4
Graphical Presentation of Data

4.1 Introduction

Graphical displays of data can be very useful for showing the main features of a data set. The appropriate form of graph depends on the nature of the variables being displayed and what aspects are to be shown. However it should always be borne in mind that the object is to provide a clear and truthful representation of the data, not to distort and not to impress with unnecessary "fancy" features.

4.2 Qualitative data: bar charts

The most useful way to display qualitative data is usually with a bar chart. The length of each bar is proportional to the frequency of the corresponding value of the variable in the sample of data. Note that the widths of the bars should be equal to avoid giving a false impression, as should the width *between* bars. [1]

Figure 4.1 shows the breakdown in MPAA ratings for the movie data set. To create a bar chart in R we use the `table` command...

```
> table(movies$mpaa)
NC-17    PG PG-13     R
   16   526   989  3316
```

... inside the `barplot` function:

```
> barplot(table(movies$mpaa), xlab="MPAA Rating",
+         ylab="Frequency", border = "black",
+         col="mistyrose")
```

Remember to load the data first!

[1] Type `colours()` into R to get a list of colours.

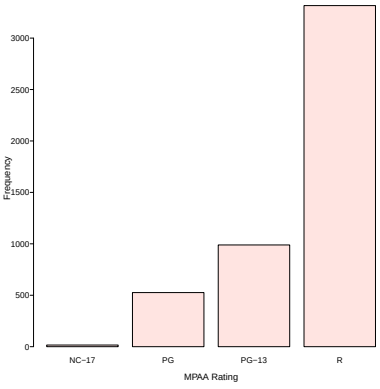

Figure 4.1: Barchart of the mpaa ratings for 4847 films.

4.3 Histograms

To represent the distribution of a sample of values of a continuous variable we can use a histogram. The range of values of the variable is divided into intervals, known as *classes*, and the frequencies in classes are represented by columns. As the variable is continuous, there are no gaps between neighbouring columns, unlike a bar chart.[2] Note also that, strictly speaking, it is the *area* of the column which is proportional to

[2] Unless, of course, a particular class has zero frequency

Rule	Formula	R command	Comment
Sturges'	$k_{ST} = \lceil \log_2 n + 1 \rceil$	Default	Tends not to be very good for $n > 30$.
Scott's	$h_{SC} = 3.49 \times s \times n^{-1/3}$	breaks = "Scott"	
Freedman-Diaconis	$h_{FR} = 2 \times IQR(x) \times n^{-1/3}$	breaks = "FD"	When the distribution is symmetric, this is very similar to Scott's rule.

Table 4.1: Standard bin width rules in R.

the frequency, not the height. The reason for this is that columns need not be of the same width. Computer software tends to use columns of the same width. However, this default can be overridden in R if you really want to do this.

Figure 4.2 shows histograms of the film budgets. When dealing with densities (relative frequency), we can easily work out the height using this formula:

$$\text{Height} = \frac{\text{frequency}}{n \times \text{ Bin-width}}.$$

When the y-axis is labelled with density or relative frequencies, the area under the histogram is one. Bin widths should be chosen so that you get a good idea of the distribution of the data, without being swamped by random variation.

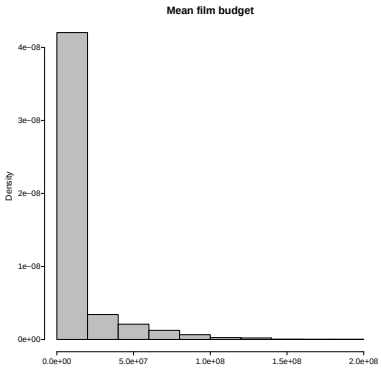

Figure 4.2: Histogram of movie budgets.

To generate Figure 4.2 in R we use the following commands:

```
> hist(movies$Budget, col="grey",
+      main="Mean film budget",
+      freq=FALSE, xlab="Budget ($)")
```

4.3.1 How many bins should we have?

First we will define the notation we will use:

- n: the sample size;
- k: the number of bins in the histogram;
- h: the bin-width.

Then the number of bins we will use to construct a histogram is:

$$k = \left\lceil \frac{\max(x) - \min(x)}{h} \right\rceil \tag{4.1}$$

where $\lceil \cdot \rceil$ is the ceiling function.
Table 4.1 gives a summary of the different rules. Briefly, R uses Sturges' rule by default, which isn't always that good. Notice that Sturges' rule gives you k, but the other rules give you the bin width. Typically, it is best to go with Scotts' rule or the Freedman-Diaconis rule.

