Using statistical methods in R: a how-to guide

Ken Butler
Lecturer (Statistics), UTSC
`butler@utsc.utoronto.ca`

February 9, 2013

Contents

Chapter 1

Getting R and R Studio

1.1 Introduction

R is a free, open-source program for doing Statistics. It is what statisticians use. It looks a bit peculiar at first, but you'll find that it's very powerful, and, with practice, you can harness that power. Because it is open-source, anyone can work on it, and a lot of people do, including a number of statisticians, who are taking care that it behaves properly. (Unlike Excel!)

1.2 Getting R

The central website where R lives is `r-project.org`. There, you can read a good deal more about what R is, read its (copious) documentation, and so on. There is a lot of R stuff, and it is stored in various sites around the world called *mirrors*, so that you can always download R and R-related stuff from somewhere near you.

The first thing to do, then, is to choose a mirror. Below the pretty pictures, click on the "CRAN mirror" link, and on the resulting page, scroll down the list until you find `http://probability.ca/cran/`. Click on that, and the top box invites you to download R, according to whether you run Windows, a Mac or Linux.

Say you're running Windows: then select that, select "Install R for the first time", then click Download in the top box. R installs the same way as any other Windows program, and you end up with a big R on your desktop. Double-clicking that will run R.

See Section 2.1 for what to do next.

1.3 Getting R Studio

The default interface to R doesn't look very pretty, and though it is perfectly usable, I recommend R Studio, which is a front end to R that allows you to keep things together, deal with output and graphs, check out help files and so on.

The RStudio web site is `rstudio.org`. The main page shows you some screenshots, links to a video showing its features, and has a Download link top right. To download, click on that, then select Download RStudio Desktop, then select your operating system (Windows, Mac, Linux). The webpage decides which operating system you have, and offers a recommended choice at the top. If you are running Windows, this downloads and installs in the usual fashion (actually, it does for other operating systems too).

Chapter 2

To begin at the beginning

2.1 Baby steps

OK, so now we have R installed and running. When we run R, it gives us Figure 2.1, or something like it:

Now what? (If you are running RStudio, this will appear in the bottom left Console window, which is the only one to be concerned with right now.)

That `>` is called a **prompt**. R is waiting for you to type something, and when you do (and hit Enter) it will do what you ask it to, including producing any output, and then give you another prompt.

What to type? Well, we can enter a small amount of data into a *variable* `x`, using the `c` function, as in Figure 2.2. The `c` means "join together the things in the brackets". R just accepts that and doesn't give any output. This means "everything worked". (If it didn't work, you'd get an error message describing what didn't work, and you could try again.)

How do we know that R stored those values in the variable `x`? Type a variable's name at the prompt to have a look at it, as in Figure 2.3. All there, in the same order that we entered them. The `[1]` means that `x` is being listed starting at its first value.

If a variable is longer than would fit on one line, you see something like Figure 2.4. The first line generates all the values from 4 to 40 inclusive. The first value on the second line, 29, is the 26th value of `y`, as evidenced by the `[26]` at the beginning of the line.

```
R version 2.13.1 (2011-07-08)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: i486-pc-linux-gnu (32-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 2.1: What you see on starting R

```
> x=c(8,9,11,7,13,22,6,14,12)
```

Figure 2.2: Entering a little data

```
> x

[1]  8  9 11  7 13 22  6 14 12
```

Figure 2.3: Seeing the contents of a variable

```
> y=4:40
> y

 [1]  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
[26] 29 30 31 32 33 34 35 36 37 38 39 40
```

Figure 2.4: Displaying a longer variable

```
> x

[1]  8  9 11  7 13 22  6 14 12

> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   6.00    8.00   11.00   11.33   13.00   22.00
```

Figure 2.5: Summary of x

```
> sum(x)

[1] 102

> length(x)

[1] 9

> sum(x)/length(x)

[1] 11.33333
```

Figure 2.6: Calculations for the mean

2.2 Measures of centre

Let's remind ourselves of what `x` was, and produce a summary of it, as shown in Figure 2.5. What's in this? Some of the things you can guess. `Min.` and `Max.` are the smallest and largest, and `Mean` is the mean (duh!).

To convince yourself that the mean is correct, you can do the calculation in 2.6. The values in `x` add up to 102, there are 9 of them, and the mean is 102 divided by 9, or 11.33.

To see what the rest of the output from `summary` is, let's sort `x` into ascending order, as in Figure 2.7. There are 9 values, so the middle one is the 5th ($5 = (9+1)/2$), which is 11. That's the **median**. Also, the value labelled `1st Qu.`

```
> sort(x)

[1]  6  7  8  9 11 12 13 14 22
```

Figure 2.7: Sorted data

```
> w=4:9
> w

[1] 4 5 6 7 8 9

> summary(w)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   4.00    5.25    6.50    6.50    7.75    9.00 
```

Figure 2.8: Summary of `4:9`

is 8, and note that 2 of the data values are less than 8 and 6 are more than 8. 8 is called the **first quartile**, since a quarter of the values (not counting 8 itself) are below it. Similarly, 13 is the **third quartile**, since three-quarters of the data values other than 13 are below it and a quarter above.

I had nine data values, which meant that the median and quartiles were actually data values (9 is one more than a multiple of 4).

If you have, say, 6 data values, the median and quartiles come out in between the data values. For example, what are the median and quartiles for 4, 5, 6, 7, 8, 9? See Figure 2.8 for what R thinks. The median here has to be between 6 and 7 to get half the data values on each side. Any value between 6 and 7 would work, but by convention we go halfway between.

If the first quartile were 5, we'd have $1/5 = 20\%$ of the data values below it, not enough, and if the median were between 5 and 6, we'd have $2/6 = 33\%$ of the data values below it, which is too many. We can't hit exactly 25%, so a sensible answer is between 5 and 6 but closer to 5.

2.3 Measures of spread, part 1

The mean and median are measures of "centre": what are the data values typically near? We also want to know about how *spread out* the data values are: are they all close to the centre, or are some of them far away?

A measure of spread that goes with the median is called the **interquartile range**. It's just the 3rd quartile minus the 1st one. For `x`, you get it as shown in Figure 2.9. For reference, I've shown you the summary again there. The interquartile range is indeed $13 - 8 = 5$.

Or you can obtain the quartiles individually as in Figure 2.10. Note that the function you need is called `quantile` with an "n". `q` is a list (in R terms, a

```
> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   6.00    8.00   11.00   11.33   13.00   22.00

> IQR(x)

[1] 5
```

Figure 2.9: Interquartile range of x

```
> q=quantile(x)
> q

  0%  25%  50%  75% 100%
   6    8   11   13   22

> q[2]

25%
  8

> q[4]

75%
 13

> q[4]-q[2]

75%
  5

> q["75%"]-q["25%"]

75%
  5
```

Figure 2.10: Quartiles of x

vector) of 5 things. The second one is the 1st quartile, and the fourth one is the 3rd quartile. You can extract the second and fourth things in `q` as shown. (If you know about "arrays" in programming, it's the same idea.)

See those percentages above the values in `q` in Figure 2.10? These are called the *names attribute* of `q`, and if a vector has a names attribute, you can also get at the values in it by using the names. You might argue that `q["75%"]` is a more transparent way of getting the 3rd quartile than `q[4]` is. (The names attribute of `q[4]` seems to have gotten carried over into the calculation of the inter-quartile range.)

The output from `quantile(x)` is sometimes known as the "five-number summary": the minimum, 1st quartile, median, 3rd quartile and maximum of the data values.

2.4 Pretty pictures and help files

2.4.1 Histograms

Enough calculation for a moment. Let's have a look at some pictures. Pictures are one of R's many strengths, and they can be configured in all sorts of ways. But that's for later. Let's start with a histogram, as shown in Figure 2.11. Four of `x`'s values are between 5 and 10, four more are between 10 and 15, none are between 15 and 20, and one is between 20 and 25. The height of the bars, on the scale labelled Frequency, tells you how many values fall into each interval.

R chose the intervals itself, but it can be changed if you want. To find out how, and also to find out how R decides whether a value of exactly 10 goes into the 5–10 bar or the 10–15 bar, we can have a look at the Help for `hist`. This can be done by typing `?hist` (or a question mark followed by the name of any other function, like `?summary` or even `?mean`.

Help files have a standard format:

Description of what the function does

Usage: how you make it go

Arguments or options that you can feed into the function. This list is often rather long, but R has defaults for most things, so you won't need to specify most of these.

Details of the computation (something conspicuously missing from Excel's documentation!)

```
> hist(x)
```

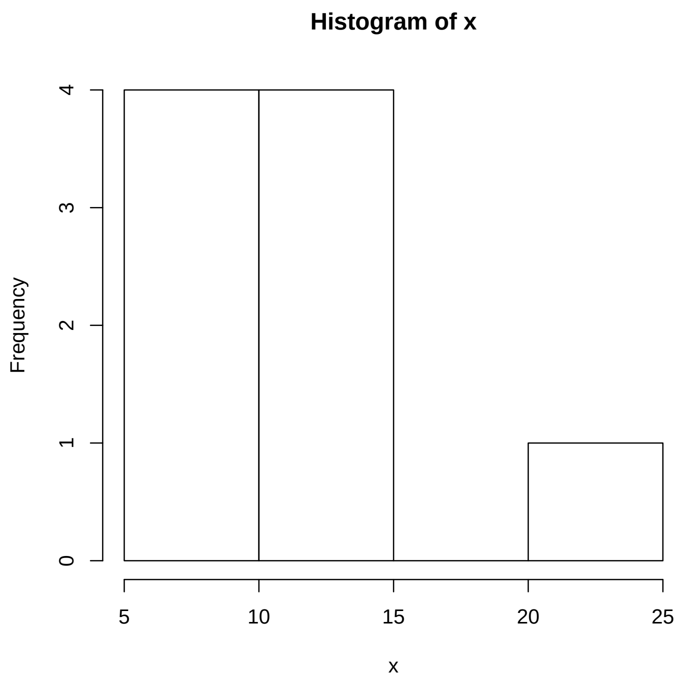


Figure 2.11: Histogram of `x`

```
> example("sd")

sd> sd(1:2) ^ 2
[1] 0.5
```

Figure 2.12: Example for `sd`

Value: what comes back from the function. For `hist`, this is (predictably) a picture of a histogram, but it also returns some values, which you can look at by storing them in a variable.

References to where the ideas for the function came from.

See Also: since you need to specify R commands/functions by name, you can look at the help for functions you *do* know by name to get ideas of which other functions you might look at.

Examples for you to copy and paste.

As for the last thing: the easiest way to run the examples is to type `example("hist")` (where the thing inside the brackets, in quotes, is what you want the examples for). The help examples for `hist` have a lot of pictures, but the example for `sd` (there is only one) looks like Figure 2.12, showing that the numbers 1 and 2 have standard deviation 0.707, so that when you square it you get 0.5.

As I said, Help files are typically long and detailed. If you want to find something, it'll probably be in there. Let me just pick out some excerpts from the help for `hist`. First, in the Arguments section:

```
  breaks: one of:

            - a vector giving the breakpoints between histogram cells,

            - a single number giving the number of cells for the
              histogram,

            - a character string naming an algorithm to compute the
              number of cells (see Details),

            - a function to compute the number of cells.

          In the last three cases the number is a suggestion only.
```

This is how you specify which intervals to use for the histogram. Going back to Usage reveals:

```
> hist(x,breaks=10)
```

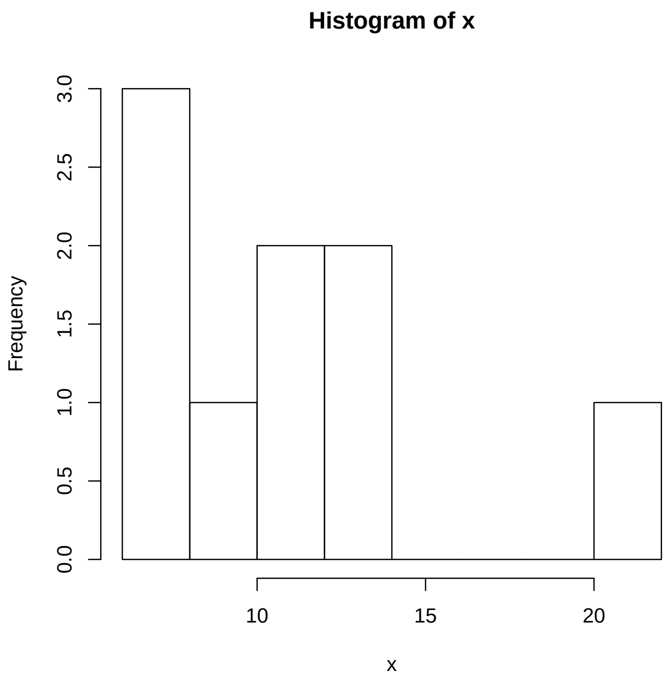


Figure 2.13: Histogram with (about) 10 bars

```
     hist(x, breaks = "Sturges",
```

and a lot of other stuff. This means that the default for `breaks` is "Sturges", which is "a character string naming an algorithm to compute the number of cells" (which depends on how much data you have).

We could choose to use (about) 10 bars, as in Figure 2.13. We got 8 bars, including the 3 missing ones, because R "prettifies" the intervals to be 6–8, 8–10, 10–12 etc.

Or we could choose to have breakpoints 4, 8, 12, etc, as in Figure 2.14. Here we got exactly what we asked for.

To answer the other question we had, about which bar a value exactly on the boundary goes into, we find:

```
     hist(x, breaks = "Sturges",
          freq = NULL, probability = !freq,
> hist(x,breaks=c(4,8,12,16,20,24))
```

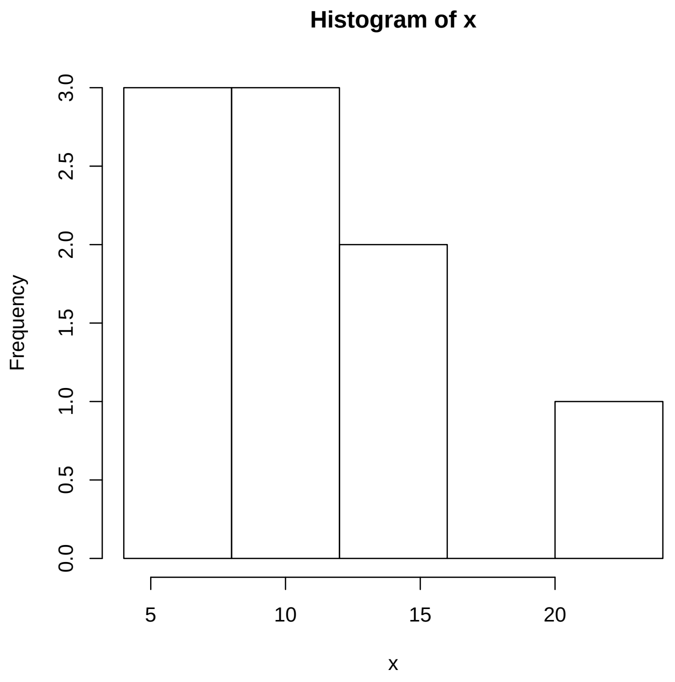


Figure 2.14: Histogram with breakpoints 4, 8, 12,. . .

```
> x

[1]  8  9 11  7 13 22  6 14 12

> stem(x)

  The decimal point is 1 digit(s) to the right of the |

  0 | 6789
  1 | 1234
  1 |
  2 | 2
```

Figure 2.15: Stemplot for `x`

```
        include.lowest = TRUE, right = TRUE,...

   right: logical; if TRUE, the histogram cells are right-closed
          (left open) intervals.
```

Undoing the mathy-speak, this means a value on the boundary by default goes into the interval for which it is on the right side. (Thus 10 would fall in 6–10, not 10–14.)

2.4.2 Stemplots and boxplots

There are some other kinds of pictures for variables like `x`. Two of these are the *stemplot* and the *boxplot*. A stemplot for our variable `x` looks like Figure 2.15.

The digits to the left of the `|` are called "stems". For example, the stems here are tens. The digits to the right of the `|` are called "leaves", and each digit represents one data point. Thus the largest data value is 22 (2 tens and 2 units), and the smallest are 6, 7, 8, and 9 (0 tens and respectively 6, 7, 8 and 9 units).

How do we know the stems are 10s? The line about "the decimal point is 1 digit to the right of the `|`". That means the smallest value is not 0.6 (which it would be if the decimal point were at the `|`), but 10 times bigger, 6. If the line had said "the decimal point is 1 digit to the *left* of the `|`", the lowest value would have been not 0.6 but 0.06.

Note the similarity between a stemplot and a sideways histogram. As it happens, the first histogram, with bars 4, 4, 0 and 1 high, is *exactly* this stemplot sideways. But a stemplot conveys extra information: because the leaves contain values, we can see from the stemplot that the largest value is 22, not just "between 20 and 25".

```
> boxplot(x)
```

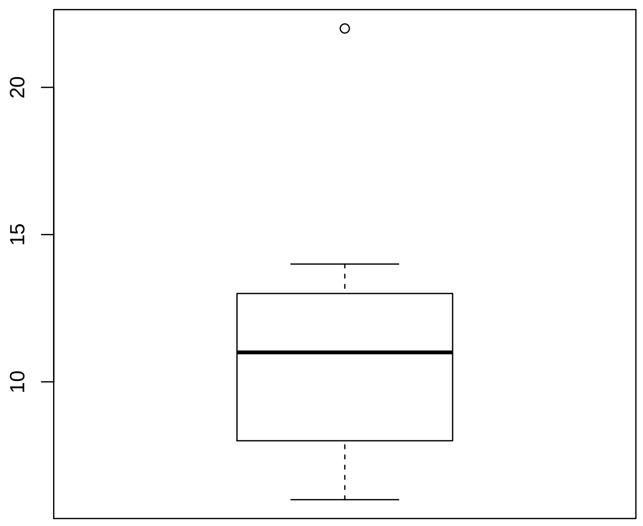


Figure 2.16: Boxplot of `x`

Note that in all these plots, there's a gap between 22 and the other values, as if to say "22 is noticeably higher than the other values".

Another graphical display worth knowing about is the *boxplot*. This depends on the median and quartiles. For our data, it looks like Figure 2.16.

The plot consists of a "box" and two "whiskers", one either side of the box. The line across the box marks the median (11). The box goes up to the 3rd quartile (13) and down to the 1st quartile (8). There is a procedure that determines whether a value is an "outlier" or unusual value. The whiskers extend to the most extreme non-unusual value, and outliers are plotted separately. The most extreme non-unusual values are 6 at the bottom and 14 at the top. The value 22 is an outlier; in fact, the graph shows you how much of an outlier it is.

```
> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   6.00    8.00   11.00   11.33   13.00   22.00

> sd(x)

[1] 4.84768

> mean(x)

[1] 11.33333
```

Figure 2.17: Standard deviation of x

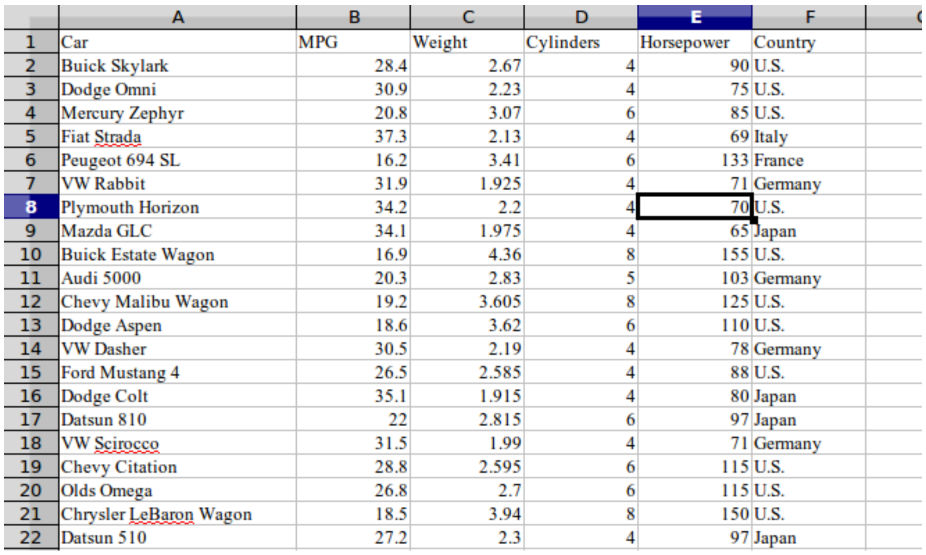
2.4.3 Spread part 2

One of the things you see from a graph is that data values are typically not all the same: there is some *variability* attached to them.

One way of measuring this is the interquartile range (3rd quartile minus 1st). On a boxplot, this is the height of the box. One way of understanding the inter-quartile range is as the spread specifically of the *middle half of the data*. The quarters of the data above the 3rd quartile and below the 1st don't enter into it. This is, in some ways, a good thing, because "off" values like 22 are not affecting our assessment of what kind of variability we have. But there can be a fine line between "typical" variability and "off" values.

In the same way that the mean uses all the data values, there is a measure of spread called the **standard deviation** that does the same. This, roughly speaking, says how far away your data values might get from the mean. It doesn't come with the output from `summary`, but is not too hard to get. You can also get the mean by itself if you don't want the other output from `summary`. See Figure 2.17.

So why do we have two measures of centre and two measures of spread? I hinted at this above, when I said that the mean and standard deviation use all the data, and the median and interquartile range don't use the extreme high and low values. In choosing which pair to use, the issue is whether you believe you have any "off" values. If you don't, it's a good idea to use everything (mean and standard deviation), but if you have any outliers, it's better not to let them have an undue influence, in which case median and interquartile range will be better. In the case of our `x`, the boxplot (especially) reveals 22 to be an outlier, which should scare us away from the mean and SD. We should describe these data using the median and interquartile range.



	A	B	C	D	E	F
1	Car	MPG	Weight	Cylinders	Horsepower	Country
2	Buick Skylark	28.4	2.67	4	90	U.S.
3	Dodge Omni	30.9	2.23	4	75	U.S.
4	Mercury Zephyr	20.8	3.07	6	85	U.S.
5	Fiat Strada	37.3	2.13	4	69	Italy
6	Peugeot 694 SL	16.2	3.41	6	133	France
7	VW Rabbit	31.9	1.925	4	71	Germany
8	Plymouth Horizon	34.2	2.2	4	70	U.S.
9	Mazda GLC	34.1	1.975	4	65	Japan
10	Buick Estate Wagon	16.9	4.36	8	155	U.S.
11	Audi 5000	20.3	2.83	5	103	Germany
12	Chevy Malibu Wagon	19.2	3.605	8	125	U.S.
13	Dodge Aspen	18.6	3.62	6	110	U.S.
14	VW Dasher	30.5	2.19	4	78	Germany
15	Ford Mustang 4	26.5	2.585	4	88	U.S.
16	Dodge Colt	35.1	1.915	4	80	Japan
17	Datsun 810	22	2.815	6	97	Japan
18	VW Scirocco	31.5	1.99	4	71	Germany
19	Chevy Citation	28.8	2.595	6	115	U.S.
20	Olds Omega	26.8	2.7	6	115	U.S.
21	Chrysler LeBaron Wagon	18.5	3.94	8	150	U.S.
22	Datsun 510	27.2	2.3	4	97	Japan

Figure 2.18: Spreadsheet of cars data

2.5 Reading in real data

Enough toy data. How do we read in serious data sets, ones that live in a spreadsheet or other kind of text file?

Figure 2.18 is a screenshot of some data I have in a spreadsheet. There are 38 rows of data altogether, of 38 different cars. For each car, 5 variables are recorded: gas mileage (miles per US gallon), weight (US tons), cylinders in engine, horsepower of engine, and which country the car comes from.

First step is to save the spreadsheet in `.csv` format. (This stands for "comma-separated values", and is a way of writing the data in a spreadsheet to a text file that other software, such as R, can read. Formulas don't translate, just the values that the formulas evaluate to.) I called the file `cars.csv`.

Figure 2.19 shows how to read this into R. `read.csv` requires at least one input: the name of the `.csv` file containing the data. In our spreadsheet had a top row naming the columns, which we'd like R to respect. (If you don't have a header row, use `header=F`. I get the best results specifying `header=` either way.)

If `read.csv` doesn't work for you, there is a more complicated command `read.table` that reads delimited data from a text file. You might have a smallish data set that you can type into Notepad (or a text file in R Studio), and you can arrange things so that values are separated by spaces and text stuff is single words. This is exsctly what `read.table` reads in. Give it a file name and an indication of

```
> cars=read.csv("cars.csv",header=T)
> cars

                        Car  MPG Weight Cylinders Horsepower Country
1             Buick Skylark 28.4  2.670         4         90    U.S.
2                Dodge Omni 30.9  2.230         4         75    U.S.
3            Mercury Zephyr 20.8  3.070         6         85    U.S.
4               Fiat Strada 37.3  2.130         4         69   Italy
5            Peugeot 694 SL 16.2  3.410         6        133  France
6                 VW Rabbit 31.9  1.925         4         71 Germany
7          Plymouth Horizon 34.2  2.200         4         70    U.S.
8                 Mazda GLC 34.1  1.975         4         65   Japan
9        Buick Estate Wagon 16.9  4.360         8        155    U.S.
10                Audi 5000 20.3  2.830         5        103 Germany
11       Chevy Malibu Wagon 19.2  3.605         8        125    U.S.
12              Dodge Aspen 18.6  3.620         6        110    U.S.
13                VW Dasher 30.5  2.190         4         78 Germany
14           Ford Mustang 4 26.5  2.585         4         88    U.S.
15               Dodge Colt 35.1  1.915         4         80   Japan
16               Datsun 810 22.0  2.815         6         97   Japan
17              VW Scirocco 31.5  1.990         4         71 Germany
18           Chevy Citation 28.8  2.595         6        115    U.S.
19               Olds Omega 26.8  2.700         6        115    U.S.
20   Chrysler LeBaron Wagon 18.5  3.940         8        150    U.S.
21               Datsun 510 27.2  2.300         4         97   Japan
22          AMC Concord D/L 18.1  3.410         6        120    U.S.
23    Buick Century Special 20.6  3.380         6        105    U.S.
24              Saab 99 GLE 21.6  2.795         4        115  Sweden
25               Datsun 210 31.8  2.020         4         65   Japan
26                 Ford LTD 17.6  3.725         8        129    U.S.
27             Volvo 240 GL 17.0  3.140         6        125  Sweden
28           Dodge St Regis 18.2  3.830         8        135    U.S.
29            Toyota Corona 27.5  2.560         4         95   Japan
30                 Chevette 30.0  2.155         4         68    U.S.
31        Ford Mustang Ghia 21.9  2.910         6        109    U.S.
32               AMC Spirit 27.4  2.670         4         80    U.S.
33 Ford Country Squire Wagon 15.5  4.054         8        142    U.S.
34                BMW 320i 21.5  2.600         4        110 Germany
35          Pontiac Phoenix 33.5  2.556         4         90    U.S.
36          Honda Accord LX 29.5  2.135         4         68   Japan
37    Mercury Grand Marquis 16.5  3.955         8        138    U.S.
38     Chevy Caprice Classic 17.0  3.840         8        130    U.S.
```

Figure 2.19: Reading in a .csv file

```
> cars$MPG

 [1] 28.4 30.9 20.8 37.3 16.2 31.9 34.2 34.1 16.9 20.3 19.2 18.6 30.5 26.5 35.1
[16] 22.0 31.5 28.8 26.8 18.5 27.2 18.1 20.6 21.6 31.8 17.6 17.0 18.2 27.5 30.0
[31] 21.9 27.4 15.5 21.5 33.5 29.5 16.5 17.0

> cars$Country

 [1] U.S.    U.S.    U.S.    Italy   France  Germany U.S.    Japan   U.S.
[10] Germany U.S.    U.S.    Germany U.S.    Japan   Japan   Germany U.S.
[19] U.S.    U.S.    Japan   U.S.    U.S.    Sweden  Japan   U.S.    Sweden
[28] U.S.    Japan   U.S.    U.S.    U.S.    U.S.    Germany U.S.    Japan
[37] U.S.    U.S.
Levels: France Germany Italy Japan Sweden U.S.
```

Figure 2.20: Referencing variables in a data frame

whether you have column headers, and you are good to go. The `read.table` function has *lots* of options, but in this simple situation you won't need to use them. Look at the `read.table` help if you want to be bamboozled!

The object `cars` is a table laid out the same way as the spreadsheet. The technical name for it is a *data frame*.

You can reference variables as shown in Figure 2.20.

A timesaver is the `attach` command, shown in Figure 2.21. Just `MPG` gets you all the MPG figures, without the need to prefix `cars$` every time; the same applies to all the other columns.

This stays in effect until you type `detach(cars)`, after which you'll need to type `cars$MPG` again.

2.6 More pretty pictures

Thus, a histogram of the MPG figures, once you've done the `attach` thing, is as simple as shown in Figure 2.22. Histograms normally have one peak, but this one has two: a primary one for 15–20 MPG, and a secondary one for 25–30. This often suggests that two (or more) different types of data have been combined into the one variable.

Figure 2.23 shows a variation on the histogram in Figure 2.22. We have added a smooth curve to show the two-peakiness of the histogram. This is kind of a smoothed-out version of the histogram, showing its essential shape without

```
> attach(cars)
> MPG

 [1] 28.4 30.9 20.8 37.3 16.2 31.9 34.2 34.1 16.9 20.3 19.2 18.6 30.5 26.5 35.1
[16] 22.0 31.5 28.8 26.8 18.5 27.2 18.1 20.6 21.6 31.8 17.6 17.0 18.2 27.5 30.0
[31] 21.9 27.4 15.5 21.5 33.5 29.5 16.5 17.0

> Country

 [1] U.S.    U.S.    U.S.    Italy   France  Germany U.S.    Japan   U.S.
[10] Germany U.S.    U.S.    Germany U.S.    Japan   Japan   Germany U.S.
[19] U.S.    U.S.    Japan   U.S.    U.S.    Sweden  Japan   U.S.    Sweden
[28] U.S.    Japan   U.S.    U.S.    U.S.    U.S.    Germany U.S.    Japan
[37] U.S.    U.S.
Levels: France Germany Italy Japan Sweden U.S.
```

Figure 2.21: Attaching a dataframe

```
> hist(MPG)
```

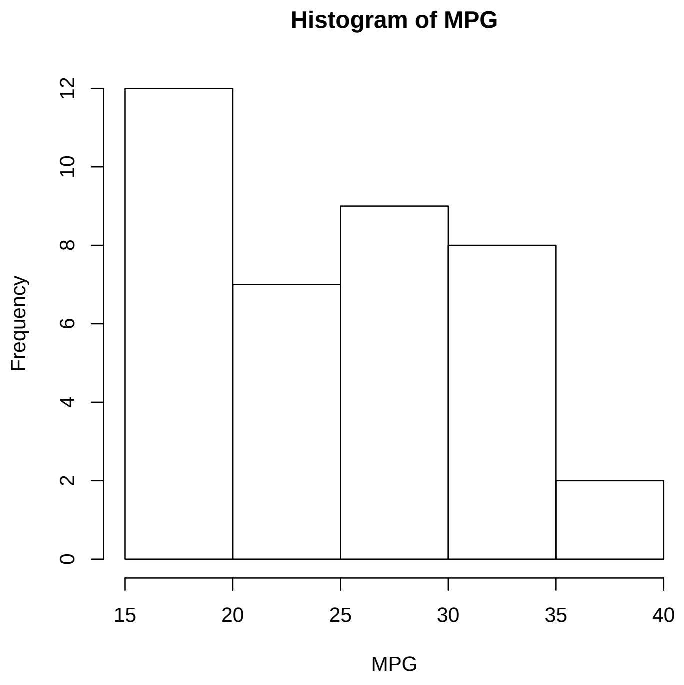


Figure 2.22: Histogram of MPG

```
> hist(MPG,freq=F)
> lines(density(MPG))
```

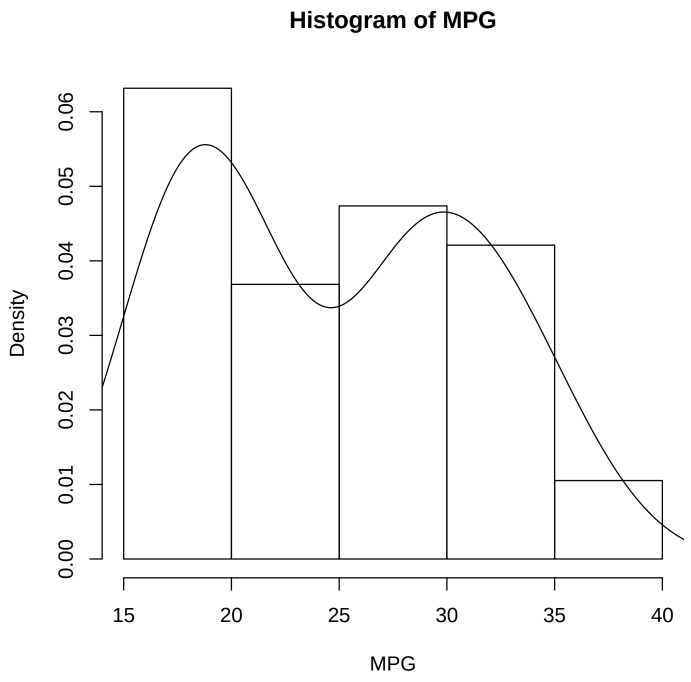


Figure 2.23: MPG histogram with density curve

```
> summary(MPG)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  15.50   18.52   24.25   24.76   30.38   37.30

> summary(Country)

 France Germany   Italy   Japan  Sweden    U.S.
      1       5       1       7       2      22
```

Figure 2.24: Summaries for MPG and Country

getting bogged down by details. To get the smooth curve to show up, we had to add an extra option to `hist`. This gets an appropriate[1] vertical scale.

Figure 2.24 shows an odd one. The summary for MPG is just like the summary for `x` above. But `Country` is just the name of the country each car comes from: categorical (nominal) rather than quantitative (numeric). `summary()` does an appropriate kind of summary for the thing it is fed. About the only appropriate summary for `Country` is a list of the countries and how many times they appear.

An appropriate picture for a categorical variable would be a bar plot, which you can get as shown in Figure 2.25. This counts up how many times each country appears in the data frame.

`plot` is, like `summary`, a command that does different things according to what you feed it. For example, feeding `plot` two quantitative variables makes a scatterplot. But more of that later.

Another use of a categorical variable is to divide data up into groups. For example, we can divide the cars up into 4, 6 and 8-cylinder engines, and we might then see how the MPG figures for the groups compare.

First, though, a boxplot of `MPG` (Figure 2.26). This shows that the median MPG is just under 25, and that the MPG figures are rather variable. Note that the boxplot didn't show the two peaks. We need a histogram (or stemplot) for that.

Boxplots are most useful for comparing distributions of different groups. You make a boxplot for MPG grouped by `Cylinders` as in Figure 2.27.

You see a couple of things: first, the typical gas mileage declines quite sharply as the number of cylinders increases, and second, the variability of gas mileage *within* each number of cylinders is quite a bit smaller than for the data as a whole (compare the heights of the boxes on this boxplot with the box on the

[1]If you care, the vertical scale is "density". The density times the width of the bar gives the proportion of all observations in that bar. For example, the 35–40 bar has density (height) 0.01 and width 5, so it has about $0.01 \times 5 = 0.05$ or 5% of all the observations.

```
> plot(Country)
```

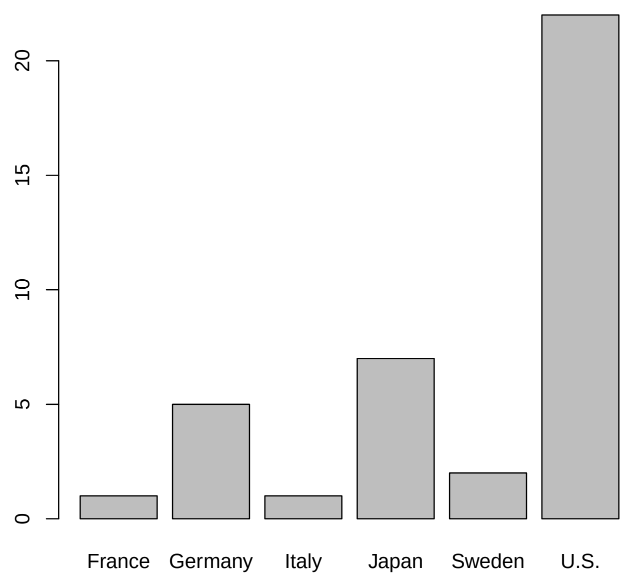


Figure 2.25: Barplot of Country

```
> boxplot(MPG)
```

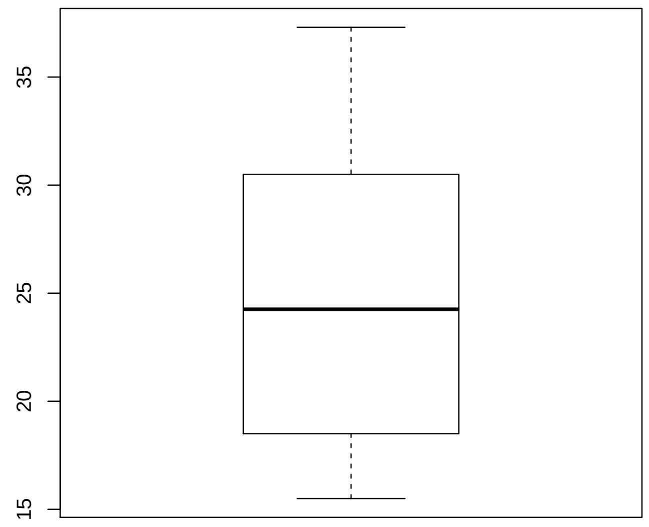


Figure 2.26: Boxplot of MPG

```
> boxplot(MPG~Cylinders)
```

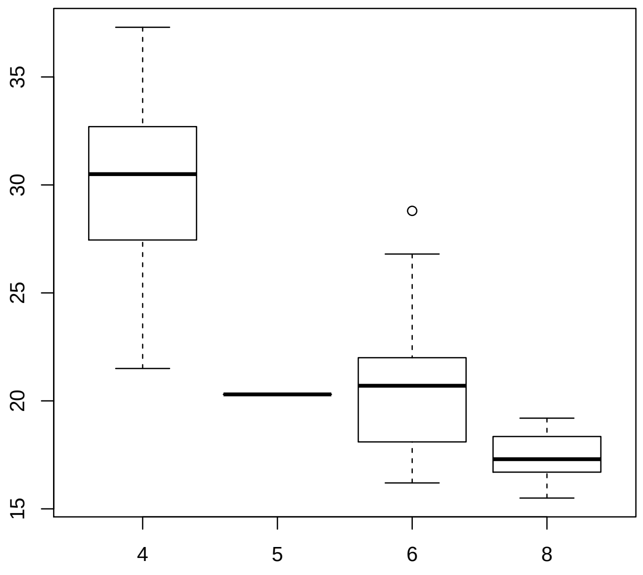


Figure 2.27: Boxplot of MPG grouped by cylinders

```
> cars[3,]

             Car  MPG Weight Cylinders Horsepower Country
3 Mercury Zephyr 20.8   3.07         6         85    U.S.

> cars[,4]

 [1] 4 4 6 4 6 4 4 4 8 5 8 6 4 4 4 6 4 6 6 8 4 6 6 4 4 8 6 8 4 4 6 4 8 4 4 4 8 8
```

Figure 2.28: Selecting a whole row or column

previous boxplot). That is, if you know how many cylinders a car's engine has, you can make a more informed guess at its gas mileage (compared to not having that knowledge).

The syntax `MPG ~ Cylinders` is known in R as a *model formula*. You can read it as "MPG as it depends on `Cylinders`". That is, do a separate boxplot of MPG for each value of `Cylinders`. The same idea is used later for specifying the response (dependent) and explanatory (independent) variables in a regression, or the response variable and factors in an ANOVA model. But not now.

2.7 Selecting stuff

How do you select individuals and variables? You can actually do some sophisticated things, but we'll start at the beginning. We'll illustrate with the cars data. First, data frames have rows (individuals) and columns (variables) that you can refer to by number. For example, the entry in the third row and fourth column is:

```
> cars[3,4]

[1] 6
```

Sometimes it's useful to select a whole row or a whole column. Do that by leaving whichever you want "all" of blank. Selecting all of the third row and then all of the fourth column is shown in Figure 2.28. The third row is all the data for the Mercury Zephyr, and the 4th column is the number of cylinders for all the cars. Thus `cars[3,4]` is the number of cylinders for the Mercury Zephyr.

You can also select *variables* by name. There is a special notation, Figure 2.29, for this, which is the same as `cars[,4]` above.

```
> cars$Cylinders

 [1] 4 4 6 4 6 4 4 4 8 5 8 6 4 4 4 6 4 6 6 8 4 6 6 4 4 8 6 8 4 4 6 4 8 4 4 4 8 8
```

Figure 2.29: Selecting a variable

```
> cars[6:9,c(3,5)]

  Weight Horsepower
6  1.925         71
7  2.200         70
8  1.975         65
9  4.360        155

> cars[6:9,c("Car","Cylinders")]

                Car Cylinders
6         VW Rabbit         4
7  Plymouth Horizon         4
8         Mazda GLC         4
9 Buick Estate Wagon        8
```

Figure 2.30: Selecting several rows and columns

You can also select several rows (or columns). This is shown in Figure 2.30. The notation `6:9` means "6 through 9", and if you want some not-joined-up collection of things, you use `c`: `c(3,5)` means "just 3 and 5". Thus the first part of Figure 2.30 selects cars 6 through 9 and columns (variables) 3 and 5, and the second part selects `Car` and `Cylinders` for those same rows.

If you're going to be referring to the variables (column names) a lot, you can `attach` a data frame as in Figure 2.31, and then refer to the variables by their own names. This works until you explicitly un-attach a data frame by `detach`ing it, Figure 2.32, and then you have to refer to this variable by `cars$Cylinders` again, since `Cylinders` now gives an error.

```
> attach(cars)
> Cylinders

 [1] 4 4 6 4 6 4 4 4 8 5 8 6 4 4 4 6 4 6 6 8 4 6 6 4 4 8 6 8 4 4 6 4 8 4 4 4 8 8
```

Figure 2.31: Attaching a data frame

```
> detach(cars)
> cars$Cylinders

 [1] 4 4 6 4 6 4 4 4 8 5 8 6 4 4 4 6 4 6 6 8 4 6 6 4 4 8 6 8 4 4 6 4 8 4 4 4 8 8
```

Figure 2.32: Detaching a data frame

The `attach` command is nice, and I use it a lot, but there is one thing to be aware of: if you already had a variable named `Cylinders`, you wouldn't (easily) be able to access it, because `Cylinders`, after `attach`, would refer to `cars$Cylinders` and not the original `Cylinders`. If you see what I mean.

Suppose now I wanted to know which cars had 8 cylinders. R has a concept called a "logical vector" which works like this:

```
> has8=(cars$Cylinders==8)
> has8

 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
[25] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[37]  TRUE  TRUE
```

The thing inside the brackets is a *condition* that is either true or false for each car (either it has 8 cylinders or it doesn't). Note the `==` to denote a *logical* equals. To the left of the brackets is an ordinary `=` to denote that we're saving the result of this condition in the variable called `has8`. `has8` is a string of trues and falses, indicating that cars number 9, 11, 20, 26, 28, 33, 37, 38 have 8 cylinders. This is kind of hard to read, so you can use a logical vector to pick out *which* cars have 8 cylinders, as in Figure 2.33. Note that `has8` picks out the rows I want, and I'm selecting all the columns, since my selection of columns is empty.

`cars` is a data frame, so `cars[has8,]` is a data frame as well. This means that you can select the countries where the 8-cylinder cars come from in any of the four ways shown in Figure 2.34. Any of those ways, we see right away that the 8-cylinder cars all come from the US.

Another idea: how do we find which car has the maximum horsepower? Two steps: first find the maximum horsepower, and then select the car(s) whose horsepower is equal to that. These are the first two lines of Figure 2.35. Another way is shown in the last two lines: `which.max` selects the numbers of the car(s) whose horsepower is equal to the maximum, and then we select those rows out of `cars`.

```
> cars[has8,]

                        Car  MPG Weight Cylinders Horsepower Country
9        Buick Estate Wagon 16.9  4.360         8        155    U.S.
11       Chevy Malibu Wagon 19.2  3.605         8        125    U.S.
20   Chrysler LeBaron Wagon 18.5  3.940         8        150    U.S.
26                 Ford LTD 17.6  3.725         8        129    U.S.
28           Dodge St Regis 18.2  3.830         8        135    U.S.
33 Ford Country Squire Wagon 15.5  4.054         8        142    U.S.
37     Mercury Grand Marquis 16.5  3.955         8        138    U.S.
38     Chevy Caprice Classic 17.0  3.840         8        130    U.S.
```

Figure 2.33: Selecting the cars that have 8 cylinders

```
> cars[has8,]$Country

[1] U.S. U.S. U.S. U.S. U.S. U.S. U.S. U.S.
Levels: France Germany Italy Japan Sweden U.S.

> cars[has8,"Country"]

[1] U.S. U.S. U.S. U.S. U.S. U.S. U.S. U.S.
Levels: France Germany Italy Japan Sweden U.S.

> cars$Country[has8]

[1] U.S. U.S. U.S. U.S. U.S. U.S. U.S. U.S.
Levels: France Germany Italy Japan Sweden U.S.

> cars[cars$Cylinders==8,"Country"]

[1] U.S. U.S. U.S. U.S. U.S. U.S. U.S. U.S.
Levels: France Germany Italy Japan Sweden U.S.
```

Figure 2.34: Countries of the cars that have 8 cylinders

```
> maxhp=max(cars$Horsepower)
> cars[cars$Horsepower==maxhp,]

                Car  MPG Weight Cylinders Horsepower Country
9 Buick Estate Wagon 16.9   4.36         8        155    U.S.

> which.max(cars$Horsepower)

[1] 9

> cars[which.max(cars$Horsepower),]

                Car  MPG Weight Cylinders Horsepower Country
9 Buick Estate Wagon 16.9   4.36         8        155    U.S.
```

Figure 2.35: Finding the car with maximum horsepower

What about the 6-cylinder cars? In Figure 2.36, we first find the horsepower of those cars, and then we draw a histogram of the values.

But `Country` is a categorical variable. What can we do with that?

`plot` is one of R's multi-coloured commands: the kind of plot that's produced is different, depending on what kind of thing that's being produced. What `plot`ting a categorical variable does is the same as this:

```
> tbl=table(cars[cars$Cylinders==6,"Country"]);
> tbl
```

```
 France Germany   Italy   Japan  Sweden    U.S.
      1       0       0       1       1       7
```

```
> barplot(tbl)
> hpsix=cars[cars$Cylinders==6,"Horsepower"]
> hpsix

 [1]  85 133 110  97 115 115 120 105 125 109

> hist(hpsix)
```

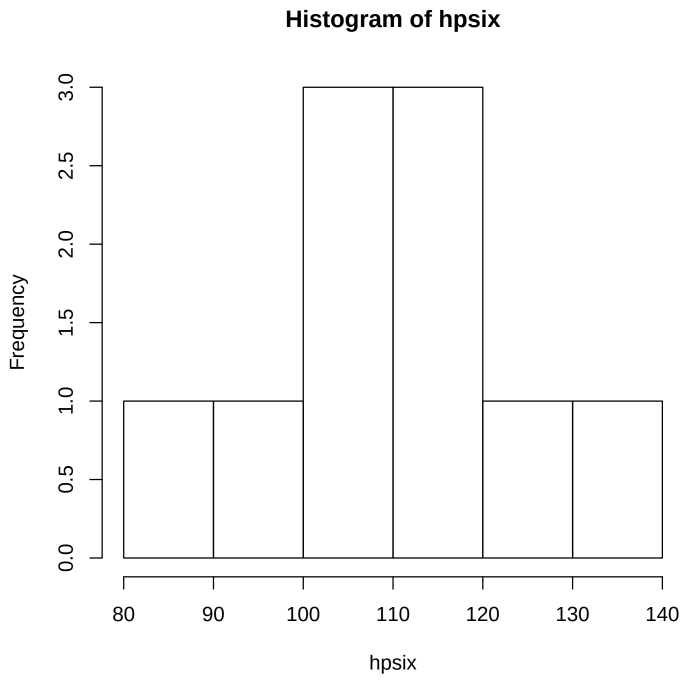


Figure 2.36: Drawing a histogram of the horsepowers of the cars that have six cylinders

```
> csix=cars[cars$Cylinders==6,"Country"]
> csix

 [1] U.S.   France U.S.   Japan  U.S.   U.S.   U.S.   U.S.   Sweden U.S.
Levels: France Germany Italy Japan Sweden U.S.

> plot(csix)
```

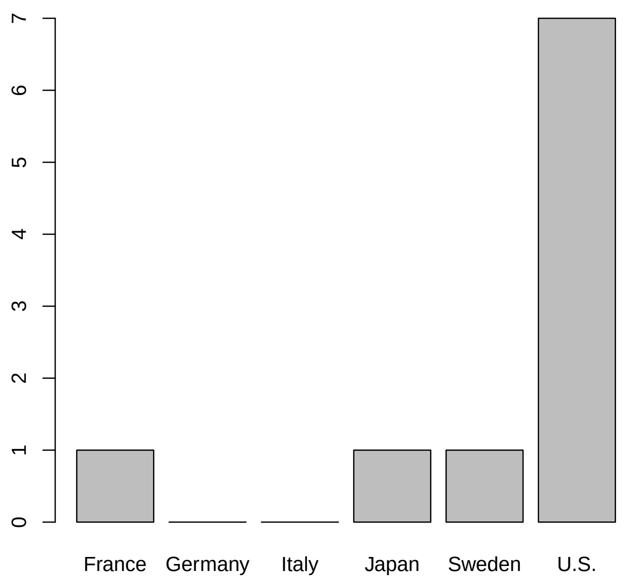
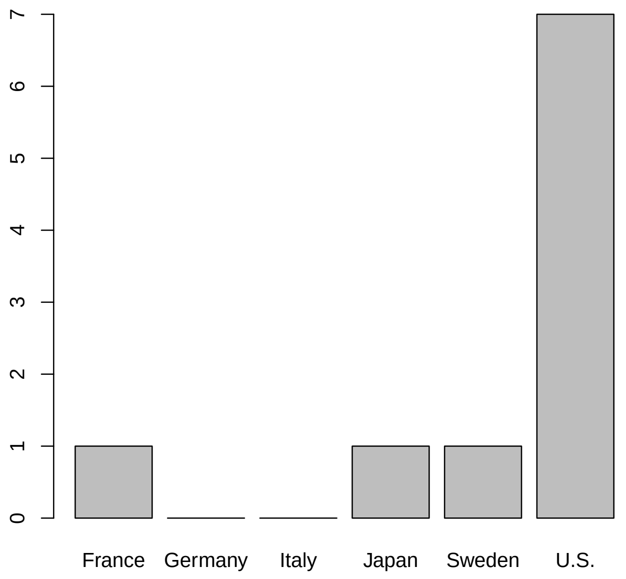


Figure 2.37: Plotting a categorical variable

```
> attach(cars)
> tbl=table(Cylinders,Country)
> detach(cars)
> tbl

         Country
Cylinders France Germany Italy Japan Sweden U.S.
        4      0       4     1     6      1    7
        5      0       1     0     0      0    0
        6      1       0     0     1      1    7
        8      0       0     0     0      0    8
```

Figure 2.38: Table of cylinders by country



This is making a bar chart in two steps: first counting up how many of the cars come from each country, and then making a chart with bars whose heights are those how-manys.

Whichever way you get the bar chart, you see that most of the 6-cylinder cars are also American, with single examples from France, Japan and Sweden.

Now suppose we want to get a barplot of how many cars of each number of cylinders come from each country. Two things are happening here:

1. we are treating cylinders as a categorical variable,
2. we are not doing any selection of rows from `cars`, so we can use the variables as is.

The first item doesn't really make any difference, except that with two categorical (or treated-as-categorical) variables, we seem to have to go via `table` and `barplot`. `table` can be fed two variables, and the output from that is interesting in itself. See Figure 2.38.

If you feed the variables to `table` in the other order, the table comes out with rows and columns interchanged, with a corresponding effect on the barplot, which you can investigate. (I also re-used the variable `tbl`, since I didn't need the old one any more.)

The table shows how many of the cars fall in each combination of country and number of cylinders. For example, 6 of the Japanese cars had 4 cylinders.

The plot in Figure 2.39 shows "stacked bars": for each country, the bars for different numbers of cylinders are stacked on top of each other. The darker the colour of the sub-bar, the fewer cylinders it corresponds to. I'm not a huge fan of stacked bars, so I went looking in the help for `barplot` to see if I could get the bars side by side.

```
  beside: a logical value.  If 'FALSE', the columns of 'height' are
          portrayed as stacked bars, and if 'TRUE' the columns are
          portrayed as juxtaposed bars.

   horiz: a logical value.  If 'FALSE', the bars are drawn vertically
          with the first bar to the left.  If 'TRUE', the bars are
          drawn horizontally with the first at the bottom.
```

So I need to include a `beside=T` in my call to `barplot`, as in Figure 2.40. (If you like your bars to be acrossways, you could similarly include `horiz=T`.)

Above each country is an array of bars showing how many cars from that country have respectively 4, 5, 6, and 8 cylinders. Most of the countries have the majority of their cars having 4-cylinder engines, except for the US, which has about an equal number of cars with each size of engine. This is, I suppose, visible in the stacked bar chart too, but I'd rather have the bars side by side so that I can compare directly how tall they are.

```
> barplot(tbl)
```

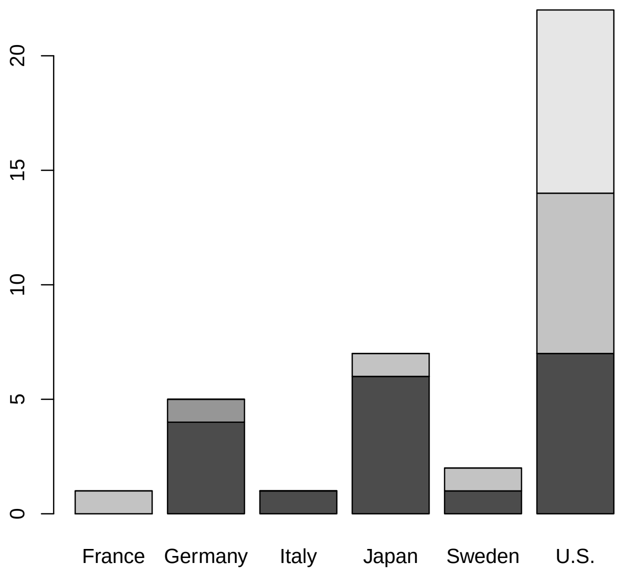


Figure 2.39: Barplot of cylinders by country

```
> barplot(tbl,beside=T)
```

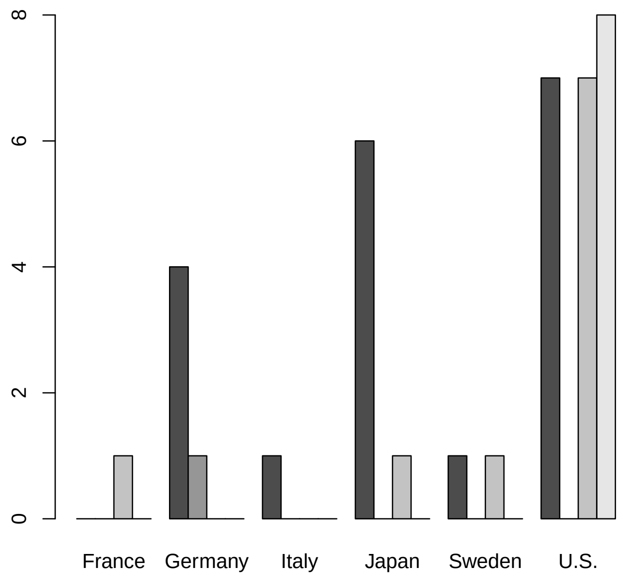


Figure 2.40: Barplot of cylinders by country with bars beside each other

```
> attach(cars)
> boxplot(MPG~Cylinders)
```

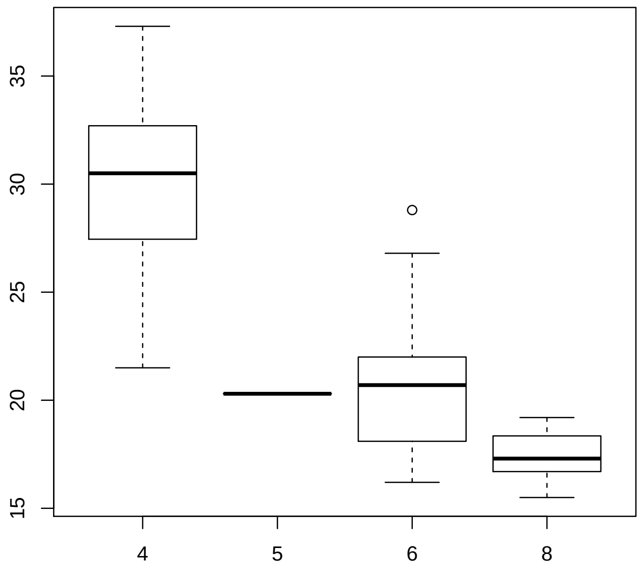


Figure 2.41: Boxplot of MPG by cylinders

2.8 Numerical summaries by subgroup

Recall our boxplot of MPG by cylinders? No? Well, Figure 2.41 shows it again.

We see that cars with more cylinders have worse gas mileage (no great surprise). What if we wanted to find the mean (or median or IQR or standard deviation) of the cars classified by number of Cylinders?

If you're a programmer, you might think of doing this with a loop, and you could probably do that in R as well. But R encourages you to think in terms of matrices and vectors: getting all the results at once.

There is a family of R functions with `apply` in their names. The one that does what we want here is `tapply`. You feed it three things: the variable you want to summarize, then the variable that divides your individuals into groups, and then finally what you want to calculate, like Figure 2.42.

```
> tapply(MPG,Cylinders,mean)

       4        5        6        8
30.02105 20.30000 21.08000 17.42500
```

Figure 2.42: Use of `tapply`

```
> tapply(MPG,Cylinders,sd)

       4        5        6        8
4.182447       NA 4.077526 1.192536

> tapply(MPG,Country,median)

 France Germany   Italy   Japan  Sweden    U.S.
   16.2    30.5    37.3    29.5    19.3    20.7
```

Figure 2.43: `tapply` for SDs and medians

As we saw in the boxplot, the typical MPG values go down pretty sharply as the number of cylinders increases (except for 5-cylinder engines, and there was only one of those). To find the standard deviations, replace `mean` by `sd`, as in Figure 2.43, which also shows how you'd make a table of median MPG by country.

The third argument to `tapply` is a *function*, one that takes a list of numbers (vector) and produces a result. This can even be a function you write yourself (see Chapter 7 for more on this). For example, suppose I write a function that takes a vector `x` and calculates the number of observations, mean, SD, median and IQR at once, as in Figure 2.44. First we calculate what we want to calculate and glue it together into a vector. Then — remember the `names` attribute? This will print out what each thing is next to its value. Let's test it on the MPG values, all of them (not subdivided by anything yet). See Figure 2.45.

Now, how do we calculate all of these things for each number of cylinders?

```
> mystats=function(x)
+   {
+     result=c(length(x),mean(x),sd(x),median(x),IQR(x))
+     names(result)=c("n","Mean","SD","Median","IQR")
+     result
+   }
```

Figure 2.44: A function to calculate sample statistics

```
> mystats(MPG)

        n      Mean        SD    Median       IQR
38.000000 24.760526  6.547314 24.250000 11.850000
```

Figure 2.45: Testing `mystats`

```
> tapply(MPG,Cylinders,mystats)

$`4`
        n      Mean        SD    Median       IQR
19.000000 30.021053  4.182447 30.500000  5.250000

$`5`
     n   Mean     SD Median    IQR
   1.0   20.3     NA   20.3    0.0

$`6`
        n      Mean        SD    Median       IQR
10.000000 21.080000  4.077526 20.700000  3.750000

$`8`
        n      Mean        SD    Median       IQR
 8.000000 17.425000  1.192536 17.300000  1.475000
```

Figure 2.46: Calculating statistics for each number of cylinders

```
> mylist=split(MPG,Cylinders)
> mylist

$`4`
 [1] 28.4 30.9 37.3 31.9 34.2 34.1 30.5 26.5 35.1 31.5 27.2 21.6 31.8 27.5 30.0
[16] 27.4 21.5 33.5 29.5

$`5`
[1] 20.3

$`6`
 [1] 20.8 16.2 18.6 22.0 28.8 26.8 18.1 20.6 17.0 21.9

$`8`
[1] 16.9 19.2 18.5 17.6 18.2 15.5 16.5 17.0

> sapply(mylist,mystats)

               4    5         6         8
n      19.000000  1.0 10.000000  8.000000
Mean   30.021053 20.3 21.080000 17.425000
SD      4.182447   NA  4.077526  1.192536
Median 30.500000 20.3 20.700000 17.300000
IQR     5.250000  0.0  3.750000  1.475000
```

Figure 2.47: Another way of calculating statistics for each number of cylinders

`mystats` is, as far as R is concerned, as good a function as `mean` or `median`, so we just feed that into `tapply` as the third thing, as in Figure 2.46.

This isn't the prettiest (it's what R calls a `list`), but you can see what it says. The measures of centre decrease as the number of cylinders increases, and so do the measures of spread. There's only one 5-cylinder car, which in MPG looks more like the 6-cylinder cars.

Let's try and pretty it up a bit by coming at it another way, as in Figure 2.47. `split` makes an R list of the MPG values separated out by Cylinders. `sapply` takes a list (which `mylist` is), and applies `mystats` to each element of it. This, as you see, makes a nice table with `Cylinders` as columns and the statistics as rows.

Suppose we wanted to calculate the mean MPG for each subgroup of `Cylinders` and `Country`? (We don't really have enough data for this to be meaningful, but we're not worrying about that right now.) We make a `list` out of all the categorical variables, like Figure 2.48.

```
> tapply(MPG,list(Country,Cylinders),mean)

               4    5        6      8
France        NA   NA 16.20000     NA
Germany 28.85000 20.3       NA     NA
Italy   37.30000   NA       NA     NA
Japan   30.86667   NA 22.00000     NA
Sweden  21.60000   NA 17.00000     NA
U.S.    30.12857   NA 22.22857 17.425
```

Figure 2.48: Table of mean MPG by country and cylinders

```
> tapply(MPG,list(Country,Cylinders),mystats)

        4         5         6         8
France  NULL      NULL      Numeric,5 NULL
Germany Numeric,5 Numeric,5 NULL      NULL
Italy   Numeric,5 NULL      NULL      NULL
Japan   Numeric,5 NULL      Numeric,5 NULL
Sweden  Numeric,5 NULL      Numeric,5 NULL
U.S.    Numeric,5 NULL      Numeric,5 Numeric,5
```

Figure 2.49: Trying to make a two-way table with `mystats`

This produces a nice table showing all the values. `NA` means that the mean couldn't be calculated (because there was no data). For example, all the 8-cylinder cars were American, so there are no means for any of the other countries for 8 cylinders.

This doesn't work for `mystats` because that function returns five values instead of just one. See Figure 2.49.

Because there are now *three* ways to classify things (`Country`, `Cylinders` and the statistic values), we'd need three dimensions to see it. But we can do something sensible by using `sapply` on a list like we did before with `mystats`, as in Figure 2.50.

Now we want to split MPG by both Country and Cylinders, so as to get all the combinations. The output from `sapply` isn't quite as nice as before, but it's still readable enough. R concocted names for the columns by gluing together `Country` and `Cylinders` and then put the statistics as rows.

I think the distinction between `NA` and `NaN` is that `NA`s cannot be calculated (not enough data), but `NaN`, which stands for "not a number", is an arithmetic error (caused in this case by trying to divide a total, 0, by a number of values, also

```
> list2=split(MPG,list(Country,Cylinders))
> head(list2)

$France.4
numeric(0)

$Germany.4
[1] 31.9 30.5 31.5 21.5

$Italy.4
[1] 37.3

$Japan.4
[1] 34.1 35.1 27.2 31.8 27.5 29.5

$Sweden.4
[1] 21.6

$U.S..4
[1] 28.4 30.9 34.2 26.5 30.0 27.4 33.5

> sapply(list2,mystats)

       France.4 Germany.4 Italy.4   Japan.4 Sweden.4    U.S..4 France.5
n             0  4.000000     1.0  6.000000      1.0  7.000000        0
Mean        NaN 28.850000    37.3 30.866667     21.6 30.128571      NaN
SD           NA  4.935247      NA  3.343451       NA  2.948284       NA
Median       NA 31.000000    37.3 30.650000     21.6 30.000000       NA
IQR          NA  3.350000     0.0  5.525000      0.0  4.300000       NA
       Germany.5 Italy.5 Japan.5 Sweden.5 U.S..5 France.6 Germany.6 Italy.6
n            1.0       0       0        0      0      1.0         0       0
Mean        20.3     NaN     NaN      NaN    NaN     16.2       NaN     NaN
SD            NA      NA      NA       NA     NA       NA        NA      NA
Median      20.3      NA      NA       NA     NA     16.2        NA      NA
IQR          0.0      NA      NA       NA     NA      0.0        NA      NA
       Japan.6 Sweden.6    U.S..6 France.8 Germany.8 Italy.8 Japan.8 Sweden.8
n            1        1  7.000000        0         0       0       0        0
Mean        22       17 22.228571      NaN       NaN     NaN     NaN      NaN
SD          NA       NA  4.063953       NA        NA      NA      NA       NA
Median      22       17 20.800000       NA        NA      NA      NA       NA
IQR          0        0  4.750000       NA        NA      NA      NA       NA
          U.S..8
n       8.000000
Mean   17.425000
SD      1.192536
Median 17.300000
IQR     1.475000

> detach(cars)
```

Figure 2.50: A second attempt

```
> 0/0

[1] NaN

> 2/0

[1] Inf
```

Figure 2.51: Dividing by zero

0). R also has an `Inf`, for "infinity", for when you divide something else by 0. See Figure 2.51.

I want to show you one more member of the `apply` family, which is just plain `apply`. This takes a matrix (such as a data frame) and applies a function to the rows or to the columns. Let's remind ourselves of what variables we had in `cars` (first line of Figure 2.52). There were six, including the identifier for the cars, but only the second, third, fourth and fifth are numeric. If you try to calculate a mean on `Country`, R will give you an error, and we don't want that! So let's just look at the four numeric variables by selecting out only those columns (second line of Figure 2.52).

Let's find the means of all those columns. `apply` needs three things: the name of the matrix (or data frame), which is the one we just constructed, then the "dimension" (1 is rows, 2 is columns), then the name of the function to apply, such as `mean`. See Figure 2.52. It doesn't matter whether your function returns one value (like `mean`) or five values (like `mystats`). `apply` handles it just fine, as you see in the last line of Figure 2.52.

The first one is just a list of all the means, arranged by variable, and the second one is a table with variables in the columns and statistics in the rows.

```
> head(cars)

             Car  MPG Weight Cylinders Horsepower Country
1  Buick Skylark 28.4  2.670         4         90    U.S.
2     Dodge Omni 30.9  2.230         4         75    U.S.
3 Mercury Zephyr 20.8  3.070         6         85    U.S.
4    Fiat Strada 37.3  2.130         4         69   Italy
5 Peugeot 694 SL 16.2  3.410         6        133  France
6      VW Rabbit 31.9  1.925         4         71 Germany

> cars.numeric=cars[,2:5]
> apply(cars.numeric,2,mean)

       MPG     Weight  Cylinders Horsepower
 24.760526   2.862895   5.394737 101.736842

> apply(cars.numeric,2,mystats)

             MPG     Weight Cylinders Horsepower
n      38.000000 38.0000000 38.000000   38.00000
Mean   24.760526  2.8628947  5.394737  101.73684
SD      6.547314  0.7068704  1.603029   26.44493
Median 24.250000  2.6850000  4.500000  100.00000
IQR    11.850000  1.2025000  2.000000   45.25000
```

Figure 2.52: Using `apply`

Chapter 3

Statistical Inference

3.1 Introduction

Displaying data is all very well, but sometimes we need to draw some conclusions about a larger **population** from which our data is a sample. (The assumption for most inference is that your data are a *simple random sample* from the population: that is, each element of the population is equally likely to be chosen in the sample, independently of other elements, like drawing from a hat or a lottery machine.)

3.2 Confidence intervals

Under the (probably unreasonable) assumption that the cars in our data set are a simple random sample of all cars, what might the mean MPG, say, of all cars be? To answer that, we can find a confidence interval for the mean. This comes from the R function `t.test` that also does hypothesis tests for the mean (coming up in Section 3.3).

Just read the bit that says "confidence interval". A 95% confidence interval for the mean MPG of all cars is from 22.61 to 26.91.

Perusal of the help for `t.test` reveals that you can get other confidence levels like 99% or 90% by passing the level in as shown in Figure 3.2. This confidence interval is longer than the previous one because we are "more confident" in it. (Precisely, in as much as 99%, as compared to 95%, of all possible samples, will the confidence interval contain the population mean.)

```
> cars=read.csv("cars.csv",header=T)
> attach(cars)
> tt=t.test(MPG)
> tt

        One Sample t-test

data:  MPG
t = 23.3125, df = 37, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 22.60848 26.91257
sample estimates:
mean of x
 24.76053
```

Figure 3.1: Confidence interval for mean MPG

```
> tt99=t.test(MPG,conf.level=0.99)
> tt99

        One Sample t-test

data:  MPG
t = 23.3125, df = 37, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
99 percent confidence interval:
 21.87645 27.64460
sample estimates:
mean of x
 24.76053
```

Figure 3.2: 99% confidence interval

```
> is.american=(Country == "U.S.")
> twot=t.test(MPG~is.american)
> twot

        Welch Two Sample t-test

data:  MPG by is.american
t = 2.0009, df = 30.748, p-value = 0.0543
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.08229979  8.46639070
sample estimates:
mean in group FALSE  mean in group TRUE
           27.18750            22.99545
```

Figure 3.3: Two-sample t-test on MPG of American vs. other cars

```
> names(twot)

[1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
[6] "null.value"  "alternative" "method"      "data.name"

> twot$conf.int

[1] -0.08229979  8.46639070
attr(,"conf.level")
[1] 0.95
```

Figure 3.4: Getting out just the CI

Reading the help file for `t.test` is a bit daunting because `t.test` also does inference for two samples, depending on what you pass into it. Let's illustrate that by dividing our cars into American and non-American ones, and see how their gas mileage compares. See Figure 3.3. This output tells you about the *difference* in means between American cars and non-American ones. The imported cars have a higher MPG by somewhere between 0 and about 8.5 miles per gallon. The confidence interval *does* include zero, though, so even though the suggestion is that imported cars are more fuel-efficient, it is plausible that there is actually no difference overall, and the difference we observed in our samples is just chance.

There is a way to get just the confidence interval out of this. As ever with R, you need to know the name of the thing you're asking for, which is what the first line of Figure 3.4 does. It looks as if `conf.int` does the thing we want. Or you can look at the help for `t.test`, and read down to where it says "Value:", where it says what those things returned from `t.test` (the things that `twot` actually contains) really are.

3.3 Hypothesis tests

A confidence interval answers a question like "what is the population mean" (or "what is the difference between the population means"). To answer a question like "could the population mean be x" or "is there a real difference between these two populations", we need a *hypothesis test*.

A hypothesis test has some ingredients:

- a **null hypothesis** that says "there is no difference between the population means" or "the population mean is given by some theory to be this."
- an **alternative hypothesis** that says that the null hypothesis is wrong: "there is a difference between the population means" or "the theory is wrong and the population mean is not what it says". We are trying to prove that the null hypothesis is wrong by collecting sufficiently good evidence against it (at which point we can **reject** the null hypothesis).
- A **significance level** α, the importance of which will become apparent in a moment. $\alpha = 0.05$ is a common choice.
- Some data, collected via a simple random sample.

Out of the test comes a **P-value**, which summarizes the strength of evidence against the null hypothesis (in favour of the alternative). We'll see that more in the examples below.

If you struggle with the logic here, bear in mind that it is analogous to a (legal) trial in court. The accused actually *is* either innocent or guilty, and the point of the trial is to make a decision about which of those the accused is. The Canadian legal system has a "presumption of innocence", which means that only if the evidence is such that the accused is proven guilty "beyond a reasonable doubt", can the court return a verdict of "guilty". If the evidence is not strong enough, the verdict has to be "not guilty".

Note that it is not the legal system's business to prove that an accused person is innocent. The job of a defence lawyer is to demonstrate a "reasonable doubt" by poking a sufficient number of holes in the prosecution's case. The presumption of innocence means that we are willing to let some actually guilty people go free, because imprisoning someone innocent is considered to be far worse.

The analogy here is that actual innocence corresponds to the null hypothesis being true, and actual guilt corresponds to the alternative hypothesis being true.

```
> twot=t.test(MPG~is.american)
> twot

        Welch Two Sample t-test

data:  MPG by is.american
t = 2.0009, df = 30.748, p-value = 0.0543
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.08229979  8.46639070
sample estimates:
mean in group FALSE  mean in group TRUE
           27.18750            22.99545
```

Figure 3.5: Two-sample t-test comparing American and other cars' MPG

Making a decision to reject the null hypothesis is like declaring the accused guilty: it is a big deal, requiring strong evidence to do so. Making a decision not to reject the null hypothesis is like declaring the accused not guilty. Note the parallelism in wording: we don't "accept the null hypothesis" in the same way that the accused is never declared "innocent".

The last part is the assessment of the evidence against the null hypothesis, which is where the P-value comes in. Let's see this in an example.

We'll compare the American and imported cars' MPG first. Let's suppose we've chosen the default $\alpha = 0.05$. The null hypothesis is that American and import cars have the same mean MPG, while the alternative is that the means are different. The P-value, as shown in Figure 3.5, is 0.0543. A *small* P-value means strong evidence against the null hypothesis. To assess this, you compare the P-value against α. In our case, the P-value is *not* smaller than α, so we cannot (quite) reject the null hypothesis. We have not been able to prove that the population mean MPG figures are different for domestic and imported cars.

You might say that the null hypothesis is declared to be "not guilty" here.

Now, you might be thinking: the only reason those American cars came out to have worse gas mileage is that they tended to have bigger engines. Couldn't we adjust for size of engine somehow? Well, there's a straightforward way, which is to compare the US and other 4-cylinder engines only. This is probably as good as anything here, because there aren't many import engines bigger than 4 cylinders anyway. (A more sophisticated method is the *analysis of covariance*, which we'll have a look at after we've studied regression and analysis of variance, from which it derives.)

Figure 3.6 shows how that one comes out. Once you compare cars with the

```
> twot4=t.test(MPG[Cylinders==4]~is.american[Cylinders==4])
> twot4

        Welch Two Sample t-test

data:  MPG[Cylinders == 4] by is.american[Cylinders == 4]
t = -0.0947, df = 16.927, p-value = 0.9257
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.963921  3.623445
sample estimates:
mean in group FALSE  mean in group TRUE
           29.95833            30.12857
```

Figure 3.6: Comparing MPG for 4-cylinder cars

same-sized engines, there is effectively no difference between the MPG figures.

Let's do another one. Suppose, in the last similar survey, the mean MPG figure had been 20. Is there evidence that the mean MPG of all cars has changed from 20, based on the results of this survey? Let's again take $\alpha = 0.05$. The null hypothesis is that the population mean is equal to 20, while the alternative is that it is not equal to 20. This is a *one-sample t*-test; we are assessing a given value based on one sample of data, rather than comparing two samples' worth of data. This time we have to specify the null-hypothesis mean, as in Figure 3.7.

The P-value is 0.00007, which is comfortably less than 0.05. This time, we can reject our null hypothesis, and conclude that the mean has indeed changed from 20. The confidence interval contains only values above 20, so it looks as if the mean MPG has increased.

Sometimes we have prior reasons to be interested in only one kind of change or difference. For instance, we might be interested only in whether MPG has *increased* since the last survey, and if it's decreased we don't care about that. (You might be testing a new drug, for example, and you only care about whether it works better than the current treatment. If it's worse you know it's not worth following up.) The point is that you have to have *prior* reasons for suspecting that things will increase. You can't do it by looking at the data first. That would be cheating!

When you're only searching for a difference one way, it's called a **one-sided test** and the P-value is figured a different way. For our cars, for example, you might feel (before looking at any numbers) that gas mileages are improving over time, so you might be interested in testing that mean MPG is now *greater than 20* as in the last survey (the alternative) against a null hypothesis that logically ought to be that the mean is *less than or equal to 20* (so that either the null or

```
> tt=t.test(MPG,mu=20)
> tt

        One Sample t-test

data:  MPG
t = 4.4821, df = 37, p-value = 6.895e-05
alternative hypothesis: true mean is not equal to 20
95 percent confidence interval:
 22.60848 26.91257
sample estimates:
mean of x
 24.76053
```

Figure 3.7: One-sample *t*-test

the alternative is true).

Let's make this happen for the car MPG. R does two-sided tests by default, so we have to specify the alternative that we want. The results are in Figure 3.8.

3.3.1 Matched pairs

Sometimes two samples is not two samples. Take a look at Figure 3.9. These are data on the foreleg and hindleg length of deer. But these are the *same* 10 deer that have both legs measured (2 measurements for each), rather than one lot of 10 deer that had their front legs measured, and a *different* 10 deer that had their hind legs measured, which is what we require for a two-sample *t* test. To get the right analysis, we need the `paired` option to `t.test`. (The analysis actually works on the differences between foreleg and hindleg length for each deer.) The analysis is shown in Figure 3.10. The P-value is small, so that we conclude that mean leg lengths differ; the confidence interval suggests that the foreleg is longer by between 1 and 5.5 centimetres.

Now let's do the *incorrect* two-sample analysis, as in Figure 3.11. This time the P-value is larger, and the confidence interval for difference in mean leg lengths is longer; there appears to be less information in the data. In fact, there is; nowhere in this (incorrect) analysis have we used the pairing, the fact that we have two measurements from each deer.

One way to understand what has happened is to plot the foreleg lengths against the corresponding hindleg lengths, as shown in Figure 3.12. I've put a lowess curve on there to guide the eye. The story is that deer with longer forelegs tend to have longer hindlegs as well. The two-sample analysis has failed to allow for

```
> tt=t.test(MPG,mu=20,alternative="greater")
> tt

        One Sample t-test

data:  MPG
t = 4.4821, df = 37, p-value = 3.448e-05
alternative hypothesis: true mean is greater than 20
95 percent confidence interval:
 22.96864      Inf
sample estimates:
mean of x
 24.76053

> detach(cars)
```

Figure 3.8: One-sided test

```
> deer=read.csv("deer.csv",header=T)
> deer

   Foreleg Hindleg
1      142     138
2      140     136
3      144     147
4      144     139
5      142     143
6      146     141
7      149     143
8      150     145
9      142     136
10     148     146
```

Figure 3.9: Matched pair data

```
> deer.paired=t.test(deer$Foreleg,deer$Hindleg,paired=T)
> deer.paired

        Paired t-test

data:  deer$Foreleg and deer$Hindleg
t = 3.4138, df = 9, p-value = 0.007703
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 1.113248 5.486752
sample estimates:
mean of the differences
                    3.3
```

Figure 3.10: Paired *t*-test output

```
> deer.two=t.test(deer$Foreleg,deer$Hindleg)
> deer.two

        Welch Two Sample t-test

data:  deer$Foreleg and deer$Hindleg
t = 1.978, df = 17.501, p-value = 0.06389
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.2122142  6.8122142
sample estimates:
mean of x mean of y
    144.7     141.4
```

Figure 3.11: Incorrect two-sample test for deer data

```
> plot(deer$Foreleg,deer$Hindleg)
> lines(lowess(deer$Foreleg,deer$Hindleg))
```

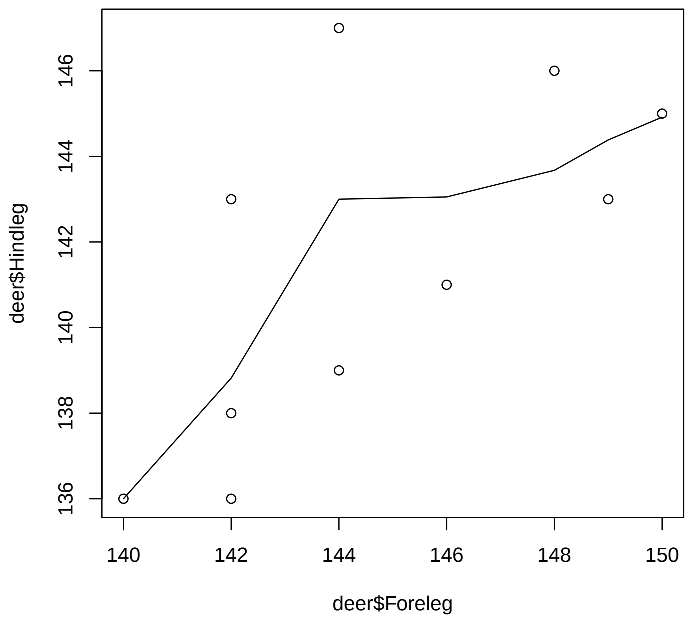


Figure 3.12: Scatterplot of deer leg lengths

the fact that some deer are just long-legged. As far as that analysis is concerned, foreleg length varies, and hindleg length varies, without acknowledging that they vary *together* as well. The paired analysis works on the differences, foreleg minus hindleg, so that if a particular deer has longer-than-average leg lengths, these will cancel each other out when taking the difference.

Another way to handle this analysis is as a *repeated measures*, or to treat deer as a *random effect*. We'll see more of that in Section 6.7. This line of thinking says that our deer are just a random sample of all possible deer, and we don't particularly care about differences between deer, just in the foreleg-hindleg comparison. This is the same attitude that we take towards blocks in Section 6.2.

3.4 About that P-value

One way of thinking about hypothesis testing is as a black box: to turn your data and hypotheses into a P-value, from which you make your decision. But what *is* a P-value exactly?

The definition is this: the P-value is **the probability of observing a value as extreme or more extreme than the one observed, if the null hypothesis is true**. That's a bit of a mouthful, but if we pull it apart, starting at the end, we get at the logic.

The end part says "if the null hypothesis is true". All of the logic behind the P-value is based on what might happen if the null hypothesis is true. There is nothing here about what might happen if the null hypothesis *isn't* true. That's got nothing to do with P-values.

Suppose you do a test and get a P-value of 0.20. Roughly speaking, that says that if the null is true, you have a 1-in-5 chance of getting a result like the one you observed. This is small, but not terrifically small, in that your result is the kind of thing that *might* occur by chance. Since we're giving the null hypothesis the benefit of the doubt, this isn't really strong enough evidence to reject it with. It's hardly "beyond all reasonable doubt".

Now suppose your P-value is 0.0000001. This is literally one in a million. That means (roughly) that if the null hypothesis were true, you would have a one in a million chance of observing a result like the one you observed. So now you have a choice: is the null hypothesis true and you observed something fantastically unlikely, or is it false? Most of us would opt for the latter.

Now how about if the P-value is 0.04? This is one in 25. So if we get this P-value, we observed something that, if the null hypothesis is true, is unlikely but far from impossible. So if we reject the null hypothesis here, we might be making a mistake and rejecting it when it's actually true. What to do?

The thing about hypothesis testing is that there is *always the risk of making a mistake*, which we have to live with. There is no way of eliminating the possibility of error. Think back to the court of law analogy: there, there are two kinds of mistake you can make: a "type 1 error", to declare a person guilty when they are in fact innocent, and a "type 2 error", to declare a person not guilty when they are in fact guilty.

Now, in a court of law, you could eliminate type 1 errors by declaring everyone not guilty. But by so doing, you will let a lot of guilty people go free (make a lot of type 2 errors). Likewise, you could eliminate type 2 errors by declaring everyone guilty, but you will lock up a lot of innocent people (make a lot of type 1 errors). The best course of action lies somewhere in between: to decide on the basis of the evidence as best you can, but to accept that mistakes will be made.

The terms "type 1 error" and "type 2 error" are actually statistical ones. A type 1 error is to reject the null hypothesis when it is actually true, and a type 2 error is to fail to reject the null hypothesis when it is actually false (ie. when you *want* to reject it). As in the legal analogy, if you eliminate one type of error, you'll make a lot of the other type. If you never reject the null hypothesis, you'll make a lot of type 2 errors, and if you always reject the null hypothesis, you'll make a lot of type 1 errors.

Your statistical training might have included the injunction to "reject the null hypothesis whenever the P-value is less than 0.05". You might have wondered what was so special about 0.05. This is a historical accident. One of the founders of Statistics was Sir Ronald Fisher, who was both influential and opinionated. He said:

> ... it is convenient to draw the line at about the level at which we can say: "Either there is something in the treatment, or a coincidence has occurred such as does not occur more than once in twenty trials."...
>
> If one in twenty does not seem high enough odds, we may, if we prefer it, draw the line at one in fifty (the 2 per cent point), or one in a hundred (the 1 per cent point). Personally, the writer prefers to set a low standard of significance at the 5 per cent point, and ignore entirely all results which fail to reach this level. A scientific fact should be regarded as experimentally established only if a properly designed experiment rarely fails to give this level of significance.

He produced a famous set of statistical tables, "Statistical Tables for Biological, Agricultural, and Medical Research". To save space, instead of giving comprehensive tables, as others had done before him, he gave critical values for which the P-value would equal a small set of values (such as 0.01 and 0.05 and maybe 0.10), which made it easy to see whether the P-value was less than 0.05. The tables in textbooks, to this day, follow this format.

So the answer to "why 0.05?" is "because that's what Fisher said". But with software like R available that can calculate the exact P-value for any test, the right thing to do is to quote the P-value you got, so that your readers can judge for themselves whether the evidence is sufficiently compelling and therefore whether they agree with your conclusion.

Most people would say that a P-value is 0.04 is evidence (though not strong evidence) with which to reject a null hypothesis. But it depends on the consequences of rejecting the null hypothesis. It might be that rejecting a null hypothesis means buying a bunch of expensive new machinery, in which case you'd want a very low P-value (0.04 might not be low enough), or you might be only tentatively believing the null hypothesis in the first place, in which case a P-value of 0.04 would be plenty small enough. It all depends on how serious it is to make an error.

3.5 Power and sample size

3.5.1 The `power.t.test` function

P-values, as we saw before, only depend on what happens *if the null hypothesis is true*. What if the null is not true? This is where we worry about how likely a type 2 error is, or more positively, the **power** of a test, which is the probability of *not* making a type 2 error: that is, the probability of correctly rejecting the null hypothesis when it is false.

To think about things a little more carefully: suppose the null hypothesis is way wrong (eg. null hypothesis says the population mean is 20, but it's actually 30). Then you are very likely to reject it, even if you only have a small sample, so the power will be large. On the other hand, if the null hypothesis is only a tiny bit wrong (null mean is 20, true mean is 20.01), you'll have a hard time rejecting the null hypothesis, even though it's wrong, so the power will be small. Also, the larger the sample size you have, the more easily you'll be able to reject the null hypothesis.

So power depends on two things: the sample size and how wrong the null hypothesis is. So you'll need to specify them both. Also, your best guess at the population standard deviation.

R has a function `power.t.test` that works out these things for you.

By way of example: let's go back to our one-sample t-test for the cars. We were testing a null hypothesis that the mean MPG for all cars was 20, against the alternative that it had changed. We had a sample of 38 cars, which had a sample SD of 6.5, so we'll take that as our best guess at the population SD. How likely are we to be able to conclude that the population mean MPG is not 20,

```
> pp=power.t.test(n=38,delta=25-20,sd=6.5,type="one.sample")
> pp

     One-sample t test power calculation

              n = 38
          delta = 5
             sd = 6.5
      sig.level = 0.05
          power = 0.9960424
    alternative = two.sided

> names(pp)

[1] "n"           "delta"       "sd"          "sig.level"   "power"
[6] "alternative" "note"        "method"
```

Figure 3.13: Power of t-test

when it is really 25, at $\alpha = 0.05$? We have to specify that this is a one-sample t test, since the default is for it to be a two-sample test. Figure 3.13 has the gory details.

The power is 0.9960424. We are pretty much assured of being able to correctly reject the null of the mean MPG being 20 if it's really 25.

Notice that there is no data in this calculation. The time to think about power is *before you collect any data*. You can answer questions like:

- what kind of departures from the null do I have a reasonable chance of detecting with my planned sample size?
- what kind of sample size do I need to have a reasonable chance of detecting a departure from the null of interest to me?

Let's take the second question first. You can feed `power.t.test` a desired power and leave the sample size unknown, and R will figure it out for you. What sample size do we need to have a 70% chance of correctly rejecting the null hypothesis that the population mean is 20, if the true mean is 21? Figure 3.14 has the answer.

The required sample size is 262.6963, which we round off to 263. In general, we round *up* always, because rounding down would give us not quite as large a power as we would like. This is how many cars we'd need to sample. If this exceeds our budget, we have to be aim lower!

