3.5.4.3. Creating coding variables/factors ①

A coding variable (also known as a grouping variable or factor) is a variable that uses numbers to represent different groups of data. As such, it is a *numeric variable*, but these numbers represent names (i.e., it is a nominal variable). These groups of data could be levels of a treatment variable in an experiment, different groups of people (men or women, an experimental group or a control group, ethnic groups, etc.), different geographic locations, different organizations, etc.

In experiments, coding variables represent independent variables that have been measured between groups (i.e., different participants were assigned to different groups). If you were to run an experiment with one group of participants in an experimental condition and a different group of participants in a control group, you might assign the experimental group a code of 1 and the control group a code of 0. When you come to put the data into R you would create a variable (which you might call **group**) and type in the value 1 for any participants in the experimental group, and 0 for any participant in the control group. These codes tell R that all of the cases that have been assigned the value 1 should be treated as belonging to the same group, and likewise for the cases assigned the value 0. In situations other than experiments, you might simply use codes to distinguish naturally occurring groups of people (e.g., you might give students a code of 1 and lecturers a code of 0). These codes are completely arbitrary; for the sake of convention people typically use 0, 1, 2, 3, etc., but in practice you could have a code of 495 if you were feeling particularly arbitrary.

We have a coding variable in our data: the one describing whether a person was a lecturer or student. To create this coding variable, we follow the steps for creating a normal variable, but we also have to tell R that the variable is a coding variable/factor and which numeric codes have been assigned to which groups.

First, we can enter the data and then worry about turning these data into a coding variable. In our data we have five lecturers (who we will code with 1) and five students (who we will code with 2). As such, we need to enter a series of 1s and 2s into our new variable, which we'll call **job**. The way the data are laid out in Table 3.6 we have the five lecturers followed by the five students, so we can enter the data as:

```
job<-c(1,1,1,1,1,2,2,2,2,2)
```

In situations like this, in which all cases in the same group are grouped together in the data file, we could do the same thing more quickly using the **rep()** function. This function takes the general form of *rep(number to repeat, how many repetitions)*. As such, *rep(1, 5)* will repeat the number 1 five times. Therefore, we could generate our **job** variable as follows:

```
job<-c(rep(1, 5),rep(2, 5))
```

Whichever method you use the end results is the same:

```
job

[1] 1 1 1 1 1 2 2 2 2 2
```

To turn this variable into a factor, we use the **factor()** function. This function takes the general form:

```
factor(variable, levels = c(x,y, … z), labels = c("label1", "label2", …
"label3"))
```

This looks a bit scary, but it's not too bad really. Let's break it down: *factor(variableName)* is all you really need to create the factor – in our case *factor(job)* would do the trick. However, we need to tell R which values we have used to denote different groups and we do this with *levels = c(1,2,3,4, …)*; as usual we use the *c()* function to list the values we have used. If we have used a regular series such as 1, 2, 3, 4 we can abbreviate this as *c(1:4)*, where the colon simply means 'all the values between'; so, *c(1:4)* is the same as *c(1,2,3,4)* and *c(0:6)* is the same as *c(0,1,2,3,4,5,6)*. In our case, we used 1 and 2 to denote the two groups, so we could specify this as *c(1:2)* or *c(1,2)*. The final step is to assign labels to these levels using *labels = c("label", ...)*. Again, we use *c()* to list the labels that we wish to assign. You must list these labels in the same order as your numeric levels, and you need to make sure you have provided a label for each level. In our case, 1 corresponds to lecturers and 2 to students, so we would want to specify labels of "Lecturer" and "Student". As such, we could write *levels = c("Lecturers", "Students")*. If we put all of this together we get this command, which we can execute to transform **job** into a coding variable:

```
job<-factor(job, levels = c(1:2), labels = c("Lecturer", "Student"))
```

Having converted **job** to a factor, **R** will treat it as a nominal variable. A final way to generate factors is to use the *gl()* function – the 'gl' stands for general (factor) levels. This function takes the general form:

```
newFactor<-gl(number of levels, cases in each level, total cases, labels =
c("label1", "label2"...))
```

which creates a factor variable called *newFactor*; you specify the number of levels or groups of the factor, how many cases are in each level/group, optionally the total number of cases (the default is to multiply the number of groups by the number of cases per group), and you can also use the *labels* option to list names for each level/group. We could generate the variable job as follows:

```
job<-gl(2, 5, labels = c("Lecturer", "Student"))
```