To use the different rules, we use the breaks argument. For example, the following piece of code:

```
> hist(movies$Budget, col="wheat",
+      main="The FD rule",
+      freq=FALSE, xlab="Budget ($)", breaks="FD")
```

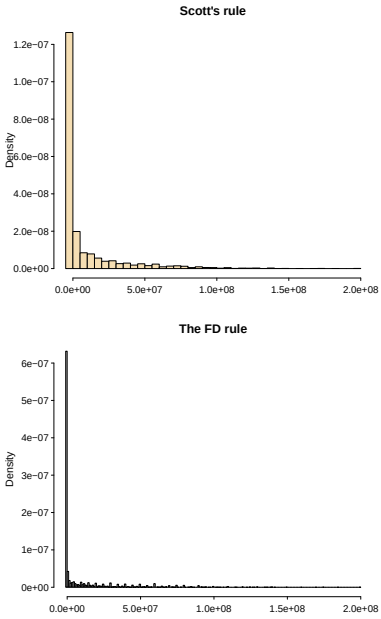

Figure 4.3: Histogram of (a) movie budgets using Scott's rule and (b) the Freedman-Diaconis rule. Compare these histograms to Figure 4.2 which uses Sturges' rule (default).

uses the Freedman-Diaconis rule - see Figure 4.3. When we compare Figure 4.3a with 4.2, we have used many more bins, which results in a *better* histogram; however, it is clear that Figure 4.3b uses too *many* bins! In practical 2, you will get a chance to experiment with the different rules yourself.

4.3.2 Example: movie data

				Number of bins		
Variable	Range	s	IQR	Sturges	Scott's	Free-Dia
Budget	2×10^8	23039711	8×10^6	14	43	212
Length	249.0	17.3	17.0	14	70	124
Rating	8.1	1.5	2.0	14	28	35

Table 4.2: Bin sizes for the movie data. In all cases $n = 4847$.

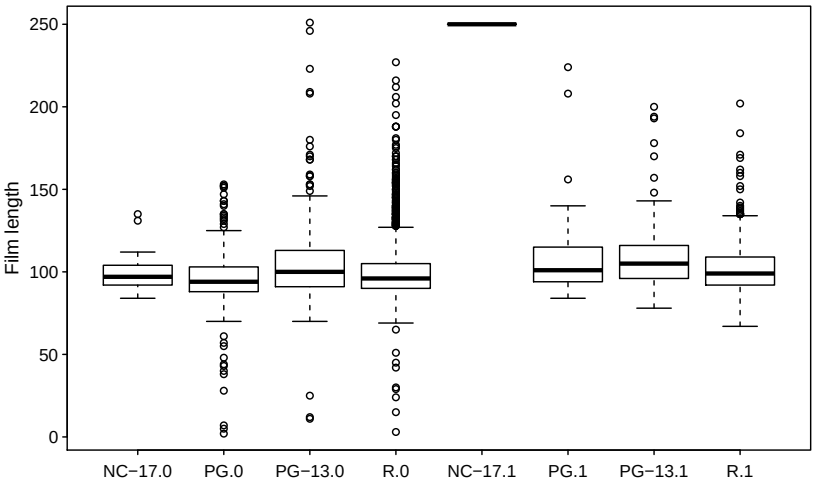

Figure 4.4: Box and whisker plots movie length split according to mpaa rating and whether the film was a romance.

4.4 *Box and whisker plots*

A box and whisker plot, often referred to simply as a boxplot, is another way to represent continuous data. This kind of plot is particularly useful for comparing two or more groups, by placing the boxplots side-by-side. Figure 4.4 and figure 4.5 shows boxplots of film length for different categories of film.

The central bar in the "box" is the sample *median*. The top and bottom of the box represent the upper and lower sample *quartiles*, respectively. Just as the median represents the 50% point of the data, the lower and upper quartiles represent the 25% and 75% points respectively.

The lower whisker is drawn from the lower end of the box to the smallest value that is no smaller than $1.5IQR$ below the lower quartile. Similarly, the upper whisker is drawn from the middle of the upper end of the box to the largest value that is no larger than $1.5IQR$ above the upper quantile. Points outside the whiskers are classified as outliers.

