3

Probability and distributions

The concepts of randomness and probability are central to statistics. It is an empirical fact that most experiments and investigations are not perfectly reproducible. The degree of irreproducibility may vary: Some experiments in physics may yield data that are accurate to many decimal places, whereas data on biological systems are typically much less reliable. However, the view of data as something coming from a statistical distribution is vital to understanding statistical methods. In this section, we outline the basic ideas of probability and the functions that R has for random sampling and handling of theoretical distributions.

3.1 Random sampling

Much of the earliest work in probability theory was about games and gambling issues, based on symmetry considerations. The basic notion then is that of a random sample: dealing from a well-shuffled pack of cards or picking numbered balls from a well-stirred urn.

In R, you can simulate these situations with the `sample` function. If you want to pick five numbers at random from the set `1:40`, then you can write

```
> sample(1:40,5)
[1]  4 30 28 40 13
```

P. Dalgaard, *Introductory Statistics with R*,
DOI: 10.1007/978-0-387-79054-1_3, © Springer Science+Business Media, LLC 2008

The first argument (`x`) is a vector of values to be sampled and the second (`size`) is the sample size. Actually, `sample(40,5)` would suffice since a single number is interpreted to represent the length of a sequence of integers.

Notice that the default behaviour of `sample` is *sampling without replacement.* That is, the samples will not contain the same number twice, and `size` obviously cannot be bigger than the length of the vector to be sampled. If you want sampling with replacement, then you need to add the argument `replace=TRUE`.

Sampling with replacement is suitable for modelling coin tosses or throws of a die. So, for instance, to simulate 10 coin tosses we could write

```
> sample(c("H","T"), 10, replace=T)
 [1] "T" "T" "T" "T" "T" "H" "H" "T" "H" "T"
```

In fair coin-tossing, the probability of heads should equal the probability of tails, but the idea of a random event is not restricted to symmetric cases. It could be equally well applied to other cases, such as the successful outcome of a surgical procedure. Hopefully, there would be a better than 50% chance of this. You can simulate data with nonequal probabilities for the outcomes (say, a 90% chance of success) by using the `prob` argument to `sample`, as in

```
> sample(c("succ", "fail"), 10, replace=T, prob=c(0.9, 0.1))
 [1] "succ" "succ" "succ" "succ" "succ" "succ" "succ" "succ"
 [9] "succ" "succ"
```

This may not be the best way to generate such a sample, though. See the later discussion of the binomial distribution.

3.2 Probability calculations and combinatorics

Let us return to the case of sampling without replacement, specifically `sample(1:40, 5)`. The probability of obtaining a given number as the first one of the sample should be $1/40$, the next one $1/39$, and so forth. The probability of a given sample should then be $1/(40 \times 39 \times 38 \times 37 \times 36)$. In R, use the `prod` function, which calculates the product of a vector of numbers

```
> 1/prod(40:36)
[1] 1.266449e-08
```

However, notice that this is the probability of getting given numbers in a given order. If this were a Lotto-like game, then you would rather be interested in the probability of guessing a given *set* of five numbers correctly. Thus you need also to include the cases that give the same numbers in a different order. Since obviously the probability of each such case is going to be the same, all we need to do is to figure out how many such cases there are and multiply by that. There are five possibilities for the first number, and for each of these there are four possibilities for the second, and so forth; that is, the number is $5 \times 4 \times 3 \times 2 \times 1$. This number is also written as 5! (5 *factorial*). So the probability of a "winning Lotto coupon" would be

```
> prod(5:1)/prod(40:36)
[1] 1.519738e-06
```

There is another way of arriving at the same result. Notice that since the actual set of numbers is immaterial, all sets of five numbers must have the same probability. So all we need to do is to calculate the number of ways to choose 5 numbers out of 40. This is denoted

$$\binom{40}{5} = \frac{40!}{5!35!} = 658008$$

In R, the `choose` function can be used to calculate this number, and the probability is thus

```
> 1/choose(40,5)
[1] 1.519738e-06
```

3.3 Discrete distributions

When looking at independent replications of a binary experiment, you would not usually be interested in whether each case is a success or a failure but rather in the total number of successes (or failures). Obviously, this number is random since it depends on the individual random outcomes, and it is consequently called a *random variable*. In this case it is a discrete-valued random variable that can take values $0, 1, \ldots, n$, where n is the number of replications. Continuous random variables are encountered later.

A random variable X has a *probability distribution* that can be described using *point probabilities* $f(x) = P(X = x)$ or the *cumulative distribution function* $F(x) = P(X \leq x)$. In the case at hand, the distribution can be worked out as having the point probabilities

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

This is known as the *binomial distribution*, and the $\binom{n}{x}$ are known as *binomial coefficients*. The parameter p is the probability of a successful outcome in an individual trial. A graph of the point probabilities of the binomial distribution appears in Figure 3.2 ahead.

We delay describing the R functions related to the binomial distribution until we have discussed continuous distributions so that we can present the conventions in a unified manner.

Many other distributions can be derived from simple probability models. For instance, the *geometric distribution* is similar to the binomial distribution but records the number of failures that occur before the first success.

3.4 Continuous distributions

Some data arise from measurements on an essentially continuous scale, for instance temperature, concentrations, etc. In practice, they will be recorded to a finite precision, but it is useful to disregard this in the modelling. Such measurements will usually have a component of random variation, which makes them less than perfectly reproducible. However, these random fluctuations will tend to follow patterns; typically they will cluster around a central value, with large deviations being more rare than smaller ones.

In order to model continuous data, we need to define random variables that can obtain the value of any real number. Because there are infinitely many numbers infinitely close, the probability of any particular value will be zero, so there is no such thing as a point probability as for discrete-valued random variables. Instead we have the concept of a *density*. This is the infinitesimal probability of hitting a small region around x divided by the size of the region. The cumulative distribution function can be defined as before, and we have the relation

$$F(x) = \int_{-\infty}^{x} f(x)\,dx$$

There are a number of standard distributions that come up in statistical theory and are available in R. It makes little sense to describe them in detail here except for a couple of examples.

The *uniform distribution* has a constant density over a specified interval (by default $[0, 1]$).

The *normal distribution* (also known as the *Gaussian distribution*) has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

depending on its mean μ and standard deviation σ. The normal distribution has a characteristic bell shape (Figure 3.1), and modifying μ and σ simply translates and widens the distribution. It is a standard building block in statistical models, where it is commonly used to describe error variation. It also comes up as an approximating distribution in several contexts; for instance, the binomial distribution for large sample sizes can be well approximated by a suitably scaled normal distribution.

3.5 The built-in distributions in R

The standard distributions that turn up in connection with model building and statistical tests have been built into R, and it can therefore completely replace traditional statistical tables. Here we look only at the normal distribution and the binomial distribution, but other distributions follow exactly the same pattern.