```
> pp=power.t.test(delta=21-20,power=0.70,sd=6.5,type="one.sample")
> pp

     One-sample t test power calculation 

              n = 262.6963
          delta = 1
             sd = 6.5
      sig.level = 0.05
          power = 0.7
    alternative = two.sided
```

Figure 3.14: Sample size calculation

```
> mypower=function(true.mean)
+ {
+   pp=power.t.test(n=38,delta=true.mean-20,sd=6.5,type="one.sample")
+   pp$power
+ }
> mypower(25)

[1] 0.9960424
```

Figure 3.15: Function for computing power at a given true mean

Now let's think about the first question. We have a fixed sample size of 38, our null is that the population mean is 20, and the thing that varies is the true population mean; we want to find out the power for a number of different true population means. ("If the true population mean is this, how much power do I have?")

The first step is to write a function that accepts the true mean, computes the power for our situation, and returns it, as Figure 3.15. That seems to work. Now we make a vector of population means that we'd like to know power for, and compute the power for each one. This is a job for `sapply`. Figure 3.16 shows you how it works.

You can eyeball this, but it's nice to have these on a graph, so that you can see how the power increases as the true mean and null mean move farther apart. Such a graph is called a **power curve**. I'll put a smooth curve[1] on here as well to guide the eye, and a dashed horizontal line at power 1, since that's as high as power gets. `abline` is a handy command for drawing lines on graphs; you can use it to draw horizontal or vertical lines, or lines with a given intercept and

[1]If you want to plot a smooth trend on a scatterplot, use `lowess`, but if you want a smooth curve to go through some points, `spline` is the one to choose.

```
> means=seq(from=20,to=25,by=0.5)
> powers=sapply(means,mypower)
> cbind(means,powers)

      means     powers
 [1,]  20.0 0.02500000
 [2,]  20.5 0.06707312
 [3,]  21.0 0.15009258
 [4,]  21.5 0.28290838
 [5,]  22.0 0.45520936
 [6,]  22.5 0.63649367
 [7,]  23.0 0.79120643
 [8,]  23.5 0.89831818
 [9,]  24.0 0.95848228
[10,]  24.5 0.98590211
[11,]  25.0 0.99604239
```

Figure 3.16: Calculating power for each sample size

slope. `grid` draws a grid on the plot, for ease of reading off values. The default colour for `grid` is a very light grey that is hard to see, so I made it black. The final result of all that is shown in Figure 3.17.

So with our sample of size 38, we have a reasonable chance to correctly reject a null mean of 20 when the true population mean is about 23 or higher.

You can also do power curves as they depend on *sample size*, for a fixed true mean (or a fixed difference between the true mean and the hypothesized mean). A larger sample will give your test more power. If you do this for a departure from the null of interest to you, you can see what kind of sample size will give you the power that you want.

`power.t.test` also works for two-sample and matched pairs t tests. This is done by specifying `type="two.sample"` (or `one.sample` or `paired`). You can control the α that all the tests use (we used the default `alpha` of 0.05), by specifying for example `sig.level=0.01`, and if your alternative hypothesis is one-sided, you specify that with `alternative="one.sided"`. For a two-sample test, the sample size you feed in or get out is the size of *each* group (the groups are assumed to be the same size); likewise, for matched pairs `n` is the number of *pairs*.

```
> plot(means,powers)
> lines(spline(means,powers))
> abline(h=1,lty=2)
> grid(col="black")
```

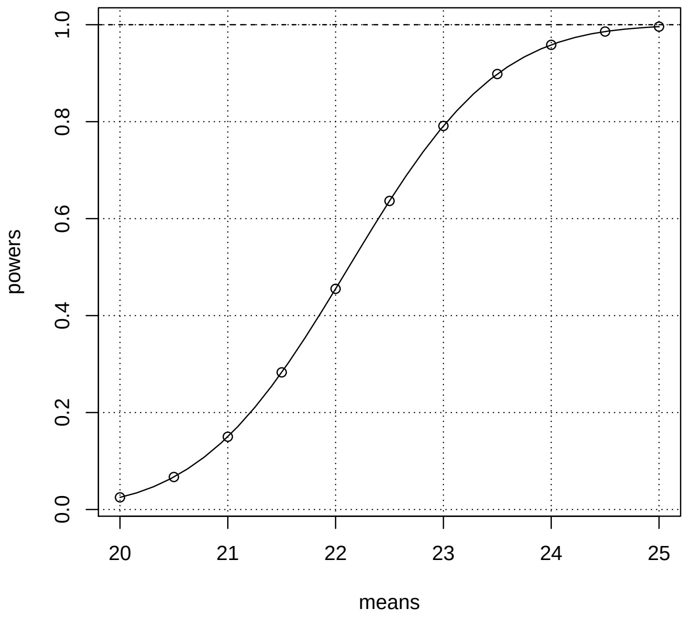


Figure 3.17: Plot of power curve

```
> set.seed(457299)
> xx=rnorm(38,23,6.5)
> t.test(xx,mu=20)

        One Sample t-test

data:  xx
t = 4.2354, df = 37, p-value = 0.000145
alternative hypothesis: true mean is not equal to 20
95 percent confidence interval:
 22.23882 26.34547
sample estimates:
mean of x
 24.29215
```

Figure 3.18: Drawing a random sample and running `t.test` on it

3.5.2 Power by simulation

`power.t.test` provides a kind of "magic" answer to "what is the power of my test?". But to understand why it works as it does, we can come at power another way: take a random sample whose mean is whatever we think the true mean is (say 23), and do a test that the population mean is, say, 20. Based on the P-value that comes from that test, we decide whether to (correctly) reject the null, or whether we don't, and make a type 2 error.

The first step is to decide what population we are sampling from. In our case, since we don't really know any better, let's make this a normal distribution, whose mean is 23 and whose SD is 6.5. (This is the same thing R does in calculating power exactly.) `rnorm` is the tool we want. It needs to be fed three things: how big the sample is (38), what the mean is (23) and what the SD is (6.5). Having drawn our random sample, we run `t.test` on it, as in Figure 3.18. I'm using `set.seed` so that my random number generation doesn't change from one compilation of this text to the next, and so the conclusions I draw don't change.

In this case, the sample mean came out to be 24, and the null value of 20 was resoundingly rejected.

Let's put this all into a function that generates a random sample and returns a decision: do we reject the null or not? Just for checking I'm going to print out the results of the t test, but not return them. See Figure 3.19.

Notice that nothing gets fed into `my.sample`, hence the pair of brackets with nothing between them. Make sure to include these brackets when you use the

```
> my.sample=function()
+ {
+   x=rnorm(38,23,6.5)
+   tt=t.test(x,mu=20)
+   print(tt)
+   tt$p.value<0.05
+ }
> my.sample()

        One Sample t-test

data:  x
t = 1.949, df = 37, p-value = 0.0589
alternative hypothesis: true mean is not equal to 20
95 percent confidence interval:
 19.92059 24.09129
sample estimates:
mean of x
 22.00594

[1] FALSE

> my.sample()

        One Sample t-test

data:  x
t = 2.0816, df = 37, p-value = 0.04435
alternative hypothesis: true mean is not equal to 20
95 percent confidence interval:
 20.06786 25.02653
sample estimates:
mean of x
  22.5472

[1] TRUE
```

Figure 3.19: Function to see whether we reject or not

```
> my.sample=function()
+ {
+   x=rnorm(38,23,6.5)
+   tt=t.test(x,mu=20)
+   tt$p.value<0.05
+ }
> res=replicate(100,my.sample())
> res

  [1] FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [13]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
 [25]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
 [37]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE
 [49]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [61]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE
 [73]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [85]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE
 [97]  TRUE  TRUE  TRUE  TRUE

> tt=table(res)
> tt

res
FALSE  TRUE
   17    83
```

Figure 3.20: Simulated power for *t*-test

function, or else things will get very confusing!

For my two tests, the first sample had a mean of 22, and the null hypothesis (that the mean was 20) was not quite rejected. In the second one, the sample mean was 22.5, and the mean of 20 was (just) rejected.

The nice thing about doing simulations in R is that you write a function to do your simulation once, and then you use `replicate` to do it as many times as you like, gluing the results together.

Let me tidy up my function by taking out the print statement, and then do 100 simulations. In how many of my simulated samples is a mean of 20 rejected? See Figure 3.20 for the answer.

The last line of my code counts up how many of my simulated samples rejected a mean of 20 (`TRUE`) or failed to reject it (`FALSE`). I had 83 rejections out of 100, for an estimated power of 0.83.

```
> res=replicate(10000,my.sample())
> tt=table(res)
> tt

res
FALSE  TRUE
 2084  7916
```

Figure 3.21: 10,000 samples

100 random samples isn't very many. 10,000 would be a lot better. That's easy enough to arrange. See Figure 3.21. This time I had 7916 rejections out of 10,000, so my estimated power is 0.7916. This is alarmingly close to the exact power, calculated above as 0.7912 for a true mean of 23. (I think I was lucky to get this close!)

3.6 Assessing normality

The *t* procedures were based on an assumption that the sample came from a normally-distributed population. That is, the distribution of values from which we have a sample is bell-shaped, with no outliers — indeed, the kind of thing for which using the mean and SD is sensible.

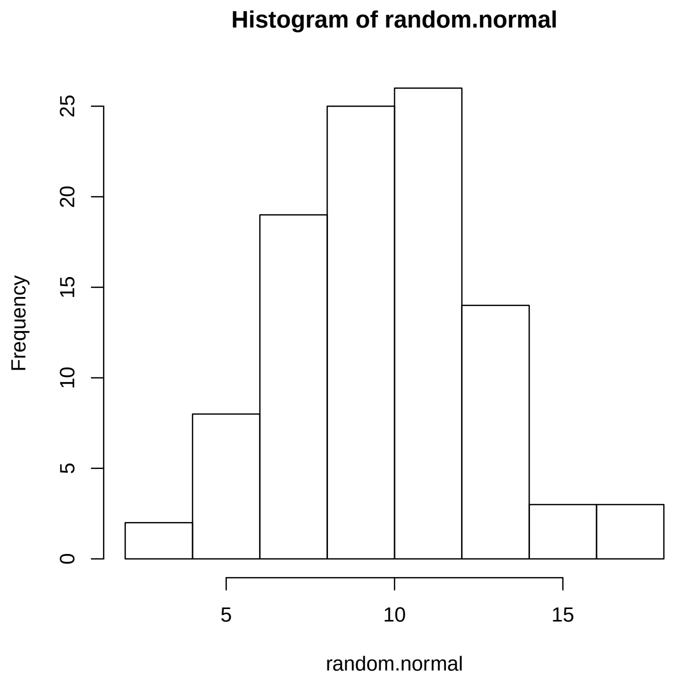
You might be thinking "what if the values in my sample have a skewed shape or outliers? What then? Well, the short answer is that the *t* procedures will often work surprisingly well. But to see whether we have cause to worry, we need to decide whether it is plausible that our data come from a normal distribution.

One way to do this is to make a histogram and see whether we have a sufficiently bell-like shape. But this is hard to judge. (How bell-like is bell-like?) A QQ plot makes the judgement easier to make, because you compare the plot to a straight line.

To see what to expect, let's first generate some random data from a normal distribution, then take a look at it, first with a histogram, and then with a QQ plot. `qqnorm` draws the plot, and `qqline` draws a line on it. See Figure 3.22. The histogram looks not too badly normal. We're looking for some random wiggles around the line, no systematic curves or anything. That's about what we have.

Now we'll try something skewed. The histogram in Figure 3.23 looks seriously skewed right. Below that, you see how it shows up on a QQ plot, which is big-time curved. The clear departure from the line says that a normal distribution is no good. Imagine the circles moved across onto the data (vertical) scale. At

```
> random.normal=rnorm(100,mean=10,sd=3)
> hist(random.normal)
```



```
> qqnorm(random.normal)
> qqline(random.normal)
```

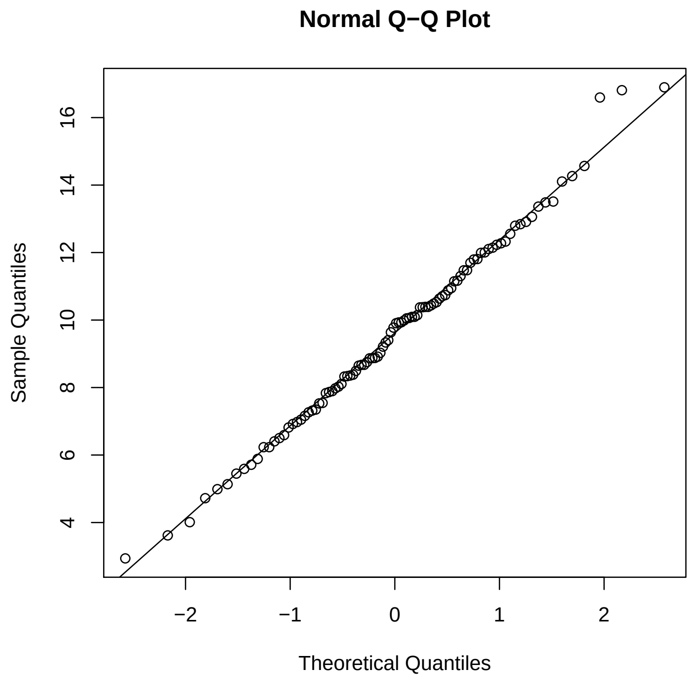
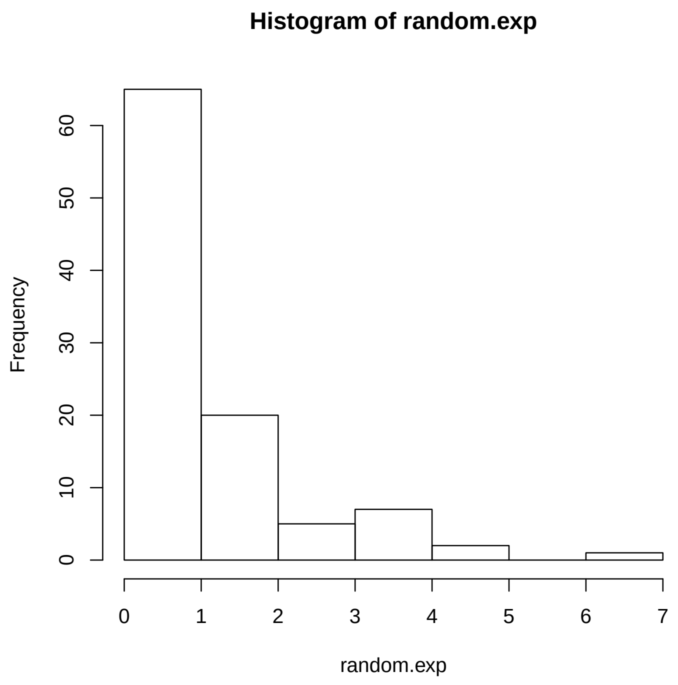


Figure 3.22: QQ plot for normal data

```
> random.exp=rexp(100,rate=1)
> hist(random.exp)
```



```
> qqnorm(random.exp)
> qqline(random.exp)
```

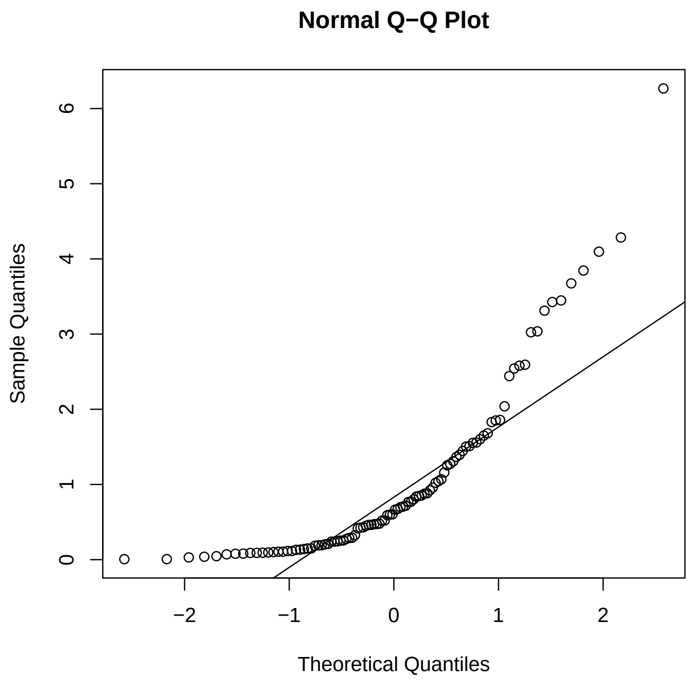


Figure 3.23: QQ plot for skewed data

```
> random.t=rt(100,df=2)
> qqnorm(random.t)
> qqline(random.t)
```

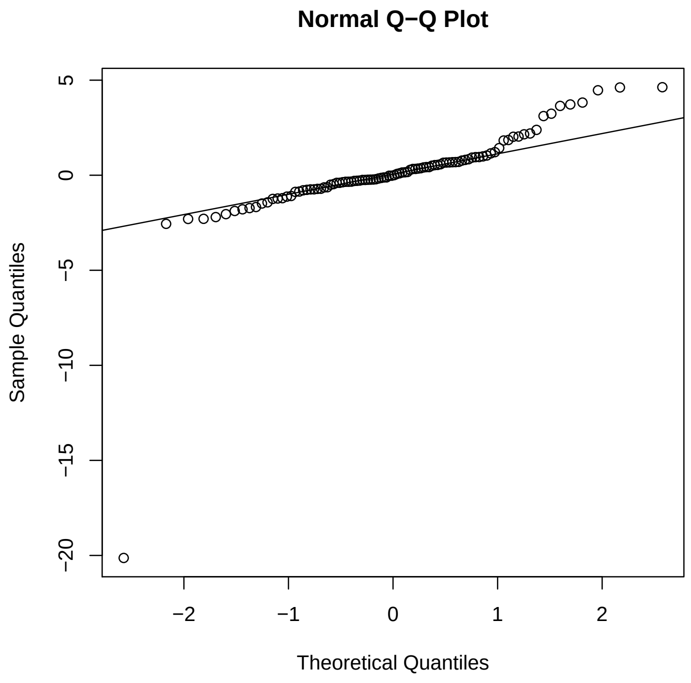


Figure 3.24: Another QQ plot

the low end, you have a lot of data bunched up together, while at the high end, the values are too spaced out to be normal. Spaced-out at the high end means that the long straggle is at the high end: skewed to the right.

A QQ plot that has a curve with data bunched up at the top end and spread out at the bottom end indicates a left-skewed distribution.

Figure 3.24 shows an odd one. Never mind about exactly *what* kind of data that first line is generating. Take a look at the plot. In the middle, the points follow the line pretty well. But at the ends, they drift away: down on the left, up on the right. Looking at the data (vertical) scale, the low values are *too* low for a normal and the high values are *too* high. The distribution is like a normal, but with more extreme (high and low) values than you'd expect.

Going back to our car data: are those MPG values normal? Figure 3.25 shows a curious S-bend shape. The data are not normal: both the high and low

```
> attach(cars)
> qqnorm(MPG)
> qqline(MPG)
```

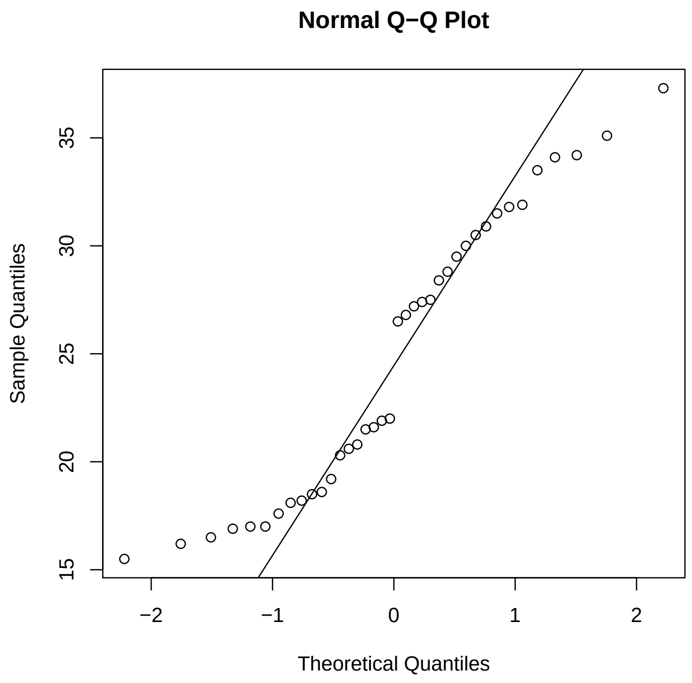


Figure 3.25: QQ plot for car MPG data

```
> mpg4=MPG[Cylinders==4]
> qqnorm(mpg4)
> qqline(mpg4)
```

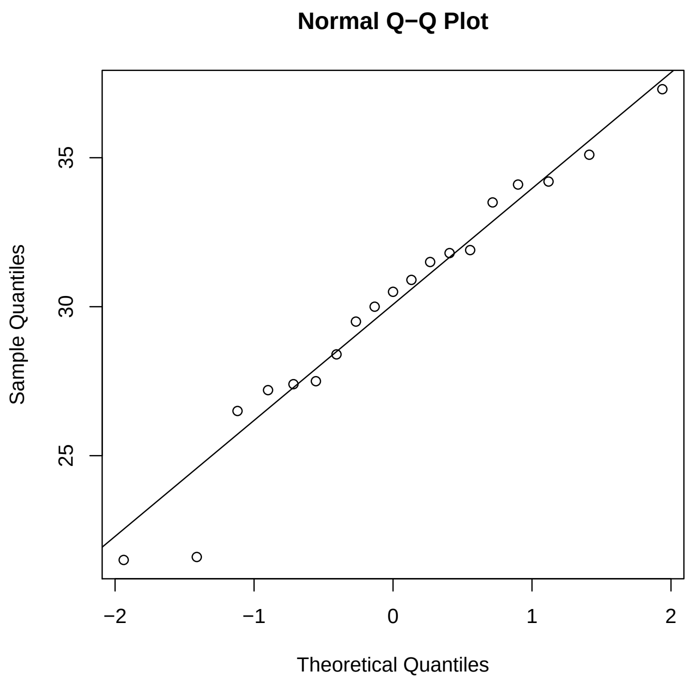


Figure 3.26: QQ plot for 4-cylinder engines

values are actually *less* extreme than the normal. Rather than agonizing over this, recall that our cars were a mixture of 4-, 6- and 8-cylinder engines, with different characteristics of fuel consumption.

So let's break things down by number of cylinders. First the 4-cylinder engines. Note, in Figure 3.26, how we select out the gas mileages for those cars with 4 cylinders. I'd say these are acceptably normal-looking.

There are only 8 8-cylinder engines; how do they look? Figure 3.27 tells the story. I'd say those are acceptably normal also, though with such a small amount of data it's hard to be sure. In other words, the MPG data are a *mixture* of normally-distributed stuff with different means (and probably different SDs also), and a mixture of normals isn't normal.

So now we know how to decide if things are normal. What to do if our data are not? First off, we might consider basing our inference on the *median* instead,

```
> mpg8=MPG[Cylinders==8]
> qqnorm(mpg8)
> qqline(mpg8)
> detach(cars)
```

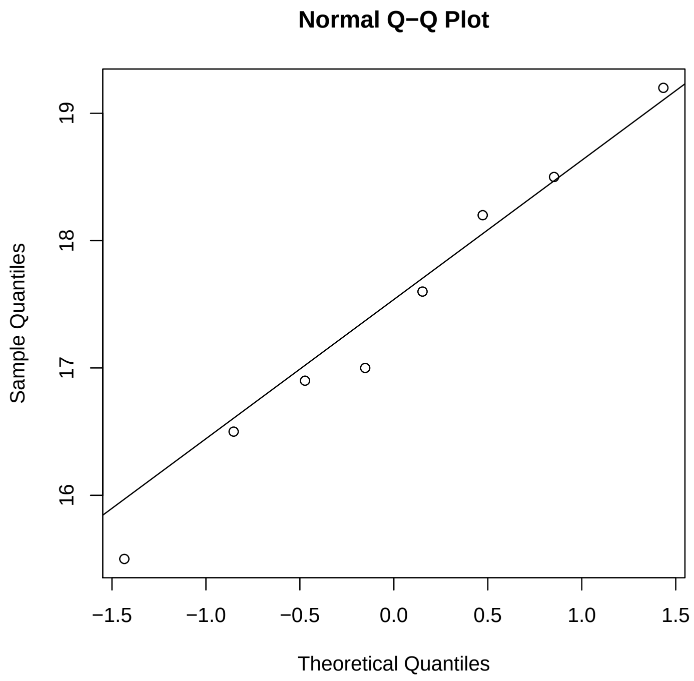


Figure 3.27: QQ plot for 8-cylinder engines

```
> pp=binom.test(389,1023)
> pp

        Exact binomial test

data:  389 and 1023
number of successes = 389, number of trials = 1023, p-value = 1.788e-14
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.3503987 0.4108009
sample estimates:
probability of success
             0.3802542
```

Figure 3.28: Confidence interval for population proportion

which is not affected by skewness or outliers. This is the domain of the *sign test*, for which see Section 3.10.

3.7 Inference for proportions

Opinion polls say things like this:

> 38% of Canadians would vote Liberal if there were a federal election tomorrow. 1,023 Canadians were sampled. The poll has a margin of error of 3 percentage points, 19 times out of 20.

What does that all mean?

The problem is that we cannot know about the political opinions of "all Canadians" without asking them. So we have to take a sample (the 1,023 Canadians mentioned above). If the sample is a simple random sample (or some other kind involving chance), the results can be very close to the truth, but the basic problem is, if you were to take another sample, you'd probably get different results. So the best we can do is to get a confidence interval for the proportion of all Canadians who would vote Liberal, which will express our uncertainty.

As for means, this can be done in two ways: via a confidence interval, and via a test. The confidence interval answers a question like "what is the population proportion of people who would vote Liberal", while the hypothesis test would answer something like "is the proportion of people who would vote Liberal in the entire population 50%?".

38% of 1023 is 389. We can get a confidence interval for the population proportion p like this. Note the similarity with `t.test`: you'll get a 95% confidence interval and a two-sided test unless you ask for something different: Figure 3.28 shows that the confidence interval goes from 0.35 to 0.411. So that's what we think the proportion of all Canadians who would vote Liberal is.

Our single best guess at the proportion of Liberal supporters is $389/1023 = 38\%$. Notice that the interval goes up and down 3 percentage points from that. This is the *margin of error*, and it reflects how uncertain we are about our estimation of the proportion of all Canadians who would vote Liberal.

Now, as in our estimation of a population mean, there's no way to give an interval that we are *certain* contains the population proportion. So we have to accept a certain chance of being wrong. Hence the "95" of the 95% interval, or, stating the same thing another way, the "19 times out of 20" in the poll.

Why funny numbers like "1,023" for the sample size? Well, the margin of error depends mostly on sample size (the bigger the sample, the smaller the margin of error). It depends on the actual proportion too, but only if that proportion is close to 0 or 1, which, for the kinds of things polls study, it usually isn't.

Let's investigate the effect of sample size on margin of error. First, we pretend that the 38% came from a sample of only 100 people, which would mean 38 Liberal supporters in the poll. This is the first line of Figure 3.29. This gives a margin of error of about 10 percentage points: not even close enough for Government work!

The second line of Figure 3.29 shows a sample of size 10,000 (3800 Liberal supporters). Now the margin of error is about 1 percentage point.

As you'd guess, the larger sample does give a more accurate result, but the issue is whether it is worth interviewing ten times as many people for the gain of going from 3 percentage points to 1. Pollsters don't generally think so; they think 3 percentage points is accurate enough, and so a sample size of about 1000 is generally used.

As with means, we can use a hypothesis test to see whether there is evidence that a population proportion is different from a null value. For example, suppose you toss a coin 70 times, and you get 45 heads (and 25 tails). Is that evidence that the coin is biased (that is, the proportion of heads in "all possible" tosses of that coin isn't 0.5)?

I admit that I made these data up.

To set this up: let p be the proportion of heads in all possible tosses of this coin. The null hypothesis is that $p = 0.5$ (coin fair), while the alternative is that $p \neq 0.5$ (biased). I think a two-sided alternative makes sense, since a bias either way would be worth knowing about. Let's also get a 90% confidence interval

```
> binom.test(38,100)

        Exact binomial test

data:  38 and 100
number of successes = 38, number of trials = 100, p-value = 0.02098
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.2847675 0.4825393
sample estimates:
probability of success
                  0.38

> binom.test(3800,10000)

        Exact binomial test

data:  3800 and 10000
number of successes = 3800, number of trials = 10000, p-value < 2.2e-16
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.3704727 0.3895972
sample estimates:
probability of success
                  0.38
```

Figure 3.29: Effect of sample size on margin of error

```
> pp=binom.test(45,70,p=0.5,conf.level=0.90)
> pp

        Exact binomial test

data:  45 and 70
number of successes = 45, number of trials = 70, p-value = 0.02246
alternative hypothesis: true probability of success is not equal to 0.5
90 percent confidence interval:
 0.5382159 0.7381169
sample estimates:
probability of success
             0.6428571
```

Figure 3.30: Test for proportion of heads

```
> binom.test(7,25,p=0.20,alternative="greater")

        Exact binomial test

data:  7 and 25
number of successes = 7, number of trials = 25, p-value = 0.22
alternative hypothesis: true probability of success is greater than 0.2
95 percent confidence interval:
 0.1394753 1.0000000
sample estimates:
probability of success
                  0.28
```

Figure 3.31: ESP testing

for p. Figure 3.30 shows the results.

The P-value here is 0.022, so this is evidence that the coin is biased (though not the strongest evidence). We can look at the confidence interval to assess how biased the coin is. The interval goes from 0.54 to 0.74, indicating that we know very little about how biased the coin might be. We need a bigger sample size to get a shorter confidence interval.

Making coins biased is actually very difficult, as least when you toss them. *Spinning* coins can be a different matter. If you take an old US penny (with the Lincoln Memorial on the “tails” side) and spin it, it will land tails up most of the time.

Dice, on the other hand, *can* be made biased. Adding a little extra weight to the 1-spot will make that face land downwards more often and make the 6-spot land upwards more often. Or you can just cheat. I have one die that has 5 on every side, and another that has 2 on three sides and 6 on the other three. Roll those dice together and you are guaranteed to get 7 or 11!

I just did an on-line ESP test, `http://www.psychicscience.org/esp3.aspx`, in fact. One way of testing for extra-sensory perception is via so-called Zener cards: this has an equal number of cards of each of five different symbols: circle, cross, wavy lines, square, star. You have to figure out what the next card will be. If you’re just guessing, you’ll get about 20% right, so $p = 0.20$ is the null, and $p > 0.20$ is the alternative hypothesis. I got 7 out of 25 right. Do I have ESP? What does Figure 3.31 say?

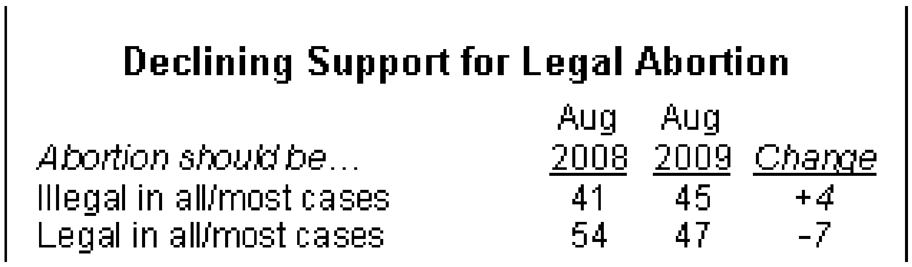
No way. I could easily get as many as 7 just by guessing. In fact, I *was* guessing.

3.8 Comparing several proportions

Instead of having one sample proportion that we are testing against some null value, we might have several proportions that we want to test against *one another*: are the population proportions all the same (null) or are any of them different (alternative)?

The most natural way of presenting data in this kind of context is as a *contingency table*. For example, I found a survey on Americans' attitudes towards abortion at different times, at `http://pewresearch.org/pubs/1361/support-for-abortion-slips`.

Let's look at this table:



Declining Support for Legal Abortion

Abortion should be...	Aug 2008	Aug 2009	*Change*
Illegal in all/most cases	41	45	+4
Legal in all/most cases	54	47	-7

These are percentages, but we want the actual counts for this. In the accompanying writeup, it says that 4,013 adults took part in the latest survey. Let's assume that 4,013 different adults took part in the 2008 survey. That means the actual numbers corresponding to those percentages were these, abbreviating the rows to "Abortion should be" plus the word shown:

```
            2008  2009
Illegal     1645  1809
Legal       2167  1886
Unknown      201   321
```

To ask whether views on abortion have changed over time, one way to proceed is to ask "Is there an association between opinion and year?". This brings us into the domain of the **chi-squared test for independence**. The null hypothesis is that there is no association between opinion and year (or your two categorical variables in general), and the alternative is that there is some association (form unspecified).

I copied that table into a file I called `pew1.txt` (you can use Notepad for this), and read it into a data frame as in Figure 3.32. I had to use `read.table` because this was just a text file with the values separated by whitespace. (Alternatively, you could type the values into a spreadsheet, save the result as a `.csv` file, then read them in using `read.csv`.) `read.table` in its simplest form requires the name of the file, and whether or not you have column headers (we do).

```
> pew1=read.table("pew1.txt", header=T)
> pew1

        X2008 X2009
Illegal  1645  1809
Legal    2167  1886
Unknown   201   321
```

Figure 3.32: Reading in the abortion data

```
> ct=chisq.test(pew1)
> ct

        Pearson's Chi-squared test

data:  pew1
X-squared = 54.8541, df = 2, p-value = 1.226e-12
```

Figure 3.33: Test for association

R treats the names of the columns as variables, and variables starting with a number are not allowed, hence the X glued onto the beginning of the years.

The test for association is shown in Figure 3.33. The P-value is 0.000000000001226251, which is extraordinarily small. We can reject the null hypothesis of no association, so there is definitely an association. Even though the change over the year looked small, there were enough people surveyed for it to be statistically significant.

When a hypothesis of no association is rejected, we usually want to find out where the association is. This can be done by comparing what we observed to what we would have expected under no association. The stuff returned to us from `chisq.test` includes the expected frequencies (these will be familiar to you if you have ever done this test by hand), and the "residuals", which, when far from zero, indicate where "no association" fails. Figure 3.34 shows those.

The top one is the actual data we had. The second table says, for example, that if there were no association (difference between years) we would have expected to see just over 2000 "legal"s both years. In the bottom table, look for the values farthest from zero. These are in the third row (indicating an increase in "don't know"s) and in the second (indicating that the proportion of "legal"s has decreased).

The interpretation of the residuals is that, compared to what you would have expected under no association, you saw more than that (positive residual) or

```
> ct$observed

        X2008 X2009
Illegal  1645  1809
Legal    2167  1886
Unknown   201   321

> ct$expected

            X2008     X2009
Illegal 1726.3547 1727.6453
Legal   2025.7428 2027.2572
Unknown  260.9025  261.0975

> ct$residuals

            X2008     X2009
Illegal -1.958023  1.957291
Legal    3.138473 -3.137301
Unknown -3.708563  3.707178
```

Figure 3.34: Observed and expected frequencies, with residuals

fewer than that (negative). This gives you an idea of why the chi-squared test for independence was significant.

The chi-squared test doesn't naturally produce a confidence interval, because there isn't usually anything to produce a confidence interval of. The one exception is when you are comparing exactly two proportions. That can be handled using `prop.test`, which is like `binom.test` that we used for one proportion. But when you have more than two proportions, a confidence interval doesn't make much sense because what are you comparing with what?

To make that fly with our data, we'll have to throw away the "don't knows", and also turn the table around so that the years are on the rows. This is done in the first four lines of Figure 3.35. The last line gets the interval.

Leaving aside those who didn't express an opinion, the proportion of people thinking abortion should be illegal was between 3.5 and 8 percentage points lower in 2008 than in 2009 (ie. it has gone up between the two times). For R, the first column of the transposed table, here `Illegal`, is counted as "success".

But when you have something other than 2 proportions being compared, you'll need to do the chi-squared test and look at the residuals.

There are other things on the website I cited, including classifications of how many people think "abortion is a critical issue" by year and political viewpoint.

```
> pew1

        X2008 X2009
Illegal  1645  1809
Legal    2167  1886
Unknown   201   321

> pu=pew1[1:2,]
> pu

        X2008 X2009
Illegal  1645  1809
Legal    2167  1886

> t(pu)

      Illegal Legal
X2008    1645  2167
X2009    1809  1886

> prop.test(t(pu))

        2-sample test for equality of proportions with continuity correction

data:  t(pu)
X-squared = 25.2186, df = 1, p-value = 5.119e-07
alternative hypothesis: two.sided
95 percent confidence interval:
 -0.08083175 -0.03526527
sample estimates:
   prop 1    prop 2
0.4315320 0.4895805
```

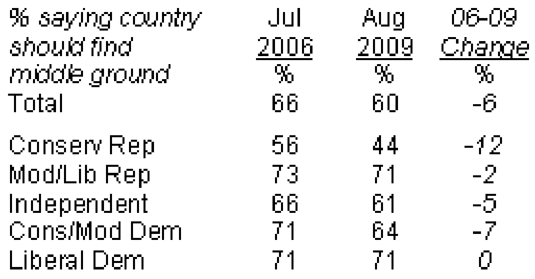
Figure 3.35: Confidence interval for difference in proportions

```
> pew2=read.table("pew2.txt",header=T)
> ct2=chisq.test(pew2)
> ct2

        Pearson's Chi-squared test

data:  pew2
X-squared = 121.0359, df = 4, p-value < 2.2e-16
```

Figure 3.36: Association between agreeing and political view

But this is a three-way table (year, viewpoint and agree/disagree), and an analysis of that takes us into the murky waters of log-linear analysis.

Let's do a partial analysis of this table:



% saying country should find middle ground	Jul 2006 %	Aug 2009 %	*06-09 Change* %
Total	66	60	-6
Conserv Rep	56	44	*-12*
Mod/Lib Rep	73	71	*-2*
Independent	66	61	*-5*
Cons/Mod Dem	71	64	*-7*
Liberal Dem	71	71	*0*

We'll just look at the 2009 survey and ask "is the proportion in favour the same for all five political groups?" Again, I need actual counts, so I'll assume those 4000 people are 1000 in each of the middle 3 groups and 500 in each of the extreme ones. (Just to show you that the totals don't all have to be the same.) That gives this table:

```
        Agree Disagree
ConRep    220      280
LibRep    710      290
Ind       610      390
ConDem    640      360
LibDem    355      145
```

I saved that into `pew2.txt`, and did the analysis of Figure 3.36. That P-value is *tiny*. There is definitely an association between political viewpoint and agreement with "the country should find a middle ground". What kind of association? Look at the residuals, Figure 3.37.

It's them Republicans! The ones on the right of the party are less likely to agree

```
> ct2$residuals

            Agree   Disagree
ConRep -5.4421152  7.1587615
LibRep  3.0288716 -3.9842908
Ind    -0.9434190  1.2410086
ConDem  0.2482682 -0.3265812
LibDem  2.1417356 -2.8173191
```

Figure 3.37: Residuals

```
> attach(cars)
> tbl=table(Cylinders,Country)
> tbl

         Country
Cylinders France Germany Italy Japan Sweden U.S.
        4      0       4     1     6      1    7
        5      0       1     0     0      0    0
        6      1       0     0     1      1    7
        8      0       0     0     0      0    8

> ct=chisq.test(tbl)
> ct

        Pearson's Chi-squared test

data:  tbl
X-squared = 22.2649, df = 15, p-value = 0.101
```

Figure 3.38: Constructing table and testing association for cylinders and country

and more likely to disagree than expected. Curiously, though, the Republicans on the left of that party are more likely to *agree* than expected. A look back at the original source puts them on a par with the most left-leaning Democrats, with the other categories rather in the middle.

Sometimes you don't have a table to begin with, and you have to construct it. For example, for our cars data, you might be interested in the association between the number of cylinders a car's engine has and which country it comes from. Remember the `table` command? Do that first, and feed its output into `chisq.test`, as in Figure 3.38. Even though there appears to be a strong association, the test does not give a significant result. This is because there isn't much data.

```
> ct$expected

         Country
Cylinders     France   Germany      Italy     Japan     Sweden       U.S.
        4 0.50000000 2.5000000 0.50000000 3.5000000 1.00000000 11.0000000
        5 0.02631579 0.1315789 0.02631579 0.1842105 0.05263158  0.5789474
        6 0.26315789 1.3157895 0.26315789 1.8421053 0.52631579  5.7894737
        8 0.21052632 1.0526316 0.21052632 1.4736842 0.42105263  4.6315789
```

Figure 3.39: Expected values for cylinders by country

```
> not5=cars[Cylinders!=5,]
> attach(not5)

The following object(s) are masked from 'cars':

    Car, Country, Cylinders, Horsepower, MPG, Weight

> isUS=(Country=="U.S.")
> tbl=table(Cylinders,isUS)
> detach(not5)
> tbl

         isUS
Cylinders FALSE TRUE
        4    12    7
        6     3    7
        8     0    8
```

Figure 3.40: Cylinders by country with categories combined

In fact, we might not even be able to trust that P-value we did get, because we should be suspicious of small expected values, especially ones smaller than 1. What do we have? Figure 3.39 says we have lots of trouble. When the expected frequencies are too small, we have to combine categories in some way.

Let's try this: we'll simply classify `Country` as US and "other", and we'll take out the 5-cylinder car, as in Figure 3.40. I began by creating a data frame `not5` that contains all the cars except the one that had 5 cylinders. Then I wanted to refer to the variables in `not5` rather than the ones in `cars`, hence that farting around with `detach` and `attach`. Then I created a new variable that determines whether a car is from the US or not. Finally I made a table and displayed it.

There is now a strongly significant association (Figure 3.41). We should check the expected frequencies, though. They are not too bad. Nothing smaller than

```
> ct=chisq.test(tbl)
> ct

        Pearson's Chi-squared test

data:  tbl
X-squared = 9.9475, df = 2, p-value = 0.006917

> ct$expected

         isUS
Cylinders    FALSE      TRUE
        4 7.702703 11.297297
        6 4.054054  5.945946
        8 3.243243  4.756757
```

Figure 3.41: Chi-squared test with categories combined

```
> ct$residuals

         isUS
Cylinders      FALSE       TRUE
        4  1.5483667 -1.2785218
        6 -0.5235017  0.4322673
        8 -1.8009007  1.4870448
```

Figure 3.42: Cylinders-country residuals

1, and nothing too much smaller than 5. (R will give you a warning if it thinks the expected frequencies are too small.)

So let's check the residuals to see why we found a significant association (as if we didn't know). Figure 3.42 says that there are fewer import 8-cylinder cars than we would have expected (actually, none at all) and more import 4-cylinder cars.

3.9 Connecting tests and confidence intervals

You might feel that tests and confidence inrervals cover mostly the same ground. A confidence interval says something like "what is my population mean", and a test says "could my population mean be 20?" These are really two sides of the same coin, so if you know something about the test result (P-value) you know something about the confidence interval, and vice versa.

Let's illustrate with the two-sample t-test comparing gas mileage of American and imported cars. We'll start with the test and default 95% CI, and then we'll get 90% and 99% CIs as well:

```
> attach(cars)


The following object(s) are masked from 'cars (position 3)':

    Car, Country, Cylinders, Horsepower, MPG, Weight


> t.test(MPG~is.american)


        Welch Two Sample t-test

data:  MPG by is.american
t = 2.0009, df = 30.748, p-value = 0.0543
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.08229979  8.46639070
sample estimates:
mean in group FALSE  mean in group TRUE
           27.18750            22.99545


> t.test(MPG~is.american,conf.level=0.90)$conf.int


[1] 0.6389167 7.7451742
attr(,"conf.level")
[1] 0.9


> t.test(MPG~is.american,conf.level=0.99)$conf.int


[1] -1.559980  9.944071
attr(,"conf.level")
[1] 0.99
```

In the second and third cases, I didn't need to see the test results again, so I just took the confidence interval out of the result.

In the first case, the P-value is just a bit bigger than 0.05, so we couldn't quite reject the null hypothesis that the mean MPG values were equal for the two populations. This is echoed by the confidence interval, where 0 is just inside the interval, so it is a "plausible" value for the difference in means.

The second interval, a 90% one, goes from 0.64 to 7.75. 0 is *not* inside that interval, so at this level 0 is *not* a plausible difference between the means.

The last interval, 99%, again *does* include zero. So at this level, the difference between the two means *could* be zero.

So what's the connection? You need to be doing a two-sided test (an alternative of "not equal"), because a confidence interval is a two-sided thing (it goes up and down from the sample mean). Then the connection is between the P-value and one minus the confidence level:

- If the P-value is greater than 0.01, the hypothesized mean will be inside the 99% confidence interval.
- If the P-value is greater than 0.05, the hypothesized mean will be inside the 95% confidence interval.
- If the P-value is greater than 0.10, the hypothesized mean will be inside the 90% confidence interval.

Otherwise the hypothesized mean will be inside the appropriate confidence interval.

In our example, the P-value was 0.0543, greater than 0.01 and 0.05, but less than 0.10. So 0 was inside the 99% and 95% confidence intervals, but outside the 90% one.

You might suspect that we could make a confidence interval for which 0 was right on the end. We can. To do that, work out $1 - 0.0543 = 0.9457$, and conclude that 0 would be right on the end of a 94.57% confidence interval. Were we right?

```
> tc=t.test(MPG~is.american,conf.level=0.9457)$conf.int
> tc

[1] 4.960726e-05 8.384041e+00
attr(,"conf.level")
[1] 0.9457

> detach(cars)
```

What do you think? (If you don't like the scientific notation, that interval goes from 0.00005 to 8.4.)

The correspondence between tests and confidence intervals is what mathematicians call "if and only if"; it works the other way around as well:

- If a value μ_0 is inside a 99% confidence interval for μ, then, for testing the null hypothesis $\mu = \mu_0$ against $\mu \neq \mu_0$, the P-value is greater than 0.01.
- If a value μ_0 is inside a 95% confidence interval for μ, then, for testing the null hypothesis $\mu = \mu_0$ against $\mu \neq \mu_0$, the P-value is greater than 0.05.
- If a value μ_0 is inside a 90% confidence interval for μ, then, for testing the null hypothesis $\mu = \mu_0$ against $\mu \neq \mu_0$, the P-value is greater than 0.10.

Checking this one out for our example, 0 is inside the 99% and 95% confidence intervals and outside the 90% one. So the P-value for testing that the mean difference is zero is greater than 0.01, greater than 0.05, but less than 0.10. As indeed it is.

The same ideas work for any confidence interval and test, whenever both of them make sense. For example, we were assessing the evidence for a coin being fair if we observed 45 heads in 70 tosses:

```
> pp=binom.test(45,70,p=0.5,conf.level=0.90)
> pp

        Exact binomial test

data:  45 and 70
number of successes = 45, number of trials = 70, p-value = 0.02246
alternative hypothesis: true probability of success is not equal to 0.5
90 percent confidence interval:
 0.5382159 0.7381169
sample estimates:
probability of success
             0.6428571
```

The P-value being less than 0.10 corresponds to 0.5 being *outside* the 90% confidence interval. Would 0.5 be inside a 99% confidence interval for the probability of heads? Try it and see:

```
> binom.test(45,70,p=0.5,conf.level=0.99)$conf.int

[1] 0.4824399 0.7833028
attr(,"conf.level")
[1] 0.99
```

It's inside. Which makes sense because the P-value, small though it is at 0.022, is not smaller than 0.01. However, the P-value is pretty close to 0.01, which is supported by 0.5 being pretty close to the end of the confidence interval.

3.10 The sign test

When you don't believe your data come from a normal distribution — in fact, the distribution appears to be sufficiently non-normal for you to be more interested in testing the median rather than the mean — you can use the sign test. This is more or less free of assumptions; as long as you have a simple random sample from your population, whatever it is, you're good to go.

Let's take our x from right back at the beginning:

```
> x=c(8,9,11,7,13,22,6,14,12)
```

and suppose for some reason we're interested in testing whether or not the population median is 7.5 (null hypothesis) against the alternative that it is not.

The logic is: if the population median really were 7.5, you'd expect to see about half the data values above 7.5 and about half below. So what you can do is relabel your data as "success" if the value is above 7.5 and "failure" if it's below. Then, you have a population of successes and failures, and you can test whether the proportion of successes in the population is a half, 0.5.

But to do *that*, you're just using a test for a proportion as in Section 3.7. The null hypothesis is that the proportion of successes is equal to 0.5 (against the alternative that it's not), and the data are how many "successes" (values above 7.5) you observed in your sample, and the number of values in your sample altogether.

Let's see if we can get R to do the heavy lifting:

```
> g=(x>7.5)
> x

[1]  8  9 11  7 13 22  6 14 12

> g

[1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE

> y=sum(g)
> y

[1] 7
```

```
> n=length(g)
> n

[1] 9
```

g is a logical vector (the logical statement `x>7.5` is either true or false for each value of `x`). Now, R counts each `TRUE` as 1 and each `FALSE` as 0, so if you add up the logical values in `g`, you'll get the total number of `TRUE`s, which here is the number of values greater than 7.5.

Now we feed `y` and `n` into `binom.test`, taking advantage of the fact that testing $p = 0.5$ is the default, so we don't need to say that:

```
> binom.test(y,n)


        Exact binomial test

data:  y and n
number of successes = 7, number of trials = 9, p-value = 0.1797
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.3999064 0.9718550
sample estimates:
probability of success
             0.7777778
```

The P-value 0.18 is not small, so there's no reason to reject the null hypothesis that the median is 7.5.

What if the hypothesized median were exactly equal to one of the data values? Then we'd throw that data value away, since it doesn't contribute to the test. For example, testing median equal to 7 goes like this, with a first step of getting rid of those values equal to 7:

```
> xx=x[x!=7]
> xx

[1]  8  9 11 13 22  6 14 12

> g=(xx>7)
> y=sum(g)
> y
```

```
[1] 7

> n=length(g)
> n

[1] 8

> binom.test(y,n)

        Exact binomial test

data:  y and n
number of successes = 7, number of trials = 8, p-value = 0.07031
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.4734903 0.9968403
sample estimates:
probability of success
                 0.875
```

`x!=7` means "`x` is not equal to 7", so `x[x!=7]` means "those values of `x` that are not equal to 7", which are the ones we want. Note that there are only 8 data values now, since we threw one away. The P-value is now 0.07, which is closer to significance than it was before.

Let's see if we can dress this up as a function. We need to feed in two things: the data and the hypothesized median, and all we need to get back is the P-value (since the other output from `binom.test` doesn't help us much):

```
> sign.test=function(mydata,median)
+   {
+     xx=mydata[mydata!=median]
+     gg=(xx>median)
+     y=sum(gg)
+     n=length(gg)
+     bt=binom.test(y,n)
+     bt$p.value
+   }
```

Let's go through that.

- The `function` line says that what comes in to the function is called `mydata` and `median` as far as the function is concerned, no matter what it was called outside.

- First line below the curly bracket defines `xx` as the values in `mydata` that are not equal to the hypothesized median.
- Then we make a vector of `TRUE` and `FALSE` according to whether the values in `xx` are greater than the fed-in median or not. (They can't be equal, since we got rid of those).
- Then we count how many "successes" we had and how many "trials", and feed them into `binom.test`, saving the results.
- Finally, we extract the P-value and return it, so that if we run the function, we'll see just that.

Let's test it out:

```
> sign.test(x,7.5)

[1] 0.1796875

> sign.test(x,7)

[1] 0.0703125
```

Did it work? Yep.

Now, how do we get a confidence interval for the population median? What we have to do is to use that idea from Section 3.9, which says here that:

> to get the confidence interval for the median, find all the values for the hypothesized median that are not rejected in a two-sided hypothesis test with the corresponding α. (That is, to get a 95% confidence interval, reject when P-value less than 0.05; to get a 90% interval, reject when P-value less than 0.10.)

So what we can do is to make a list of values to hypothesize for the median (we no longer have a null hypothesis to guide us), and then run our `sign.test` function on the same data for each of those hypothesized medians.

Our data values go from 6 to 22, so let's go from 5 to 23 in steps of 0.5, which is what the first line below does. Then we have to go through our hypothesized medians and run the sign test with our data on each one. I'll do it with a loop this time, partly for the variety, and partly because I realize that I didn't structure the `sign.test` function properly to use `sapply` without jumping through additional hoops:

```
> hmed=seq(from=5,to=23,by=0.5)
> for (i in hmed)
+   {
+     p=sign.test(x,i)
+     print(c(i,p))
+   }
```

```
[1] 5.00000000 0.00390625
[1] 5.50000000 0.00390625
[1] 6.0000000 0.0078125
[1] 6.5000000 0.0390625
[1] 7.0000000 0.0703125
[1] 7.5000000 0.1796875
[1] 8.0000000 0.2890625
[1] 8.5000000 0.5078125
[1] 9.0000000 0.7265625
[1] 9.5 1.0
[1] 10  1
[1] 10.5  1.0
[1] 11  1
[1] 11.5  1.0
[1] 12.0000000  0.7265625
[1] 12.5000000  0.5078125
[1] 13.0000000  0.2890625
[1] 13.5000000  0.1796875
[1] 14.0000000  0.0703125
[1] 14.5000000  0.0390625
[1] 15.0000000  0.0390625
[1] 15.5000000  0.0390625
[1] 16.0000000  0.0390625
[1] 16.5000000  0.0390625
[1] 17.0000000  0.0390625
[1] 17.5000000  0.0390625
[1] 18.0000000  0.0390625
[1] 18.5000000  0.0390625
[1] 19.0000000  0.0390625
[1] 19.5000000  0.0390625
[1] 20.0000000  0.0390625
[1] 20.5000000  0.0390625
[1] 21.0000000  0.0390625
[1] 21.5000000  0.0390625
[1] 22.0000000  0.0078125
[1] 22.50000000  0.00390625
[1] 23.00000000  0.00390625
```

(A technicality: I can't just name a variable inside a loop to print it; I have to use the `print` function explicitly.)

All right, what do we have?

Let's start with a 95% confidence interval. We want all the values of the median for which the P-value is greater than 0.05. Notice that the P-values (second column of the output) go up and then down, so there'll be two places where they pass 0.05. Those are: between 6.5 and 7.0, and between 14.0 and 14.5. So 7.0 and 14.0 are inside the confidence interval, and 6.5 and 14.5 are outside. Hence the 95% CI for the median goes from 7.0 to 14.0.

Similarly, we can find the 90% confidence interval for the population median, which is all those values for the median where the P-value is bigger than 0.10: from 7.5 to 13.5.

The 90% interval is a smidgen narrower than the 95% one, as it should be. The reader can verify (I hope) that the 99% confidence interval goes from 6.5 all the way up to 21.5.

All of these confidence intervals are pretty wide, for a couple of reasons:

1. We don't have much data: only 9 observations, and sometimes we have to throw one of them away.
2. The sign test makes rather wasteful use of the data: only whether each value is above or below the hypothesized median. Contrast that with a t-test: the exact value of each observation is used in calculating the sample mean and sample SD. *But*, when your population isn't normal, maybe using all the data values is not what you want to do.

In case you care, here is the easiest way I could find to do it with `sapply`:

```
> st2=function(m,x)
+   {
+     sign.test(x,m)
+   }
> pvals=sapply(hmed,st2,x)
> cbind(hmed,pvals)
```

```
      hmed      pvals
 [1,]  5.0 0.00390625
 [2,]  5.5 0.00390625
 [3,]  6.0 0.00781250
 [4,]  6.5 0.03906250
 [5,]  7.0 0.07031250
 [6,]  7.5 0.17968750
 [7,]  8.0 0.28906250
 [8,]  8.5 0.50781250
 [9,]  9.0 0.72656250
[10,]  9.5 1.00000000
[11,] 10.0 1.00000000
[12,] 10.5 1.00000000
[13,] 11.0 1.00000000
[14,] 11.5 1.00000000
[15,] 12.0 0.72656250
[16,] 12.5 0.50781250
[17,] 13.0 0.28906250
[18,] 13.5 0.17968750
[19,] 14.0 0.07031250
[20,] 14.5 0.03906250
[21,] 15.0 0.03906250
[22,] 15.5 0.03906250
[23,] 16.0 0.03906250
[24,] 16.5 0.03906250
[25,] 17.0 0.03906250
[26,] 17.5 0.03906250
[27,] 18.0 0.03906250
[28,] 18.5 0.03906250
[29,] 19.0 0.03906250
[30,] 19.5 0.03906250
[31,] 20.0 0.03906250
[32,] 20.5 0.03906250
[33,] 21.0 0.03906250
[34,] 21.5 0.03906250
[35,] 22.0 0.00781250
[36,] 22.5 0.00390625
[37,] 23.0 0.00390625
```

First I had to define a new function with the hypothesized median and data `x` the *other way around*, so that it fits the template for `sapply`. Then I also had to pass the data `x` into `sapply`. Then `cbind` prints out two vectors (the hypothesized medians and P-values returned from `sapply`) side by side.

Chapter 4

Regression

4.1 Introduction

Regression and ANOVA have the common goal that you have a numerical **response** or outcome variable, and one or more other things that might make a difference to the outcome. Then you want to find out whether those other things really do make a difference to the outcome.

Some examples, going back to the cars data and taking `MPG` as our response:

- Does MPG depend on the weight of the car? (Simple regression.)
- Does MPG depend on which country the car comes from? (One-way ANOVA)
- Does MPG depend on the weight and/or the horsepower of the car? (Multiple regression.)
- Does MPG depend on country and/or the number of cylinders? As long as we treat number of cylinders as a categorical variable dividing the cars into groups, this is a two-way ANOVA.
- Does MPG depend on country and/or the horsepower of the engine? (Analysis of covariance.)

The distinction is this: if you have all quantitative explanatory variables, it's a regression; if you have all categorical explanatory variables, it's an ANOVA; if you have a mixture, it's an analysis of covariance.

In some ways, this is an artificial distinction, because these are all "linear models", and in R, `lm` can be used to fit them all and test them for significance.

But they come from different historical places, and it is often useful to treat the methods in different ways.

You might have encountered ANOVA as a way of testing for differences between groups. This is true, but I never really understood what it was doing until I realized that we were really looking to see whether knowing what group an observation was in helped you predict what its response variable might be.

4.2 Simple regression

Simple regression is not especially simple; it's just called that because you have only one explanatory variable. For example, you might be predicting MPG from weight.

The first step in a proposed regression analysis is to draw a picture. The appropriate picture is a scatterplot, which is just a xy-plot of the data, like Figure 4.1.

The first variable you give is the explanatory, or x, variable. The second one is the response, or y, variable. (If you just want to assess the relationship between two variables without thinking of one of them as a response, you can plot them either way around.)

So now we look at this plot. What do we see? When the weight is smaller, on the left, the gas mileage is usually higher, but when the weight is larger, the MPG is usually smaller. It's not a perfect relationship — it usually isn't — but there seems to be enough of a trend to be interesting.

Next, a couple of things you might want to add to the plot. These get added to the "current plot", the last `plot` statement you issued. The first is called a "scatterplot smoother", and gives you a sense of what kind of trend there is, as shown in Figure 4.2. This looks like a straight line trend, at least up until a weight of 3 tons or so, when it seems to level off a bit.

Another thing you might want to plot is the points with a different symbol according to another variable, such as `Cylinders`. R has a list of characters that it uses for plotting (see the help for `points`). These are characters 4, 6 and 8 in the list. See Figure 4.3. I also added a legend, which involved a little jiggery-pokery to pick out the unique different values for `Cylinders` and the points to plot with them.

Another way, plotting the actual symbols 4, 6, and 8, is as in Figure 4.4. Note the two-step process where we first draw the plot with nothing on it, and then we use `text` to put the right thing in the right places.

Note, in both cases, that the 4-cylinder cars are top and left (low weight, good MPG) while the 8-cylinder cars are bottom and right (high weight, bad MPG).

```
> cars=read.csv("cars.csv",header=T)
> attach(cars)

The following object(s) are masked from 'cars (position 3)':

    Car, Country, Cylinders, Horsepower, MPG, Weight

> plot(Weight,MPG)
```

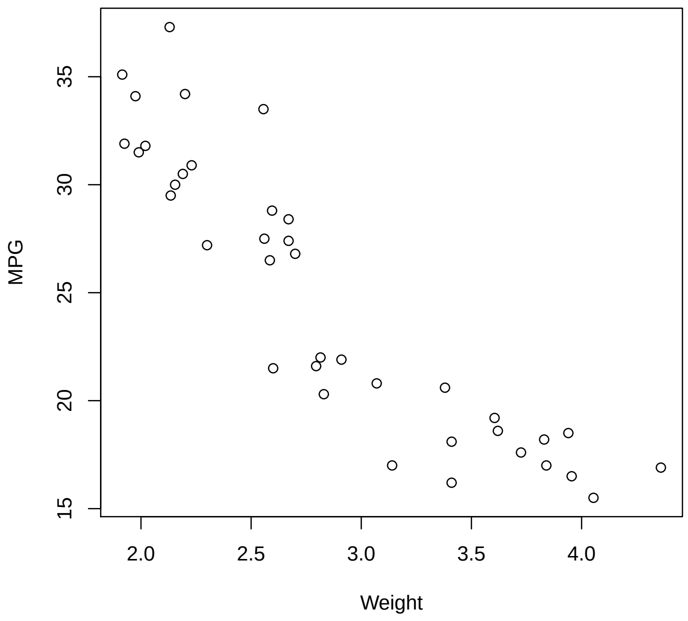


Figure 4.1: Scatterplot of weight and MPG

```
> plot(Weight,MPG)
> lines(lowess(Weight,MPG))
```

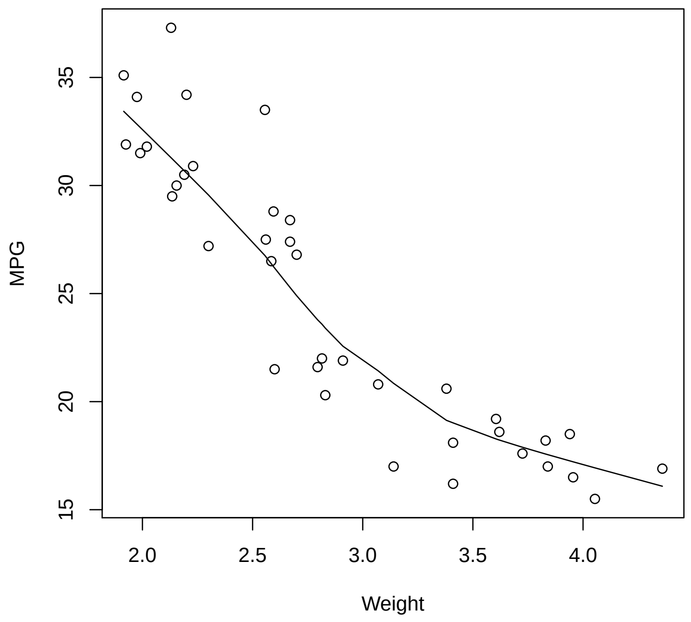


Figure 4.2: Scatterplot with lowess curve

```
> plot(Weight,MPG,pch=Cylinders)
> legend("topright",legend=unique(Cylinders),pch=unique(Cylinders))
```

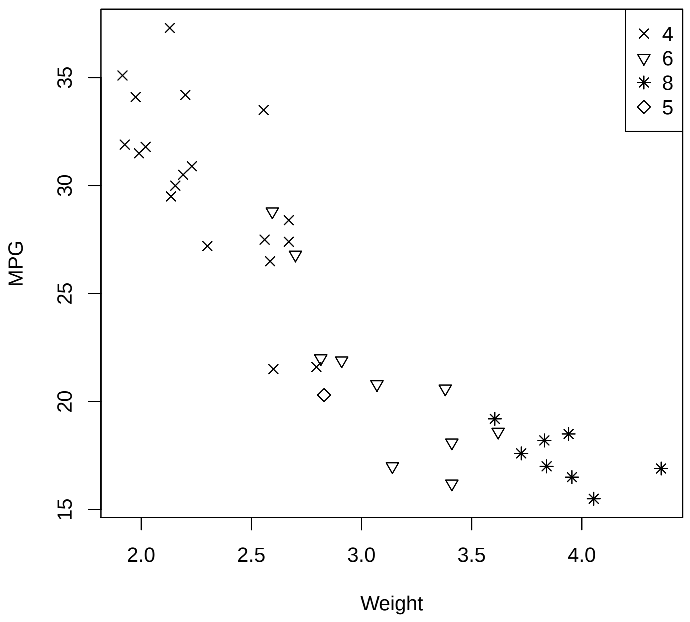


Figure 4.3: Scatterplot with points identified by cylinders

```
> plot(Weight,MPG,type="n")
> text(Weight,MPG,labels=Cylinders)
```

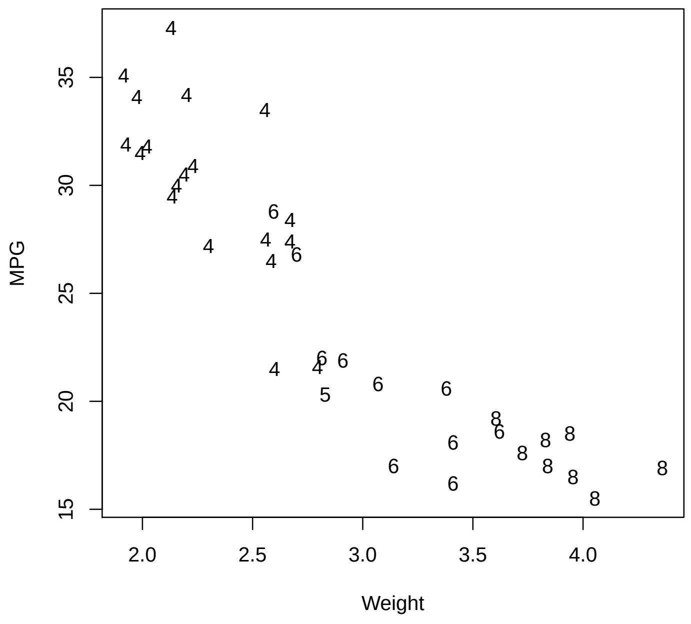


Figure 4.4: Plotting actual numbers of cylinders

```
> plot(Weight,MPG,type="n")
> text(Weight,MPG,labels=Country)
```

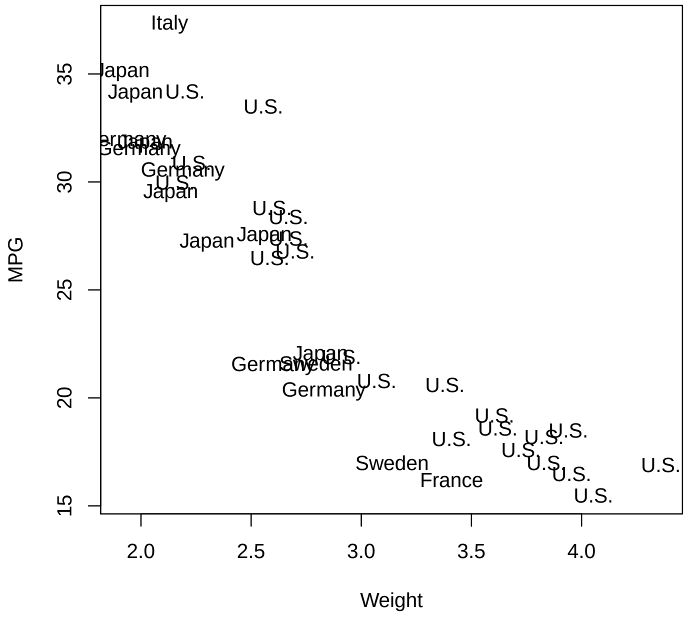


Figure 4.5: Using countries as labels

The latter plotting approach works best when you have single-character values to plot. Figure 4.5 shows what happens if you try to plot by country. The labels for `Country` overwrite each other, making a messy-looking plot. (The actual points are at the *centre* of the country names.)

A couple of fixes: make the country names smaller, like Figure 4.6. `cex` stands for "character expansion".

Or we could abbreviate the country names. There is a handy command `abbreviate` that produces guaranteed-unique abbreviations of at least a given length, as shown in Figure 4.7. The first two lines demonstrate how `abbreviate` works by testing it on a list of Greek letters.

Finally, we could label each point by which car it represents. This time we want to actually plot the points, and put the labels not on top of the points, but (say) to the right of them, which is what `pos=4` does, using small characters so the labels don't overlap too much. See Figure 4.8.

```
> plot(Weight,MPG,type="n")
> text(Weight,MPG,labels=Country,cex=0.5)
```

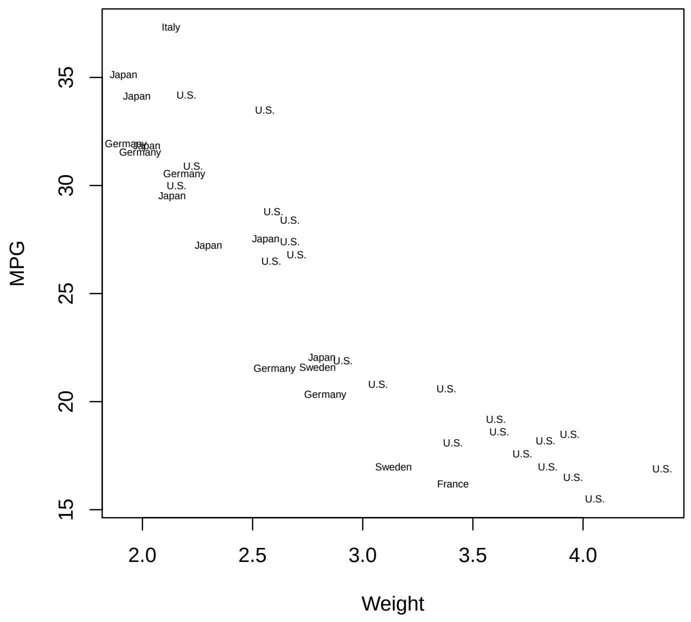


Figure 4.6: Countries as smaller labels

```
> greeks=c("alpha","beta","epsilon","eta","eta","epsilon")
> abbreviate(greeks,1)

  alpha    beta epsilon     eta     eta epsilon
    "a"     "b"    "ep"    "et"    "et"    "ep"

> country1=abbreviate(Country,1)
> country1

   U.S.    U.S.    U.S.   Italy  France Germany    U.S.   Japan    U.S. Germany
    "U"     "U"     "U"     "I"     "F"     "G"     "U"     "J"     "U"     "G"
   U.S.    U.S. Germany    U.S.   Japan   Japan Germany    U.S.    U.S.    U.S.
    "U"     "U"     "G"     "U"     "J"     "J"     "G"     "U"     "U"     "U"
  Japan    U.S.    U.S.  Sweden   Japan    U.S.  Sweden    U.S.   Japan    U.S.
    "J"     "U"     "U"     "S"     "J"     "U"     "S"     "U"     "J"     "U"
   U.S.    U.S.    U.S. Germany    U.S.   Japan    U.S.    U.S.
    "U"     "U"     "U"     "G"     "U"     "J"     "U"     "U"

> plot(Weight,MPG,type="n")
> text(Weight,MPG,labels=country1)
```

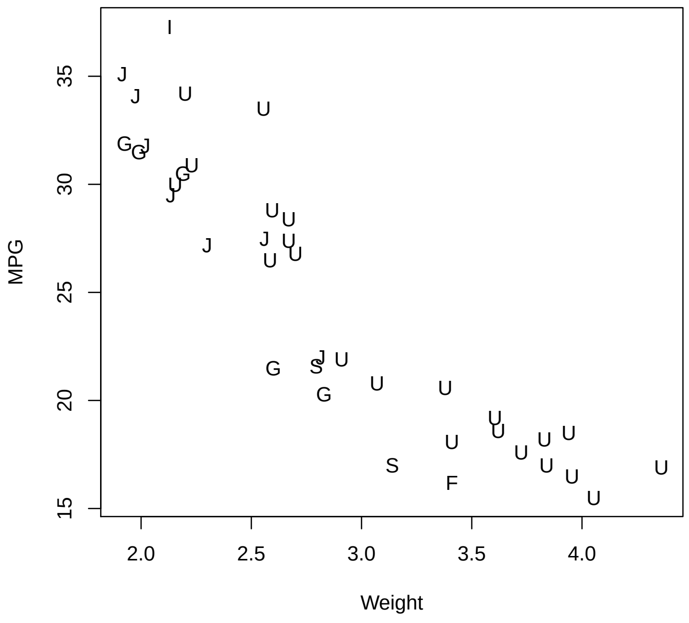


Figure 4.7: Abbreviated country names

```
> plot(Weight,MPG)
> text(Weight,MPG,Car,pos=4,cex=0.5)
```

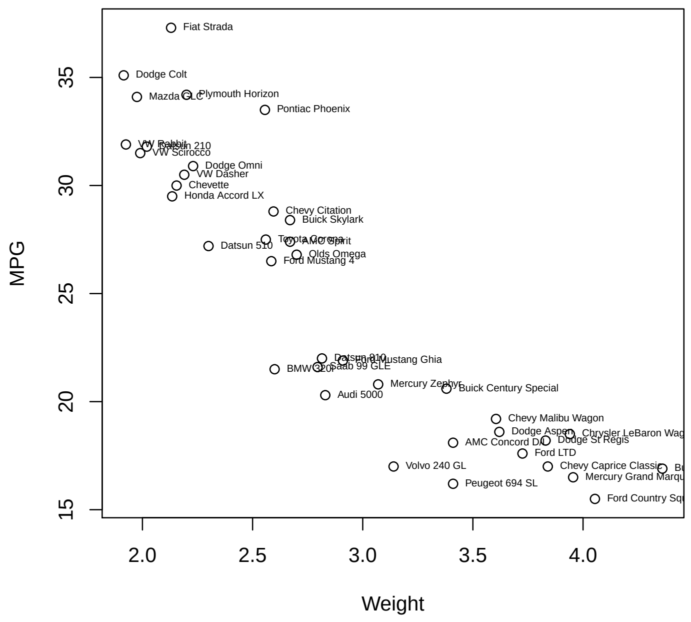


Figure 4.8: Scatterplot with cars labelled

```
> cor(Weight,MPG)

[1] -0.9030708

> cars.numeric=cars[,2:5]
> cor(cars.numeric)

                  MPG     Weight  Cylinders Horsepower
MPG         1.0000000 -0.9030708 -0.8055110 -0.8712821
Weight     -0.9030708  1.0000000  0.9166777  0.9172204
Cylinders  -0.8055110  0.9166777  1.0000000  0.8638473
Horsepower -0.8712821  0.9172204  0.8638473  1.0000000
```

Figure 4.9: Correlation matrix for cars

Another thing you might want to do is to put the best straight line on the scatter plot, but we'll get to that in a minute.

4.2.1 Correlation

Enough pictures for a moment. One of the numerical ways you can summarize a scatter plot is by a **correlation**. This can take two or more variables, or even a data frame. (It's an error to ask for the correlation of a data frame that contains any categorical variables, so I created one with just the quantitative ones, in columns 2 through 5.)

The correlation is a number between -1 and 1. -1 means a perfect straight line going downhill, 1 means a perfect straight line going uphill, and 0 means no straight-line association at all. In our case, we got -0.903, which indicates a pretty strong and straight relationship going downhill. This is what we saw on the scatter plot. If the association were stronger or straighter, the correlation would be closer to -1.

When you feed `cor` more than two variables, you get a *correlation matrix*, showing the correlation between the variable in the row and the variable in the column. Our correlation matrix is in Figure 4.9. You can see the -0.903 again in the first row (of numbers) and second column, and also in the second row and first column. The correlation between a variable and itself is always 1. The correlations here are all pretty close to 1 or -1; for example, a car with more cylinders is likely to be heavier (correlation 0.917) and to have more horsepower (0.864). Since high on MPG goes with low on everything else, all the correlations with MPG are negative.

So the process is:

1. Look at a **scatterplot** to decide whether you have any kind of relationship, and if so, what kind of relationship you have.
2. Once you've decided a straight line describes what's going on, the **correlation** summarizes how strong the relationship is.
3. To find out *which* straight line describes the relationship, you need **regression** (coming up below).

4.2.2 Regression preliminaries

There are only two things that you need to describe a straight line: where it starts from (called the **intercept**) and how fast it goes up or down (called the **slope**). There's nothing else to it. If you're trying to describe a curve, you have to capture its curviness somehow, but a straight line just keeps on going the same for ever, so once you have the intercept and slope, you are done.

So *which* intercept and slope do we need? There is a principle called **least squares** that produces good lines when you don't have any points too far from the trend. (If you *do*, there can be weirdness, but we won't worry about that for now.) The R function `lm` fits least squares regressions, and actually analyses of variance as well.

4.2.3 Fitting a regression line

The starting point for `lm` is a "model formula". We saw this when we were drawing side-by-side boxplots and wanted to divide our data up into groups. The response variable (the one you're trying to predict) goes on the left side, then a , the the explanatory variable (the one you're predicting the response from) on the right.

`lm` calculates a lot of stuff that you might need later, so it's a good idea to save the output in a variable, and then you can look at what you want to look at a bit at a time. Let's predict `MPG` from `Weight`. Off we go, Figure 4.10.

Printing the output from `lm` just gives you what the regression was of (next to "Call") and the estimated intercept and slope. Here, the intercept says that a car that weighs nothing would go 48.7 MPG (which doesn't make much sense), and the slope says that for each additional ton of weight, the MPG decreases by 8.4 miles per gallon on average (which *does* make sense).

That's all fine and well, but if we are going to be good statisticians, we want to know at least a couple more things: how confident are we in our estimate of the slope, and is the trend reproducible (thinking of our cars as a random sample from all possible cars)? That's what `summary` (also in Figure 4.10) shows us.

```
> ans=lm(MPG~Weight)
> ans

Call:
lm(formula = MPG ~ Weight)

Coefficients:
(Intercept)       Weight
     48.707       -8.365

> summary(ans)

Call:
lm(formula = MPG ~ Weight)

Residuals:
    Min      1Q  Median      3Q     Max
-5.4595 -1.9004  0.1686  1.4032  6.4091

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   48.708      1.954   24.93  < 2e-16 ***
Weight        -8.365      0.663  -12.62 8.89e-15 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.851 on 36 degrees of freedom
Multiple R-squared: 0.8155,        Adjusted R-squared: 0.8104
F-statistic: 159.2 on 1 and 36 DF,  p-value: 8.889e-15

> confint(ans)

                2.5 %    97.5 %
(Intercept) 44.745245 52.669745
Weight      -9.709267 -7.019932
```

Figure 4.10: Regression for cars

The accuracy of the slope is summarized by its standard error, here 0.663 (look in Coefficients). The `confint` command gets confidence intervals for parameters (intercept and slope(s)) in a regression. This is shown in Figure 4.10 as well. By default it gives you 95% intervals (for all the parameters); use `level=` or investigate `parm=` if you want something else. Here, we are pretty sure that the slope is between -7 and -10.

If the slope were zero, there would be no dependence of `MPG` on `Weight` (since the predicted MPG would be the same all the way along). R tests whether the slope is zero: in the Coefficients table, look along to the end of the Weight line. The last number is the P-value for testing that the slope is zero. Here, it is very small indeed, indicating that we have way more of a trend than we'd expect by chance. The three stars on the end of the line indicate a P-value less than 0.001. This is something that should be reproducible.

The F test for a regression (at the bottom of the `summary` output) tells us whether none of the explanatory variables have any effect on the response (the null) or whether one or more of them do (the alternative). Since we only have one explanatory variable here, this tells us the same as the test for the slope. The P-value is identical.

When you are doing a multiple regression with more than one x-variable (coming up), this is your starting point, answering the initial question of "is *anything* helping?".

4.2.4 Quality control

How do we know that the regression line we fitted makes any sense? Looking at the scatterplot is a start, but sometimes things are a little harder to see. Here's an example where the right picture makes things obvious.

If you eyeball Figure 4.11, you'll see that y goes up by steps of 2 until halfway along, and then it goes up by steps of 1. In other words, the rate of increase isn't constant, so a straight line shouldn't work here.

But the regression output looks pretty good, as shown in Figure 4.12. The slope is accurately estimated, and is significantly different from zero. The "multiple R-squared" is the square of the correlation between x and y, 0.9694, so the actual correlation must be over 0.98. What's not to like?

Let's have a look at the data, the "fitted values" or y-values predicted from the line, and the "residuals", the differences between what we observed and what we predicted. This is the first line in Figure 4.13. The first two and last two predicted values are too big, and the ones in the middle are too small, with the prediction for $x = 4$ being the worst of all. Plotting the fitted values against the residuals, as in Figure 4.13, is the standard way of looking at things.

```
> x=1:8
> y=c(3,5,7,9,10,11,12,13)
> plot(x,y)
```

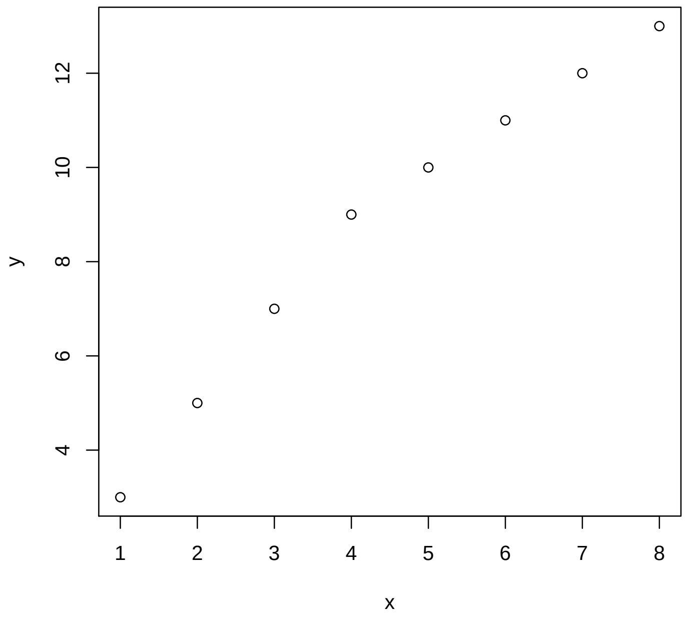


Figure 4.11: Is this straight?

```
> lmxy=lm(y~x)
> summary(lmxy)

Call:
lm(formula = y ~ x)

Residuals:
     Min       1Q   Median       3Q      Max 
-0.83333 -0.36310 -0.04762  0.40476  0.95238 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)   2.4286     0.5148   4.717  0.00327 ** 
x             1.4048     0.1019  13.779 9.09e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1 

Residual standard error: 0.6607 on 6 degrees of freedom
Multiple R-squared: 0.9694,	Adjusted R-squared: 0.9643 
F-statistic: 189.9 on 1 and 6 DF,  p-value: 9.087e-06
```

Figure 4.12: Regression of non-linear data

The clear pattern, up and down, of the residuals means that you can predict how far off the line you are, and which way, from how far along you are. This is bad news, because if something predictable is happening, we should be able to predict it, in the model, and what's left over from the regression should have *no pattern.*

I'm going to cheat now, and fit a multiple regression model that should make things better. Don't worry too much about what I'm doing; just have a look at the residual plot that comes out the other end, which is Figure 4.14.

Much better. Can you predict the residual from the fitted value now?

With that in mind, let's go back to our cars. The original scatterplot is shown in Figure 4.15 with a hint of a curve in the trend (the decrease in MPG is less for larger weights). The residual plot is shown in Figure 4.15. The trend has been removed, making any pattern in the residuals easier to see. It's not as clear as the one we saw just now, but look: the residuals are located top left, bottom middle, top right, with nothing much anywhere else. This indicates also a curved trend.

If you want to guide the eye, you can use a scatterplot smoother like `lowess`, as we did before. See Figure 4.17. This brings out the curve rather more clearly. We'll attack these data again later, when we come to look at transformations.