To do this in R we use the following commands:

```
> par(mfrow=c(2, 1))
> boxplot(movies$Length, ylab="Film length",
+         col="bisque")
> boxplot(movies$Length ~ movies$mpaa, ylab="Film length",
+         col="bisque")
> boxplot(movies$Length ~ movies$mpaa + movies$Romance,
+         ylab="Film length")
```

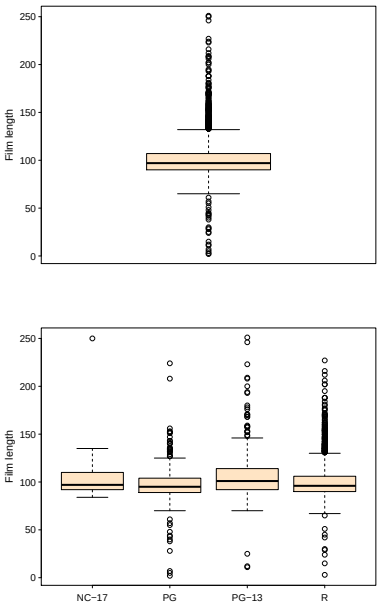

Figure 4.5: Box and whisker plots of (a) film length (b) film length split according to the mpaa rating.

4.4.1 Boxplot example 1

For the data set:

0.1	0.1	0.2	0.4	0.4	0.8	0.8	0.8
0.9	0.9	1.0	1.4	1.6	2.0	2.4	3.5

Table 4.3: An example data set.

construct a boxplot.

Solution
First we calculate the median and quartiles:

Median	1^{st} quartile	3^{rd} quartile	IQR
0.85	0.4	1.55	1.15

Table 4.4: Summary statistics for the first example data in Table 4.3.

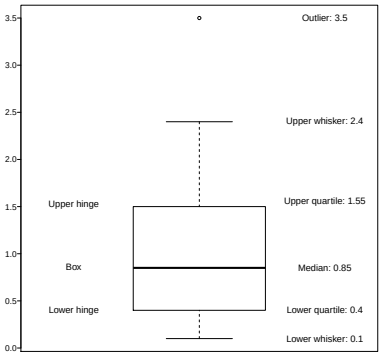

Figure 4.6: Boxplot for the first example data set in table 4.3 and summary statistics in table 4.4.

4.4.2 Boxplot example 2

For the following data:

9.0	32.8	33.0	34.9	35.4	39.7	41.6	42.0
42.3	43.2	46.9	49.2	51.6	51.7	55.0	81.0

Table 4.5: An example data set.

construct a boxplot.

Solution
First we calculate the necessary summary statistics:

Median	1^{st} quartile	3^{rd} quartile	IQR	W_L	W_U
42.1	35.02	51.00	15.98		

Table 4.6: Summary statistics for the second example data in table 4.5.

Command	Comment	Example
`table`	Contingency table	`table(x)`
`barplot`	Generate a bar chart	`barplot(table(x))`
`hist`	Histogram	`hist`
`plot`	Scatter plot. See Practical 2.	`plot(x, y)`
`points`	Add points to a plot. See Practical 2.	`points(x,y)`
`lines`	Add lines to a plot. See Practical 2.	`lines(x,y)`
`boxplot`	Box and whiskers plot	`boxplot(x)`

Table 4.7: Summary of R commands in this chapter.

5
Control Statements and Functions

5.1 Functions

A very powerful aspect of R is that it is relatively easy to write your own functions. Functions can take inputs (or *arguments*) and return a *single* value. Let's look at some simple functions.

5.1.1 Basic functions

This function takes in a single argument x and returns x^2:

```
> Fun1 = function(x) {
+   return (x*x)
+ }
```

The key elements in the function call are:

- The word `function`;
- The brackets () which enclose the **argument** list.
- A sequence of statements in curly braces { }.
- A `return` statement.

We *call* `Fun1` in the following manner:

```
> Fun1(5)
[1] 25
> y = Fun1(10)
> y
[1] 100
> z = c(1, 2, 3, 4)
> Fun1(z)
[1]  1  4  9 16
```

Of course, the old saying 'Garbage in, Garbage out' is true:

```
> Fun1()
Error in Fun1() : argument "x" is missing, with no default
> Fun1("5")
Error in x * x : non-numeric argument to binary operator
```

The error messages give you an idea of what went wrong. Other variations to this simple function are:

```
> Fun2 = function(x=1) {
+   return (x*x)
+ }
> Fun2()
[1] 1
> Fun2(4)
[1] 16
> Fun3 = function(x, y) {
+   return (x*y)
+ }
> Fun3(3, 4)
[1] 12
```

5.1.2 A more useful function

Here the function below takes in a vector, plots a histogram and returns a vector containing the mean and standard deviation:

```
> Investigate = function(values) {
+   hist(values)
+   m_std = c(mean(values), sd(values))
+   return(m_std)
+ }
```

Once we have created our function, we can put it to good use[1]:

```
> Investigate(movies$Rating)
[1] 5.522715 1.451864
```

[1] Obviously, a histogram would also be created – it's just not shown here. See Section 4.3 for examples of histograms.

