the same way as described in section 4.4.3. Therefore, we would do this as follows:

```
+ opts(panel.grid.major = theme_line(colour = "Blue"))
```

Similarly, we could make the axes have blue lines with:

```
+ opts(axis.line = theme_segment(colour = "Blue"))
```

or dashed lines by using:

```
+ opts(axis.line = theme_segment(linetype = 2))
```

The possibilities are endless, and I can't explain them all without killing several more rainforests, but I hope that you get the general idea.

What have I discovered about statistics? ①

This chapter has looked at how to inspect your data using graphs. We've covered a lot of different graphs. We began by covering some general advice on how to draw graphs and we can sum that up as minimal is best: no pink, no 3-D effects, no pictures of Errol your pet ferret superimposed on the graph – oh, and did I mention no pink? We have looked at graphs that tell you about the distribution of your data (histograms, boxplots and density plots), that show summary statistics about your data (bar charts, error bar charts, line charts, drop-line charts) and that show relationships between variables (scatterplots). Throughout the chapter we looked at how we can edit graphs to make them look minimal (and of course to colour them pink, but we know better than to do that, don't we?).

We also discovered that I liked to explore as a child. I was constantly dragging my dad (or was it the other way around?) over piles of rocks along any beach we happened to be on. However, at this time I also started to explore great literature, although unlike my cleverer older brother who was reading Albert Einstein's papers (well, Isaac Asimov) as an embryo, my literary preferences were more in keeping with my intellect, as we will see.

R packages used in this chapter

ggplot2

R functions used in this chapter

file.path()	ggplot()
geom_boxplot()	ggsave()
geom_density()	labs()
geom_histogram()	opts()
geom_line()	qplot()
geom_point()	stat_summary()
geom_smooth()	Sys.getenv()

Key terms that I've discovered

Bar chart	Line chart
Boxplot (box–whisker plot)	Outlier
Chartjunk	Regression line
Density plot	Scatterplot
Error bar chart	

Smart Alex's tasks

- **Task 1**: Using the data from Chapter 3 (which you should have saved, but if you didn't, re-enter it from Table 3.6), plot and interpret the following graphs: ①
  - An error bar chart showing the mean number of friends for students and lecturers.
  - An error bar chart showing the mean alcohol consumption for students and lecturers.
  - An error line chart showing the mean income for students and lecturers.
  - An error line chart showing the mean neuroticism for students and lecturers.
  - A scatterplot with regression lines of alcohol consumption and neuroticism grouped by lecturer/student.

- **Task 2**: Using the **Infidelity** data from Chapter 3 (see Smart Alex's Task 3), plot a clustered error bar chart of the mean number of bullets used against self and partner for males and females. ①

Answers can be found on the companion website.

Further reading

Tufte, E. R. (2001). *The visual display of quantitative information* (2nd ed.). Cheshire, CT: Graphics Press.
Wainer, H. (1984). How to display data badly. *American Statistician*, *38*(2), 137–147.
Wickham, H. (2009). *ggplot2: Elegant graphics for data analysis*. New York: Springer.
Wilkinson, L. (2005). *The grammar of graphics*. New York: Springer-Verlag.

Wright, D. B., & Williams, S. (2003). Producing bad results sections. *The Psychologist*, *16*, 646–648. (This is a very accessible article on how to present data. Dan usually has this article on his website so Google Dan Wright to find where his web pages are located.)

Web resources:

http://junkcharts.typepad.com/ is an amusing look at bad graphs.
http://had.co.nz/ggplot2/ is the official *ggplot2* website (and very useful it is, too).

Interesting real research

Fesmire, F. M. (1988). Termination of intractable hiccups with digital rectal massage. *Annals of Emergency Medicine*, *17*(8), 872.