```
> cbind(x,y,fitted.values(lmxy),residuals(lmxy))

  x  y
1 1  3  3.833333 -0.8333333
2 2  5  5.238095 -0.2380952
3 3  7  6.642857  0.3571429
4 4  9  8.047619  0.9523810
5 5 10  9.452381  0.5476190
6 6 11 10.857143  0.1428571
7 7 12 12.261905 -0.2619048
8 8 13 13.666667 -0.6666667

> plot(fitted.values(lmxy),residuals(lmxy))
```

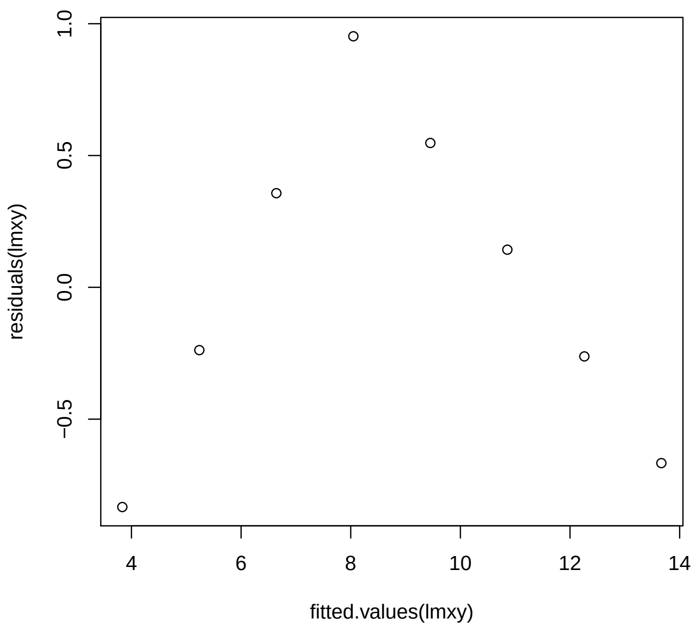


Figure 4.13: Residual plot

```
> xsq=x*x
> lmxy2=lm(y~x+xsq)
> plot(fitted.values(lmxy2),residuals(lmxy2))
```

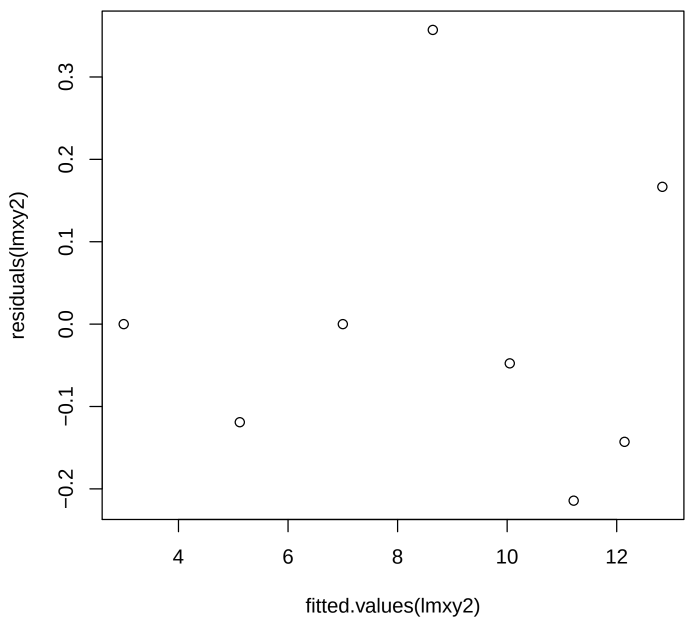


Figure 4.14: Residual plot

```
> plot(Weight,MPG)
```

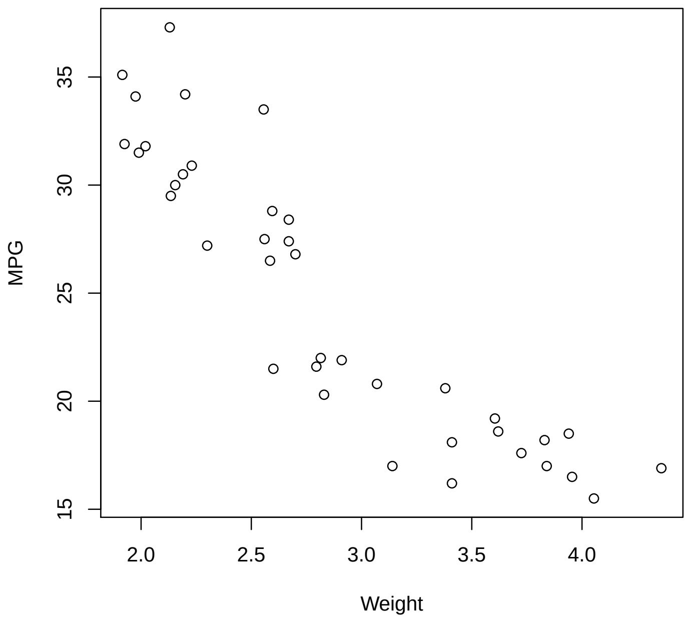


Figure 4.15: Scatterplot of weight and MPG

```
> plot(fitted.values(ans),residuals(ans))
```

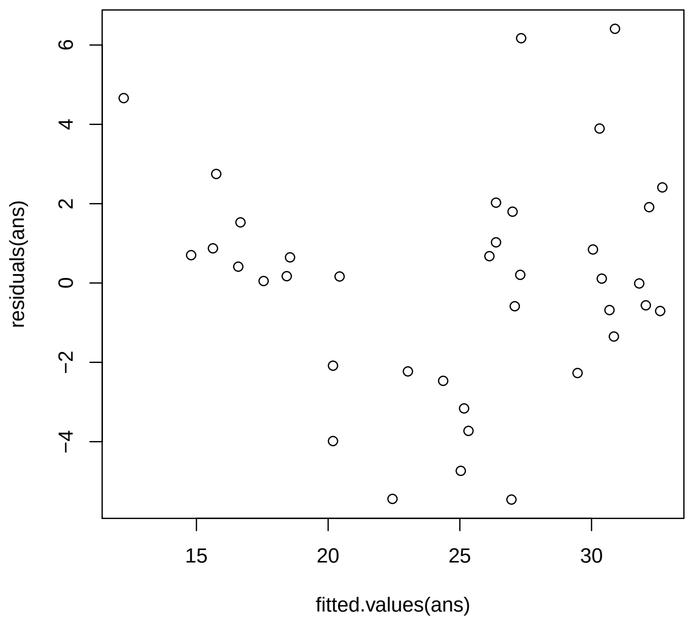


Figure 4.16: Residual plot for cars

```
> plot(fitted.values(ans),residuals(ans))
> lines(lowess(fitted.values(ans),residuals(ans)))
```

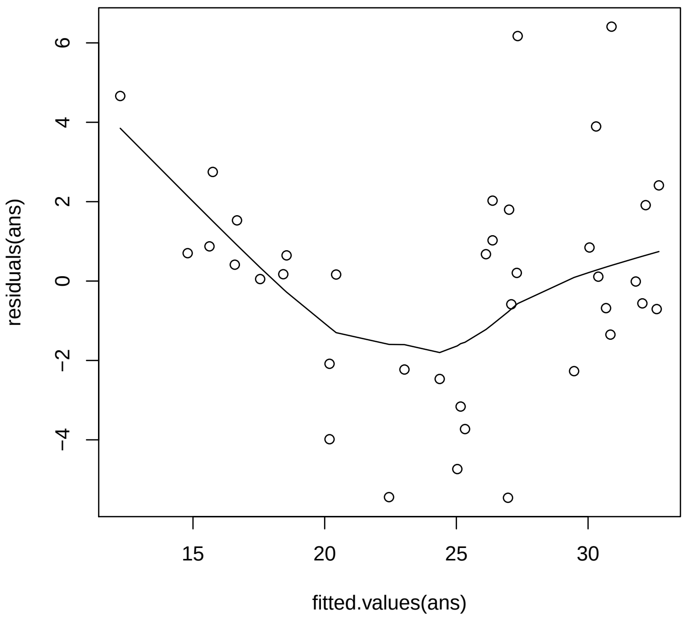


Figure 4.17: Residual plot with `lowess`

```
> par(mfrow=c(2,2))
> plot(ans)
```

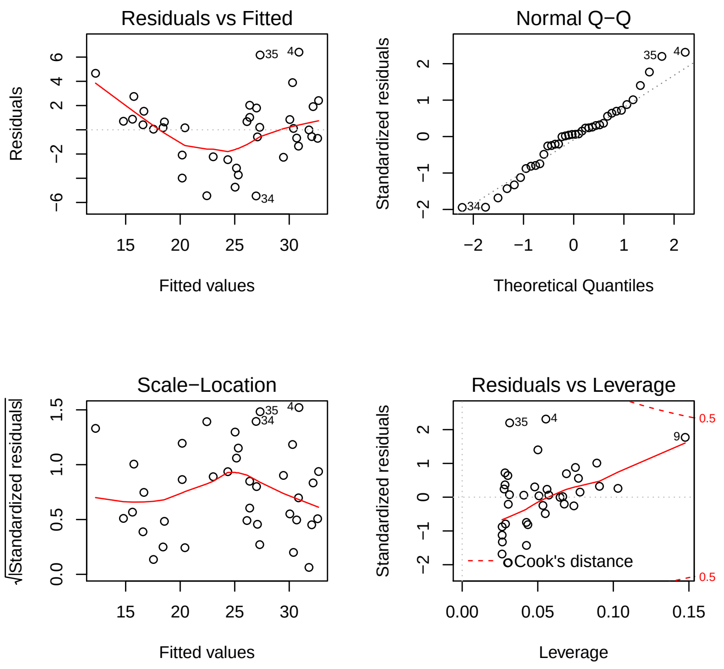


Figure 4.18:

Plotting a regression fit gives you four useful plots, of which I'll explain two. The first command in Figure 4.18 puts the four plots together on one page.

The top left plot in Figure 4.18 is the residual plot we just had. The normal QQ plot (here top right) is testing the *residuals* for normality (the assumption behind the fitting and tests is that the residuals should be normally distributed). The residuals follow the dotted line pretty well, except perhaps at the top end, where there are some larger-than-expected residuals, especially cars 4 and 35. The stuff in square brackets below is to pick out the data values and residuals for just those cars:

```
> highres=c(4,35)
> fv=fitted.values(ans)
> cbind(cars[highres,c("Car","MPG","Weight")],fv[highres])
```

```
               Car   MPG Weight fv[highres]
```

```
> pairs(cars.numeric)
```

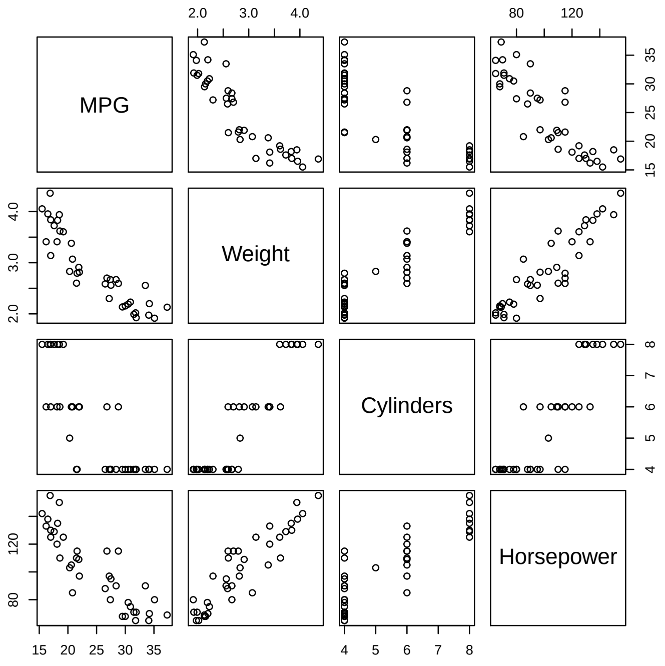


Figure 4.19: Scatterplot matrix of car numeric variables

```
4      Fiat Strada 37.3  2.130    30.89090
35 Pontiac Phoenix 33.5  2.556    27.32758
```

These cars both have higher MPG than you'd otherwise expect. The Fiat Strada has the highest MPG of all the cars, including some that weigh less than its 2.3 tons, and the Pontiac Phoenix has very good gas mileage for a car that weighs over 2.5 tons. A look at the original scatterplot suggests that its gas mileage of 33.5 is typical of a car weighing maybe 2.2 tons.

You can study the other plots and decide what they tell you!

4.3 Multiple regression

A multiple regression is like a simple regression, but with more than one explanatory variable.

Perhaps a good place to start is with a "scatterplot matrix": all the scatterplots for all the pairs of variables. See Figure 4.19. This shows the same kind of thing that the correlation matrix showed: all the variables are pretty well correlated with each other, positively or negatively. But the multiple regression tells a different story, as we'll see.

Fitting a multiple regression is simplicity itself: just put all the explanatory variables on the right-hand side of the model formaula, separated by plus signs:

```
> ans2=lm(MPG~Weight+Cylinders+Horsepower)
> ans2


Call:
lm(formula = MPG ~ Weight + Cylinders + Horsepower)

Coefficients:
(Intercept)       Weight    Cylinders   Horsepower
    49.3802      -7.3898       0.7459      -0.0736
```

We get one intercept, plus a slope for each explanatory variable. But we are getting ahead of ourselves. We should first check that this regression has any predictive value at all. Let's start at the *bottom* of the output from `summary`:

```
> summary(ans2)


Call:
lm(formula = MPG ~ Weight + Cylinders + Horsepower)

Residuals:
    Min      1Q  Median      3Q     Max
-4.4522 -1.5929 -0.3554  1.1088  6.6481

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  49.3802     1.9690  25.078  < 2e-16 ***
Weight       -7.3898     2.0797  -3.553  0.00114 **
Cylinders     0.7459     0.7252   1.029  0.31097
Horsepower   -0.0736     0.0441  -1.669  0.10433
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.796 on 34 degrees of freedom
Multiple R-squared: 0.8324,       Adjusted R-squared: 0.8176
F-statistic: 56.28 on 3 and 34 DF,  p-value: 2.838e-13
```

Look at the F-test on the last line: the P-value is 0.0000000000003. Something here definitely helps to predict MPG. Now we can look at the rest of the output to find out what.

Take a look at those slope estimates. The one for `Weight` is -7.39, which is similar to before. The one for `Cylinders` is 0.746, *positive*, which seems to mean that a car with more cylinders gets *better* MPG. How can that be? Also, the slope for `Horsepower` is -0.0736, close to zero, which seems to mean that there's almost no change in MPG as the horsepower increases.

The key to interpreting slopes in multiple regression is to say that they assess the contribution of a variable *after all the other variables in the regression have been adjusted for*. So, if you think of two cars with the same weight and horsepower, the one with more cylinders is predicted to get better MPG. Likewise, if you have two cars of the same weight and number of cylinders, increasing the horsepower will decrease the MPG only slightly.

If you are familiar with the term, slopes in a multiple regression describe "marginal" changes: a change just in that one variable while leaving everything else the same.

Another way of assessing the value of the variables in predicting MPG is to look at their P-values. The way to assess these is to ask "does this variable help in predicting the response, over and above the contributions of the other variables?". The null hypothesis is that it doesn't, and the alternative is that it does, so you are again looking for small P-values.

Let's reproduce our summary table from above, so we can have a look at that:

```
> summary(ans2)


Call:
lm(formula = MPG ~ Weight + Cylinders + Horsepower)

Residuals:
    Min      1Q  Median      3Q     Max 
-4.4522 -1.5929 -0.3554  1.1088  6.6481 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  49.3802     1.9690  25.078  < 2e-16 ***
Weight       -7.3898     2.0797  -3.553  0.00114 ** 
Cylinders     0.7459     0.7252   1.029  0.31097    
Horsepower   -0.0736     0.0441  -1.669  0.10433    
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

```
Residual standard error: 2.796 on 34 degrees of freedom
Multiple R-squared: 0.8324,       Adjusted R-squared: 0.8176
F-statistic: 56.28 on 3 and 34 DF,  p-value: 2.838e-13
```

The P-value for `Weight` is 0.00114. Weight definitely has an impact on MPG, even allowing for the other variables. But the P-value for `Cylinders` is 0.311, which is definitely not small. That's why the slope came out with the "wrong" sign: it was just chance. `Cylinders` definitely has nothing to add. The P-value for `Horsepower` is 0.104 is only smallish, so that variable doesn't have much to say either. R doesn't even put any stars on the end of that line.

So what do we do now?

One way of tackling a regression is to do as we did: start with all the variables that might be useful, and see which are significant. Then take out the ones that are not significant, and repeat until everything is significant. This is called backward elimination. Other ways are to start with nothing, and add the variable(s) that are most significant. Or you can do a mixture of the two, called stepwise regression, but that is not recommended.

I'm going to show you two different approaches. The first is to take out the least significant variables one at a time, and the second is to take out everything nonsignificant and see what happens.

For the first approach, take out `Cylinders`:

```
> ans3=lm(MPG~Weight+Horsepower)
> summary(ans3)

Call:
lm(formula = MPG ~ Weight + Horsepower)

Residuals:
    Min      1Q  Median      3Q     Max
-4.5754 -1.7809 -0.0461  1.5237  6.0916

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 48.94199    1.92398  25.438  < 2e-16 ***
Weight      -6.06455    1.63381  -3.712 0.000712 ***
Horsepower  -0.06703    0.04367  -1.535 0.133815
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.799 on 35 degrees of freedom
```

```
Multiple R-squared: 0.8272,       Adjusted R-squared: 0.8173
F-statistic: 83.76 on 2 and 35 DF,  p-value: 4.554e-14
```

The P-value for `Horsepower` has actually gotten *bigger*. Because `Cylinders` is no longer there, the slopes and P-values have moved around a bit. (They will because the three explanatory variables are correlated with each other; a heavy car is likely to have more cylinders and more horsepower in its engine).

So `Horsepower` comes out too, and we are left with our simple regression model:

```
> summary(ans)


Call:
lm(formula = MPG ~ Weight)

Residuals:
    Min      1Q  Median      3Q     Max
-5.4595 -1.9004  0.1686  1.4032  6.4091

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   48.708      1.954   24.93  < 2e-16 ***
Weight        -8.365      0.663  -12.62 8.89e-15 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.851 on 36 degrees of freedom
Multiple R-squared: 0.8155,       Adjusted R-squared: 0.8104
F-statistic: 159.2 on 1 and 36 DF,  p-value: 8.889e-15
```

The second way of doing things is to pull out both `Cylinders` and `Horsepower` at once. We have to be a little careful about this, since taking out one might make the other one significant. But there is a way to test this all in one go. Recall that `ans` was the regression with only `Weight` in it, and `ans2` was the regression with all 3 x's in it. Feed these two into the `anova` function, with the smallest (fewest x's) model first, to get:

```
> a=anova(ans,ans2)
> a


Analysis of Variance Table

Model 1: MPG ~ Weight
Model 2: MPG ~ Weight + Cylinders + Horsepower
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1     36 292.57
2     34 265.85  2    26.722 1.7087 0.1963
```

The interpretation is "is `ans2` an improvement over `ans`?" The null is that it is not, and the alternative is that is. So we are looking for a small P-value. Is 0.196 small? No way. So the other variables have nothing to add over `Weight`.

When we were doing simple regression, we noticed (perhaps by looking at the residual plot) that the relatinship between `MPG` and `Weight` looked a bit curved rather than linear. There are basically two ways around that: fix up the response, or fix up the explanatory variables. The first of those is the domain of Transformations, which we look at in Section 6.8. The second we can think about now. What we are going to to is to fit a parabola, which is a particular (simple) kind of curve. This will give us some insight as to whether a curve is *really* better than a straight line.

First we define a new variable to be the explanatory variable squared (multiplied by itself). Then we add this to the regression (making it a multiple regression again) and test the squared term for significance. If it *is* significant, a curve really does fit better than a straight line.

```
> wsq=Weight*Weight
> ans3=lm(MPG~Weight+wsq)
> summary(ans3)


Call:
lm(formula = MPG ~ Weight + wsq)

Residuals:
    Min      1Q  Median      3Q     Max
-4.4017 -1.6442 -0.0489  1.1502  7.1115

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  75.3716     8.2133   9.177 7.64e-11 ***
Weight      -27.1279     5.6807  -4.775 3.16e-05 ***
wsq           3.1158     0.9383   3.321  0.00211 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.521 on 35 degrees of freedom
Multiple R-squared: 0.8597,       Adjusted R-squared: 0.8517
F-statistic: 107.3 on 2 and 35 DF,  p-value: 1.18e-15
```

```
> par(mfrow=c(2,2))
> plot(ans3)
```

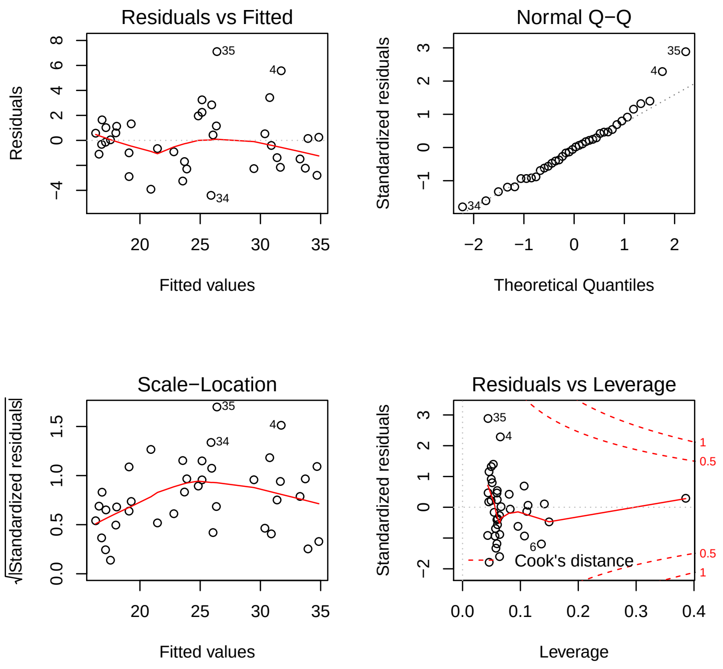


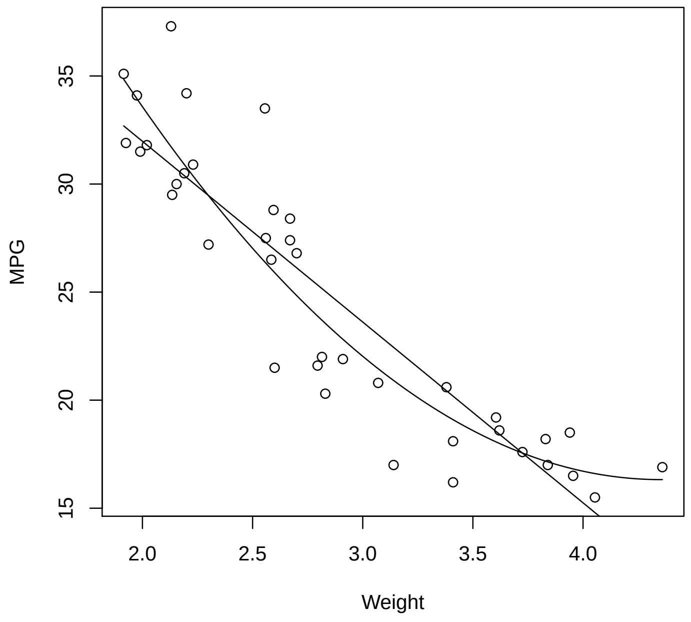
Figure 4.20: Diagnostic plots for curved regression

OK, so the quadratic (squared) term is indeed significant. The curve is real; the curve really is a better description of the data than the straight line is. Let's check the plots. Are there any other problems?

The residuals look nicely random now, though two of the cars (#4 and #35) still have positive residuals that are maybe a bit large (look at the normal QQ plot top right). Comparison with the corresponding plots from the regression with just `Weight` reveals that things are a bit better.

I thought I would produce a scatterplot with both the line and the curve on it, so that you can see how the curve conforms better to the data.

```
> plot(Weight,MPG)
> lines(spline(Weight,fitted.values(ans3)))
> lines(spline(Weight,fitted.values(ans)))
```



One word of warning with curves like this, though; they always curve down and then up (or up and then down), so you need to be careful that they are actually behaving according to the data.

4.4 Predictions, prediction intervals and extrapolation

Now would be a good place to mention the perils of extrapolation, that is, predicting outside the range of the data. In our case, that would be cars with weight between 2 and about 4.5 tons. Let's see what happens when we predict the gas mileage for a car with weight 6 tons, from the linear regression `ans` and the curved one `ans3`.

First I need to create a mini data frame of values to predict. The data frame I create has to have a thing called `Weight`, which I can do as below. Also, for doing the prediction from the curve, I need a thing called `wsq` which is the weight squared. This is a bit complicated, so let's take it steps. First, the creation of the data frame:

```
> w.pred=6
> wsq.pred=w.pred*w.pred
> pred.df=data.frame(Weight=w.pred,wsq=wsq.pred)
> pred.df

  Weight wsq
1      6  36
```

Now the actual predictions. The workhorse for this is `predict`, which is another multi-coloured R function; whatever you feed it, it will try to do a prediction for it. Within reason.

`predict` needs two things: a regression fit, and a data frame of values to predict for. The latter we created just above, and the former can be `ans` (the straight line) or `ans3` (the curve). (There was an `ans2`, which, if you recall, was the multiple regression predicting `MPG` from everything else. But we decided that was no better than just using `Weight`.)

Below, I'm saving each of the predictions in a variable, and then using `cbind` to glue the predictions as extra "columns" on the end of the data frame. In a moment, you'll see why I'm doing this rather than just using `c`, which, for this prediction, would be just as good. Also, bear in mind that for the cars we had, the predictions from the line and the curve were quite similar. Here we go:

```
> ans.pred=predict(ans,pred.df)
> ans3.pred=predict(ans3,pred.df)
> cbind(pred.df,ans.pred,ans3.pred)

  Weight wsq  ans.pred ans3.pred
1      6  36 -1.480104  24.77174
```

The predicted MPG values for a car of weight 6 tons were -1.48 from the line and 24.77 from the curve. These are *completely different*, and, what's more, neither of them make much sense. An `MPG` figure cannot possibly be negative, and also the figure of 24.77 shows a *higher* gas mileage than the heaviest cars in our data set. What's going on?

To shed some light, let's do not just one prediction but a bunch. Let's predict from 2 tons all the way up to 6. I structured my code above so that I could just copy and paste, changing a very few things along the way. Can you see what's happening below?

```
> w.pred=2:6
> wsq.pred=w.pred*w.pred
> pred.df=data.frame(Weight=w.pred,wsq=wsq.pred)
> pred.df
```

```
  Weight wsq
1      2   4
2      3   9
3      4  16
4      5  25
5      6  36
```

Notice now that R knows that when you want to multiply a vector of numbers by itself, you want to multiply *each number* by itself and make a vector out of the results. (This is not the same as matrix multiplication, if you know about that, which R can also do, but differently from this.)

All right, the predictions:

```
> ans.pred=predict(ans,pred.df)
> ans3.pred=predict(ans3,pred.df)
> cbind(pred.df,ans.pred,ans3.pred)
```

```
  Weight wsq  ans.pred ans3.pred
1      2   4 31.978296  33.57890
2      3   9 23.613696  22.02984
3      4  16 15.249096  16.71229
4      5  25  6.884496  17.62626
5      6  36 -1.480104  24.77174
```

For weights up to 4 tons (where our data ends), the predictions are very similar, but after that, things start to go crazy. The straight line just keeps on going down and down. It doesn't know it's "supposed" to stop at zero! The predicted MPG figures just keep on decreasing by about 8 MPG per ton. For ever. That's what straight lines do.

The predictions from the curve level off around the end of our data, which is what the data suggest, and then — start going up again! This is what parabolas do: they turn the corner and start going the other way. This is a mathematical feature of this kind of curve, but is entirely unsupported by our data! (You might imagine that if we had some heavier vehicles in our data set, the MPG figures would go down at a decreasing rate, never reaching zero. Then the curve we fitted might not do such a good job.)

Now I'm going to add something to the predictions, a so-called "prediction interval". This is saying, suppose I had another car, of weight 2, 3, 4, 5 or 6 tons. Then what would be a (95%) confidence interval for the predicted MPG each time? I just add an `interval="p"` to each call to `predict` to get this:

```
> ans.pred=predict(ans,pred.df,interval="p")
> ans3.pred=predict(ans3,pred.df,interval="p")
> cbind(pred.df,ans.pred,ans3.pred)
```

```
  Weight wsq       fit        lwr       upr      fit       lwr      upr
1      2   4 31.978296 26.0071938 37.949397 33.57890 28.203076 38.95473
2      3   9 23.613696 17.7535136 29.473878 22.02984 16.752425 27.30725
3      4  16 15.249096  9.1955288 21.302663 16.71229 11.279126 22.14545
4      5  25  6.884496  0.3602431 13.408749 17.62626  8.880822 26.37169
5      6  36 -1.480104 -8.6982988  5.738091 24.77174  7.497564 42.04592
```

The first pair of `lwr` and `upr` columns are the lower and upper limits of the prediction intervals from the line. As you go off the end of the data, these intervals get gradually wider, and at least the upper ends of the intervals haven't gone below zero yet. (These intervals are valid if the straight line model continues to hold, which we don't believe.)

The last two columns give us the prediction intervals for the curve. Within the data, they are not much different from the predictions for the line, but once you go off the end of the data, the intervals get *way* wider, reflecting that even if that curve continued, we would be almost completely ignorant about the MPG of a car that weighed 6 tons. When you get a prediction interval like that, you *know* you are extrapolating!

So we've learned that this curve models the data we have quite well, but it soon starts to make no sense beyond that. Is there anything else we can do? Yes, instead of adding a curved part to the explanatory variable, we can incorporate the curve into the *response* by making a **transformation**, which is the subject of Section 6.8. In the case of our cars data, we end up with what I think is quite a satisfying model for gas consumption.

4.5 Quality control for multiple regression

Let's have a look at some data on salaries of mathematicians. There are four variables: the salary (response) in thousands of dollars, an index of work quality (`workqual`), the number of years of experience (`experience`), and an index of publication success (`pubsucc`). The aim is to use the other variables to predict salary.

Figure 4.21 shows an analysis. All three explanatory variables are worth keeping, in that their small P-values say that it would be a bad idea to take any of them out. (Each variable has something to contribute over and above the others.)

The first step in doing quality control is to look at the output from plotting the regression object, the plot at the bottom of Figure 4.21. The first question to ask is whether there is any relationship between the residuals and fitted values. There shouldn't be. The plot shows that there really isn't. The residuals look acceptably normal, as shown in the QQ plot of residuals, and the third plot shows whether the residuals tend to vary in size as the fitted values get bigger (they really don't).

Any problems in these plots suggest that there is something wrong with the response that needs fixing (eg. by transformation, Section 6.8).

The other kinds of plots that ought to be looked at are of the residuals against the explanatory variables individually. These are shown in Figure 4.22. I've added a `lowess` curve to each one. There appears to be no relationship between residuals and any of the explanatory variables. So we are good.

Let's have a look at some data that I tinkered with to make something fail. I'll spare you the details of how I made up the data (first two lines of Figure 4.23), and concentrate on what I got.

The story is shown in Figure 4.23. First, some of the data. There are 24 observations on two x-variables `x1` and `x2`, with a response variable `y`. Then a multiple regression predicting `y` from `x1` and `x2`, with a look at the results. Nothing strange here, just a very strong dependence of `y` on the two explanatory variables.

The diagnostic plots are shown in Figure 4.24. Are there any problems here? You bet. What is happening is most clearly shown on the plot of residuals against fitted values. There is a clear curve, so there is a curve in the relationship. A problem here generally indicates a problem with the response, rather than the explanatory variables, but we can plot the residuals against them too,

```
> mathsal=read.table("mathsal.txt",header=T)
> ms1.lm=lm(salary~workqual+experience+pubsucc,data=mathsal)
> summary(ms1.lm)

Call:
lm(formula = salary ~ workqual + experience + pubsucc, data = mathsal)

Residuals:
    Min      1Q  Median      3Q     Max
-3.2463 -0.9593  0.0377  1.1995  3.3089

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 17.84693    2.00188   8.915 2.10e-08 ***
workqual     1.10313    0.32957   3.347 0.003209 **
experience   0.32152    0.03711   8.664 3.33e-08 ***
pubsucc      1.28894    0.29848   4.318 0.000334 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.753 on 20 degrees of freedom
Multiple R-squared: 0.9109,      Adjusted R-squared: 0.8975
F-statistic: 68.12 on 3 and 20 DF,  p-value: 1.124e-10

> par(mfrow=c(2,2))
> plot(ms1.lm)
```

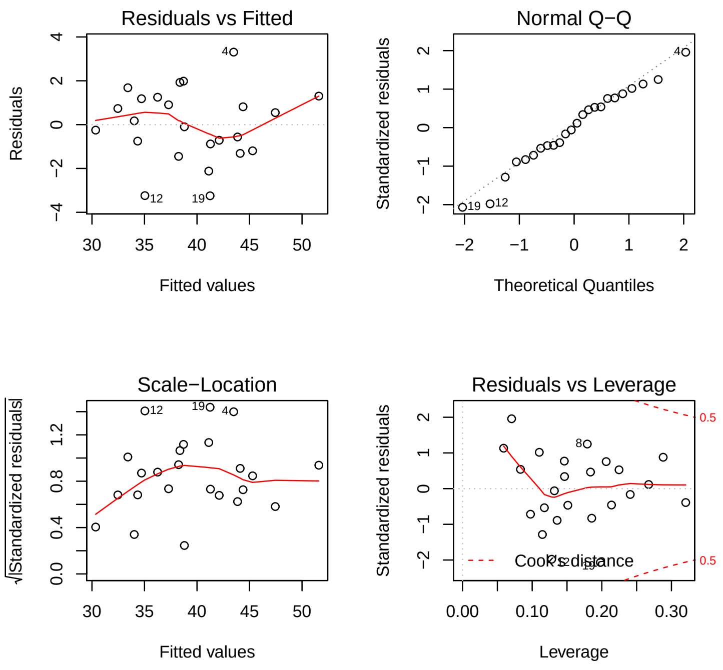


Figure 4.21: Analysis of mathematician salary data

```
> par(mfrow=c(2,2))
> plot(mathsal$workqual,resid(ms1.lm))
> lines(lowess(mathsal$workqual,resid(ms1.lm)))
> plot(mathsal$experience,resid(ms1.lm))
> lines(lowess(mathsal$experience,resid(ms1.lm)))
> plot(mathsal$pubsucc,resid(ms1.lm))
> lines(lowess(mathsal$pubsucc,resid(ms1.lm)))
```

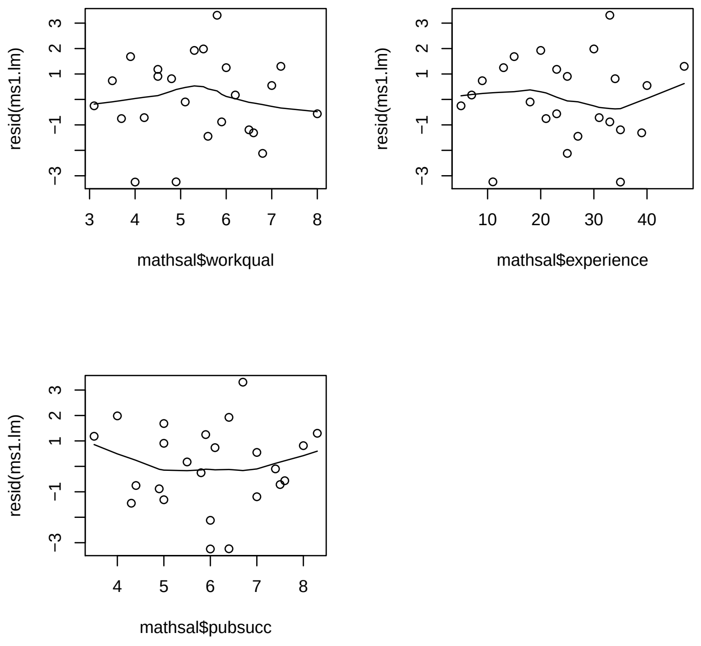


Figure 4.22: Plots of residuals against explanatory variables

```
> madeup=expand.grid(x1=0:4,x2=0:4,KEEP.OUT.ATTRS=F)
> madeup$y=3*(madeup$x1+2*madeup$x2+3+rnorm(25,0,0.5))^2
> head(madeup)

  x1 x2         y
1  0  0  38.09906
2  1  0  42.48683
3  2  0  91.24105
4  3  0 112.38970
5  4  0 104.16102
6  0  1  50.00597

> madeup.lm=lm(y~x1+x2,data=madeup)
> summary(madeup.lm)

Call:
lm(formula = y ~ x1 + x2, data = madeup)

Residuals:
   Min     1Q Median     3Q    Max
-77.75 -25.39 -14.60  19.81 123.70

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -58.260     22.216  -2.622   0.0156 *
x1            53.687      7.025   7.642 1.25e-07 ***
x2           108.504      7.025  15.445 2.73e-13 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 49.68 on 22 degrees of freedom
Multiple R-squared: 0.931,        Adjusted R-squared: 0.9247
F-statistic: 148.5 on 2 and 22 DF,  p-value: 1.683e-13
```

Figure 4.23: Made-up data to illustrate regression quality control

```
> par(mfrow=c(2,2))
> plot(madeup.lm)
```

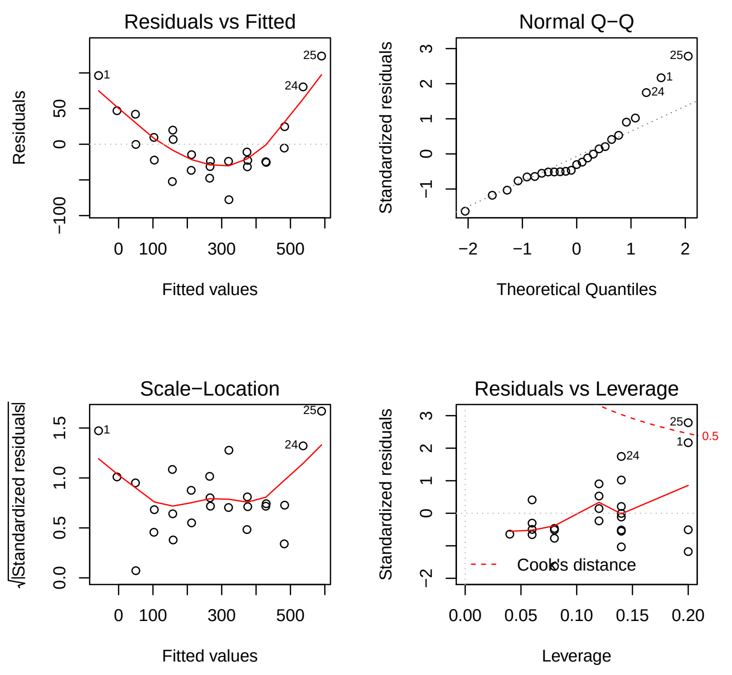


Figure 4.24: Diagnostic plots for made-up data

```
> par(mfrow=c(1,2))
> r=resid(madeup.lm)
> x1=madeup$x1
> x2=madeup$x2
> plot(x1,r)
> lines(lowess(x1,r))
> plot(x2,r)
> lines(lowess(x2,r))
```

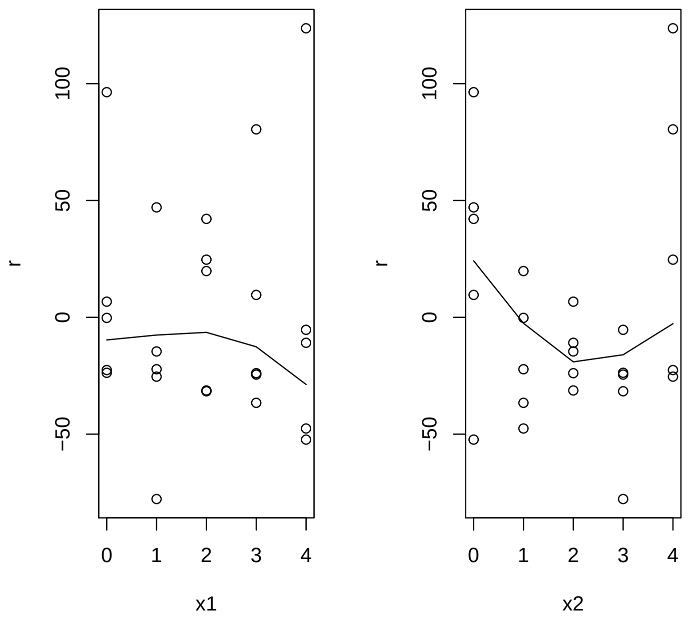


Figure 4.25: Plots of residuals against explanatory

```
> shingles=read.table("shingles.txt",header=T)
> head(shingles)

  sales promotion active competing potential
1  79.3       5.5     31        10         8
2 200.1       2.5     55         8         6
3 163.2       8.0     67        12         9
4 200.1       3.0     50         7        16
5 146.0       3.0     38         8        15
6 177.7       2.9     71        12        17
```

Figure 4.26: (Part of) roofing shingles data

as in Figure 4.25.

Note that I first made simpler names for the variables, then I made a one-row by two-column plotting area, and made the two plots with `lowess` curves on them. There is some evidence of curvature on both of these. This further suggests that fixing up `y1` is the thing to try first. As it happens, replacing y with $\sqrt{y}$ fixes everything up nicely.

Let's look at another example. This one is about sales of roofing shingles, and what that might depend on. The response is sales last year (in thousands of squares), and the explanatory variables are:

- promotional expenditures (thousands of dollars)
- number of active accounts
- number or competing brands
- district potential (measured somehow)

The data (some of it, anyway) are shown in Figure 4.26. The first thing to think about is a plot. Plotting the entire data frame produces Figure 4.27. (All the variables are numerical ones.) Look along the top row: it looks as if sales has an upward linear relationship with the number of accounts, and a downward linear one with the number of competing brands, and not much of a relationship with anything else. Let's see whether a regression bears this out.

The regression in Figure 4.28 does indeed support our guess from the scatterplots. The first thing to look at is the F-test at the bottom, which is strongly significant. *Something* is helping to predict sales. Now we can look at the P-values in the last column of the table of coefficients. The P-values for `active` and `competing` are *very* small, and those for `promotion` and `potential` are not.

```
> plot(shingles)
```

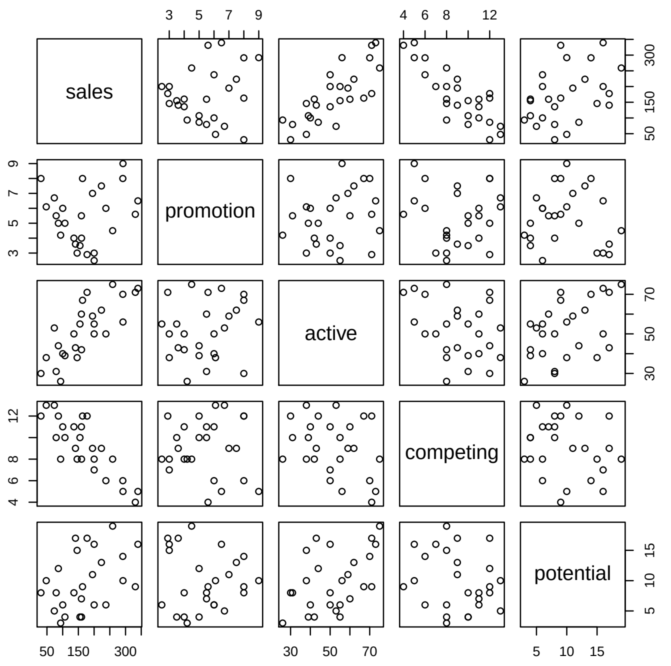


Figure 4.27: Scatterplot matrix of shingles data

```
> shingles.lm1=lm(sales~promotion+active+competing+potential,data=shingles)
> summary(shingles.lm1)

Call:
lm(formula = sales ~ promotion + active + competing + potential,
    data = shingles)

Residuals:
     Min       1Q   Median       3Q      Max
-19.0906  -5.9796   0.8968   6.5667  14.7985

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 178.3203    12.9603  13.759 5.62e-12 ***
promotion     1.8071     1.0810   1.672    0.109
active        3.3178     0.1629  20.368 2.60e-15 ***
competing   -21.1850     0.7879 -26.887  < 2e-16 ***
potential     0.3245     0.4678   0.694    0.495
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 9.604 on 21 degrees of freedom
Multiple R-squared: 0.9892,     Adjusted R-squared: 0.9871
F-statistic: 479.1 on 4 and 21 DF,  p-value: < 2.2e-16
```

Figure 4.28: Regression of sales on everything

Sometimes looking at the scatterplots is *too* simple-minded in that number of active accounts and number of competing brands might be highly related with each other, so that you really need only one of them in the regression. But if you look back at Figure 4.27, you'll see (in the third row and fourth column) that `active` and `competing` have nothing to do with each other. So they both have something to contribute.

On the other hand, there appears in Figure 4.27 to be a weak upward relationship between `sales` and `potential` (first row, last plot). So `potential` is potentially (ha!) a useful variable. But `potential` also has something of a relationship with `active`, so that whatever `potential` has to say about `sales` is also said by `active`, and therefore `potential` has nothing to add. Hence its non-significance.

The next step is to take out the non-significant `promotion` and `potential`. This is done in Figure 4.29. Before we actually look at the `summary` from this model, we should check that taking out these two variables hasn't somehow made the fit significantly worse. That's the purpose of the `anova`. The P-value there is large, so the fit is *not* significantly worse, and we are good to go on.

Looking at the `summary` of `shingles.lm2`, both variables `active` and `competing` are as significant as they could be (the P-values are both 0 to at least 15 decimals!). So we've arrived at a good model, and the R-squared is very high indeed. But we are not done yet: we need to do our quality control and make sure that nothing funny is happening with the residuals.

We plot the fitted model object in Figure 4.30. Before doing that, however, we have to make room for four plots, or else we'll only see the first one!

The top left plot is of residuals against fitted values. We want to see randomness here, and no trend for the residuals to change as the fitted values get bigger. That's what we've got. Three or so of the observations are off in sales by about 20 thousand dollars, but these residuals are not out of line with the rest. The red lowess curve on this plot is more or less horizontal. At the top right is a normal QQ plot of the residuals; they are about as normally-distributed as you could wish to see. The most positive and most negative residuals are right on the dotted line.

The only plot I have any concerns with at all is the bottom left one. This shows the *size* of the residuals against the fitted values. The residuals, looking at the lowess curve, are smallest in size on average when the fitted value is about 150, and larger otherwise. But I'm willing to believe this is just coincidence.

Not quite done yet. In a multiple regression, we should also plot the residuals against each of the explanatory variables, just in case there are any problems with those that didn't show up yet. I'm plotting these side by side, as shown in Figure 4.31. There was a lot of copying and pasting in constructing those

```
> shingles.lm2=lm(sales~active+competing,data=shingles)
> anova(shingles.lm2,shingles.lm1)

Analysis of Variance Table

Model 1: sales ~ active + competing
Model 2: sales ~ promotion + active + competing + potential
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1     23 2210.4
2     21 1937.1  2    273.31 1.4814 0.2501

> summary(shingles.lm2)

Call:
lm(formula = sales ~ active + competing, data = shingles)

Residuals:
     Min       1Q   Median       3Q      Max
-18.4136  -6.1499  -0.5683   6.2472  20.3185

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 186.6940    12.2587   15.23 1.66e-13 ***
active        3.4081     0.1458   23.37  < 2e-16 ***
competing   -21.1930     0.8028  -26.40  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 9.803 on 23 degrees of freedom
Multiple R-squared: 0.9876,       Adjusted R-squared: 0.9866
F-statistic: 918.3 on 2 and 23 DF,  p-value: < 2.2e-16
```

Figure 4.29: Model 2, taking out the non-significant variables

```
> par(mfrow=c(2,2))
> plot(shingles.lm2)
```

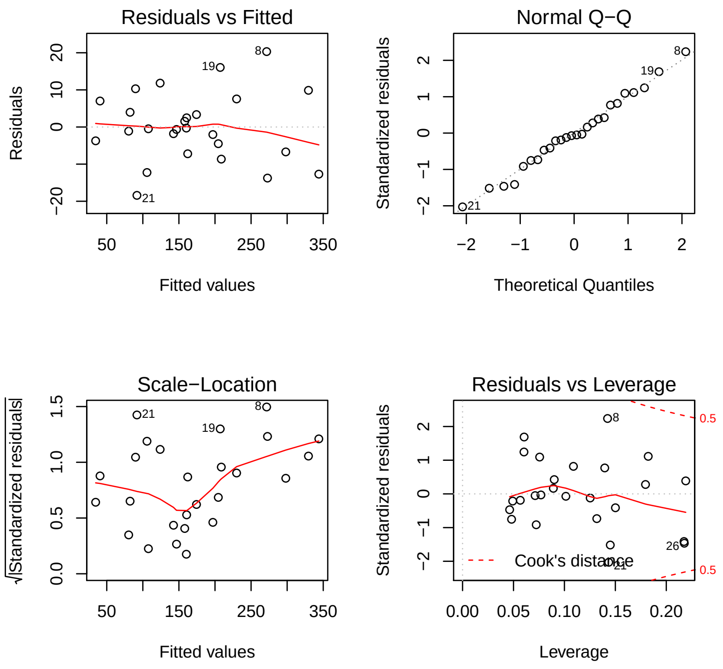


Figure 4.30: Fitted model plot for Model 2

```
> attach(shingles)
> plot(active,residuals(shingles.lm2))
> lines(lowess(active,residuals(shingles.lm2)))
```

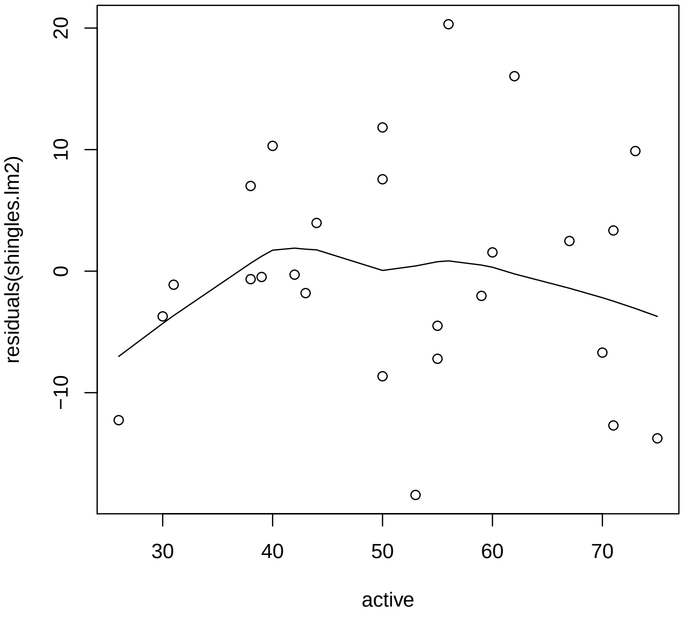


Figure 4.31: Plots of residuals against `active`

commands! I've added a lowess curve to each plot.

The plot of residuals against `active` is in Figure 4.31 and against `competing` is in Figure 4.32.

I'm a little worried by the curves in these plots. There's a hint of an up-and-down (or down-and-up) in each. I'm willing to believe that the lowess trend is curved in the `active` plot because the first three residuals happened to be negative, but the curve on the `competing` plot seems a little more substantial.

One way to see whether this is real or happenstance is to add squared terms in the two explanatory variables to the model and see whether they are significant. This is done in Figure 4.33. The `anova` tests whether the two squared terms together offer any improvement, and the t tests in the `summary` output say whether either of them individually have anything to say.

```
> plot(competing,residuals(shingles.lm2))
> lines(lowess(competing,residuals(shingles.lm2)))
> detach(shingles)
```

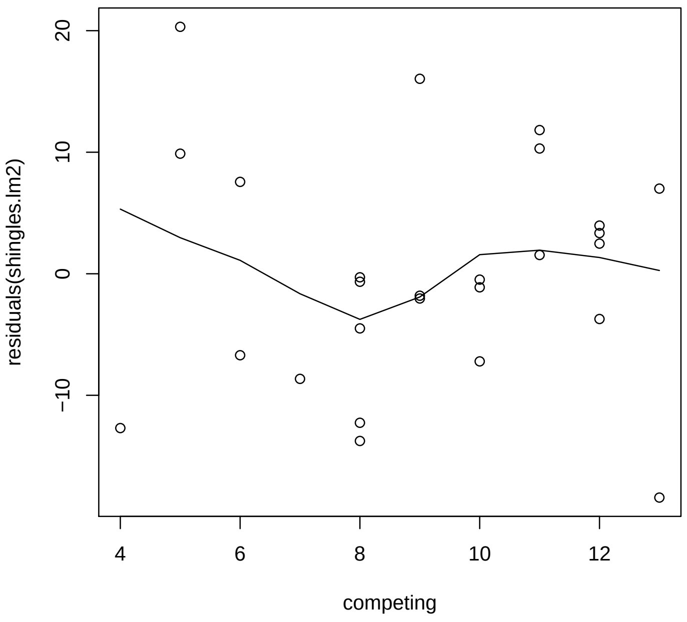


Figure 4.32: Plot of residuals against `competing`

```
> attach(shingles)
> actsq=active*active
> compsq=competing*competing
> detach(shingles)
> shingles.lm3=lm(sales~active+competing+actsq+compsq,data=shingles)
> anova(shingles.lm2,shingles.lm3)

Analysis of Variance Table

Model 1: sales ~ active + competing
Model 2: sales ~ active + competing + actsq + compsq
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1     23 2210.4
2     21 1991.8  2    218.65 1.1527  0.335

> summary(shingles.lm3)

Call:
lm(formula = sales ~ active + competing + actsq + compsq, data = shingles)

Residuals:
     Min       1Q   Median       3Q      Max
-21.8491  -5.2099  -0.2234   5.5086  16.0810

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 159.98513   34.72340   4.607 0.000152 ***
active        4.95488    1.04609   4.737 0.000112 ***
competing   -23.37219    5.56428  -4.200 0.000402 ***
actsq        -0.01518    0.01010  -1.503 0.147618
compsq        0.11325    0.30870   0.367 0.717394
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 9.739 on 21 degrees of freedom
Multiple R-squared: 0.9889,       Adjusted R-squared: 0.9867
F-statistic: 465.8 on 4 and 21 DF,  p-value: < 2.2e-16
```

Figure 4.33: Regression including squared terms

The `anova` has a P-value of 0.335, which is far from significant. The two individual P-values, from the `summary` table, 0.148 and 0.717, are also neither nearly significant. In other words, the squared terms add nothing to the regression, and we don't do any better by including them. Including them in the regression, therefore, was a bit of an overreaction to the residual plots. But it was a good idea to check.

Chapter 5

Designing and running experiments

5.1 Introduction

A statistical experiment is not something done in a lab by people in white coats (though it could be). The aim of a statistical experiment is to do something under more-or-less controlled conditions to find out something about what causes what. An example of this is to find out whether a new drug works in treating a disease. You take one group of people who have this disease, and you give them the new drug. At the same time, you take another group of disease sufferers and give them the current standard treatment. Then you assess the improvement of the people in each group, and if the new group comes out sufficiently better, you declare the new drug to be better than the standard treatment.

Why "sufficiently better"? Well, the new drug might have come out better by chance (this is a 50-50 proposition if there was really no difference in effectiveness between new and old). So you want the new drug to be "more better" than that. This is where statistical significance, small P-values and all that comes in.

5.2 Comparison, control and randomization

5.2.1 Comparison

There are three major aspects to statistical experimentation, and why it looks as it does. The first is comparison. Let's pursue our drug example a bit.

One way to test our new drug is to test it out on a group of people who have the disease, and see whether they get better. If they do, at least on average, you declare the drug a success.

The problem is, how do you know these people wouldn't have gotten better on their own (eg. if the disease is the common cold), or whether the standard treatment would have been just as good? The thing is, you don't, unless you include this in your experiment. There needs to be some element of *comparison*, and demonstration that what you're really testing is better than the competition.

The statistical word for the new drug (or new procedure or whatever it is) is **treatment**, even if it's not a treatment in the medical sense. So you need to have a **treatment group** of people who get the treatment. You also need a group of people who don't get the treatment (the standard drug, or nothing at all). These people are called the **control group**.

The other part of "comparison" is that whatever you're testing needs to be not just *better* than the competition, but *significantly better*. Why is this? Well, suppose your new drug and the standard drug are equally good. In the jargon, there is no **treatment effect**. Then, people being people, the results won't be absolutely identical between the new drug and the standard one. One of them, and it might be the new drug, will come out better just by chance. But we are looking for more than that: we want the new drug to come out better because it *is* better. This is exactly what statistical significance tells you. It says, "*if* the two drugs were actually equally good, how likely am I to observe this kind of difference?" This is the P-value. If the answer is "not very", ie. the P-value is small, then we are entitled to conclude that the new drug is better.

The other thing about comparison, when your individuals are people, is called the "placebo effect". People who receive something that looks like a treatment, or who take part in an experiment, even if they're actually in the control group, will tend to show some kind of effect. This is well documented. For people, anyway. Animals or inanimate objects tend not to react this way. If you're dealing with people, you try to make the control non-treatment look as much like the real treatment as possible. This might mean giving a pill that contains no active ingredient, or designing an exercise program that does not include what you are testing the effect of. Also, you might want to organize things (where possible) so that the people administering the treatment don't know whether it's the real one or a fake one. This is a way of avoiding all kinds of bias from people who know what's going on and could influence the results. It's called **blinding**.

5.2.2 Control

The idea of doing science experiments in a lab means that you don't have to deal with the messy real world. The benefit of this is that you have *control* over what happens in your experiment: your results are not going to get affected so much by things outside your experiment.

Statistical experiments have a "protocol", for the people running the experiment, and (if they are human) for the individuals in the experiment as well. For example, if you are testing out a new treatment for the common cold, you don't want the individuals in your experiment taking other cold remedies while your experiment is going on. If they did, you wouldn't know whether an individual got better because of your new treatment, or because of whatever else they were taking.

The bottom line here is that any other variable that might make a difference to your results should be controlled. Or, if it's impossible to do that, you randomize over it, which is dealt with in Section 5.2.3.

There's a distinction between two kinds of factors in this kind of work. For the factor we're interested in (like the new drug vs. the standard drug, or doing laundry at different temperatures to see which is best), we *want* to try different values or **levels**, such as "new drug", "standard drug", "nothing" or "cold", "warm", "hot", to see what difference they make. Other factors, the ones we don't care about, such as the severity of the cold or the kind of fabric being washed, we should try to control, so that all the individuals (people or items to be washed) are held constant.

Holding other things constant can be difficult, and besides, it can make it difficult to generalize our results to "all fabrics" or "all severities of cold". There are a couple of ways around this, one of which is randomization (again), and the other is to include these other things that you can measure into the design of the experiment as additional factors — not because you care about them, but because they might make a difference to the results. Other things like age and gender, which you can measure but not control, fall into this category.

So the story is "control what you can, and randomize the rest". What is this randomization that I speak of? Coming up next.

5.2.3 Randomization

The point of randomization is that who ends up in your different groups has *nothing to do* with anything else (and is *totally unpredictable ahead of time*). Is this a good thing?

Well, imagine a study of a new treatment of a serious disease. The doctor running the study decides that all the most severe cases should get the old treatment "because the new treatment isn't proven yet". Lo and behold, at the end of the study, the new treatment comes out best. What does this prove? Not much. The new treatment could have come out better because it actually *was* better. Or because the old treatment was handicapped by being given all the most severe cases. *We can't tell which.* The variables "disease severity" and "treatment new/old" are, in the jargon, **confounded**: we cannot disentangle their effects, and therefore we have no way to know which deserves the credit.

So what can we do that's better? We can assign the cases to the new treatment or not **at random** (for example, by flipping a coin). That way, getting the new treatment or the old one has nothing to do with anything else, and so if the new treatment comes out better, you can be more confident that it came out better because it *is* better, not because of anything else. Another way to look at this: if the individuals going into the two groups are chosen at random, the two groups should be about the same going in (with regard to *any* other variables, including any you *did not* think of. So if they come out different at the end, it must be because of the thing you changed: namely, the treatment.

This is a little bit too bold, because the groups could have come out different at the end by chance: for example, you might have happened to get some of the more serious cases in one group just by chance. But that is taken care of by the test of significance: "is the difference we saw greater than would have expected by chance?"

All right, how do we get R to do the randomization? Let's imagine we have 20 individuals, numbered 1 through 20, and we have a new treatment and a standard treatment.

Let's start with the coin-tossing idea, and then think about how we can do better. This is shown in Figure 5.1. First we set up our people. The function `rbinom` takes random samples from binomial distributions. We want single successes or failures (in the new treatment group or not (this is the variable `r`), and when we want to use this, we want them to be `TRUE` and `FALSE`, which is what the variable `sel` is ("selected for new treatment"). To get a list of *which* people are in each group, we use our logical vector `sel` to pick out the people respectively in and not-in the new-treatment group. So people, so people 1, 4, 6, 9 and so on get the new treatment, and people 2, 3, 5, 7 and so on get the old one. This would work just the same if `people` contained a list of names; `people[sel]` would pick out the *names* of the people getting the new treatment.

The last line indicates a problem with the coin-tossing method. We happened to get 11 people in the new-treatment group. There is, in general, no guarantee that the groups will be the same size, because there is no *guarantee* that you will get 10 heads when you toss a coin 20 times. But you get the best (most powerful) tests if your groups are the same size. How to arrange this?

```
> people=1:20
> r=rbinom(20,1,0.5)
> r

 [1] 1 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1 0 1 1

> sel=as.logical(r)
> sel

 [1]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE
[13] FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE

> people[sel]

 [1]  1  4  6  9 10 11 15 16 17 19 20

> people[!sel]

[1]  2  3  5  7  8 12 13 14 18

> table(sel)

sel
FALSE  TRUE
    9    11
```

Figure 5.1: Choosing individuals by coin-tossing

```
> sample(people,10)

 [1]  1  7  3 11 15 10 12 17 18  9

> s=sample(people)
> s[1:10]

 [1] 11  4 17  8 18 13  9 15  1 10

> s[11:20]

 [1] 14 20  6  5  7  2 12 19 16  3

> s4=sample(people)
> s4[1:5]

[1] 16 13 14  2  9

> s4[6:10]

[1]  4  7  5 17 15

> s4[11:15]

[1]  6 10  3 18 20

> s4[16:20]

[1] 12  1  8 19 11
```

Figure 5.2: Using `sample` to select groups

```
> females=1:15
> males=16:30
> f.sampled=sample(females)
> m.sampled=sample(males)
> c(f.sampled[1:5],m.sampled[1:5])

 [1] 14  2 12  9 11 27 17 26 24 18

> c(f.sampled[6:10],m.sampled[6:10])

 [1]  1  3 13  5  6 29 23 19 25 30

> c(f.sampled[11:15],m.sampled[11:15])

 [1]  8 15 10  4  7 28 16 22 20 21
```

Figure 5.3: Randomly selecting males and females for each of 3 groups

The key is the R function `sample`. The basic usage is shown on the first line of Figure 5.2. You feed `sample` a list of things to sample from (could be numbers or names) and how many of them to sample, and you get back the sampled numbers or people. In this case, people 1, 7, 3, 11 and so on get the new treatment, and the rest get the old one.

Another way of doing this is shown in the next three lines of Figure 5.2. If you run `sample` without specifying a size of sample, R will "sample" everyone; that is, it will shuffle the list of individuals into a random order. Then you can pick out the first 10 people to get the new treatment and people 11 through 20 to get the old one, as shown. (These groups are different because I ran `sample` again, and thus used a new bunch of random numbers.)

This idea can be used no matter how many groups you want to divide your individuals into. The end of Figure 5.2 shows how you can use `sample` to randomly divide those individuals 1 through 20 into *four* groups of size 5. First shuffle the entire list, and then pick out the ones you want for each group.

Now, what about if your individuals are (equally) split into males and females, and you want to assign individuals randomly to treatment groups of the same size, with the same number of males and females in each group? This is known in the jargon as "stratified sampling". What you do is first to sample the males and females *separately*, and then combine them into the groups. Let's suppose we now have 30 individuals, 15 females and 15 males, and we want to randomly assign them to 3 treatment groups (of size 10) with the same number (5) of males and females in each group.

Figure 5.3 shows the way. Let's number the females 1 through 15 and the males

16 through 30. Then we shuffle both lists (independently), and glue together the shuffled individuals we want for each group. You can see that we do indeed have 5 females and 5 males in each group.

Another way is to make a data frame with all the combinations of variables you want, and then shuffle it. This is shown in Figure 5.4. The steps are: first create vectors containing the levels of each factor (the treatments, numbered 1 through 3, and the genders). These need to be factors, so we turn them into factors. Then we use `expand.grid` to create a data frame with all $3 \times 2 = 6$ combinations. Now, we have 30 individuals at our disposal, so we have to replicate this data frame 5 times ($6 \times 5 = 30$). The data frame `exp.df2` is the original data frame repeated 5 times. Next, we shuffle the numbers 1 through 30 (the number of rows in `exp.df2`, or the number of individuals we have), and create the actual experimental protocol in `my.df` by taking the rows of `ex.df2` in shuffled order. Since the rows are already shuffled, you just read off the first female on your list to get treatment 2, the second female to get treatment 1, the first male on your list to get treatment 1, and so on.

The row names need not concern you, but you might be able to work out how R constructed them. The single numbers are from the last replication of the data frame produced by `expand.grid`, and the two-part row names are the row of `exp.df` and the replication number.

The factors in an experiment can be of two types: ones that you can control, like "treatment" above, and ones that you cannot control, like "gender". The reason for including ones that you cannot control is that you think they might make a difference, so you are controlling for them: making the groups come out more similar than they otherwise would.

Any number of factors work just the same way here. Use `expand.grid` on all the factors, and "replicate" the resulting data frame as many times as you need to take care of all the individuals you have at your disposal.

I used the word "replicate" on purpose, to look ahead to the next section.

It's actually easier if you have only factors that you can control. Figure 5.5 shows an experiment on two controllable factors `temperature` and `pressure`. We have 8 individuals (presumably machines in this case), and there are $2 \times 2 = 4$ combinations of `temperature` and `pressure`, so we can replicate this twice, shuffle it, and get the design shown. (I'm using the same data frame `design` that I am constructing step by step.) Now you take your 8 machines in the order they come, and assign them to the lines in the final `design`.

The curious thing about my randomization is that you get those repeats of, for example, temperature 70 and pressure 8. This just happened to come out that way; if you do it again, you are unlikely to get those repeats. If this bothers you, you take the `design` from line 3 of Figure 5.5, shuffle it first and *then* replicate

```
> treatments=c("T1","T2","T3")
> gender=c("M","F")
> exp.df=expand.grid(trt=treatments,g=gender)
> exp.df

  trt g
1  T1 M
2  T2 M
3  T3 M
4  T1 F
5  T2 F
6  T3 F

> r=rep(1:6,5)
> exp.df2=exp.df[r,]
> shuf=sample(1:30)
> my.df=exp.df2[shuf,]
> my.df

    trt g
5.4  T2 F
4    T1 F
1.1  T1 M
4.2  T1 F
5.1  T2 F
1.3  T1 M
6    T3 F
5    T2 F
1.2  T1 M
6.2  T3 F
3.4  T3 M
2.1  T2 M
3    T3 M
1.4  T1 M
4.3  T1 F
2.4  T2 M
5.2  T2 F
5.3  T2 F
3.2  T3 M
4.4  T1 F
1    T1 M
3.3  T3 M
4.1  T1 F
2.3  T2 M
6.3  T3 F
6.4  T3 F
6.1  T3 F
3.1  T3 M
2.2  T2 M
2    T2 M
```

Figure 5.4: Creating a randomized data frame

```
> temperature=factor(c(50,70))
> pressure=factor(c(8,10))
> design=expand.grid(temperature,pressure)
> r=rep(1:4,2)
> design=design[r,]
> shuf=sample(1:8)
> design=design[shuf,]
> design

    Var1 Var2
1.1   50    8
2     70    8
2.1   70    8
3     50   10
3.1   50   10
4.1   70   10
4     70   10
1     50    8
```

Figure 5.5: Temperature and pressure experiment

it. This would go through all the combinations of temperature and pressure once (in some randomized order), and then repeat that.

5.2.4 Replicate

The experimental designs in the previous section have a feature in common: repeated measurements *under the same experimental conditions* on different individuals. This is useful, statistically speaking, because different individuals, even under the same conditions, are not likely to give exactly the same results. There is going to be a certain amount of variability among individuals, no matter how careful we are, and repeating things under identical conditions will give us a sense of how big that variability is. This kind of repetition is what replication is. ("Replicate" really means "copy", where what you are "copying" is the experimental conditions; you can't hope to copy the results!)

The other virtue of replication is that by doing it, you end up with more data than you had before. Having more data is a good thing, statistically speaking, because you can expect to get closer to the "true" situation (that you can never know for sure) — for example, with more data, your sample mean is likely to be closer to the population mean (whatever it is). Of course, getting more data costs money, so you have a tradeoff between your budget and the quality of results you are likely to get. It's the old story of "no free lunch!".

Chapter 6

Analysis of standard experimental designs

6.1 One-way ANOVA

The theory is that exercise stresses bones (in a good way) and helps them grow stronger. So an experiment was done on rats to test this. 30 rats were used, and they were randomly assigned to one of three groups, to receive varying amounts of exercise. These groups are usually called **treatments** in Statistics, even if they aren't what you would think of as medical treatments. At the end of the experiment, the rats' bone density was measured (this is the response variable), and the research question was "is the bone density different, on average, between treatment groups?". This is our alternative hypothesis, with the null hypothesis being that there is no difference in bone density between the groups.

OK, so what *were* the treatments here? Amounts of jumping required of the rats (8 jumps per day). The high-jumping group had to jump 60 cm, the low-jumping group had to jump 30 cm, and the no-jumping group didn't have to jump at all. The last group is known in the jargon as a **control group**; this serves as a baseline comparison so that the researchers would know something about bone density in rats that didn't jump at all, and could then get a handle on how much of an effect the jumping had.

Most experiments have a control group of some kind, so that you can see whether the treatment you are really testing has an effect after all. In medical trials, most people react favourably to any kind of thing that looks like a drug (this is called the **placebo effect**) and so these trials routinely include a drug that does nothing, and the research question is "does the new drug do better than placebo?".

The reason for randomization is to ensure that who ends up in which group has nothing to do with anything else that might make a difference. In the jumping rats example, the experimenter might otherwise (consciously or unconsciously) put the healthiest-looking rats in the high-jump group and thereby make the jumping look more effective than it really is.

Here's the way the jumping rats data looks. For some reason the groups are also indicated by number (which we ignore):

```
Control  1 611
Control 1 621
Control 1 614
Control 1 593
Control 1 593
Control 1 653
Control 1 600
Control 1 554
Control 1 603
Control 1 569
Lowjump 2 635
Lowjump 2 605
Lowjump 2 638
Lowjump 2 594
Lowjump 2 599
Lowjump 2 632
Lowjump 2 631
Lowjump 2 588
Lowjump 2 607
Lowjump 2 596
Highjump 3 650
Highjump 3 622
Highjump 3 626
Highjump 3 626
Highjump 3 631
Highjump 3 622
Highjump 3 643
Highjump 3 674
Highjump 3 643
Highjump 3 650
```

The variables are separated by spaces rather than commas, so `read.table` is what we need. Also, we don't have a header row naming the variables, so we'd better say that:

```
> rats=read.table("jumping.txt",header=F)
> head(rats)
```

```
       V1 V2  V3
1 Control  1 611
2 Control  1 621
3 Control  1 614
4 Control  1 593
5 Control  1 593
6 Control  1 653
```

I just discovered the `head` command yesterday. It prints out the first few rows of a data frame. Here it's the same as `rats[1:6,]`, only easier to type.

Note also that since we didn't have any variable names, R created some for us. They're not very mnemonic, though. But we can fix that:

```
> names(rats)
```

```
[1] "V1" "V2" "V3"
```

```
> names(rats)=c("group","group.number","density")
> head(rats)
```

```
    group group.number density
1 Control            1     611
2 Control            1     621
3 Control            1     614
4 Control            1     593
5 Control            1     593
6 Control            1     653
```

Now we're good to go.

One point of view here is that we are predicting the (numeric) variable `density` from the categorical variable `group`. This sounds like a regression (kind of), so we can use `lm`, as indeed we can:

```
> #detach(cars)
> attach(rats)
> rats.lm=lm(density~group,data=rats)
> rats.lm
```

```
Call:
lm(formula = density ~ group, data = rats)

Coefficients:
  (Intercept)  groupHighjump   groupLowjump
        601.1           37.6           11.4
```

This isn't the kind of output we were expecting, but it does make some kind of sense, read the right way. The group that's missing (which, happily, *is* the control group), is used as a baseline; here the intercept is the mean of the control group. Then the "slopes" for the other groups say how the means for those groups compare to the mean of the baseline group. Here, the mean for the high jump group is 37.6 higher than for the no-jump group, and the mean for the low jump group is 11.4 higher than for the no-jump group. But is that significant? We don't know yet.

```
> anova(rats.lm)

Analysis of Variance Table

Response: density
          Df  Sum Sq Mean Sq F value   Pr(>F)
group      2  7433.9  3716.9  7.9778 0.001895 **
Residuals 27 12579.5   465.9
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

This is testing the null hypothesis that the groups all have the same mean, against the alternative that they are not all the same. Here, the null should be rejected (this is a two-star rejection). So there are some differences somewhere. But where? To find out which groups are different from which, we have to do a second stage. But to do that, it's better to do the first stage differently:

```
> rats.aov=aov(density~group,data=rats)
> anova(rats.aov)

Analysis of Variance Table

Response: density
          Df  Sum Sq Mean Sq F value   Pr(>F)
group      2  7433.9  3716.9  7.9778 0.001895 **
Residuals 27 12579.5   465.9
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

To find out which groups differ from which, we need a **multiple comparisons** method. There are lots of them. The problem is that by comparing all the groups with each other, you're (potentially) doing a large number of tests, thus giving yourself a large chance to (possibly incorrectly) reject a null hypothesis that the groups you're comparing have equal means. With three groups, as here, you're doing three comparisons (1 vs 2, 1 vs 3, 2 vs 3). With five groups, you'd be doing 10 comparisons. Thus you get 3 (or 10) chances to make a mistake.

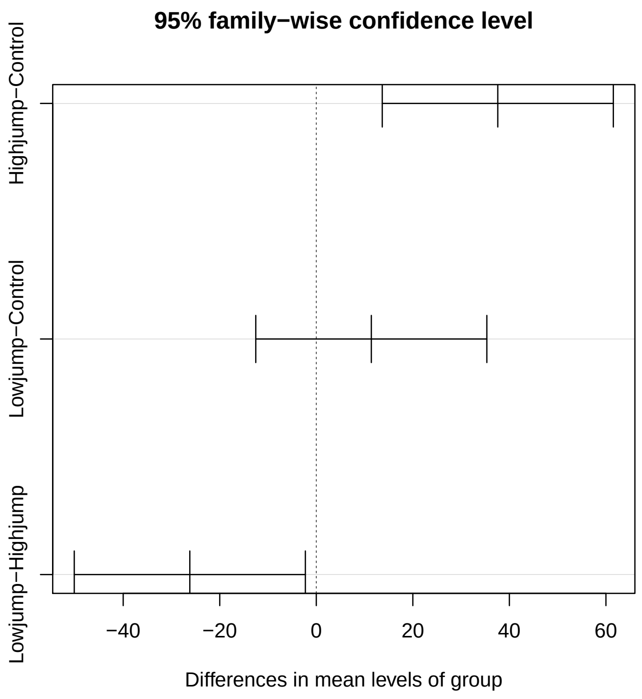
The different multiple comparisons methods allow for this. They organize things so that you get a "familywise error rate" of 0.05, or whatever you want, no matter how many comparisons you're doing. My favourite is Tukey's method, also known as "honestly significant differences" (Tukey liked catchy names). It's based on the idea that if all the group means are really the same, in your data the largest will be a little bigger than the smallest, and you (well, he) can work out how much bigger it's likely to be. Then you say that group means in your data that differ by more than this value are significantly different.

Here's how you make it go in R, having used `aov` first. You can also plot the results, as shown:

```
> rats.tukey=TukeyHSD(rats.aov)
> rats.tukey
```

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = density ~ group, data = rats)

$group
                  diff       lwr       upr     p adj
Highjump-Control  37.6  13.66604 61.533957 0.0016388
Lowjump-Control   11.4 -12.53396 35.333957 0.4744032
Lowjump-Highjump -26.2 -50.13396 -2.266043 0.0297843
```

```
> plot(rats.tukey)
```



The output shows the appropriately adjusted P-values for comparing each pair of groups (in the last column). This shows that the mean for high jumping is significantly different (larger) than for both low jumping and no jumping at all, but that there is no significant difference between low jumping and control.

To the left of the P-values are 95% (by default) confidence intervals for the true difference in means between the groups. We are confident that the mean for high jumping is higher than the mean for control (the interval contains only positive differences in means), and, likewise, we are confident that the mean for low jumping is lower than the mean for high jumping. The interval for low jumping and control includes both positive and negative values, so the comparison could go either way (we don't have enough evidence to decide).

Like `t.test`, `TukeyHSD` can be fed a `conf.level` if you want, say, 90% (0.90) or 99% (0.99) confidence intervals instead.

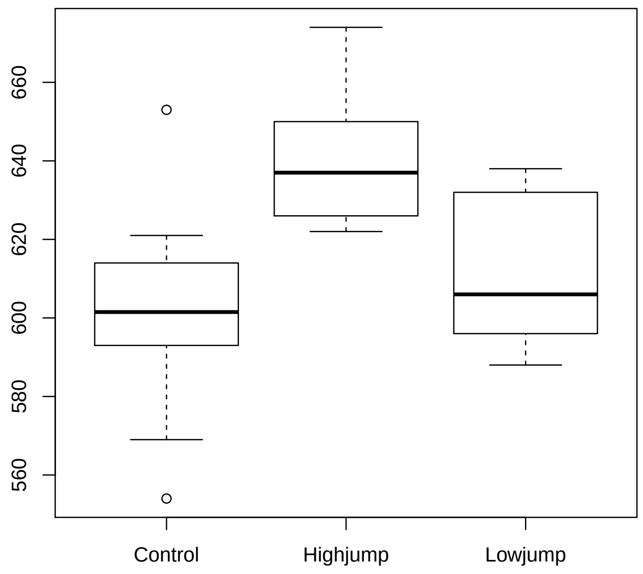
The plot displays those intervals graphically. The dotted line marks a mean difference of zero, so if the interval crosses that line, the difference in means could be zero (we have no evidence of a difference). If the interval is all one side of the zero line, one mean is significantly larger than the other.

The conclusion we draw from this analysis is that it is not the actual fact of jumping that makes a difference. It has to be high jumping rather than low jumping to make a difference. (The conclusion might be that there is a "threshold" level of jumping, or exercise generally, that has an impact on bone density.)

Before we leave this example, we should do some quality control. The major assumption hiding behind ANOVA is that all the groups should have the same spread, even if they have different means. (There is also an assumption of normality, but that is less crucial. Only outliers need to concern us.)

One way to assess that is with side-by-side boxplots, which you can produce like this:

```
> boxplot(density~group,data=rats)
```



The low-jump group has a somewhat higher spread, and the control group has a couple of outliers, including one value above 650. For myself, I would be more concerned about the outliers than the unequal spread; the spreads are never going to be exactly the same.

Numerically, it is often useful to give the means and standard deviations of each group. Way back, I wrote a function called `mystats` that gives these, and we learned how to do this with groups (using `split` and `sapply`):

```
> mystats=function(x)
+   {
+     result=c(length(x),mean(x),sd(x),median(x),IQR(x))
+     names(result)=c("n","Mean","SD","Median","IQR")
+     result
+   }
> mylist=split(density,rats$group)
> mylist
```

```
$Control
 [1] 611 621 614 593 593 653 600 554 603 569

$Highjump
 [1] 650 622 626 626 631 622 643 674 643 650

$Lowjump
 [1] 635 605 638 594 599 632 631 588 607 596
```

```
> sapply(mylist,mystats)
```

```
        Control  Highjump   Lowjump
n       10.0000  10.00000  10.00000
Mean   601.1000 638.70000 612.50000
SD      27.3636  16.59351  19.32902
Median 601.5000 637.00000 606.00000
IQR     20.2500  22.25000  35.00000
```

I can live with those unequal SDs. Note, though, how the outliers in the control group show up: the control group SD is the biggest, even though its interquartile range is the smallest.

If the spreads are really too unequal, then a transformation (Section 6.8) may help. What commonly happens is that as the mean gets larger, the spread gets larger too; that is the case where a transformation is most useful.

From a strategic point of view, the time to do this quality control is right at the beginning, since none of it depends on a model being fitted. (This is unlike regression.) Thus, if you see any problems, you can fix them, or express your reservations about them, before you do any analysis.

Now we're done with the rats:

```
> detach(rats)
```

One more thing: sometimes your data come to you like this, with the groups separately:

```
> Alcohol=c(51,5,19,18,58,50,82,17)
> AB.Soap=c(70,164,88,111,73,119,20,95)
> Soap=c(84,51,110,67,119,108,207,102)
> Water=c(74,135,102,124,105,139,170,87)
```

These data came from a handwashing experiment. The response is bacteria growth, with the groups being different ways of washing hands. The alcohol was an ethanol spray, and the second one is antibacterial soap. A lower amount of bacteria is better!

To do an ANOVA, we have to create a data frame to run `aov` on. This is done by combining the separate vectors of response values into an R list, and then using the function `stack` to put them together into a data frame. The odd-looking construction of the list, with the variable names twice, is to ensure that those variable names make it into the data frame. I had to do a bit of playing around to get this, since the `stack` help page is not especially helpful. It just says "this works for lists", but I had to find out how!