Four fundamental items can be calculated for a statistical distribution:

- Density or point probability
- Cumulated probability, distribution function
- Quantiles
- Pseudo-random numbers

For all distributions implemented in R, there is a function for each of the four items listed above. For example, for the normal distribution, these are named `dnorm`, `pnorm`, `qnorm`, and `rnorm` (*d*ensity, *p*robability, *q*uantile, and *r*andom, respectively).

3.5.1 Densities

The density for a continuous distribution is a measure of the relative probability of "getting a value close to x". The probability of getting a value in a particular interval is the area under the corresponding part of the curve.

For discrete distributions, the term "density" is used for the point probability — the probability of getting exactly the value x. Technically, this is correct: It is a density with respect to counting measure.

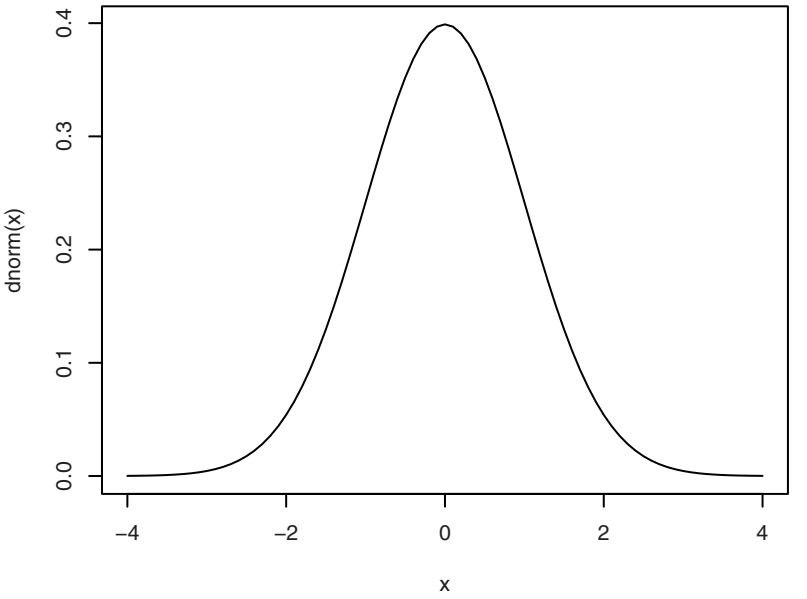

Figure 3.1. Density of normal distribution.

The density function is likely the one of the four function types that is least used in practice, but if for instance it is desired to draw the well-known bell curve of the normal distribution, then it can be done like this:

```
> x <- seq(-4,4,0.1)
> plot(x,dnorm(x),type="l")
```

(Notice that this is the letter 'l', not the digit '1').

The function `seq` (see p. 15) is used to generate equidistant values, here from -4 to 4 in steps of 0.1; that is, $(-4.0, -3.9, -3.8, \ldots, 3.9, 4.0)$. The use of `type="l"` as an argument to `plot` causes the function to draw lines between the points rather than plotting the points themselves.

An alternative way of creating the plot is to use `curve` as follows:

```
> curve(dnorm(x), from=-4, to=4)
```

This is often a more convenient way of making graphs, but it does require that the y-values can be expressed as a simple functional expression in x.

For discrete distributions, where variables can take on only distinct values, it is preferable to draw a pin diagram, here for the binomial distribution with $n = 50$ and $p = 0.33$ (Figure 3.2):

```
> x <- 0:50
> plot(x,dbinom(x,size=50,prob=.33),type="h")
```

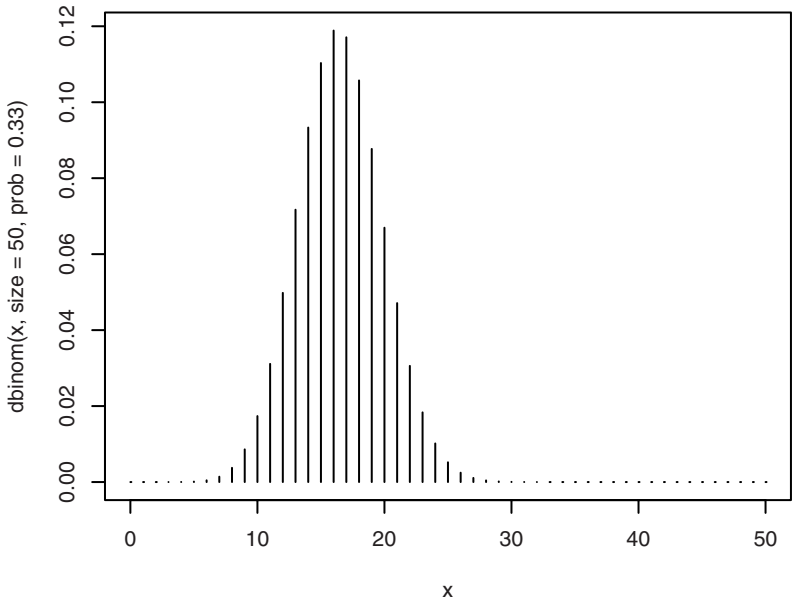

Figure 3.2. Point probabilities in binom(50, 0.33).

Notice that there are three arguments to the "d-function" this time. In addition to x, you have to specify the number of trials n and the probability parameter p. The distribution drawn corresponds to, for example, the number of 5s or 6s in 50 throws of a symmetrical die. Actually, `dnorm` also takes more than one argument, namely the mean and standard deviation, but they have default values of 0 and 1, respectively, since most often it is the standard normal distribution that is requested.

The form `0:50` is a short version of `seq(0,50,1)`: the whole numbers from 0 to 50 (see p. 15). It is `type="h"` (as in *h*istogram-like) that causes the pins to be drawn.

3.5.2 *Cumulative distribution functions*

The cumulative distribution function describes the probability of "hitting" x or less in a given distribution. The corresponding R functions begin with a 'p' (for probability) by convention.

Just as you can plot densities, you can of course also plot cumulative distribution functions, but that is usually not very informative. More often, actual numbers are desired. Say that it is known that some biochemical measure in healthy individuals is well described by a normal distribution with a mean of 132 and a standard deviation of 13. Then, if a patient has a value of 160, there is

```
> 1-pnorm(160,mean=132,sd=13)
[1] 0.01562612
```

or only about 1.5% of the general population, that has that value or higher. The function `pnorm` returns the probability of getting a value smaller than its first argument in a normal distribution with the given mean and standard deviation.