5.1.3 Variable scope

When we call a function, R first looks for *local* variables, then *global* variables. For example, `Fun4` uses a global variable:

R scoping rules are actually a bit more complicated than described below. R uses something called *lexical* scope, but this doesn't affect us.

```
> blob = 5
> Fun4 = function() {
+   return(blob)
+ }
> Fun4()
[1] 5
```

However, in `Fun5`, we use a local variable:

```
> Fun5 = function() {
+   blob = 6
+   return(blob)
+ }
> Fun5()
[1] 6
> blob
[1] 5
```

As a general rule, functions should only use **local** variables. This makes your code more portable and less likely to have bugs.

5.2 The `cat` command

A useful function to help debugging is the `cat` function. This function is used to print messages to the screen. For example,

```
> x = 5
> cat(x, "\n")
5
> (y = cat(x, "\n"))
5
NULL
```

We will use the `cat` function in the next section.

5.3 Conditionals

Conditional statements are features of a programming language which perform different computations or actions depending on whether a condition evaluates to `TRUE` or `FALSE`. They are used in almost all computer programs.

5.3.1 *If statements*

The basic structure of an `if` statement is:

```
> if(expr) {
+   ##do something
+ }
```

where `expr` is evaluated to be either `TRUE` or `FALSE`. The following example illustrates `if` statements in R:

```
> x = 5
> y = 5
> if(x<5) {
+   y = 0
+ }
> y
[1] 5
```

In this code chunk, `x` $<$ `5` evaluates to be `FALSE` so the following brackets are not evaluated. We test for greater than in a similar manner:

```
> x = 5
> y = 5
> if(x > 0) {
+   y = 0
+ }
> y
[1] 0
```

Here `x` $>$ `0` evaluates to be `TRUE` so, `y` is set equal to 0. If we wanted to test for equality with zero, then we would use `==`.

5.3.2 *If else statements*

We can link together a number of `if` statements

```
> x = 0
> if(x > 0) {
+   cat("x is greater than zero")
+ } else if(x < 0) {
+   cat("x is less than zero")
+ } else {
+   cat("x must be zero!")
+   cat("\n")
+ }
x must be zero!
```

The final `else` is optional. We can also use `if` statements in functions, for example to check that our data is negative we can construct the following function:

```
> IsNegative = function(value) {
+   I = FALSE
+   if(value < 0) {
+     I = TRUE
+   }
+   return(I)
+ }
> IsNegative(1)
[1] FALSE
> IsNegative(-5.6)
[1] TRUE
```

A more sophisticated function could be:

```
> IsGreaterThan = function(value1, value2) {
+   is_greater_than = FALSE
+   if(value1 > value2) {
+     is_greater_than = TRUE
+   }
+   return(is_greater_than)
+ }
```

Which we can then call:

```
> IsGreaterThan(-5, -6)
[1] TRUE
> IsGreaterThan(10, 10)
[1] FALSE
```

5.4 *Control statements*

At times we would like to perform some operation on a vector or a data frame. Often R has built-in functions that will do this for you, e.g. `mean`, `sd`,... Other times we have to write our own functions. For example, suppose we want to calculate $\sum_{i=1}^{10} i^2$.

In R we can use a `for` loop:

```
> x = 0
> for(i in 1:10) {
+   x = x + i^2
+ }
> x
[1] 385
```

Or perhaps $\sum_{j=-5}^{-1} e^j/j^2$, then:

```
> total = 0
> for(j in -5:-1) {
+   total = total + exp(j)/j^2
+ }
> total
[1] 0.4086594
```

A more tricky example: Calculate $\sum e^k/k^2$, for $k = 3, 6, 9, \ldots, 21$:

```
> total = 0
> for(i in 1:7) {
+   k = i*3
+   total = total + exp(k)/k^2
+ }
> total
[1] 3208939
```

Exercise: Using the inbuilt R function `sum`, calculate the above summations without using `for` loops.