The end result is a fully-fledged coding variable (or factor):

```
[1] Lecturer Lecturer Lecturer Lecturer Lecturer Student Student Student
Student Student
```

With any factor variable you can see the factor levels and their order by using the **levels()** function, in which you enter the name of the factor. So, to see the levels of our variable job we could execute:

```
levels(job)
```

which will produce this output:

```
[1] "Lecturer" "Student"
```

In other words, we know that the variable job has two levels and they are (in this order) *Lecturer* and *Student*. We can also use this function to set the levels of a variable. For example, imagine we wanted these levels to be called *Medical Lecturer* and *Medical Student*, we could execute:

```
levels(job)<-c("Medical Lecturer", "Medical Student")
```

This command will rename the levels associated with the variable job (note, the new names are entered as text with speech marks, and are wrapped up in the *c()* function). You can also use this function to reorder the levels of a factor – see R's Souls' Tip 3.13.

This example should clarify why in experimental research grouping variables are used for variables that have been measured between participants: because by using a coding variable it is impossible for a participant to belong to more than one group. This situation should occur in a between-group design (i.e., a participant should not be tested in both the experimental and the control group). However, in repeated-measures designs (within subjects) each participant is tested in every condition and so we would not use this sort of coding variable (because each participant does take part in every experimental condition)

3.8. Saving data ①

Having spent hours typing in data, you might want to save it. As with importing data, you can export data from R in a variety of formats. Again, for the sake of flexibility we recommend exporting to tab-delimited text or CSV (see R's Souls' Tip 3.11) because these formats can be imported easily into a variety of different software packages (Excel, SPSS, SAS, STATA, etc.). To save data as a tab-delimited file, we use the *write.table()* command and for a CSV we can use *write.csv()*.

The *write.table()* command takes the general form:

```
write.table(dataframe, "Filename.txt", sep="\t", row.names = FALSE)
```

We replace *dataframe* with the name of the dataframe that we would like to save and "Filename.txt" with the name of the file.[7] The command *sep=""* sets the character to be used to separate data values: whatever you place between the "" will be used to separate data values. As such, if we want to create a CSV file we could write *sep* = "," (which tells R to separate values with a comma), but to create a tab-delimited text file we would write *sep* = "\t" (where we have written \t between quotes, which represents the tab key), and we could also create a space-delimited text file by using *sep* = " " (note that there is a space between the quotes). Finally, *row.names* = *FALSE* just prevents **R** from exporting a column of row numbers (the reason for preventing this is because **R** does not name this column so it throws the variable names out of sync). Earlier on we created a dataframe called *metallica*. To export this dataframe to a tab-delimited text file called **Metallica Data.txt**, we would execute this command:

```
write.table(metallica, "Metallica Data.txt", sep="\t", row.names = FALSE)
```

The *write.csv()* command takes the general form:

```
write.csv(dataframe, "Filename.csv")
```

As you can see, it is much the same as the *write.table()* function. In fact, it is the *write.table()* function but with *sep* = "," as the default.[8] So, to save the metallica dataframe as a CSV file we can execute:

```
write.csv(metallica, "Metallica Data.csv")
```

3.9. Manipulating data ③

3.9.1. Selecting parts of a dataframe ②

Sometimes (especially with large dataframes) you might want to select only a small portion of your data. This could mean choosing particular variables, or selecting particular cases. One way to achieve this goal is to create a new dataframe that contains only the variables or cases that you want. To select cases, we can execute the general command:

```
newDataframe <- oldDataframe[rows, columns]
```

[7] Remember that if you have not set a working directory during your session then this filename will need to include the full location information. For example, "C:/Users/Andy F/Documents/Data/R Book Examples/Filename.txt" or "~/Documents/Data/R Book Examples/Filename.txt" in MacOS. Hopefully, it is becoming ever clearer why setting the working directory is a good thing to do.

[8] If you live in certain parts of western Europe, you might want to use *write.csv2()* instead which outputs the file in the format conventional for that part of the world: it uses ';' to separate values, and ',' instead of '.' to represent the decimal point.