Another typical application occurs in connection with statistical tests. Consider a simple sign test: Twenty patients are given two treatments each (blindly and in randomized order) and then asked whether treatment A or B worked better. It turned out that 16 patients liked A better. The question is then whether this can be taken as sufficient evidence that A actually is the better treatment or whether the outcome might as well have happened by chance even if the treatments were equally good. If there was no difference between the two treatments, then we would expect the number of people favouring treatment A to be binomially distributed with $p = 0.5$ and $n = 20$. How (im)probable would it then be to obtain what we have observed? As in the normal distribution, we need a tail probability, and the immediate guess might be to look at

```
> pbinom(16,size=20,prob=.5)
[1] 0.9987116
```

and subtract it from 1 to get the upper tail — but this would be an error! What we need is the probability of *the observed or more extreme*, and `pbinom` is giving the probability of 16 or less. We need to use "15 or less" instead.

```
> 1-pbinom(15,size=20,prob=.5)
[1] 0.005908966
```

If you want a two-tailed test because you have no prior idea about which treatment is better, then you will have to add the probability of obtaining equally extreme results in the opposite direction. In the present case, that

means the probability that four or fewer people prefer A, giving a total probability of

```
> 1-pbinom(15,20,.5)+pbinom(4,20,.5)
[1] 0.01181793
```

(which is obviously exactly twice the one-tailed probability).

As can be seen from the last command, it is not strictly necessary to use the `size` and `prob` keywords as long as the arguments are given in the right order (positional matching; see Section 1.2.2).

It is quite confusing to keep track of whether or not the observation itself needs to be counted. Fortunately, the function `binom.test` keeps track of such formalities and performs the correct binomial test. This is further discussed in Chapter 8.

3.5.3 *Quantiles*

The quantile function is the inverse of the cumulative distribution function. The p-quantile is the value with the property that there is probability p of getting a value less than or equal to it. The median is by definition the 50% quantile.

Some details concerning the definition in the case of discontinuous distributions are glossed over here. You can fairly easily deduce the behaviour by experimenting with the R functions.

Tables of statistical distributions are almost always given in terms of quantiles. For a fixed set of probabilities, the table shows the boundary that a test statistic must cross in order to be considered significant at that level. This is purely for operational reasons; it is almost superfluous when you have the option of computing p exactly.

Theoretical quantiles are commonly used for the calculation of confidence intervals and for power calculations in connection with designing and dimensioning experiments (see Chapter 9). A simple example of a confidence interval can be given here (see also Chapter 5).

If we have n normally distributed observations with the same mean μ and standard deviation σ, then it is known that the average $\bar{x}$ is normally distributed around μ with standard deviation $\sigma/\sqrt{n}$. A 95% confidence interval for μ can be obtained as

$$\bar{x} + \sigma/\sqrt{n} \times N_{0.025} \leq \mu \leq \bar{x} + \sigma/\sqrt{n} \times N_{0.975}$$

where $N_{0.025}$ is the 2.5% quantile in the normal distribution. If $\sigma = 12$ and we have measured $n = 5$ persons and found an average of $\bar{x} = 83$, then

we can compute the relevant quantities as ("sem" means *standard error of the mean*)

```
> xbar <- 83
> sigma <- 12
> n <- 5
> sem <- sigma/sqrt(n)
> sem
[1] 5.366563
> xbar + sem * qnorm(0.025)
[1] 72.48173
> xbar + sem * qnorm(0.975)
[1] 93.51827
```

and thus find a 95% confidence interval for μ going from 72.48 to 93.52. (Notice that this is based on the assumption that σ is known. This is sometimes reasonable in process control applications. The more common case of estimating σ from the data leads to confidence intervals based on the t distribution and is discussed in Chapter 5.)

Since it is known that the normal distribution is symmetric, so that $N_{0.025} = -N_{0.975}$, it is common to write the formula for the confidence interval as $\bar{x} \pm \sigma/\sqrt{n} \times N_{0.975}$. The quantile itself is often written $\Phi^{-1}(0.975)$, where Φ is standard notation for the cumulative distribution function of the normal distribution (`pnorm`).

Another application of quantiles is in connection with Q–Q plots (see Section 4.2.3), which can be used to assess whether a set of data can reasonably be assumed to come from a given distribution.

3.5.4 Random numbers

To many people, it sounds like a contradiction in terms to generate random numbers on a computer since its results are supposed to be predictable and reproducible. What is in fact possible is to generate sequences of "pseudo-random" numbers, which for practical purposes behave *as if* they were drawn randomly.

Here random numbers are used to give the reader a feeling for the way in which randomness affects the quantities that can be calculated from a set of data. In professional statistics, they are used to create simulated data sets in order to study the accuracy of mathematical approximations and the effect of assumptions being violated.

The use of the functions that generate random numbers is straightforward. The first argument specifies the number of random numbers to compute, and the subsequent arguments are similar to those for other functions related to the same distributions. For instance,

```
> rnorm(10)
 [1] -0.2996466 -0.1718510 -0.1955634  1.2280843 -2.6074190
 [6] -0.2999453 -0.4655102 -1.5680666  1.2545876 -1.8028839
> rnorm(10)
 [1]  1.7082495  0.1432875 -1.0271750 -0.9246647  0.6402383
 [6]  0.7201677 -0.3071239  1.2090712  0.8699669  0.5882753
> rnorm(10,mean=7,sd=5)
 [1]  8.934983  8.611855  4.675578  3.670129  4.223117  5.484290
 [7] 12.141946  8.057541 -2.893164 13.590586
> rbinom(10,size=20,prob=.5)
 [1] 12 11 10  8 11  8 11  8  8 13
```

3.6 Exercises

3.1 Calculate the probability for each of the following events: (a) A standard normally distributed variable is larger than 3. (b) A normally distributed variable with mean 35 and standard deviation 6 is larger than 42. (c) Getting 10 out of 10 successes in a binomial distribution with probability 0.8. (d) $X < 0.9$ when X has the standard uniform distribution. (e) $X > 6.5$ in a χ^2 distribution with 2 degrees of freedom.

3.2 A rule of thumb is that 5% of the normal distribution lies outside an interval approximately $\pm 2s$ about the mean. To what extent is this true? Where are the limits corresponding to 1%, 0.5%, and 0.1%? What is the position of the quartiles measured in standard deviation units?

3.3 For a disease known to have a postoperative complication frequency of 20%, a surgeon suggests a new procedure. He tests it on 10 patients and there are no complications. What is the probability of operating on 10 patients successfully with the traditional method?

3.4 Simulated coin-tossing can be done using `rbinom` instead of `sample`. How exactly would you do that?

4

Descriptive statistics and graphics

Before going into the actual statistical modelling and analysis of a data set, it is often useful to make some simple characterizations of the data in terms of summary statistics and graphics.

4.1 Summary statistics for a single group

It is easy to calculate simple summary statistics with R. Here is how to calculate the mean, standard deviation, variance, and median.

```
> x <- rnorm(50)
> mean(x)
[1] 0.03301363
> sd(x)
[1] 1.069454
> var(x)
[1] 1.143731
> median(x)
[1] -0.08682795
```

Notice that the example starts with the generation of an artificial data vector `x` of 50 normally distributed observations. It is used in examples throughout this section. When reproducing the examples, you will not get exactly the same results since your random numbers will differ.