```
> hw.list=list(Alcohol=Alcohol,AB.Soap=AB.Soap,Soap=Soap,Water=Water)
> hw.list


$Alcohol
[1] 51  5 19 18 58 50 82 17

$AB.Soap
[1]  70 164  88 111  73 119  20  95

$Soap
[1]  84  51 110  67 119 108 207 102

$Water
[1]  74 135 102 124 105 139 170  87


> hw.df=stack(hw.list)
> hw.df


   values     ind
1      51 Alcohol
2       5 Alcohol
3      19 Alcohol
```

```
4       18 Alcohol
5       58 Alcohol
6       50 Alcohol
7       82 Alcohol
8       17 Alcohol
9       70 AB.Soap
10     164 AB.Soap
11      88 AB.Soap
12     111 AB.Soap
13      73 AB.Soap
14     119 AB.Soap
15      20 AB.Soap
16      95 AB.Soap
17      84    Soap
18      51    Soap
19     110    Soap
20      67    Soap
21     119    Soap
22     108    Soap
23     207    Soap
24     102    Soap
25      74   Water
26     135   Water
27     102   Water
28     124   Water
29     105   Water
30     139   Water
31     170   Water
32      87   Water
```

First you see how the list comes out, and then, once you have that, there's no great difficulty in using `stack` to make a data frame.

The data frame that comes out has two columns. The first, `values`, is all the response values joined together. The second, `ind`, labels the group that those response values came from. This is exactly what we need for `aov`.

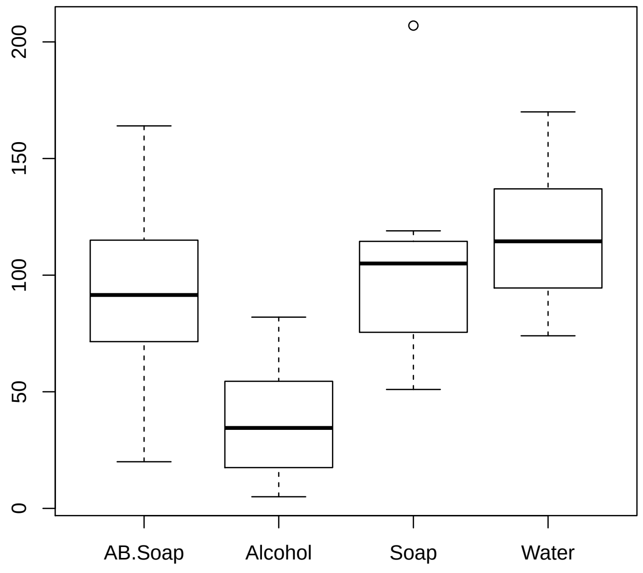
Since we now have the data in the right format, let's speed through an analysis. First, the quality control:

```
> attach(hw.df)
> sapply(hw.list,mystats)
```

```
        Alcohol  AB.Soap      Soap     Water
n       8.00000  8.00000   8.00000   8.00000
```

```
Mean   37.50000 92.50000 106.00000 117.00000
SD     26.55991 41.96257  46.95895  31.13106
Median 34.50000 91.50000 105.00000 114.50000
IQR    35.00000 40.75000  32.50000  37.75000
```

```
> boxplot(values~ind)
```



I cunningly used the list that I created before running `stack`, because you can use `sapply` to run a function (here `mystats`) on all the elements of a list. The group standard deviations could be more similar, though I think I can live with them. I don't think the SD increases with the mean, although the group (Alcohol) with the smallest mean does also have the smallest SD.

The boxplots reveal that the interquartile ranges are very similar indeed, though the Soap group has one clear outlier and the `AB.Soap` group, though it has no outliers, must have high and low values that are very close.

So, with a few reservations, we proceed to the analysis:

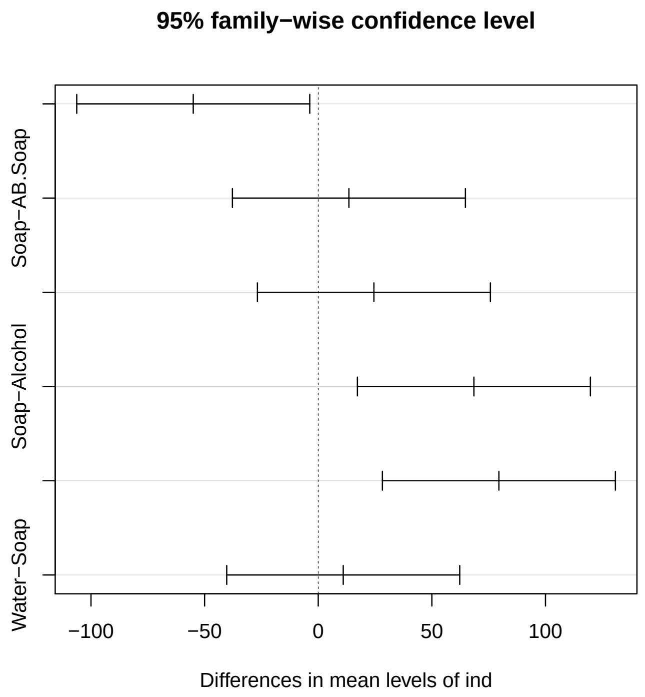
```
> values
```

```
 [1]  51   5  19  18  58  50  82  17  70 164  88 111  73 119  20  95  84  51 110
[20]  67 119 108 207 102  74 135 102 124 105 139 170  87

> ind

 [1] Alcohol Alcohol Alcohol Alcohol Alcohol Alcohol Alcohol Alcohol AB.Soap
[10] AB.Soap AB.Soap AB.Soap AB.Soap AB.Soap AB.Soap AB.Soap Soap    Soap
[19] Soap    Soap    Soap    Soap    Soap    Soap    Water   Water   Water
[28] Water   Water   Water   Water   Water
Levels: AB.Soap Alcohol Soap Water

> hw.aov=aov(values~ind)
> anova(hw.aov)

Analysis of Variance Table

Response: values
          Df Sum Sq Mean Sq F value   Pr(>F)
ind        3  29882  9960.7  7.0636 0.001111 **
Residuals 28  39484  1410.1
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

There are definitely some differences among the forms of handwashing. The boxplots suggest that the alcohol spray is the most effective, but are there any other significant differences? (My guess is not, since the data overall are rather variable and we only have 8 observations per group.)

```
> hw.tukey=TukeyHSD(hw.aov)
> hw.tukey

  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = values ~ ind)

$ind
                 diff        lwr        upr     p adj
Alcohol-AB.Soap -55.0 -106.26415  -3.735849 0.0319648
Soap-AB.Soap     13.5  -37.76415  64.764151 0.8886944
Water-AB.Soap    24.5  -26.76415  75.764151 0.5675942
Soap-Alcohol     68.5   17.23585 119.764151 0.0055672
Water-Alcohol    79.5   28.23585 130.764151 0.0012122
Water-Soap       11.0  -40.26415  62.264151 0.9355196
```

```
> plot(hw.tukey)
```



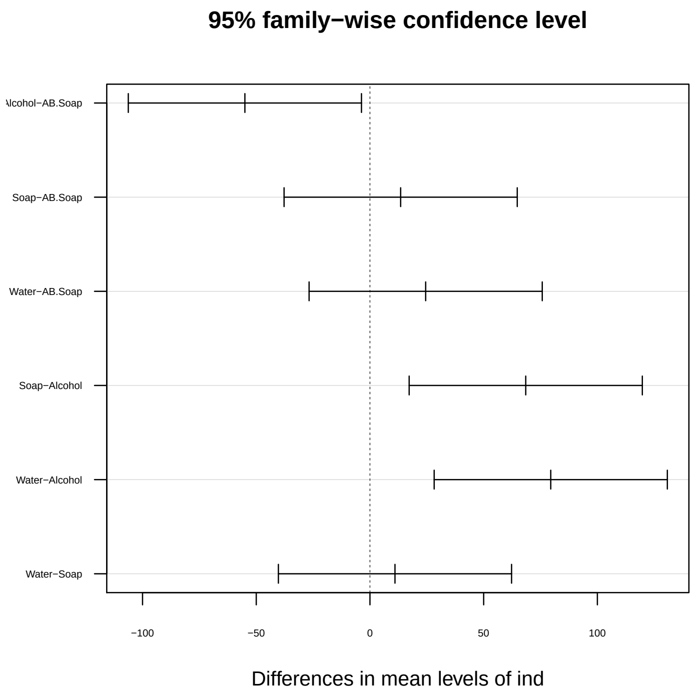
My guess was correct. (Yes, I did make the guess before doing the computations!) Only the three differences involving Alcohol were significant, with the rest being comfortably non-significant. The plot conveys the same information, though it's a bit confusing since three of the labels didn't get printed. They are in the same order as in `hw.tukey`.

This business of disappearing labels bugged me. So I did some investigation, and got them to appear. There is an enormous number of adjustable "graphical parameters". One of them is `las`, which controls how the numbers on the axes get printed. `las=1` means all the axis stuff is horizontal. You might remember `cex` from when we were making plots of the cars' gas mileages by weight, and wanted to label the points by which cars they were. `cex=0.5` makes characters *within* the plot half the regular size. It turns out that there is a corresponding thing for the characters on the axis, called `cex.axis`. So I tried setting that to 0.5 to see what would happen. (I found out all of this from the help for `par`, which describes all the graphical parameters.)

Graphical parameters can be specified on the `plot` line, or beforehand by something like `par(las=1,cex.axis=0.5)`. We did that when we did those four plots on one page for regression.

So here we go:

```
> plot(hw.tukey,las=1,cex.axis=0.5)
> detach(hw.df)
```



Much better. Whether it was worth all that trouble is another question.

6.2 Randomized blocks design

The people, animals or things on which an experiment is run are called **experimental units**. (Usually, if they're people, they are called **subjects** instead — would you like to be called an "experimental unit"?) In the last example, the experimental units were hands; in the bone density example, they were rats. In a one-way analysis, we are assuming that the experimental units are all *a priori* equal; we have no reason to suspect that particular units will be better (in terms of the response) than others.

But imagine this: a study is comparing the effectiveness of four different skin creams for treating a certain skin disease. There are 60 subjects available. But some of them have more severe cases of the disease than others. (You might expect the treatments to be less effective on more severe cases.) So the subjects can be arranged in **blocks**: the 20 most severe cases in block 1, the 20 next most severe in block 2, and the 20 least severe in block 3. Then 5 subjects from each block are randomly chosen to get each of the four skin creams.

This deserves a picture, but I couldn't find one online. I may have to draw one.

The point of all of this is that without the blocking, the most severe cases might all get cream A, which would make cream A look bad (even if it really isn't). The least severe cases might all get cream D, making cream D look good.

But with the blocking, some of the most severe cases are guaranteed to receive *each* of the four creams, and some of the least severe cases are likewise guaranteed to receive each of the creams. So we should have a fair comparison of the creams.

I'm going to make up some data for this. Let's imagine we have just 12 subjects. We have 4 creams and 3 blocks, so that only leaves one observation per cream-block combination. We'll imagine the response variable is a score out of 100 which reflects how well the skin disease has cleared up.

	Skin cream			
Block	A	B	C	D
Most severe	12	14	22	17
Middle	51	55	61	54
Least severe	80	83	92	88

The response variable seems to be all over the place, but let's forget our statistics for a moment and just try to make sense of this. We do best to compare apples with apples, so let's look first at the most severe cases (first row). None of the creams are much good, but C is better than the rest.

Now look at the middling-severe cases. The scores are all higher, but once again C comes out best. Likewise, with the least severe cases, the scores are higher again, but again C comes out best. So cream C ought to be the best overall. We would hope that our analysis would bear this out.

First, the data. Let me put all the response values into a vector first:

```
> yy=c(12,14,22,17,51,55,61,54,80,83,92,88)
> yy

 [1] 12 14 22 17 51 55 61 54 80 83 92 88
```

and then we use a trick to get all the combinations of factor and block. I think if you see it in action you'll be able to work out how to use it:

```
> handcreams=expand.grid(cream=c("A","B","C","D"),severity=c("Most","Middle","Least"))
> handcreams

   cream severity
1      A     Most
2      B     Most
3      C     Most
4      D     Most
5      A   Middle
6      B   Middle
7      C   Middle
8      D   Middle
9      A    Least
10     B    Least
11     C    Least
12     D    Least
```

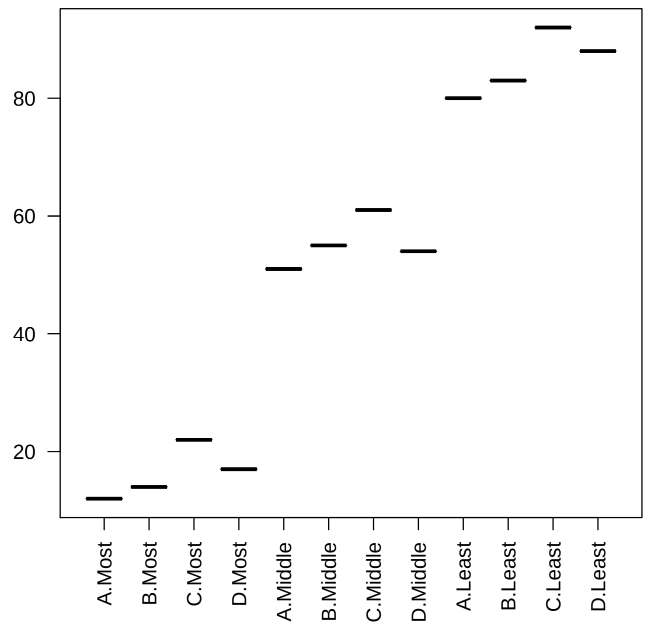
This is a data frame. Let's glue our response onto it:

```
> handcreams$bacteria=yy
> handcreams

   cream severity bacteria
1      A     Most       12
2      B     Most       14
3      C     Most       22
4      D     Most       17
5      A   Middle       51
6      B   Middle       55
7      C   Middle       61
8      D   Middle       54
9      A    Least       80
10     B    Least       83
11     C    Least       92
12     D    Least       88
```

and you can check that the response values did indeed get matched up to the right combinations of cream and severity.

Now we can attach the data frame and go ahead and do our analysis. We'll start with some quality control which will look a bit silly because there's only one observation per cream-severity combination, and they are not really boxplots at all. As with my Tukey plot earlier, some of the group labels disappeared, and the fix is the same as before: adjust `las` and `cex`. This time I want the labels perpendicular to the axis, so I set `las` accordingly:

```
> attach(handcreams)
> boxplot(bacteria~cream*severity,las=2,cex=0.5)
```



Take a look at the one-two-three-four patterns. Within each severity level, cream C is highest and A lowest (no matter what the overall level). Also, the "least" values are consistently higher than the "middle" values which are consistently higher than the "most" values.

This consistency of pattern means that it is sensible to talk about a "cream effect" regardless of severity, and sensible to talk about a "severity effect" regardless of cream. This property of the data is called **additivity**, and is an assumption hidiing behind this kind of ANOVA.

Notice how the model formula in `aov` below looks like a multiple regression.

```
> handcream.aov=aov(bacteria~cream+severity)
> anova(handcream.aov)

Analysis of Variance Table

Response: bacteria
```

```
          Df Sum Sq Mean Sq  F value    Pr(>F)
cream      3  182.9    61.0   32.279 0.0004248 ***
severity   2 9708.7  4854.3 2569.941 1.585e-09 ***
Residuals  6   11.3     1.9
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

There is a strongly significant effect of `severity` (2nd line of table). This is no surprise, since it's the reason we made blocks in the first place. Generally speaking, we *expect* there to be a differences among blocks, so we shouldn't get too excited about that.

The interesting thing is the test for differences among creams. This is strongly significant too. The interpretation is as in multiple regression: *after you've accounted for differences among blocks*, there is a significant difference between creams. If you *don't* allow for differences among blocks, like this:

```
> handcream.aov2=aov(bacteria~cream)
> anova(handcream.aov2)

Analysis of Variance Table

Response: bacteria
          Df Sum Sq Mean Sq F value Pr(>F)
cream      3  182.9   60.97  0.0502 0.9841
Residuals  8 9720.0 1215.00
```

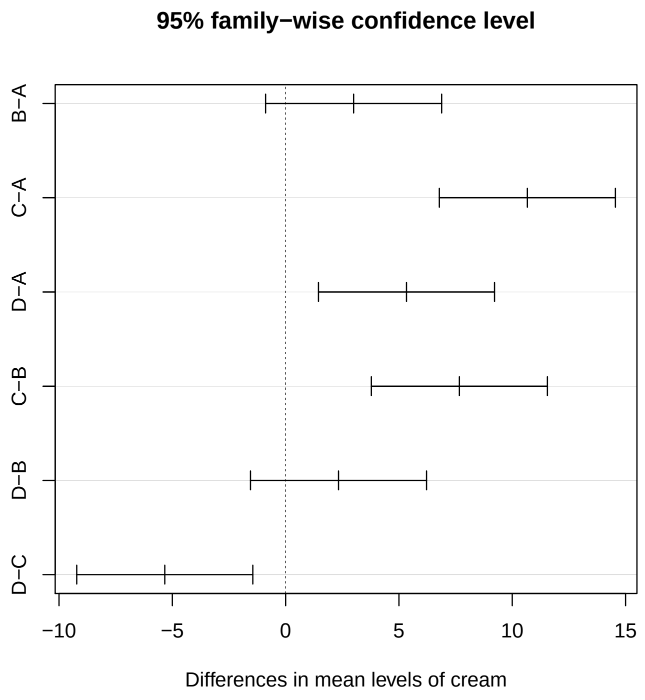
you don't get any significance at all. The moral of the story is, if you think it'll make a difference, put it in the model (even if you're not primarily interested in it).

So there are differences among creams. Let's see if we can figure out what they are. Note the use of `which` to tell `TukeyHSD` that we only want to look for differences among creams. Differences among blocks are expected and not interesting.

```
> handcream.tukey=TukeyHSD(handcream.aov,which="cream")
> handcream.tukey


  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = bacteria ~ cream + severity)
```

```
$cream
          diff       lwr       upr     p adj
B-A  3.000000 -0.884619  6.884619 0.1274284
C-A 10.666667  6.782048 14.551286 0.0003227
D-A  5.333333  1.448714  9.217952 0.0124296
C-B  7.666667  3.782048 11.551286 0.0019796
D-B  2.333333 -1.551286  6.217952 0.2597878
D-C -5.333333 -9.217952 -1.448714 0.0124296
```

```
> plot(handcream.tukey)
```



This one takes a little untangling. Cream C is significantly better than all the others. D is better than A but not B. A and B are not significantly different either. In the old days (when I was an undergrad), people used to draw a "lines" picture to summarize the differences, like this:

```
Cream   A   B   D   C
            -----
        -----
```

This means that cream B occupies a kind of middle ground between D and A, and we don't have enough data to resolve the situation unambiguously. But I think we've done pretty well out of only 12 observations.

Finally,

```
> detach(handcreams)
```

Another example of a randomized blocks design is our matched-pairs example of deer leg lengths, laid out in Figure 6.1. We have to re-organize the data first. The original data frame has the foreleg lengths in one column (for the 10 deer) and the hindleg lengths in another. So we have to arrange all the leg lengths in one column, and then have a second column that says whether it was a foreleg or a hindleg. That's a job for `stack`. The problem with `stack` is that it gives the columns silly names, so we'll give them more reasonable ones. The first column is the leg length, and the second is which leg it was. We also need to know which deer we're talking about. The data are the foreleg measurements for the 10 deer, then the hindleg measurements. The `rep` command here takes the list 1 through 10, and then that is repeated enough times (twice) to make `indiv` have length 20. Then we glue that to our data frame `s`, and take a look at it.

Next, we take a look at the randomized blocks ANOVA for our re-formatted data frame, as in Figure 6.2. The P-value for `leg` is identical to the P-value for the matched pairs t test. This is not a coincidence; both analyses are testing for a difference in mean leg length, allowing for possible differences among deer. You can do the analysis either way.

6.3 Two-way ANOVA

Let's proceed right away to an example:

Nitrogen dioxide is a known pollutant, but its effects are not well known. A study was carried out of protein leakage in the lungs of mice exposed to nitrogen dioxide for 10, 12, and 14 days. Half the mice were exposed to the nitrogen dioxide; the other half were not, and served as a control group. The response variable is the percent of serum fluorescence, with high values indicating more protein leakage. One third of the animals in the exposed and control groups had their serum fluorescence measured at 10, 12 and 14 days.

I typed the data into a file and read it in like this:

```
> no2=read.table("fluor.txt",header=T)
> no2
> deer=read.csv("deer.csv",header=T)
> deer

   Foreleg Hindleg
1      142     138
2      140     136
3      144     147
4      144     139
5      142     143
6      146     141
7      149     143
8      150     145
9      142     136
10     148     146

> s=stack(list(Foreleg=deer$Foreleg,Hindleg=deer$Hindleg))
> names(s)

[1] "values" "ind"

> names(s)[1]="length"
> names(s)[2]="leg"
> indiv=rep(1:10,length=20)
> s$indiv=indiv
> s

   length     leg indiv
1     142 Foreleg     1
2     140 Foreleg     2
3     144 Foreleg     3
4     144 Foreleg     4
5     142 Foreleg     5
6     146 Foreleg     6
7     149 Foreleg     7
8     150 Foreleg     8
9     142 Foreleg     9
10    148 Foreleg    10
11    138 Hindleg     1
12    136 Hindleg     2
13    147 Hindleg     3
14    139 Hindleg     4
15    143 Hindleg     5
16    141 Hindleg     6
17    143 Hindleg     7
18    145 Hindleg     8
19    136 Hindleg     9
20    146 Hindleg    10
```

Figure 6.1: Organizing the deer data

```
> s.aov=aov(length~leg+indiv,data=s)
> anova(s.aov)

Analysis of Variance Table

Response: length
          Df  Sum Sq Mean Sq F value  Pr(>F)
leg        1  54.450  54.450  4.7136 0.04437 *
indiv      1  54.123  54.123  4.6853 0.04494 *
Residuals 17 196.377  11.552
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> deer.t=t.test(deer$Foreleg,deer$Hindleg,paired=T)
> deer.t

        Paired t-test

data:  deer$Foreleg and deer$Hindleg
t = 3.4138, df = 9, p-value = 0.007703
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 1.113248 5.486752
sample estimates:
mean of the differences
                    3.3
```

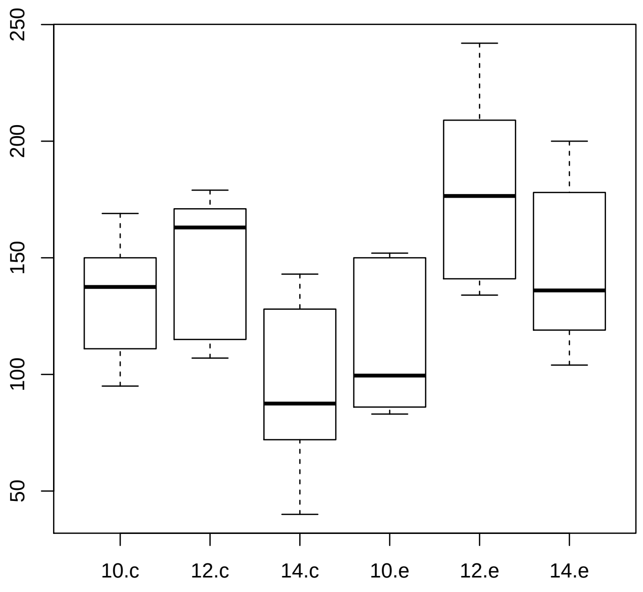
Figure 6.2: Randomized blocks ANOVA for deer data

```
   group days fluor
1      c   10   143
2      c   12   179
3      c   14    76
4      c   10   169
5      c   12   160
6      c   14    40
7      c   10    95
8      c   12   107
9      c   14    99
10     c   10   111
11     c   12   115
12     c   14    72
13     c   10   132
14     c   10   150
15     c   12   171
16     c   12   166
17     c   14   143
18     c   14   128
19     e   10   152
20     e   10    83
21     e   10    91
22     e   10    86
23     e   10   150
24     e   10   108
25     e   12   141
26     e   12   152
27     e   12   201
28     e   12   242
29     e   12   209
30     e   12   134
31     e   14   119
32     e   14   104
33     e   14   125
34     e   14   147
35     e   14   200
36     e   14   178
```

```
> attach(no2)
```

A plot to kick things off. With two factors, we do boxplots of fluorescence against the exposure/group combinations. Note the syntax to get the combinations. This syntax is going to appear again later. It's the same syntax we used to get the hand cream "boxplots".

```
> rm(days)
> boxplot(fluor~days*group,data=no2)
```



Let's try to compare like with like, group first. For the control group (on the left), fluorescence goes up a bit between 10 and 12 days, then down a lot between 12 and 14. For the exposed group, fluorescence goes up a lot between 10 and 12 days, then down a bit between 12 and 14. In both cases, 12 days is the highest, but for one group 10 is the lowest and for the other 14.

Turning it around: comparing the days for each group, at 10 days the fluorescence is higher for the control group, but at 12 and 14 days, the fluorescence is slightly higher for the exposed group.

We don't have anything like the consistency of pattern we saw with the hand creams. Additivity doesn't seem to work; which number of days gives the lowest fluorescence depends on which group you're in.

When additivity fails, you have to look at the **interaction** between the two factors. In fact, any time you have a two-factor ANOVA with more than one observation per combination, you fit an interaction and then throw it away if it's not significant. Here's what we get here. I have to do some shenanigans first, in that `days` is actually a numeric variable; when we come to do Tukey, we need an actual categorical factor in there, which `d` is:

```
> d=factor(days)
> d


 [1] 10 12 14 10 12 14 10 12 14 10 12 14 10 10 12 12 14 14 10 10 10 10 10 10 12
[26] 12 12 12 12 12 14 14 14 14 14 14
Levels: 10 12 14


> fluor.aov=aov(fluor~d*group,data=no2)
> anova(fluor.aov)


Analysis of Variance Table

Response: fluor
          Df Sum Sq Mean Sq F value   Pr(>F)
d          2  15464  7731.8  6.2584 0.005351 **
group      1   3721  3721.0  3.0120 0.092916 .
d:group    2   8686  4343.1  3.5155 0.042495 *
Residuals 30  37062  1235.4
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

The interaction is significant (3rd line of the table), so the fluorescence depends not just on group or days, but on the combination of them.

Before we go to Tukey, let me describe R's notation for ANOVA. Suppose we have two factors `a` and `b`. Then `a+b` describes the model with "main effects" for `a` and `b` only, as in randomized blocks. In that case, we are assuming that `a` and `b` act additively: the effect of `a` is the same over all levels of `b`. Go back and look at the "boxplot" for the randomized blocks example, where you see that the creams come out in (more or less) the same order whether the severity was. This is additivity, and for the fluorescence data we don't have that. If things look additive, you can fit a model like this: `response=a+b`.

If not, you need to include the interaction. In R terms, the interaction is called `a:b`, but you won't often use this notation, because the correct procedure is to fit the main effects *and* the interaction. The notation for that is `a*b`. (If you've ever used SAS, you'll probably find that confusing, by the way.) So you can fit an ANOVA with interaction either as `response~a*b` or `response~a+b+a:b`.

Now, when it comes to Tukey, you need the bit of the output that compares all the factor-level combinations, which in our case is the bit labelled `d:group` at the end of the output below.

```
> fluor.tukey=TukeyHSD(fluor.aov)
```

Since we're ignoring part of the Tukey output, we can just ask for the bit we want as below, with the quotes being necessary as : has special meaning to R:

```
> fluor.tukey$"d:group"
```

```
                diff         lwr        upr       p adj
12:c-10:c  16.333333  -45.389578  78.056245 0.964396970
14:c-10:c -40.333333 -102.056245  21.389578 0.372626293
10:e-10:c -21.666667  -83.389578  40.056245 0.890203500
12:e-10:c  46.500000  -15.222911 108.222911 0.228832871
14:e-10:c  12.166667  -49.556245  73.889578 0.990264722
14:c-12:c -56.666667 -118.389578   5.056245 0.086447851
10:e-12:c -38.000000  -99.722911  23.722911 0.437597887
12:e-12:c  30.166667  -31.556245  91.889578 0.675065824
14:e-12:c  -4.166667  -65.889578  57.556245 0.999943947
10:e-14:c  18.666667  -43.056245  80.389578 0.938281125
12:e-14:c  86.833333   25.110422 148.556245 0.002227027
14:e-14:c  52.500000   -9.222911 114.222911 0.131781533
12:e-10:e  68.166667    6.443755 129.889578 0.023727072
14:e-10:e  33.833333  -27.889578  95.556245 0.562783705
14:e-12:e -34.333333  -96.056245  27.389578 0.547401314
```

```
> tapply(fluor,list(no2$d,no2$group),mean)
```

```
          c        e
10 133.3333 111.6667
12 149.6667 179.8333
14  93.0000 145.5000
```

There is typically rather a lot of output to study from one of these. Here, we have 2 groups and 3 numbers of days, so there are $2 \times 3 = 6$ combinations (the 6 groups we saw on the boxplot), and $6(6-1)/2 = 15$ pairs of groups to compare.

The `tapply` thing at the bottom says "for the variable `fluor`, make me a table cross-classified by `d` and `group` containing the means."

If you cast your eye down the Tukey output, you'll see only 2 P-values less than 0.05: the comparison between `12:e` and `14:c` (that is, fluorescence at 12 days for the exposed group and fluorescence at 14 days for the control group), and also the comparison between `12:e` and `10:e`. (Notice how R is constructing names for the groups using `:`.)

What does that mean? Look at the output from `tapply`. `12:e` has the highest mean of all the groups, and `14:c` and `10:e` have the lowest means of all the groups. Only these differences are significant; all the other differences between groups could be chance.

How you make sense of that is, as we statisticians like to say, "subject-matter dependent", meaning "it's your problem, not mine!"

Finally,

```
> detach(no2)
```

Let me show you now how to handle a case where the interaction is not significant. It'll end up looking a lot like the hand-creams randomized blocks example we did.

This is a study of using "scaffolds" made of extra-cellular material to repair serious wounds. Three types of scaffolds were tested (on mice). The response variable was percent glucose phosphated isomerase in the cells in the region of the wound, a higher value being better. The response variable, called `gpi` in the analyses below, was measured 2, 4 and 8 weeks after the tissue repair. This required a different mouse for each number of weeks. Three mice were tested at each combination of scaffold type (labelled `material` below) and number of weeks, for a total of 27 mice.

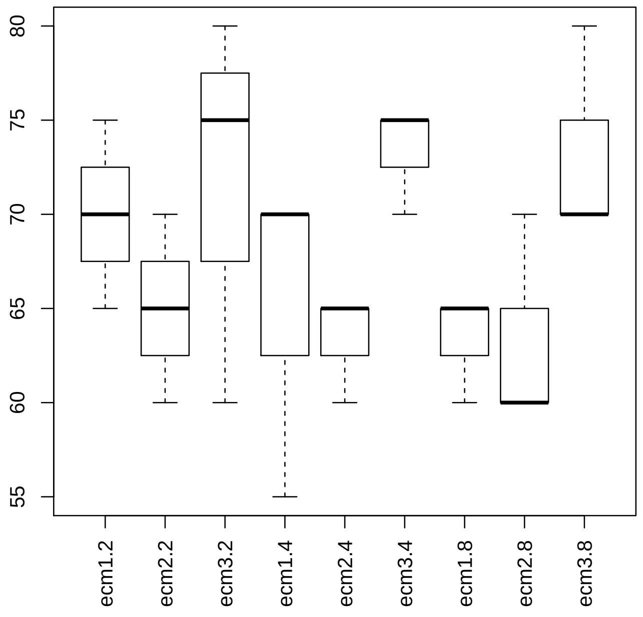
The first step is to read in the data and make some boxplots. `Days` in the data file is a number, so we want to create a factor out of it, which I called `df`. I also had to do the `las` thing to get all the labels to show up:

```
> scaffold=read.table("scaffold.txt",header=T)
> scaffold$df=factor(scaffold$days)
> scaffold
```

```
   material days gpi df
1      ecm1    2  70  2
2      ecm1    2  75  2
3      ecm1    2  65  2
4      ecm1    4  55  4
5      ecm1    4  70  4
6      ecm1    4  70  4
7      ecm1    8  60  8
8      ecm1    8  65  8
9      ecm1    8  65  8
10     ecm2    2  60  2
11     ecm2    2  65  2
```

```
12     ecm2   2  70  2
13     ecm2   4  60  4
14     ecm2   4  65  4
15     ecm2   4  65  4
16     ecm2   8  60  8
17     ecm2   8  70  8
18     ecm2   8  60  8
19     ecm3   2  80  2
20     ecm3   2  60  2
21     ecm3   2  75  2
22     ecm3   4  75  4
23     ecm3   4  70  4
24     ecm3   4  75  4
25     ecm3   8  70  8
26     ecm3   8  80  8
27     ecm3   8  70  8
```

```
> attach(scaffold)
> boxplot(gpi~material*df,las=3)
```



Some of the boxplots look odd because each one is based on only three observations. But, given that, additivity looks pretty good: for any number of days, `ecm3` is highest and `ecm2` is lowest. Also, for each of the materials, the `gpi` for 2 and 4 days is about the same, and the one for 8 days is less. Also, with only three observations per combo, the equal-group-SD assumption is hard to assess.

Anyway, we wouldn't expect the interaction to be significant. Are we right?

```
> scaffold1.aov=aov(gpi~material*df)
> anova(scaffold1.aov)

Analysis of Variance Table

Response: gpi
            Df Sum Sq Mean Sq F value  Pr(>F)
material     2 385.19 192.593  5.3333 0.01517 *
df           2  24.07  12.037  0.3333 0.72086
material:df  4  59.26  14.815  0.4103 0.79891
Residuals   18 650.00  36.111
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Very much so. In fact, it doesn't look as if there's an effect of days either. So let's take out the interaction and think about taking out `df` as well:

```
> scaffold2.aov=aov(gpi~material+df)
> anova(scaffold2.aov)

Analysis of Variance Table

Response: gpi
          Df Sum Sq Mean Sq F value   Pr(>F)
material   2 385.19 192.593  5.9739 0.008467 **
df         2  24.07  12.037  0.3734 0.692691
Residuals 22 709.26  32.239
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

OK, so `df` can come out too. What actually happens is that its sum of squares and degrees of freedom get transferred to the Residuals line, so we end up with a better test for materials, both in the ANOVA and in the Tukey that follows.

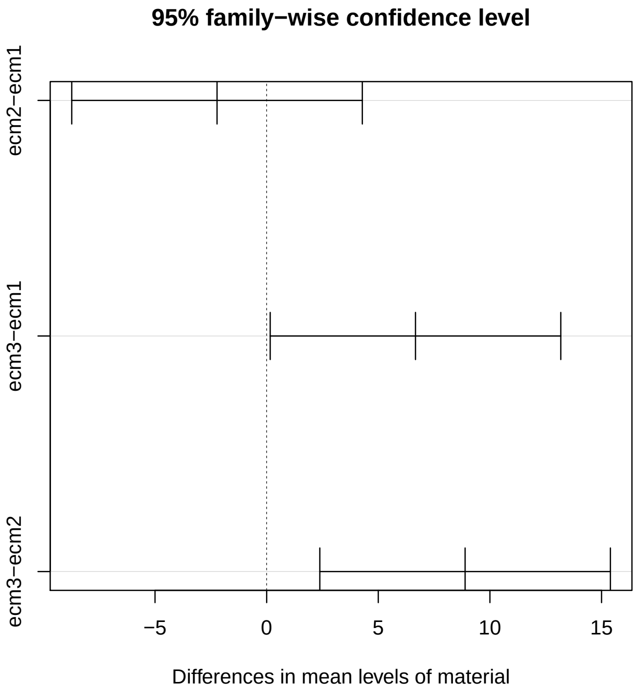
```
> scaffold3.aov=aov(gpi~material)
> anova(scaffold3.aov)
```

```
Analysis of Variance Table

Response: gpi
          Df Sum Sq Mean Sq F value   Pr(>F)
material   2 385.19 192.593   6.303 0.006308 **
Residuals 24 733.33  30.556
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

```
> scaffold3.tukey=TukeyHSD(scaffold3.aov)
> scaffold3.tukey
```

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = gpi ~ material)

$material
               diff        lwr       upr     p adj
ecm2-ecm1 -2.222222 -8.7296194  4.285175 0.6744928
ecm3-ecm1  6.666667  0.1592695 13.174064 0.0439208
ecm3-ecm2  8.888889  2.3814917 15.396286 0.0062543
```

```
> plot(scaffold3.tukey)
```



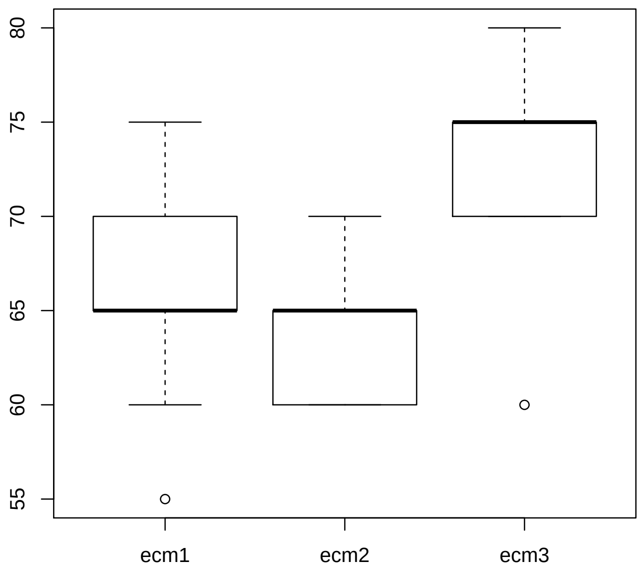
So we end up concluding that there is a significant difference among materials (from the ANOVA). The nature of that difference is perhaps best seen from the Tukey plot: there's no significant difference between `ecm1` and `ecm2`, but `ecm3` is better than both, if only marginally significant against `ecm1`.

What we've come down to is a one-way ANOVA with 9 observations per `material` group (since `days` had no significant effect), so maybe now we can assess our equal-SD assumption by drawing boxplots just by `material` and calculating the SDs by group:

```
> tapply(gpi,list(material),sd)
```

```
    ecm1     ecm2     ecm3
6.009252 4.166667 6.180165
```

```
> boxplot(gpi~material)
```



The SDs look pretty similar, with that of `ecm2` being smallest, which is what you see from the boxplot also. There is a tiny suspicion that groups with larger means also have larger SDs, but I'm not going to worry about that.

Tidying up after ourselves:

```
> detach(scaffold)
```

6.4 2^k factorials and fractional factorials

6.4.1 2^k factorial designs

In industrial settings (and others as well), there can be lots of factors that might affect a response, but typically only a few of them actually do. Also, with many factors, there are in principle a lot of interactions to worry about, but typically only a few of them are actually relevant.

Another thing we've seen is that if the factors have a lot of levels, a lot of data needs to be collected. One way of limiting the data required is to insist that each of the k factors has only two levels, "low" and "high" if you will. Since the

```
> hl=c("+","-")
> des=expand.grid(A=hl,B=hl,C=hl)
> des

  A B C
1 + + +
2 - + +
3 + - +
4 - - +
5 + + -
6 - + -
7 + - -
8 - - -

> r=sample(1:8)
> des[r,]

  A B C
1 + + +
7 + - -
6 - + -
5 + + -
3 + - +
2 - + +
4 - - +
8 - - -
```

Figure 6.3: Designing a 2^3 factorial experiment

levels of the factors are (usually) under our control, this is something that can be arranged. The advantage of this is that you don't need *too* much data: for example, with 3 factors at two levels each, you have $2^3 = 8$ observations.

Designing one of these things contains ideas we've seen before. The example in Figure 5.5 is actually a (replicated) 2^2 factorial experiment, since temperature and pressure there have only two possible values.

The key is the use of `expand.grid` to generate all possible combinations of factor levels. Let's design a 2^3 factorial design (*three* factors, each at two levels. We'll call the factors A, B and C, and we'll suppose each one has a "high" level, denoted "+", and a low level, "-".

The design process is shown in Figure 6.3. First, we set up a variable containing the levels. Then we do all combinations of those levels, naming the factors A, B and C. The actual design is shown next. But before we actually use it, we'll want to randomize the order. The `sample` line shuffles the numbers 1 through

8 (since there are 8 observations that need to be collected) and then we list the rows of `des` shuffled, using the variable `r`. If you wanted to, you could replicate this, eg. by using the same shuffling again.

An unreplicated design like the one in Figure 6.3 allows you to assess all the main effects and interactions up to but *not* including the highets-order one. Thus, in the design of Figure 6.3, we can estimate `A, B, C` main effects, `A:B, A:C, B:C` interactions, but not the `A:B:C` interaction. To get any tests at all, we have to assume that the `A:B:C` interaction is zero, and use its sum of squares for error. Then we can test anything else.

If you replicate the design (eg. do Figure 6.3 twice), you can estimate all the interactions. But if we are in the exploratory stage, probably the highest order interactions (and maybe even some of the main effects) will be zero.

Let's see an example. This is an engineering experiment (from `http://www.itl.nist.gov/div898/handb`
to assess the influence of three factors, Pressure, Speed and Force, on a production tool, with the aim of producing the most uniform product. (I don't know how "product uniformity" was measured, but that's what the response is). Each of the three factors has a "high" and a "low" setting, the details of which we won't bother with here, because they won't affect the analysis. A replicated 2^3 design was used, with two observations taken at each setting of the factors. The analysis is shown in Figure 6.4. (In the actual experiment, the order of the settings was randomized, as in Figure 6.3.)

Let's go through Figure 6.4. First we remind ourselves of the variable we created to symbolize "high" and "low". Then we use this to create a 2^3 factorial design for the three variables `Pressure`, `Speed` and `Force`. Now, we actually wanted to replicate this, so the creation of `eng.design` takes rows 1 through 8 of `d1` (ie. all of it), and glues another copy of rows 1 through 8 onto the bottom. (The function `rbind` could also be used). `dim` is just to verify that `eng.design` has the right number of rows and columns, which it does: 16 rows (observations), 3 columns (variables). `yy` is the actual data, copied from the website (and checked several times to make sure that each value was going with the right combination of the three factors). I glue this onto the data frame, in the variable `response`.

Now comes the actual analysis. (All that work to set up the data!) You could `attach` the data frame, or you can do as I did here, which is to specify `data=` inside `aov`, which means "get the variables needed for the model formula from this data frame". Finally, the ANOVA table shows that the three main effects are strongly significant. The three-way interaction is not, and it appears that none of the two-way interactions are either. Let's take them all out. (There is a mathematical property called "orthogonality" happening here which means that the sums of squares for the factors you remove just get moved into "error", so that the sums of squares and mean squares for the factors you keep don't change. But the error (residual) sum of squares *does* change, so the P-values for the tests of the factors you keep will change.)

```
> hl

[1] "+" "-"

> d1=expand.grid(Pressure=hl,Speed=hl,Force=hl)
> d1

  Pressure Speed Force
1        +     +     +
2        -     +     +
3        +     -     +
4        -     -     +
5        +     +     -
6        -     +     -
7        +     -     -
8        -     -     -

> eng.design=d1[c(1:8,1:8),]
> dim(eng.design)

[1] 16  3

> yy=c(-3,0,1,2,-1,2,1,6,-1,-1,0,3,0,1,1,5)
> eng.design$response=yy
> eng.design.1=aov(response~Pressure*Speed*Force,data=eng.design)
> anova(eng.design.1)

Analysis of Variance Table

Response: response
                     Df Sum Sq Mean Sq F value    Pr(>F)
Pressure              1  25.00  25.000    40.0 0.0002267 ***
Speed                 1  30.25  30.250    48.4 0.0001176 ***
Force                 1  12.25  12.250    19.6 0.0022053 **
Pressure:Speed        1   2.25   2.250     3.6 0.0943498 .
Pressure:Force        1   2.25   2.250     3.6 0.0943498 .
Speed:Force           1   0.00   0.000     0.0 1.0000000
Pressure:Speed:Force  1   1.00   1.000     1.6 0.2415040
Residuals             8   5.00   0.625
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 6.4: Analysis of 2^3 factorial

```
> eng.design.2=aov(response~Pressure+Speed+Force,data=eng.design)
> anova(eng.design.2)

Analysis of Variance Table

Response: response
          Df Sum Sq Mean Sq F value    Pr(>F)
Pressure   1  25.00  25.000  28.571  0.000175 ***
Speed      1  30.25  30.250  34.571 7.484e-05 ***
Force      1  12.25  12.250  14.000  0.002813 **
Residuals 12  10.50   0.875
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> attach(eng.design)
> tapply(response,Pressure,mean)

    +     -
-0.25  2.25

> tapply(response,Speed,mean)

     +      -
-0.375  2.375

> tapply(response,Force,mean)

    +     -
0.125 1.875

> tapply(response,list(Pressure,Speed),mean)

      +    -
+ -1.25 0.75
-  0.50 4.00

> detach(eng.design)
```

Figure 6.5: Analysis 2 of 2^3 factorial

Figure 6.5 shows the analysis with just the main effects included. All three of them are very strongly significant.

Since none of the interactions were significant, we can look at tables of means for each of the factors individually. That's what those first three `tapply` commands are doing. In each case, we see that the "low" setting produces a larger mean.

If any of the interactions had been significant, we would have had to find combination means, as exemplified in the last `tapply`. This gives means for all combinations of `Pressure` (rows) and `Speed` (columns). This example shows that the mean is higher by about 2 for the low setting of `Pressure` compared to the high setting, regardless of what `Speed` is, and the mean is 2-and-a-bit higher for the low setting of `Speed` compared to the high setting, almost regardless of of what `Pressure` is. (This interaction had a P-value of about 0.10, which is why the change in means isn't more consistent.)

Another way of designing a factorial experiment is via use of the function `FrF2` from the package of the same name. Even though it was designed for fractional factorial designs (Section 6.4.2), it can be used for complete factorial designs as well.

6.4.2 Fractional factorial designs

Sometimes the number of factors (even with only two levels each) will require you to collect too much data, more than your budget will allow. For example, with 6 factors, you'd have $2^6 = 64$ observations to collect. This might be too many, especially in the early stages of an investigation when several of those factors might not even be important. But you might be willing to collect half (32) or a quarter (16) as many observations as that. The question is, what does this mean you have to give up?

I mentioned at the end of the previous section that (unless you replicate) you can't test the highest-order interaction for significance. Let's think about the 2^3 design. In that case, you can't test `A:B:C`; it is "confounded with error" in that unless you assume this interaction is zero, you can't test anything.

Now, let's consider a half-fraction of the design in 6.3. What we're going to do is to count up the number of plus signs in each row of `des`, and pick out just those rows where there's an odd number of them. There are four, rows 1, 4, 6, and 7. Now, think of `+` as meaning $+1$ and `-` as meaning -1, and look at this design, shown in Figure 6.6. If you multiply the "numbers" for `B` and `C` together in Figure 6.6, you always get the same thing as `A`. But in the original design, Figure 6.3, sometimes they'll be the same and sometimes they'll be different. What this means is that, in the half-fraction Figure 6.6, you *can't distinguish* the effects of `A` and `B:C`. So you would assume that the `B:C` interaction is zero and that when you test for `A`, it *really is* `A` and not `B:C`. So in a fractional

```
> odd=c(1,4,6,7)
> des[odd,]

  A B C
1 + + +
4 - - +
6 - + -
7 + - -
```

Figure 6.6: Half-fraction of 2^3 design

factorial design like this half-fraction of a 2^3 factorial design, what you give up is the ability to test higher-order interactions, so that you have to assume they are zero.

It would be nice to generate a design more automatically than looking to see whether there is an odd or even number of plus signs. Fortunately, there is a package `FrF2` that will do just this. As usual, you'll need to install it first. Figure 6.7 illustrates how it works.

The basic operation is the first `FrF2` line. You feed in two things: the *second* one is the number k of factors you have (3, as earlier), and the *first* one is the number of observations you want to have to collect. This needs to be a suitable fraction of 2^k; in this case it is half of it. The design comes out with the factors labelled A, B, C and so on, and the levels of the factors labelled 1 and -1. Things are configurable; the second line shows how you name the levels of the factors (the same names for all the factors); the third line shows how you give the factors names different from A, B and C, and the fourth line shows that if you've given the factors names, you don't need to tell `FrF2` how many of them there are.

At the end, I've illustrated how you would randomize the rows of one of these designs. It's the same technique as before: figure out how many rows n you have, shuffle the numbers from 1 to n, and use that vector of shuffled numbers to pick out the rows you want.

Let's look at an example: a statistics class conducted an experiment to see how different settings on a catapult would affect the distance a projectile (a plastic golf ball) would travel. The following description of the five factors was taken from the website `http://www.itl.nist.gov/div898/handbook/pri/section4/pri472.htm`. I have modified the experimental design (and eliminated some of the data) so as to make it a fractional factorial design as we have described it.

- Factor 1 = band height (height of the pivot point for the rubber bands — levels were 2.25 and 4.75 inches)

```
> library(FrF2)
> FrF2(4,3)

   A  B  C
1 -1 -1  1
2  1  1  1
3  1 -1 -1
4 -1  1 -1
class=design, type= FrF2

> FrF2(4,3,default.levels=c("low","high"))

     A    B    C
1 high high high
2  low high  low
3 high  low  low
4  low  low high
class=design, type= FrF2

> FrF2(4,3,factor.names=c("Temperature","Pressure","Smoke"),default.levels=c("low","high"))

  Temperature Pressure Smoke
1         low     high   low
2         low      low  high
3        high     high  high
4        high      low   low
class=design, type= FrF2

> design=FrF2(4,factor.names=c("Temperature","Pressure","Smoke"),default.levels=c("low","high"))
> design

  Temperature Pressure Smoke
1         low      low  high
2        high     high  high
3        high      low   low
4         low     high   low
class=design, type= FrF2

> r=sample(1:4)
> design[r,]

  Temperature Pressure Smoke
2        high     high  high
3        high      low   low
4         low     high   low
1         low      low  high
class=design, type= FrF2
```

Figure 6.7: Illustrating FrF2

- Factor 2 = start angle (location of the arm when the operator releases— starts the forward motion of the arm — levels were 0 and 20 degrees)
- Factor 3 = rubber bands (number of rubber bands used on the catapult — levels were 1 and 2 bands)
- Factor 4 = arm length (distance the arm is extended — levels were 0 and 4 inches)
- Factor 5 = stop angle (location of the arm where the forward motion of the arm is stopped and the ball starts flying — levels were 45 and 80 degrees)

A full factorial design with 5 factors would have $2^5 = 32$ observations. This has 16, which is half that. So we should not expect to test higher order interactions.

Figure 6.8 shows what happens when we try to test all possible interactions for the catapult data. The `anova` table shows that even if we go up only to two-way interactions, we run out of degrees of freedom (there are 5 main effects, 10 two way interactions, and only $16 - 1 = 15$ total degrees of freedom, so there is no sum of squares for error and no tests. We have a couple of ways to go: one way is to take out the least significant stuff, which we'll do next, and another way is to look at a "half-normal plot", which we'll do a bit later.

Now, we don't have any "official" way of assessing the least significant stuff, since we have no tests. But we know how a test would go: we'd take a mean square and divide it by the mean square for error to get an F statistic. So the smallest mean squares are going to go with the smallest F values which will go with the largest P-values. The two smallest mean squares go with the `start:length` and `height:stop` interactions, so let's get rid of those. The analysis is shown in Figure 6.9.

There are a couple of things to note here. One of them is in the model formula. When you see something like $(\texttt{stuff} + \texttt{morestuff})^2$ in the model formula, this means "all the main effects and interactions between two of the variables in the list". $(\texttt{stuff} + \texttt{morestuff} + \texttt{yetmorestuff})^3$ would include three way interactions, and so on. After that, there were two two-way interactions that I wanted to remove, so the `-` in the model formula means "don't include this". You can read the whole thing as "all the main effects and 2-way interactions except for these two". The other thing to note is that all the P-values are pretty small. This, I think, is a consequence of taking out the least significant terms first, so that the error sum of squares (which is very "flimsy" since it is based on only 2 degrees of freedom) is smaller than it really ought to be, and that is making the F values larger. We'll see a way around this in a minute, but to press on with this approach, let's (rather arbitrarily) choose to remove now all the terms with a P-value bigger than 0.06. That'll prune off three more terms, and get us 5 degrees of freedom for error. Since only 5 of the 10 two-way interactions remain, let's just specify the model directly in the model formula. The process is in Figure 6.10.

```
> catapult=read.table("catapult.txt",header=T)
> head(catapult)

  distance height start bands length stop order
1    28.00   3.25     0     1      0   80     1
2   126.50   4.75    20     2      4   80     3
3   126.50   4.75     0     2      4   45     4
4    45.00   3.25    20     2      4   45     5
5    35.00   4.75     0     1      0   45     6
6    28.25   4.75    20     1      0   80     8

> attach(catapult)
> catapult.1=aov(distance~height*start*bands*length*stop)
> anova(catapult.1)

Analysis of Variance Table

Response: distance
              Df Sum Sq Mean Sq F value Pr(>F)
height         1 2909.3  2909.3
start          1 1963.6  1963.6
bands          1 5157.0  5157.0
length         1 6490.3  6490.3
stop           1 2322.0  2322.0
height:start   1  122.4   122.4
height:bands   1  344.6   344.6
start:bands    1  161.0   161.0
height:length  1  353.9   353.9
start:length   1   19.7    19.7
bands:length   1  926.4   926.4
height:stop    1    0.2     0.2
start:stop     1  114.2   114.2
bands:stop     1  128.0   128.0
length:stop    1  157.8   157.8
Residuals      0    0.0
```

Figure 6.8: Analysis 1 of the catapult data

```
> catapult.2=aov(distance~(height+start+bands+length+stop)^2-
+   start:length-height:stop)
> anova(catapult.2)

Analysis of Variance Table

Response: distance
              Df Sum Sq Mean Sq F value   Pr(>F)
height         1 2909.3  2909.3 292.640 0.003400 **
start          1 1963.6  1963.6 197.517 0.005025 **
bands          1 5157.0  5157.0 518.743 0.001922 **
length         1 6490.3  6490.3 652.857 0.001528 **
stop           1 2322.0  2322.0 233.572 0.004254 **
height:start   1  122.4   122.4  12.310 0.072510 .
height:bands   1  344.6   344.6  34.660 0.027660 *
height:length  1  353.9   353.9  35.600 0.026959 *
start:bands    1  161.0   161.0  16.192 0.056569 .
start:stop     1  114.2   114.2  11.490 0.077104 .
bands:length   1  926.4   926.4  93.190 0.010561 *
bands:stop     1  128.0   128.0  12.873 0.069664 .
length:stop    1  157.8   157.8  15.875 0.057604 .
Residuals      2   19.9     9.9
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 6.9: Analysis 2 of the catapult data

```
> catapult.3=aov(distance~height+start+bands+length+stop+
+   height:bands+height:length+start:stop+bands:length+length:stop)
> anova(catapult.3)

Analysis of Variance Table

Response: distance
              Df Sum Sq Mean Sq F value    Pr(>F)
height         1 2909.3  2909.3 33.7338 0.0021342 **
start          1 1963.6  1963.6 22.7686 0.0050075 **
bands          1 5157.0  5157.0 59.7977 0.0005778 ***
length         1 6490.3  6490.3 75.2575 0.0003365 ***
stop           1 2322.0  2322.0 26.9248 0.0034992 **
height:bands   1  344.6   344.6  3.9954 0.1020903
height:length  1  353.9   353.9  4.1037 0.0986426 .
start:stop     1  114.2   114.2  1.3245 0.3018403
bands:length   1  926.4   926.4 10.7424 0.0220137 *
length:stop    1  157.8   157.8  1.8299 0.2340877
Residuals      5  431.2    86.2
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 6.10: Analysis 3 of the catapult data

```
> catapult.4=aov(distance~height+start+bands+length+stop+bands:length)
> anova(catapult.4)

Analysis of Variance Table

Response: distance
             Df Sum Sq Mean Sq F value    Pr(>F)
height        1 2909.3  2909.3 18.6794 0.0019276 **
start         1 1963.6  1963.6 12.6076 0.0062089 **
bands         1 5157.0  5157.0 33.1116 0.0002748 ***
length        1 6490.3  6490.3 41.6722 0.0001174 ***
stop          1 2322.0  2322.0 14.9090 0.0038399 **
bands:length  1  926.4   926.4  5.9484 0.0374303 *
Residuals     9 1401.7   155.7
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 6.11: Analysis 4 of the catapult data

The error mean square has gone way up, and so have many of the P-values. It seems that the only interaction worth keeping is `bands:length`. So let's take out the others and see where we are. This is Figure 6.11. Now it seems that we can stop. There is nothing non-significant to remove[1].

I hope you were a bit bothered by the arbitrary nature of the early part of the analysis, caused by our having to "invent" degrees of freedom for error. Another approach, which gets around this, is called the **half-normal plot**. The idea is that an "effect" is calculated for each term in the ANOVA table of Figure 6.8 (the one with no P-values). If nothing is significant, these should all have a normal distribution, so we draw a QQ plot and see whether these effects lie on a straight line or not. Any that seem off the line are probably significant. So this gives us a rather more direct way of deciding what to keep. That's the basic idea. A refinement is that the effects being positive or negative don't matter much, so we pretend they're all positive and only plot the positive half of the QQ plot. Any effect that's large will be off the line to the top right.

The function for plotting half-normal plots is `qqnorm.aov` from package `gplots`. Actually, if you load this package and then call `qqnorm` on output from `aov`, that'll be enough. The first analysis, `catapult.1`, with all the effects, is the one to use. This is shown in Figure 6.12.

What you do is to start at the bottom left of the plot. These are the small, insignificant, effects. You look to see which of those appear to lie on a straight

[1]or, if you hate double negatives as much as I do, "everything is signficant and so must stay".

```
> library(gplots)
> qqnorm(catapult.1)
```

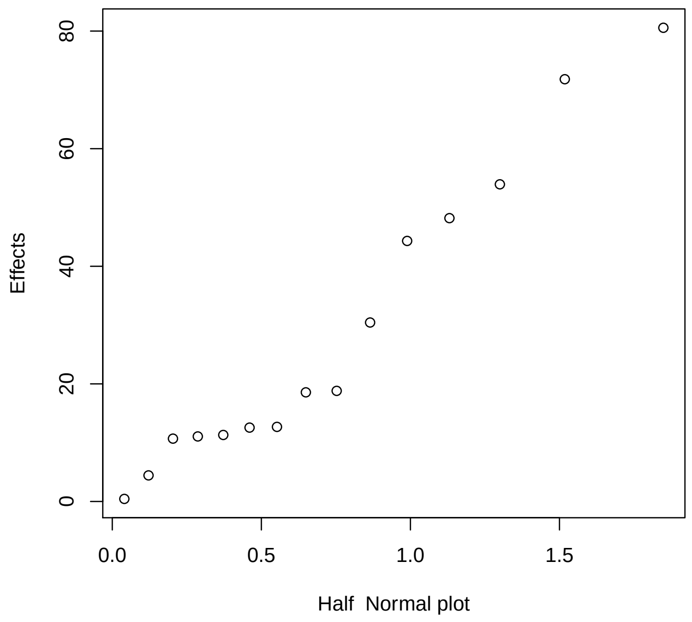


Figure 6.12: Half-normal plot for catapult data

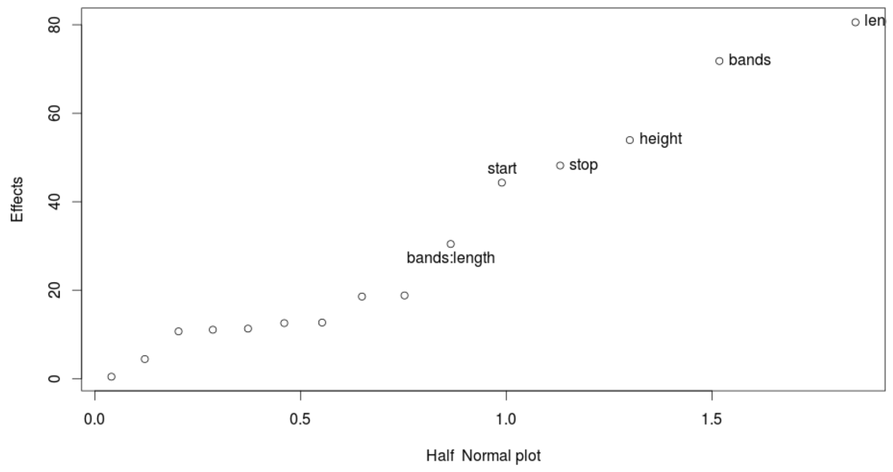


Figure 6.13: Half-normal plot with effects identified

line, and where the trail of points starts curving up. This is a question of judgement too. *My* judgement is that the effects up to about 20 lie on a line, and starting with the effect that's about 30, the effects lie above the straight line passing through the other points.

If you're using R Studio, you can call `qqnorm.aov` as

```
> qqnorm(catapult.1,label=T)
```

and then you can click on any of the effects that look large. This works a bit counter-intuitively. When you click on a circle, nothing appears to happen, but click all the circles you want and then press ESC. Some numbers will appear in the console window, and on the plot will appear the names of the variables whose effects you clicked. I got the picture shown in Figure 6.13. If you don't want to go that far: the effects are related to the F statistics in the ANOVA table (large on one is large on the other); the largest 6 F statistics are the interesting ones, which is the 5 main effects plus the bands-length interaction. That's the same six terms as appear in Figure 6.11, which are indeed significant.

As a second example of a half-normal plot, let's look again at the engineering experiment of Section 6.4.1. This is shown in Figure 6.14. I begin with the ANOVA table (from the first analysis) as a reminder. The half-normal plot is below. It looks to me as if the three biggest effects are significant, which correspond to the three main effects[2].

[2]If you draw a line through the *three smallest* effects, it would pass through the three

```
> anova(eng.design.1)

Analysis of Variance Table

Response: response
                     Df Sum Sq Mean Sq F value    Pr(>F)
Pressure              1  25.00  25.000    40.0 0.0002267 ***
Speed                 1  30.25  30.250    48.4 0.0001176 ***
Force                 1  12.25  12.250    19.6 0.0022053 **
Pressure:Speed        1   2.25   2.250     3.6 0.0943498 .
Pressure:Force        1   2.25   2.250     3.6 0.0943498 .
Speed:Force           1   0.00   0.000     0.0 1.0000000
Pressure:Speed:Force  1   1.00   1.000     1.6 0.2415040
Residuals             8   5.00   0.625
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> qqnorm(eng.design.1)
```

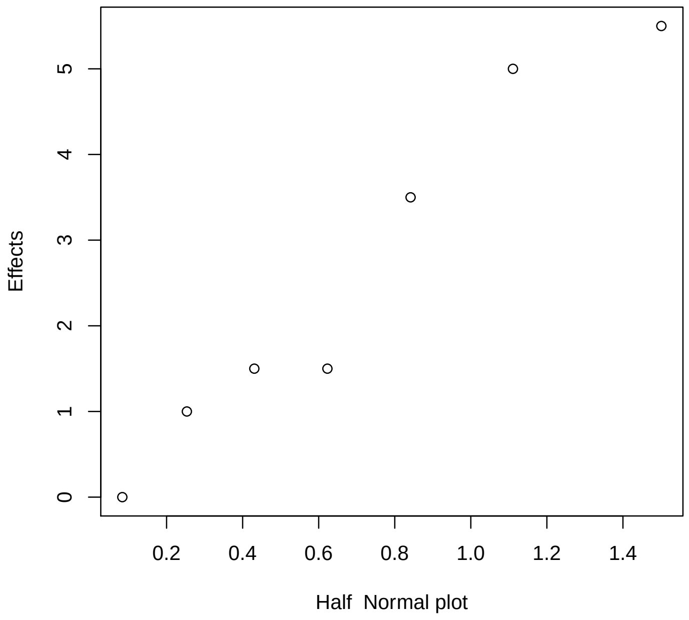


Figure 6.14: Half-normal plot of engineering experiment

Going back to the catapult data, we should probably do some quality control. A kind of bare minimum is to check that the residuals are roughly normal. This is shown in Figure 6.15. Don't get confused by the similarity with Figure 6.12: since we are feeding `qqnorm` a list of numbers rather than a fitted model object, we get a regular normal QQ plot rather than a half-normal plot of effects.

I'd say these are approximately normal, though there is a tendency for the lowest value(s) to be too low and the highest values to be too high. On the website where I got these data, they tried a log transform. The motivation for that was that there was a negative fitted value (see above the plot); since the response variable was distance, it would have to be positive.

A final remark: when all your factors have two levels, there's no reason to do Tukey or any other multiple comparisons method. You know whether the two levels have significantly different responses, so you've found out all you could know about which levels differ. Use `tapply` or something like that to make a table of means.

6.4.3 Blocking and fractional factorials

Your experiment might have to be run under different conditions for some of the trials, eg. at night, where light or temperature might make a difference. The easiest way to handle that is to include "block" in your experiment, with two levels, so now it becomes a 2^{k+1} factorial, or some fraction of that. If your block naturally has four levels, you have two "blocking variables", each with two levels.

For example, you might have 3 "real" factors A, B, C. You might be able to collect all $2^3 = 8$ observations (in which case a complete factorial design would work). But you may be able to do only 4 runs during the day, while the other 4 have to be at night. It makes sense to have two blocks, in which case (depending on your point of view) you use one half-fraction of a 2^3 factorial for the day runs and the other half for the night runs. Or, you add "time" as another, blocking, factor, with levels "day" and "night", and run a half fraction of a 2^4 design.

Some possibilities are shown in Figure 6.16. The first is to get a half-fraction of a 2^3 factorial, say that these observations belong to block 1, and the missing four belong to block 2. The second is to declare `Block` an additional factor. You see that, apart from the randomized order, `Block=1` corresponds to the half-fraction above, and `Block=-1` corresponds to the missing half-fraction. The third, and in practice best, way is to supply a `blocks=` argument to `FrF2` which says how

largest effects as well, and you wouldn't know what to conclude, but the fourth point would be *below* the line, and you're looking for points *above*. Another way to see this is that the three biggest effects are a lot bigger than the four smallest, so drawing the line at three seems sensible. This is a lot like a (right-to-left) scree plot. See Figure 11.12 and the accompanying discussion for more on that.

```
> res=residuals(catapult.4)
> qqnorm(res)
> qqline(res)
> fitted.values(catapult.4)
```

```
        1         2         3         4         5         6         7         8
 34.45313 115.45313 113.51562  64.39062  37.32812  39.26562  86.48438 -11.79688
        9        10        11        12        13        14        15        16
 40.23438  35.42188  35.85938  31.04688  32.98438  82.10938  37.35938 110.64062
```

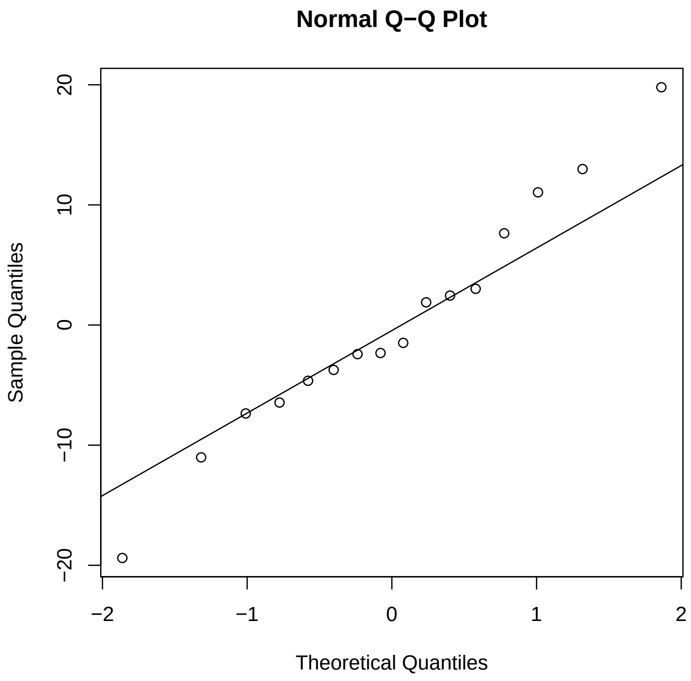


Figure 6.15: QQ plot of residuals

```
> FrF2(4,3)

   A  B  C
1 -1 -1  1
2 -1  1 -1
3  1 -1 -1
4  1  1  1
class=design, type= FrF2

> FrF2(8,factor.names=c("A","B","C","Block"))

   A  B  C Block
1  1  1  1     1
2 -1 -1 -1    -1
3  1 -1  1    -1
4  1  1 -1    -1
5 -1 -1  1     1
6 -1  1  1    -1
7 -1  1 -1     1
8  1 -1 -1     1
class=design, type= FrF2

> FrF2(8,3,blocks=2)

  run.no run.no.std.rp Blocks  A  B  C
1      1         3.1.3      1  1 -1 -1
2      2         4.1.4      1  1  1  1
3      3         1.1.1      1 -1 -1  1
4      4         2.1.2      1 -1  1 -1
  run.no run.no.std.rp Blocks  A  B  C
5      5         6.2.2      2 -1  1  1
6      6         7.2.3      2  1 -1  1
7      7         5.2.1      2 -1 -1 -1
8      8         8.2.4      2  1  1 -1
class=design, type= FrF2.blocked
NOTE: columns run.no and run.no.std.rp are annotation, not part of the data frame
```

Figure 6.16: Including a blocking factor

```
> FrF2(8,3,blocks=2,replications=2)

  run.no run.no.std.rp Blocks  A  B  C
1      1       1.1.1.1    1.1 -1 -1  1
2      2       3.1.3.1    1.1  1 -1 -1
3      3       2.1.2.1    1.1 -1  1 -1
4      4       4.1.4.1    1.1  1  1  1
  run.no run.no.std.rp Blocks  A  B  C
5      5       5.2.1.1    2.1 -1 -1 -1
6      6       7.2.3.1    2.1  1 -1  1
7      7       8.2.4.1    2.1  1  1 -1
8      8       6.2.2.1    2.1 -1  1  1
   run.no run.no.std.rp Blocks  A  B  C
9       9       3.1.3.2    1.2  1 -1 -1
10     10       2.1.2.2    1.2 -1  1 -1
11     11       1.1.1.2    1.2 -1 -1  1
12     12       4.1.4.2    1.2  1  1  1
   run.no run.no.std.rp Blocks  A  B  C
13     13       7.2.3.2    2.2  1 -1  1
14     14       8.2.4.2    2.2  1  1 -1
15     15       6.2.2.2    2.2 -1  1  1
16     16       5.2.1.2    2.2 -1 -1 -1
class=design, type= FrF2.blocked
NOTE: columns run.no and run.no.std.rp are annotation, not part of the data frame
```

Figure 6.17: 2^3 design replicated 2 times in 4 blocks

many blocks you have (which has to be a power of 2, like 2 or 4). The run of `FrF2` with `blocks=2` gives the exact same assignment of combinations of A, B, C to blocks as the other two.

`FrF2` is very versatile: it can handle complete or fractional 2^k designs, with blocks (use `blocks=`) or replications (`replications=`). Figure 6.17 shows a 2^3 design replicated twice in four blocks altogether (two blocks *within* each replication, for four altogether, hence the syntax).

6.5 Other experimental designs

6.5.1 Latin square designs

One of the problems with factorial designs is that you can need a lot of data, and ideally you want replication as well. Sometimes you can get away with less data, if you are willing to make some extra assumptions. In the case of a Latin

```
            Factor 2
           1  2  3  4
Factor 1
       1   1  2  3  4
       2   2  3  4  1
       3   3  4  1  2
       4   4  1  2  3
```

Figure 6.18: Latin square design for factors with 4 levels

square design, the assumptions are these:

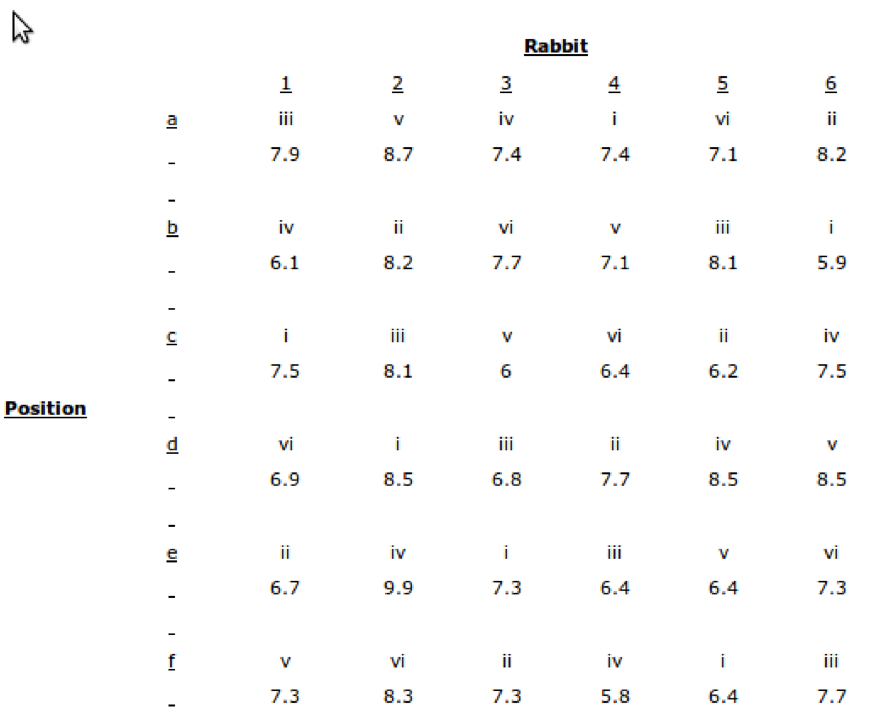
- You have exactly three factors.
- All the factors have exactly the same number of levels.
- There are no interactions between the factors.

If you are willing to go with these rather restrictive assumptions, you can use a Latin square design. I'll show you how it works with three factors, each having four levels. Consult Figure 6.18. I've labelled the four levels of each factor 1, 2, 3, 4. To analyze these three factors you collect $4^2 = 16$ observations. The rows and columns of Figure 6.18 show what levels of factors 1 and 2 you use; the number in the table shows what level of factor 3 you use in combination of those levels of factor 1 and 2.

By way of example, an experiment was carried out in which the skins of rabbits' backs were injected with a diffusing factor. Six injection sites were available on each rabbit's back. A Latin square design was anticipated, so therefore six rabbits were used. Another factor was the order of injection; six orders were used. The response variable was the area of blister in square centimetres. The data are shown in Figure 6.19. The rabbits are labelled with numbers, the injection sites by letters, and the order of injection by Roman numerals.

Now, we have to get these into a data frame with one observation per row. The function `expand.grid` will help us some (with the rabbits and the locations), but we'll have to do the orders and the response values by hand. Figure 6.20 shows the process.

First, we do all the combinations of rabbit and position. I'm doing `rabbit` first because `expand.grid` varies the first thing fastest, and by doing it this way we can read the order and the response along the rows of Figure 6.19. Next comes `order`. I used regular numbers rather than Roman numerals to save some typing. A couple of other things to note: I didn't have to put all those numbers on one line, because R was waiting for the final close-bracket to know



		Rabbit					
		1	2	3	4	5	6
Position	a	iii 7.9	v 8.7	iv 7.4	i 7.4	vi 7.1	ii 8.2
	b	iv 6.1	ii 8.2	vi 7.7	v 7.1	iii 8.1	i 5.9
	c	i 7.5	iii 8.1	v 6	vi 6.4	ii 6.2	iv 7.5
	d	vi 6.9	i 8.5	iii 6.8	ii 7.7	iv 8.5	v 8.5
	e	ii 6.7	iv 9.9	i 7.3	iii 6.4	v 6.4	vi 7.3
	f	v 7.3	vi 8.3	ii 7.3	iv 5.8	i 6.4	iii 7.7

Figure 6.19: Data for Latin square example

```
> rabbitdata=expand.grid(rabbit=1:6,position=c("a","b","c","d","e","f"))
> rabbitdata$order=c(3,5,4,1,6,2,
+   4,2,6,5,3,1,
+   1,3,5,6,2,4,
+   6,1,3,2,4,5,
+   2,4,1,3,5,6,
+   5,6,2,4,1,3)
> rabbitdata$blister=c(79,87,74,74,71,82,
+   61,82,77,71,81,59,
+   75,81,60,64,62,75,
+   69,85,68,77,85,85,
+   67,99,73,64,64,73,
+   73,83,73,58,64,77)
> head(rabbitdata)

  rabbit position order blister
1      1        a     3      79
2      2        a     5      87
3      3        a     4      74
4      4        a     1      74
5      5        a     6      71
6      6        a     2      82
```

Figure 6.20: Organizing the rabbit data

```
> rabbit.aov=aov(blister~factor(rabbit)+position+factor(order),data=rabbitdata)
> anova(rabbit.aov)

Analysis of Variance Table

Response: blister
               Df  Sum Sq Mean Sq F value  Pr(>F)
factor(rabbit)  5 1283.33 256.667  3.9096 0.01235 *
position        5  383.33  76.667  1.1678 0.35919
factor(order)   5   56.33  11.267  0.1716 0.97013
Residuals      20 1313.00  65.650
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 6.21: ANOVA for rabbit data

that I was done. Also, storing the result as I did added a variable `order` to the data frame. Lastly, we enter the responses, reading along the rows, in the same way as `order`. I saved myself some work by omitting the decimal points, so that my variable `blister` is 10 times the one in Figure 6.19. I need to keep that in mind, though it shouldn't affect the ANOVA.

Now we come to the actual analysis. This works using `aov`, same as any other analysis of variance. The variables `rabbit` and `order` were numbers, but we want them to be treated in the ANOVA as factors, hence the `factor`s on the `aov` line.

The only significant effect was that of `rabbit`; the position and order were not significant. (The researchers were curious about an effect of order, but there isn't one.) Figure 6.22 shows what happens when you remove the non-significant terms. The nature of the Latin square is that you get an ordinary ANOVA in the remaining factors after you take one or two out. `rabbit` has become more strongly significant. You can go a couple of ways with this: you can run a Tukey to find which rabbits differ in `blister` from which, or you can think of `rabbit` as a blocking factor, in which case you don't care about differences among rabbits. (Or you can even treat `rabbit` as a random effect, along the lines of Section 6.7.)

Assuming that you were interested in differences among rabbits, the Tukey in Figure 6.22[3] shows that rabbit 2 is significantly different from rabbits 3, 4 and 5, but no other differences are significant. The `tapply`, which comes out above the plot, shows the mean blister areas for each rabbit; rabbit 2 had a noticeably

[3]The `las` makes the rabbit comparisons on the vertical axis come out sideways, and the `cex` shrinks them so you can actually see them.

```
> rabbit.aov2=aov(blister~factor(rabbit),data=rabbitdata)
> anova(rabbit.aov2)

Analysis of Variance Table

Response: blister
               Df Sum Sq Mean Sq F value   Pr(>F)
factor(rabbit)  5 1283.3 256.667  4.3933 0.004044 **
Residuals      30 1752.7  58.422
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> rabbit.tukey=TukeyHSD(rabbit.aov2)
> plot(rabbit.tukey,las=1,cex=0.75)
> with(rabbitdata,
+      tapply(blister,factor(rabbit),mean),
+      )

       1        2        3        4        5        6
70.66667 86.16667 70.83333 68.00000 71.16667 75.16667
```

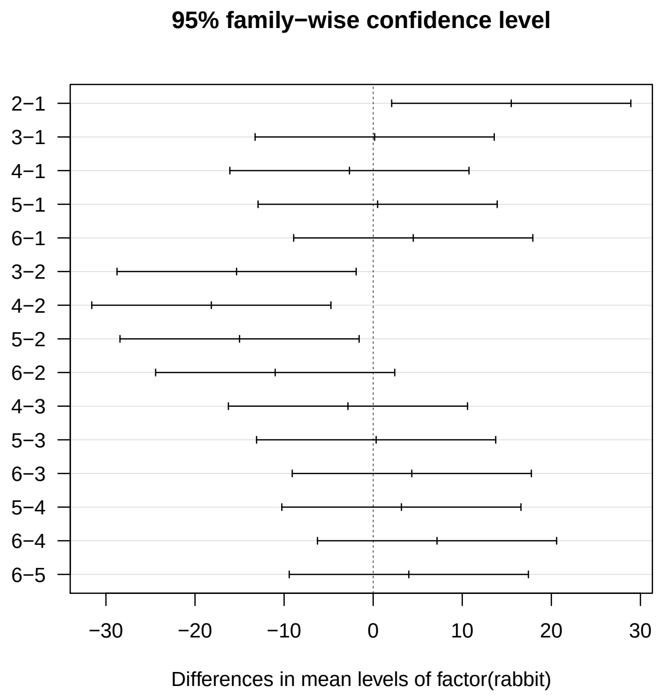


Figure 6.22: Second ANOVA and Tukey for rabbit data

larger mean blister area than the other rabbits[4].

******** set.seed, plus make sure what comes out is what's described.

Now, let's think about designing a Latin square experiment. We return to the 3 factors with 4 levels each that we began with. This is reproduced in Figure 6.23. We follow the procedure using `expand.grid` that we used in entering the rabbit data. Then we add the third factor by hand, shuffle the numbers 1 through 16 (since there are 16 observations to collect) and look at the shuffled data frame. As it happens, setting all factors to level 1 is the first "run", the second one is factor 1 level 4, factor 2 level 1, factor 3 level 4, and so on. When you collect the data, you glue the values for the response variable onto the data frame as with the rabbit data, and away you go.

Under the even more unlikely conditions that you have *four* (or even `five`) factors all with the same number of levels, with no interactions, there exist such things as a Graeco-Latin square and a hyper-Graeco-Latin square that will enable you to do an analysis very much like the one you saw in Figure 6.21.

6.5.2 Split-plot designs

Take a look at the data in Figure 6.24. There appears to be nothing too strange here: we are studying the corrosion resistance of steel bars (that's the response variable `corrres`) as it depends on two variables: furnace temperature `temp`, and the coating `coating`. So we go ahead and do a two-factor ANOVA, as shown in Figure 6.25. This shows that there is a strong effect of temperature, but no interaction or effect of coating. Right?

Wrong! What we have assumed in this analysis is "complete randomization". That is, all the combinations of temperature and coating ($3 \times 4 = 12$ of them) were arranged in a data frame, replicated twice, and then shuffled. So each of the 24 observations were the result of setting the furnace to the right temperature, putting the right coating on a steel bar, putting it in the furnace for the appointed time, and then measuring its corrosion resistance.

Except that this is not what happened. Setting a furnace to a certain temperature is hard to do (it's rather like pre-heating your oven at home): you set the temperature to what you want, you *wait*, and in an industrial furnace you check that the temperature has stabilized at the value you want. This is a time-consuming process. So what was done instead was to randomize the

[4]The `with` command, which is an alternative to `attach`ing and then `detach`ing again, works like this: you supply it a data frame, and then something to do to the variables in that data frame, in this case calculate a table of means. It looks a bit confusing all on one line, so I've split it into three, taking advantage of R not caring which lines things are on. The first line is the data frame, the second is the "what to do", and the third is the final closing bracket, on a line by itself so that I don't get confused by what might otherwise be several close-brackets.

```
            Factor 2
           1  2  3  4
Factor 1
       1   1  2  3  4
       2   2  3  4  1
       3   3  4  1  2
       4   4  1  2  3

> mydesign=expand.grid(fac2=1:4,fac1=1:4)
> mydesign$fac3=c(1,2,3,4,2,3,4,1,3,4,1,2,4,1,2,3)
> shuf=sample(1:16)
> mydesign=mydesign[shuf,]
> mydesign

   fac2 fac1 fac3
4     4    1    4
5     1    2    2
6     2    2    3
9     1    3    3
15    3    4    2
10    2    3    4
14    2    4    1
8     4    2    1
3     3    1    3
2     2    1    2
13    1    4    4
11    3    3    1
12    4    3    2
7     3    2    4
16    4    4    3
1     1    1    1
```

Figure 6.23: Latin square design

```
> coatings=read.csv("coating.csv",header=T)
> coatings$temp=factor(coatings$temp)
> coatings

   furnace.run coating temp corrres
1            1      c1  360      67
2            1      c2  360      73
3            1      c3  360      83
4            1      c4  360      89
5            2      c1  370      65
6            2      c2  370      91
7            2      c3  370      87
8            2      c4  370      86
9            3      c1  380     155
10           3      c2  380     127
11           3      c3  380     147
12           3      c4  380     212
13           4      c1  360      33
14           4      c2  360       8
15           4      c3  360      46
16           4      c4  360      54
17           5      c1  370     140
18           5      c2  370     142
19           5      c3  370     121
20           5      c4  370     150
21           6      c1  380     108
22           6      c2  380     100
23           6      c3  380      90
24           6      c4  380     153
```

Figure 6.24: Coatings data

```
> coatings.1=aov(corrres~temp*coating,data=coatings)
> summary(coatings.1)

             Df Sum Sq Mean Sq F value  Pr(>F)
temp          2  26519   13260  10.226 0.00256 **
coating       3   4289    1430   1.103 0.38602
temp:coating  6   3270     545   0.420 0.85180
Residuals    12  15561    1297
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 6.25: Wrong analysis of coatings data

temperatures (360, 370 and 380 twice each), set the furnace to the appointed temperature, and then put 4 steel bars, each with different coatings, all in the furnace at the same time (randomizing their locations within the furnace). So there were two separate randomizings going on, temperatures, and within each temperature, the locations of coatings.

So this has to be accounted for in the analysis.

The term "split-plot design" comes from agricultural field trials, where you might have one factor, say "irrigation method", that is hard to restrict to small areas (it's easier to irrigate a whole field the same way). So you divide your experimental area into "whole plots" (like the smallest area you can restrict an irrigation method to), and you randomize your hard-to-change factor over the whole plots. *Within* each whole plot, you do a second level of randomizing, which might be varieties of crop or fertilizers, something you can easily do within small areas or "split plots". For example, you might have four fields and two irrigation methods, where you can irrigate a whole field one way, but not an area smaller than that. So your fields are whole plots, and you randomize irrigation methods over your four fields (each method appears twice). Then, *within* each field you randomize again, let's say fertilizers, to the split plots. For instance, you might have three fertilizers, so you create three split plots within each field and randomly assign one fertilizer to each. (Or, you might be able to create six split plots within a field, and then your randomization has each fertilizer appear twice, since there are twice as many split plots as fertilizers.) See how there is a "C within B within A" thing happening?

How does that apply to our coated steel bars? Well, what's within what? The coatings are randomized over locations within a furnace run, and the temperatures are randomized over the six furnace runs (so that each temperature appears twice). In the jargon, the 6 furnace runs are the whole plots. The temperatures are randomized within these. Then the locations within the furnace are the split plots; the coatings are randomized to those. So it's furnace run, and within that temperature, and within *that*, coatings. This is what we need to figure out to get the analysis right.

To do a non-standard ANOVA in R, we need to add an `Error` term to our model formula to get the within-ness right. This is why I needed to have that variable `furnace.run` in the data frame.

Figure 6.26 shows the correct analysis for the coatings data. The `Error` is added onto the end of the model formula. Inside `Error` is a list of things, separated by slashes. These are the whole plots followed by whatever is inside the whole plots. What is left is the split plots, where the coatings go (the locations within the furnace). This doesn't need to be specified, because the coatings are at the lowest level and they will be correctly tested at that level anyway.