This command creates a new dataframe (called *newDataframe*) that contains the specified *rows* and *columns* from the old dataframe (called *oldDataframe*). Let's return to our lecturer data (in the dataframe that we created earlier called *lecturerData*); imagine that we wanted to look only at the variables that reflect some aspect of their personality (for example, alcohol intake, number of friends, and neuroticism). We can create a new dataframe (*lecturerPersonality*) that contains only these three variables by executing this command:

```
lecturerPersonality <- lecturerData[, c("friends", "alcohol", "neurotic")]
```

Note first that we have not specified *rows* (there is nothing before the comma); this means that all rows will be selected. Note also that we have specified *columns* as a list of variables with each variable placed in quotes (be careful to spell them exactly as they are in the original dataframe); because we want several variables, we put them in a list using the *c()* function. If you look at the contents of the new dataframe you'll see that it now contains only the three variables that we specified:

```
> lecturerPersonality

     friends alcohol   neurotic
1        5      10        10
2        2      15        17
3        0      20        14
4        4       5        13
5        1      30        21
6       10      25         7
7       12      20        13
8       15      16         9
9       12      17        14
10      17      18        13
```

Similarly, we can select specific cases of data by specifying an instruction for *rows* in the general function. This is done using a logical argument based on one of the operators listed in Table 3.5. For example, let's imagine that we wanted to keep all of the variables, but look only at the lecturers' data. We could do this by creating a new dataframe (*lecturer Only*) by executing this command:

```
lecturerOnly <- lecturerData[job=="Lecturer",]
```

Note that we have not specified *columns* (there is nothing after the comma); this means that all variables will be selected. However, we have specified *rows* using the condition *job == "Lecturer"*. Remember that the '==' means 'equal to', so we have basically asked **R** to select any rows for which the variable **job** is exactly equal to the word 'Lecturer' (spelt exactly as we have). The new dataframe contains only the lecturers' data:

```
> lecturerOnly
    Name         DoB      job  friends alcohol income   neurotic
1 Ben    1977-07-03 Lecturer       5      10  20000        10
2 Martin 1969-05-24 Lecturer       2      15  40000        17
3 Andy   1973-06-21 Lecturer       0      20  35000        14
4 Paul   1970-07-16 Lecturer       4       5  22000        13
5 Graham 1949-10-10 Lecturer       1      30  50000        21
```

We can be really cunning and specify both rows and columns. Imagine that we wanted to select the personality variables but only for people who drink more than 10 units of alcohol. We could do this by executing:

```
alcoholPersonality <- lecturerData[alcohol > 10, c("friends", "alcohol",
"neurotic")]
```

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.

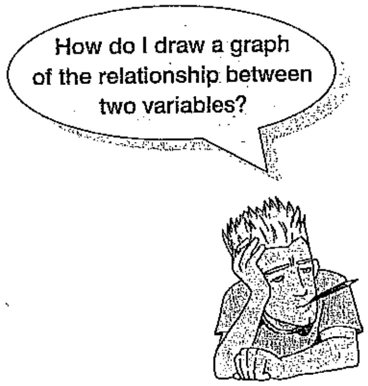
Finally, if we wanted to differentiate rating types by their shape rather than using a colour, we could change the colour aesthetic to be the shape aesthetic:

```
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

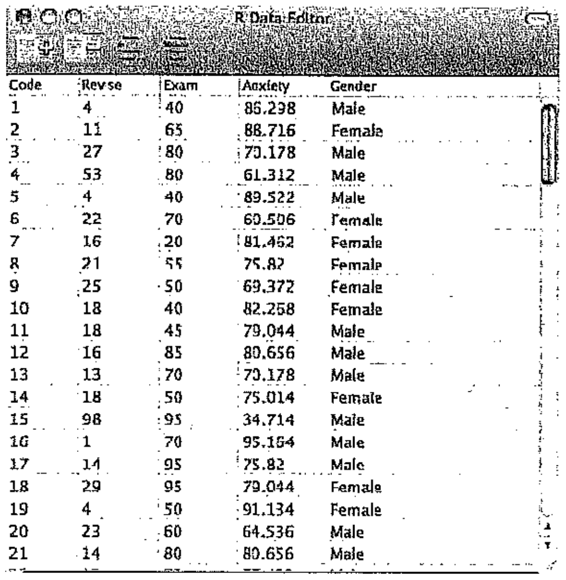
This type of scatterplot is for looking at just two variables. For example, a psychologist was interested in the effects of exam stress on exam performance. So, she devised and validated a questionnaire to assess state anxiety relating to exams (called the Exam Anxiety Questionnaire, or EAQ). This scale 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:

137

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	Revise	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	82.258	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 %")
```