P. Dalgaard, *Introductory Statistics with R*,
DOI: 10.1007/978-0-387-79054-1_4, © Springer Science+Business Media, LLC 2008

Empirical quantiles may be obtained with the function `quantile` like this:

```
> quantile(x)
         0%         25%         50%         75%        100%
-2.60741896 -0.54495849 -0.08682795  0.70018536  2.98872414
```

As you see, by default you get the minimum, the maximum, and the three *quartiles* — the 0.25, 0.50, and 0.75 quantiles — so named because they correspond to a division into four parts. Similarly, we have *deciles* for $0.1, 0.2, \ldots, 0.9$, and *centiles* or *percentiles.* The difference between the first and third quartiles is called the *interquartile range* (IQR) and is sometimes used as a robust alternative to the standard deviation.

It is also possible to obtain other quantiles; this is done by adding an argument containing the desired percentage points. This, for example, is how to get the deciles:

```
> pvec <- seq(0,1,0.1)
> pvec
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> quantile(x,pvec)
         0%         10%         20%         30%         40%
-2.60741896 -1.07746896 -0.70409272 -0.46507213 -0.29976610
        50%         60%         70%         80%         90%
-0.08682795  0.19436950  0.49060129  0.90165137  1.31873981
       100%
 2.98872414
```

Be aware that there are several possible definitions of empirical quantiles. The one R uses by default is based on a sum polygon where the ith ranking observation is the $(i-1)/(n-1)$ quantile and intermediate quantiles are obtained by linear interpolation. It sometimes confuses students that in a sample of 10 there will be 3 observations below the first quartile with this definition. Other definitions are available via the `type` argument to `quantile`.

If there are missing values in data, things become a bit more complicated. For illustration, we use the following example.

The data set `juul` contains variables from an investigation performed by Anders Juul (Rigshospitalet, Department for Growth and Reproduction) concerning serum IGF-I (insulin-like growth factor) in a group of healthy humans, primarily schoolchildren. The data set is contained in the `ISwR` package and contains a number of variables, of which we only use `igf1` (serum IGF-I) for now, but later in the chapter we also use `tanner` (Tanner stage of puberty, a classification into five groups based on appearance

of primary and secondary sexual characteristics), `sex`, and `menarche` (indicating whether or not a girl has had her first period).

Attempting to calculate the mean of `igf1` reveals a problem.

```
> attach(juul)
>  mean(igf1)
[1] NA
```

R will not skip missing values unless explicitly requested to do so. The mean of a vector with an unknown value is unknown. However, you can give the `na.rm` argument (*n*ot *a*vailable, *rem*ove) to request that missing values be removed:

```
> mean(igf1,na.rm=T)
[1] 340.168
```

There is one slightly annoying exception: The `length` function will not understand `na.rm`, so we cannot use it to count the number of nonmissing measurements of `igf1`. However, you can use

```
> sum(!is.na(igf1))
[1] 1018
```

The construction above uses the fact that if logical values are used in arithmetic, then `TRUE` is converted to 1 and `FALSE` to 0.

A nice summary display of a numeric variable is obtained from the `summary` function:

```
> summary(igf1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   25.0   202.2   313.5   340.2   462.8   915.0   321.0
```

The `1st Qu.` and `3rd Qu.` refer to the empirical quartiles (0.25 and 0.75 quantiles).

In fact, it is possible to summarize an entire data frame with

```
> summary(juul)
      age            menarche           sex
 Min.   : 0.170   Min.   :  1.000   Min.   :1.000
 1st Qu.: 9.053   1st Qu.:  1.000   1st Qu.:1.000
 Median :12.560   Median :  1.000   Median :2.000
 Mean   :15.095   Mean   :  1.476   Mean   :1.534
 3rd Qu.:16.855   3rd Qu.:  2.000   3rd Qu.:2.000
 Max.   :83.000   Max.   :  2.000   Max.   :2.000
 NA's   : 5.000   NA's   :635.000   NA's   :5.000
      igf1           tanner          testvol
 Min.   : 25.0   Min.   :  1.000   Min.   :  1.000
 1st Qu.:202.2   1st Qu.:  1.000   1st Qu.:  1.000
```

```
 Median :313.5   Median :  2.000   Median :  3.000
 Mean   :340.2   Mean   :  2.640   Mean   :  7.896
 3rd Qu.:462.8   3rd Qu.:  5.000   3rd Qu.: 15.000
 Max.   :915.0   Max.   :  5.000   Max.   : 30.000
 NA's   :321.0   NA's   :240.000   NA's   :859.000
```

The data set has `menarche`, `sex`, and `tanner` coded as numeric variables even though they are clearly categorical. This can be mended as follows:

```
> detach(juul)
> juul$sex <- factor(juul$sex,labels=c("M","F"))
> juul$menarche <- factor(juul$menarche,labels=c("No","Yes"))
> juul$tanner <- factor(juul$tanner,
+                       labels=c("I","II","III","IV","V"))
> attach(juul)
> summary(juul)
      age          menarche     sex           igf1
 Min.   : 0.170   No  :369   M   :621   Min.   : 25.0
 1st Qu.: 9.053   Yes :335   F   :713   1st Qu.:202.2
 Median :12.560   NA's:635   NA's:  5   Median :313.5
 Mean   :15.095                         Mean   :340.2
 3rd Qu.:16.855                         3rd Qu.:462.8
 Max.   :83.000                         Max.   :915.0
 NA's   : 5.000                         NA's   :321.0
 tanner       testvol
 I   :515   Min.   :  1.000
 II  :103   1st Qu.:  1.000
 III : 72   Median :  3.000
 IV  : 81   Mean   :  7.896
 V   :328   3rd Qu.: 15.000
 NA's:240   Max.   : 30.000
            NA's   :859.000
```

Notice how the display changes for the factor variables. Note also that `juul` was detached and reattached after the modification. This is because modifying a data frame does not affect any attached version. It was not strictly necessary to do it here because `summary` works directly on the data frame whether attached or not.

In the above, the variables `sex`, `menarche`, and `tanner` were converted to factors with suitable level names (in the raw data these are represented using numeric codes). The converted variables were put back into the data frame `juul`, replacing the original `sex`, `tanner`, and `menarche` variables. We might also have used the `transform` function (or `within`):

```
> juul <- transform(juul,
+   sex=factor(sex,labels=c("M","F")),
+   menarche=factor(menarche,labels=c("No","Yes")),
+   tanner=factor(tanner,labels=c("I","II","III","IV","V")))
```

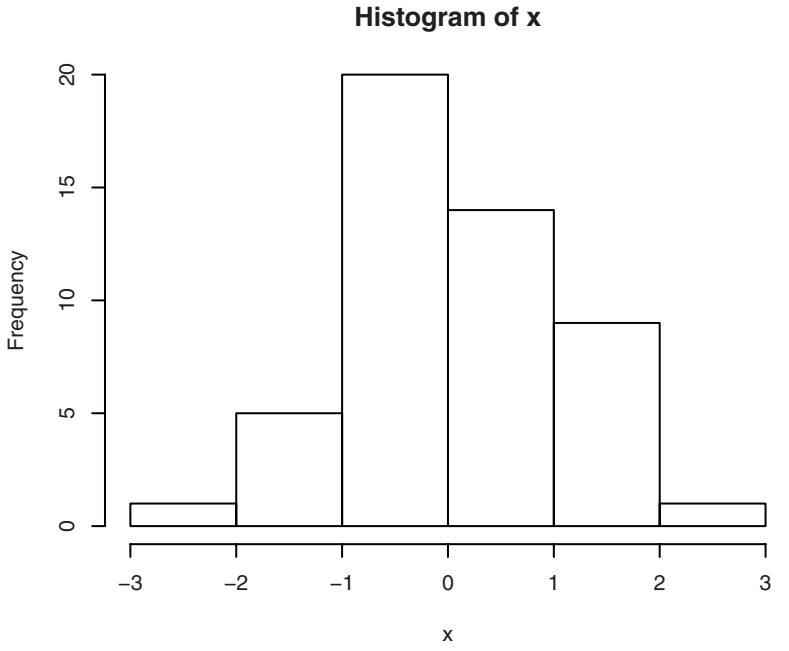

Figure 4.1. Histogram.