The conclusions from Figure 6.26 are exactly the opposite from what we had

```
> coatings$furnace.run=factor(coatings$furnace.run)
> coatings.2=aov(corrres~coating*temp+Error(furnace.run/temp),data=coatings)
> summary(coatings.2)

Error: furnace.run
          Df Sum Sq Mean Sq F value Pr(>F)
temp       2  26519   13260   2.755  0.209
Residuals  3  14440    4813

Error: Within
             Df Sum Sq Mean Sq F value  Pr(>F)
coating       3   4289  1429.7  11.480 0.00198 **
coating:temp  6   3270   545.0   4.376 0.02407 *
Residuals     9   1121   124.5
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 6.26: Correct split-plot analysis for coatings data

before! This time, the interaction and coating effect are both significant, and the temperature is not.

These analyses are *very* easy to mess up. One check that you have is that the “outer” variable `temp` gets tested in its own ANOVA table first, and the inner variable `coating` gets tested at the next level. The other problem I kept having is that my variables kept ending up as something other than factors. The test to see whether this has happened is that you have the wrong number of degrees of freedom in your ANOVA table(s): it should be one less than the number of levels of the factor.

Since the interaction between coating and temperature is significant, it would be nice to look at a plot that indicates why the interaction might be significant. A significant interaction here means that the effect of coating is not constant over temperatures, but depends on which temperature you’re looking at. A handy tool (that I hadn’t discovered when I wrote the ANOVA sections) is `interaction.plot`. The plot for our data is shown in Figure 6.27. This requires three things: first, the factor that’s going on the horizontal axis; second, the factor that goes to make the lines; third, the response variable. The idea of the plot is that if the effects of one factor are consistent over the effects of the other (“additive”, in the jargon), then the lines will be parallel; if there is an interaction, they won’t be, and looking at why they’re not parallel will give you a sense of why there was a significant interaction.

Looking at Figure 6.27, coatings C4 and C2 seem to be in contradiction to one another. Coating C4’s corrosion resistance keeps on going up as the temperature

```
> attach(coatings)
> interaction.plot(temp,coating,corrres)
> detach(coatings)
```

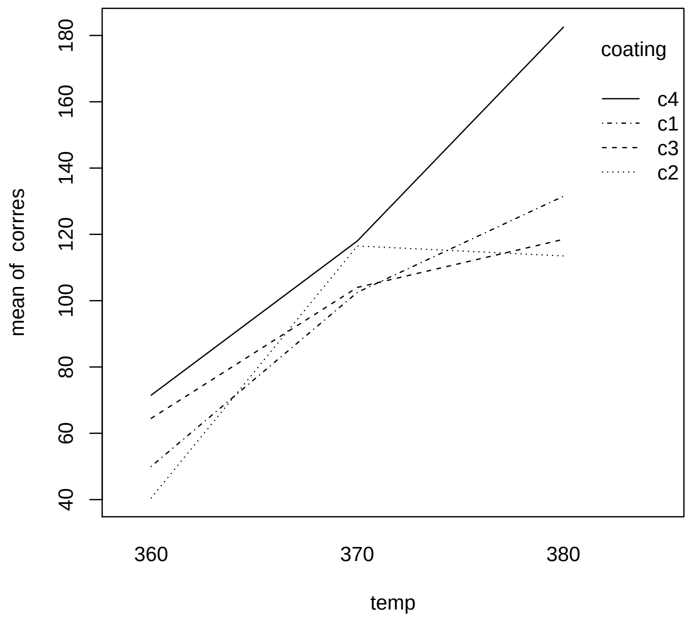


Figure 6.27: Interaction plot of temperature and coating

```
> prepost=read.table("ancova.txt",header=T)
> head(prepost)

  drug before after
1    a      5    20
2    a     10    23
3    a     12    30
4    a      9    25
5    a     23    34
6    a     21    40
```

Figure 6.28: Memory test data

goes up, while coating C2's increases from 360 to 370, but actually goes *down* a little between 370 and 380. If there were no interaction, the two coatings would have a similar "trace" of corrosion resistance over temperature — maybe one would be higher than the other, but the traces would be parallel.

Coatings C1 and C3 lie somewhere in between C2 and C4, in that their increase in corrosion resistance between temperatures 370 and 380 is a bit less than between 360 and 370, but at least it's still going up.

This is not the most severe non-parallelism you're ever likely to see, which is why the interaction is not highly significant.

6.6 Analysis of covariance

This is the name given to an analysis of variance with a "covariate" or another continuous variable attached to the analysis. In R terms there isn't that much to it, because it's just another kind of `lm`. `lm` doesn't care whether your explanatory variables are numerical or categorical; it will handle them equally well. But there is a change of emphasis when it comes to handling things.

Let's imagine we are testing two new drugs that are supposed to improve memory. Let's call them `a` and `b`. But you might imagine that some people are just better at memorizing things than others. So the experiment is designed this way: each subject is randomly allocated to drug `a` or `b`, but we give them a memory test before administering the drug as well as after. Some of the data are shown in Figure 6.28.

One way you might handle this is like a matched pairs: you measure the difference in scores, after minus before, and then you treat this difference as a response variable. This analysis is shown in Figure 6.29. It shows a significant effect of `drug`. (That is, whether or not the drugs are actually any good, there

```
> attach(prepost)
> diff=after-before
> prepost.0=aov(diff~drug)
> summary(prepost.0)

            Df Sum Sq Mean Sq F value   Pr(>F)
drug         1  180.0   180.0   22.22 0.000173 ***
Residuals   18  145.8     8.1
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 6.29: “Paired differences” analysis of memory test data

is a difference between how much of a change they cause in memory test scores.)

This is all right here, since the test scores before and after are on the same scale, but they might not have been. Another way of handling things is to predict the after score using a linear model (a kind of mixed-up regression/ANOVA) predicting after score from before score (numerical) and drug (categorical). This would work regardless of the scales of the variables. Before we get to that, though, we can plot the after scores against the before scores, labelling the drug groups with different symbols. This is shown in Figure 6.30. The after scores for drug A (blue circles) are generally higher than for drug B (red crosses) if you compare values with similar `before` scores. This suggests that there might be a significant `drug` effect, once you allow for `before` score. (When `before` is higher, so is `after` regardless of drug, which confuses the issue.)

Our first analysis is shown in Figure 6.31. I've included the interaction between `before` and `drug`. But before we look too closely at the results of the analysis, let's do some predictions, so that we can see what this model is doing. If you compare predictions for drug A and drug B for the same before score, what you see is that the predicted `after` scores are getting a little further apart: that is to say, drug A is “more better” as the `before` score increases. Perhaps a better way to see this is on a plot. We can repeat the previous plot, with the different coloured points showing the results for the different drugs, and add lines showing the predicted values out of `prepost.1.pred`. (This is why I did two separate lots of predictions in Figure 6.31.)

Our plot is shown in Figure 6.32. Both lines show the predictions going up as the `before` score goes up, but the blue line (drug A) is going up faster, so that the difference between drugs is getting bigger as the before score increases. In other words, the lines are not parallel.

It turns out that fitting the interaction between the numerical and categorical variable is what allowed this non-parallelism to happen. If we look at the coefficients for this model, in Figure 6.33, drug A is the “baseline”. The intercept

```
> mycols=c("blue","red")
> mych=c(1,3)
> plot(before,after,col=mycols[drug],pch=mych[drug])
> legend("topleft",levels(drug),col=mycols,pch=mych)
```

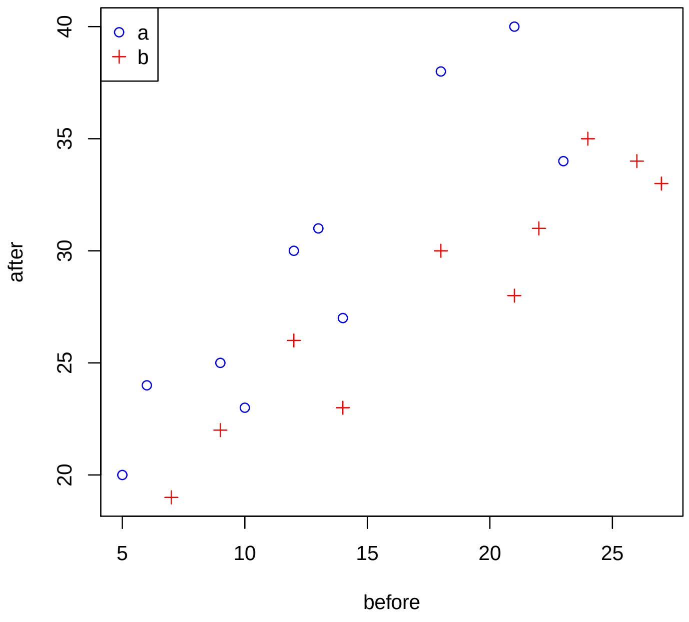


Figure 6.30: Plot of after scores against before scores, labelled by group

```
> prepost.1=lm(after~before*drug)
> prepost.a=expand.grid(before=c(5,15,25),drug="a")
> prepost.b=expand.grid(before=c(5,15,25),drug="b")
> prepost.a.pred=predict(prepost.1,prepost.a)
> prepost.b.pred=predict(prepost.1,prepost.b)
> cbind(prepost.a,prepost.a.pred)

  before drug prepost.a.pred
1      5    a       21.29948
2     15    a       31.05321
3     25    a       40.80693

> cbind(prepost.b,prepost.b.pred)

  before drug prepost.b.pred
1      5    b       18.71739
2     15    b       25.93478
3     25    b       33.15217
```

Figure 6.31: ANCOVA 1 with predictions

of 16 says that a person on drug A with a before score of 0 would have an after score of 16 (about what you would guess from Figure 6.32). The `before` coefficient is just less than 1, so that as `before` goes up by 1, for a person on drug A, the after score is predicted to go up by just less than 1. That leaves the -0.25 coefficient for `before:drugb`. That means that, for a subject on drug B, you work out their predicted `after` score from the other coefficients, *and then subtract 0.25 for each point of before score.* This is what makes the lines grow (slowly, 0.25 not being very big) further apart.

The interpretation here is not too difficult, because the predicted drug A after score is always higher than the drug B after score, for all the before scores in the range of our data. But you could imagine that the slopes might be more different than here, and that the lines could even cross over, so that for some `before` scores, drug A might be better, and for the others, drug B might be better. This would be troublesome to interpret.

Now, you might say that the two lines' slopes here are not very different, and you might wonder whether they are significantly different. This is assessed by testing the interaction term. The process is seen in the last two lines of Figure 6.33. The strategy is, "fit the model without, and see whether it makes the fit a lot worse". So we fit the model with just main effects for `before` and `drug`, and use `anova` to compare the two fits. The one without the interaction is not significantly worse (or, the one with it is not significantly better), so the interaction can be taken out.

```
> plot(before,after,col=mycols[drug],pch=mych[drug])
> legend("topleft",levels(drug),col=mycols,pch=mych)
> lines(prepost.a$before,prepost.a.pred,col="blue")
> lines(prepost.b$before,prepost.b.pred,col="red")
```

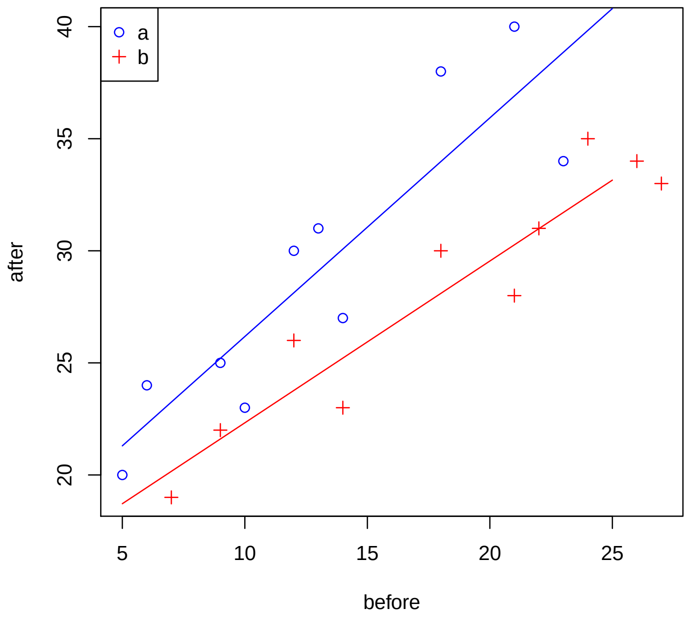


Figure 6.32: Plot of predicted after scores from model with interaction

```
> summary(prepost.1)

Call:
lm.default(formula = after ~ before * drug)

Residuals:
    Min      1Q  Median      3Q     Max 
-4.8562 -1.7500  0.0696  1.8982  4.0207 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept)   16.4226     2.0674   7.944 6.08e-07 ***
before         0.9754     0.1446   6.747 4.69e-06 ***
drugb         -1.3139     3.1310  -0.420    0.680    
before:drugb  -0.2536     0.1893  -1.340    0.199    
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1 

Residual standard error: 2.622 on 16 degrees of freedom
Multiple R-squared: 0.8355,       Adjusted R-squared: 0.8046 
F-statistic: 27.09 on 3 and 16 DF,  p-value: 1.655e-06 

> prepost.2=lm(after~before+drug)
> anova(prepost.2,prepost.1)

Analysis of Variance Table

Model 1: after ~ before + drug
Model 2: after ~ before * drug
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1     17 122.32                           
2     16 109.98  1    12.337 1.7948 0.1991
```

Figure 6.33: Summary of 1st model and testing interaction for significance

```
> summary(prepost.2)

Call:
lm.default(formula = after ~ before + drug)

Residuals:
    Min      1Q  Median      3Q     Max 
-3.6348 -2.5099 -0.2038  1.8871  4.7453 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  18.3600     1.5115  12.147 8.35e-10 ***
before        0.8275     0.0955   8.665 1.21e-07 ***
drugb        -5.1547     1.2876  -4.003 0.000921 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1 

Residual standard error: 2.682 on 17 degrees of freedom
Multiple R-squared: 0.817,        Adjusted R-squared: 0.7955 
F-statistic: 37.96 on 2 and 17 DF,  p-value: 5.372e-07
```

Figure 6.34: Investigating the no-interaction model

The coefficients for the no-interaction model are shown in Figure 6.34. The `before` coefficient of 0.8 says that *regardless of drug*, a one-point increase in `before` goes with a 0.8 increase in `after`. The `drugb` coefficient of -5 says that being on drug B rather than drug A goes with a 5-point decrease in `after` score, *regardless* of the before score. In other words, drug A is better all the way along, and the amount by which it is better is the same all the way along.

To see this better, we can make a plot as before (the end of Figure 6.34). The code looks a bit complicated, but it isn't really. We make two lots of predictions: one for drug A at various before scores (the same ones as before), and one for drug B ditto. Then we plot the data, and a line for each of the drug predictions.

Notice that the lines in Figure 6.35 are now *parallel*: the predicted difference between drugs for a given `before` score is the same all the way along. This is a consequence of taking out the interaction term.

We would imagine that the effect of `before` score is definitely significant, and the effect of `drug` looks consistent enough to be significant. We can confirm this by (a) fitting a model without `before` and comparing its fit to `prepost.2` using `anova`, and (b) fitting a model without `drug` and comparing its fit to `prepost.2` using `anova`. There is a shortcut to this, which is the function `drop1`. This looks at a model and assesses the effect of dropping things one at a time, which is

```
> prepost.a.pred=predict(prepost.2,prepost.a)
> prepost.b.pred=predict(prepost.2,prepost.b)
> cbind(prepost.a,prepost.a.pred)

  before drug prepost.a.pred
1      5    a       22.49740
2     15    a       30.77221
3     25    a       39.04703

> cbind(prepost.b,prepost.b.pred)

  before drug prepost.b.pred
1      5    b       17.34274
2     15    b       25.61756
3     25    b       33.89237

> plot(before,after,col=mycols[drug],pch=mych[drug])
> legend("topleft",levels(drug),col=mycols,pch=mych)
> lines(prepost.a$before,prepost.a.pred,col="blue")
> lines(prepost.b$before,prepost.b.pred,col="red")
```

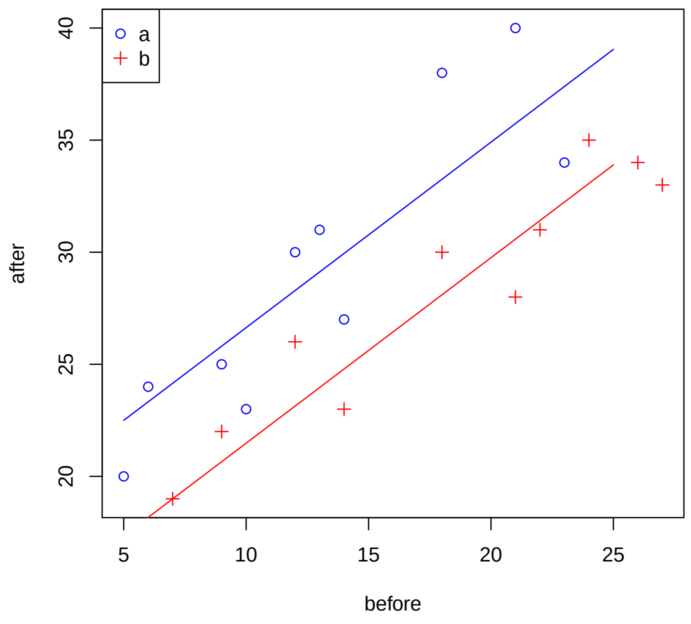


Figure 6.35: Plotting the no-interaction predictions

```
> drop1(prepost.2,test="F")

Single term deletions

Model:
after ~ before + drug
       Df Sum of Sq    RSS    AIC F value    Pr(>F)
<none>              122.32 42.218
before  1    540.18 662.50 74.006  75.074 1.211e-07 ***
drug    1    115.31 237.63 53.499  16.025 0.0009209 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Figure 6.36: Dropping terms from `prepost.2`

exactly what we want here. `drop1` is shown in Figure 6.36. The function needs to know which fitted model we want to drop things from, and also needs to know what test we want to use to assess the significance of dropping things. The right test for an `lm` model is `test="F"`[5].

The P-values in Figure 6.36 are both less than 0.001 (in the case of `before`, almost as small as 0.0000001). So they both have to stay. The model `prepost.2` with the parallel lines is the one we stick with.

6.7 Random effects

In the analyses of variance we have seen, we have treated the explanatory variables as having "fixed effects": that is, the levels of the variables we saw were the only ones we were interested in. But sometimes, the levels of the variables we observed were just a sample of the ones we might have seen. This is commonly the case when one of the variables is "subjects"; the particular people or animals that took part in our experiment were just typical representatives of the people or animals that might have done.

When the variable has a random effect, we have to handle it a different way. Instead of saying that each level has an effect on the *mean*, we say that there is variation due to the random effect, with a variance specific to the variable, which can be estimated. Now, to test that a variable has no effect, we test that its specific variability is zero.

There can be both fixed and random effects in an experiment; if that is the case,

[5]This means an F test, not that the option `test` should be FALSE, which would be written `test=F` without the quotes.

we have a *mixed model*.

When there is at least one random effect, we cannot rely on the F-tests in fixed-effects ANOVA, even for fixed effects. The right tests in the presence of random effects can be different. R handles all this, if done the right way.

Let's start with our deer leg length data, Figure 6.37. Leg length might depend on which leg is being measured, and on which deer we're measuring. We analyzed this as a matched-pairs design and a randomized block ANOVA (with fixed effects), and found that there was a significant difference in mean leg lengths (and also a significant difference among individual deer).

There are a couple of packages that do random- and mixed-effects modelling. The one I'm using here is called `lme4`; there is also `nlme`, which operates slightly differently. They both do more general things than we see here, but I won't worry too much about that.

The basic operation of `lme4` is akin to `lm` (or `aov`), with a model formula that contains the fixed effects but with the random effects specified differently. In our case, `leg` is a fixed effect (there are only two possible legs!), but `indiv` is a random effect. So the model formula looks like `length leg+(1|indiv)`. In our "simple" case, you specify the random effect(s) as a `1|` followed by a factor that has a random effect. Here, we just have `indiv` as a random effect, so the model term is `1|indiv` (in brackets, for technical reasons). This is all laid out in Figure 6.38, where you also see the output of the analysis of variance.

The ANOVA table just tests the fixed effects, here `leg`. In this case (though not always), the F-value for an effect of `leg` is exactly the same as in the randomized blocks analysis. So in the end it didn't matter whether we treated `indiv` as a fixed or a random effect. You see that there is no P-value here. A general way of testing for significance of terms in a mixed model is to fit a second model *without* the thing you're testing, and then use `anova` to compare the two, simpler model first.

When you have an interaction, though, it can make a difference whether you treat a factor as fixed or random. Recall one of our two-way ANOVA examples: some mice were exposed to nitrogen dioxide, and some were not, and the serum fluorescence in each mouse was measured at either 10, 12 or 14 days. The fixed-effects analysis is shown in Figure 6.40.

Now, presumably, the experimenter chose 10, 12 and 14 days for a reason. But you might imagine these as a random sample of all possible numbers of days. There is a (tiny) case for treating days as a random effect. Let's see what happens with this.

First off, we have a fixed factor, group, and a random factor, days. We also want to fit an interaction. An interaction containing a random factor is itself random, which means that `days:group` is random, not fixed. The analysis is

```
> s

   length     leg indiv
1     142 Foreleg     1
2     140 Foreleg     2
3     144 Foreleg     3
4     144 Foreleg     4
5     142 Foreleg     5
6     146 Foreleg     6
7     149 Foreleg     7
8     150 Foreleg     8
9     142 Foreleg     9
10    148 Foreleg    10
11    138 Hindleg     1
12    136 Hindleg     2
13    147 Hindleg     3
14    139 Hindleg     4
15    143 Hindleg     5
16    141 Hindleg     6
17    143 Hindleg     7
18    145 Hindleg     8
19    136 Hindleg     9
20    146 Hindleg    10

> anova(s.aov)

Analysis of Variance Table

Response: length
          Df  Sum Sq Mean Sq F value  Pr(>F)
leg        1  54.450  54.450  4.7136 0.04437 *
indiv      1  54.123  54.123  4.6853 0.04494 *
Residuals 17 196.377  11.552
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 6.37: Deer leg length data

```
> library(lme4)
> s.lme=lmer(length~leg+(1|indiv),data=s)
> anova(s.lme)

Analysis of Variance Table
    Df Sum Sq Mean Sq F value
leg  1  54.45   54.45  11.654
```

Figure 6.38: Analysis treating deer as a random factor

```
> s2.lme=lmer(length~(1|indiv),data=s)
> anova(s2.lme,s.lme)

Data: s
Models:
s2.lme: length ~ (1 | indiv)
s.lme: length ~ leg + (1 | indiv)
       Df    AIC    BIC  logLik  Chisq Chi Df Pr(>Chisq)
s2.lme  3 115.83 118.81 -54.913
s.lme   4 109.49 113.47 -50.746 8.3345      1    0.00389 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 6.39: Analysis of the effect of `leg`

```
> head(no2)

  group days fluor
1     c   10   143
2     c   12   179
3     c   14    76
4     c   10   169
5     c   12   160
6     c   14    40

> no2.aov=aov(fluor~group*factor(days),data=no2)
> anova(no2.aov)

Analysis of Variance Table

Response: fluor
                   Df Sum Sq Mean Sq F value   Pr(>F)
group               1   3721  3721.0  3.0120 0.092916 .
factor(days)        2  15464  7731.8  6.2584 0.005351 **
group:factor(days)  2   8686  4343.1  3.5155 0.042495 *
Residuals          30  37062  1235.4
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 6.40: Fixed effects analysis of fluorescence data

```
> ddf=factor(no2$days)
> no2.lme=lmer(fluor~group+(1|ddf)+(1|ddf:group),data=no2)
> no2.lme.2=lmer(fluor~group+(1|ddf),data=no2)
> anova(no2.lme.2,no2.lme)

Data: no2
Models:
no2.lme.2: fluor ~ group + (1 | ddf)
no2.lme: fluor ~ group + (1 | ddf) + (1 | ddf:group)
          Df    AIC    BIC  logLik Chisq Chi Df Pr(>Chisq)
no2.lme.2  4 374.72 381.05 -183.36
no2.lme    5 375.67 383.59 -182.83 1.047      1     0.3062
```

Figure 6.41: Random-effects analysis of fluorescence data

shown in Figure 6.41.

So we have to specify one fixed effect, group, and *two* random effects, one for days, and one for the days-group interaction. To save myself some typing, I defined `ddf` to be `days` turned into a factor. Then our model formula looks like

$$\texttt{fluor group} + (1|\texttt{ddf}) + (1|\texttt{ddf} : \texttt{group})$$

and that produces the fitted model `no2.lme`. Normally, we'd do `anova` or `summary` of our fitted model object, but at this point there isn't much to see.

The fixed-effects strategy now is to test the interaction for significance. The way that works here is to fit a second model, here labelled `no2.lm3.2`, *without* the interaction. Then, running `anova` with *two* models tests whether the bigger one (the one with more stuff in it) is really an improvement over the smaller one. If it isn't, we go with the smaller one. That's done with the line

$$\texttt{anova}(\texttt{no2.lme.2}, \texttt{no2.lme}).$$

The output from this shows that the interaction doesn't add anything significant to the model, so we pull out Occam's Razor and shave off the interaction term. Note that in the fixed-effects analysis, the interaction was significant, so turning `days` into a random effect has made a substantial difference to the results.

A word of caution here: the P-values that come out of this are only approximate, and the displayed values can sometimes be twice as much as the true values.

A brief explanation of this follows. Feel free to skip. This issue is not a problem with R, but rather a problem with the theory. It is especially a concern when testing the significance of random effects, as we are doing here. When doing that, recall that we are testing that the specific variance of the random effect is zero. Now, this is as small as a variance (or standard deviation) can be, so we are "on the edge" of the set of possible values for the variance. When you are "on the boundary of the parameter space", the standard theory about testing significance doesn't apply, but the only way you can get any P-values at all is to pretend that it does.

All right, the interaction wouldn't even be significant if the P-value were a half of what we see, so the interaction stays out.

The next stage is to see whether there is a `group` effect. Test this in the same way; try taking it out, and see if the fit is significantly better with it in (or worse with it out). This part of the analysis is shown in Figure 6.42. Bear in mind that `no2.lme.2` contains groups and days, and we'll see if we can take `group` away from that.

The P-value is just less than 0.10, and, bearing in mind that we might need to halve it, we can say that there is a marginally significant effect of groups.

```
> no2.lme.3=lmer(fluor~(1|ddf),data=no2)
> anova(no2.lme.3,no2.lme.2)

Data: no2
Models:
no2.lme.3: fluor ~ (1 | ddf)
no2.lme.2: fluor ~ group + (1 | ddf)
          Df    AIC    BIC  logLik  Chisq Chi Df Pr(>Chisq)
no2.lme.3  3 375.32 380.07 -184.66
no2.lme.2  4 374.72 381.05 -183.36 2.6086      1     0.1063
```

Figure 6.42: Testing for a significant `group` effect

Compare that with the fixed-effects analysis, where there was a signification interaction between groups and days. The Tukey analysis there came out with a few group-days combinations that had significantly different mean fluorescence than others, but it was not clear whether one group had a significantly higher mean than the other overall, because the pattern was all mixed up.

In the same way that we are not interested in a test for blocks, we are not really interested in the test for the random effect of days: we expect it to be there, so it is not worth testing.

Another random-effects model: Three machines are being tested (fixed effect) by six different people (random effect). The response variable is a rating that measures quality and quantity of articles produced.

Unfortunately, the data look like this:

```
 1 1 52.0  1 2 51.8  1 2 52.8  1 3 60.0  1 4 51.1  1 4 52.3
   1 5 50.9  1 5 51.8  1 5 51.4  1 6 46.4  1 6 44.8  1 6 49.2
   2 1 64.0  2 2 59.7  2 2 60.0  2 2 59.0  2 3 68.6  2 3 65.8
   2 4 63.2  2 4 62.8  2 4 62.2  2 5 64.8  2 5 65.0  2 6 43.7
   2 6 44.2  2 6 43.0  3 1 67.5  3 1 67.2  3 1 66.9  3 2 61.5
   3 2 61.7  3 2 62.3  3 3 70.8  3 3 70.6  3 3 71.0  3 4 64.1
   3 4 66.2  3 4 64.0  3 5 72.1  3 5 72.0  3 5 71.1  3 6 62.0
   3 6 61.4  3 6 60.5
```

These are machine, person and rating, but with multiple measurements in a single row. How can we get this into the form we want? Figure 6.43 shows one way. There are three steps: read the data one value at a time from the file into a *vector*, which is what `scan` does. Then arrange it into the right *shape*, which is what `matrix` does. This takes a string of numbers in a vector, and arranges it into a *matrix*, an array of numbers in rows and columns, like a data frame. `matrix` expects some options: first, the vector of values to go into the

```
> raw.data=scan("machine.txt")
> raw.data

  [1]  1.0  1.0 52.0  1.0  2.0 51.8  1.0  2.0 52.8  1.0  3.0 60.0  1.0  4.0 51.1
 [16]  1.0  4.0 52.3  1.0  5.0 50.9  1.0  5.0 51.8  1.0  5.0 51.4  1.0  6.0 46.4
 [31]  1.0  6.0 44.8  1.0  6.0 49.2  2.0  1.0 64.0  2.0  2.0 59.7  2.0  2.0 60.0
 [46]  2.0  2.0 59.0  2.0  3.0 68.6  2.0  3.0 65.8  2.0  4.0 63.2  2.0  4.0 62.8
 [61]  2.0  4.0 62.2  2.0  5.0 64.8  2.0  5.0 65.0  2.0  6.0 43.7  2.0  6.0 44.2
 [76]  2.0  6.0 43.0  3.0  1.0 67.5  3.0  1.0 67.2  3.0  1.0 66.9  3.0  2.0 61.5
 [91]  3.0  2.0 61.7  3.0  2.0 62.3  3.0  3.0 70.8  3.0  3.0 70.6  3.0  3.0 71.0
[106]  3.0  4.0 64.1  3.0  4.0 66.2  3.0  4.0 64.0  3.0  5.0 72.1  3.0  5.0 72.0
[121]  3.0  5.0 71.1  3.0  6.0 62.0  3.0  6.0 61.4  3.0  6.0 60.5

> m=matrix(raw.data,ncol=3,byrow=T)
> machine.data=data.frame(mach=factor(m[,1]),pers=factor(m[,2]),rating=m[,3])
> head(machine.data)

  mach pers rating
1    1    1   52.0
2    1    2   51.8
3    1    2   52.8
4    1    3   60.0
5    1    4   51.1
6    1    4   52.3
```

Figure 6.43: Reading in the machine data

```
> machines.1=lmer(rating~mach+(1|pers)+(1|pers:mach),data=machine.data)
> machines.2=lmer(rating~mach+(1|pers),data=machine.data)
> anova(machines.2,machines.1)

Data: machine.data
Models:
machines.2: rating ~ mach + (1 | pers)
machines.1: rating ~ mach + (1 | pers) + (1 | pers:mach)
           Df    AIC    BIC   logLik  Chisq Chi Df Pr(>Chisq)
machines.2  5 257.11 266.04 -123.557
machines.1  6 203.68 214.38  -95.839 55.435      1  9.659e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> names(machines.1)

NULL
```

Figure 6.44: Random-effects analysis of machine data

matrix, and then either the number of rows (`nrow`) or number of columns that the matrix should have. Here, we know that each observation consists of three things: the machine, the person and the rating, so we specify three columns with `ncol=3`. Finally, the default is to fill the matrix down the columns, but we want to fill it along the rows, so we have to specify `byrow=T`. (I always seem to want to fill matrices by rows, so I always need to say `byrow=T`, it seems.)

The last step is to turn the matrix into a data frame, using `data.frame`. While we have the opportunity, let's give our variables sensible names, and let's make them categorical so they'll be ready for the analysis.

Now we can do our random-effects analysis. Since our `person`s were a random sample of all possible people, `person` is a random effect. We are only interested in these three machines, though, so `machine` is a fixed effect. We have enough data to assess an interaction (there are generally two or three observations per person-machine combination), so we'll put an interaction in our model, bearing in mind that the interaction has a random effect in it, so it's random too.

The analysis is as shown in Figure 6.44. We fit the complete model, including interaction, first. Then we try taking the interaction out, and see whether the fit is significantly worse. It is (the P-value is very small), so we have to leave the interaction in. This is a bit of a hindrance when it comes to interpreting the results, because it says that relative performance on the different machines is different for different people, and so we can't say how the machines compare overall. (If the interaction had not been significant, we would have had a fixed effect of machines that we could test for significance, and if it is, we would

```
> attach(machine.data)
> rat.mean=tapply(rating,list(mach,pers),mean)
> rat.mean

     1        2    3        4        5        6
1 52.0 52.30000 60.0 51.70000 51.36667 46.80000
2 64.0 59.56667 67.2 62.73333 64.90000 43.63333
3 67.2 61.83333 70.8 64.76667 71.73333 61.30000

> detach(machine.data)
```

Figure 6.45: Mean ratings by person and machine

have been able to make a statement about how the mean ratings compare by machine. But as it is, we can't.

Let's try to understand this by looking at the means, as shown in Figure 6.45. The people are in the columns, and the machines in the rows. For these six people (which, remember, are only a sample of all people who might be using these machines), machine 3 always produces the highest average ratings, sometimes by a little (eg. person 4), and sometimes by a lot (person 6). Likewise, the ratings for machine 1 are usually the lowest, but the difference from machine 2 varies quite a bit. The story is inconsistent enough to say that the interaction is significant (the model with the interaction explains the data sufficiently better than the one without).

We might also try to display this on a plot. On the horizontal axis needs to go the machine (1 to 3), and on the vertical goes the rating, with lines joining the means coming from the same person. I did a bit of thinking about and experimenting with this, with my final deliberations summarized in Figure 6.46.

Often times, when you're constructing a graph bit by bit (and I wanted a line for each person), the best way to start is to set up the plotting area, but not actually plot anything in it. I wanted the horizontal axis to cover 1, 2 and 3, which it would by feeding in `1:3` as the x-variable. Getting the y axis right requires a bit more thought, but eventually I realized it needed to go from the very smallest rating mean anywhere, to the very biggest one. These are the `min` and `max` elements of the `rat.mean` matrix. I thought I'd calculate those first so that the first `plot` line wouldn't be more cluttered up than necessary. Then I can plot `1:3` against, say, the first column of `rat.mean` (the means for person 1), setting the limits of the y scale using `ylim` and the min and max values I found. I use `type="n"` to not actually plot anything. The problem then is that axis labels are not very illuminating, so by explicitly setting `xlab` and `ylab` I can make them say what I want.

Now to actually plot the means, person by person. `points` seems to be easiest

```
> rat.min=min(rat.mean)
> rat.max=max(rat.mean)
> plot(1:3,rat.mean[,1],type="n",ylim=c(rat.min,rat.max),ylab="mean rating",xlab="machine")
> for (i in 1:6)
+   {
+     points(1:3,rat.mean[,i],type="b")
+   }
```

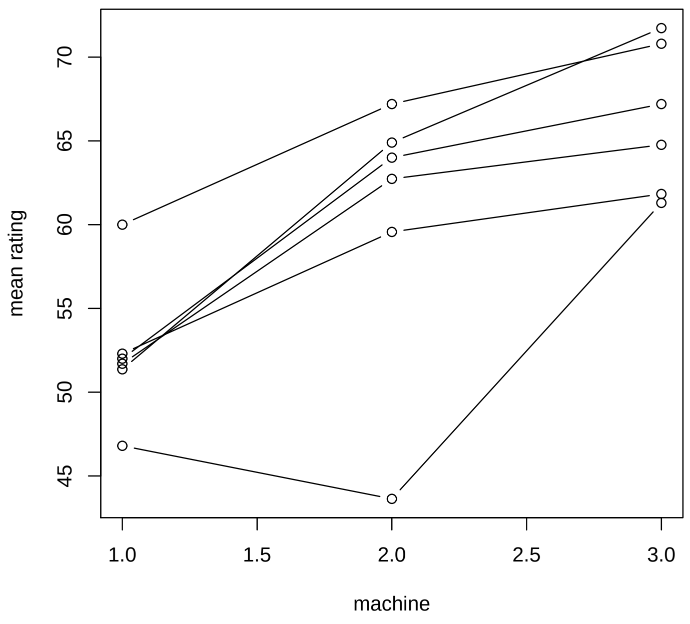


Figure 6.46: Plot of means by machine for each person

for this, since we're plotting points that we happen to want joined by lines, rather than lines themselves. What we need is to plot `1:3` against each column of `rat.mean`, plotting both the points and lines joining them, which is done by `type="b"`. So something like

$$\texttt{points}(1:3, \texttt{rat.mean}[,1], \texttt{type} = "\texttt{b}")$$

will plot the machine means for person 1, then you can repeat for persons 2, 3,. . . . The programmer in me wouldn't let me do that, so I used a loop, as you see in Figure 6.46.

Looking at the plot itself, if there were no interaction, those lines would all be (more or less) parallel to each other. For five of the people, the parallelism is not so bad, but the person with the lowest mean rating on all the machines definitely has a different "profile" from the others. This is person 6, who was the one person to have a lower mean rating on machine 2 than on machine 1. My guess is that person 6 is largely responsible for making the interaction significant, and repeating the analysis without person 6 might tell a different story.

Concerning the interpretation, it's always awkward when the random effect interacts with the treatment, because it says that the effect of treatment (machine) differs among people. Since our people were only a random sample of all possible people, this doesn't say anything much about the preferability of the machines.

An experiment was conducted to assess the effect of two different drugs on blood histamine levels over time in dogs. 8 dogs were used, and each dog was measured at 4 times, 0, 1, 3 and 5 minutes after taking their drug.

This is an example of a **repeated measures** experiment, where each dog provides four measurements to the data set. (This is the same sort of thing as the matched pairs experiment we saw earlier.) We'd expect measurements on the same dog to be correlated, rather than independent; also, our dogs are presumably a sample of "all possible dogs", rather than the only ones we are interested in. So we should treat `dog` as a random effect.

The data come to us as shown in Figure 6.47. Let's see if we can start by plotting the dogs' log-histamine levels over time. This format of data is convenient for plotting, since the four measurements we want to plot together are all on one line. Following the same idea as in Figure 6.46, we want times (0, 1, 3, 5) on the horizontal axis, and log-histamine levels on the vertical. This time, though, we want to distinguish the two drugs, which we can do with different line types. I'm going to do this with some copying and pasting rather than a loop this time, despite my programming instincts, because it seems that a loop would hide the ideas rather then making them clear.

Anyway, the procedure is shown in Figure 6.48. We start by extracting the part of `dogs` that has the log=histmaine values in it, finding the min and max log-

```
> dogs=read.table("dogs2.txt",header=T)
> dogs

          Drug junk   lh0   lh1   lh3   lh5
1     Morphine    N -3.22 -1.61 -2.30 -2.53
2     Morphine    N -3.91 -2.81 -3.91 -3.91
3     Morphine    N -2.66  0.34 -0.73 -1.43
4     Morphine    N -1.77 -0.56 -1.05 -1.43
5 Trimethaphan    N -3.51 -0.48 -1.17 -1.51
6 Trimethaphan    N -3.51  0.05 -0.31 -0.51
7 Trimethaphan    N -2.66 -0.19  0.07 -0.22
8 Trimethaphan    N -2.41  1.14  0.72  0.21
```

Figure 6.47: Data for dogs repeated measures experiment

histamine levels, and drawing a blank graph with the right scales and axis labels, using `type="n"` so as not to plot anything. Then we do the actual plotting for each dog using `points`. We plot the appropriate row of `lh` (which are the log-histamine values from `dogs`) against `days`, using `type="b"` to plot both points and lines. The first four dogs were given Morphine, so we plot their data with a solid line, and the last four were given Trimethaphan, so we plot their data with a dashed line.

The overall picture in Figure 6.48 is this: for (almost) all the dogs, the log-histamine level rises to a maximum at 1 day and then declines after that. This pattern is common to both drugs, so we're not looking at a drug-time interaction. The log-histamine level seems generally highed for the Trimethophan, so we would expect to see an overall drug effect. The log-histamine levels look more variable for the dogs on Morphine, and maybe we should be concerned about that. But the eight lines look pretty near parallel.

Though this data set would be in the right format for SAS, it isn't for the analysis in R, because we want all the responses in *one* column. (Don't worry about the variable `junk`; that was part of another study.) So we have some work to do first.

You might recognize from our previous work that this is a job for `stack`. It seems to be easiest to `stack` the whole data frame and ignore the warning (caused by `drug` not being a number). This is shown in Figure 6.49. The resulting data frame, which I called `sdogs`, for "stacked dogs", has $8 \times 4 = 32$ rows, some of which are shown in Figure 6.49.

The problem now is that we don't know which dog or `Drug` is associated with each row of the new data frame. So we have to re-associate these with our re-organized data, bearing in mind that all the `lh0` measurements are first, then all the `lh1`'s, and so on. So for `Drug` we need four Morphines and then

```
> days=c(0,1,3,5)
> lh=dogs[,3:6]
> min.hist=min(lh)
> max.hist=max(lh)
> plot(days,lh[1,],ylim=c(min.hist,max.hist),xlab="Days",ylab="Log-histamine",type="n"
> points(days,lh[1,],type="b",lty="solid")
> points(days,lh[2,],type="b",lty="solid")
> points(days,lh[3,],type="b",lty="solid")
> points(days,lh[4,],type="b",lty="solid")
> points(days,lh[5,],type="b",lty="dashed")
> points(days,lh[6,],type="b",lty="dashed")
> points(days,lh[7,],type="b",lty="dashed")
> points(days,lh[8,],type="b",lty="dashed")
```

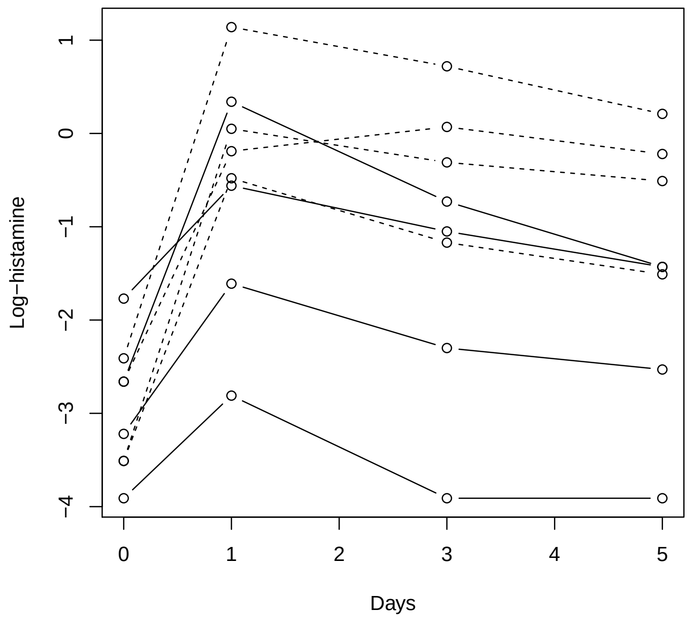


Figure 6.48: Plot of the dog histamine level data

```
> sdogs=stack(dogs)
> sdogs[c(1:4,29:32),]

   values ind
1   -3.22 lh0
2   -3.91 lh0
3   -2.66 lh0
4   -1.77 lh0
29  -1.51 lh5
30  -0.51 lh5
31  -0.22 lh5
32   0.21 lh5
```

Figure 6.49: Reorganizing the `dogs` data frame

```
> druglist=c("Morphine","Trimethaphan")
> dr=rep(druglist,each=4,length=32)
> dr

 [1] "Morphine"     "Morphine"     "Morphine"     "Morphine"     "Trimethaphan"
 [6] "Trimethaphan" "Trimethaphan" "Trimethaphan" "Morphine"     "Morphine"
[11] "Morphine"     "Morphine"     "Trimethaphan" "Trimethaphan" "Trimethaphan"
[16] "Trimethaphan" "Morphine"     "Morphine"     "Morphine"     "Morphine"
[21] "Trimethaphan" "Trimethaphan" "Trimethaphan" "Trimethaphan" "Morphine"
[26] "Morphine"     "Morphine"     "Morphine"     "Trimethaphan" "Trimethaphan"
[31] "Trimethaphan" "Trimethaphan"

> dg=rep(1:8,length=32)
> sdogs$drug=factor(dr)
> sdogs$dog=factor(dg)
> names(sdogs)[1]="log.histamine"
> names(sdogs)[2]="days"
> sdogs[c(1:4,29:32),]

   log.histamine days         drug dog
1          -3.22  lh0     Morphine   1
2          -3.91  lh0     Morphine   2
3          -2.66  lh0     Morphine   3
4          -1.77  lh0     Morphine   4
29         -1.51  lh5 Trimethaphan   5
30         -0.51  lh5 Trimethaphan   6
31         -0.22  lh5 Trimethaphan   7
32          0.21  lh5 Trimethaphan   8
```

Figure 6.50: Getting the data in the right format

four Trimethaphans, with the whole thing repeated until it's of length 32. We can use `gl` for this, or `rep`, which is perhaps easier. `rep` requires up to four things: something to repeat, here the vector `c("Morphine","Trimethaphan")`, the number of times each element of the vector needs to be repeated (`each`), the number of times to repeat the whole thing (`times`), and the length of the final list (`length`). This is shown in Figure 6.50.

Next, we have to know which dog each measurement comes from. If you look at `sdogs`, you'll see that it has all the `lh0` measurements for all 8 dogs, then all the `lh1` measurements, and so on. So the dogs are 1 through 8, repeated 4 times (or to a length of 32). We can take care of that with another use of `rep`; this time we don't have to specify `each` because the default is for each value in the list to be repeated once, which is what we want.

Two more things, (the last five lines of Figure 6.50), then we'll be ready to go. First, the names of the first two columns of `sdogs` came from `stack`, and aren't really what we want. So we'll fix them up. Second, we glue the results of `rep`, for drugs and dogs, onto the data frame, and take a look at part of it. It looks about right.

Now, we have another thing to think about. The measurements on the same dog are probably correlated (rather than being independent, as R will assume unless we state otherwise). Because these are measurements over time, you'd expect measurements closer together in time to be more highly correlated than measurements farther apart in time. (This is as opposed to, say, measurements for four different activities done by an individual, where you might guess that the correlation between the score on one activity and on another might be the same for all pairs of activities.) Now, there is a package `nlme` that has stuff that will handle this, but I can't get it to work, so we'll stick with `lme4`, which means we are not accommodating the correlation.

OK, off we go. The commands and output are in Figure 6.51. The model is that `log.histamine` depends on days and drug (fixed effects) and dog (random effect). I'm using the `data=` version instead of having to `attach sdogs`. The dogs random effect is specified, as before, by `1|dog`. We begin by fitting the model including everything, to obtain `dogs1.lme`.

Next, we test `days` for significance by taking it out. This gives `dog2.lme`. To test `days` for significance, we see if the model without it is significantly worse than the model with it. This is the purpose of the first `anova`. The P-value is very small, so there definitely is an effect of `days`, which is what the plot suggested in Figure 6.48. That plot also suggested that there might be a drug effect, which we test next by taking out `drug`, as in `dogs3.lm3`, and testing to see whether taking out `drug` was a bad idea. The P-value for `drug` is less than 0.10 but greater than 0.05. So its significance is marginal. Perhaps this is not a complete surprise, since some of the dogs on Morphine did do just as well as the dogs on Trimethaphan, as seen in Figure 6.48.

```
> dogs1.lme=lmer(log.histamine~days+drug+(1|dog),data=sdogs)
> dogs2.lme=lmer(log.histamine~drug+(1|dog),data=sdogs)
> anova(dogs2.lme,dogs1.lme)

Data: sdogs
Models:
dogs2.lme: log.histamine ~ drug + (1 | dog)
dogs1.lme: log.histamine ~ days + drug + (1 | dog)
          Df     AIC     BIC  logLik  Chisq Chi Df Pr(>Chisq)
dogs2.lme  4 115.199 121.062 -53.600
dogs1.lme  7  84.658  94.919 -35.329 36.541      3  5.755e-08 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> dogs3.lme=lmer(log.histamine~days+(1|dog),data=sdogs)
> anova(dogs3.lme,dogs1.lme)

Data: sdogs
Models:
dogs3.lme: log.histamine ~ days + (1 | dog)
dogs1.lme: log.histamine ~ days + drug + (1 | dog)
          Df    AIC    BIC  logLik  Chisq Chi Df Pr(>Chisq)
dogs3.lme  6 85.945 94.739 -36.973
dogs1.lme  7 84.658 94.919 -35.329 3.2866      1    0.06985 .
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 6.51: Repeated measures using a mixed model

```
> dogs1.lme

Linear mixed model fit by REML
Formula: log.histamine ~ days + drug + (1 | dog)
   Data: sdogs
   AIC  BIC logLik deviance REMLdev
 86.84 97.1 -36.42    70.66   72.84
Random effects:
 Groups   Name        Variance Std.Dev.
 dog      (Intercept) 0.83327  0.91284
 Residual             0.35069  0.59219
Number of obs: 32, groups: dog, 8

Fixed effects:
                 Estimate Std. Error t value
(Intercept)       -3.5563     0.5129  -6.933
dayslh1            2.4413     0.2961   8.245
dayslh3            1.8713     0.2961   6.320
dayslh5            1.5400     0.2961   5.201
drugTrimethaphan   1.2000     0.6786   1.768

Correlation of Fixed Effects:
            (Intr) dyslh1 dyslh3 dyslh5
dayslh1     -0.289
dayslh3     -0.289  0.500
dayslh5     -0.289  0.500  0.500
drgTrmthphn -0.661  0.000  0.000  0.000
```

Figure 6.52: Output from chosen model for dogs data

I'm going to say that we'll stick with the model including both `days` *and* `drug`, and have a look at the output from that model. This is shown in Figure 6.52. In the table of Random Effects, there are two things: the residual SD (0.6), and a specific SD for dogs (0.9). The fact that the SD for dogs is larger than the residual SD suggests that the dogs were quite variable among themselves. We rather suspected this, and in fact had no great desire to test that the `dog` random effect was significant.

Moving on to the fixed effects, this tells you how, according to the model, the effects of `days` and `drug` stack up, on average. The first number of days (0) and the first drug (Morphine) are used as a baseline; their "estimates" are zero, and the intercept is the predicted `log.histamine` for a dog on the baseline drug at the baseline number of days (Morphine and 0). The other things in the `Estimate` column say how the other levels stack up against the baseline. The estimates for 1, 3 and 5 days are all positive, with 1 day being highest and 3 and 5 being a bit lower (but not as low as 0 days). This pattern is consistent enough for this effect of `day` to be strongly significant. For drugs, the mean `log.histamine` for the dogs on Trimethaphan is 1.2 higher than the dogs on Morphine (other things being equal), but there is a fair bit of uncertainty attached to this, which is why its P-value doesn't reach the 0.05 level of significance.

So, to summarize, that pattern of "high at 1 day, then decreasing" seems to be reproducible (at least, some kind of change over time is), and the log-histamine levels are possibly higher on Trimethaphan than on Morphine, other things being equal.

6.8 Transformations

You might feel that all this straight line stuff is rather restrictive, since most relationships are curves, so why not just fit a curve? The problem is that there is only one way for a straight line to be straight, but there are infinitely many ways for a curve to be curved. So a much easier way is to cajole a curved relationship into being straight, and then to use our straight-line stuff. Likewise, when doing ANOVA, we can cajole data into having constant standard deviation across groups.

So how do we make a curved relationship straight, or make standard deviations equal? We *transform the response variable.* How? Using the recipe below, called the Ladder of Powers:

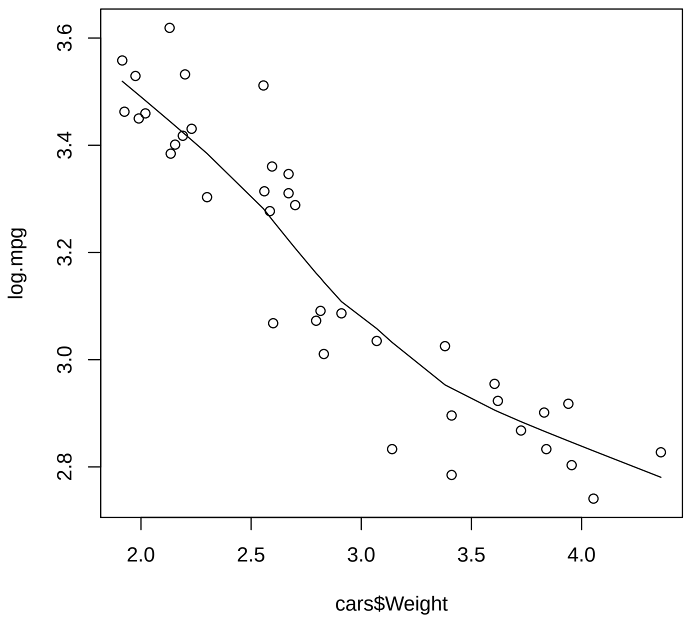
Power	What to do	Comments
1	Nothing	The original data
0.5	Square root	Try for count data
0	Logarithm	Try where percent change matters
-0.5	1 over square root	Rare
-1	1 over response	Try when response is ratio

If you don't have any idea what to try, go a step or two down the ladder, then see if you've not gone far enough (trend still curved) or if you've gone too far (trend curved the other way).

6.8.1 Transformations for regression

Let's see how this works with our cars data for predicting gas mileage from weight. We saw that the original relationship was curved, so something should be done. Let's try logs first. I'll put a lowess curve on the scatter plot to see if that helps. (These are actually "natural logs", but you'll get the same results with different numbers if you use, say, base 10 logs.)

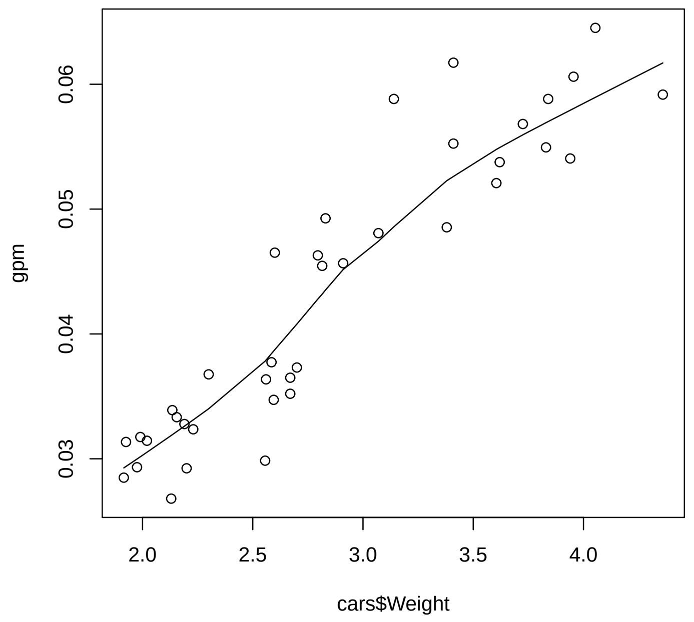
```
> log.mpg=log(cars$MPG)
> plot(cars$Weight,log.mpg)
> lines(lowess(cars$Weight,log.mpg))
```



That's still bendy in the same way that the original relationship was bendy. So we haven't gone far enough.

Here's where we can use some actual science. Miles per gallon is a ratio: miles travelled divided by gallons used. The reciprocal transformation (power -1) turns a ratio the other way up: that is, gallons per mile. This is a perfectly respectable measure of fuel consumption; indeed, it's a multiple of litres per 100 km. So not only is this further down the latter of powers, but it's scientifically reasonable. Does it also straighten things out? I'll define `gpm` to be "gallons per mile", 1 over miles per gallon:

```
> gpm=1/cars$MPG
> plot(cars$Weight,gpm)
> lines(lowess(cars$Weight,gpm))
```



I'd say that's pretty straight, though there is a teeny bit of a bend the other way. (This relationship goes up rather than down because a larger weight goes with a *smaller* gallons per mile.) Let's try a regression or two. First, we'll see if a parabola fits better than a straight line, remembering that `wsq` was weight-squared:

```
> gpm.1=lm(gpm~Weight+wsq,data=cars)
> summary(gpm.1)
```

```
Call:
lm.default(formula = gpm ~ Weight + wsq, data = cars)

Residuals:
       Min         1Q     Median         3Q        Max
-0.0095370 -0.0033971  0.0004451  0.0021991  0.0101275

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.020807   0.014144  -1.471  0.15020
Weight       0.029746   0.009782   3.041  0.00445 **
wsq         -0.002424   0.001616  -1.500  0.14252
```

```
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.004342 on 35 degrees of freedom
Multiple R-squared: 0.8666,        Adjusted R-squared: 0.8589
F-statistic: 113.7 on 2 and 35 DF,  p-value: 4.917e-16
```

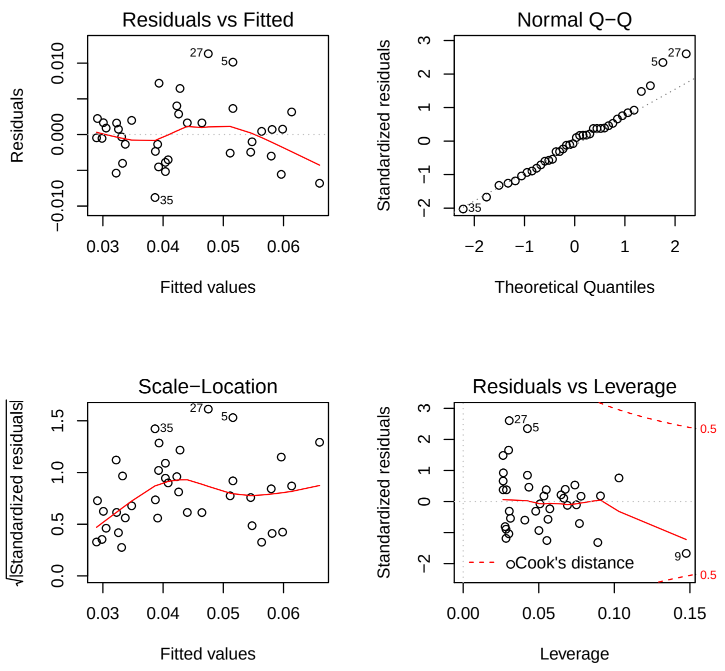
No, we don't need the weight-squared term (its P-value of 0.143 is not small). So let's take it away:

```
> gpm.2=lm(gpm~Weight,data=cars)
> summary(gpm.2)
```

```
Call:
lm.default(formula = gpm ~ Weight, data = cars)

Residuals:
       Min         1Q     Median         3Q        Max 
-0.0088064 -0.0029074  0.0000633  0.0019049  0.0113197 

Coefficients:
              Estimate Std. Error t value Pr(>|t|)    
(Intercept) -0.0000623  0.0030266  -0.021    0.984    
Weight       0.0151485  0.0010271  14.748   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.004416 on 36 degrees of freedom
Multiple R-squared: 0.858,        Adjusted R-squared: 0.854
F-statistic: 217.5 on 1 and 36 DF,  p-value: < 2.2e-16
```

That looks good. We should check the residuals:

```
> par(mfrow=c(2,2))
> plot(gpm.2)
```



Two of the cars, #5 and #27, have larger positive residuals than we might expect. This might be the reason that the plot of residuals vs. fitted values (top left) still looks a bit curved. Take those away and you'd have a more or less random pattern of points, I think. I can live with that.

Also, in the bottom left point, there should be no association between size of residual and fitted value. We saw two of those points just now; the other one, #35, has a large *negative* residual. I think it's those three observations that are making the red lowess curve on the lower left plot wiggle. I can live with that too.

Let's remind ourselves of what the regression is:

```
> summary(gpm.2)

Call:
lm.default(formula = gpm ~ Weight, data = cars)

Residuals:
       Min         1Q     Median         3Q        Max 
-0.0088064 -0.0029074  0.0000633  0.0019049  0.0113197
```

```
Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.0000623  0.0030266  -0.021    0.984
Weight       0.0151485  0.0010271  14.748   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.004416 on 36 degrees of freedom
Multiple R-squared: 0.858,	Adjusted R-squared: 0.854
F-statistic: 217.5 on 1 and 36 DF,  p-value: < 2.2e-16
```

The intercept is very close to 0 (indeed, not significantly different from zero). This is nice, because it says that a car weighing nothing uses no gas. So it's a very simple relationship; ignoring the intercept, it says that the predicted gallons per mile is the car's weight in tons mutiplied by 0.0151.

We did a bunch of predictions earlier. To do that, we constructed a data frame:

```
> pred.df
```

```
  Weight wsq
1      2   4
2      3   9
3      4  16
4      5  25
5      6  36
```

which allows us to do predictions of gas mileage for cars of weight 2 through 6 tons. Even though this data frame contains values for weight squared, that's OK, because we're just not going to use those:

```
> predict.gpm=predict(gpm.2,pred.df,interval="p")
> predict.gpm
```

```
         fit        lwr        upr
1 0.03023461 0.02098428 0.03948493
2 0.04538306 0.03630457 0.05446155
3 0.06053152 0.05115344 0.06990959
4 0.07567997 0.06557271 0.08578723
5 0.09082842 0.07964612 0.10201072
```

These are predicted gallons per mile, so we have to undo our transformation to get things back in miles per gallon. That means taking reciprocals of everything again. Taking the reciprocals of the confidence interval endpoints is at least approximately correct. Let's also make a nicer display:

```
> predict.mpg=1/predict.gpm
> cbind(pred.df$Weight,predict.mpg)
```

```
         fit      lwr       upr
1 2 33.07468 47.65472 25.326117
2 3 22.03465 27.54474 18.361578
3 4 16.52032 19.54903 14.304188
4 5 13.21354 15.25025 11.656747
5 6 11.00977 12.55554  9.802891
```

Because we took reciprocals, the ends of the confidence intervals came out the wrong way around. But no matter. Everything looks, to my mind, entirely sensible: a predicted gas mileage of 11 miles per gallon for a car of weight 6 tons looks reasonable. Of course, assuming that this straight line relationship is going to go on is shakier than actually having data to support that it does go on beyond 4 tons.

One rather curious thing about these intervals is that they get *narrower* as the weight gets bigger, when we have less data. What is happening is that the predictions for *gallons per mile* are getting less accurate as the weight increases beyond 4 tons, which is to say that the `gpm` numbers are all getting bigger and further apart. Undoing the transformation means that the predicted *miles per gallon* numbers are getting smaller and closer together.

But I think we can be very happy with this model.

Another way to handle transformations in regression is via "Box-Cox transformations", which is a semi-automatic way to find a good transformation of the reponse variable. It uses the Ladder of Powers, and tries a bunch of different possible transformations. The end result is a plot showing which kinds of transformations are plausible in the light of the data.

Let's try this out on the cars data. The R function is called `boxcox`, and it lives in the `MASS` package, which you'll need to install first (using `install.packages("MASS")`) if you don't already have it. See Section for more on installing packages.

The procedure for running `boxcox` is shown in Figure 6.53. You feed `boxcox` a model formula, like you would use in regression. There are other options for controlling things, but the default output is a plot.

The single best power transformation is where the curve is highest, marked by the middle of the three vertical dotted lines. This is about −0.6. But the data might tell you a lot about the power transformation, or only a little, so the

```
> library(MASS)
> boxcox(MPG~Weight)
```

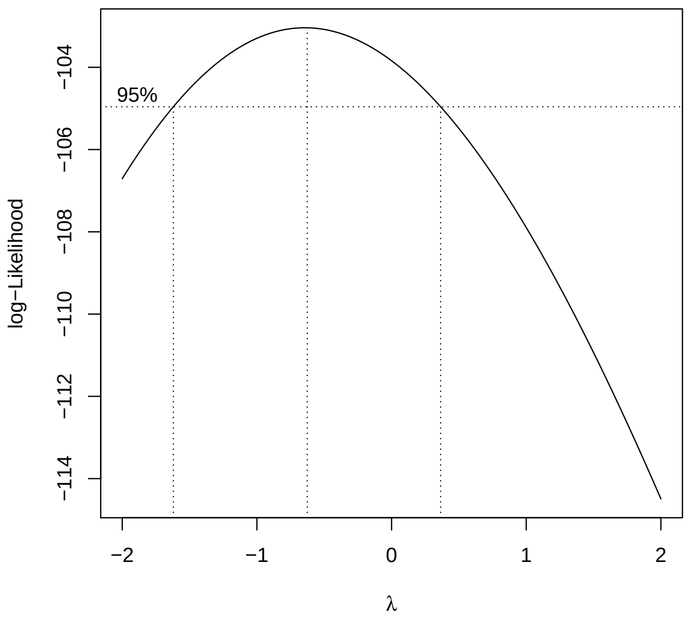


Figure 6.53: `boxcox` on cars data

right thing to look at is the confidence interval for the power transformation. Here that goes from about −1.6 to about 0.4. Any power within that interval is supported by the data.

What we are looking for out of this is a nice round number for the power transformation. So 0 (log transformation) and −1 (reciprocal transformation, as we found above) are reasonable choices. The Box-Cox method is blind to whatever theory might guide the choice, but we found that the −1 transformation had a physical interpretation of gallons of fuel consumed per mile, so that would seem to be a good choice, both from the physical and Box-Cox points of view.

6.8.2 Transformations for ANOVA

I collected all the NHL (hockey) scores from January to March 2012. Is there evidence that some teams score at a higher rate than others?

Here's (a little of) my data:

```
> scores=read.csv("~/sports/nhl.csv",header=T)
> head(scores)
```

```
          team goals
1    Nashville     5
2      Calgary     3
3 Philadelphia     2
4       Ottawa     3
5   New Jersey     2
6     San Jose     3
```

```
> attach(scores)
```

There are 30 teams in the NHL, which is too many for a nice clear analysis. So I'm going to select 6, Philadelphia, Pittsburgh, Winnipeg, Columbus, Chicago and Minnesota. R has a handy function called `%in%`. I make a list of the teams I want, and then pick out the lines of the data frame that have one of my wanted teams as their `team`. Then I print out the first few lines of `team` and whether or not I selected them, to make sure I got the right ones.

```
> wanted.teams=c("Philadelphia","Pittsburgh","Winnipeg","Columbus","Chicago","Minnesota
> to.select=team %in% wanted.teams
> head(data.frame(to.select,team),n=20)
```

```
   to.select         team
1      FALSE    Nashville
2      FALSE      Calgary
3       TRUE Philadelphia
4      FALSE       Ottawa
5      FALSE   New Jersey
6      FALSE     San Jose
7      FALSE    Vancouver
8      FALSE     Edmonton
9       TRUE      Chicago
10     FALSE     Colorado
11     FALSE  Los Angeles
12     FALSE      Buffalo
13     FALSE     Edmonton
14     FALSE      Toronto
15     FALSE    Tampa Bay
16     FALSE   Washington
17     FALSE      Calgary
18     FALSE     Carolina
19     FALSE    St. Louis
20     FALSE      Phoenix
```

Every time `to.select` is true, the team in the original data frame is one of the ones I want to select. So that's good. Now I can use `to.select` to create a new data frame with just the teams I want. At the same time, I'll `detach` the original data frame, which I don't need any more, and `attach` the new one:

```
> detach(scores)
> scores2=scores[to.select,]
> head(scores2)
```

```
           team goals
3  Philadelphia     2
9       Chicago     3
26     Winnipeg     3
28    Minnesota     0
34     Winnipeg     0
36 Philadelphia     5
```

```
> attach(scores2)
> scores2$team=factor(as.character(team))
```

The row numbers in `scores2` are the ones that were selected from the original data frame `scores`.

I'm going to use my `mystats` function to find the mean and SD of each team's goalscoring. I want to array all the goals scored by team before I do that, which is what the `unstack` command does. This takes two things: a data frame, and a model formula saying how things are to be collected together (numerical on the left, categorical on the right)

```
> mylist=unstack(scores2,goals~team)
> head(mylist)
```

```
$Chicago
 [1] 3 4 0 2 5 5 2 4 6 3 2 1 2 4 1 2 3 0 2 4 6 3 2 1 0 1 5 2 2 1 4 2 4 4 5 5 2 1
[39] 1 4 5

$Columbus
 [1] 1 1 4 2 4 1 4 0 2 1 2 0 2 3 3 2 3 3 2 1 2 6 0 2 2 2 5 3 3 1 1 0 3 2 1 5 3 2
[39] 4 4 5

$Minnesota
 [1] 0 1 5 2 2 1 1 5 3 4 1 1 1 2 1 1 3 0 2 3 1 4 0 4 0 0 1 3 3 0 3 2 3 1 0 2 3 4

$Philadelphia
 [1] 2 5 3 4 2 3 2 5 1 4 5 3 1 4 4 2 0 4 2 3 7 4 5 0 5 0 6 1 3 5 1 1 3 3 2 3 1 2
[39] 4 3 7 3

$Pittsburgh
 [1] 1 1 1 0 4 6 2 4 5 4 3 5 0 2 2 2 8 4 1 6 2 2 8 4 4 5 2 3 2 5 5 5 2 8 5 4 5 3
[39] 3 5

$Winnipeg
 [1] 3 0 2 3 0 1 2 1 4 3 1 0 2 2 1 0 2 3 5 1 4 4 5 4 4 2 3 7 3 2 3 5 3 3 4 4 1 4
[39] 2 4 2
```

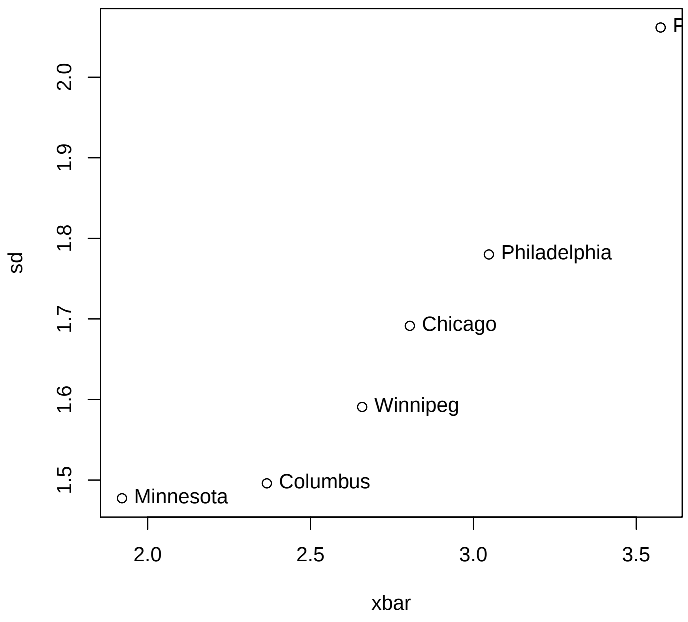
Now we work out the mean and SD of goals scored for each team, by using `sapply` on the list we just created. Then we extract the means and SDs, and plot them against each other.

```
> msd=sapply(mylist,mystats)
> msd
```

```
       Chicago  Columbus Minnesota Philadelphia Pittsburgh  Winnipeg
n    41.000000 41.000000 38.000000    42.000000  40.000000 41.000000
Mean  2.804878  2.365854  1.921053     3.047619   3.575000  2.658537
SD    1.691442  1.495929  1.477426     1.780002   2.061708  1.590751
```

```
Median  2.000000  2.000000  2.000000     3.000000   4.000000  3.000000
IQR     2.000000  2.000000  2.000000     2.000000   3.000000  2.000000
```

```
> xbar=msd[2,]
> sd=msd[3,]
> plot(xbar,sd)
> text(xbar,sd,dimnames(msd)[[2]],pos=4)
```



It's pretty clear that a higher mean number of goals per game goes with a higher SD, which indicates that a transformation of `goals` is called for.

The standard procedure for counts is to try square roots first, so we do that and repeat the above:

```
> mylist=unstack(scores2,sqrt(goals)~team)
> msd=sapply(mylist,mystats)
> msd
```

```
          Chicago   Columbus  Minnesota Philadelphia Pittsburgh   Winnipeg
n      41.0000000 41.0000000 38.0000000   42.0000000 40.0000000 41.0000000
```

```
Mean    1.5560961  1.4142601  1.2056269    1.6280334  1.7791249  1.5041545
SD      0.6269203  0.6122623  0.6929299    0.6378184  0.6482437  0.6371477
Median  1.4142136  1.4142136  1.4142136    1.7320508  2.0000000  1.7320508
IQR     0.5857864  0.7320508  0.7320508    0.5857864  0.8218544  0.5857864
```

That could barely have worked better. The standard deviations of goals scored for the six teams are almost identical.

Now we can do a one-way ANOVA to answer the question we had way back at the beginning:

```
> scores.aov=aov(sqrt(goals)~team)
> anova(scores.aov)


Analysis of Variance Table

Response: sqrt(goals)
           Df Sum Sq Mean Sq F value   Pr(>F)
team        5  7.452 1.49049  3.6124 0.003618 **
Residuals 237 97.788 0.41261
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

So there is definitely a difference in mean goals scored per game among the teams. Which ones are different? Tukey again:

```
> scores.tukey=TukeyHSD(scores.aov)
> scores.tukey


  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov.default(formula = sqrt(goals) ~ team)

$team
                             diff          lwr        upr     p adj
Columbus-Chicago      -0.14183602 -0.549454941 0.26578290 0.9176723
Minnesota-Chicago     -0.35046915 -0.766055316 0.06511702 0.1525307
Philadelphia-Chicago   0.07193732 -0.333248033 0.47712267 0.9957550
Pittsburgh-Chicago     0.22302886 -0.187129769 0.63318748 0.6241674
Winnipeg-Chicago      -0.05194159 -0.459560509 0.35567733 0.9991324
Minnesota-Columbus    -0.20863313 -0.624219295 0.20695304 0.7010352
Philadelphia-Columbus  0.21377334 -0.191412012 0.61895869 0.6544045
```

```
Pittsburgh-Columbus      0.36486488 -0.045293747 0.77502350 0.1125458
Winnipeg-Columbus        0.08989443 -0.317724488 0.49751335 0.9883899
Philadelphia-Minnesota   0.42240647  0.009206944 0.83560599 0.0418572
Pittsburgh-Minnesota     0.57349801  0.155420530 0.99157548 0.0014783
Winnipeg-Minnesota       0.29852756 -0.117058606 0.71411373 0.3099427
Pittsburgh-Philadelphia  0.15109154 -0.256648678 0.55883175 0.8948555
Winnipeg-Philadelphia   -0.12387891 -0.529064260 0.28130644 0.9513582
Winnipeg-Pittsburgh     -0.27497045 -0.685129073 0.13518818 0.3887667
```

Only Pittsburgh-Minnesota and Philadelphia-Minnesota are signficant; that is, based on this sample of games, Pittsburgh and Philadelphia score more goals per game than Minnesota, but there is no evidence of any difference in mean scoring rates between any of the other teams.

6.8.3 Mathematics of transformations

Sometimes, theory will suggest a particular shape of relationship, and by taking logs you can make it linear and use a linear regression.

Examples, with x being your explanatory variable, y being your response, and a and b being parameters you are trying to estimate. My logs are natural logs:

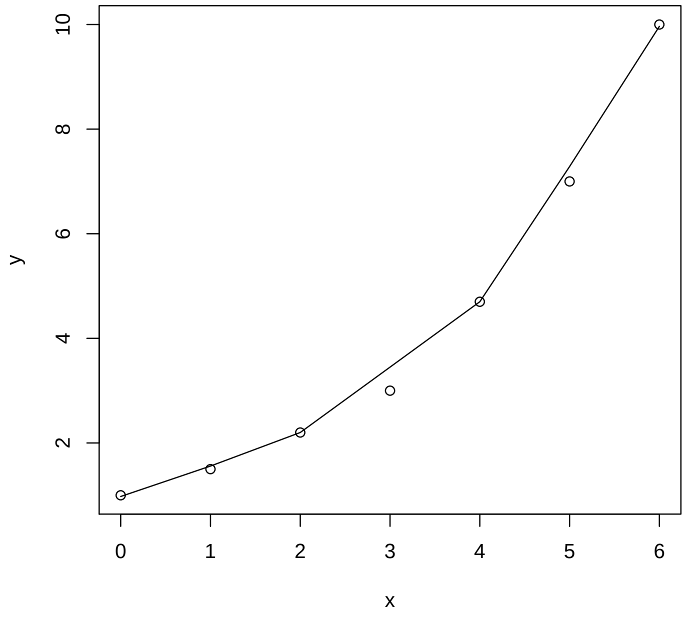
- Relationship is $y = ax^b$. Logs gives $\log y = \log a + b \log x$. Use $\log y$ as your transformed response and $\log x$ as your transformed explanatory variable. The intercept of your regression is $\log a$ and the slope is b.
- Relationship is $y = ab^x$ (exponential growth or exponential decay). Logs gives $\log y = \log a + x \log b$. Take logs of y but don't take logs of x. The intercept of your regression is $\log a$ and the slope is $\log b$.

Here's an example (I made this one up):

```
> eg
```

```
  x    y
1 0  1.0
2 1  1.5
3 2  2.2
4 3  3.0
5 4  4.7
6 5  7.0
7 6 10.0
```

```
> plot(eg)
> lines(lowess(eg))
```



That looks like exponential growth to me. (Plotting the data frame worked because it has two numeric variables. If you feed `plot` a data frame, it works hard to find a suitable way of plotting the things in it.)

So we need to take logs of y but not x:

```
> logy=log(eg$y)
> eg.lm=lm(logy~eg$x)
> summary(eg.lm)

Call:
lm.default(formula = logy ~ eg$x)

Residuals:
        1         2         3         4         5         6         7 
-0.003969  0.017648  0.016792 -0.056901  0.008201  0.022700 -0.004473 

Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 0.003969   0.020327   0.195    0.853
eg$x        0.383848   0.005638  68.088 1.29e-08 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.02983 on 5 degrees of freedom
Multiple R-squared: 0.9989,       Adjusted R-squared: 0.9987
F-statistic:  4636 on 1 and 5 DF,  p-value: 1.294e-08
```

In our exponential growth relationship, $\log a = 0.00397$ and $\log b = 0.384$, so (`exp` is the inverse of `log`):

```
> ec=exp(coefficients(eg.lm))
> ec
```

```
(Intercept)        eg$x
   1.003976    1.467923
```

```
> exp(fitted.values(eg.lm))
```

```
        1         2         3         4         5         6         7
 1.003976  1.473760  2.163365  3.175653  4.661613  6.842887 10.044828
```

That means that the values start at 1, and each one is 1.47, or about 1.5, times bigger than the one before. (I actually made up the data so that each value was “about” 1.5 times the value before.)

Chapter 7

Rolling your own

7.1 Do you need to roll your own?

There is a huge number of add-ons for R. Because R is an open-source endeavour, there is a large community sharing functions and data of all kinds. The unit of adding to R is the **package**. There are packages covering subject areas (such as `ape`, for analyses of phylogenetics and evolution), textbooks (such as `AER`, "Applied Econometrics with R"), statistical methods (such as `leaps`, which does regression subset selection) and data sets (such as `datasets`, surprisingly enough).

When attempting something that isn't in the "base" of R, your first port of call should be to find out whether there's a package that does what you want. There is a whole website of R packages at `http://cran.r-project.org/web/packages/`. If a web search lands you there, that's where you want to be.

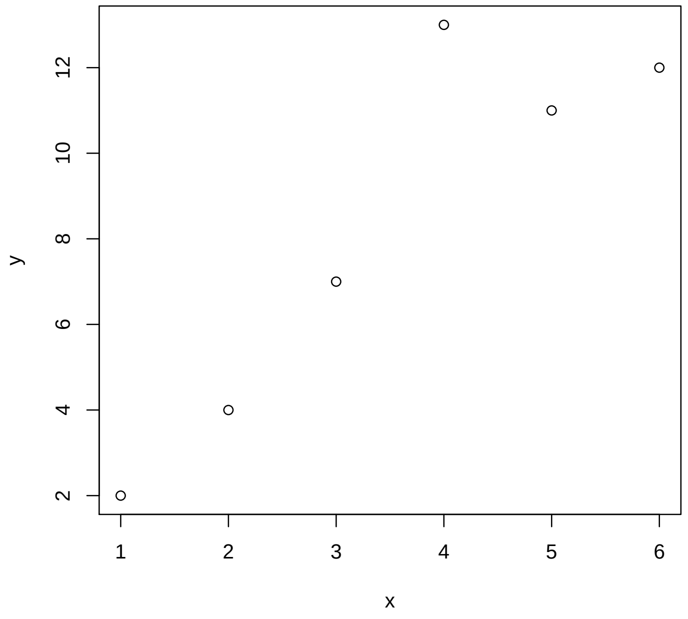
For example, I was looking for a package that implements the Theil slope estimator. This is a way of estimating the slope of a regression line that is not affected by outliers. I did a web search for "theil slope R CRAN" and found it in several packages, including one called `zyp`.

I clicked on the Packages tab in the bottom-right window, then clicked Install Packages at the top left. In the dialog box that popped up, I typed `zyp` in the Packages line and left everything else as is. Once you click Install, you'll see a whole lot of red text in the Console window, finishing with the word DONE. I saw `DONE (zyp)` on mine.

Now we have to start up the package we downloaded, and figure out what function we need from it. In the Packages window, scroll down through the list of packages until you find the one you want. Mine, of course, is right at the bottom. Click the checkbox to the left of its name to make it available (the command-line command is called `library`), and click the blue package name to go to its help. You'll see the names of a lot of functions. Going to the name of the package gives you a description of the whole package, which might give you a clue as to which function you need. You can go back to the "front page" of the package help using the back arrow at the top left of what has now become a Help window.

I think I want `zyp.sen`, so I click on its name to read its help, and find that it works a lot like `lm`. Let's test it out. First, some data and a scatterplot:

```
> x=1:6
> y=c(2,4,7,13,11,12)
> plot(x,y)
```



You see that the y-value for $x = 4$ seems way off. What effect will that have on the line estimated this way? We'll also fit a regular regression line for comparison:

```
> xy.zs=zyp.sen(y~x)
> xy.zs
```

```
Call:
NULL

Coefficients:
Intercept          x
      0.5        2.0
```

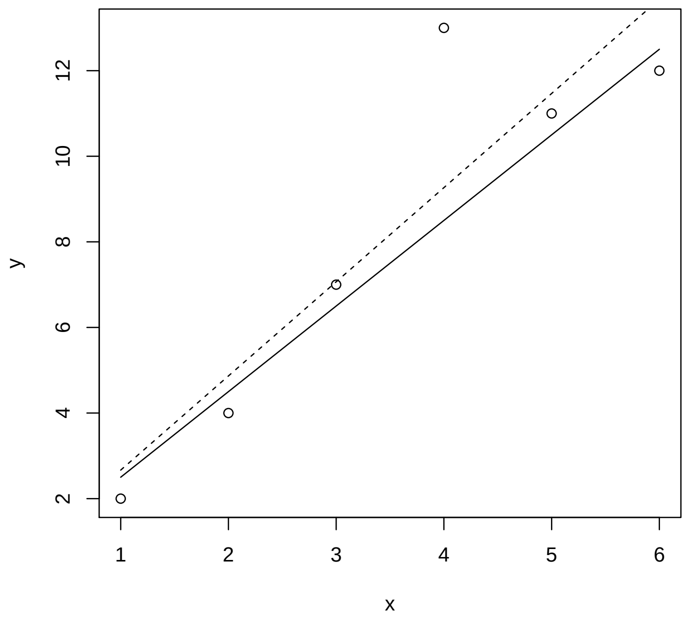
```
> xy.lm=lm(y~x)
> xy.lm
```

```
Call:
lm.default(formula = y ~ x)

Coefficients:
(Intercept)            x
     0.4667       2.2000
```

The regular regression line starts off a bit lower, but is a bit steeper.

Also, a plot with both lines on it. Note the use of `lty` to plot the lines differently. Also, `zyp.sen` only gives me the estimated intercept and slope, and not the fitted values, so I have to calculate those first:

```
> b=xy.zs$coefficients
> fit.zs=b[1]+b[2]*x
> fit.lm=fitted.values(xy.lm)
> plot(x,y)
> lines(x,fit.zs,lty="solid")
> lines(x,fit.lm,lty="dashed")
```



The help for `lty` is in the help for `par`, where you can find the legitimate values for `lty`. In our case, the Theil-Sen line is solid, and the regression line is dashed. You can see that the regression line has been pulled upwards by the fourth point (the regression line tries to go close to *all* the points), but the Theil-Sen line seems to completely ignore that point, going close to the other five.

7.2 The function

The basic unit of code that you build yourself is the **function**. This is a way of putting together your ideas in a form that they can be re-used. Here's an example:

```
> twostats=function(x)
+ {
+   xbar=mean(x)
+   s=sd(x)
+   c(xbar,s)
+ }
```

The anatomy of a function is this:

- It all starts with the `function` line. Here, this says that you are defining a function called `twostats` (it can be anything you choose, except names that R already knows), and after that, in brackets, the things you'll feed into the function. In this case, there's just one thing, a variable's worth of data. What you call this is up to you.
- An open curly bracket. Anything from here up to the close curly bracket is part of the function. (If your function has only one line of stuff, the curly brackets are optional, but I tend to put them in anyway so as not to forget them when they are needed.)
- Some R statements, using your variable name(s) from the `function` line to refer to the input. I called my input `x`, so I'm referring to that when I calculate my mean and standard deviation.
- The last line of the function just appears to be printing out something. That something is what comes back from the function. In my case it's a vector of two numbers: the mean and SD glued together.
- Finally, a close curly bracket to balance the open one from earlier. This ends the definition of the function. As long as there are no errors, R will happily accept this, and the function is now defined and can be used.

Some examples of using the function below. It will find the mean and SD of anything you throw at it:

```
> twostats(cars$MPG)

[1] 24.760526  6.547314

> twostats(1:5)

[1] 3.000000 1.581139

> twostats(c(120,115,87,108))

[1] 107.50000  14.52584

> s=twostats(1:10)
> s
```

```
[1] 5.50000 3.02765
```

Notice that you can store the results in a variable, or even use the function you defined in an `apply`-type thing:

```
> cars.numeric[1:5,]

   MPG Weight Cylinders Horsepower
1 28.4   2.67         4         90
2 30.9   2.23         4         75
3 20.8   3.07         6         85
4 37.3   2.13         4         69
5 16.2   3.41         6        133

> apply(cars.numeric,2,twostats)

           MPG    Weight Cylinders Horsepower
[1,] 24.760526 2.8628947  5.394737  101.73684
[2,]  6.547314 0.7068704  1.603029   26.44493
```

This shows the mean and SD of the numeric variables in our `cars` data frame. Next, we'll find the mean and SD of MPG for each number of Cylinders:

```
> attach(cars.numeric)

The following object(s) are masked from 'cars (position 27)':

    Cylinders, Horsepower, MPG, Weight
The following object(s) are masked from 'cars (position 28)':

    Cylinders, Horsepower, MPG, Weight

> mylist=split(MPG,Cylinders)
> mylist

$`4`
 [1] 28.4 30.9 37.3 31.9 34.2 34.1 30.5 26.5 35.1 31.5 27.2 21.6 31.8 27.5 30.0
[16] 27.4 21.5 33.5 29.5

$`5`
[1] 20.3
```

```
$`6`
 [1] 20.8 16.2 18.6 22.0 28.8 26.8 18.1 20.6 17.0 21.9

$`8`
[1] 16.9 19.2 18.5 17.6 18.2 15.5 16.5 17.0
```

```
> sapply(mylist,twostats)
```

```
             4    5         6         8
[1,] 30.021053 20.3 21.080000 17.425000
[2,]  4.182447   NA  4.077526  1.192536
```

```
> detach(cars.numeric)
```

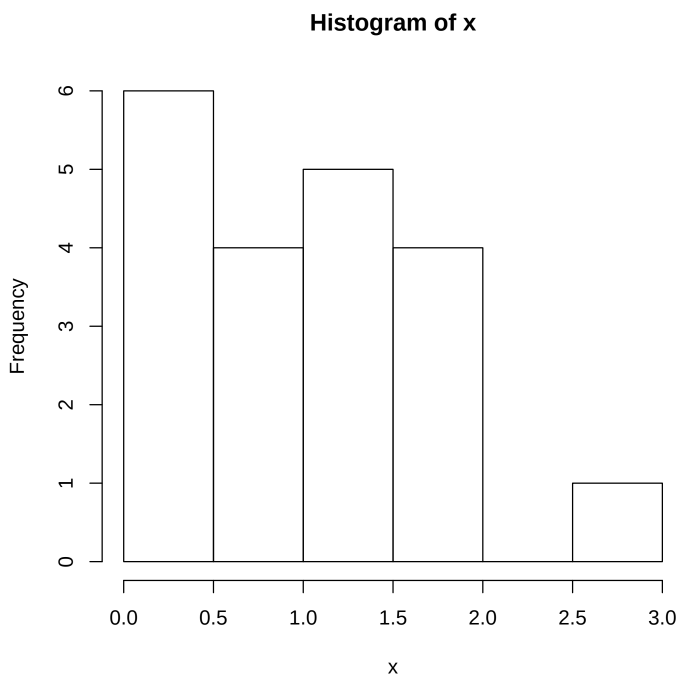
These are the mean (top) and SD (bottom) MPG values for each number of cylinders. (You can see that if I'd given the result of my `mystats` function a `names` attribute, I wouldn't have had to remember which was the mean and which the SD.)