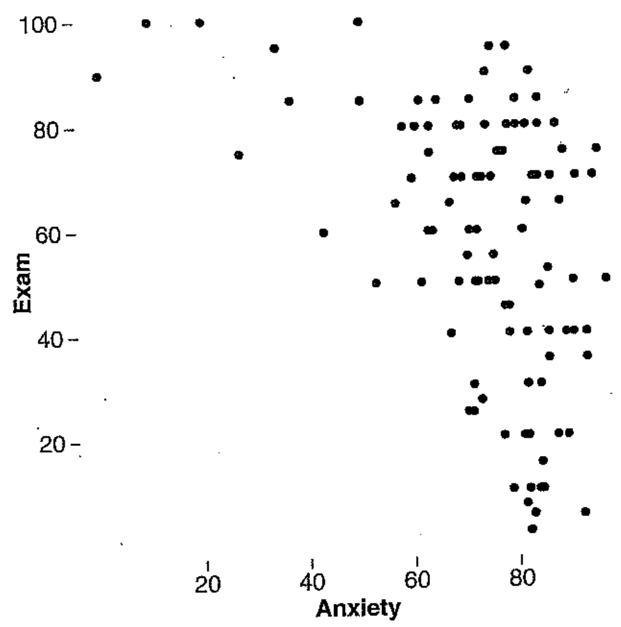
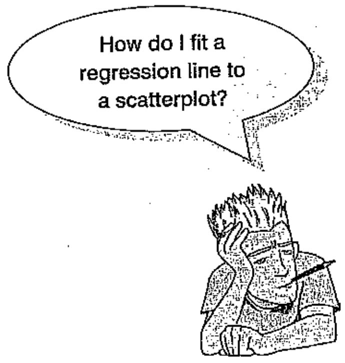


FIGURE 4.13 Scatterplot of exam anxiety and exam performance

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'

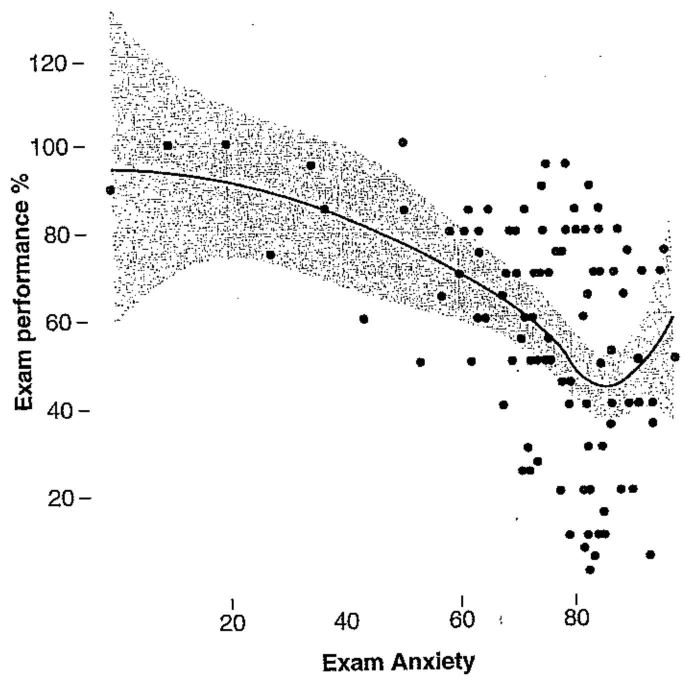


FIGURE 4.14 Scatterplot of exam anxiety against exam performance with a smoother added