4.2 Graphical display of distributions

4.2.1 Histograms

You can get a reasonable impression of the shape of a distribution by drawing a histogram; that is, a count of how many observations fall within specified divisions ("bins") of the *x*-axis (Figure 4.1).

```
> hist(x)
```

By specifying `breaks=n` in the `hist` call, you get *approximately n* bars in the histogram since the algorithm tries to create "pretty" cutpoints. You can have full control over the interval divisions by specifying `breaks` as a vector rather than as a number. Altman (1991, pp. 25–26) contains an example of accident rates by age group. These are given as a count in age groups 0–4, 5–9, 10–15, 16, 17, 18–19, 20–24, 25–59, and 60–79 years of age. The data can be entered as follows:

```
> mid.age <- c(2.5,7.5,13,16.5,17.5,19,22.5,44.5,70.5)
> acc.count <- c(28,46,58,20,31,64,149,316,103)
> age.acc <- rep(mid.age,acc.count)
```

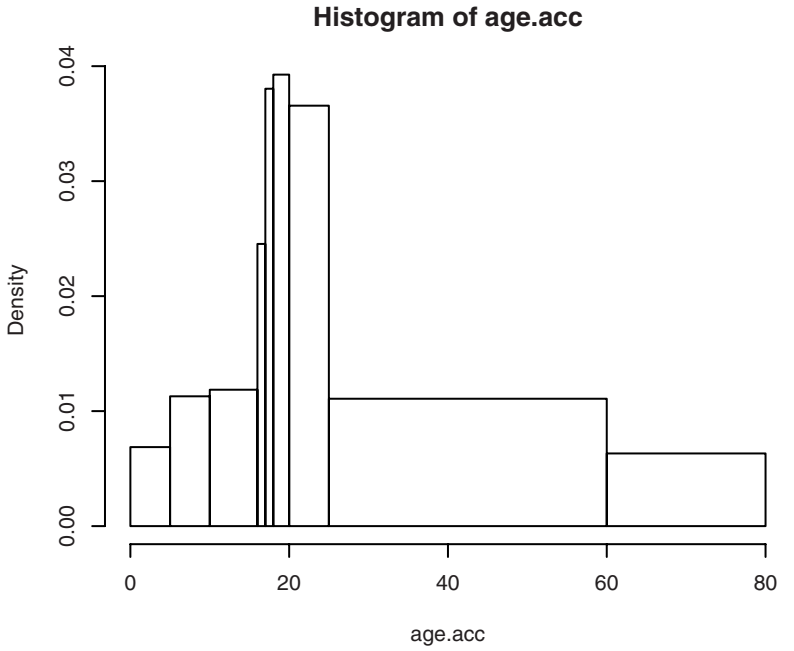

Figure 4.2. Histogram with unequal divisions.

```
> brk <- c(0,5,10,16,17,18,20,25,60,80)
> hist(age.acc,breaks=brk)
```

Here the first three lines generate pseudo-data from the table in the book. For each interval, the relevant number of "observations" is generated with an age set to the midpoint of the interval; that is, 28 2.5-year-olds, 46 7.5-year-olds, etc. Then a vector `brk` of cutpoints is defined (note that the extremes need to be included) and used as the `breaks` argument to `hist`, yielding Figure 4.2.

Notice that you automatically got the "correct" histogram where the *area* of a column is proportional to the number. The *y*-axis is in density units (that is, proportion of data per *x* unit), so that the total area of the histogram will be 1. If, for some reason, you want the (misleading) histogram where the column height is the raw number in each interval, then it can be specified using `freq=T`. For equidistant breakpoints, that is the default (because then you can see how many observations have gone into each column), but you can set `freq=F` to get densities displayed. This is really just a change of scale on the *y*-axis, but it has the advantage that it becomes possible to overlay the histogram with a corresponding theoretical density function.

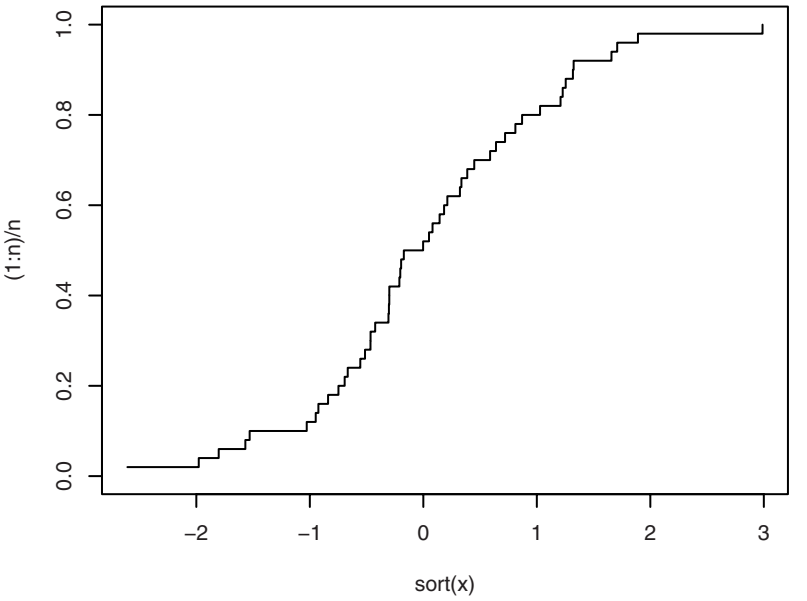

Figure 4.3. Empirical cumulative distribution function.

4.2.2 *Empirical cumulative distribution*

The empirical cumulative distribution function is defined as the fraction of data smaller than or equal to x. That is, if x is the kth smallest observation, then the proportion k/n of the data is smaller than or equal to x (7/10 if x is no. 7 of 10). The empirical cumulative distribution function can be plotted as follows (see Figure 4.3) where `x` is the simulated data vector from Section 4.1:

```
> n <- length(x)
> plot(sort(x),(1:n)/n,type="s",ylim=c(0,1))
```

The plotting parameter `type="s"` gives a *s*tep function where (x, y) is the left end of the steps and `ylim` is a vector of two elements specifying the extremes of the y-coordinates on the plot. Recall that `c(...)` is used to create vectors.