7.3 The power of functions

The power of using functions is that you can encapsulate a whole complicated calculation into one "bundle", and then use the function you defined to repeat the whole complicated thing over again without having to worry about its innards.

Here's an example of that. Here's a skewed distribution, which I'm drawing a sample of 20 values from:

```
> set.seed(457299)
> x=rexp(20,1)
> hist(x)
```



Drawing a histogram of this sample reveals how skewed it is.

Now, suppose I want to see what kind of **means** I might get if I draw random samples (repeatedly) from this distribution. So first I define a function to draw a random sample of size nn from this distribution and return the mean. Having written it, I test it a few times:

```
> exp.mean=function(nn)
+   {
+     x=rexp(nn,1)
+     mean(x)
+   }
> exp.mean(20)

[1] 1.040792

> exp.mean(20)

[1] 0.9468177

> exp.mean(20)
```

```
[1] 1.19596

> exp.mean(20)

[1] 1.103339
```

A rule of programming is to write the most general thing you think you might need. Here I let the sample size be input to the function, so if I want a sample size other than 20, I don't have to rewrite the function.

Here you see that the sample means vary a bit, because by random sampling you're going to get different members of the population every time, so the sample mean is going to be (a bit) different.

Having defined our function to return the mean of one random sample, it's no problem to have it return 100 sample means and draw a histogram of them. There are actually two ways to do it. The first uses a "loop", which you might be familiar with if you've ever done any programming:

```
> ans=numeric(0)
> for (i in 1:100)
+   {
+     m=exp.mean(20)
+     ans=c(ans,m)
+   }
> ans

  [1] 0.5953901 0.7307934 0.9525824 1.6137642 0.7524616 1.1137032 0.9765812
  [8] 1.4119415 0.5974550 0.9124757 1.0699372 0.9505348 0.9222793 1.3680092
 [15] 0.8006096 0.7144267 0.8558708 1.0886128 1.4352454 0.9389111 0.9393441
 [22] 1.0957610 1.1482924 0.9526408 0.9004013 1.1681222 0.9544354 1.0444964
 [29] 1.0916456 0.9978768 1.3425093 0.6907574 0.8646838 0.7536693 1.0122495
 [36] 1.1136691 1.1365984 1.1487547 0.9507187 0.6081595 0.8252231 0.9020131
 [43] 1.4331116 1.0429275 0.9642786 0.8598349 0.9810133 1.0846337 1.0230059
 [50] 0.8728306 1.1340795 1.1464560 0.9134928 1.0513848 0.9010568 0.6627840
 [57] 1.2432685 1.1283063 1.2159073 0.8719889 1.0591415 1.0294409 1.0025213
 [64] 0.8845995 1.1100575 0.6199469 0.8608528 1.3606431 1.0278700 0.7397541
 [71] 1.0891893 1.1737105 1.2101996 1.0549312 1.0148108 1.5922892 0.9374209
 [78] 1.0165927 0.6533772 0.8379020 0.6216183 1.3011871 0.8544172 1.0176961
 [85] 1.0849363 1.0959883 0.8100988 1.0634409 0.7917569 0.9176847 0.8928590
 [92] 1.2463448 1.2642569 1.3123927 0.6834713 1.5867340 0.8413346 0.9662048
 [99] 1.0623588 0.7816818
```

First we initialize the variable `ans`, which is going to hold our results. The `numeric(0)` thing says that the variable is initialized to be empty, but "if there was anything in it, they would be numbers". Then the loop: "do the following 100 times", enclosed in curly brackets. Call our function to get a mean of a random sample from this skewed population (notice how the innards of the function are irrelevant to us here), and then glue the result onto the end of `ans`. Finally, display our results.

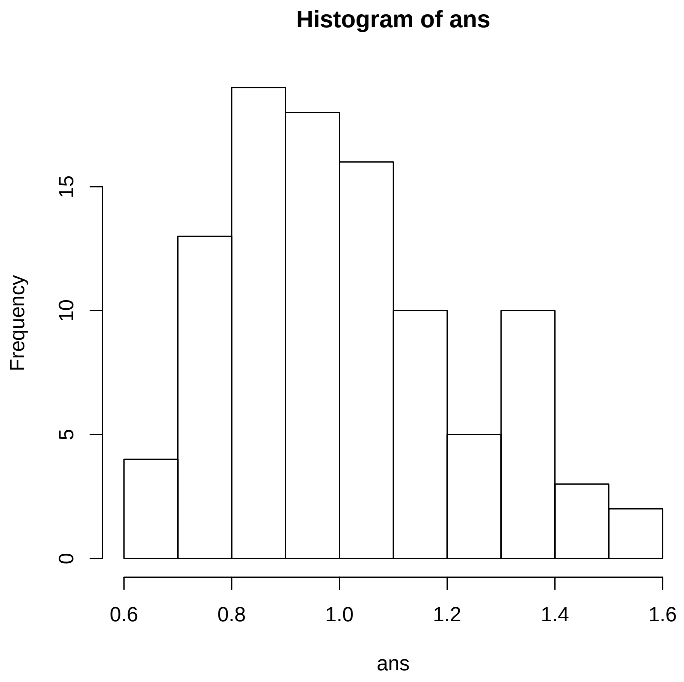
Eyeballing the results, you can see that even though the individual values in the sample might be quite a bit bigger than 1, the sample means are all somewhere near 1. Before we look at the shape of the distribution of sample means, though, let's have a look at a non-loop way of doing this, which, in R, is cleaner (and actually quite a bit quicker). `replicate` is a variant of `sapply` that will do the same thing as many times as you ask, which is ideal for simulations like this:

```
> ans=replicate(100,exp.mean(20))
> ans
```

```
  [1] 1.3943454 1.0094177 0.7623067 1.0895621 0.8797796 1.2504372 0.8224480
  [8] 1.4293340 0.8162799 0.8985124 0.6823115 1.4664997 1.0021541 0.7739709
 [15] 1.1292508 0.9969465 1.0242879 1.1791911 0.8603585 0.9837684 0.9236030
 [22] 0.8585515 1.3936862 1.3998916 0.8551937 0.6590934 1.0599416 0.8677731
 [29] 0.7769904 1.0798930 0.6050184 0.9666763 1.2383593 1.1578533 1.4298924
 [36] 1.0373164 0.9608770 1.0629154 1.3359719 0.9773096 0.9405015 0.8510367
 [43] 1.2117225 0.9608114 1.0536352 0.9998788 0.9753906 0.6705000 0.9312135
 [50] 0.8899296 0.9802517 0.8522665 0.7121971 1.3353620 0.7285403 0.8767662
 [57] 1.3971567 0.7000580 1.1044345 1.0163699 1.1503674 0.9767239 0.8036265
 [64] 0.9151995 1.1805739 1.0211275 1.2110536 1.3961973 1.1133059 0.9384418
 [71] 0.9363415 0.7841919 1.0460548 0.7467652 1.5394018 0.7525785 1.0731557
 [78] 1.1980688 1.3522068 1.3650985 1.0756254 0.7786717 0.8419265 0.9600557
 [85] 0.8368524 0.8740481 0.8713538 1.5844980 1.1846367 0.7259034 1.1165755
 [92] 1.0112292 0.7135493 1.0440840 1.3413771 0.8946160 0.8159957 1.2545248
 [99] 0.7803344 0.9586299
```

The answers are not the same, of course, but they are qualitatively similar (meaning, they look almost the same). The original population was seriously skewed. How about these sample means? Look at a histogram:

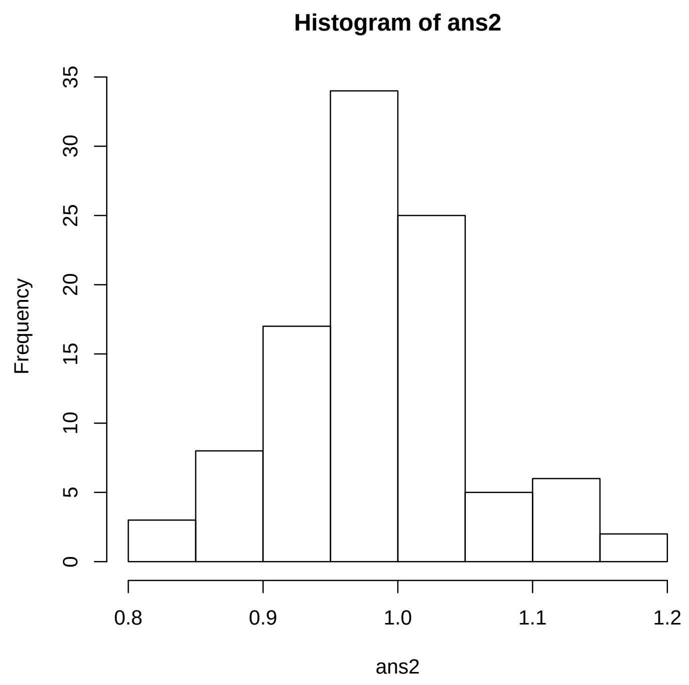
```
> hist(ans)
```



This distribution is a good bit less skewed than the population was. In fact, it looks almost symmetric.

What happens if we want to take samples of size 200 instead of 20? No problem: change 20 to 200 in the call to `replicate`, is all:

```
> ans2=replicate(100,exp.mean(200))
> hist(ans2)
```



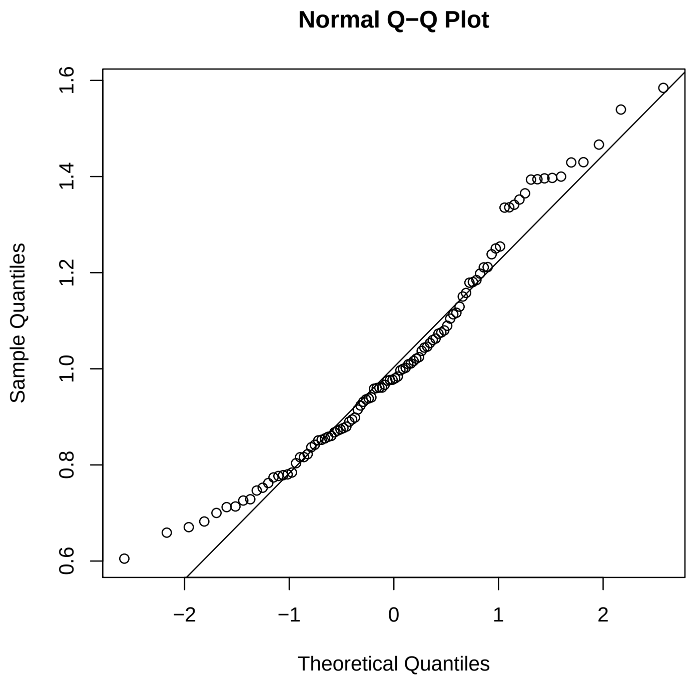
What does that look like? Take a chocolate-chip cookie if you said "normal distribution". The skewness has pretty much gone away.

This is the reason that we can use trhings like t-tests for means even if the population is far from normal: what matters is how normal the distribution *of the sample means* is.

There is a mathematical result saying that if you draw a sample from any population at all (of the kind that you might encounter in practice), and you have a "large" sample, the distribution of sample means will be approximately normal. It's called the Central Limit Theorem. In our example, samples of size 20 were on the edge but samples of size 200 were definitely large enough for the normality to work.

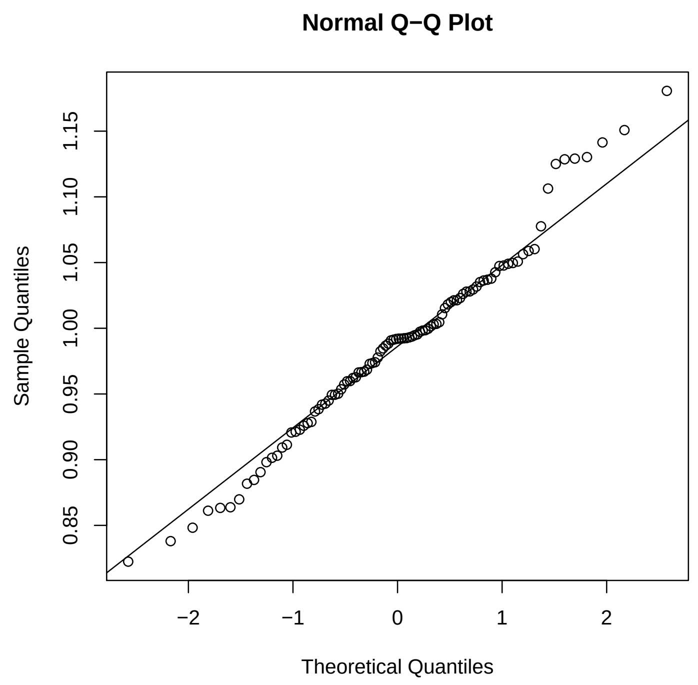
This is emphasized in normal QQ plots for the sample means. First, the samples of size 20:

```
> qqnorm(ans)
> qqline(ans)
```



which is maybe a bit skewed, and then the samples of size 200:

```
> qqnorm(ans2)
> qqline(ans2)
```



which is pretty darn normal.

The power of functions, in summary: define a function to do a simple thing, and then call that function to do complicated things. Which then becomes a simple thing if you define a function to do it. Rinse and repeat.

7.4 Lists

You may have noticed that some of R's functions return a bunch of things. For example, `lm` returns regression coefficients, test statistics, P-values, residuals and the like. These are not all the same kinds of things: the residuals are a vector as long as you have data points; the coefficients are a vector as long as you have explanatory variables, the F-statistic is a number, and so on. A data frame requires everything to be the same size, so these cannot be represented in a data frame. What R does, and what you can do as well, is to use a data structure called a `list`.

Figure 7.1 shows the basic procedure. I invented a variable called `x` which is just the numbers 1 through 10, and I found the mean and standard deviation of `x`. Then I make a list consisting of those two numbers. The resulting list,

```
> x=1:10
> mean(x)

[1] 5.5

> sd(x)

[1] 3.02765

> mylist=list(mean(x),sd(x))
> mylist

[[1]]
[1] 5.5

[[2]]
[1] 3.02765

> mylist[[2]]

[1] 3.02765

> mybetterlist=list(xbar=mean(x),s=sd(x))
> mybetterlist

$xbar
[1] 5.5

$s
[1] 3.02765

> mybetterlist$s

[1] 3.02765
```

Figure 7.1: mean and standard deviation in a list, part 1

called `mylist`, has two things in it, numbered `[[1]]` and `[[2]]`. The next line pulls out the standard deviation.

My second attempt gives the things in the list names. The definition of `mybetterlist` shows how it goes. It's the same way you construct a data frame where the things in it (variables) have names. The names show up when you print the list, and you can access the elements individually by referring to them by name.

So when you want to return a bunch of disparate things from a function, you can do that using a list as well. Let's make a function that returns the mean, SD and five-number summary of a list of numbers, and test it on the variable `x` that we defined in Figure 7.1. The procedure is shown in Figure 7.2. First we obtain the things we need (remember that the `quantile` function obtains percentiles), and then we put them together into a list, with the components of the list having names. Then we test it on our `x`, and also illustrate pulling out just one thing.

The last line gives the "structure" of our list object; that is, what it has in it. There are three things in our list: `xbar` and `s`, which are numbers (`num`), and also `five`, which is a vector of 5 numbers (hence the `1:5`) that has a names attribute (there are also 5 names, with the first few being shown).

So when you do `fit$coefficients` to get the regression coefficients out of a fitted model object, you are actually accessing a list. As an illustration of this on a two-sample *t*-test, see Figure 7.3. Most of these you can guess. `chr` means "character string". So the last line pulls out the string describing the alternative hypothesis, which is the default since we didn't change it.

This means that if you just need one piece of a complicated object, such as for example the P-value, as for example when you're doing a power analysis by simulation, you now know how to get it. (It seems wasteful to do a whole two-sample *t*-test and just pull out one part of the result, but better to have the computer waste some electrons than to have you waste brain power!)

Elements of lists can even be lists themselves. One of the parts of an `lm` model fit is like that. To illustrate on the same data as we used for the two-sample *t*-test, see Figure 7.4.

Lists are R's most flexible data structure. Data frames are a special case of lists; they are lists in which all the elements are vectors of the same length. Data frames have some special "stuff" — for example, when you print a data frame it looks different from when you print a list — but the notation for examining one variable from a data frame is the same as for examining one (named) element from a list.

One more advantage to returning a list from a function is shown in Figure 7.5. If you want to apply a function on all the variables in a data frame, `sapply` will then make you a nice table of the results.

```
> list.stats=function(x)
+   {
+     m=mean(x)
+     sdev=sd(x)
+     q=quantile(x,c(0,0.25,0.5,0.75,1))
+     list(xbar=m,s=sdev,five=q)
+   }
> st=list.stats(x)
> st

$xbar
[1] 5.5

$s
[1] 3.02765

$five
   0%   25%   50%   75%  100%
 1.00  3.25  5.50  7.75 10.00

> st$five

   0%   25%   50%   75%  100%
 1.00  3.25  5.50  7.75 10.00

> str(st)

List of 3
 $ xbar: num 5.5
 $ s   : num 3.03
 $ five: Named num [1:5] 1 3.25 5.5 7.75 10
  ..- attr(*, "names")= chr [1:5] "0%" "25%" "50%" "75%" ...
```

Figure 7.2: Mean, standard deviation and five-number summary in a list, part 2

```
> x=factor(c(1,1,1,1,2,2,2,2))
> y=c(5,7,8,10,9,11,13,14)
> fit=t.test(y~x)
> str(fit)

List of 9
 $ statistic  : Named num -2.79
  ..- attr(*, "names")= chr "t"
 $ parameter  : Named num 5.98
  ..- attr(*, "names")= chr "df"
 $ p.value    : num 0.0315
 $ conf.int   : atomic [1:2] -7.975 -0.525
  ..- attr(*, "conf.level")= num 0.95
 $ estimate   : Named num [1:2] 7.5 11.8
  ..- attr(*, "names")= chr [1:2] "mean in group 1" "mean in group 2"
 $ null.value : Named num 0
  ..- attr(*, "names")= chr "difference in means"
 $ alternative: chr "two.sided"
 $ method     : chr "Welch Two Sample t-test"
 $ data.name  : chr "y by x"
 - attr(*, "class")= chr "htest"

> fit$alternative

[1] "two.sided"
```

Figure 7.3: Structure of `t.test` model object

```
> fit1=lm(y~x)
> fit1$model$x

[1] 1 1 1 1 2 2 2 2
Levels: 1 2

> fit1$model$y

[1]  5  7  8 10  9 11 13 14
```

Figure 7.4: Part of an `lm` model fit

```
> q=function(x)
+   {
+     list(xbar=mean(x),s=sd(x),m=median(x),iqr=IQR(x))
+   }
> x=1:10
> y=15:24
> d=data.frame(x=x,y=y)
> sapply(d,q)

     x       y
xbar 5.5     19.5
s    3.02765 3.02765
m    5.5     19.5
iqr  4.5     4.5
```

Figure 7.5: Returning a list from a function

Chapter 8

R Studio, and serious work with it

8.1 R Studio

******** folders

You may have been trying out R Studio by using the "console" at the bottom left to enter commands. This works, but it's no advance over the unadorned R interface, and doesn't let you save your commands or output or graphs. The R Studio interface has four parts. Let's take them in turn.

Top left is a Notepad-like editor window. You can use this for a couple of things:

- a place to save (text) output from the bottom-left Console window that you would like to keep.
- a place to put commands that you might want to run again. I use this a lot, as a record of what I've done so that I can do it again.

To create a new window here, select File and New, and choose Text File for the first kind of file, and R Script for the second. (You can also use a text file as a place to make notes.)

R Studio has all kinds of commands to make an R Script window useful. Some are:

- Move the cursor to a line and press Control-Enter to run the R code on that line.

- Select a block of text and press Control-Enter to run the code you selected.
- Select the Source button at the top right of an R Script window to execute the commands in the whole file. This is an easy way to re-do a whole analysis.

Bottom left is the console window, where R commands actually get run, and textual output appears. You can enter R commands at the prompt in the Console window, or run them from an R Script window using Control-Enter or Source. To navigate between windows (usually an R Script window and the Console), click on the place you want to be. I often find myself trying to type a command in the Console window that I wanted to go in an R Script window, and vice versa.

The console window has a command history. Use the up and down arrows to scroll through commands that you ran previously. There is also "tab-completion"; type part of a command and then press the TAB key. You'll get a list of possible completions of commands that begin with what you typed. You can use the up and down arrows to move between the possible completions. Further to the right is a brief description of what the currently selected completion does. If you want to know more, you can press F1 and the help for this function appears in the bottom right window.

The top right window has a dual purpose. By default (Workspace tab) it holds a list of all the variables you have in your R workspace. If the variable has a single-number value like "7", the value will appear in this window. If a variable contains a collection of values (an R vector), you'll see something like `numeric[5]`, meaning a vector with 5 numbers in it. R Studio seems to distinguish between "data" (matrices and data frames) and "values" (vectors). Either way, double-clicking on a variable will display its value(s).

The other purpose of the top right window (History tab) is to show the R commands you've typed, run or otherwise constructed. The commands are listed with the most recent at the bottom. You can also search for commands matching something you type (using the magnifying glass box at the top right of this window). You can select one or more or these commands and run them again (by clicking on To Console) or send them to your top-left code window (click on To Source).

The bottom-right window is a kind of grab-bag for everything else. Let's take the tabs in turn:

Files lists the files in your current folder/directory. Clicking on a file opens it: in the top-left Files window if it's a text file, in its own window if it's a picture. You can navigate up a folder, or click on the folder names above the list of files to navigate to another directory entirely. Plus the usual stuff: create a new folder, delete files, rename files and so on.

Plots Whenever you make a graph (histogram, scatter plot or whatever), it appears in the bottom right window on the Plots tab. When you create a new plot, the Plots tab pops to the front and shows you your new graph. You can use the arrows at the top left of this window to cycle through your previous graphs (R Studio appears to keep about 30 of them around). The Export button allows you to save a graph to a file, or to copy it to the clipboard for pasting into something else.

Packages are libraries of R code (functions, data and so on) that people have created and shared on the Comprehensive R Archive Network. There are all kinds of packages to do all kinds of things: some that are subject-area specific, and some that have not-quite-mainstream stuff that hasn't made it into the "base" part of R. By clicking on the name of a package here you can access its help files, for the package itself and for any functions (or other objects) contained within it. This is a nice way to get to the help for a package you've installed. Installing and using packages is described in Section 7.1. There's a button at the top left of this window that helps you install packages.

Help What it says. Any time you access a help file, either by something like `?lm` from the Console window, or by navigating through package help files, you'll find your way here. There is also a help search, with completion: if you type part of a function name, all the functions that start with those letters are shown, and you can use the mouse or the up/down arrows to select one, whose help is then displayed. The search only matches the beginnings of function names, though; I haven't been able to find a subject-based search here. In that case, you can fire up your favourite search engine and enter something like `multiple regression r`, which usually turns up something useful: a name of a function or a package or something.

8.2 Projects

If you are doing a serious piece of work with R and you want to keep everything together, R Studio provides a mechanism to do that, by means of its "Project" feature. This will help you keep your commands, variables and so on together. All the stuff you need is in the Project menu.

To start a new project, select New Project from the Project menu. It's sensible to keep all the stuff for a project together in one folder. To this end, R Studio offers you the choice of creating a new folder or using an existing one.

Chapter 9

Literate programming and Sweave

9.1 Introduction

One of the hazards of using statistical software is that you have to copy and paste your results into whatever you are using to write up your results. It is altogether too easy to have the "wrong" version of an analysis in your document, because you changed your mind about which data to use, or which version of an analysis to conduct, and these can be very hard to catch.

If you use LaTeX , as I am right now, you can combine LaTeX with R using software called Sweave. R Studio makes this (relatively) painless. The idea is that you embed specially marked pieces of text called **code chunks** into your document. Then R is run on your code chunks, and the code and resulting output are embedded into your document. (This is configurable on a per-chunk basis; you can choose not to show the commands or output. See below.)

To kick things off, you need to create a LaTeX document. (I'm assuming you know something about LaTeX for this, and have it installed on your computer.) In R Studio, select File and New, and select "R Sweave", which, on my R Studio, is the top one of the bottom block of four choices. This gives you a skeleton LaTeX document that looks like this, with some blank lines excised:

```
\documentclass{article}

\begin{document}
```

```
<<>>=

@
```

Figure 9.1: Empty code chunk

```
\end{document}
```

After the `SweaveOpts` line, you can enter any text or LaTeX commands you choose. R Studio makes the latter easier for you; find the Format button above the window you're typing in, click it, and that pops down a menu of LaTeX things you might want to do. In the case of our example document, we can do something like this:

```
\documentclass{article}

\begin{document}


\section{Introduction}

This is some random text here.

\end{document}
```

9.2 Code chunks

Now, we can add a code chunk. Let's add some data (two variables) and calculate summaries for them. To insert a code chunk, find the Chunks button top right of where you are typing, and, from its pop-down menu, select Insert Chunk. If you like the keyboard better, pressing control-alt-I will insert a code chunk as well. You'll see something like Figure 9.1 appear in your document. The first line marks the beginning of your code chunk, and the `@` marks the end. In between you can put any R commands you like.

Let's invent some data, print it out and produce a summary. That will come out looking like Figure 9.2.

You can imagine code chunks as being linked together. That is, if you now have another code chunk that refers to `x` and `y`, they'll be the same `x` and `y` that you created before. Let's make a scatter plot of our two variables. That produces Figure 9.3.

```
<<>>=
x=1:10
y=c(4,5,8,11,12,14,15,16,21,21)
x
y
summary(y)
@

> x=1:10
> y=c(4,5,8,11,12,14,15,16,21,21)
> x

 [1]  1  2  3  4  5  6  7  8  9 10

> y

 [1]  4  5  8 11 12 14 15 16 21 21

> summary(y)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   4.00    8.75   13.00   12.70   15.75   21.00
```

Figure 9.2: Code chunk and output to define data, print, and summarize

```
<<fig=TRUE>>=
plot(x,y)
@
```

```
> plot(x,y)
```

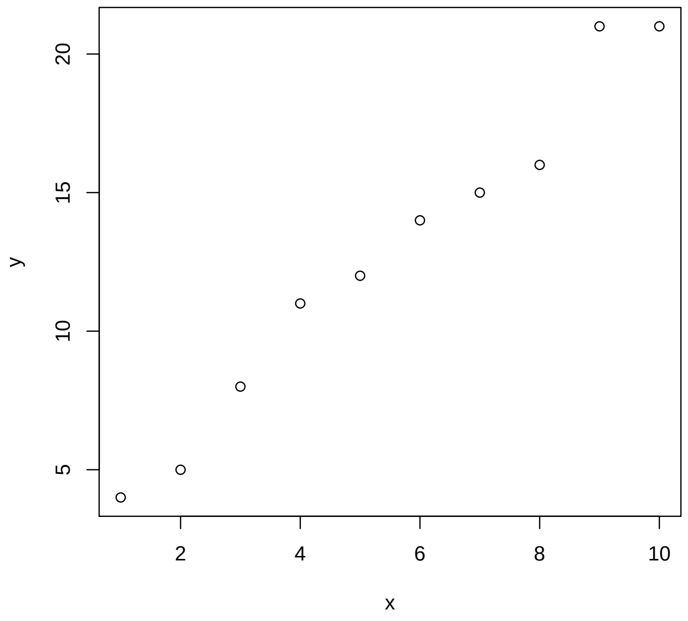


Figure 9.3: Plot of y vs. x

In between the angle brackets at the start of a code chunk, there are some options you can place. If you are making a plot, you must put the `fig=TRUE` in there. If not, you must not. Getting this wrong produces some hard-to-find errors. (I know, from painful personal experience!)

Also, there can be only one plot per code chunk. If you have more than one, you need to put them in separate code chunks. Sometimes, as when you make a plot of a regression object (to get those residual plots), one command makes more than one plot. If you don't take special measures, you'll only see the first of the plots. What you have to do is to get them all (in that case there are four plots) is to issue a command like `par(mfrow=c(r,c))` before you call for the plot. This produces an array of plots all on one page, where you replace `r` by the number of rows you want in your array, and `c` by the number of columns. In the case of the four regression plots, you probably want a 2 by 2 array; doing something else, like the code chunk and output in Figure 9.4, will look silly.

Note that when you type `fig=`, RStudio will offer you a choice of `TRUE` and `FALSE`. The page of plots looks silly because we made a 3-by-3 array to hold our plots, and then there were only four of them.

What happens if we draw another plot now? As shown in Figure 9.5, this "undoes" the `mfrow` thing and starts a new plot. That is, `mfrow` only lasts for the current code chunk, but it will keep filling with plots during the current code chunk. Example of that is shown in Figure 9.6.

This is the exception to "one plot per code chunk". The rule is really "one page of plots per code chunk." I'm not sure I really want the plots as elongated as that, but that's how they came out. If you flip the things fed into `mfrow` around, you get Figure 9.7 instead. I think I like this even less.

A squarish collection of plots seems to work best, as shown in Figure 9.8.

There are lots of options you can specify for a code chunk. Let's imagine a code chunk that just prints `x` and `y`. This is in Figure 9.9.

Nothing special here. But you might want to hide the code that produced the output, which happens like this, as shown in Figure 9.10.

Or you might want to show the code and hide the output (if, for example, you're talking about your programming strategy, admittedly a daft idea in this case), with the results shown in Figure 9.11:

Or, if you're really crazy, you can do this, with code chunk and "results" shown in Figure 9.12:

```
<<fig=TRUE>>=
par(mfrow=c(3,3))
test.lm=lm(y~x)
summary(test.lm)
plot(test.lm)
@
```

```
> par(mfrow=c(3,3))
> test.lm=lm(y~x)
> summary(test.lm)

Call:
lm.default(formula = y ~ x)

Residuals:
     Min       1Q   Median       3Q      Max
-1.53333 -0.55000  0.06667  0.31667  1.53333

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   2.0667     0.6716   3.077   0.0152 *
x             1.9333     0.1082  17.861 9.89e-08 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.9832 on 8 degrees of freedom
Multiple R-squared: 0.9755,     Adjusted R-squared: 0.9725
F-statistic:   319 on 1 and 8 DF,  p-value: 9.893e-08

> plot(test.lm)
```

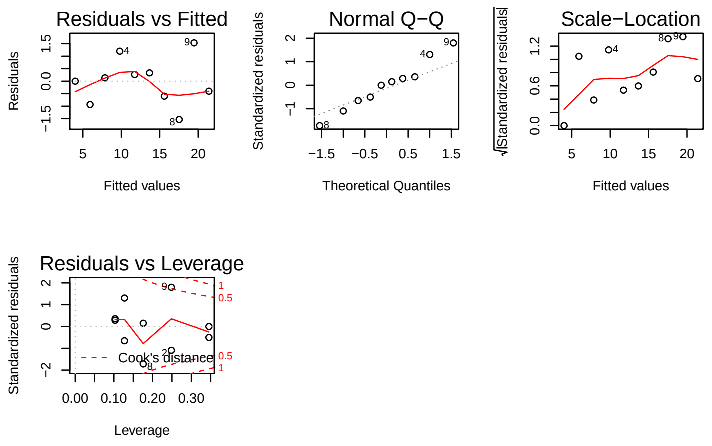


Figure 9.4: Plot array with silly choice of rows and columns

```
<<fig=TRUE>>=
qqnorm(y)
qqline(y)
@
```

```
> qqnorm(y)
> qqline(y)
```

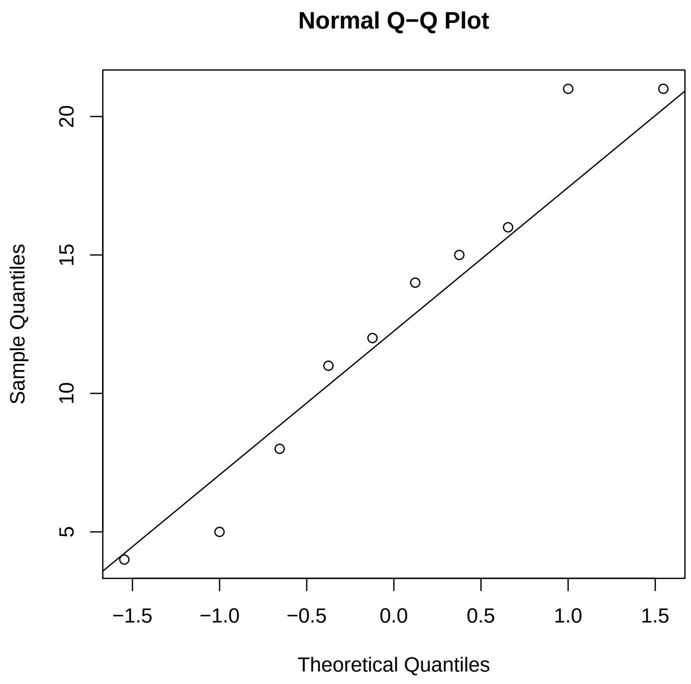


Figure 9.5: Another plot after `par(mfrow=c(3,3))`

```
<<fig=TRUE>>=
par(mfrow=c(1,2))
boxplot(y)
qqnorm(y)
qqline(y)
@
```

```
> par(mfrow=c(1,2))
> boxplot(y)
> qqnorm(y)
> qqline(y)
```

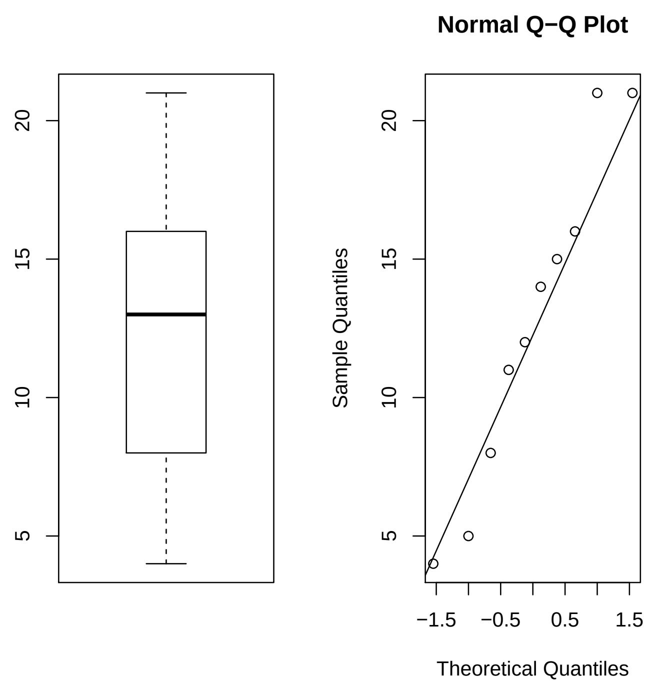


Figure 9.6: Multiple plots of different types

```
> par(mfrow=c(2,1))
> boxplot(y)
> qqnorm(y)
> qqline(y)
```

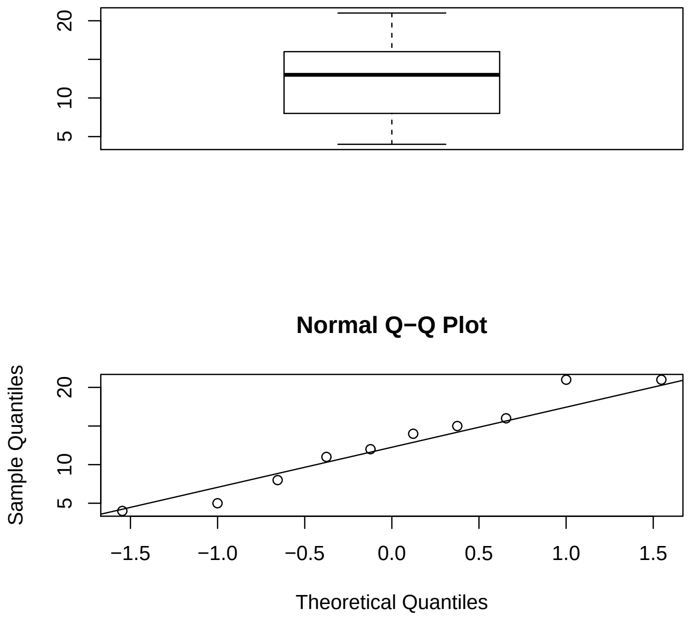


Figure 9.7: 2-by-1 array of plots

```
> par(mfrow=c(2,3))
> for (i in 1:6)
+   {
+     zzz=rnorm(100,10,3)
+     qqnorm(zzz)
+     qqline(zzz)
+   }
```

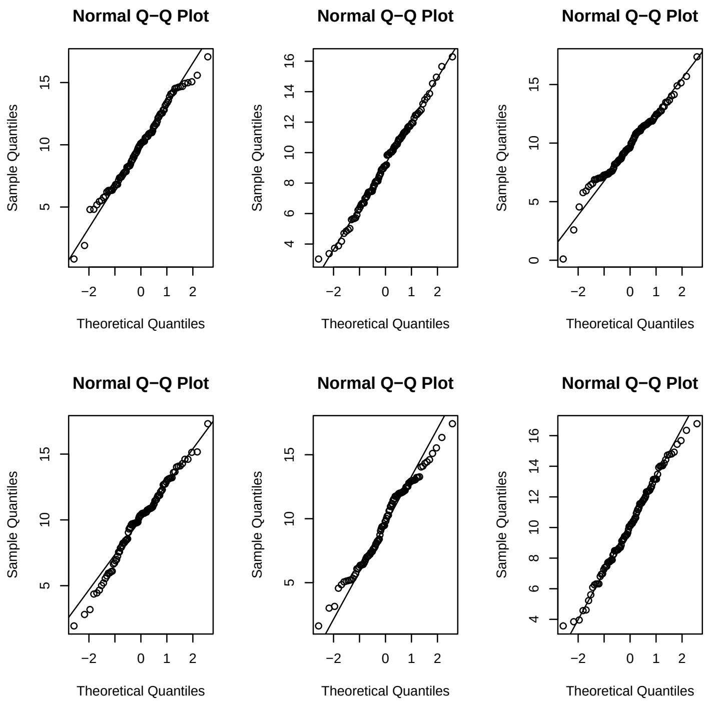


Figure 9.8: A rectangular array of plots

```
<<>>=
x
y
@

> x

 [1]  1  2  3  4  5  6  7  8  9 10

> y

 [1]  4  5  8 11 12 14 15 16 21 21
```

Figure 9.9: x and y

```
<<echo=FALSE>>=
x
y
@

 [1]  1  2  3  4  5  6  7  8  9 10

 [1]  4  5  8 11 12 14 15 16 21 21
```

Figure 9.10: Hiding the code

```
<<eval=FALSE>>=
x
y
@

> x
> y
```

Figure 9.11: Output hidden

```
<<echo=FALSE,eval=FALSE>>=
x
y
@
```

Figure 9.12: Hiding the code and the results

9.3 More code chunk options

Code chunks have have a **label**, so that you can refer to them again later. Specifying a label is easy: any code chunk option without an = in it is considered to be a label. Below, A is a label:

```
<<A>>=
x+1
@
```

```
> x+1
```

```
 [1]  2  3  4  5  6  7  8  9 10 11
```

You can re-use labelled code chunks inside another code chunk like this:

```
<<>>=
<<A>>
y+3
@
```

```
> x+1
```

```
 [1]  2  3  4  5  6  7  8  9 10 11
```

```
> y+3
```

```
 [1]  7  8 11 14 15 17 18 19 24 24
```

whereby x+1 gets printed out again. Note that code chunk A was replaced by what it produced.

You can also refer to an R variable in your text. What you do is to include \Sexpr{} in your text, and inside your curly brackets you can put anything that evaluates to a number, for example:

- a piece of arithmetic, like \Sexpr{1/7}, which comes out as 0.142857142857143.
- a "scalar" variable, that is, one that has only a single number in it. I have a variable called p floating around, so \Sexpr{p} gives 0.00390625. This is as opposed to a "vector" like my y above, which has several numbers in it.

- an element of a vector, like the 4th value in vector `y` above: `\Sexpr{y[4]}`, which gives 11.
- A single value from a data frame. For example, let's pull out the name and weight of the 15th car using `\Sexpr{cars[15,]$Car}` and `\Sexpr{cars[15,]$Weight}`: the 15th car is Dodge Colt and it weighs 1.915 tons.

You'll notice that numbers printed out this way often contain too many digits. The R command `format` is your friend here. It has two options that are often useful, `digits` to specify the number of significant digits, and `scientific=F` to suppress printing of things in the E notation (for very small numbers). You can either use the call to `format` inside your `\Sexpr`, like this:

`Pi is close to \Sexpr{format(3+1/7,digits=4)}` produces "Pi is close to 3.143".

Or you can use a "hidden" code chunk, and do all the hard work in R, like this. It's best to assign the intermediate results to variables so that they don't get printed out by mistake:

```
<<echo=FALSE>>=
nearpi=3+1/7
appx=format(nearpi,digits=4)
@
```

and then say "The approximate answer is 3.143", using `\Sexp{appx}`.

With very small numbers, like P-values, you might end up with something like 1.030928e-04. Not only does this have too many digits, but the scientific notation does not (unless you are used to reading it) give you too good of a sense of how small the number really is. (The number is actually one divided by 9700.) To forbid scientific notation and print the number out starting with a bunch of zeros, if that's what it starts with, you can do this:

```
One over 9700 is approximately
\textbackslash Sexpr{format(1/9700,digits=3,scientific=F)}.
```

which gives:

One over 9700 is approximately 0.000103.

Or, again, you can do the `format` work in an R hidden code chunk, as above, saving the result in a variable `pval`, say:

```
<<echo=FALSE>>=
r=1/9700
```

```
pval=format(r,digits=3,scientific=F)
@
```

and then `\Sexpr{pval}` gives 0.000103.

9.4 Producing a document

There are two steps to turning your LaTeX and code chunks into a document. The first step is to let R loose on your code chunks and produce some output. This is done by an R function called `Sweave`, which creates chunks of LaTeX in your document in place of those code chunks. Then the second step is to run LaTeX on the result. Fortunately, R Studio automates this all for you. All you need to do is to click on the Compile PDF button above the document window, or, if you prefer keys, control-shift-i. R Studio then goes through the above two steps, and, if all goes well, it will pop up a PDF file with your compiled document.

If all doesn't go well, there will be errors. The first thing to figure out is whether you have an R error or a LaTeX error.

Recall the `x` and `y` from above, which are the same length, and think about the code below:

```
<<>>=
z=c(y,1)
lm(z~x)
@
```

`z` is a vector with one more number in it than `x`, and the response and explanatory variables have to have the same length in regression (the x's and y's have to be paired up). So this is an R error.

When you try to produce a PDF with this in it, in the bottom left window, under the "Compile PDF" tab, you'll get a line beginning with a red X telling you that "variable lengths differ" and the first line of the chunk where it happened. That has to be fixed before you get a PDF. You might also get an R warning, which is an X on a yellow background. This won't stop things running, but you should probably understand why it happening. (If you know that you're getting a warning but it won't affect your results, then you can leave it be.)

This is an R error too:

```
<<>>=
m=1:5
```

```
sd(m,opption=7)
@
```

because `sd` doesn't have an option by that name (there is not even an "option" spelled properly). The error is `unused argument(s) (opption=7)` with the line number (the start of the chunk), so that ought not to be too hard to find in your code.

If all is well with your R, you'll see, flashing past in the bottom left window, Sweave processing your code chunks. If at the end of this you see "You can now run (pdf) latex on...", your file with code chunks ran successfully through R, and there should be a `.tex` file containing the output from those code chunks.

In the bottom left window you'll see whether LaTeX ran on this document. There are two parts to the "compile PDF" tab (with buttons to switch between them at the top right of the window). The Output tab will show you your LaTeX document being processed. If there's an error, the Issues part will pop up, and your error(s) will appear, again with X's on a red background. (You might have to scroll past a bunch of warnings to see them. I always get "overfull hbox" and "underfull vbox" warnings when I run LaTeX. This just means that some lines came out too long or a page ought to have been filled up with more blocks of stuff, but putting anything else in would have made the page too long.) The line numbers referred to here are in the LaTeX file, which you may have to open to check. For example, my text and code chunks are in `r-howto.Rnw`, so my processed code chunks are in `r-howto.tex`, a file I never normally need to look at. But if there are LaTeX errors, I might need to.

LaTeX errors can come from two sources: the process of obtaining the R output (usually a missing or superfluous `fig=TRUE`), or a mistake by you in the LaTeX. When you've found out where in the LaTeX the errors are, you can go back to your `.Rnw` file and correct them there.

If a PDF file pops up, everything worked. Now you have the task of proofreading to make sure that the document says what you wanted it to say. (I am typing this on a GO bus, and when there is a bump in the road, there might be a typo, and I might catch it later, or not.)

Chapter 10

Logistic regression

10.1 Introduction

Regression has as its aim to predict a response variable from one or more explanatory variables. The response variable there is typically something measured. But what if the response variable is something *categorized*, for example, something like this:

```
dose status
0 lived
1 died
2 lived
3 lived
4 died
5 died
```

Each of 6 rats were given some dose of a particular poison, and the response recorded was whether each rat lived or died (after a certain time period). This reponse falls into one of two categories ("lived" or "died"), so an ordinary regression is not appropriate.

Enter **logistic regression**. In its basic incarnation, logistic regression works on a two-level response like this one, and predicts the *probability* of one of the levels, as it depends on the explanatory variable(s).

10.2 Odds and log-odds

You can skip this section on a first reading, though it's worth studying if you want to know what's really going on in logistic regression.

Probabilities are awkward things to predict, because they have to be between 0 and 1, and a prediction from an ordinary regression could be anything.

What turns out to be easier to work with is the **odds**. What's that? Well, suppose you have something that has probability $\frac{3}{4}$ of happening. Gamblers don't think in terms of probabilities: they say "this thing has 3 chances to happen and 1 chance not to, so the odds are 3 to 1 (on)". Likewise, something of probability $\frac{1}{10}$ has 1 chance to happen against 9 chances not to, so the odds are 1 to 9 (usually called "9 to 1 against").

Speaking mathematically for a moment, if p is a probability, the corresponding odds d are

$$d = \frac{p}{1-p}$$

.

You can check that $p = \frac{3}{4}$ corresponds to $d = 3/1 = 3$, and $p = \frac{1}{10}$ goes with $d = 1/9 = 0.11$.

We can even go a step further, and take the logarithm of the odds (usually the natural logarithm). This means that when $p > 0.5$, the odds will be greater than 1, and so the log-odds will be positive. When $p < 0.5$, the odds are less than 1, and so the log-odds will be negative.

The advantage of this for a modelling point of view is that *every* possible value for the log-odds, no matter how positive or negative, corresponds to a legitimate probability. So if you model the log-odds of the probability in terms of explanatory variables, you will always get a plausible predicted probability. This is what logistic regression does.

Handy formulas, with u denoting the log odds:

$$u = \ln\left(\frac{p}{1-p}\right) = \ln p - \ln(1-p)$$

and

$$p = \frac{e^u}{1+e^u} = 1/[(1+\exp(-u)].$$

For example, $p = 3/4 = 0.75$ goes with $u = 1.1$, and $p = 1/10 = 0.1$ goes with $u = -2.2$.

So when you have a logistic regression, you get the intercept and slope(s) for the log-odds of the success probability. R has a handy function `plogis` for con-

```
> x=1:3
> u=-0.1+0.5*x
> u

[1] 0.4 0.9 1.4

> plogis(u)

[1] 0.5986877 0.7109495 0.8021839

> exp(0.5)

[1] 1.648721
```

Figure 10.1: Calculations of estimated probabilities from a logistic regression

verting a log-odds to a probability, and also `qlogis` for converting a probability to a log-odds. Figure 10.1 shows some example calculations. Suppose a logistic regression has intercept -0.1 and slope 0.5. Then the predicted log-odds are shown as `u`. Note that, because the slope is 0.5, the log-odds increase by 0.5 each time `x` increases by 1. The predicted *probabilities* are shown underneath `u`. They are *not* equally spaced out; indeed, if they were, they would soon go over 1.

The steady 0.5 increase in log-odds is kind of hard to interpret. But if you add something to a log, you are multiplying by e-to-that-something. In our example, that means that every time you increase the log-odds by 0.5, you are multiplying the odds themselves by $\exp(0.5) = 1.65$. This provides a way of interpreting slopes in logistic regression.

10.3 Example 1: the rats

Let's see if we can produce an analysis of the data in Section 10. With only five rats, and a categorical lived/died response, we shouldn't expect to be able to conclude much, however. Figure 10.2 shows the result of plotting the rat data. This is surprisingly uninformative, given R's usual efforts at making plots. Status 1 is "died" and status 2 is "lived", so that with a little effort you can make out that higher doses tend to go with dying. I put a lowess curve on, which suggests the trend. Don't worry about it not being straight, because linearity is in log-odds, not in probabilities.

Another way of making a plot (for data like this) is to flip explanatory and response around and make a boxplot of dose as it depends on status. This is shown in Figure 10.3. A pause for consideration reveals that those rats who

```
> rats=read.table("rat.txt",header=T)
> rats

  dose status
1    0  lived
2    1   died
3    2  lived
4    3  lived
5    4   died
6    5   died

> attach(rats)
> plot(dose,status)
> lines(lowess(dose,status))
```

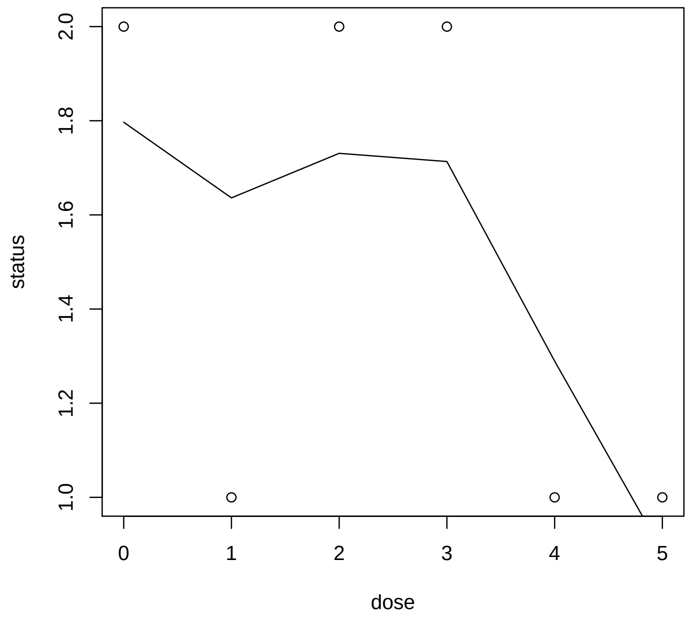


Figure 10.2: Plot 1 of rat data

```
> boxplot(dose~status)
```

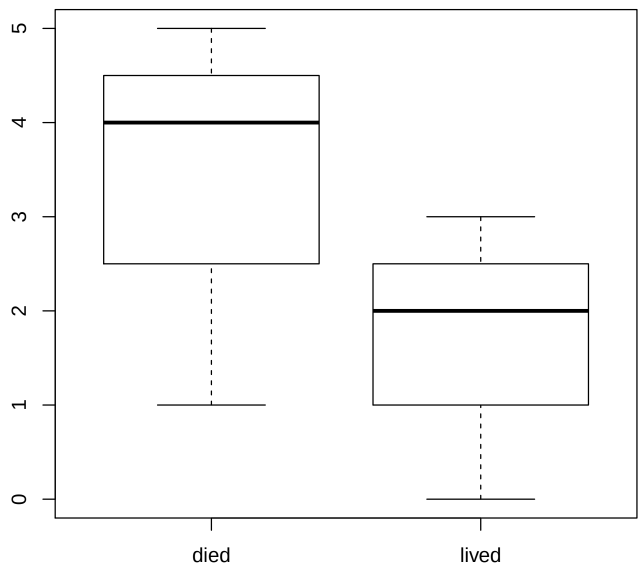


Figure 10.3: Plot 2 of rat data

died tended to get larger doses than those who lived. If the poison really is a poison, you'd expect more of it to tend to go with death, which is what the plot shows. In other words, you'd expect a larger dose to have a *decreasing* effect on the chances of survival.

All right, to the analysis. This is shown in Figure 10.4. First off, notice that we are now using `glm` instead if `lm`. A logistic regression is a so-called *generalized linear model* instead of a plain old linear model. (The `family="binomial"` thing on the first line gets a logistic regression as opposed to some other kind of generalized linear model; there are several.) But the output looks the same: there is an intercept and a slope, with a significance test for the slope. The slope is negative, but not significantly different from zero. But what are we actually predicting the probability of? From looking in the Details section of the help for `glm`, you find out a lot of things, including that in the response variable, the first category is considered to be failure and the other(s) are considered success, and R predicts the probability of success. The `levels` command shows that the first category is `died`, so we are predicting the probability of living. The

```
> rats.lr=glm(status~dose,data=rats,family="binomial")
> levels(status)

[1] "died"  "lived"

> summary(rats.lr)

Call:
glm(formula = status ~ dose, family = "binomial", data = rats)

Deviance Residuals:
      1        2        3        4        5        6
 0.5835  -1.6254   1.0381   1.3234  -0.7880  -0.5835

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   1.6841     1.7979   0.937    0.349
dose         -0.6736     0.6140  -1.097    0.273

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 8.3178  on 5  degrees of freedom
Residual deviance: 6.7728  on 4  degrees of freedom
AIC: 10.773

Number of Fisher Scoring iterations: 4

> rats.lr$fitted

        1         2         3         4         5         6
0.8434490 0.7331122 0.5834187 0.4165813 0.2668878 0.1565510

> detach(rats)
```

Figure 10.4: Logistic regression analysis of rat data

```
dose lived died
0 10 0
1 7 3
2 6 4
3 4 6
4 2 8
5 1 9
```

Figure 10.5: Consolidated data for the 60 rats

slope being negative indicates that the probability of surviving goes down as the dose increases, but its large P-value indicates that this trend is not statistically significant. This latter is hardly surprising given that we only have five data points. This is all you *need* to know, but if you want to know more you can go back and read Section 10.2.

You can get the fitted values the same way as you would from an `lm` object. These are shown in the last line of Figure 10.4, and you can see how the predicted survival probabilities go down from 0.84 at dose 0 to 0.16 at dose 5.

If you read Section 10.2, you'll realize that the predicted log-odds for a dose of 4 in the model above is about $u = 68 - 0.67(4) = -1.01$. Then we turn the log-odds into a probability by the formula $p = 1/(1 + \exp(-u))$, which here gives $p = 1/(1 + 2.78) = 0.27$. Also, the slope of -0.67 means that the odds of survival become $\exp(-0.67) = 0.51$ times greater, that is about 2 times smaller, each time the dose increases by 1. This is something that is interpretable.

10.4 Example 2: more rats

In the previous example, we said that it was hard to get a statistically significant relationship out of only 5 observations. So how about if we had 10 rats for each dose? We have a choice about how we lay out the data for this: we could have one rat on each line, with a "lived" or "died" on the end of each line, and 60 lines in the data file (and get very tired of typing "lived" or "died" each time.) Or, we could collect together all the rats that had the same dose, and say something like "out of the 10 rats that got dose 1, 7 of them lived and 3 died". This allows you to condense the data for the 60 rats into 6 lines, one for each dose, as shown in Figure 10.5. For R, the two columns of response are successes and failures.

The easiest way to get a plot for condensed data like this is to work out the proportion of rats surviving at each dose, and plot that against dose. I've drawn a lowess curve on this one as well, but the pattern is pretty clear. My `lprop` or "living proportion" is the number of survivors at each dose divided by the

```
> rat2=read.table("rat2.txt",header=T)
> attach(rat2)
> lprop=lived/(lived+died)
> plot(dose,lprop)
> lines(lowess(dose,lprop))
```

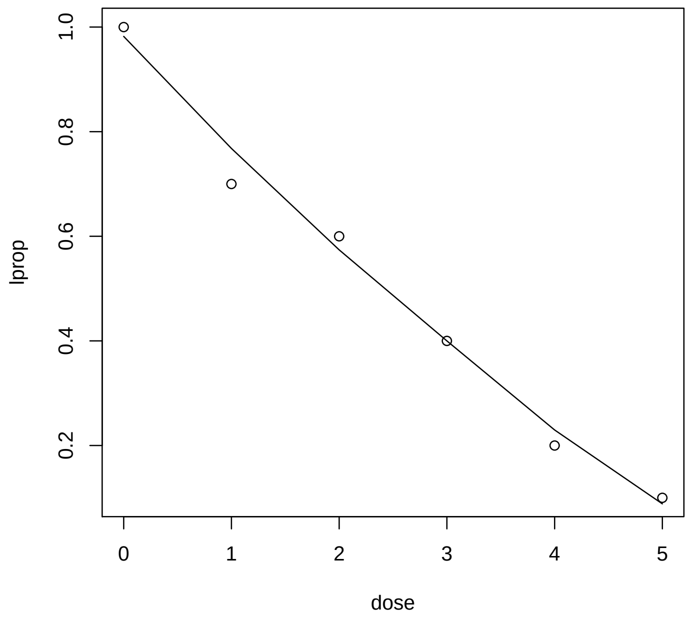


Figure 10.6: Plot for Example 2

total number of rats at that dose. (R makes this kind of thing easy because when you add or divide two vectors it adds or divides them element by element, which is usually exactly what you want.) Note that `lprop` can't go below zero, so that the trend *has* to curve once the dose gets big enough, even though it looks pretty straight here.

Now for the analysis, shown in Figure 10.7. We're using `glm` again, but now we have a twist in the way we specify the response: it's in *two* variables `lived` and `died`, so we have to `cbind` them together on the left side of the model formula. (Putting `lived+died` on the left doesn't work.) This combined response then depends on dose. The `summary` of the fitted model object indicates that `dose` once again has a negative slope, so that the survival probability does indeed go down as the dose increases, but this time the P-value is tiny, so that there is a definite effect of dose on survival probability, and it's not just chance.

You might recall that for plain-jane regression, there is an F-test for the regression as a whole, which is identical to the t-test for the slope when there is only one slope. There is something similar here, except that it is no longer identical. To get the test for the model as a whole, we first fit a model with just an intercept. This is what `rat2.lr0` is. The notation `1` on the right side of the model formula means "the intercept". Since there is nothing else here, the intercept has to be specified explicitly. Then we compare the fit of the two models using `anova`, just as we did in regression. Here we're asking "does a model which has the survival probability depending on dose fit better than one in which the survival probability is the same, regardless of dose?". In the context of `glm`, `anova` won't do you a test unless you tell it which one; `Chisq` is a good choice. You see that the P-value for the significance of "everything", ie. dose, is very small, but it's *not* identical with the test for the slope of `dose`. Usually, however, the conclusions are similar (and if they're not, there's usually a reason!).

Lastly, the fitted probabilities. These go down from 0.91 at dose 0 to 0.09 at dose 5.

Suppose we now want to use our model to predict survival probabilities at some different doses, say the ones halfway between the doses for which our observations were made. We do this in two steps: first, we make a data frame containing the new doses to predict for (the first two lines in Figure 10.8), and then we call on `predict` to do the actual predictions. Something to keep in mind here is that unless you say otherwise, you'll get predicted *log-odds* instead of predicted probabilities. Hence the `type="response"`.

You see that the predicted survival probability at dose 2.5 is a half. Dose 2.5 (in this case) is sometimes called the "median lethal dose" (even when you're not actually killing things): the prediction is that half the rats getting this dose will survive and half will not.

Sometimes you want to give a sense of how accurate your predictions are. For

```
> rat2.lr=glm(cbind(lived,died)~dose,family="binomial")
> summary(rat2.lr)

Call:
glm(formula = cbind(lived, died) ~ dose, family = "binomial")

Deviance Residuals:
      1        2        3        4        5        6
 1.3421  -0.7916  -0.1034   0.1034   0.0389   0.1529

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   2.3619     0.6719   3.515 0.000439 ***
dose         -0.9448     0.2351  -4.018 5.87e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 27.530  on 5  degrees of freedom
Residual deviance:  2.474  on 4  degrees of freedom
AIC: 18.94

Number of Fisher Scoring iterations: 4

> rat2.lr0=glm(cbind(lived,died)~1,family="binomial")
> anova(rat2.lr0,rat2.lr,test="Chisq")

Analysis of Deviance Table

Model 1: cbind(lived, died) ~ 1
Model 2: cbind(lived, died) ~ dose
  Resid. Df Resid. Dev Df Deviance  Pr(>Chi)
1         5     27.530
2         4      2.474  1   25.056 5.568e-07 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> cbind(dose,rat2.lr$fitted)

  dose
1    0 0.9138762
2    1 0.8048905
3    2 0.6159474
4    3 0.3840526
5    4 0.1951095
6    5 0.0861238
```

Figure 10.7: Analysis for Example 2

```
> newdose=seq(0.5,4.5,1)
> rat2.new=data.frame(dose=newdose)
> p=predict(rat2.lr,rat2.new,type="response")
> cbind(rat2.new,p)

  dose         p
1  0.5 0.8687016
2  1.5 0.7200609
3  2.5 0.5000000
4  3.5 0.2799391
5  4.5 0.1312984
```

Figure 10.8: Predictions for different doses

example, you might want to make a plot. This requires you to add the `se.fit` argument to `predict`, returning you a list with two components, `fit` and `se.fit`. With a reasonable amount of data, going up and down from the fit by twice the standard error will make you a confidence interval for each prediction. Then you can plot the results. Figure 10.9 shows you how.

First, we calculate the lower and upper limits of the confidence intervals, by going down and up by twice the standard errors. Then we plot the fitted probabilities against the dose, using points and lines. I explicitly made the y-axis go from 0 to 1, because otherwise our confidence limits might go off the bottom or top of the plot. Then we add lines for the confidence limits, upper and then lower. I'm joining the values by lines (hence `type="l"`) and making the lines dashed (hence `lty`).

10.5 Multiple logistic regression

Multiple logistic regression is to logistic regression as multiple regression is to regression: you still have one (category of) response, but you have more than one explanatory variable.

Let's proceed to an example. Our data set consists of 107 patients with blood poisoning severe enough to warrant surgery. We want to relate survival to other variables (usually called "risk factors" in this kind of work). Most of these are labelled 1 for "present" and 0 for "absent":

- death from sepsis (response)
- shock
- malnutrition

```
> p2=predict(rat2.lr,rat2.new,type="response",se.fit=T)
> lcl=p2$fit-2*p2$se.fit
> ucl=p2$fit+2*p2$se.fit
> plot(newdose,p2$fit,ylim=c(0,1),type="b")
> lines(newdose,ucl,type="l",lty="dashed")
> lines(newdose,lcl,type="l",lty="dashed")
> detach(rat2)
```

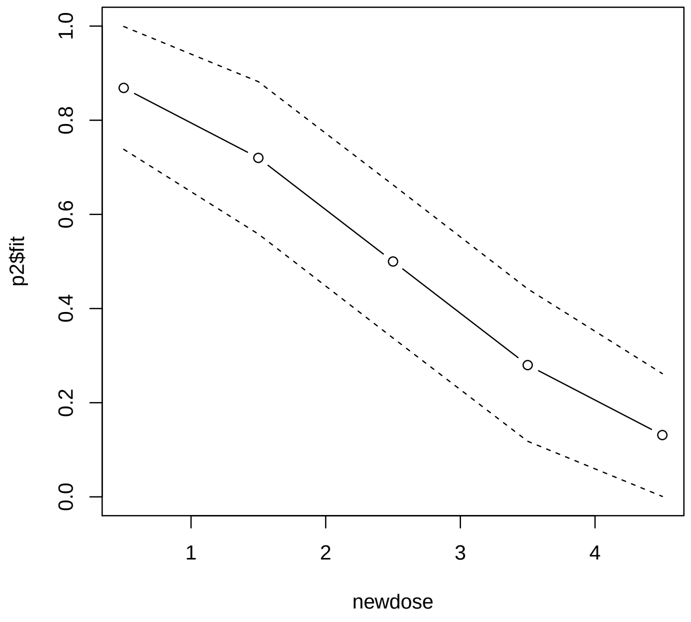


Figure 10.9: Confidence intervals for predictions

- alcoholism
- age (numerical, years)
- bowel infarction

With multiple explanatory variables, it isn't easy to draw pictures, so we'll leap into an analysis and draw some pictures afterwards.

To start with, I read in the data (one patient per line) and then listed some of the rows (the rows that would have been produced by `head` weren't very interesting). Patients 7, 9, and 12 had none of the risk factors and survived. Patient 10 was old and had malnutrition but survived, and the two patients here that died both had alcoholism (patients 8 and 11). All the variables except `age` are really factors, but treating them as numerically 1 and 0 is fine (and actually makes interpretation easier). For the response, R treats the first "category", 0, of `death` as a failure and 1 and a success, so we will be modelling the probability of death.

As with multiple regression, we can start by including everything in the model and taking out what doesn't seem to have anything to add.

The output from `glm` indicates that everything is significant except for `malnut` (and even that is close). Let's try taking `malnut` out and confirming that the fit is not significantly worse.

This is model `sepsis.lr2`, shown in Figure 10.11. The P-value from `anova` is indeed very similar to the summary table from `sepsis.lr` (Figure 10.10), and confirms at the 0.05 level that `malnut` has nothing to add.

So model `sepsis.lr2` is the one we'd like to use. Let's eyeball the summary table. All of the coefficients are positive, which means that having any of the risk factors *increases* the probability of death from sepsis (which is rather what you'd expect).

In the spirit of Section 10.2, we can interpret the slopes by taking e-to-them and saying that they have this multiplicative effect on the odds. This is shown in Figure 10.12, done so that we can just read off the answers. The numbers are quite scary: being in shock increases the odds of death from sepsis by 40 times, being an alcoholic increases them by 24 times, having a bowel infarction increases the odds by 11 times, and being one year older increases the odds by 1.1 times. This last may not seem like much, but it compounds: being 40 years older increases the odds by $1.1^{40} = 35$ times.

When an explanatory variable is 0 or 1 like this, the slope coefficient measures how much effect the presence (as opposed to the absence) of the risk factor on the log-odds of the probability of death. There is really nothing to check here assumptions-wise. But the numerical variable `age` is a different story: the

```
> sepsis=read.table("sepsis.txt",header=T)
> sepsis[7:12,]

   death shock malnut alcohol age bowelinf
7      0     0      0       0  21        0
8      1     0      0       1  69        0
9      0     0      0       0  57        0
10     0     0      1       0  76        0
11     1     0      0       1  66        1
12     0     0      0       0  48        0

> sepsis.lr=glm(death~shock+malnut+alcohol+age+bowelinf,data=sepsis,family=binomial)
> summary(sepsis.lr)

Call:
glm(formula = death ~ shock + malnut + alcohol + age + bowelinf,
    family = binomial, data = sepsis)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.3277  -0.4204  -0.0781  -0.0274   3.2946

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -9.75391    2.54170  -3.838 0.000124 ***
shock        3.67387    1.16481   3.154 0.001610 **
malnut       1.21658    0.72822   1.671 0.094798 .
alcohol      3.35488    0.98210   3.416 0.000635 ***
age          0.09215    0.03032   3.039 0.002374 **
bowelinf     2.79759    1.16397   2.403 0.016240 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 105.528  on 105  degrees of freedom
Residual deviance:  53.122  on 100  degrees of freedom
AIC: 65.122

Number of Fisher Scoring iterations: 7
```

Figure 10.10: Analysis of sepsis data

```
> sepsis.lr2=glm(death~shock+alcohol+age+bowelinf,data=sepsis,family=binomial)
> anova(sepsis.lr2,sepsis.lr,test="Chisq")

Analysis of Deviance Table

Model 1: death ~ shock + alcohol + age + bowelinf
Model 2: death ~ shock + malnut + alcohol + age + bowelinf
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1       101     56.073
2       100     53.122  1   2.9504  0.08585 .
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> summary(sepsis.lr2)

Call:
glm(formula = death ~ shock + alcohol + age + bowelinf, family = binomial,
    data = sepsis)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-1.26192  -0.50391  -0.10690  -0.04112   3.06000

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -8.89459    2.31689  -3.839 0.000124 ***
shock        3.70119    1.10353   3.354 0.000797 ***
alcohol      3.18590    0.91725   3.473 0.000514 ***
age          0.08983    0.02922   3.075 0.002106 **
bowelinf     2.38647    1.07227   2.226 0.026039 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 105.528  on 105  degrees of freedom
Residual deviance:  56.073  on 101  degrees of freedom
AIC: 66.073

Number of Fisher Scoring iterations: 7
```

Figure 10.11: Analysis of sepsis data, part 2

```
> exp(coef(sepsis.lr2))

 (Intercept)        shock      alcohol          age     bowelinf
1.371288e-04 4.049560e+01 2.418914e+01 1.093990e+00 1.087502e+01

> 1.093^40

[1] 35.05956
```

Figure 10.12: Assessing effect of variables on odds

```
> plot(sepsis.lr2)
```

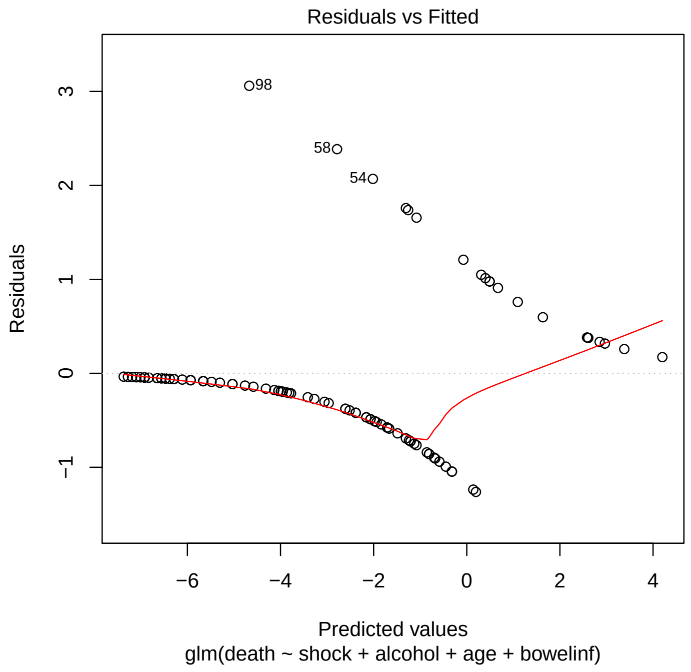


Figure 10.13: Multiple logistic regression: residuals vs. fitted values

```
> plot(sepsis$age,residuals(sepsis.lr2))
```

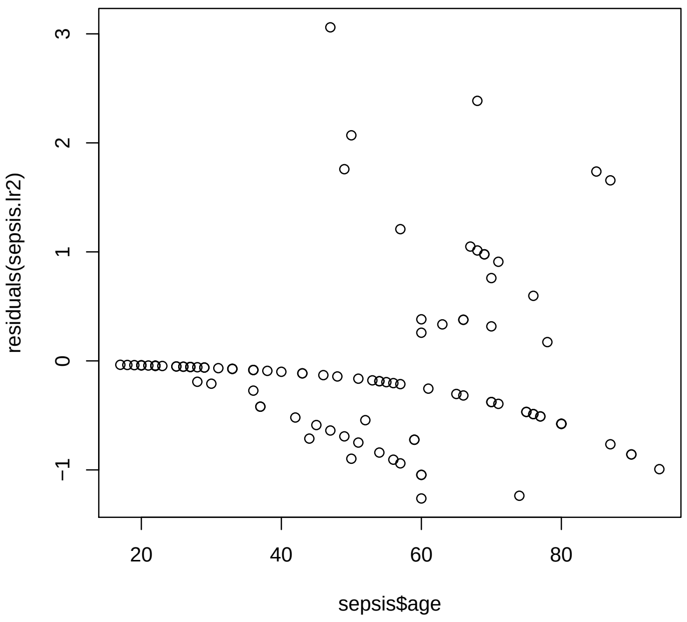


Figure 10.14: Multiple regression: residuals against age

```
> sepsis[98,]

   death shock malnut alcohol age bowelinf
98     1     0      0       0  47        0

> fitted.values(sepsis.lr2)[98]

         98
0.009262298

> residuals(sepsis.lr2)[98]

      98
3.060001
```

Figure 10.15: Patient 98

assumption is one of "proportional odds": that is, that each year increase in age is associated with the same change in log-odds, no matter what increase you start from. This is something that ought to be checked.

The garden-variety regression plot is of residuals against fitted values, looking for any overall problems with the model. Our plot is shown in Figure 10.13. Just plotting the fitted model object gets this. (These are fitted values on the log-odds scale.) With logistic regression, you have two "strands" of residuals, positive ones corresponding to deaths (successes) and negative ones corresponding to survivals (failures). This makes it hard to check the plot for patternlessness. (The residuals close to 0 on the left are for actual survivors who had a large estimated chance of surviving; the ones close to 0 on the right are for actual deaths who had a large estimated chance of dying. The interesting points are the ones at the opposite ends of the strands: for example, patient 98 died, but was predicted to have a very small chance of doing so.)

Now, to assess whether the assumed proportional-odds relationship between age and survival was good enough, we should look at a plot of the residuals against age. The plot looks a bit odd because of the line of points through the middle of it. These are the patients who had no risk factors and survived. There are quite a few of these; 63, in fact, out of 107. The downward curve of the "line" is showing the age effect; as patients get older, their chance of dying increases even if they have no risk factors, so if they survive, their residuals start at 0 and become more negative. Anyway, the acid test of this plot is the question "if you know age, but nothing else, can you predict the residual?". I don't think you can, so I think the proportional-odds assumption for `age` is good.

Patient 98 rather stands out, so we ought to take a look. See Figure 10.15. This is a patient of age 47, with no risk factors, who nonetheless died. The estimated

```
 Exposure None Moderate Severe
    5.8    98       0        0
   15.0    51       2        1
   21.5    34       6        3
   27.5    35       5        8
   33.5    32      10        9
   39.5    23       7        8
   46.0    12       6       10
   51.5     4       2        5
```

Figure 10.16: Miners' lung disease data

probability of death is just under 1%. The residuals in this case are not just observed minus expected, but something more complicated. Even so, 3.06 is large for a residual here, but perhaps not *so* large, given that we have over 100 patients.

10.6 Multiple response categories: ordered response

What if we have more than two response categories? Well, what we do depends on whether the categories have a natural *order* to them, like severity of a disease, or grade in a course, or whether they don't (preferred political party, favourite colour).

Let's tackle the ordered response case first. Coal miners are susceptible to lung disease, in particular pneumoconiosis. It is believed that exposure to coal dust is what causes the disease, so that miners who are exposed longer are more likely to have a more severe case of the disease. The data are shown in Figure 10.16. These are frequencies, the number of miners in each exposure group that were suffering disease of the severity shown. Do the data support the hypothesis that longer exposure is associated with more severe disease?

A cursory look at the data suggests that the data *do* support the hypothesis. None of the miners exposed for 5.8 years had any disease, whereas there are a notable number of Severe cases for the miners exposed longer than, say, 30 years.

I copied the data as you see in Figure 10.16 into a file (actually, I scanned it and OCRed it). So reading it in the obvious way will work, as shown in Figure 10.17.

```
> miners=read.table("miners-tab.txt",header=T)
> miners

  Exposure None Moderate Severe
1      5.8   98        0      0
2     15.0   51        2      1
3     21.5   34        6      3
4     27.5   35        5      8
5     33.5   32       10      9
6     39.5   23        7      8
7     46.0   12        6     10
8     51.5    4        2      5
```

Figure 10.17: Reading in the miners data

Just so you know that I get things wrong too: I scanned these data in from a book and used OCR to get the values. I wondered why `read.table` was reading them in as text, even after I tried to persuade it that they were integers. After some time and head-scratching, I realized that the two 0's had been OCRd as letter O's rather than number 0's. A "D'oh" moment.

By way of plots, I would like to plot the proportions of None, Moderate and Severe for each exposure group, against exposure, so that we can see any trends. This turns out to be a bit fiddly to accomplish, so we'll do it in two steps. The first part is in Figure 10.18. First we pull out the `exposures` (the first column of `miners`) and the observed frequencies (everything except the first column). Then we pull off the names of the `severity` categories and save them for later. Now comes the actual calculation. We want to find the row totals of our matrix of frequencies, and divide each row by them. "Row total" or "column total" is a job for `apply`. Feed it the matrix for which you want something calculated, a 1 for rows (or a 2 for columns), and what you want calculated (the `sum` of each row). You see that we have 8 totals, one for each row (and you can quickly check them to make sure they're about right). Now, here's where R is smart. If you divide the matrix of frequencies by the row totals, R will figure out (because you have 8 things to divide by and there are 8 rows) that you want to divide each *row* by the corresponding row total[1].

Then we want to plot exposure against category proportion for each severity category. My strategy for this is to draw an empty plot (`type="n"`) with axes set up properly (I make sure the vertical scale goes from 0 to 1). Then I use `lines` (or I suppose `points` would also work) to draw the lines for each severity group. I use `type="b"` to draw points connected by lines, and I use different

[1]R isn't actually mind-reading here. It has rules for figuring out what to do when you divide a matrix by a vector. If there are fewer things in one, here 24 in the matrix and 8 in the vector of totals, it re-uses the smaller one, in this case working down the columns.

```
> exposures=miners[,1]
> freqs=miners[,-1]
> sev=colnames(freqs)
> totals=apply(freqs,1,sum)
> totals

[1] 98 54 43 48 51 38 28 11

> obsprop=freqs/totals
> obsprop

       None   Moderate     Severe
1 1.0000000 0.00000000 0.00000000
2 0.9444444 0.03703704 0.01851852
3 0.7906977 0.13953488 0.06976744
4 0.7291667 0.10416667 0.16666667
5 0.6274510 0.19607843 0.17647059
6 0.6052632 0.18421053 0.21052632
7 0.4285714 0.21428571 0.35714286
8 0.3636364 0.18181818 0.45454545
```

Figure 10.18: Calculating observed proportions

plotting characters (`pch`) and colours (`col`) to distinguish the lines. Finally, the plot was crying out for a legend. I had always been frightened by R's `legend` command, but there's not much to it. First thing you feed in is a location or position for the legend to go (one of the corners of the plot usually works), then the thing making the different lines on the plot that you want to label, here the levels of `severity`, then the ways in which those different lines are labelled on the plot, which in our case was with the colours 1 through 3 and plotting characters 1 throught 3. If you just used colours, you'd leave out `pch` in the legend.

Let's see if we can see what the plot is telling us. As exposure increases, the proportion of "none" is dropping, and as exposure goes beyond 40 years, the proportion of "severe" increases dramatically. The proportion of "moderate" increases slowly up to an exposure of about 30, and then levels off; it doesn't really have much to say once you've looked at "none" and "severe".

The contingency table representation works nicely for drawing the plot, but it doesn't work for fitting models. The reason is that `miners` is in "wide" format, with several observations (frequencies) on one line, but for modelling we need one observation per line, so-called "long" format. This comes up often enough that R has a function called `reshape` for handling it. I describe this function in greater detail at the end of Section 15.2.

```
> plot(exposures,obsprop[,1],type="n",ylim=c(0,1))
> lines(exposures,obsprop[,1],type="b",col=1,pch=1)
> lines(exposures,obsprop[,2],type="b",col=2,pch=2)
> lines(exposures,obsprop[,3],type="b",col=3,pch=3)
> legend("topright",sev,col=1:3,pch=1:3,title="Severity")
```

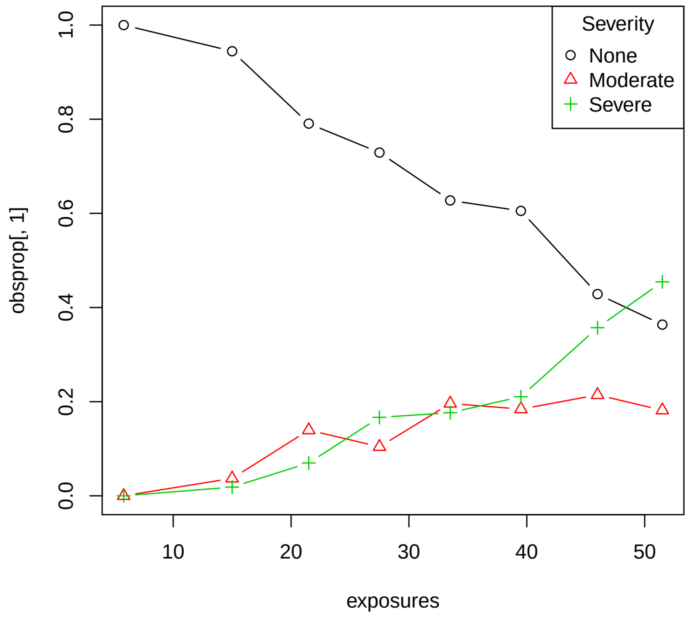


Figure 10.19: Plotting category proportions against exposure

```
> miners.long=reshape(miners,varying=c("None","Moderate","Severe"),
+   v.names="frequency",timevar="severity",direction="long")
> miners.long

    Exposure severity frequency id
1.1      5.8        1        98  1
2.1     15.0        1        51  2
3.1     21.5        1        34  3
4.1     27.5        1        35  4
5.1     33.5        1        32  5
6.1     39.5        1        23  6
7.1     46.0        1        12  7
8.1     51.5        1         4  8
1.2      5.8        2         0  1
2.2     15.0        2         2  2
3.2     21.5        2         6  3
4.2     27.5        2         5  4
5.2     33.5        2        10  5
6.2     39.5        2         7  6
7.2     46.0        2         6  7
8.2     51.5        2         2  8
1.3      5.8        3         0  1
2.3     15.0        3         1  2
3.3     21.5        3         3  3
4.3     27.5        3         8  4
5.3     33.5        3         9  5
6.3     39.5        3         8  6
7.3     46.0        3        10  7
8.3     51.5        3         5  8
```

Figure 10.20: Converting between wide and long format

The process here is shown in Figure 10.20. The `reshape` command looks a bit complicated, but that's because we have quite a bit of information to convey. First, we need to say which data frame we're talking about. Then we need to say which columns of the "wide" data are going to be combined together into one. These are the three columns of frequencies, and we feed them in as `varying`. Those three columns are all instances of frequencies, so we'll call the combined column `frequency`. This goes in as `v.names`. Then, the long-format data set is going to keep track of which row and column each original data value (frequency) came from. The rows can be figured out by `reshape` (it's the column(s) that were *not* mentioned in `varying`), but the columns need to be named. They are all instances of `severity`, so that gets fed in as `timevar`[2]. Finally, we have a look at our new data frame `miners.long`. You might like to figure out where everything went. (We don't need `id` here, since we kept track of `Exposure`.)

My analysis of the miners data is shown in Figure 10.21. We're using a function called `polr`, which lives in the MASS package. This works like `lm`, but it does "Proportional Odds Logistic Regression", hence the name. This is logistic regression with multiple ordered response categories. The syntax is much like `lm` (or `glm`), but the thing on the left of the model formula, here `severity.fac`, has to be an *ordered factor*. On the right is our one explanatory variable, `exposure`. The first line of code is creating the ordered factor to be the response. The `severity` column of `miners.long` is just numbers 1, 2, 3 for "none", "moderate", "severe". The important thing to remember here is that if you have consolidated data (with explanatory-variable combinations and frequencies), you need to specify the frequency variable in `weights`. Otherwise, R will assume that all your frequencies are 1, which is probably not what you want!

I have used the `data=` option in `polr` (the same as in `lm` and `glm`) to specify a data frame for R to find the variables in, if it can't find them otherwise. This saves `attach`ing the data frame, and then having to remember to `detach` later.

The `summary` of the fitted model object is actually not very helpful. About the only thing of value are the coefficients for the explanatory variables, and they don't even have P-values. There are a couple of ways of getting P-values for them (here just `exposure`). One is to treat the t-value on the end of the line as if it were normal, and say that anything larger in size than about 2 is significant (the two-sided P-value for 2 is 0.05). Here, the t-value is 8, so this is strongly significant. Or you can fit another model without the thing you're testing, and use `anova` with `test="Chisq"` to test the difference. Here, model `miners.lr0` has just an intercept, and running `anova` on the two models gives a P-value that is basically 0. There *is* an effect of exposure, without a doubt. Which is what our plot was showing, but this puts a P-value to it.

To understand the nature of the effect, we can use `predict`. As before, using

[2]The reason for the odd name `timevar` becomes clearer at the very end of Section **??**.

```
> library(MASS)
> severity.fac=ordered(miners.long$severity)
> miners.lr=polr(severity.fac~Exposure,weights=frequency,data=miners.long)
> summary(miners.lr)

Call:
polr(formula = severity.fac ~ Exposure, data = miners.long, weights = frequency)

Coefficients:
          Value Std. Error t value
Exposure 0.0959    0.01194   8.034

Intercepts:
    Value   Std. Error t value
1|2  3.9558  0.4097     9.6558
2|3  4.8690  0.4411    11.0383

Residual Deviance: 416.9188
AIC: 422.9188

> miners.lr0=polr(severity.fac~1,weights=frequency,data=miners.long)
> anova(miners.lr0,miners.lr,test="Chisq")

Likelihood ratio tests of ordinal regression models

Response: severity.fac
     Model Resid. df Resid. Dev   Test    Df LR stat. Pr(Chi)
1        1       369   505.1621
2 Exposure       368   416.9188 1 vs 2     1 88.24324       0
```

Figure 10.21: Analysis of miners data

```
> exp=data.frame(Exposure=exposures)
> p3=predict(miners.lr,exp,type="probs")
> preds=cbind(exp,p3)
> preds

  Exposure         1          2          3
1      5.8 0.9676920 0.01908912 0.01321885
2     15.0 0.9253445 0.04329931 0.03135614
3     21.5 0.8692003 0.07385858 0.05694115
4     27.5 0.7889290 0.11413004 0.09694093
5     33.5 0.6776641 0.16207145 0.16026444
6     39.5 0.5418105 0.20484198 0.25334756
7     46.0 0.3879962 0.22441555 0.38758828
8     51.5 0.2722543 0.21025011 0.51749563
```

Figure 10.22: Predictions for exposures in given data set

`predict` on the original fitted model object will produce predictions for each exposure in the data set, or you can make a data frame of new exposures and feed that into `predict` as the second thing. Since we have a list of all the exposures in `exposures`, that's not hard to arrange (first line). Then we feed that into `predict` along with the fitted model object `miners.lr`. If you don't specify the `type` argument, you get predicted response categories, which is likely not what you want.