5.5 *Putting it all together*

Rather than have to constantly write R code to solve the summations in §5.4 we can create a function to solve the general form:

$$\sum_{i=i_s}^{i_e} \frac{e^i}{i^2} \quad \text{for } i = i_s,\ i_s + j,\ i_s + 2j,\ \ldots, i_e\ .$$

Can you see what's wrong with the function `Summation1`? There's a prize for anyone who can spot this – and write a function which works all the time!

So in R we have:

```
> Summation1 = function(i_s, i_e, j) {
+   total = 0
+   for(i in 1:(i_e/j)) {
+     k = i*j
+     total = total + exp(k)/k^2
+   }
+   return(total)
+ }
> Summation1(3, 21, 3)
[1] 3208939
```

5.6 The *apply family*

R has been designed with manipulating data in mind. Due to this, there are two important functions that are unique to R.

Probably not unique, but not common in other programming languages.

5.6.1 The `apply` function

We use the `apply` function when we want to *apply* the same function to every row or column of a data frame. For example, suppose we have a data frame with three columns:

```
> (df4 = data.frame(c1 = 1:4, c2 = 4:7, c3 = 2:5))
  c1 c2 c3
1  1  4  2
2  2  5  3
3  3  6  4
4  4  7  5
```

The `apply` function takes (at least) three arguments. The first argument is the data frame, the second the number `1` or `2` indicating row or column and the third a function to apply to each row or column. So

```
> apply(df4, 1, mean)
[1] 2.333333 3.333333 4.333333 5.333333
```

calculates the mean value of every row, while

```
> apply(df4, 2, sd)
      c1       c2       c3
1.290994 1.290994 1.290994
```

calculates the standard deviation of every column.
Suppose one of the columns was non-numeric

```
> (df5 = data.frame(c1 = 1:3, c2 = 4:6,
+    c3 = LETTERS[1:3]))
  c1 c2 c3
1  1  4  A
2  2  5  B
3  3  6  C
```

then taking the mean doesn't really make sense:

```
> apply(df5, 1, mean)
[1] NA NA NA
```

Instead, we remove the column, then calculate the mean:

```
> apply(df5[ ,1:2], 1, mean)
[1] 2.5 3.5 4.5
```

5.6.2 The `tapply` function

The function `tapply` is very useful, but at first glance can be tricky to understand. It's best described using an example:

```
> tapply(movies$Length, movies$mpaa, mean)
    NC-17         PG      PG-13          R
110.18750  97.23384 104.97877 100.18818
```

In the above code, we have calculated the average movie length conditional on its MPAA rating. So the average length of a PG movie is 97 minutes and the average NC-17 movie length is 110mins. With `tapply` we can do we very interesting things. For example, in the next piece of code, we plot the average movie length conditional on it's rating:

```
> tapply(movies$Length, movies$Rating, mean)[1:6]
   1  1.2  1.3  1.4  1.5  1.6
85.5 93.0 87.5 85.0 67.0 86.0
> rating_by_len = tapply(movies$Length, movies$Rating,
+   mean)
> plot(names(rating_by_len), rating_by_len)
```

Imagine trying to produce Figure 5.1 in Excel!

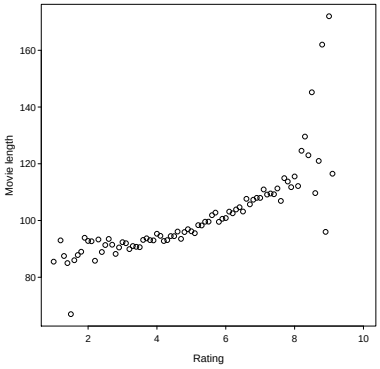

Figure 5.1: Plot of mean movie length conditional on it's rating.

5.7 Help

R has a very good help system. If you need information about a particular function – say `plot` – then typing `?plot` in a R terminal will bring up the associated help page.

The internet is another very good source of R help. Unfortunately, using Google isn't particularly useful since the letter "R" appears on most web pages! However, you can use

http://www.rseek.org/

Using this search engine limits searches to R web-pages.

Summary of R commands

Command	Comment
`for`	A for loop. See §5.4
`if` or `else`	A conditional statement. See §5.3.
`function`	An R function constructor. See §5.1
`cat`	Print command. See §5.2
`apply` or `tapply`	See §5.6

Table 5.1: Summary of R commands in this chapter.