Some more elaborate displays of empirical cumulative distribution functions are available via the `ecdf` function. This is also more precise regarding the mathematical definition of the step function.

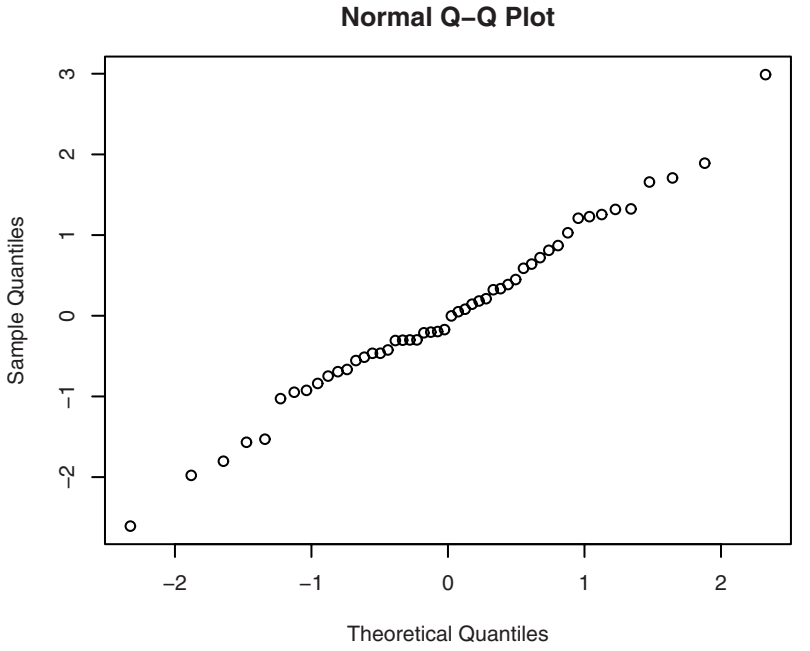

Figure 4.4. Q–Q plot using `qqnorm(x)`.

4.2.3 Q–Q plots

One purpose of calculating the empirical cumulative distribution function (c.d.f.) is to see whether data can be assumed normally distributed. For a better assessment, you might plot the kth smallest observation against the expected value of the kth smallest observation out of n in a standard normal distribution. The point is that in this way you would expect to obtain a straight line if data come from a normal distribution with *any* mean and standard deviation.

Creating such a plot is slightly complicated. Fortunately, there is a built-in function for doing it, `qqnorm`. The result of using it can be seen in Figure 4.4. You only have to write

```
> qqnorm(x)
```

As the title of the plot indicates, plots of this kind are also called "Q–Q plots" (quantile versus quantile). Notice that x and y are interchanged relative to the empirical c.d.f. — the observed values are now drawn along the y-axis. You should notice that with this convention the distribution has heavy tails if the outer parts of the curve are steeper than the middle part.

Some readers will have been taught "probability plots", which are similar but have the axes interchanged. It can be argued that the way R draws the plot is the better one since the theoretical quantiles are known in advance, while the empirical quantiles depend on data. You would normally choose to draw fixed values horizontally and variable values vertically.

4.2.4 Boxplots

A "boxplot", or more descriptively a "box-and-whiskers plot", is a graphical summary of a distribution. Figure 4.5 shows boxplots for IgM and its logarithm; see the example on page 23 in Altman (1991).

Here is how a boxplot is drawn in R. The box in the middle indicates "hinges" (nearly quartiles; see the help page for `boxplot.stats`) and median. The lines ("whiskers") show the largest or smallest observation that falls within a distance of 1.5 times the box size from the nearest hinge. If any observations fall farther away, the additional points are considered "extreme" values and are shown separately.

The practicalities are these:

```
> par(mfrow=c(1,2))
> boxplot(IgM)
> boxplot(log(IgM))
> par(mfrow=c(1,1))
```

A layout with two plots side by side is specified using the `mfrow` graphical parameter. It should be read as "*m*ulti*f*rame, *row*wise, 1×2 layout". Individual plots are organized in one row and two columns. As you might guess, there is also an `mfcol` parameter to plot columnwise. In a 2×2 layout, the difference is whether plot no. 2 is drawn in the top right or bottom left corner.

Notice that it is necessary to reset the layout parameter to `c(1,1)` at the end unless you also want two plots side by side subsequently.

4.3 Summary statistics by groups

When dealing with grouped data, you will often want to have various summary statistics computed within groups; for example, a table of means and standard deviations. To this end, you can use `tapply` (see Section 1.2.15). Here is an example concerning the folate concentration in red blood cells according to three types of ventilation during anesthesia (Alt-

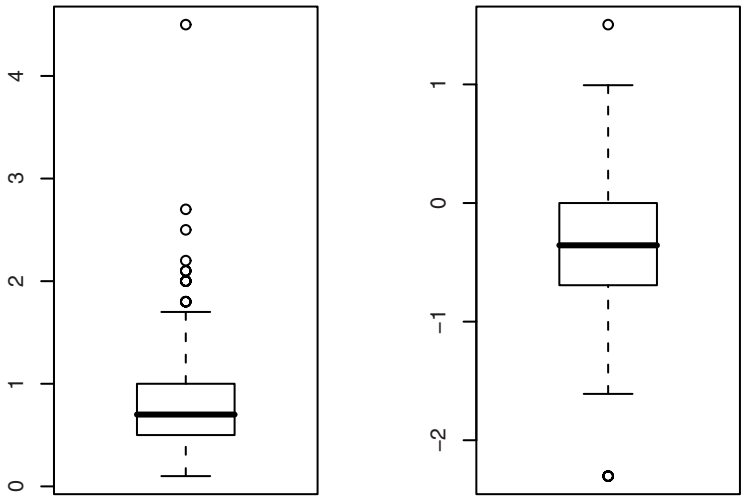

Figure 4.5. Boxplots for IgM and log IgM.

man, 1991, p. 208). We return to this example in Section 7.1, which also contains the explanation of the category names.

```
> attach(red.cell.folate)
> tapply(folate,ventilation,mean)
N2O+O2,24h  N2O+O2,op     O2,24h
  316.6250   256.4444   278.0000
```

The `tapply` call takes the `folate` variable, splits it according to `ventilation`, and computes the mean for each group. In the same way, standard deviations and the number of observations in the groups can be computed.

```
> tapply(folate,ventilation,sd)
N2O+O2,24h  N2O+O2,op     O2,24h
  58.71709   37.12180   33.75648
> tapply(folate,ventilation,length)
N2O+O2,24h  N2O+O2,op     O2,24h
         8          9          5
```