This all shows how the predicted probability of no disease drops off as exposure increases, and the probability of severe disease increases dramatically at the end.

As a final flourish, it would be nice to plot these fitted probabilities on the graph that we began with. This time, let's plot the observed proportions as points, and the predicted probabilities as lines. My plot is shown in Figure 10.23.

This seems like a lot of work (*eight* lines for one plot). But there's nothing much new here. The `plot` line sets up the graph but doesn't plot anything (so it doesn't matter much what I plot, as long as it contains the full range of exposures). I've added a couple of things: making sure the vertical scale goes between 0 and 1, and giving the axes proper labels. I wasn't sure whether to use "Probability" or "Proportion" on the vertical axis. Next, we plot the data points, using `points` but otherwise borrowing from Figure **??** (and also taking away `type="both"` since we only want the points). Then we plot the three lines, one at a time, using `preds` that I created in Figure 10.22 (this is why I saved it in a variable). Finally, we add a legend, stolen from Figure **??**.

The plot shows how the predicted probabilities depend on exposure. They appear to track the data pretty well. There don't appear to be any exposure-

```
> plot(preds[,1],preds[,2],type="n",ylim=c(0,1),xlab="Exposure",ylab="Probability")
> points(exposures,obsprop[,1],col=1,pch=1)
> points(exposures,obsprop[,2],col=2,pch=2)
> points(exposures,obsprop[,3],col=3,pch=3)
> lines(preds[,1],preds[,2],col=1,pch=1)
> lines(preds[,1],preds[,3],col=2,pch=2)
> lines(preds[,1],preds[,4],col=3,pch=3)
> legend("topright",sev,col=1:3,pch=1:3)
```

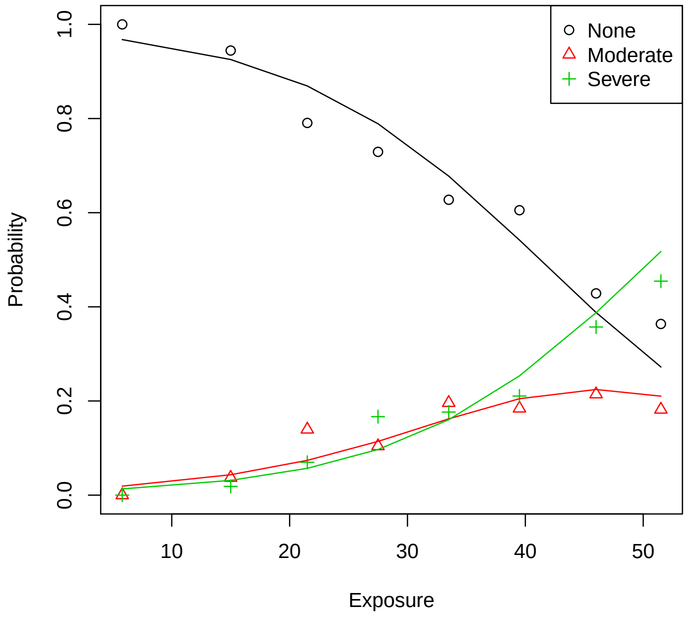


Figure 10.23: Plot of fitted probabilities and observed proportions

```
> brandpref=read.csv("mlogit.csv")
> head(brandpref)

  brand sex age
1     1   0  24
2     1   0  26
3     1   0  26
4     1   1  27
5     1   1  27
6     3   1  27

> brandpref$sex=factor(brandpref$sex)
> brandpref$brand=factor(brandpref$brand)
```

Figure 10.24: Brand preference data

severity combinations that are predicted badly. Also, from this plot, we see that once you get beyond an exposure of 45 or so, the most likely severity is "severe", having previously been "none". And the predicted probability of "moderate" does indeed take a downturn at the end, because the probability of "severe" is increasing faster than the probability of "none" is decreasing. I think this would be a great graph to put into a paper on this subject.

10.7 Multiple response categories: unordered response

When we have several response categories that are not ordered, such as favourite colours, political parties or brands of a product, we can't use the ideas of the previous section. To see what we *do* do, let's think about an example of brand preferences for a product, as they depend on gender and age. Some of the data are shown in Figure 10.24. There are 736 individuals, with one line per individual, so I'm not showing you the whole data frame! (`sex` got recorded as 1 (female) and 0 (male), so we turn that into a factor, which is really what it is. Likewise for brand preference.)

Apart from having to use a different function from a different package, there isn't much to this. The function is `multinom` from package `nnet`, but it works like `polr` does. So if you've mastered that, you'll be all right. As ever, if you get a complaint about `nnet` not existing, you'll have to install it first (through `install.packages`).

The basics are shown in Figure 10.25. You feed `multinom` a model formula, saying what is to be predicted from what, and optionally a data frame containing

```
> library(nnet)
> brands.1=multinom(brand~age+sex,data=brandpref)

# weights:  12 (6 variable)
initial  value 807.480032
iter  10 value 702.976983
final  value 702.970704
converged

> summary(brands.1)

Call:
multinom(formula = brand ~ age + sex, data = brandpref)

Coefficients:
  (Intercept)       age      sex1
2   -11.77469 0.3682075 0.5238197
3   -22.72141 0.6859087 0.4659488

Std. Errors:
  (Intercept)        age      sex1
2    1.774614 0.05500320 0.1942467
3    2.058030 0.06262657 0.2260895

Residual Deviance: 1405.941
AIC: 1417.941

> head(fitted(brands.1),n=10)

          1          2           3
1  0.9479582 0.05022928 0.001812497
2  0.8942963 0.09896238 0.006741279
3  0.8942963 0.09896238 0.006741279
4  0.7728764 0.20868979 0.018433857
5  0.7728764 0.20868979 0.018433857
6  0.7728764 0.20868979 0.018433857
7  0.8511463 0.13611419 0.012739473
8  0.8511463 0.13611419 0.012739473
9  0.7728764 0.20868979 0.018433857
10 0.8511463 0.13611419 0.012739473
```

Figure 10.25: Multinomial logistic regression for brand preferences data

```
> summary(brandpref)

 brand   sex          age
 1:207   0:269   Min.   :24.0
 2:307   1:466   1st Qu.:32.0
 3:221           Median :32.0
                 Mean   :32.9
                 3rd Qu.:34.0
                 Max.   :38.0

> brands.newdata=expand.grid(age=c(24,28,32,35,38),sex=factor(0:1))
> brands.predict=predict(brands.1,brands.newdata,type="probs")
> cbind(brands.newdata,brands.predict)

   age sex          1          2           3
1   24   0 0.94795822 0.05022928 0.001812497
2   28   0 0.79313204 0.18329690 0.023571058
3   32   0 0.40487271 0.40810321 0.187024082
4   35   0 0.13057819 0.39724053 0.472181272
5   38   0 0.02598163 0.23855071 0.735467663
6   24   1 0.91532076 0.08189042 0.002788820
7   28   1 0.69561789 0.27143910 0.032943012
8   32   1 0.29086347 0.49503135 0.214105181
9   35   1 0.08404134 0.43168592 0.484272746
10  38   1 0.01623089 0.25162197 0.732147148
```

Figure 10.26: Predictions for brand preferences data

the data. The `summary` is not hugely useful. You get intercepts and slopes on the log-odds scale. R uses the first response category as a baseline, so everything is relative to that: you get one intercept for each non-baseline category (here 2, since there are 3 categories of response) and one slope for each variable for each non-baseline category (here 4, 2 for `age` and 2 for `sex`). These are hard to interpret, so looking at the fitted probabilities makes more sense. There are two ways to do that: we can look at the fitted values from the fitted model object, as here with `fitted`, or we can do predictions. Running `predict` on the fitted model object will do that, only here we have rather too many lines to look at. So let's create a new data frame with a variety of sexes and ages, and `predict` for that. This is shown in Figure 10.26.

First, by running `summary` on the data frame, we can see what kinds of ages and sexes we have. There were not-dissimilar numbers overall preferring each brand, with brand 2 being the favourite. The ages varied from 24 to 38. Look at the values of Q1 and the median: they are both 32, which means a *lot* of the people in the data set are aged 32.

```
> brands.2=multinom(brand~age,data=brandpref)

# weights:  9 (4 variable)
initial  value 807.480032
iter  10 value 706.796323
iter  10 value 706.796322
final  value 706.796322
converged

> anova(brands.2,brands.1)

Likelihood ratio tests of Multinomial Models

Response: brand
      Model Resid. df Resid. Dev   Test    Df LR stat.    Pr(Chi)
1       age      1466   1413.593
2 age + sex      1464   1405.941 1 vs 2     2 7.651236 0.02180495
```

Figure 10.27: Comparing models with and without sex

So let's do some predictions for ages between 24 and 38 for both sexes. First, we make a new data frame to predict from, using `expand.grid` to do all the combinations. Then: if you just run `predict` on the fitted model object and the data frame of new data, you'll just get a predicted response category. This is just the same as `polr` for ordered response categories. So adding `type="probs"` (or just `type="p"`) will get predicted probabilities.

Looking at the predicted probabilities in Figure 10.26, younger males prefer brand 1, but as they get older, they prefer brand 1 less and brand 3 more. For females (the bottom five rows with `sex=1`), the pattern is similar. Which might make you wonder: is `sex` significant? Well, try fitting a model without `sex` and see whether the fit is significantly worse. This is shown in Figure 10.27.

The strategy is the same as we have seen before: fit a model without the variable you are testing, and use `anova` to compare the model fits, smaller model first. The fit doesn't look much worse without `sex`, but because we have so much data, it *is* actually significant. So we ought to keep `sex` in the model. (There is an argument of "practical importance" here: is the difference between sexes worth worrying about from a marketing point of view?). There is clearly an effect of *age*, so I'm not even testing that, but you could do it the same way.

Now, we assumed that the relationship between `age` and brand preference was linear in log-odds. Now, you may not care for that terminology, but that results in the predicted probabilities being constrained in terms of the way they can behave. So we should look at a plot of the residuals against age. Except that there are three columns of residuals, one for each response category. So we

```
> plot(brandpref$age,resid(brands.1)[,1],type="n",xlab="age",ylab="residual")
> points(brandpref$age,resid(brands.1)[,1])
> points(brandpref$age,resid(brands.1)[,2])
> points(brandpref$age,resid(brands.1)[,3])
```

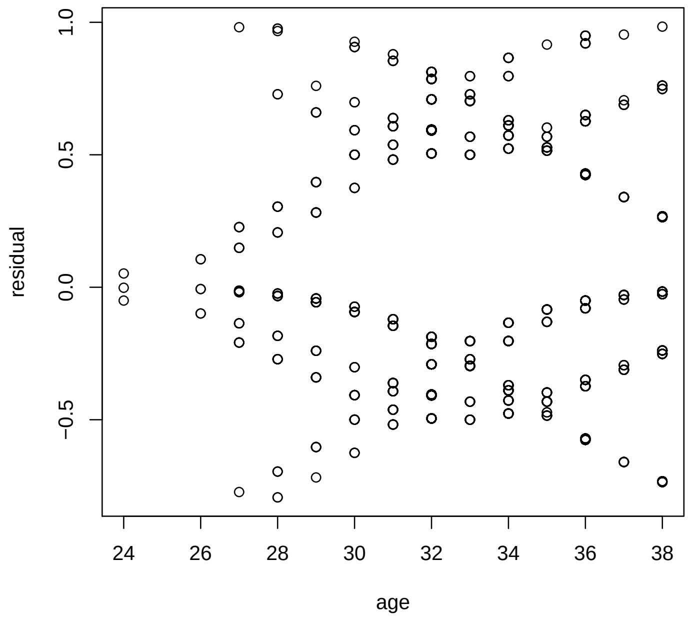


Figure 10.28: Residual plots for brand data

can plot these against age, one at a time. In Figure 10.28, I used my usual strategy of creating an empty plot first, taking the opportunity to create some meaningful axis labels (with `xlab` and `ylab`), and then plotting the three sets of residuals one at a time with `points` (or I could have used a loop, but three columns was few enough for me to copy and paste). There appear to be (and are) some individual strings of residuals, corresponding to observed success and failure for each age-sex group over ages, and there should be more of the closer-to-zero residuals (resulting in the circles being darker), but it's hard to say that, given an age, you can predict what a randomly chosen residual would be. So this is good. I think.

A last hurrah on this one is a fitted probability plot. I'm going to predict for more ages, so I have a better plot. This is all shown in Figure 10.29. The prediction is just the same as in Figure 10.26, except for more different ages. `brands.predict2` again has three columns (predicted probabilities of preferring each brand), so we'll plot them using different symbols and colours.

The actual `plot` command looks a little odd this time. This is another way to get the x and y axes covering the right values. Since we're not actually going to plot anything until we get to the lines inside the loop, we can either set the axis scales using `xlim` and `ylim` in the `plot` command, or we can just "plot" the lower and upper limits of x and y.

Within the loop (I'm using a loop this time), we plot the actual predicted probabilities, using the text in `i` (to plot the actual brand numbers 1, 2 and 3). I thought about using the plotting characters (ie. `pch`), but figured that plotting the actual brand numbers preferred would be better. Hence the use of `text` rather than `points`. Remember that `text` needs (at least) three things: the two variables giving the locations at which some text is going to be drawn, the variable containing the text that is going to be drawn there, and then any optional things like colours. I had a little trouble with colours this time, so I created a vector of colours. The syntax of `ifelse` is that if the logical first thing is true (`sex==1` means female), `mycol` should be red, otherwise `mycol` should be blue. Then I used that vector of colours to tell `text` what colour to make the numbers on the plot. (I had thought about using pink and blue, but the pink didn't show up very well.)

The conclusion is that males are slightly more in favour of brand 1 than females all the way along, females are slightly more in favour of brand 2 all the way along, and males and females have basically identical preferences for brand 3 for any age. In case you were wondering from Figure 10.26, there is a small range of ages where people of both sexes are predicted to favour brand 2, ages 32–34 or so. (The age range is slightly wider for females, because they have a generally more favourable attitude towards brand 2.)

The actual data layout is a bit wasteful, with one line per person, because there are only so many combinations of age, sex, and brand-preferred.

```
> brands.newdata=expand.grid(age=c(24:38),sex=factor(0:1))
> brands.predict2=predict(brands.1,brands.newdata,type="probs")
> plot(c(24,38),c(0,1),type="n",xlab="age",ylab="predicted probability")
> mycol=ifelse(brands.newdata$sex==1,"red","blue")
> for (i in 1:3)
+   {
+     text(brands.newdata$age,brands.predict2[,i],i,col=mycol)
+   }
> legend("topright",legend=levels(brands.newdata$sex),fill=c("blue","red"))
```

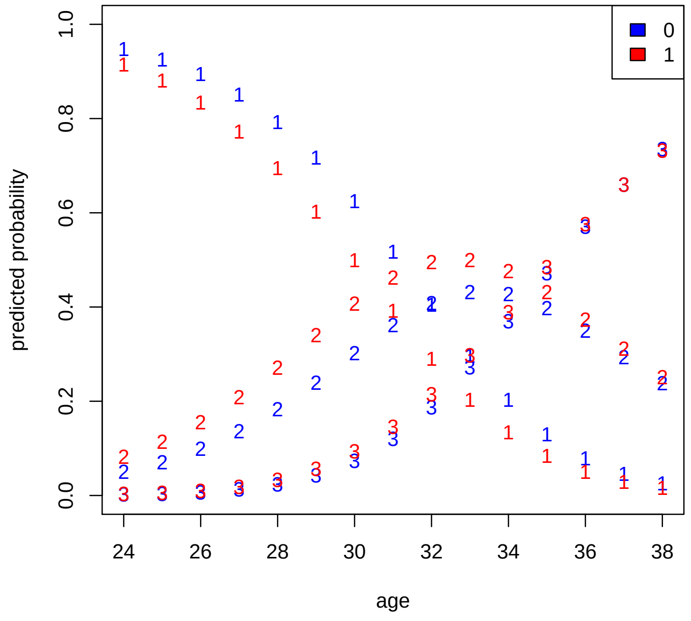


Figure 10.29: Fitted probability plot for brand preference data

```
> attach(brandpref)
> brand.agg=aggregate(brandpref,list(sex,age,brand),length)
> detach(brandpref)
> dim(brand.agg)

[1] 65  6

> dim(brandpref)

[1] 735   3

> brand.agg[c(10,20,30,40,50,60),]

   Group.1 Group.2 Group.3 brand sex age
10       1      30       1    10  10  10
20       0      36       1     4   4   4
30       1      31       2     9   9   9
40       1      36       2    19  19  19
50       0      31       3     2   2   2
60       0      36       3    16  16  16

> dimnames(brand.agg)[[2]]=c("sex","age","brand","freq","junk1","junk2")
```

Figure 10.30: Using aggregate on brand preference data

One way of collecting together data that is "repeats" is the `aggregate` command. What this does is to calculate something like a mean or median for each group, and then to glue everything back into a data frame, rather than leaving the results as a table like `tapply` does (which would then have to be attacked via `melt` as in Section 10.6).

`aggregate` requires three things: a data frame, a factor or `list` of factors[3] to divide the data up into groups, and a function to apply to each group. We are going to be calculating frequencies, so `length` will count up the number of things in each class. The process is shown in Figure 10.30. The `dim` command gives the `dim`ensions of the aggregated data set. This has only 65 rows instead of the original 735.

The output of `aggregate` is a bit odd, so we'll have to bash it into shape. The columns with the names of the original variables are actually the frequencies, while the columns called `Group.1` etc. are the actual values of the variables. Thus, for example, if you look at row 20 of `brand.agg`, you see that for males (category 0 in `Group.1`) of age 36 (`Group.2`) and who prefer Brand 1 (`Group.3`), there are 4 of these (4 appears in the last 3 columns). Likewise, there are 9 females of age 31 who prefer brand 2. The first three columns are the "factors"

[3]I know `age` isn't in any way a factor, but here we're treating it as if it is.

```
> brand.agg.1=multinom(brand~age+sex,data=brand.agg,weights=freq)

# weights:  12 (6 variable)
initial  value 807.480032
iter  10 value 702.976983
final  value 702.970704
converged

> summary(brand.agg.1)

Call:
multinom(formula = brand ~ age + sex, data = brand.agg, weights = freq)

Coefficients:
  (Intercept)       age      sex1
2   -11.77469 0.3682075 0.5238197
3   -22.72141 0.6859087 0.4659488

Std. Errors:
  (Intercept)        age      sex1
2    1.774614 0.05500320 0.1942467
3    2.058030 0.06262657 0.2260895

Residual Deviance: 1405.941
AIC: 1417.941
```

Figure 10.31: Multinomial logistic regression with aggregated data

(including age) as we fed them into `aggregate`.

These names are no good, so we need to fix them up. `dimnames(brand.agg)` is a list whose second element is the column names, so the last line of Figure 10.30 sets them to be something more sensible.

I fitted the same multinomial logistic regression to the aggregated data in Figure 10.30. The one different thing is that we have to specify the frequencies in `weights=`, as in Figure 10.21. But the output from `summary` is exactly the same as in Figure 10.25. This is as it should be.

Chapter 11

Cluster analysis

11.1 Introduction

The idea of a cluster analysis is that you have a measure of distance or dissimilarity between each pair of a set of objects, and you want to divide the objects into groups or "clusters" so that the objects in the same cluster are similar and the ones in different clusters are dissimilar.

You can also calculate a distance from data on a number of variables and use that as input to a cluster analysis.

11.2 Hierarchical cluster analysis

Hierarchical cluster analysis starts with each object in a cluster by itself and at each stage combines the two "most similar" clusters to make a new, bigger cluster. This continues until there is only one cluster, containing all the objects. The main interest in a hierarchical cluster analysis is the clustering "story", which is illustrated graphically in a plot called a **dendrogram**. This enables you both to pick what looks like a reasonable number of clusters, and to find out which objects belong in those clusters.

Let's illustrate with an example. Figure 11.1 shows the names of the numbers from one to ten in eleven European languages[1]. What we want to do is to arrange the languages into groups of "similar" ones. How to measure similarity? We'll do something rather crude: we'll just look at the first letter of each number name,

[1]I had to use `as.is` to stop R reading in the words as factors. `as.is` keeps them as text strings.

```
> lang2=read.table("one-ten.txt",header=T,as.is=1:11)
> lang2

   English Norwegian Danish Dutch German French Spanish Italian   Polish
1      one        en     en   een   eins     un     uno     uno    jeden
2      two        to     to  twee   zwei   deux     dos     due      dwa
3    three       tre    tre  drie   drei  trois    tres     tre     trzy
4     four      fire   fire  vier   vier quatre  cuatro quattro   cztery
5     five       fem    fem  vijf   funf   cinq   cinco  cinque     piec
6      six      seks   seks   zes  sechs    six    seis     sei    szesc
7    seven       sju    syv zeven sieben   sept   siete   sette   siedem
8    eight      atte   otte  acht   acht   huit    ocho    otto    osiem
9     nine        ni     ni negen   neun   neuf   nueve    nove dziewiec
10     ten        ti     ti  tien   zehn    dix    diez   dieci dziesiec
   Hungarian   Finnish
1        egy      yksi
2      ketto     kaksi
3      harom     kolme
4       negy     nelja
5         ot     viisi
6        hat     kuusi
7        het seitseman
8      nyolc kahdeksan
9     kilenc  yhdeksan
10       tiz  kymmenen
```

Figure 11.1: One to ten in various European languages

```
> lang=read.table("languages.txt",header=T)
> lang

   en no dk nl de fr es it pl hu fi
en  0  2  2  7  6  6  6  6  7  9  9
no  2  0  1  5  4  6  6  6  7  8  9
dk  2  1  0  6  5  6  5  5  6  8  9
nl  7  5  6  0  5  9  9  9 10  8  9
de  6  4  5  5  0  7  7  7  8  9  9
fr  6  6  6  9  7  0  2  1  5 10  9
es  6  6  5  9  7  2  0  1  3 10  9
it  6  6  5  9  7  1  1  0  4 10  9
pl  7  7  6 10  8  5  3  4  0 10  9
hu  9  8  8  8  9 10 10 10 10  0  8
fi  9  9  9  9  9  9  9  8  9  8  0
```

Figure 11.2: Dissimilarities between the languages

and we'll measure the dissimilarity between a pair of languages by how many of the ten number names start with a different letter. For example, in English and Norwegian, only *one* and *en*, and *eight* and *atte*, start with different letters. The other eight number names start with the same letters. So the dissimilarity between English and Norwegian is 2. On the other hand, all ten of the number names in Polish and Hungarian start with different letters, so their dissimilarity is 10.

Figure 11.2 shows the entire table of dissimilarities. I used R to work these out. I'll show you how a bit later.

So now we have the dissimilarities among the languages. The next problem we face is that we have to combine "similar" clusters. We can measure similarity or dissimilarity among *languages*, but how to do the same for *clusters* of languages? It turns out that there is no one best way to do this, but lots of possible ways. We'll just look at three: single linkage, complete linkage and Ward's method.

Imagine, for a moment, people on Facebook. You can define a dissimilarity there to be 1 if two people are Facebook friends, else 2 if they have any friends in common, else 3 if any of their friends have friends in common, and so on. This makes a dissimilarity among *people*. But now think of each person's Facebook *friends*, which are clusters of people. How do you measure dissimilarity between *clusters*?

There is no one best way to do this, but lots of plausible ways. We'll concentrate on three:

single linkage: dissimilarity between clusters A and B is the *smallest* dissimilarity between an object in A and an object in B.

complete linkage: dissimilarity between clusters A and B is the *largest* dissimilarity between an object in A and an object in B.

Ward's method: assesses the extent to which objects within clusters are similar to each other (the "within-cluster variance"), and measures the dissimilarity between clusters by how much the within-cluster variance would increase if the clusters were combined. Ward's method is a kind of compromise between single linkage at one extreme and complete linkage at the other.

To pursue the Facebook analogy: say the single-linkage distance between my friends and yours is 0 if we have *any* friends in common, else 1 if *any* of my friends are friends with *any* of yours, and so on. The complete-linkage distance is 0 if we have *all* our friends in common, else 1 if *all* of my friends are friends with your friends (and vice versa), and so on. The distinction between single linkage and complete linkage is the same as the distinction between "any" and "all". Ward's method will describe my friends and your frie ds as worth combining if there *tends* to be a small dissimilarity between the people in my friends and the people in yours, taken as one big group of people,

Now, to actually do the clustering. Three steps: first, turn the matrix of dissimilarities into a `dist` object, which is what the clustering function expects. If you just type `d` to have a look at what `d` contains, you'll see something that looks a lot like `lang`, but if you bypass its own print method, using `print.default`, you see something of its innards. Second, do the clustering using `hclust`. This expects two things: a `dist` object, and the clustering method to be used. As usual, saving the results from `hclust` into a variable enables you to do things like drawing a plot. This is the third thing: the resulting plot is called a **dendrogram**, which is a fancy name for "tree diagram". All of this is illustrated in Figure 11.3.

The dendrogram shows how the clusters were formed. In this case, Spanish, French and Italian were joined into a cluster, and also Norwegian and Danish. Polish got joined to the former cluster, and English, German and Dutch got joined to the latter one. These two clusters were then joined together, and finally the big cluster was joined up with Hungarian and Finnish, which seem to have nothing to do with any of the other languages, or with each other.

Another way to look at the clustering process is shown in Figure 11.4. The `merge` element of the clustering results shows the clustering story. (I've also listed out the language labels to aid in following what's going on.) First objects 2 and 3 (Norwegian and Danish) are formed into a cluster. Next, objects 6 and 8 (French and Italian) are joined into a cluster. Next, object 7 (Italian) is joined onto the cluster formed at step 2 (French and Italian). Next, object 1 (English) is joined onto the cluster formed at step 1 (Norwegian and Danish). And so on.

```
> d=as.dist(lang)
> print.default(d)

 [1]  2  2  7  6  6  6  6  7  9  9  1  5  4  6  6  6  7  8  9  6  5  6  5  5  6
[26]  8  9  5  9  9  9 10  8  9  7  7  7  8  9  9  2  1  5 10  9  1  3 10  9  4
[51] 10  8 10  9  8
attr(,"Labels")
 [1] "en" "no" "dk" "nl" "de" "fr" "es" "it" "pl" "hu" "fi"
attr(,"Size")
[1] 11
attr(,"call")
as.dist.default(m = lang)
attr(,"class")
[1] "dist"
attr(,"Diag")
[1] FALSE
attr(,"Upper")
[1] FALSE

> lang.hcs=hclust(d,method="single")
> plot(lang.hcs)
```

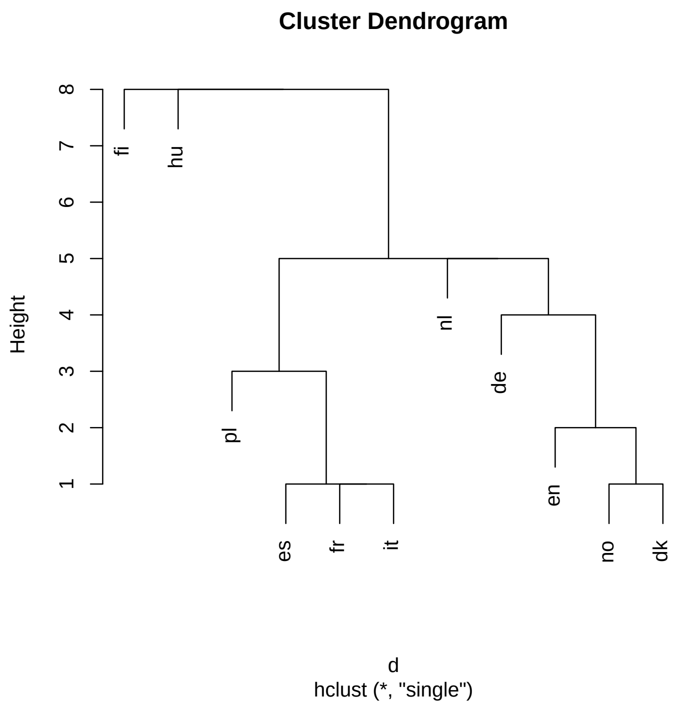


Figure 11.3: Single-linkage clustering of languages data

```
> lang.hcs$labels

 [1] "en" "no" "dk" "nl" "de" "fr" "es" "it" "pl" "hu" "fi"

> lang.hcs$merge

      [,1] [,2]
 [1,]   -2   -3
 [2,]   -6   -8
 [3,]   -7    2
 [4,]   -1    1
 [5,]   -9    3
 [6,]   -5    4
 [7,]   -4    6
 [8,]    5    7
 [9,]  -10    8
[10,]  -11    9

> cutree(lang.hcs,4)

en no dk nl de fr es it pl hu fi
 1  1  1  1  1  2  2  2  2  3  4
```

Figure 11.4: The clustering process in numbers

A negative number indicates an object (language, here), and a positive number indicates a cluster formed at an earlier step. Thus step 5 is to add Polish to the cluster containing French, Italian and Spanish.

The last line lists which cluster each language belongs to. The name `cutree` is short for "cut the tree". In this case, if you cut the tree at the point where you get 4 clusters, this tells you which languages belong to which cluster.

What is rather interesting is that these results make sense from a linguistic point of view: there is a tight cluster of Romance languages (French, Spanish, Italian), a looser cluster of Germanic languages (English, Norwegian, Danish, German, Dutch), and two languages that are loosely related to one another and not to anything else (Hungarian and Finnish, which are two distantly-related members of the Finno-Ugric language group).

Perhaps the best way to compare the clustering methods on this data set is to look at the dendrograms together. Figure 11.5 shows how that can be done.

First, we have to actually *run* the other clustering methods, saving their results. My naming convention here is `lang` for languages data, `hc` for `hclust` (or "hierarchical clustering"), and one more letter for the clustering method. Some people like `fit1, fit2, ...`. Whatever helps you keep track is fine.

Then we need to create a plotting region to accommodate all three of the plots. A 2-by-2 grid seems to work all right. Then plotting all three of the dendrograms will enable us to see them together. (You can experiment with different-sized arrays of plots. Maybe 3 rows and 1 column will also work. I'd rather see the plots enlarged *horizontally* if anything.)

The major differences seem to be:

- When (if at all) Hungarian and Finnish get joined together: not until the very end in single linkage, late in complete linkage, and (relatively) early in Ward's method.
- Dutch and German: they get clustered together early in Ward, later in complete linkage, and not at all in single linkage (they get added to the Germanic group one at a time).

Why did things come out this way? To gain some insight, let's take a detour to some data arranged along a line, and see how *those* observations cluster.

The clusterings, in Figure 11.6, are quite different. The clusterings have in common that they join points 1 and 2 (1 apart) and 6 and 7 (also 1 apart), so that we now have these two clusters, plus objects 3, 4, 5 and 8 as "clusters" by themselves. What should get joined together next? Let's make a table, shown in Table 11.1.

```
> lang.hcc=hclust(d,method="complete")
> lang.hcw=hclust(d,method="ward")
> par(mfrow=c(2,2))
> plot(lang.hcs)
> plot(lang.hcc)
> plot(lang.hcw)
```

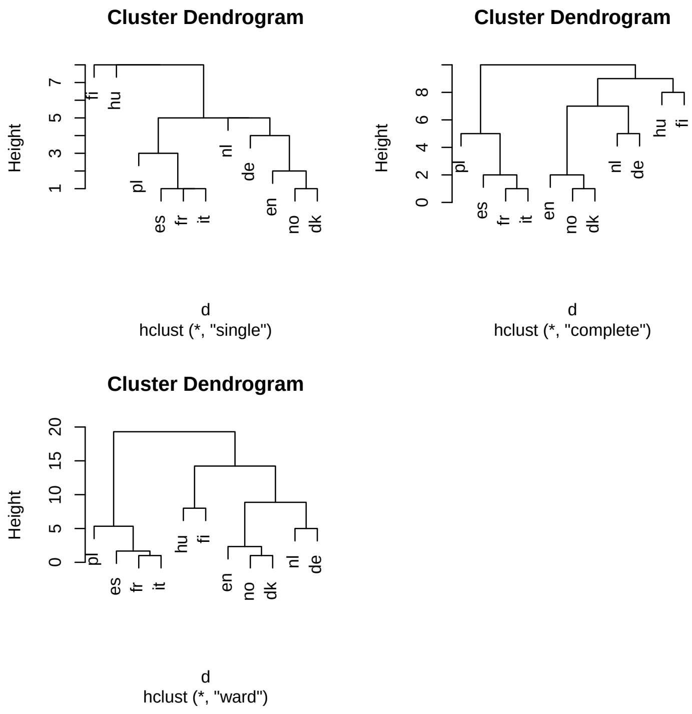


Figure 11.5: Dendrograms for all three clustering methods

```
> myvar=c(4,  5,  7,  9, 11, 13, 14, 16)
> d=dist(myvar)
> d

   1  2  3  4  5  6  7
2  1
3  3  2
4  5  4  2
5  7  6  4  2
6  9  8  6  4  2
7 10  9  7  5  3  1
8 12 11  9  7  5  3  2

> d.s=hclust(d,method="single")
> d.c=hclust(d,method="complete")
> par(mfrow=c(1,2))
> plot(d.s)
> plot(d.c)
```

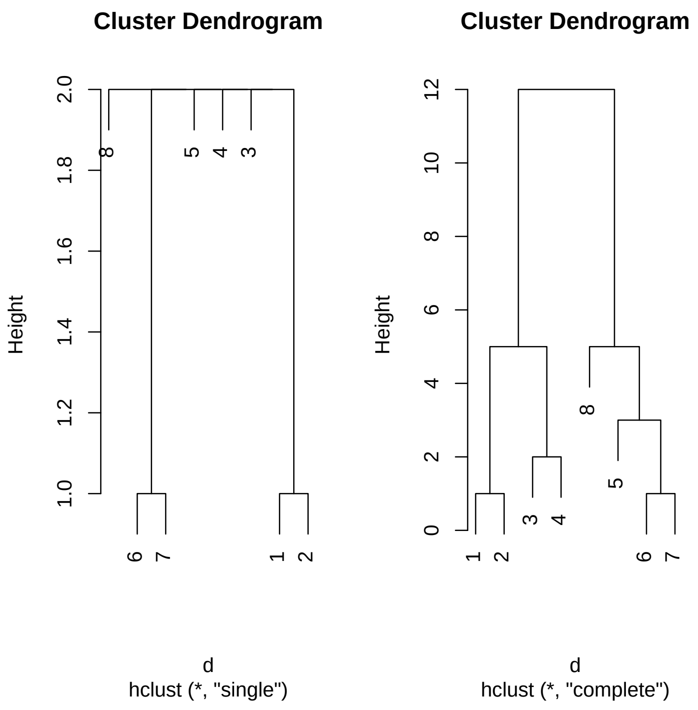


Figure 11.6: Clustering some simple data

Cluster 1	Cluster 2	Single	Complete
1,2	3	2	3
1,2	4	4	5
1,2	5	6	7
1,2	6,7	8	10
1,2	8	11	12
3	4	2	2
3	5	4	4
3	6,7	6	7
3	8	9	9
4	5	2	2
4	6,7	4	5
4	8	7	7
5	6,7	2	3
5	8	5	5
6,7	8	2	3

Table 11.1: Inter-cluster distances

Let's take complete linkage first. There are two candidates for the next cluster: objects 3 and 4, and objects 4 and 5. R chose, as we see in Figure 11.6, objects 3 and 4. You would *think* that object 5 would get immediately joined to this cluster. But the distance between object 5 and $\{3, 4\}$ is now 4 (the largest of the 3–5 and 4–5 distances), and so the smallest remaining inter-cluster distance is now 3, and R chooses to join object 5 to cluster $\{6, 7\}$. You see that the dendrogram is not unique; R could have chosen to join object 8 to cluster $\{6, 7\}$.

Now, how about single linkage? There are an awful lot of single-linkage distances that are 2 (Table 11.1), and this winds up meaning that any cluster could be joined to any other. For example, if we join 3 to $\{1, 2\}$, the distance from that cluster to object 4 is now 2 (the *smallest* of the distances between 4 and anything in that cluster). There is a kind of domino-toppling effect in that as soon as you join an object to a cluster, the distance from the new cluster to another object becomes 2, and so all the clusters end up joining together at distance 2, as shown in Figure 11.6. In general, single-linkage tends to easily combine long strung-out clusters, because one object in one cluster happens to be close to one object in another cluster.

Ward's method winds up very similar to complete linkage in this case.

Going back to our languages, there's one more thing we can do here. One of the aims of cluster analysis is to find a good number of clusters. That can be done by "chopping the tree". If you look at the dendrogram for Ward's method in Figure 11.5, you see that the languages seem to fall into three groups. Three groups looks like a good choice because there is a long vertical span (between a height of about 8 and 14) where the number of clusters stays at 3. R will draw boxes around a number of clusters you specify, which helps for describing them. The function is `rect.hclust`; it needs two things: the output from the appropriate `hclust`, and the number of clusters you want. There are other options, which the help file will tell you about, but this is the way I use it most. Figure 11.7 shows how it works. (`rect.hclust` is like `lines` in that you have to draw a plot first, and then `rect.hclust` adds the boxes to it.) I also show the output for `cutree` with three clusters, so that you can see the correspondence.

Now, how to get those dissimilarities from the number words. The first part of the procedure is shown in Figure 11.8. Since we're comparing only first letters, we don't need the other letters, so we might as well get rid of them. The `substr` function does this. The first two lines of code show how it works: you supply a character string, or a vector of character strings, and you say which character you want to start from and which you want to finish at. These first two lines are just to show you how it works. As you see, they extract the first two characters of each of those Greek letters.

Now, `lang2` is a *matrix*, with languages in the columns and the different numbers in the rows. So we treat the matrix as a bunch of columns and `apply substr` to each column. Beyond the column of the matrix that `substr` is working on, we need to pass to `substr` the place to start and the place to stop. These are passed to `apply` as shown; `apply` knows that any "extra" things passed to it get handed on to the function being applied. So this `apply` extracts just the first character from each column of number names and glues them back together into a matrix of the same form. This is `lang3`.

Next, we have to count up how many of these first letters are different for each pair of languages. As ever in R, if we can write a function to do this for one pair of languages, we can use it to produce the whole thing. `count.diff` does this. It takes a matrix of character strings, like `lang3`, and two column numbers. The logical vector `eq` is TRUE everywhere the two languages being compared start with different letters and FALSE otherwise. Since, in R, TRUE has value 1 and FALSE has value 0, summing up the things in `eq` will give you the number of TRUEs, that is, the number of first letters that are different. Then we use `count.diff` on English (column 1) and Norwegian (column 2) to find that their dissimilarity is 2. Likewise, `count.diff(lang3,9,10)` is 10, the dissimilarity between Polish and Hungarian.

Now, to calculate the dissimilarity matrix for all pairs of points, we have to use `count.diff` repeatedly. I'm sure there is a clever way of doing that, but the fact that it's all possible *pairs* makes it difficult to see how it might work. So I'm resorting to a loop, or, precisely, a pair of loops, one inside the other. Figure 11.9 shows how it goes. First set up a matrix to hold the dissimilarities. This has to be 11×11 because there are 11 languages. We'll fill it with zeroes to start with. Then we loop through all the pairs of languages, and set the `[i,j]` element of the matrix to be the dissimilarity between languages `i` and `j`, calculated using the `count.diff` function we defined earlier. The last step is to

```
> cutree(lang.hcw,3)

en no dk nl de fr es it pl hu fi
 1  1  1  1  1  2  2  2  2  3  3

> plot(lang.hcw)
> rect.hclust(lang.hcw,3)
```

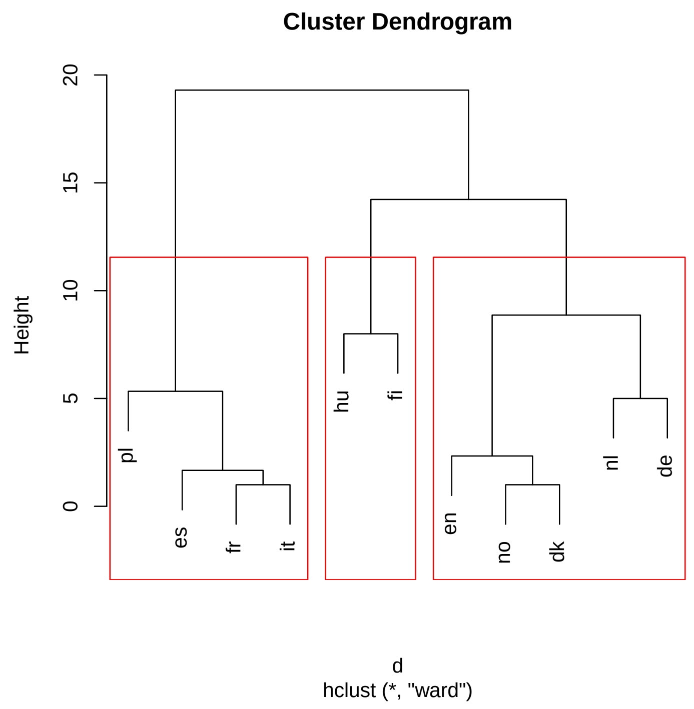


Figure 11.7: Boxes around three clusters for the Ward output

```
> v=c("alpha","beta","gamma")
> substr(v,1,2)

[1] "al" "be" "ga"

> lang3=apply(lang2,2,substr,1,1)
> lang3

      English Norwegian Danish Dutch German French Spanish Italian Polish
 [1,] "o"     "e"       "e"    "e"   "e"    "u"    "u"     "u"     "j"
 [2,] "t"     "t"       "t"    "t"   "z"    "d"    "d"     "d"     "d"
 [3,] "t"     "t"       "t"    "d"   "d"    "t"    "t"     "t"     "t"
 [4,] "f"     "f"       "f"    "v"   "v"    "q"    "c"     "q"     "c"
 [5,] "f"     "f"       "f"    "v"   "f"    "c"    "c"     "c"     "p"
 [6,] "s"     "s"       "s"    "z"   "s"    "s"    "s"     "s"     "s"
 [7,] "s"     "s"       "s"    "z"   "s"    "s"    "s"     "s"     "s"
 [8,] "e"     "a"       "o"    "a"   "a"    "h"    "o"     "o"     "o"
 [9,] "n"     "n"       "n"    "n"   "n"    "n"    "n"     "n"     "d"
[10,] "t"     "t"       "t"    "t"   "z"    "d"    "d"     "d"     "d"
      Hungarian Finnish
 [1,] "e"       "y"
 [2,] "k"       "k"
 [3,] "h"       "k"
 [4,] "n"       "n"
 [5,] "o"       "v"
 [6,] "h"       "k"
 [7,] "h"       "s"
 [8,] "n"       "k"
 [9,] "k"       "y"
[10,] "t"       "k"

> # input columns i and j, count number of different first letters
> count.diff=function(lang3,i,j)
+ {
+   eq=(lang3[,i]!=lang3[,j])
+   sum(eq)
+ }
> # get dissimilarity of English and Norwegian
> count.diff(lang3,1,2)

[1] 2

> count.diff(lang3,9,10)

[1] 10
```

Figure 11.8: Getting dissimilarities from number words, part 1

```
> d=matrix(0,11,11)
> for (i in 1:11)
+ {
+   for (j in 1:11)
+   {
+     d[i,j]=count.diff(lang3,i,j)
+   }
+ }
> d

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
 [1,]    0    2    2    7    6    6    6    6    7     9     9
 [2,]    2    0    1    5    4    6    6    6    7     8     9
 [3,]    2    1    0    6    5    6    5    5    6     8     9
 [4,]    7    5    6    0    5    9    9    9   10     8     9
 [5,]    6    4    5    5    0    7    7    7    8     9     9
 [6,]    6    6    6    9    7    0    2    1    5    10     9
 [7,]    6    6    5    9    7    2    0    1    3    10     9
 [8,]    6    6    5    9    7    1    1    0    4    10     9
 [9,]    7    7    6   10    8    5    3    4    0    10     9
[10,]    9    8    8    8    9   10   10   10   10     0     8
[11,]    9    9    9    9    9    9    9    9    9     8     0

> dd=as.dist(d)
```

Figure 11.9: Getting dissimilarities for number words, part 2

```
> vital=read.table("birthrate.dat",header=T)
> head(vital)

  birth death infant        country
1  24.7   5.7   30.8        Albania
2  13.4  11.7   11.3 Czechoslovakia
3  11.6  13.4   14.8        Hungary
4  13.6  10.7   26.9        Romania
5  17.7  10.0   23.0           USSR
6  13.4  11.6   13.0  Ukrainian_SSR

> summary(vital)

     birth           death           infant              country
 Min.   : 9.70   Min.   : 2.20   Min.   :  4.5   Afghanistan: 1
 1st Qu.:14.50   1st Qu.: 7.80   1st Qu.: 13.1   Albania    : 1
 Median :29.00   Median : 9.50   Median : 43.0   Algeria    : 1
 Mean   :29.23   Mean   :10.84   Mean   : 54.9   Angola     : 1
 3rd Qu.:42.20   3rd Qu.:12.50   3rd Qu.: 83.0   Argentina  : 1
 Max.   :52.20   Max.   :25.00   Max.   :181.6   Austria    : 1
                                                 (Other)    :91
```

Figure 11.10: Birth, death and infant mortality rates

turn `d` into a `dist` object suitable for using in `hclust`.

11.3 K-means cluster analysis

With hierarchical cluster analysis, the focus is on the clustering story: when do the various individuals get joined on to other clusters of individuals? A clustering with a given number of clusters is rather a by-product of this. Another approach is to name a number of clusters in advance, and say "give me this many clusters; I don't care about more or fewer". K-means works this way: for a given number `k` of clusters, it finds the "best" way of grouping the individuals into that many clusters, for a precise definition of "best". It works, though, for a data matrix (containing measurements on a number of variables for each individual) rather than dissimilarities.

Here's an example. I have data on birth rates, death rates and infant mortality rates for a number of countries, as shown in Figure 11.10. You can see the age of the data from the names of these countries!

Now, we have to do some massaging before we can use these data. The infant mortality rates are bigger and more variable than the other variables, but this

```
> scale(1:4)

           [,1]
[1,] -1.1618950
[2,] -0.3872983
[3,]  0.3872983
[4,]  1.1618950
attr(,"scaled:center")
[1] 2.5
attr(,"scaled:scale")
[1] 1.290994

> vital.s=scale(vital[,1:3])
> head(vital.s)

          birth       death     infant
[1,] -0.3343913 -1.10512932 -0.5240199
[2,] -1.1685431  0.18588887 -0.9480013
[3,] -1.3014168  0.55167736 -0.8719021
[4,] -1.1537793 -0.02928083 -0.6088162
[5,] -0.8511225 -0.17989961 -0.6936125
[6,] -1.1685431  0.16437190 -0.9110388
```

Figure 11.11: Standardizing

doesn't really *mean* anything; it's just because of the units the variables are measured in. So the right thing to do, since we want to treat these variables on an equal footing, is to *standardize* them first. This means subtracting the mean from each value and dividing by the standard deviation. This is the same procedure as computing z-scores, if you remember that.

Figure 11.11 shows the procedure. First, I illustrate what happens if you standardize the numbers 1 through 4: the mean is 2.5, which gets subtracted off, and the standard deviation is 1.29, and after the mean has been subtracted off, each value is divided by the standard deviation. The result is a list of numbers that has mean zero and SD 1.

Now to do this with our actual data. If you feed `scale` a matrix, it standardizes each *column*, which is exactly what we want (it usually will be, since variables are generally columns). The standardized rates are positive where the original variable was above average, and negative when below. (Note that we don't want to standardize the *countries*, so we just feed the first three columns of `vital` into `scale`.)

Now, k-means requires us to specify a number of clusters to find. If we did, we would call the function `kmeans` with two things: our matrix of standardized variables, and the number of clusters we want.

But we don't have any idea what the right number of clusters might be. What we can do instead (which seems kind of wasteful, but computing power is cheap) is to fit every possible number of clusters, and see which one we like the best. There's a plot called a *scree plot* that helps with this. The calculation and plot are shown in Figure 11.12. We are going to use a loop and collect up the quantity called the "total within-cluster sum of squares" for each number of clusters from 2 to 20. If the total within-cluster sum of squares is small, that means that the objects within each cluster are similar to each other (good), but you can always make it smaller by taking a large number of clusters (bad). The extreme case is when each object is in a cluster by itself, when the total within cluster sum of squares is *zero* (there is no variability within clusters). So we'll try to strike a balance. We'll see in a moment how to do this. Take a look at the plot in Figure 11.12. I added a grid to make reading things easier.

A scree plot (which we will also see in conjunction with principal components and factor analysis later) always goes downhill, like the side of a mountain. Eventually the "mountain" gives way to the random bits of rock, or "scree", that have fallen off the side of the mountain. Certainly the stuff beyond 10 or so on the horizontal axis is scree; this corresponds to a good number of clusters being less than 10.

What we are looking for is an "elbow" or "corner" on the scree plot. There appears to be one at 6 clusters. This happens because the drop in total within-cluster sum of squares (`wss` on the y-axis) is large from 5 to 6 (from 43 to 34), but

```
> wss=numeric(0)
> for (i in 2:20)
+   {
+     wss[i]=sum(kmeans(vital.s,i)$withinss)
+   }
> plot(2:20,wss[2:20],type="b")
> grid(col="black")
```

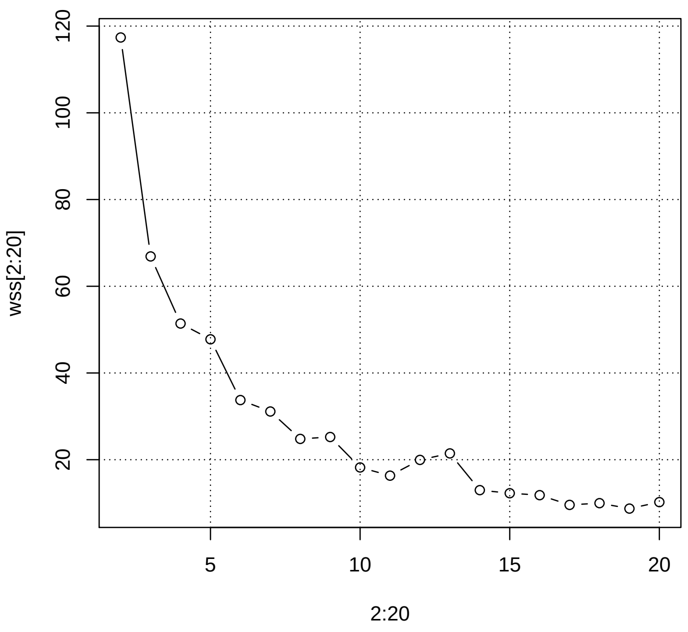


Figure 11.12: Scree plot for vital statistics data

```
> vital.km=kmeans(vital.s,5)
> vital.km$cluster

 [1] 4 4 4 4 4 4 1 4 5 5 4 2 4 4 4 4 4 4 4 4 4 3 1 4 4 5 5 1 4 5 2 5 5 4 4 5 1 5
[39] 1 1 5 5 1 3 5 1 1 1 5 4 4 4 4 4 4 5 5 5 5 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 4
[77] 1 4 5 5 1 5 4 5 3 1 3 3 1 3 3 1 3 1 5 1 1

> vital.km$centers

       birth      death     infant
1  1.2092406  0.7441347  1.0278003
2 -0.2199722  2.1116577 -0.4544435
3  1.3043848  2.1896567  1.9470306
4 -0.9718075 -0.4786352 -0.8960213
5  0.3856254 -0.5597569  0.2061981

> vital.km$withinss

[1]  7.802531  0.761109  2.882333 20.712315 10.713781

> vital.km$size

[1] 18  2  8 43 26
```

Figure 11.13: Analysis with 5 clusters

much smaller after 6 (from 34 to 32 and then gradually decreasing thereafter). This suggests that a good number of clusters is 5, the number *before* the elbow. You could also argue for an elbow at 4, and therefore three clusters would be good. This is much an art as a science.

If the plot doesn't seem to be shedding any light, you can always "zoom in" on part of it by adding something like `xlim=c(3,10)` to the `plot` command to look only at between 3 and 10, or `ylim=c(10,50)` to focus on `wss` between 10 and 50.

Now let's find 5 clusters, as shown in Figure 11.13. There is a lot of information in the fitted model object. Let's have a look at some of it:

- `cluster` is the number of the cluster that each country belongs to. This is the same as `cutree` gives, when used on output from a hierarchical clustering.

- `centers` are the values of the three variables at the centre of each cluster. Thus cluster 3 is higher than average on everything, and cluster 4 is lower than average on everything.

- `withinss` is the within-cluster sum of squares for each cluster. (The total of these values is 43.5, which is the value for `wss` plotted on the scree plot, Figure 11.12, for 5 clusters.) A cluster with a small within-cluster sum of squares is compact (all the objects in that cluster are close together), while a cluster with large `withinss` is dispersed. Cluster 2 is compact (we'll see in a moment why), and clusters 4 looks especially dispersed.

- `size` is the number of objects in each cluster. Cluster 2 had only two countries in it, which is why it was very compact. A cluster with many countries will have a harder time being compact, like cluster 4.

Now it would be nice to see which countries are in which cluster. This is shown in Figure 11.14. The countries are in the 4th column of `vital`. They got read in as levels of a factor, which they are not, so we'll turn them into character strings first. Then we want to split up the countries according to which cluster they're in, which is a job for `split`.

Let's have a look at our clusters. Cluster 3 is easiest to characterize. With the highest birth, death and infant mortality rates, these are countries that suffer from civil war, famine and so on: very poor countries. Cluster 1 also has above-average rates, but not as high as Cluster 3; these are developing countries. Cluster 2 has only two countries in it, the seemingly unrelated Mexico and pre-boom Korea, where birth and infant mortality rates are a little lower than average but the death rate is a lot higher. Cluster 4 is the "first world", including the then Eastern Bloc countries. The countries in Cluster 5 have a slightly lower than average death rate and slightly higher than average birth rate; these are countries were not yet part of the "first world", but nor yet are they as poor as the countries in clusters 1 and 3.

A word of warning: there is random number generation involved in `kmeans`, so if you repeat this analysis, you might get different answers. There is an option `nstart` to `kmeans`; if, say, you set `nstart=10`, R runs K-means 10 times, from different random starting points. The clustering it returns is the one out of these 10 that had the smallest total within-cluster sum of squares for the number of clusters you asked for. I got different results when I ran `kmeans` again on my data, which is a suggestion that using `nstart` might have been a good idea. There is no guarantee, even this way, that you'll always get the same solution, but your chances are better using `nstart`.

```
> cy=as.character(vital[,4])
> split(cy,vital.km$cluster)

$`1`
 [1] "Bolivia"    "Iran"       "Bangladesh" "Botswana"   "Gabon"
 [6] "Ghana"      "Namibia"    "Swaziland"  "Uganda"     "Zaire"
[11] "Cambodia"   "Nepal"      "Congo"      "Kenya"      "Nigeria"
[16] "Sudan"      "Tanzania"   "Zambia"

$`2`
[1] "Mexico" "Korea"

$`3`
[1] "Afghanistan"  "Sierra_Leone" "Angola"       "Ethiopia"     "Gambia"
[6] "Malawi"       "Mozambique"   "Somalia"

$`4`
 [1] "Albania"              "Czechoslovakia"       "Hungary"
 [4] "Romania"              "USSR"                 "Ukrainian_SSR"
 [7] "Chile"                "Uruguay"              "Finland"
[10] "France"               "Greece"               "Italy"
[13] "Norway"               "Spain"                "Switzerland"
[16] "Austria"              "Canada"               "Israel"
[19] "Kuwait"               "China"                "Singapore"
[22] "Thailand"             "Bulgaria"             "Former_E._Germany"
[25] "Poland"               "Yugoslavia"           "Byelorussia_SSR"
[28] "Argentina"            "Venezuela"            "Belgium"
[31] "Denmark"              "Germany"              "Ireland"
[34] "Netherlands"          "Portugal"             "Sweden"
[37] "U.K."                 "Japan"                "U.S.A."
[40] "Bahrain"              "United_Arab_Emirates" "Hong_Kong"
[43] "Sri_Lanka"

$`5`
 [1] "Ecuador"      "Paraguay"     "Oman"         "Turkey"       "India"
 [6] "Mongolia"     "Pakistan"     "Algeria"      "Egypt"        "Libya"
[11] "Morocco"      "South_Africa" "Zimbabwe"     "Brazil"       "Columbia"
[16] "Guyana"       "Peru"         "Iraq"         "Jordan"       "Lebanon"
[21] "Saudi_Arabia" "Indonesia"    "Malaysia"     "Philippines"  "Vietnam"
[26] "Tunisia"
```

Figure 11.14: Membership of each cluster

Chapter 12

Multi-dimensional scaling

12.1 Introduction

Some random text.

The object of multi-dimensional scaling is to produce a "map" of the individuals starting from a collection of distances or dissimilarities. This is an alternative to a cluster analysis. To see how the analyses compare for the languages data, compare Section 11.2 and Section 12.3. Clustering and scaling share as their aim to detect which individuals are "similar" (in same cluster, close together on a map), and which are "dissimilar" (in different clusters, far apart on a map).

12.2 Metric multi-dimensional scaling

The classical example of metric multi-dimensional scaling is to start with distances between cities and see how well a map of the cities is reproduced. This is a classical example because a pair of cities that is twice as far apart as another pair of cities in actuality should also be twice as far apart on the map.

Metric multi-dimensional scaling has a "solution" that can be worked out. I put "solution" in quotes because it's not unique, but whatever solution you get is equally good. I'll explain more about that when we look at the multi-dimensional scaling map for the cities.

My cities are in Europe, and the distances between them are road distances in kilometres. See Figure 12.1. First, we read the distances in, and then we convert them into a `dist` object. The `-1` column selection on the cities means "select all

```
> cities=read.csv("europe.dat",header=T)
> dd=as.dist(cities[,-1])
> dd

           Amsterdam Athens Barcelona Berlin Cologne Copenhagen Edinburgh
Athens          3082
Barcelona       1639   3312
Berlin           649   2552      1899
Cologne          280   2562      1539    575
Copenhagen       904   3414      2230    743     730
Edinburgh       1180   3768      2181   1727    1206       1864
Geneva          1014   2692       758   1141     765       1531      1536
London           494   3099      1512   1059     538       1196       656
Madrid          1782   3940       628   2527    1776       2597      2372
Marseille       1323   2997       515   1584    1208       1914      1860
Munich           875   2210      1349    604     592       1204      1743
Paris            515   3140      1125   1094     508       1329      1082
Prague           973   2198      1679    354     659       1033      1872
Rome            1835   2551      1471   1573    1586       2352      2467
Vienna          1196   1886      1989    666     915       1345      2098
           Geneva London Madrid Marseille Munich Paris Prague Rome
Athens
Barcelona
Berlin
Cologne
Copenhagen
Edinburgh
Geneva
London        867
Madrid       1386   1704
Marseille     443   1192   1143
Munich        591   1075   1877      1034
Paris         546    414   1268       809    827
Prague        954   1204   2307      1397    363  1094
Rome         1093   1799   2099       856    969  1531   1370
Vienna       1055    912   2617      1414    458  1285    312 1168
```

Figure 12.1: Road distances between European cities

the columns except the first", which has the city names in it. The city names are also the row headers, so they find their way into `dd`, as you see.

Now for the analysis. This is shown in Figure 12.2. The function we use is called `cmdscale`, and it requires two things: a `dist` object, and the number of dimensions the solution should have. A two-dimensional solution will draw a map on a sheet of paper; it's a little difficult to plot a three-dimensional solution! By default, `cmdscale` returns just one thing: the coordinates of the points for plotting on a map. We want to plot these points, but we want to label the points by which city they belong to. This means a `plot` to plot the mapped locations (as circles), followed by a `text` to add labels to those points. `text` requires the points to be plotted, and the things to be plotted there (by looking at `city.mds` you see that the row names of that are what we need). The last thing, `pos`, positions the labels so that they don't overwrite the circles; `pos=4` positions the labels to the right of the circles.

One small annoyance is that the name "Athens" disappears off the right side of the map. This could be fixed by extending the x-axis a bit more, eg. by specifying `xlim=c(-1500,3000)` in the `plot` command.

Now, this is where you need to call on your knowledge of European geography. I've given you a helping hand in Figure 12.3. Roughly speaking, the corners of our region are: Edinburgh, northwest; Copenhagen, northeast; Athens, southeast, Madrid, southwest. How does that correspond to our map? I think if you rotate our map about 45 degrees, it'll correspond reasonably well to the real thing.

This brings us to a key point about multidimensional scaling: since it is only based on distances, the map it produces could be rotated (R doesn't know, here, which way is north) and it could also be flipped over, for example by having east and west reversed but north and south correct. This is what I meant earlier when I said put "solution" in quotes; if you have one solution, any other solution differing from it by a rotation or flipping-over is equally good in terms of reproducing the distances. Mathematicians use the term "reflection" for flipping-over, the idea being that if you look at a map of Europe in a mirror, the distances will all be correct even if the orientation is wrong.

Now, how do we know the answer we got is any good? We can gain some insight into this by choosing a non-default option in `cmdscale`. This adds a couple of things to what is returned by `cmdscale`. Figure 12.4 shows the gory details. The *eigenvalues* are generally huge numbers in scientific notation; I used `format` to print them out in human-readable form. With 2 dimensions, you are looking for the first two eigenvalues to be clearly bigger than the others; I think that's true here. The second eigenvalue is over three times bigger than the third, so adding a third dimension doesn't reproduce the distances much better. Also, the `GOF` component of the result gives a couple of R-squared-like measures of fit (with a maximum of 1). Here they are over 70%. The last two lines of code make a

```
> city.mds=cmdscale(dd,2)
> city.mds

                  [,1]        [,2]
Amsterdam   -348.162277   528.26574
Athens      2528.610410  -509.52081
Barcelona   -695.970779  -984.60927
Berlin       384.178025   634.52387
Cologne        5.153446   356.72299
Copenhagen  -187.104072  1142.59261
Edinburgh   -882.179667   893.77411
Geneva      -161.260754  -330.09391
London      -433.798179   427.08450
Madrid     -1364.334083 -1068.80684
Marseille   -389.700778  -706.25599
Munich       345.259974   -66.83890
Paris       -556.152364    64.86436
Prague       531.887973   253.60513
Rome         380.950521  -870.47195
Vienna       842.622605   235.16435

> plot(city.mds)
> text(city.mds,row.names(city.mds),pos=4)
```

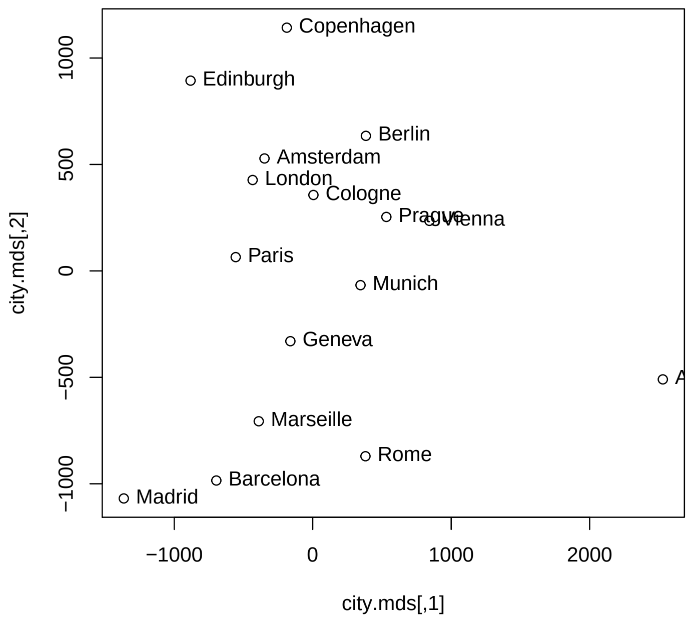


Figure 12.2: Multi-dimensional scaling for the cities

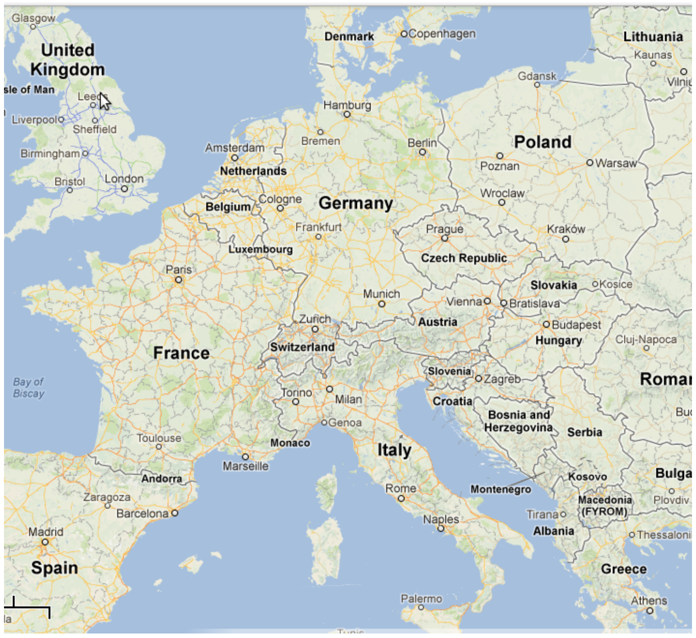


Figure 12.3: Map of Europe

```
> city2.mds=cmdscale(dd,2,eig=T)
> format(city2.mds$eig,scientific=F)

 [1] "11754342.065788399428129" " 6960876.237940214574337"
 [3] " 2034973.310557601274922" " 1408540.947581887012348"
 [5] "  457252.591273685451597" "  356301.402791602537036"
 [7] "  136423.478899375768378" "   18248.147736219922081"
 [9] "       -0.000000001164153" "   -3109.639123534318060"
[11] "  -70994.260254319757223" "  -82323.289817726705223"
[13] " -108940.313708261819556" " -377200.258322515059263"
[15] " -747384.564129272825085" "-1184037.419713367009535"

> city2.mds$GOF

[1] 0.7281918 0.8092382

> city3.mds=cmdscale(dd,3,eig=T)
> city3.mds$GOF

[1] 0.8073707 0.8972296
```

Figure 12.4: Non-default option in `cmdscale`

```
> dp=dist(city.mds)
> r=dd-dp
> sort(r)

  [1] -378.7685159 -200.7729778 -136.7718022 -124.6497268 -124.2528790
  [6] -124.1037948 -123.8724512 -117.4144115 -112.7580840 -110.9463691
 [11] -110.3495996 -105.9980075 -104.5780586  -98.4417012  -97.0845032
 [16]  -91.0088573  -87.1657601  -83.9829586  -79.0451207  -64.7924180
 [21]  -57.7003036  -54.5551578  -47.3590489  -45.6458549  -38.9711566
 [26]  -38.7745569  -30.3266519  -30.1538927  -27.5024862  -25.7656428
 [31]  -21.5240875  -17.7761109  -12.5082066  -10.2892981   -7.9428931
 [36]   -7.8293167   -5.4240311   -4.8026517   -3.8488260    0.7186579
 [41]    2.2173926    2.9059699    7.0622939    7.0853433    8.8175231
 [46]   20.1193235   20.1528663   23.7616186   25.4784737   25.5547419
 [51]   31.6729483   32.8524823   35.9111676   42.7795843   42.8747028
 [56]   47.8213623   48.7899807   51.0852032   52.6085091   53.6040691
 [61]   54.0842566   57.8019412   58.0037358   58.0867429   58.3097247
 [66]   59.8226410   60.4168603   62.2666375   66.2535356   66.3388146
 [71]   68.0007178   68.0467602   68.7547070   72.0649708   74.0537140
 [76]   76.1679253   79.8126971   85.9694590   86.6593404   87.7796398
 [81]   91.7738556   93.4448620  101.1378739  103.1186612  105.0713273
 [86]  115.5856179  121.4797941  122.2667738  135.5277697  152.5619679
 [91]  164.5747991  168.7999424  184.3514099  185.8937344  189.8359082
 [96]  191.2782786  206.9849577  215.1302441  222.8553438  235.2291860
[101]  235.8344943  251.7308706  257.6374569  260.3224543  266.8550393
[106]  268.9116169  293.4095070  297.1930835  302.5550740  319.7746224
[111]  327.4926539  342.4821154  351.0603952  361.4437810  373.2193114
[116]  388.0471783  434.3776676  439.1579786  532.8748594 1125.7312636
```

Figure 12.5: Comparing actual and predicted distances

three-dimensional map; we're not going to look at the map itself, but just see whether it fits any better. It does, a little; both measures are now over 80%.

Another way to understand how well the actual distances match up with the predicted distances is to calculate the predicted distances from the scaling coordinates, and calculate the residuals (observed minus predicted). Any residuals that are large (positively or negatively) indicate city pairs that are not well reproduced on the map. This turns out to be ludicrously simple to arrange (Figure 12.5): run `dist` on the coordinates that come out of (the default) `cmdscale`, then calculate the residuals as the difference between the measured road distances and the predicted ones. The last line produces the residuals sorted by size; residuals less than -350 (just one of these) and bigger than 350 look a bit sizeable. Eyeballing the residual matrix produces Table 12.1.

City	City	residual
Edinburgh	Amsterdam	532
Copenhagen	London	439
Edinburgh	Berlin	434
Barcelona	Rome	388
Vienna	London	-378
Athens	Rome	373
London	Amsterdam	361
Madrid	Athens	351

Table 12.1: Table of residuals sorted by absolute size

If you look at the large positive residuals, they generally have to do with cities at the extremes of the map (Edinburgh, Copenhagen, Athens). But some of the map distances from these cities are reproduced not so badly. It's not just being extreme; look back at the map, Figure 12.3. The distances in the data sets are road distances, but the most direct route between the pairs of cities with large positive residuals is over water. So the road distances are not accurate representations of where things should end up on a map. (This is perhaps why a three-dimensional representation was a bit better than a two-dimensional one.)

As another example, let's use our vital statistics data (our example for K-means clustering). We had data on birth rates, death rates and infant mortality rates for a large number of countries. Because these variables had different means and SDs, we decided to `scale` them to have the same mean and SD, so that no one variable had a disproportionate influence on the result. We'll use that standardized data, `vital.s`, again here.

The first thing we have to do is to calculate a distance matrix. R has a function `dist` that converts data to distances. There are different ways in which this can be done; here we'll use the default, Euclidean distance. This is shown in Figure 12.6. Next, we run `cmdscale` to get map coordinates for the countries. We're asking for the default two dimensions, so we don't need to specify that. Plotting the object returned from `cmdscale` gives the map, to which we add the country names, abbreviated.

We want to plot a few letters to represent each country. How are we going to get those? We can use a handy command called `abbreviate`, designed for just such a thing. This function requires two things: a vector of names to abbreviate (here the countries, in the fourth column of `vital`), and the minimum length of the abbreviations produced. The abbreviations are guaranteed to be different for each country. If two letters are enough to distinguish a country from the others, R will use two letters; if not, R will use more. The first few abbreviations are shown. The USSR is rather interestingly labelled `US`!

Now, to plot the abbreviations instead of the circles we would otherwise get,

```
> vital.s=scale(vital[,1:3])
> vital.d=dist(vital.s)
> vital.map=cmdscale(vital.d)
> plot(vital.map)
```

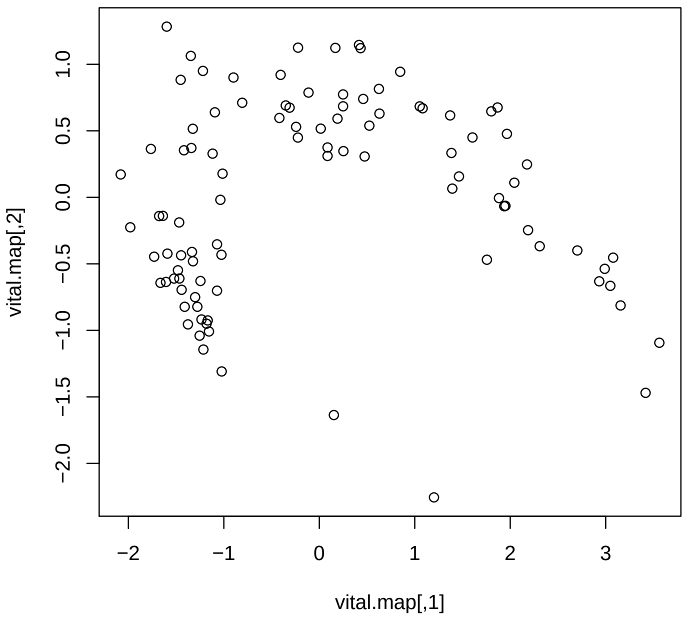


Figure 12.6: Metric MDS for vital statistics data

```
> head(vital[,4])

[1] Albania        Czechoslovakia Hungary        Romania        USSR
[6] Ukrainian_SSR
97 Levels: Afghanistan Albania Algeria Angola Argentina Austria ... Zimbabwe

> ca=abbreviate(vital[,4],2)
> head(ca)

       Albania Czechoslovakia        Hungary        Romania            USSR
         "Alb"           "Cz"           "Hn"           "Rm"            "US"
 Ukrainian_SSR
         "U_S"

> plot(vital.map,type="n")
> text(vital.map,ca,cex=0.5)
```

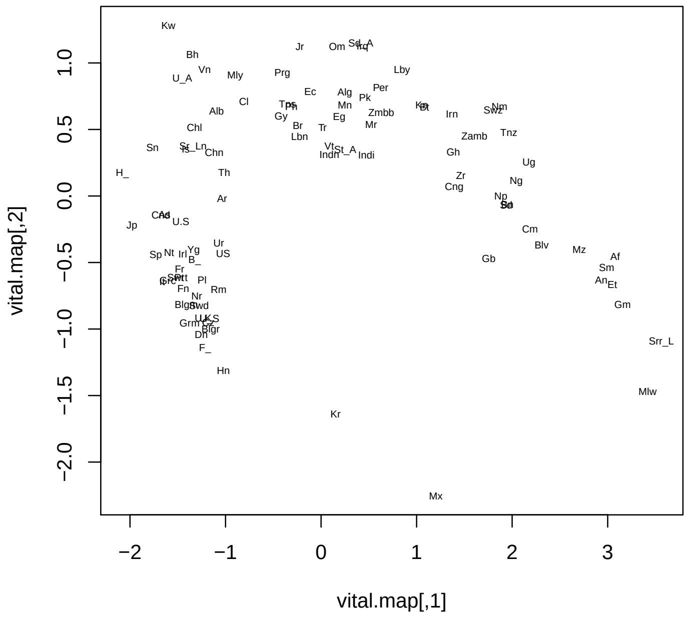


Figure 12.7: Better map of vital statistics data

we plot nothing (which merely sorts out the axes), and then use `text` on the resulting empty graph to put the country abbreviations in the right places. One more thing: with regular-sized text, the names will be too crowded, so we'll make the text smaller by "expanding" it to half its normal size, using `cex`. The abbreviations are now a bit small, but some of them still overlap, so this is probably about as well as we're going to do.

Most of the countries on the map in Figure 12.7 lie on a kind of horseshoe with the first world on the left and the third world on the right. You can see Mexico and Korea off by themselves at the bottom.

It's interesting to compare the arrangement of countries on this map with the results of the K-means cluster analysis on the same data. You can see why Mexico and Korea came out in a cluster by themselves, and there seem to be four (or maybe five) groups of countries arranged along the horseshoe, with the third-world group on the right being more dispersed than the others. (Is that one group or two?) You can check for yourself how well the groups that appear to be formed on this map correspond to the K-means clustering. Or to any of the different results that K-means clustering produces.

*********** cube example, how 3 dimensions much better than 2

12.3 Non-metric multidimensional scaling

***** languages again

Sometimes the dissimilarity matrix is not metric, in that on our map we don't want places twice as far apart in reality to be twice as far apart on the map. We would be happy for places that are farther apart just to be farther apart on the map. In other words, dissimilarity in that case is an ordinal scale, not a ratio one. An example of data that we would want to treat ordinally is the languages data, shown in Figure 12.8. Languages whose dissimilarity is 4 rather than 2 are not "twice as dissimilar" in the same way that cities 400 km apart are twice as far from each other as cities 200 km apart.

For the languages data, therefore, we only want languages whose dissimilarity is high to be farther apart from those whose dissimilarity is low.

Unlike metric multidimensional scaling, there is no exact solution to this. Indeed, we wouldn't expect there to be, because the two farthest-apart individuals could be moved farther apart on the map, and the map would be equally good. The process, therefore, is to start with a guess at the locations of the individuals, see where the order of the dissimilarities fails to match the order of the distances on the map, re-locate points, and try again. Repeat until no improvement is possible.

```
> lang

   en no dk nl de fr es it pl hu fi
en  0  2  2  7  6  6  6  6  7  9  9
no  2  0  1  5  4  6  6  6  7  8  9
dk  2  1  0  6  5  6  5  5  6  8  9
nl  7  5  6  0  5  9  9  9 10  8  9
de  6  4  5  5  0  7  7  7  8  9  9
fr  6  6  6  9  7  0  2  1  5 10  9
es  6  6  5  9  7  2  0  1  3 10  9
it  6  6  5  9  7  1  1  0  4 10  9
pl  7  7  6 10  8  5  3  4  0 10  9
hu  9  8  8  8  9 10 10 10 10  0  8
fi  9  9  9  9  9  9  9  8  9  8  0
```

Figure 12.8: Languages data: dissimilarities

The quality of the map is described by a **stress** quantity that measures how the observed dissimilarities correspond with map distances. This is rather the opposite of the `GOF` measure that came out of metric multi-dimensional scaling.

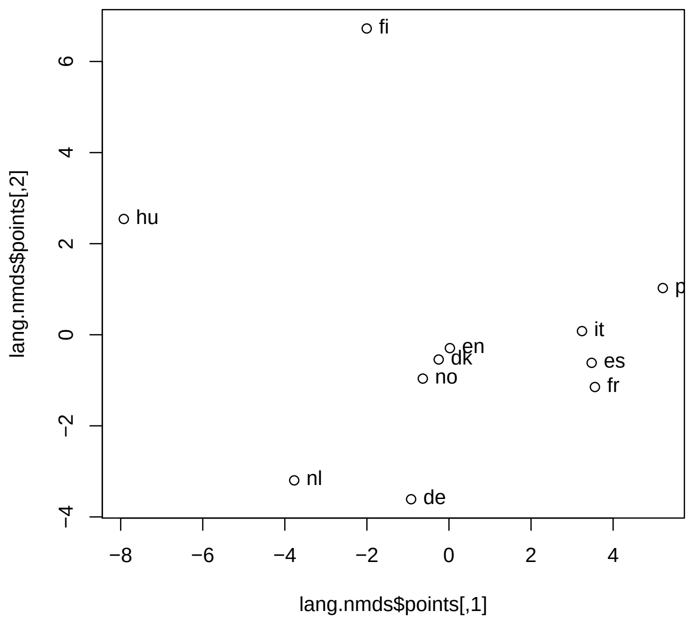
********** scale for good, adequate etc. stress

The R function that does this is called `isoMDS`, which lives in the MASS package. The first step is to install that package, if you don't already have it (by `install.packages("MASS")`). `isoMDS` is used like `cmdscale`: you feed it a distance matrix (eg. from `dist`) and a desired number of dimensions, typically 2 (called `k`).

The process is shown in Figure 12.9. First we have to use the `MASS` package. Then we turn the dissimilarity matrix into a `dist` object, and feed it into `isoMDS`, saving the result. The result is a list with two components, one of which is the stress value, and the other is a list of coordinates. The stress here is 5.3 (percent), which is nice and small. Now we make a plot. We can plot the points, but that won't make much sense without the languages attached. The language abbreviations are the `row.names` attribute of `lang.nmds$points`, so we'll pull those out and save them in a variable with a short name. Then we plot the points, and label each one with the name of the language it belongs to. You recall (don't you?) that `pos=4` prints the text just to the *right* of the point that it belongs to.

Looking at the map, you see a tight group of Romance languages (French, Spanish, Italian), an even tighter group of Scandinavian languages (English, Norwegian, Danish), with Dutch and German somewhere near the Scandinavian languages, Polish somewhere near the Romance languages, and Hungarian and Finnish off by themselves. Note the similarity in conclusion from the cluster

```
> library(MASS)
> dd=as.dist(lang)
> lang.nmds=isoMDS(dd,k=2)

initial  value 12.404671
iter   5 value 5.933653
iter  10 value 5.300747
final  value 5.265236
converged

> lang.nmds

$points
          [,1]        [,2]
en  0.02229429 -0.29368916
no -0.63766804 -0.96290425
dk -0.25008532 -0.54313754
nl -3.77093517 -3.19707086
de -0.92214625 -3.61145509
fr  3.55647800 -1.14723538
es  3.47737980 -0.61672972
it  3.24133841  0.08015195
pl  5.21156555  1.02530833
hu -7.92390808  2.54182929
fi -2.00431320  6.72493243

$stress
[1] 5.265236

> n=row.names(lang.nmds$points)
> plot(lang.nmds$points)
> text(lang.nmds$points,n,pos=4)
```



analyses.

One small annoyance is that we see only half of `pl` (for Polish). The reason for this is that when R lays out the scales for the plot, it only considers where the points are going to go (since it doesn't know we're about to plot some labels to the right of the points). The right-most point is going to have its label not quite fit. We can fix this by looking at the plot, and then realizing that if the x-axis stretched as far as 6, we'd have enough room for the label. Then we try again by specifying `xlim` in the call to plot, as

$$\texttt{plot}(\texttt{lang.nmds\$points}, \texttt{xlim} = (\texttt{c}(-8, 6)))$$

which makes the x-axis go from -8 to 6. (There are cleverer approaches that involve finding the maximum x coordinate of all the points, and setting `xlim` to go from the minimum x coordinate to the maximum one plus a little bit, but looking at the plot and fixing it up is the most you're likely to want to do.)

A final remark about the languages. If you look at Figure 11.1, you'll see that the Dutch and German number words are very similar structurally. They just happen not to begin with the same letter all that often! With Dutch, German and what I just called the Scandinavian languages, you can see that the number words are similar but begin with different but similar-sounding letters like t and d, f and v. So the crudeness of our measure of dissimilarity got us in the end, but I think you'll agree that we were able to draw some good conclusions along the way.

Chapter 13

Discriminant analysis

13.1 Introduction

Cluster analysis is all about finding groups when you have no prior knowledge about what they might be (or you do, but you want to see whether the groups you have in mind correspond to what comes out of the analysis). Discriminant analysis starts from the idea that you have known groups and a number of variables, and you want to see what it is about those variables that go with the individuals being in different groups.

13.2 Example: two groups

Our example is shown in Figure 13.1. Eight plants were grown; four of them were given a high dose of fertilizer, and the other four were given a low dose. There were two response variables; seed yield, and weight per seed.

There are a couple of possible research questions. One is the ANOVA kind of question: does the amount of fertilizer affect the seed yield or weight or both? This is the domain of Chapter 15. Another kind of question is "how would you characterize the difference between the groups, in terms of the other variables measured?". This is what discriminant analysis does.

The plot in Figure 13.1 shows the two quantitative variables plotted against each other, with the two groups shown by symbols of different colour (`col=fno`) and different shape (`pch=fno`). We had to do a little fiddling beforehand: the fertilizer `fert` is a factor, not a number, so we had to turn it into a number using `as.integer` first.

```
> fertilizer=read.table("fertilizer.txt",header=T)
> fertilizer

  fert yield weight
1  low    34     10
2  low    29     14
3  low    35     11
4  low    32     13
5 high    33     14
6 high    38     12
7 high    34     13
8 high    35     14

> fno=as.integer(fertilizer$fert)
> plot(fertilizer$yield,fertilizer$weight,col=fno,pch=fno)
```

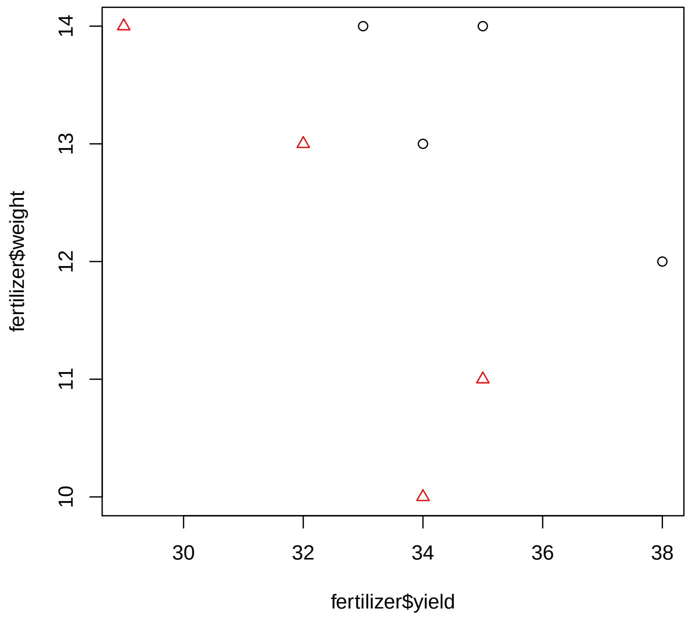


Figure 13.1: Fertilizer data

```
> library(MASS)
> f.lda=lda(fert~yield+weight,data=fertilizer)
> f.lda

Call:
lda(fert ~ yield + weight, data = fertilizer)

Prior probabilities of groups:
high  low
 0.5  0.5

Group means:
     yield weight
high  35.0  13.25
low   32.5  12.00

Coefficients of linear discriminants:
              LD1
yield  -0.7666761
weight -1.2513563
```

Figure 13.2: Discriminant analysis for fertilizer groups

R's function for doing discriminant analysis is called `lda` ("linear discriminant analysis") and it lives in the package called MASS.

The basic analysis is shown in Figure 13.2. This contains the group means on the two variables. Our analysis shows that high fertilizer had a slightly higher yield and weight than low fertilizer. But this is not the whole story, as the plot in Figure 13.1 shows: when the yield and weight are high *together* the fertilizer is high, and when they are low *together* the fertilizer is low.