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)
```

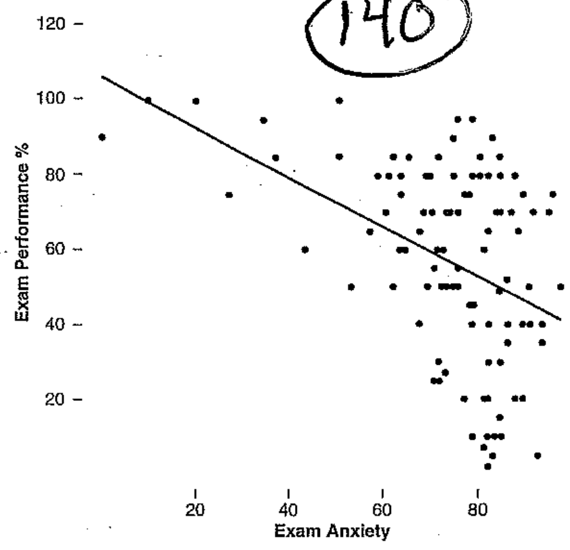
We can also change the colour and transparency of the confidence interval using the *fill* and *alpha* aesthetics, respectively. For example, if we want the confidence interval to be blue like the line itself, and we want it fairly transparent we could specify:

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

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

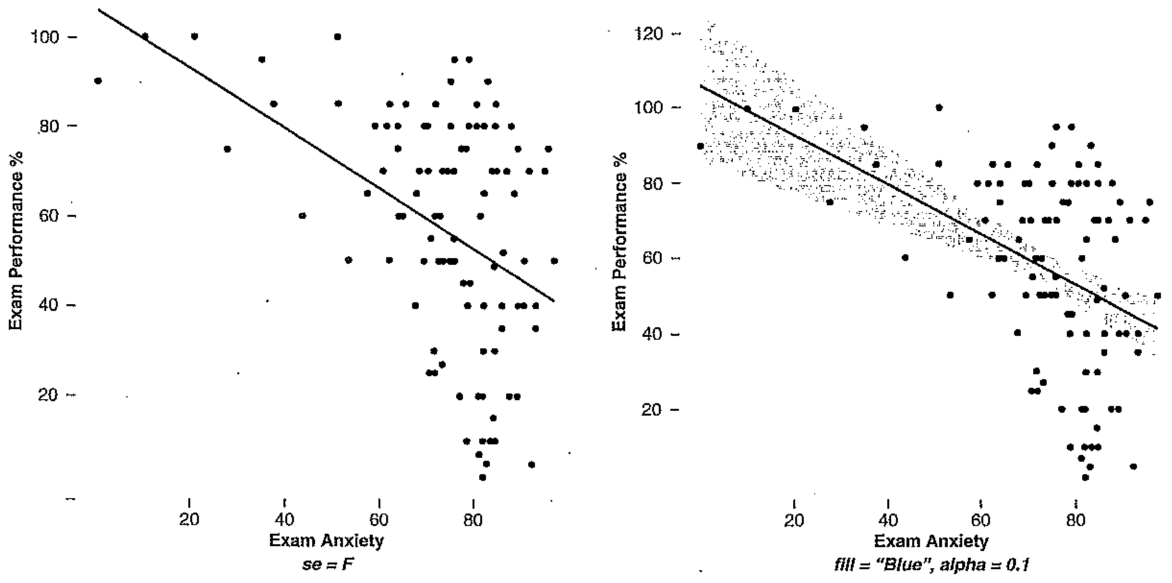
[7] You'll notice that the figure doesn't have a red line but what you see on your screen does, that's because this book isn't printed in colour which makes it tricky for us to show you the colourful delights of R. In general, use the figures in the book as a guide only and read the text with reference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

FIGURE 4.15
A simple scatterplot with a regression line added



(140)

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.)

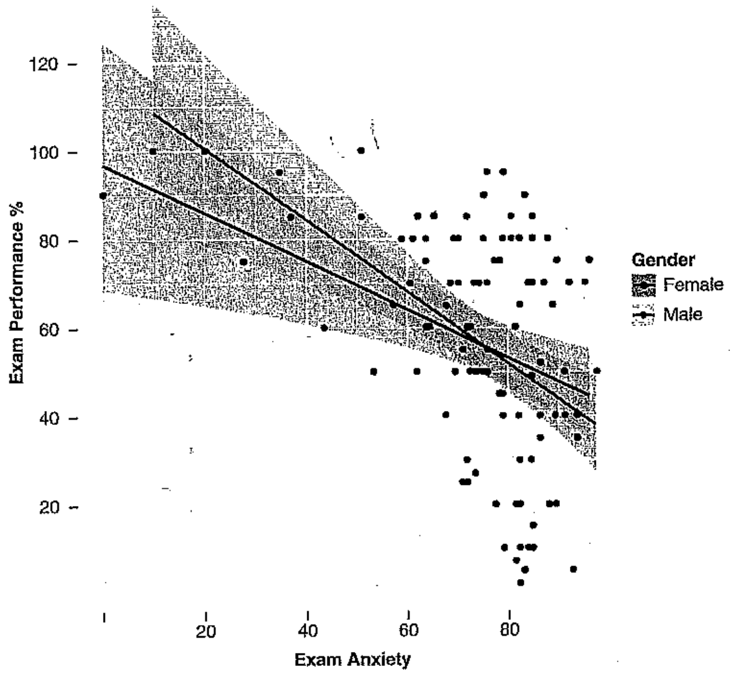


FIGURE 4.17 Scatterplot of exam anxiety and exam performance split by gender