Try something like this for a nicer display:

```
> xbar <- tapply(folate, ventilation, mean)
> s <- tapply(folate, ventilation, sd)
> n <- tapply(folate, ventilation, length)
> cbind(mean=xbar, std.dev=s, n=n)
               mean  std.dev n
N2O+O2,24h 316.6250 58.71709 8
N2O+O2,op  256.4444 37.12180 9
O2,24h     278.0000 33.75648 5
```

For the `juul` data, we might want the mean `igf1` by `tanner` group, but of course we run into the problem of missing values again:

```
> tapply(igf1, tanner, mean)
  I  II III  IV   V
 NA  NA  NA  NA  NA
```

We need to get `tapply` to pass `na.rm=T` as a parameter to `mean` to make it exclude the missing values. This is achieved simply by passing it as an additional argument to `tapply`.

```
> tapply(igf1, tanner, mean, na.rm=T)
       I       II      III       IV        V
207.4727 352.6714 483.2222 513.0172 465.3344
```

The functions `aggregate` and `by` are variations on the same topic. The former is very much like `tapply`, except that it works on an entire data frame and presents its results as a data frame. This is useful for presenting many variables at once; e.g.,

```
> aggregate(juul[c("age","igf1")],
+           list(sex=juul$sex), mean, na.rm=T)
  sex      age     igf1
1   M 15.38436 310.8866
2   F 14.84363 368.1006
```

Notice that the grouping argument in this case must be a list, even when it is one-dimensional, and that the names of the list elements get used as column names in the output. Notice also that since the function is applied to all columns of the data frame, you may have to choose a subset of columns, in this case the numeric variables.

The indexing variable is not necessarily part of the data frame that is being aggregated, and there is no attempt at "smart evaluation" as there is in `subset`, so you have to spell out `juul$sex`. You can also use the fact that data frames are list-like and say

```
> aggregate(juul[c("age","igf1")], juul["sex"], mean, na.rm=T)
  sex      age     igf1
1   M 15.38436 310.8866
2   F 14.84363 368.1006
```

(the "trick" being that indexing a data frame with single brackets yields a data frame as the result).

The `by` function is again similar, but different. The difference is that the function now takes an entire (sub-) data frame as its argument, so that you can for instance summarize the Juul data by sex as follows:

```
> by(juul, juul["sex"], summary)
sex: M
      age         menarche    sex          igf1         tanner
 Min.   : 0.17   No  :  0   M:621   Min.   : 29.0   I   :291
 1st Qu.: 8.85   Yes :  0   F:  0   1st Qu.:176.0   II  : 55
 Median :12.38   NA's:621           Median :280.0   III : 34
 Mean   :15.38                      Mean   :310.9   IV  : 41
 3rd Qu.:16.77                      3rd Qu.:430.2   V   :124
 Max.   :83.00                      Max.   :915.0   NA's: 76
                                    NA's   :145.0
    testvol
 Min.   :  1.000
 1st Qu.:  1.000
 Median :  3.000
 Mean   :  7.896
 3rd Qu.: 15.000
 Max.   : 30.000
 NA's   :141.000
------------------------------------------------
sex: F
      age         menarche    sex          igf1         tanner
 Min.   : 0.25   No  :369   M:  0   Min.   : 25.0   I   :224
 1st Qu.: 9.30   Yes :335   F:713   1st Qu.:233.0   II  : 48
 Median :12.80   NA's:  9           Median :352.0   III : 38
 Mean   :14.84                      Mean   :368.1   IV  : 40
 3rd Qu.:16.93                      3rd Qu.:483.0   V   :204
 Max.   :75.12                      Max.   :914.0   NA's:159
                                    NA's   :176.0
    testvol
 Min.   : NA
 1st Qu.: NA
 Median : NA
 Mean   :NaN
 3rd Qu.: NA
 Max.   : NA
 NA's   :713
```

The result of the call to `by` is actually a list of objects that has has been wrapped as an object of class `"by"` and printed using a print method for that class. You can assign the result to a variable and access the result for each subgroup using standard list indexing.

The same technique can also be used to generate more elaborate statistical analyses for each group.

4.4 Graphics for grouped data

In dealing with grouped data, it is important to be able not only to create plots for each group but also to compare the plots between groups. In this section we review some general graphical techniques that allow us to display similar plots for several groups on the same page. Some functions have specific features for displaying data from more than one group.

4.4.1 Histograms

We have already seen in Section 4.2.1 how to obtain a histogram simply by typing `hist(x)`, where `x` is the variable containing the data. R will then choose a number of groups so that a reasonable number of data points fall in each bin while at the same time ensuring that the cutpoints are "pretty" numbers on the x-axis.

It is also mentioned there that an alternative number of intervals can be set via the argument `breaks`, although you do not always get exactly the number you asked for since R reserves the right to choose "pretty" column boundaries. For instance, multiples of 0.5 MJ are chosen in the following example using the `energy` data introduced in Section 1.2.14 on the 24-hour energy expenditure for two groups of women.

In this example, some further techniques of general use are illustrated. The end result is seen in Figure 4.6, but first we must fetch the data:

```
> attach(energy)
> expend.lean <- expend[stature=="lean"]
> expend.obese <- expend[stature=="obese"]
```

Notice how we separate the `expend` vector in the `energy` data frame into two vectors according to the value of the factor `stature`.

Now we do the actual plotting:

```
> par(mfrow=c(2,1))
> hist(expend.lean,breaks=10,xlim=c(5,13),ylim=c(0,4),col="white")
> hist(expend.obese,breaks=10,xlim=c(5,13),ylim=c(0,4),col="grey")
> par(mfrow=c(1,1))
```

We set `par(mfrow=c(2,1))` to get the two histograms in the same plot. In the `hist` commands themselves, we used the `breaks` argument as already mentioned and `col`, whose effect should be rather obvious. We also used `xlim` and `ylim` to get the same x and y axes in the two plots. However, it is a coincidence that the columns have the same width.

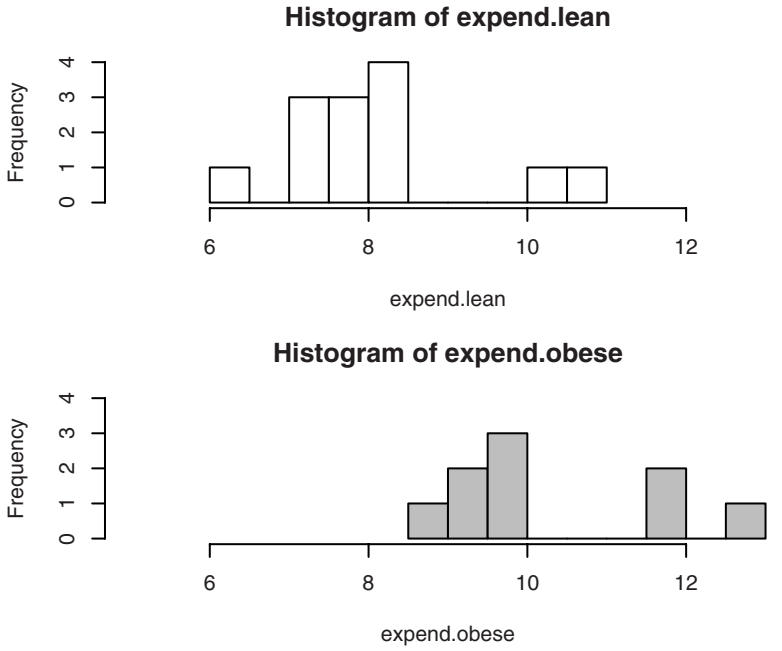

Figure 4.6. Histograms with refinements.