How does this come out of the discriminant analysis? Well, a clue is in "coefficients of linear discriminant", where the cofficients are of the same sign. This suggests that when the two variables are both high, the observation is in one group, and when they are both low, the observation is in the other.

The key calculation is of the **discriminant scores** for each observation (plant). This shows where each plant comes on the overall high-low scale, and thus gives a hint about which group it belongs to. R chooses the discriminant function (or functions, here only one) to best separate out the groups. The easiest way to see the discriminant scores is to plot the fitted `lda` object. This shows, for each group separately, how the discriminant scores stack up. You can see that the plants in the high fertilizer group have a negative discriminant score, and the plants in the low fertilizer group have a positive one. It seems pretty clear

```
> plot(f.lda)
```

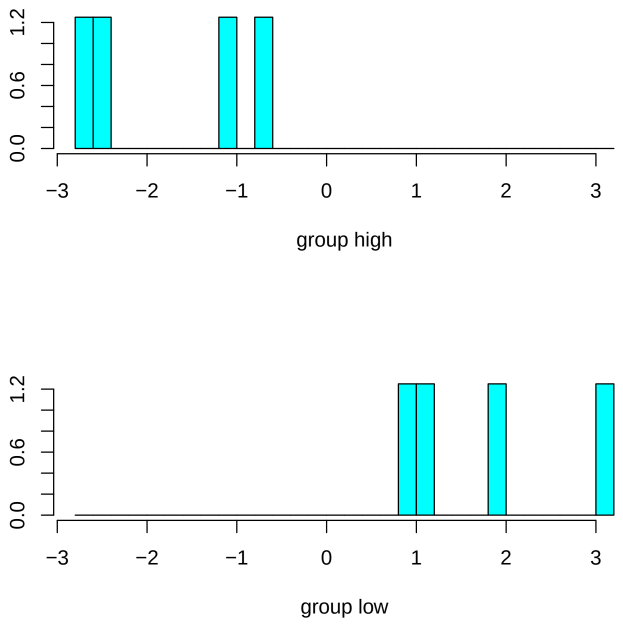


Figure 13.3: Plot of discriminant scores

```
> f.pred=predict(f.lda)
> f.pred$class

[1] low  low  low  low  high high high high
Levels: high low

> table(fertilizer$fert,f.pred$class)

       high low
  high    4   0
  low     0   4

> round(f.pred$posterior,4)

    high    low
1 0.0000 1.0000
2 0.0012 0.9988
3 0.0232 0.9768
4 0.0458 0.9542
5 0.9818 0.0182
6 0.9998 0.0002
7 0.9089 0.0911
8 0.9999 0.0001

> f.pred$x

         LD1
1  3.0931414
2  1.9210963
3  1.0751090
4  0.8724245
5 -1.1456079
6 -2.4762756
7 -0.6609276
8 -2.6789600

> cbind(fertilizer,round(f.pred$posterior,4),f.pred$x)

  fert yield weight   high    low        LD1
1  low    34     10 0.0000 1.0000  3.0931414
2  low    29     14 0.0012 0.9988  1.9210963
3  low    35     11 0.0232 0.9768  1.0751090
4  low    32     13 0.0458 0.9542  0.8724245
5 high    33     14 0.9818 0.0182 -1.1456079
6 high    38     12 0.9998 0.0002 -2.4762756
7 high    34     13 0.9089 0.0911 -0.6609276
8 high    35     14 0.9999 0.0001 -2.6789600
```

Figure 13.4: Discriminant scores and posterior probabilities

which are the high and low fertilizer plants just from their yield and weight, as they go to make up the discriminant score.

You can do prediction on `lda` objects, as shown in Figure 13.4. This produces several things of interest. First, the predicted group membership of each observation. The best way to display this is to tabulate it with the actual group memberships; here, we see that all eight plants got classified into the correct group. Second, the "posterior probabilities". These are R's best guess at how likely each observation is to come from each group. You can see from this how clear-cut it is to judge which group each observation came from. Here, the probabilities are all close to 0 or 1, so it was pretty clear-cut. The only plant for which there was any doubt was #7, but even than it was rated more than 90% "high". I used `round` to display only a few decimal places, and, as a side effect, to get rid of the scientific notation that R often uses. Third, the discriminant scores. It's nice to have these for plotting, as we see later.

The last `cbind` just puts everything together. This shows that all the plants with negative discriminant score were correctly predicted to be high fertilizer, and all those with positive discriminant score were correctly predicted to be low. For example, plant 1 has a middling yield but the lowest weight, which makes it a "low" beyond all reasonable doubt.

`predict`, by default, is run on the original data. But you can predict for new data as well. One reason to do that is to produce some nicer plots that illustrate what the discriminant analysis is doing.

Our procedure is shown in Figure 13.5. We're going to predict for a bunch of combinations of yield and weight values that reflect the data values we had. We'll generate a bunch of new yield values that go from 29 to 38 in steps of 0.5 (`yy`), and a bunch of new weight values that go from 10 to 14 in steps of 0.5 (`ww`). The "steps of" values don't matter very much; we need enough new points to get the idea, but not so many that they overwhelm the plot. Then we make a data frame containing all combinations of `yield` and `weight`, which is what `expand.grid` does. Some of the data frame is shown next. In `expand.grid`, the first variable changes fastest, and the last slowest. You see that we have all the yields for weight 10, then all the yields for weight 10.5, and so on. I didn't *have* to use `expand.grid`; any way that got me the yield/weight combinations I wanted would have worked. Note that I used the same names `yield` and `weight` in my new data frame that I had in the original one; this is important because `predict` has to be able to match up the variables to know what to predict for.

Obtaining predictions for these new yields and weights is a piece of cake: just call `predict` on the fitted `lda` object, and give the data frame of new values second. The predictions, in `f.pred`, have `class` (group) predictions for the new values, `posterior` predicted probabilities for the new values, and discriminant scores `x` for the new values.

```
> yy=seq(29,38,0.5)
> ww=seq(10,14,0.5)
> f.new=expand.grid(yield=yy,weight=ww)
> f.new[16:25,]

   yield weight
16  36.5   10.0
17  37.0   10.0
18  37.5   10.0
19  38.0   10.0
20  29.0   10.5
21  29.5   10.5
22  30.0   10.5
23  30.5   10.5
24  31.0   10.5
25  31.5   10.5

> f.pred=predict(f.lda,f.new)
```

Figure 13.5: Predicting for new data

I can think of three nice plots that one might draw for these data. Each of these is overlaid on the plot of the original yield and weight values (and thus depend on our only having two variables).

First is a plot showing the predicted group membership, as shown in Figure 13.6. We begin by plotting the data using different colours and symbols, just as before. Now we'll extract the predicted group memberships from `f.pred` and abbreviate them to H for high and L for low, uppercase (which is what `toupper` does). In other people's R code, you'll see a lot of one function applied to the result of another function. I'm trying to avoid it in this book, because I think it's confusing, but if you start from the *inside* you should be OK: get the abbreviations first, and then convert them to uppercase.

Now we add the one-letter uppercase predicted group memberships to our plot. `points` adds points to a plot without starting a new plot (which `plot` would do). I'm adding two wrinkles here. The first two things fed in to `points` are the places you want to plot something. The `col` argument plots the points a different colour for each (predicted) group, as in the original `plot`, but I don't want to use the same colours as the data points, since then you won't be able to see them! The values of `g` are actually "high" and "low", so we turn them into numbers; colours number 1 and 2 (black and red) are in the original plot, and colours 3 and 4 (green and blue) are used for the points we're adding. The other wrinkle is `cex`, or "character expansion": we want to make the array of green and blue points smaller than the data points so that we can still see them.

```
> plot(fertilizer$yield,fertilizer$weight,col=fno,pch=fno)
> g=f.pred$class
> ga=toupper(abbreviate(g,1))
> points(f.new$yield,f.new$weight,col=as.numeric(g)+2,cex=0.3)
```

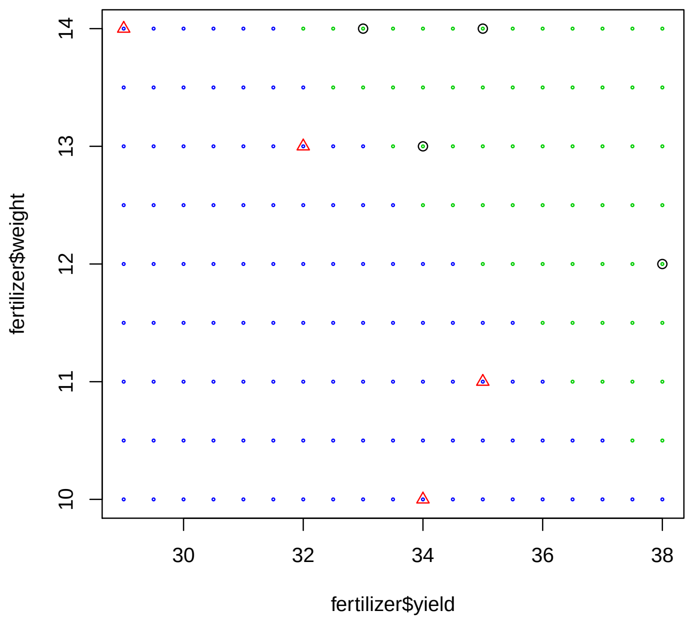


Figure 13.6: Plot of predicted group membership

You might (or might not) guess that the boundary between the predicted-highs and predicted-lows is actually a *straight line* (the resolution of the plot doesn't make it clear). The data points look pretty firmly within their respective areas, with the possible exception of the point with yield 34 and weight 13. This is the observation whose posterior probability of being a "high" was only 0.91, the least clear-cut of the lot. You can see how it is closest to the boundary.

The second and third plots involve *contours*. This is a little awkward to do in R, so we'll take a digression first.

Figure 13.7 shows some x and y values and a function, the innards of which don't need to concern you. `outer` evaluates the function for each value of `x` and each value of `y`, in all combinations, rather like `expand.grid`, but storing the results in a matrix. I've listed the values of `x`, `y` and `z` below the `outer` call so that you can see what happened. There are 8 values of `x`, shown in the rows, and 9 values of `y`, in the columns. In the 4th row, the 3rd value of `z` is 4. This is the result of evaluating the function at the 4th value of `x`, 1.0, and the 3rd value of `y`, -1.0. If you care to, you can evaluate $(1-y)^2+100(x-y^2)^2$ for this x and y and verify that you really do get 4.

Anyway, the purpose of getting the function evaluated at a number of points, and arranging them in a matrix, is to produce a plot. If you feed `x`, `y` and `z` into `persp`, you get a 3D or wireframe plot. Here, you see that the function increases dramatically to the left and less dramatically to the right, with a kind of curving river valley from the back right to the middle left to the front right. This is one way to look at a function of two variables.

Another way to look at this is shown in Figure 13.8. This is a *contour plot.* The numbers on the contours show the height of the function at that point, and the contours, or "level lines", connect up all the points that have the same value of the function. Once again you see the rapid increase of the function to the top left and bottom left (the contours are close together, indicating a rapid change), a more modest increase to the right, and something like a curving river valley. If you look carefully, you'll see that x and y have come out on unexpected axes. This is so that the stories from `persp` and `contour` come out the same. You can flip things back by calling `contour` as `contour(y,x,z)` if you prefer.

Figure 13.9 shows a variation on this contour plot. I wanted to get a better picture of the "river valley", so I chose the heights myself at which I wanted the contours drawn. This is done by specifying `levels=` a vector of heights. If you follow the 50 contour all the way around (it goes off the left side of the graph for a moment), you'll see why this is called "Rosenbrock's banana function". (The minimum of the function is 0 at $x=\pm1, y=1$, but at $x=0, y=0$ the function value is only 1. This explains the three little areas enclosed by height-5 contours in the river valley.)

All right, now we have what we need to draw contour plots for our fertilizer

```
> x=seq(-0.5,3,0.5)
> y=seq(-2,2,0.5)
> banf=function(x,y)
+  {
+    (1-y)^2+100*(x-y^2)^2
+  }
> z=outer(x,y,FUN="banf")
> x

[1] -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0

> y

[1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0

> z

     [,1]  [,2] [,3]  [,4] [,5]  [,6] [,7]  [,8] [,9]
[1,] 2034 762.5  229  58.5   26  56.5  225 756.5 2026
[2,] 1609 512.5  104   8.5    1   6.5  100 506.5 1601
[3,] 1234 312.5   29   8.5   26   6.5   25 306.5 1226
[4,]  909 162.5    4  58.5  101  56.5    0 156.5  901
[5,]  634  62.5   29 158.5  226 156.5   25  56.5  626
[6,]  409  12.5  104 308.5  401 306.5  100   6.5  401
[7,]  234  12.5  229 508.5  626 506.5  225   6.5  226
[8,]  109  62.5  404 758.5  901 756.5  400  56.5  101

> persp(x,y,z)
```

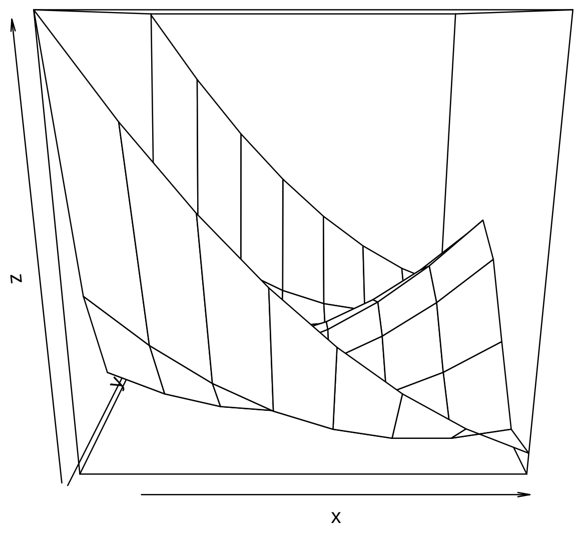


Figure 13.7: Rosenbrock's banana function in 3D

```
> contour(x,y,z)
```

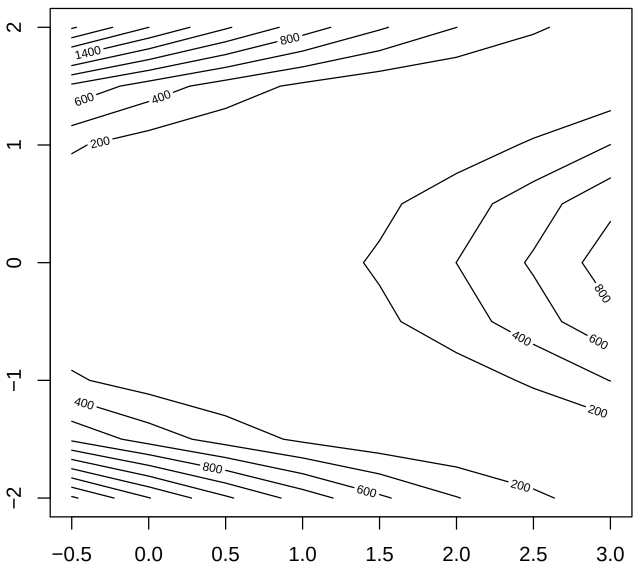


Figure 13.8: Rosenbrock's banana function, contours

```
> contour(x,y,z,levels=c(0,5,50,100,200,400,800))
```

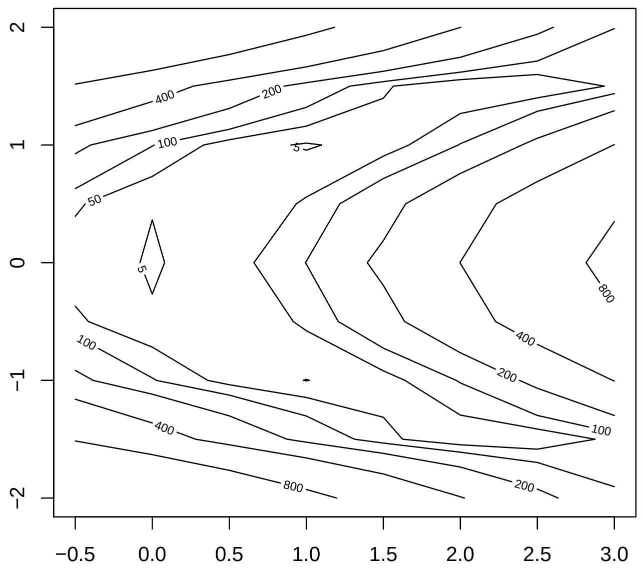


Figure 13.9: Contours by specifying `levels`

```
> plot(fertilizer$yield,fertilizer$weight,col=fno,pch=fno)
> yy=seq(29,38,0.5)
> ww=seq(10,14,0.5)
> z=matrix(f.pred$x,length(yy),length(ww),byrow=F)
> contour(yy,ww,z,add=T)
```

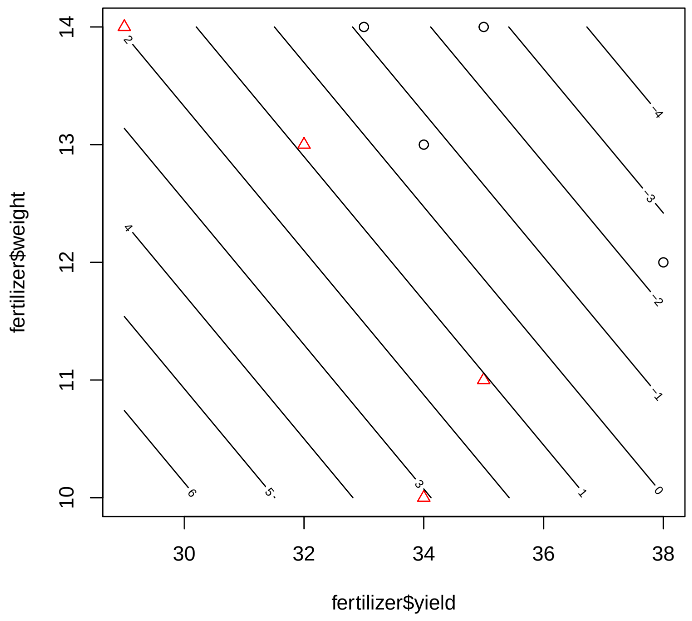


Figure 13.10: Contour plot of discriminant function

data. We are going to overlay contours on our original plot of the data: first, of the discriminant function, and second of the posterior probability of being a high-fertilizer plant.

Figure 13.10 shows how to do that. First we plot the original data, and then remind ourselves of the new yield and weight values we were predicting for. The discriminant function values are in `f.pred$x`, but they are in one long string. We need them to be in a matrix, as for the `contour` example above. A little bit of thinking reveals that they need to go *down* the columns of a properly-sized matrix. The size of the matrix is that it has as many rows as we have `yield` values (the first variable) and as many columns as we have `weight` values (the second variable). R has a function `matrix` that turns a vector into a matrix this way, but by default it goes along the rows instead of down the columns. But R

```
> plot(fertilizer$yield,fertilizer$weight,col=fno,pch=fno)
> z=matrix(f.pred$posterior[,1],length(yy),length(ww),byrow=F)
> contour(yy,ww,z,add=T)
```

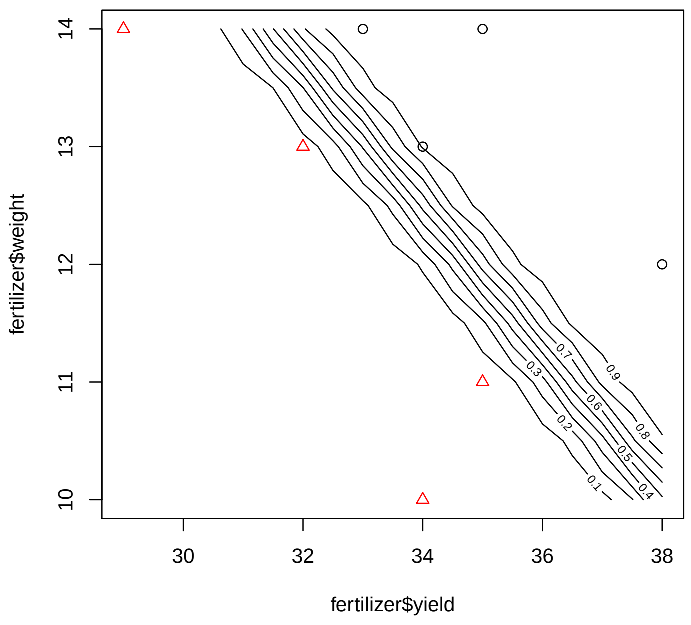


Figure 13.11: Contour plot of posterior probability of High

being R, we can fix that, by setting `byrow=F`.

Now we can draw the contours of the discriminant function. One thing: if we do `contour` as before, the original plot of the data values will be overwritten. Setting `add=T` adds the contours to the current plot, in the same way that `lines` or `points` work.

What is perhaps striking in Figure 13.10 is that the contours of the discriminant function are equally spaced straight lines. This is where the "linear" of "linear discriminant analysis" comes from. The discriminant function is negative in the top right, where a plant is declared to be "high fertilizer", and positive in the bottom left, where the "low fertilizer" plants live. The lines are slanted in such a way as to "best" separate the groups (see how the 0 contour has highs on one side and lows on the other). Contour lines that were horizontal or vertical wouldn't separate the highs from the lows, which is why looking just at yield or just at weight didn't tell the whole story.

```
> plot(fertilizer$yield,fertilizer$weight,col=fno,pch=fno)
> contour(yy,ww,z,add=T,levels=0.50,col="red")
> contour(yy,ww,z,add=T,levels=c(0.001,0.01,0.1,0.25,0.75,0.9,0.99,0.999))
```

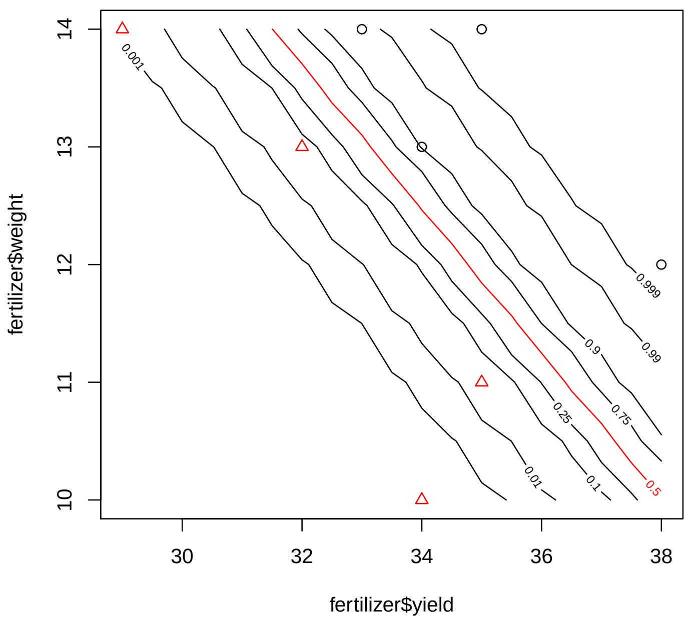


Figure 13.12: Revised contour plot of posterior probabilities

Another contour plot we can draw is of the posterior probability of “high”. This is in the output from `predict` in `f.pred$posterior[,1]`. (The first column is prob. of high, the second of low.)

The procedure, shown in Figure 13.11, is really just the same as in Figure 13.10, so I hope you can follow the steps. We construct a matrix from the posterior probabilities, then feed that into `contour`.

The plot shows a “river” of contours (no longer linear) running in between the four highs and the four lows. Which side of the river a plant is determines whether it is high or low fertilizer, and, if you will, the distance from the middle of the river determines how clear-cut the decision is. The plant with yield 34 and weight 13 is on the “edge” of the river drawn with these contours.

Actually, the contour of interest is really 0.5 (a plant on this contour would be equally well classified as high or low). So let’s change the contours from R’s

```
> jobs=read.table("profile.txt",header=T)
> jobs

           job reading dance tv ski
1  bellydancer       7    10  6   5
2  bellydancer       8     9  5   7
3  bellydancer       5    10  5   8
4  bellydancer       6    10  6   8
5  bellydancer       7     8  7   9
6   politician       4     4  4   4
7   politician       6     4  5   3
8   politician       5     5  5   6
9   politician       6     6  6   7
10  politician       4     5  6   5
11       admin       3     1  1   2
12       admin       5     3  1   5
13       admin       4     2  2   5
14       admin       7     1  2   4
15       admin       6     3  3   3
```

Figure 13.13: Jobs and leisure activities data

default, to replace the moderate probabilities with more extreme ones. This is done with `levels`, as shown in Figure 13.12. I've also plotted the key 0.5 contour in red, by first plotting the level 0.50 contour with `col="red"`, and then plotting all the other contours in the default colour of black.

This plot shows just how clear-cut some of the decisions are.

13.3 Example: three groups

Having seen how discriminant analysis works with two groups, let's try it with three. Our example is a fictitious one about people in three occupational groups (politicians, administrators and bellydancers), who are each asked to rate how much they like four different leisure activities: reading, TV watching, dancing and skiing. The data are shown in Figure 13.13. The ratings are on a scale of 0–10, higher is better.

Can we use the scores on the different activities to distinguish the jobs? Let's see what a discriminant analysis produces, as in Figure 13.14. The model formula in the `lda` line looks like a regression but "predicting" the group from the other variables (separated by plusses).

```
> library(MASS)
> jobs.lda=lda(job~reading+dance+tv+ski,data=jobs)
> jobs.lda

Call:
lda(job ~ reading + dance + tv + ski, data = jobs)

Prior probabilities of groups:
      admin bellydancer  politician
  0.3333333   0.3333333   0.3333333

Group means:
            reading dance  tv ski
admin           5.0   2.0 1.8 3.8
bellydancer     6.6   9.4 5.8 7.4
politician      5.0   4.8 5.2 5.0

Coefficients of linear discriminants:
                LD1        LD2
reading -0.01297465  0.4748081
dance   -0.95212396  0.4614976
tv      -0.47417264 -1.2446327
ski      0.04153684  0.2033122

Proportion of trace:
   LD1    LD2
0.8917 0.1083

> plot(jobs.lda)
```

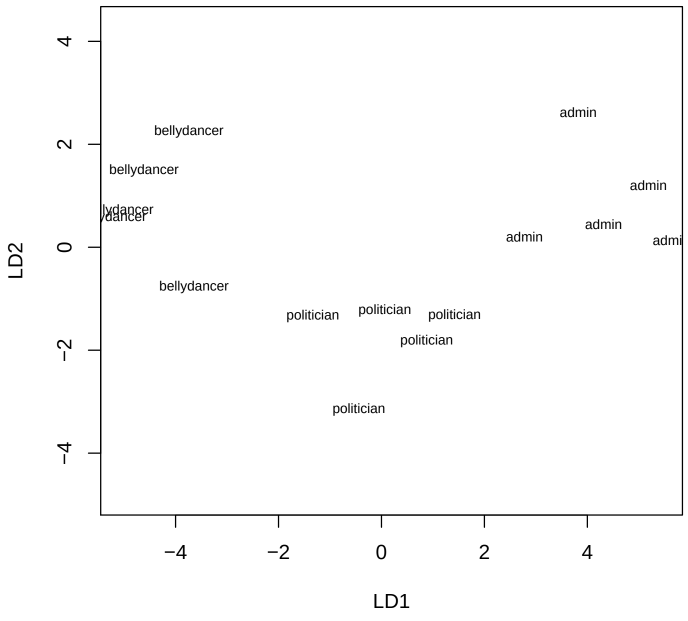


Figure 13.14: Discriminant analysis for jobs data

For the output: first up, the means on each activity for each job. The highest mean is 9.4, showing that the bellydancers love dancing (surprise). The lowest scores are 1.8 and 2.0, showing that the administrators hate dancing and TV-watching. If you were going to suggest a good way to separate the groups, you might look at dancing scores, which would separate the bellydancers from (especially) the administrators. The skiing scores don't have much to add, since they separate the groups in the same way as the dancing scores, only less dramatically. You might look at the TV-watching scores, since they distinguish the politicians from the administrators better than anything else.

The coefficients of linear discriminants tell you what `lda` came up with for separating the groups. You're looking for coefficients far away from zero, either plus or minus. On `LD1`, reading and skiing have nothing to say, but `LD1` will be most negative when the dance score is high (especially) or when the TV-watching score is high (not so much). This is about what we said above. On `LD2`, the most influential score is the TV one (negatively), with everything else having a small positive influence.

How come we got two discriminants? The number you can get comes from one less than the number of groups (here 2) and the number of variables (here 4). Take the smaller of those. That explains why we got two discriminants here and one before.

There are 4 variables here, so we can't plot them in any useful way. `Plot`ting the `lda` model object does one of several things: if there's only one discriminant, it plots histograms of the discriminant scores by group (as we saw above). With two discriminants, it plots the discriminant scores against each other (as here), and with more than two, it plots each one against each other one.

On the plot, you see the bellydancers on the left, the administrators on the right, and the politicians at the bottom. The groups look pretty well separated. LD1 actually separates all 3 of the groups, but LD2 helps to distinguish the politicians from the rest. (If you look back at the means, even though the TV-watching ratings for bellydancers and politicians are about the same, the bellydancers' other ratings are higher, and the politicians' are about the same. This is how LD2 distinguishes bellydancers from politicians. For the administrators, it's the difference in TV-watching ratings that dominates LD2.)

Now, let's have a look at the predictions, shown in Figure 13.15. First, a summary of how well the jobs were predicted from the activity scores. This is done by tabulating the actual jobs against the ones that `jobs.pred` predicted. I've added one extra thing here: feeding in a list for `dnn` labels the rows and columns of the table. Here, we see that all 15 individuals were correctly classified: the groups seem pretty distinct.

The last line gives the data, the discriminant scores (rounded to 2 decimals), and the posterior probabilities (rounded to 4 decimals), all side by side. The only

```
> jobs.pred=predict(jobs.lda)
> table(jobs$job,jobs.pred$class,dnn=list("actual","predicted"))

             predicted
actual        admin bellydancer politician
  admin           5           0          0
  bellydancer     0           5          0
  politician      0           0          5

> pp=round(jobs.pred$posterior,4)
> ds=round(jobs.pred$x,2)
> cbind(jobs,ds,pp)

           job reading dance tv ski   LD1   LD2  admin bellydancer politician
1  bellydancer       7    10  6   5 -5.24  0.58 0.0000      1.0000     0.0000
2  bellydancer       8     9  5   7 -3.74  2.25 0.0000      1.0000     0.0000
3  bellydancer       5    10  5   8 -4.61  1.49 0.0000      1.0000     0.0000
4  bellydancer       6    10  6   8 -5.10  0.72 0.0000      1.0000     0.0000
5  bellydancer       7     8  7   9 -3.64 -0.77 0.0000      0.9973     0.0027
6   politician       4     4  4   4  1.42 -1.33 0.0028      0.0000     0.9972
7   politician       6     4  5   3  0.88 -1.83 0.0001      0.0000     0.9999
8   politician       5     5  5   6  0.06 -1.23 0.0000      0.0000     1.0000
9   politician       6     6  6   7 -1.33 -1.33 0.0000      0.0021     0.9979
10  politician       4     5  6   5 -0.44 -3.15 0.0000      0.0000     1.0000
11       admin       3     1  1   2  5.63  0.14 1.0000      0.0000     0.0000
12       admin       5     3  1   5  3.82  2.62 1.0000      0.0000     0.0000
13       admin       4     2  2   5  4.32  0.44 1.0000      0.0000     0.0000
14       admin       7     1  2   4  5.19  1.20 1.0000      0.0000     0.0000
15       admin       6     3  3   3  2.78  0.20 0.9821      0.0000     0.0179
```

Figure 13.15: Predictions for jobs data

```
> plot(jobs.lda)
> jobs.new=expand.grid(reading=1:10,dance=1:10,tv=1:10,ski=1:10)
> jobs.pred2=predict(jobs.lda,jobs.new)
> gg=as.numeric(jobs.pred2$class)
> points(jobs.pred2$x,col=gg,cex=0.3)
```

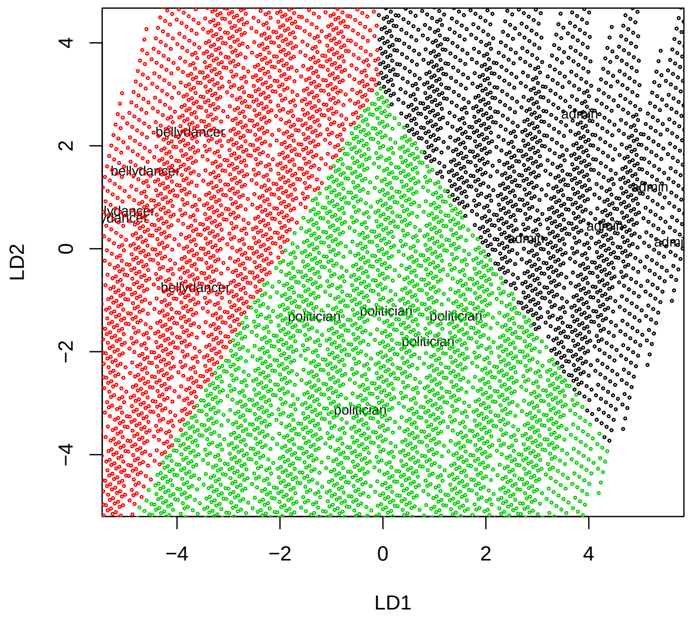


Figure 13.16: Group membership regions

individual about whom there is any doubt at all is number 15, who is reckoned to have a 2% chance of being a politician. So the groups really *are* clear-cut.

Why is individual 15 less surely an administrator than the others? Well, this person likes dancing as much as any of the administrators, and likes watching TV more than any of their colleagues. So this means that LD1 should be *lower* than their colleagues' (higher scores for negative things on LD1), and that LD2 should also be lower, because that's based mostly on TV-watching. (Individual 11 scores even lower on LD2 because they hate everything!) So individual 15 is more like a politician than any of the other administrators, but not very much.

Let's see whether we can make a plot of the regions, by LD1 and LD2, that favour each job. We do this in the same way that we made the other plots for the fertilizer data, but with a small twist given that we can't actually plot the data, only the discriminant scores.

First, we plot the fitted model object (which plots the discriminant scores, labelling the individuals). Then our strategy is to add a whole bunch of coloured circles to the plot, with colour according to the predicted job.

To make that strategy happen, we first need to construct a new data frame of activity ratings to predict from. I've used `expand.grid` again. The observed scores were all whole numbers from 1 to 10, so I've used that as the basis. (Recall that `1:10` means the numbers 1 through 10 in order.) The data frame `jobs.new` is *huge* (it has $10^4 = 10000$ rows), but we don't need to worry about that. Then we feed `jobs.new` into `predict` as the second thing. I called the prediction object `jobs.pred2`, though perhaps `jobs.pred.new` would have been better. `jobs.pred2` has the same things in it as `jobs.pred`: `class` has predicted group membership, `posterior` has the posterior probabilities of membership in each group, and `x` has the two columns of discriminant scores.

Now we need to plot the discriminant scores in `jobs.pred2` on the plot, with each point being an appropriate colour according to which group it's predicted to be in. First we turn the predicted group membership into numbers (saved in `gg`), and then we plot the points that are the discriminant scores for our multitude of predictions, in a colour according to predicted group, and making the plotted circles smaller (so they don't interfere with the labels for the individuals in our data set). The final plot is shown in Figure 13.16. The administrator region appears in black, the bellydancer region in red, and the politician region in a very non-partisan green. The boundaries of the regions are again straight lines.

What surprised me what that the `politician` region extended up so far: as long as LD1 is close to 0, LD2 can be up to about 3. But in any case, you can see the administrator that is more politician-like than the others, and one bellydancer who is more politician-like than the rest (though that posterior probability was only 0.0027).

13.4 Cross-validation

We have been assessing how well a discriminant analysis is performing by seeing how well the data get classified into their own groups. But this is cheating, really: the same data that are being used to estimate the discriminant functions are also being used to see how well they work.

Using the data in this way (sometimes called "resubstitution") tends to give an optimistically good assessment of how well the analysis is doing. What can we do that is more honest? One way is called "leave-one-out cross-validation". In our `jobs` data set we had 15 observations. We could use 14 of them to estimate the posterior probabilities for the 15th, repeating for each of the observations

```
> jobs.cv=lda(job~reading+dance+tv+ski,data=jobs,CV=T)
> table(jobs$job,jobs.cv$class,dnn=list("actual","predicted"))

             predicted
actual        admin bellydancer politician
  admin           5           0          0
  bellydancer     0           4          1
  politician      0           0          5

> post=round(jobs.cv$posterior,4)
> cbind(jobs,jobs.cv$class,post)

           job reading dance tv ski jobs.cv$class  admin bellydancer politician
1  bellydancer       7    10  6   5   bellydancer 0.0000      1.0000     0.0000
2  bellydancer       8     9  5   7   bellydancer 0.0000      1.0000     0.0000
3  bellydancer       5    10  5   8   bellydancer 0.0000      1.0000     0.0000
4  bellydancer       6    10  6   8   bellydancer 0.0000      1.0000     0.0000
5  bellydancer       7     8  7   9    politician 0.0000      0.0006     0.9994
6   politician       4     4  4   4    politician 0.0060      0.0000     0.9940
7   politician       6     4  5   3    politician 0.0008      0.0000     0.9992
8   politician       5     5  5   6    politician 0.0000      0.0000     0.9999
9   politician       6     6  6   7    politician 0.0000      0.0087     0.9913
10  politician       4     5  6   5    politician 0.0000      0.0000     1.0000
11       admin       3     1  1   2         admin 1.0000      0.0000     0.0000
12       admin       5     3  1   5         admin 1.0000      0.0000     0.0000
13       admin       4     2  2   5         admin 0.9999      0.0000     0.0001
14       admin       7     1  2   4         admin 1.0000      0.0000     0.0000
15       admin       6     3  3   3         admin 0.8188      0.0000     0.1812
```

Figure 13.17: Cross-validation for the jobs data set

(so there is 15 times as much work, but R can handle it). This is honest, because each prediction has nothing to do with the observation it is predicting for.

Let's see how this works out for our jobs data. Is it still true that the three groups of individuals are very well separated? The analysis is in Figure 13.17. Cross-validation is requested by feeding `CV=T` into the call to `lda`. What comes back from `lda` in this case is kind of like an abbreviated version of what would have come back from `predict`: it's a list with predicted groups `class` and predicted posterior probabilities `posterior`.

To see how good a job the classification has done now, we can tabulate the actual jobs and the predicted ones. The table shows now that one of the bellydancers has been classified as a politician! To see what happened, let's print out the data, predicted job and (rounded) posterior probabilities side by side and take a look.

The individual getting misclassified is #5. The posterior probabilities say that this person is almost certainly a politician. How did the cross-validation make such a difference? Well, when we came to predict for #5, there were only the other 4 bellydancers left, and they have very similar rating profiles. So the bellydancers were considered to be a very tight-knit group, and #5 was quite different from this. This meant that it didn't look as if #5 was a bellydancer at all.

Also #15 was "potentially" a politician. This individual liked TV and dancing the most out of the administrators, with a profile somewhat like that of the politicians.

One snag of doing `lda` this way is that you don't get any discriminant scores or plots. So you would do this to see whether it changes the posterior probabilities noticeably from the default way. If it doesn't, the plots, discriminant scores and so on from the default analysis can be trusted.

******** remote sensing

Chapter 14

Survival analysis

14.1 Introduction

There are some regression-type analyses, where you have one or more explanatory variables that might predict a response variable that is:

- counted or measured: regression (or possibly ANOVA)
- categorical: logistic regression (logistic regression, or one of the variants for multiple categories).

A logistic regression might be used, as in Section 10.5, to predict whether or not a person will survive having some disease. But a more nuanced analysis would also take into account *how long* that person survived. This is where **survival analysis** comes into the picture. Survival analysis can also handle the common situation where the event (eg. death) has not happened by the end of the study (the person might be still alive and you don't know when they might die). But knowing that a patient has survived for six months (say) without dying is informative, even if you don't know how long they survived altogether. So such people (known in the jargon as "censored") need to be included in the analysis.

We'll be using a model called the **Cox proportional hazards model**. This has a lot of nice features, including the ability to handle censored data, and there is some sophisticated mathematics behind it — which we don't need to worry about! As far as we are concerned, it all looks like a kind of regression model.

```
> library(survival)
> dance=read.table("dancing.txt",header=T)
> dance

   Months Quit Treatment Age
1       1    1         0  16
2       2    1         0  24
3       2    1         0  18
4       3    0         0  27
5       4    1         0  25
6       5    1         0  21
7      11    1         0  55
8       7    1         1  26
9       8    1         1  36
10     10    1         1  38
11     10    0         1  45
12     12    1         1  47

> attach(dance)
> mth=Surv(Months,Quit)
> mth

 [1]  1   2   2   3+  4   5  11   7   8  10  10+ 12
```

Figure 14.1: The dancing data

14.2 Example 1: still dancing?

We begin with a frivolous example. Imagine that you are the owner of a dance studio, and you want to know how long women tend to stick with dance lessons until they quit. (So the event of interest is "quit", the survival time is "time until quitting" and "censored" corresponds to "hasn't quit yet".)

Figure 14.1 shows the data. We need to use the package `survival`, so we'll get hold of that first. There are four columns: the number of months a dancer was observed dancing, an indication of whether the dancer actually quit (1) or was still dancing at the end of the study, a "treatment" (a visit to a dance competition, which might have an inspirational effect) and the age of the dancer at the start of the study.

If you think about it, you'll see that `Months` and `Quit` are a kind of combined response: you need to know them both to have a proper sense of what was being measured (how long was the woman observed dancing for, and do we know that she quit?). This needs to go into the model somehow. This is what the `Surv` statement does: it creates the appropriate response variable. The last line shows

```
> dance.1=coxph(mth~Treatment+Age)
> summary(dance.1)

Call:
coxph(formula = mth ~ Treatment + Age)

  n= 12, number of events= 10

              coef exp(coef) se(coef)      z Pr(>|z|)
Treatment -4.44915   0.01169  2.60929 -1.705   0.0882 .
Age       -0.36619   0.69337  0.15381 -2.381   0.0173 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

          exp(coef) exp(-coef) lower .95 upper .95
Treatment   0.01169     85.554 7.026e-05    1.9444
Age         0.69337      1.442 5.129e-01    0.9373

Concordance= 0.964  (se = 0.125 )
Rsquare= 0.836   (max possible= 0.938 )
Likelihood ratio test= 21.68  on 2 df,   p=1.956e-05
Wald test            = 5.67  on 2 df,   p=0.0587
Score (logrank) test = 14.75  on 2 df,   p=0.0006274
```

Figure 14.2: Cox proportional-hazards model for dance data

you what it does: a number is a number of months which went with the event (quitting), and a number with a plus sign is a censored observation (that had no quitting observed). You see the two dancers that were not observed to quit: one that was observed for three months and one for ten.

Next, in Figure 14.2, comes the model-fitting. Instead of `lm` or `glm`, we are using `coxph`. The response is the `Surv` object, and the explanatory variables are whatever they are, on the right side of the squiggle, glued together with plus signs. As usual, the way to look at the model with with `summary`.

Since we don't have much data, let's use $\alpha = 0.10$ to assess significance of things.

The test(s) of whether anything has an effect on survival time are at the bottom. There are actually three tests, that test slightly different things, but usually they will be in broad agreement. They are all significant here, though in the case of the Wald test, only just.

As to *what* has an effect on survival time: the table under "number of events" is the one to look at. The P-value for treatment is 0.0882 (marginal, but significant at the 0.10 level), and for age is 0.0173 (definitely significant).

```
> dance.new=expand.grid(Treatment=c(0,1),Age=c(20,40))
> dance.new

  Treatment Age
1         0  20
2         1  20
3         0  40
4         1  40

> s=survfit(dance.1,newdata=dance.new)
> t(dance.new)

          [,1] [,2] [,3] [,4]
Treatment    0    1    0    1
Age         20   20   40   40

> summary(s)

Call: survfit(formula = dance.1, newdata = dance.new)

 time n.risk n.event survival1 survival2 survival3 survival4
    1     12       1  8.76e-01  9.98e-01  1.00e+00     1.000
    2     11       2  3.99e-01  9.89e-01  9.99e-01     1.000
    4      8       1  1.24e-01  9.76e-01  9.99e-01     1.000
    5      7       1  2.93e-02  9.60e-01  9.98e-01     1.000
    7      6       1 2.96e-323  1.70e-04  6.13e-01     0.994
    8      5       1  0.00e+00  1.35e-98  2.99e-06     0.862
   10      4       1  0.00e+00  0.00e+00  3.61e-20     0.593
   11      2       1  0.00e+00  0.00e+00  0.00e+00     0.000
   12      1       1  0.00e+00  0.00e+00  0.00e+00     0.000
```

Figure 14.3: Using survfit to predict survival times

Now, having found out that both variables have at least some effect on survival time, it would be nice to do some predictions of probability of surviving until certain times under various conditions of age and treatment. The function is called `survfit`, though it actually works rather like `predict`. The details are shown in Figure 14.3.

First we create a new data frame of treatments (yes and no) and ages (20 and 40) to predict, using `expand.grid` to do all the combinations. We take a look at the new data frame. Then we run `survfit` in the same kind of way that we'd run `predict`, feeding it a fitted model object and a data frame of values to predict for.

The complicated thing about this is that we don't have just one prediction for

```
> plot(s)
```

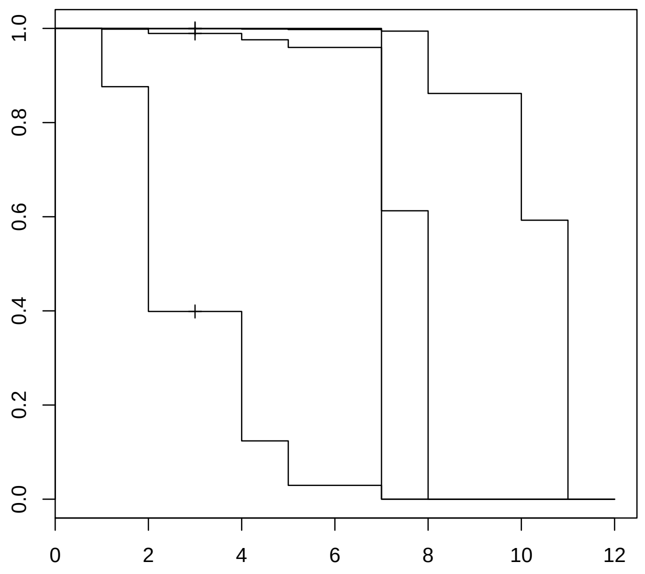


Figure 14.4: Basic plot of survfit object

each treatment-age combo, we have a bunch: one for each of ten different times, in fact. By using `summary` on the `survfit` object, we see the four "survival curves" in columns, one for each of the four *rows* of `dance.new`. Above the `summary` of `s`, I've "transposed" the `dance.new` data frame, so you can see what those four columns refer to. There is undoubtedly a slicker way of doing this, but this'll do for now.

This all isn't especially intuitive, so let's see whether we can plot the survival curves. The obvious idea looks like Figure 14.4. It has four survival curves all right, but we can't tell them apart! So we have to try again. Plotting a `survfit` object allows you to tinker with the line type and colour of the survival curves, so we'll take advantage of that to get a better plot.

The process is shown in Figure 14.5. First we remind ourselves of how we made the `dance.new` data frame, since we're going to mimic that for the colours and line types. My plan is to use colours to distinguish the treatments, and line types to distinguish the ages. So first I set up the colours: red is going to be for no-treatment, and blue for treatment. A dashed line is going to be for age

```
> dance.new=expand.grid(Treatment=c(0,1),Age=c(20,40))
> colours=c("red","blue")
> line.types=c("dashed","solid")
> draw.new=expand.grid(colour=colours,linetype=line.types,stringsAsFactors=F)
> cbind(dance.new,draw.new)

  Treatment Age colour linetype
1         0  20    red   dashed
2         1  20   blue   dashed
3         0  40    red    solid
4         1  40   blue    solid

> plot(s,col=draw.new$colour,lty=draw.new$linetype)
> legend(x=10.1,y=1.04,legend=c(20,40),title="Age",lty=line.types)
> legend(x=10.1,y=0.83,legend=0:1,title="Treatment",fill=colours)
```

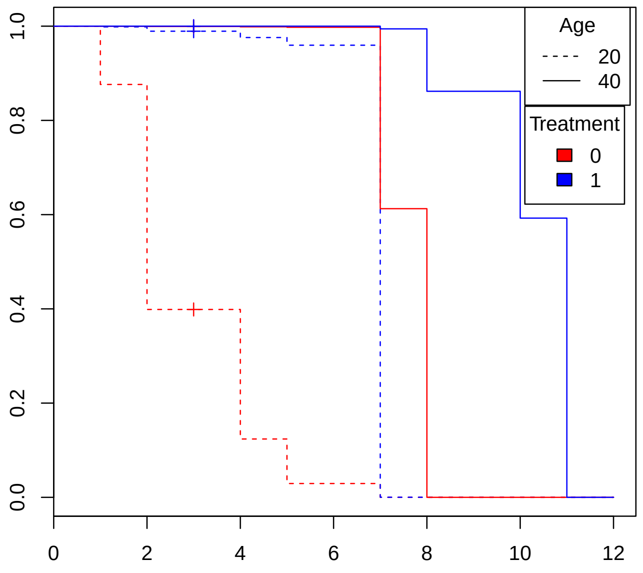
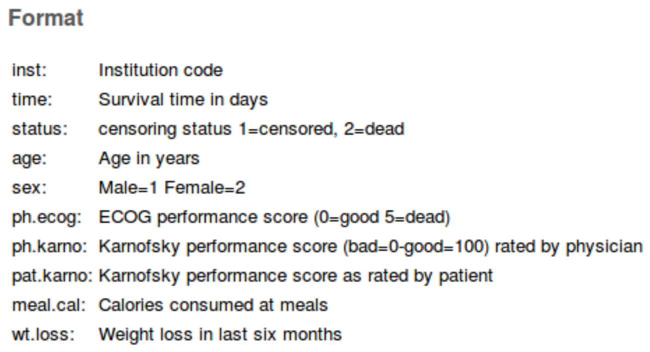


Figure 14.5: Better plot of survfit object



Format

inst:	Institution code
time:	Survival time in days
status:	censoring status 1=censored, 2=dead
age:	Age in years
sex:	Male=1 Female=2
ph.ecog:	ECOG performance score (0=good 5=dead)
ph.karno:	Karnofsky performance score (bad=0-good=100) rated by physician
pat.karno:	Karnofsky performance score as rated by patient
meal.cal:	Calories consumed at meals
wt.loss:	Weight loss in last six months

Figure 14.6: Lung cancer data

20, and a solid line for age 40. Now I set up another `expand.grid` to mimic the one that I used to make `dance.new`: colour where I had treatment, line type where I had age. Note that `expand.grid`, like `read.csv`, turns text strings into factors by default, and we don't want that here, so I turn it off. (The line types and colours have to be text strings when we eventually feed them into `plot`.) Then I list out `dance.new` and `draw.new` side by side to make sure I have the correspondence right. I do.

Finally, I actually draw the plot, setting `col` to the colours I carefully set up and `lty` to the line types. Breathing a large sigh of relief (and yes, I didn't get this right the first time either), we see the plot at the bottom of Figure 14.5. I added two legends, one for treatment and one for age, so that it is clear what the line types and colours mean. An alternative way to do the treatment (colour) legend is

```
legend(x=10.5,y=0.8,legend=0:1,title="Treatment",pch="+",col=colours)
```

which displays the different coloured crosses. I explain at the end of Section 14.3 how the `x` and `y` arguments to `legend` compare with something like `topright`.

As time (the horizontal scale) passes, the predicted survival probability drops from 1 (top left) towards zero (bottom right). A survival curve that stays higher longer is better, so treatment is better than not (the blue curves are to the right of the same-line-type red ones) and being older is better than being younger (the solid curves are to the right of the same-colour dashed ones). This plot is much the easiest way to see what has what kind of effect on survival.

14.3 Example 2: lung cancer

Here is a rather more realistic example of survival data. When you load in an R package, you also tend to get data sets to illustrate the functions in the package. One such is `lung`. This is a data set measuring survival in patients with advanced lung cancer. Along with survival, a number of "performance scores" are included, which measure how well patients can perform daily activities. Sometimes a high score is good, but sometimes it is bad! The list of variables is shown in Figure 14.6, which comes directly from the help file for the data set (just type `?lung`).

Figure 14.7 shows some of the data, the creation of our response variable (`status` 2 is "dead", the event of interest) and the result of fitting a proportional hazards model to all the variables (except `inst`, which I think I forgot). The summary shows that something is definitely predicting survival time (the three tests at the bottom of the output). As to what is, `sex` and `ph.ecog` definitely are, `age`, `pat.karno` and `meal.cal` are definitely not, and the others are in between. So let's take out the three variables that are definitely not significant, and try again. (I'm skipping some steps that you would definitely want to carry out here, like checking residuals, proportionality of hazards and so on.)

Our second model is shown in Figure 14.8. The summary shows that those variables we weren't sure about are not significant after all, so they can come out too. This leaves only `sex` and `ph.ecog`, which are both strongly significant, as shown in Figure 14.9.

We seem to have reached a good model. So now we can make a plot of the predicted survival curves. The last line shows that even though `ph.ecog` scores can theoretically go up to 5, they only went up to 3 in our data set (and there was only one person at that). So we need four colours to represent the possible `ph.ecog` scores, and two line types to represent sexes. (The line types can be hard to distinguish, so I'd rather use four different colours than four different line types.)

The process is shown in Figure 14.10. There is a lot to it, but we can take it one step at a time. We need four colours to represent `ph.ecog`, so we'll set them up as blue, red, green and black. Also, we need two line types for `sex`, so we'll make them solid and dashed. Now we use `expand.grid` twice, once to set up the sexes and `ph.ecog` values for prediction, and then something exactly the same but with `line.types` for `sex` and `colours` for `ph.ecog`. Then I list them out side by side for checking. Everything matches up where it should.

Now I run `survfit` to get the predicted survival curves, using our `lung.new` data frame to predict from. Then I plot it, using our carefully organized colours and line types. The last two steps are to make legends explicating the colours and line types. You can use something like `topright` to position the legend,

```
> head(lung)

  inst time status age sex ph.ecog ph.karno pat.karno meal.cal wt.loss
1    3  306      2  74   1       1       90       100     1175      NA
2    3  455      2  68   1       0       90        90     1225      15
3    3 1010      1  56   1       0       90        90       NA      15
4    5  210      2  57   1       1       90        60     1150      11
5    1  883      2  60   1       0      100        90       NA       0
6   12 1022      1  74   1       1       50        80      513       0

> attach(lung)
> resp=Surv(time,status==2)
> lung.1=coxph(resp~age+sex+ph.ecog+ph.karno+pat.karno+meal.cal+wt.loss)
> summary(lung.1)

Call:
coxph(formula = resp ~ age + sex + ph.ecog + ph.karno + pat.karno +
    meal.cal + wt.loss)

  n= 168, number of events= 121
   (60 observations deleted due to missingness)

                coef  exp(coef)   se(coef)      z Pr(>|z|)
age        1.065e-02  1.011e+00  1.161e-02  0.917  0.35906
sex       -5.509e-01  5.765e-01  2.008e-01 -2.743  0.00609 **
ph.ecog    7.342e-01  2.084e+00  2.233e-01  3.288  0.00101 **
ph.karno   2.246e-02  1.023e+00  1.124e-02  1.998  0.04574 *
pat.karno -1.242e-02  9.877e-01  8.054e-03 -1.542  0.12316
meal.cal   3.329e-05  1.000e+00  2.595e-04  0.128  0.89791
wt.loss   -1.433e-02  9.858e-01  7.771e-03 -1.844  0.06518 .
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

          exp(coef) exp(-coef) lower .95 upper .95
age          1.0107     0.9894    0.9880    1.0340
sex          0.5765     1.7347    0.3889    0.8545
ph.ecog      2.0838     0.4799    1.3452    3.2277
ph.karno     1.0227     0.9778    1.0004    1.0455
pat.karno    0.9877     1.0125    0.9722    1.0034
meal.cal     1.0000     1.0000    0.9995    1.0005
wt.loss      0.9858     1.0144    0.9709    1.0009

Concordance= 0.651  (se = 0.031 )
Rsquare= 0.155   (max possible= 0.998 )
Likelihood ratio test= 28.33  on 7 df,   p=0.0001918
Wald test            = 27.58  on 7 df,   p=0.0002616
Score (logrank) test = 28.41  on 7 df,   p=0.0001849
```

Figure 14.7: The data and model 1

```
> lung.2=coxph(resp~sex+ph.ecog+ph.karno+wt.loss)
> summary(lung.2)

Call:
coxph(formula = resp ~ sex + ph.ecog + ph.karno + wt.loss)

  n= 213, number of events= 151
   (15 observations deleted due to missingness)

              coef exp(coef)  se(coef)      z Pr(>|z|)
sex      -0.623193  0.536229  0.176389 -3.533 0.000411 ***
ph.ecog   0.753428  2.124270  0.194168  3.880 0.000104 ***
ph.karno  0.013814  1.013910  0.009908  1.394 0.163241
wt.loss  -0.009008  0.991032  0.006511 -1.383 0.166516
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

         exp(coef) exp(-coef) lower .95 upper .95
sex         0.5362     1.8649    0.3795    0.7577
ph.ecog     2.1243     0.4707    1.4519    3.1080
ph.karno    1.0139     0.9863    0.9944    1.0338
wt.loss     0.9910     1.0090    0.9785    1.0038

Concordance= 0.64  (se = 0.027 )
Rsquare= 0.136   (max possible= 0.998 )
Likelihood ratio test= 31.06  on 4 df,   p=2.97e-06
Wald test            = 30.6  on 4 df,   p=3.701e-06
Score (logrank) test = 30.99  on 4 df,   p=3.083e-06
```

Figure 14.8: Model 2

```
> lung.3=coxph(resp~sex+ph.ecog)
> summary(lung.3)

Call:
coxph(formula = resp ~ sex + ph.ecog)

  n= 227, number of events= 164
   (1 observation deleted due to missingness)

           coef exp(coef) se(coef)      z Pr(>|z|)
sex     -0.5530    0.5752   0.1676 -3.300 0.000967 ***
ph.ecog  0.4875    1.6282   0.1122  4.344  1.4e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

        exp(coef) exp(-coef) lower .95 upper .95
sex        0.5752     1.7384    0.4142    0.7989
ph.ecog    1.6282     0.6142    1.3067    2.0288

Concordance= 0.642  (se = 0.026 )
Rsquare= 0.12   (max possible= 0.999 )
Likelihood ratio test= 29.05  on 2 df,   p=4.91e-07
Wald test            = 28.96  on 2 df,   p=5.145e-07
Score (logrank) test = 29.41  on 2 df,   p=4.104e-07

> table(ph.ecog)

ph.ecog
  0   1   2   3
 63 113  50   1
```

Figure 14.9: Model 3

```
> colours=c("blue","red","green","black")
> line.types=c("solid","dashed")
> lung.new=expand.grid(sex=c(1,2),ph.ecog=0:3)
> plot.new=expand.grid(lty=line.types,col=colours,stringsAsFactors=F)
> cbind(lung.new,plot.new)

  sex ph.ecog    lty   col
1   1       0  solid  blue
2   2       0 dashed  blue
3   1       1  solid   red
4   2       1 dashed   red
5   1       2  solid green
6   2       2 dashed green
7   1       3  solid black
8   2       3 dashed black

> s=survfit(lung.3,newdata=lung.new)
> plot(s,col=plot.new$col,lty=plot.new$lty)
> legend(x=900,y=0.8,legend=0:3,fill=colours,title="ph.ecog")
> legend(x=900,y=1,legend=1:2,lty=line.types,title="Sex")
```

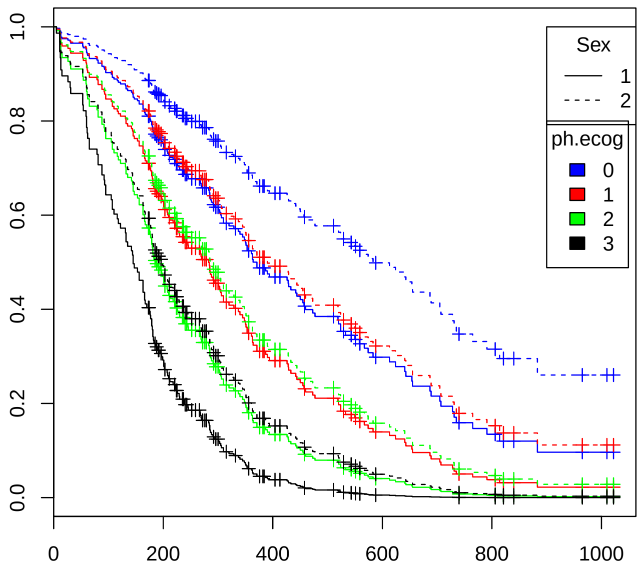


Figure 14.10: Plotting the survival curves

or you can use `x` and `y` coordinates on the plot instead. Since I have two legends, I'm using coordinates so that they don't overwrite each other. I had to experiment a bit to get the locations right. After the coordinates for where to place the legend go the quantities to put in the legend (the `sex` one could be `c("Male","Female")` instead of `1:2`), then what symbols to put in the legend, `fill` for coloured blobs, `lty` for line types, and finally a title to distinguish the legends from each other.

Another way to do a legend is to list out all eight combinations of colour and line type. But I haven't figured that out yet.

Figure 14.10 is the typical look of a set of survival curves. In this case, you can compare the two curves of the same colour to see that females (dashed) have better survival than males (solid), and comparing curves of the same type you can see that predicted survival gets progressively worse as `ph.ecog` score increases. This is not surprising since a lower score is better.

Something curious: the difference between males and females is almost exactly equivalent in survival to one point on the `ph.ecog` scale. For instance, the females with `ph.ecog` score of 2 (green dashed) are almost equivalent in survival to males with an `ph.ecog` score of 1 (red solid). I think this is because the coefficients in Figure 14.9 for `sex` and `ph.ecog` are almost equal in size.

Chapter 15

Multivariate ANOVA and repeated measures

15.1 Multivariate analysis of variance

The ANOVA models we have seen so far have one key thing in common: they have a *single* response variable. Consider, however, the data shown in Figure 15.1. Both seed `yield` and seed `weight` are responses to the amount of `fertilizer` applied to each of the 8 plants.

One thing we can try is to do one ANOVA for each response, as shown in Figure 15.2. In our case we find that neither variable is significantly different between the two fertilizer types.

But this is not the end of the story. If we plot yield against weight, with the points labelled according to whether they were high or low fertilizer, something interesting appears. See Figure 15.3. The high-fertilizer plants, in black, are all at the top right, while the low-fertilizer plants, in red, are relatively speaking at the bottom left. This seems like a difference between high and low fertilizer that should be detectable by a test.

If you just look at `yield`, the highest yield (38) is a high-fertilizer and the lowest two yields (32 and 29) are low-fertilizer, but the ones in the middle are mixed up. Likewise, if you look just at weight, the lowest two (10 and 11) are low fertilizer, but the ones in the middle are mixed up. So the problem is that looking at either variable individually doesn't tell the whole story. You have to look at them *in combination* to explain what's going on. The story of the plot seems to be "high `yield` and high `weight` *both* go with high fertilizer".

```
> hilo=read.table("manova1.txt",header=T)
> hilo

  fertilizer yield weight
1        low    34     10
2        low    29     14
3        low    35     11
4        low    32     13
5       high    33     14
6       high    38     12
7       high    34     13
8       high    35     14
```

Figure 15.1: Data for MANOVA

```
> attach(hilo)
> hilo.y=lm(yield~fertilizer)
> anova(hilo.y)

Analysis of Variance Table

Response: yield
           Df Sum Sq Mean Sq F value Pr(>F)
fertilizer  1   12.5 12.5000  2.1429 0.1936
Residuals   6   35.0  5.8333

> hilo.w=lm(weight~fertilizer)
> anova(hilo.w)

Analysis of Variance Table

Response: weight
           Df Sum Sq Mean Sq F value Pr(>F)
fertilizer  1  3.125   3.125  1.4706 0.2708
Residuals   6 12.750   2.125
```

Figure 15.2: ANOVAs for seed yield and weight data

```
> plot(yield,weight,col=fertilizer)
> lv=levels(fertilizer)
> legend("topright",legend=lv,fill=c(1,2))
```

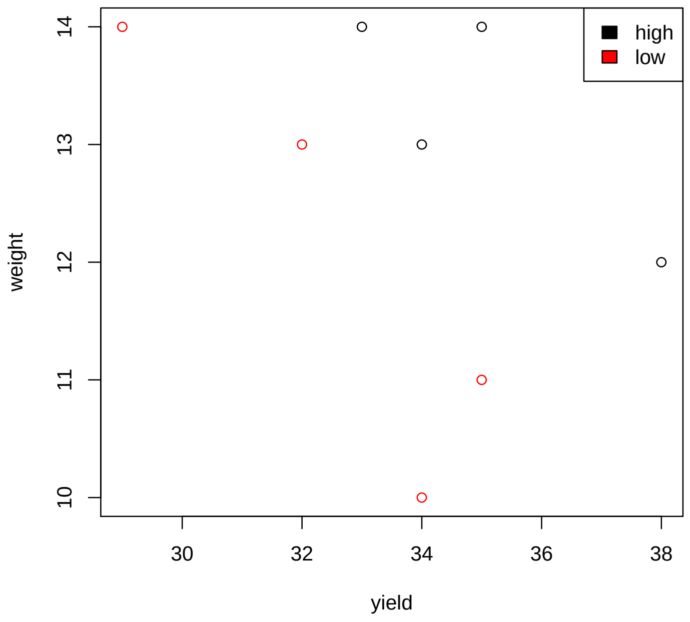


Figure 15.3: Plot of `yield` and `weight`, labelled by `fertilizer`

```
> response=cbind(yield,weight)
> hilo.1=manova(response~fertilizer)
> summary(hilo.1)

           Df  Pillai approx F num Df den Df  Pr(>F)
fertilizer  1 0.80154   10.097      2      5 0.01755 *
Residuals   6
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 15.4: MANOVA of yield and weight data

```
> library(car)
> hilo.2.lm=lm(response~fertilizer)
> hilo.2=Manova(hilo.2.lm)
> hilo.2

Type II MANOVA Tests: Pillai test statistic
           Df test stat approx F num Df den Df  Pr(>F)
fertilizer  1   0.80154   10.097      2      5 0.01755 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 15.5: MANOVA another way

The appropriate analysis is shown in Figure 15.4. This is a multivariate analysis of variance or MANOVA, and it asks "does fertilizer amount influence either response singly, *or a combination of them*?" Since the P-value is small, the answer is "yes", and the explanation is what we described as "the story of the plot" above.

A significant "Pillai" value, as in the example above, means the same kind of thing as the F test in an ordinary ANOVA: the factor (here `fertilizer`) has some kind of effect on the responses.

Another way to do the MANOVA is to use the capital-M `Manova` function from the `car` package. This is shown in Figure 15.5. There's a little more setup involved here: first you run your multivariate response through `lm`, and then you feed *that* fitted model object into `Manova`. Just listing the result gives you the same thing as `summary` does in Figure 15.4. You could also run `summary(hilo.2)`, which gives you more detail, in this case four different tests with identical P-values. (I'm showing you this variant because that's what we'll use for repeated measures later.)

Now, to figure out *what* kind of effect fertilizer has on the responses, we no

```
> peanuts=read.table("peanuts.txt",header=T)
> attach(peanuts)
> loc.fac=factor(location)
> var.fac=factor(variety)
> response=cbind(y,smk,w)
> peanuts.1=lm(response~loc.fac*var.fac)
> peanuts.2=Manova(peanuts.1)
> peanuts.2

Type II MANOVA Tests: Pillai test statistic
                Df test stat approx F num Df den Df   Pr(>F)
loc.fac          1   0.89348  11.1843      3      4 0.020502 *
var.fac          2   1.70911   9.7924      6     10 0.001056 **
loc.fac:var.fac  2   1.29086   3.0339      6     10 0.058708 .
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 15.6: MANOVA of peanut data

longer have anything like Tukey. With two responses, we were able to draw a plot (labelling the points by group), which indicated why the MANOVA was significant. With, say, three responses, we would not be able to draw a graph. A useful technique is discriminant analysis (see Chapter 13). This answers the question of "what combination of responses best separates the groups", thinking about how you might guess which group an observation belongs to based on the values for its response variables. (Yes, this is kind of backwards.)

A multivariate analysis of variance can have any number of response variables, which might be (and probably are, in practice) correlated with each other. On the right of the squiggle can be any combination of main effects and interactions, as you would find in any ANOVA. You get a test for each main effect and interaction on the right.

As an example of that, consider a study of peanuts. Three different varieties of peanuts (labelled, mysteriously, 5, 6 and 8) were planted in two different locations. Three response variables were measured, which we'll call `y`, `smk` and `w`. The analysis is shown in Figure 15.6.

After reading in the data, we note that `location` and `variety` are both numbers, so we create the variables `loc.fac` and `var.fac` which are those numbers as factors (so that R doesn't think those numbers are meaningful as numbers). Then we create our response matrix by gluing together the three columns of response variables. I'm using `Manova` this time (to warm us up for Section 15.2), so I do the two-step process, first making an `lm` which actually contains our model (with location and variety, and their interaction, as factors on the right, and the combined response on the left). Then we get the multivariate tests we want by running `Manova`.

The MANOVA indicates that the interaction is not quite significant (P-value 0.058), but the main effects both are. The next step might be to fit a model without the interaction, or, since we have only 12 observations, to collect some more data. Anyway, we might think of locations as a blocking variable, so that the significance of `fac.loc` is not a surprise. So then the principal interest is in how the varieties are different. One simple idea is the individual ANOVAs of the three responses, although as we saw with the seed yield and weight example, this can be too simple-minded (when what makes a difference is a *combination* of the responses). Nonetheless, these come almost for free from the MANOVA, so it would be silly not to look at them (Figure 15.7). Bearing in mind that we are looking for variety differences, all three responses seem to have something to say, with `smk` and `w` having the smallest P-values. (After we adjust, say by Bonferroni or Holm, for having looked at three P-values at once, the largest one, 0.041, might not even be significant.) The apparently significant interactions are a mild concern also, but I'm going to ignore that.

15.2 Repeated measures

Another kind of experiment that can be handled by MANOVA is the "repeated measures" design. This happens when all your response variables are the same thing measured under different conditions. Typically the different conditions are different times. This requires slightly different handling than a regular MANOVA.

Let's exemplify. Imagine you have three groups of people: some politicians, some administrators and some bellydancers (yes, really). They each try a number of different activities, and rate each one out of 10. The question is, do people with different jobs tend to prefer different activities? The data are shown in Figure 15.8.

Figure 15.9 shows the setup. We need to use `Manova` for this, so the first things to do are to set up the response matrix by gluing together the four responses (activity scores). Then we run `lm`, using our combined response as the response, and the job as explanatory (it is already a factor).

Now we have to tell `Manova` that those four responses are all activity scores. We'll use the generic name `activity` for this (often `time` is what links the responses together, if they are measurements taken at different times). First, we have to make a list of the activities, which is often most easily done by pulling off the column names of the response matrix (which is what `colnames` does). Also, we need to make a data frame of the activities, rather as you would do for `predict`.

```
> summary.aov(peanuts.1)

 Response y :
                Df  Sum Sq Mean Sq F value  Pr(>F)
loc.fac          1   0.701   0.701  0.0404 0.84743
var.fac          2 196.115  98.058  5.6460 0.04177 *
loc.fac:var.fac  2 205.102 102.551  5.9048 0.03824 *
Residuals        6 104.205  17.367
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

 Response smk :
                Df  Sum Sq Mean Sq F value  Pr(>F)
loc.fac          1  162.07  162.07  2.7617 0.14761
var.fac          2 1089.02  544.51  9.2786 0.01459 *
loc.fac:var.fac  2  780.69  390.35  6.6517 0.03003 *
Residuals        6  352.11   58.68
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

 Response w :
                Df  Sum Sq Mean Sq F value  Pr(>F)
loc.fac          1  72.521  72.521  4.5882 0.07594 .
var.fac          2 284.102 142.051  8.9872 0.01567 *
loc.fac:var.fac  2  85.952  42.976  2.7190 0.14435
Residuals        6  94.835  15.806
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> smk.aov=aov(smk~loc.fac*var.fac)
> smk.tukey=TukeyHSD(smk.aov,"var.fac")
> smk.tukey

  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov.default(formula = smk ~ loc.fac * var.fac)

$var.fac
      diff         lwr      upr     p adj
6-5 17.325   0.7046688 33.94533 0.0426394
8-5 22.200   5.5796688 38.82033 0.0150114
8-6  4.875 -11.7453312 21.49533 0.6599956
```

Figure 15.7: Individual ANOVAs of peanut data

```
> active=read.table("profile.txt",header=T)
> active

           job reading dance tv ski
1  bellydancer       7    10  6   5
2  bellydancer       8     9  5   7
3  bellydancer       5    10  5   8
4  bellydancer       6    10  6   8
5  bellydancer       7     8  7   9
6   politician       4     4  4   4
7   politician       6     4  5   3
8   politician       5     5  5   6
9   politician       6     6  6   7
10  politician       4     5  6   5
11       admin       3     1  1   2
12       admin       5     3  1   5
13       admin       4     2  2   5
14       admin       7     1  2   4
15       admin       6     3  3   3
```

Figure 15.8: Activities data

```
> attach(active)
> response=cbind(reading,dance,tv,ski)
> active.1=lm(response~job)
> activity=colnames(response)
> activity.df=data.frame(activity)
```

Figure 15.9: Setting up the repeated measures analysis

```
> active.2=Manova(active.1,idata=activity.df,idesign=~activity)
> active.2

Type II Repeated Measures MANOVA Tests: Pillai test statistic
             Df test stat approx F num Df den Df    Pr(>F)
(Intercept)   1   0.98545   812.60      1     12 2.156e-12 ***
job           2   0.88035    44.14      2     12 2.935e-06 ***
activity      1   0.72086     8.61      3     10  0.004017 **
job:activity  2   1.43341     9.28      6     22 4.035e-05 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 15.10: Repeated measures by MANOVA for the activities data

```
           job reading dance tv ski
1  bellydancer       7    10  6   5
6   politician       4     7  3   2
11       admin       3     6  2   1
```

Figure 15.11: Interaction not significant

Now to put that preparation to work. See Figure 15.10. When you have a repeated measures experiment, `Manova` expects two extra things to be fed to it. The first is the data frame of response variable names, here `activity.df`, which goes with `idata=`. The second, `idesign`, specifies the “within-subject design”. We don’t have anything complicated beyond `activity`, so what you do is to specify a “one-sided model formula”, which is a squiggle followed by the variable containing the list of repeated-measures variables (here `activity`). Don’t forget the squiggle, or else you and R will get very confused.

The output is rather more complicated than before, because not only do we have a test for `job` (that is, whether the activity scores, as a foursome, depend on job), but also a test for `activity` and an interaction between `activity` and `job`. The test for `activity` is asking whether there is a trend over activities, regardless of job, and the interaction is asking whether that trend is different for the different jobs. As ever, since the interaction is significant, that’s where we stop. There is definitely a trend of scores over activities, and that trend is different for each job.

If you go back and look at the original data in Figure 15.8, you’ll see that this hardly comes as a surprise. The bellydancers love dancing, more than TV-watching, while the people in the other jobs like dancing less, and about the same as TV-watching. The administrators don’t care for anything much, except perhaps reading. Notice that the three jobs have about the same attitudes towards reading, but are very different on everything else.

These test results are rather confusing, so let me see if I can make up some data where things are not significant, so we can see why.

Figure 15.11 shows what would happen if the `job:activity` interaction were not significant. Although the scores differ by jobs (so you would expect to see a `job` effect), the *pattern* is always the same: dancing is highest and skiing is lowest. Also, the politicians are consistently 3 points lower than the bellydancers (and the administrators one point lower than that), regardless of activity.

Figure 15.12 shows what would happen if there were no `activity` effect. The scores on all the activities are the same for each activity (if not for each job, so there is still a job effect).

Lastly, Figure 15.13 shows what would happen if `job` were not significant (but

```
           job reading dance tv ski
1  bellydancer       7     7  7   7
6   politician       4     4  4   4
11       admin       2     2  2   2
```

Figure 15.12: Activity not significant

```
           job reading dance tv ski
1  bellydancer       7     8  5   3
6   politician       7     8  5   3
11       admin       7     8  5   3
```

Figure 15.13: Job not significant

`activity` still is significant). Each job has the same score on a particular activity, but the activities differ one from another.

Let's have a look at another example. 8 dogs are randomized to one of two different drugs. The response variables are the log of the blood histamine concentration of the dogs 0, 1, 3 and 5 minutes after being given the drug. Does the drug affect the histamine concentration, and how does this effect play out over time?

Figure 15.14 shows the data. We have four measurements of the same thing at different times for each dog, so this is genuine repeated measures over time. So we'll tackle this via MANOVA as in the last example. See figure 15.15.

The process is: create a response variable matrix by gluing together the (here four) response variables. Then run that through `lm`, the four-variable response possibly depending on drug. Then we create the stuff for the `Manova` that describes the repeated measures: make a list of the four responses and call them collectively `time`. Make that into a data frame. Feed the `lm` model object into `Manova`, along with our data frame of variable names (in `idata=`) and our "within-subjects design" `idesign=`, not forgetting the squiggle since this is a one-sided model formula.

The results are at the bottom of Figure 15.15. The drug by time interaction is significant, so the effect of the drug on the histamine level depends on time, and vice versa. We should try to understand the interaction before we think about analyzing the main effects.

The nicest way to understand an interaction is via `interaction.plot`. Unfortunately, that requires data in "long" format (with one observation per line, and thus 32 lines), but we have "wide" format: as in Figure 15.14, there are only 8 lines but four data values per line.

```
> dogs=read.table("dogs.txt",header=T)
> dogs

          drug x   lh0   lh1   lh3   lh5
1     Morphine N -3.22 -1.61 -2.30 -2.53
2     Morphine N -3.91 -2.81 -3.91 -3.91
3     Morphine N -2.66  0.34 -0.73 -1.43
4     Morphine N -1.77 -0.56 -1.05 -1.43
5 Trimethaphan N -3.51 -0.48 -1.17 -1.51
6 Trimethaphan N -3.51  0.05 -0.31 -0.51
7 Trimethaphan N -2.66 -0.19  0.07 -0.22
8 Trimethaphan N -2.41  1.14  0.72  0.21

> attach(dogs)
```

Figure 15.14: Dog histamine data

```
> response=cbind(lh0,lh1,lh3,lh5)
> dogs.lm=lm(response~drug)
> times=colnames(response)
> times.df=data.frame(times)
> dogs.manova=Manova(dogs.lm,idata=times.df,idesign=~times)
> dogs.manova

Type II Repeated Measures MANOVA Tests: Pillai test statistic
            Df test stat approx F num Df den Df   Pr(>F)
(Intercept)  1   0.76347  19.3664      1      6 0.004565 **
drug         1   0.34263   3.1272      1      6 0.127406
times        1   0.94988  25.2690      3      4 0.004631 **
drug:times   1   0.89476  11.3362      3      4 0.020023 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 15.15: MANOVA analysis of dog data

```
> detach(dogs)
> d2=reshape(dogs,varying=c("lh0","lh1","lh3","lh5"),sep="",direction="long")
> head(d2,n=12)

            drug x time    lh id
1.0     Morphine N    0 -3.22  1
2.0     Morphine N    0 -3.91  2
3.0     Morphine N    0 -2.66  3
4.0     Morphine N    0 -1.77  4
5.0 Trimethaphan N    0 -3.51  5
6.0 Trimethaphan N    0 -3.51  6
7.0 Trimethaphan N    0 -2.66  7
8.0 Trimethaphan N    0 -2.41  8
1.1     Morphine N    1 -1.61  1
2.1     Morphine N    1 -2.81  2
3.1     Morphine N    1  0.34  3
4.1     Morphine N    1 -0.56  4
```

Figure 15.16: Reshape on dogs data

There is, fortunately, a function called `reshape` that will turn data from one format into the other. The process is shown in Figure 15.16. `reshape` needs three (or more) things: a data frame, a list of variables that are repeated measurements of something, and which shape of data you'd like to get.

First, we detach the `dogs` data frame, since we're going to be creating a new one and `attach`ing it, and we don't want to get variable names confused. Then comes the actual `reshape`: first the data frame, then `varying` is a list of variables that are measurements of the same thing under different circumstances (here, the log-histamine measurements at the four different times) that are going to be combined into one. Last comes `direction`, which is the shape of data frame you want to get: we want "long", since we had "wide". Now, if your repeated measurements have the right kind of names, this is almost all you need. Our responses are `lh` followed by a number (the number representing the time), separated by nothing. Thus, if we tell R that our separator is nothing, by `sep=""`, we'll get the right thing. We might have named our variables `lh.0`, `lh.1` etc., in which case `sep="."` would have been the right thing.

Finally, the first 12 lines of the "long" data frame are shown. See how all the log-histamine values have been collected into one column `lh`, and a new column `time` has been created with the values 0, 1, 3 and 5. We also have a column `id` saying which dog each measurement came from. Thus the line labelled `2.1` is dog number 2 (`id`) who took Morphine, measured at time 1, with a log-histamine of -2.81.

```
> attach(d2)

The following object(s) are masked from 'package:datasets':

    lh

> interaction.plot(time,drug,lh)
```

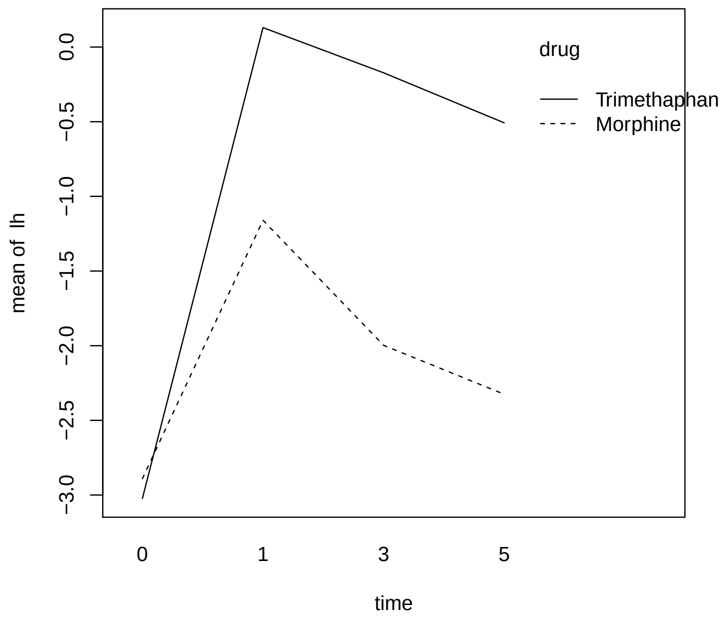


Figure 15.17: Interaction plot for dogs data

```
> detach(d2)
> attach(dogs)
> response=cbind(lh1,lh3,lh5)
> dogs.lm=lm(response~drug)
> times=colnames(response)
> times.df=data.frame(times)
> dogs.manova=Manova(dogs.lm,idata=times.df,idesign=~times)
> dogs.manova

Type II Repeated Measures MANOVA Tests: Pillai test statistic
            Df test stat approx F num Df den Df   Pr(>F)
(Intercept)  1   0.54582   7.2106      1      6 0.036281 *
drug         1   0.44551   4.8207      1      6 0.070527 .
times        1   0.85429  14.6569      2      5 0.008105 **
drug:times   1   0.43553   1.9289      2      5 0.239390
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 15.18: Repeated measures analysis ignoring time 0

This format is ideal for feeding into `interaction.plot`. In fact, I was surprised to discover, after struggling with `reshape`, that getting an interaction plot could now hardly be simpler, as shown in Figure 15.17. I put `time` first, since we are used to seeing time on the horizontal scale. The two traces are for the two different drugs. There is a similar pattern, in that for both drugs. `lh` starts low, increases sharply to time 1, and then decreases slowly after that. But the lines are not parallel, since the increase from time 0 to time 1 is much greater for trimethaphan. That is undoubtedly the reason for the significant interaction.

The next step is blatant “cherrypicking”, but I’m going to do it anyway. Suppose we ignore the time 0 measurements. Then the interaction plot would contain two basically parallel lines, and the significant interaction should go away. We’d also expect to see a significant `drug` effect, and maybe a significant `time` effect as well.

The analysis is presented, without comment, in Figure 15.18. There’s nothing new here. The analysis shows that the interaction is indeed nonsignificant, the time effect is significant, and the drug effect is nearly significant. (There’s a lot of variability between dogs and within drugs, which doesn’t show up in the interaction plot.)

I skimmed over part of `reshape` above, because the `dogs` data frame was in good shape to be turned into long format. Figure 15.19 shows how you might turn the jobs and leisure activities data frame into long format. The new things are: `v.names`, which gives a name for the column to contain the combined repeated

```
> detach(dogs)
> activity.list=c("reading","dance","tv","ski")
> active.long=reshape(active,varying=activity.list,v.names="score",
+   timevar="activity",direction="long")
> head(active.long,n=20)

             job activity score id
1.1  bellydancer        1     7  1
2.1  bellydancer        1     8  2
3.1  bellydancer        1     5  3
4.1  bellydancer        1     6  4
5.1  bellydancer        1     7  5
6.1   politician        1     4  6
7.1   politician        1     6  7
8.1   politician        1     5  8
9.1   politician        1     6  9
10.1  politician        1     4 10
11.1       admin        1     3 11
12.1       admin        1     5 12
13.1       admin        1     4 13
14.1       admin        1     7 14
15.1       admin        1     6 15
1.2  bellydancer        2    10  1
2.2  bellydancer        2     9  2
3.2  bellydancer        2    10  3
4.2  bellydancer        2    10  4
5.2  bellydancer        2     8  5
```

Figure 15.19: Turning jobs data frame into long format

responses (scores, over the different activities) and `timevar`, which is a name for what is being repeated over (this defaults to `time`, but here is `activities`). Now, if you wish, you can make an interaction plot here too. The interaction was significant, so the lines should not be parallel. This is, as they say, left as an exercise for the reader.

Chapter 16

Principal components

Chapter 17

Factor analysis

Chapter 18

Time series

Chapter 19

Spatial data and kriging