As a practical remark, when working with plots like the above, where more than a single line of code is required, it gets cumbersome to use command recall in the R console window every time something needs to be changed. A better idea may be to start up a script window or a plain-text editor and cut and paste entire blocks of code from there (see Section 2.1.3). You might also take it as an incentive to start writing simple functions.

4.4.2 Parallel boxplots

You might want a set of boxplots from several groups in the same frame. `boxplot` can handle this both when data are given in the form of separate vectors from each group and when data are in one long vector and a parallel vector or factor defines the grouping. To illustrate the latter, we use the `energy` data introduced in Section 1.2.14.

Figure 4.7 is created as follows:

```
> boxplot(expend ~ stature)
```

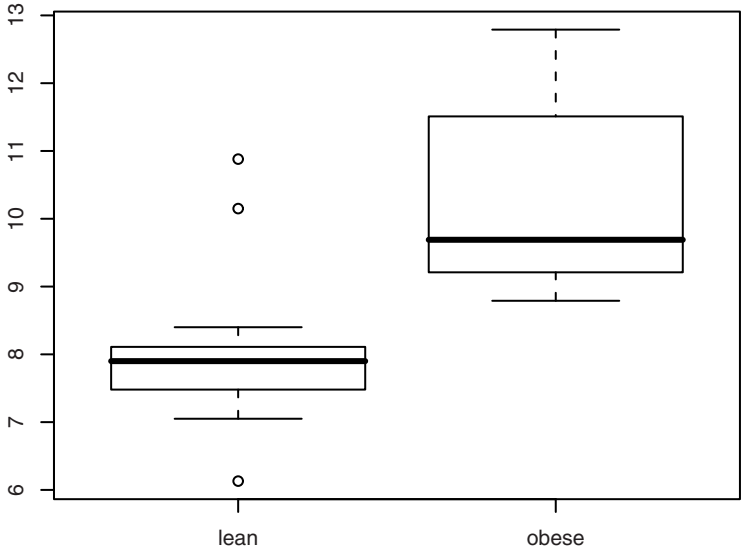

Figure 4.7. Parallel boxplot.

We could also have based the plot on the separate vectors `expend.lean` and `expend.obese`. In that case, a syntax is used that specifies the vectors as two separate arguments:

```
> boxplot(expend.lean,expend.obese)
```

The plot is not shown here, but the only difference lies in the labelling of the *x*-axis. There is also a third form, where data are given as a single argument that is a list of vectors.

The bottom plot has been made using the complete `expend` vector and the grouping variable `fstature`.

Notation of the type `y ~ x` should be read "`y` described using `x`". This is the first example we see of a *model formula*. We see many more examples of model formulas later on.

4.4.3 Stripcharts

The boxplots made in the preceding section show a "Laurel & Hardy" effect that is not really well founded in the data. The cause is that the in-

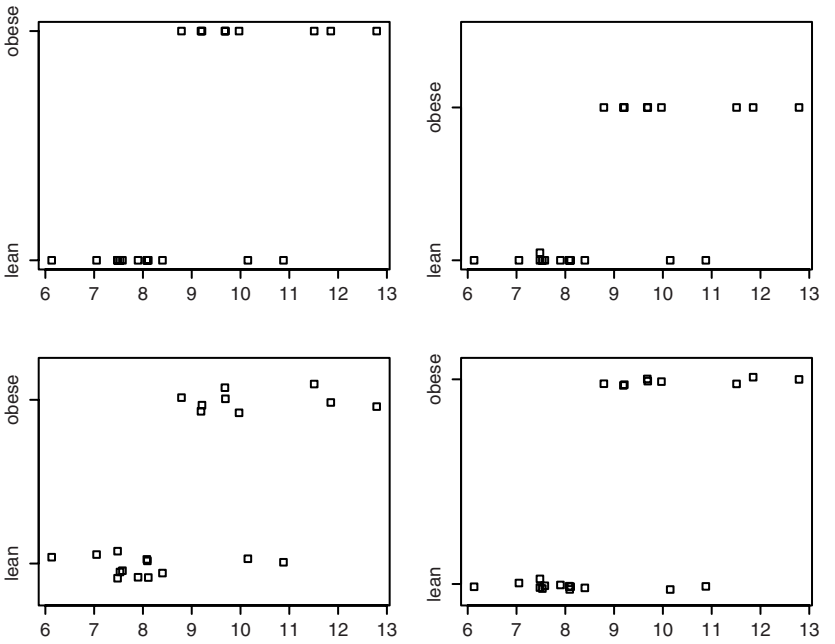

Figure 4.8. Stripcharts in four variations.

terquartile range is quite a bit larger in one group than in the other, making the boxplot appear "fatter". With groups as small as these, the quartiles will be quite inaccurately determined, and it may therefore be more desirable to plot the raw data. If you were to do this by hand, you might draw a dot diagram where every number is marked with a dot on a number line. R's automated variant of this is the function `stripchart`. Four variants of stripcharts are shown in Figure 4.8.

The four plots were created as follows:

```
> opar <- par(mfrow=c(2,2), mex=0.8, mar=c(3,3,2,1)+.1)
> stripchart(expend ~ stature)
> stripchart(expend ~ stature, method="stack")
> stripchart(expend ~ stature, method="jitter")
> stripchart(expend ~ stature, method="jitter", jitter=.03)
> par(opar)
```

Notice that a little `par` magic was used to reduce the spacing between the four plots. The `mex` setting reduces the interline distance, and `mar` reduces the number of lines that surround the plot region. This can be done for these plots since they have neither main title, subtitle, nor x and y labels.

All the original values of the changed settings can be stored in a variable (here `opar`) and reestablished with `par(opar)`.

The first plot is a standard stripchart, where the points are simply plotted on a line. The problem with this is that some points can become invisible because they are overplotted. This is why there is a `method` argument, which can be set to either `"stack"` or `"jitter"`.

The former method stacks points with identical values, but it only does so if data are *completely identical*, so in the upper right plot, it is only the two replicates of 7.48 that get stacked, whereas 8.08, 8.09, and 8.11 are still plotted in almost the same spot.

The "jitter" method offsets all points a random amount vertically. The standard jittering on plot no. 3 (bottom left) is a bit large; it may be preferable to make it clearer that data are placed along a horizontal line. For that purpose, you can set `jitter` lower than the default of 0.1, which is done in the fourth plot.

In this example we have not bothered to specify data in several forms as we did for `boxplot` but used `expend~stature` throughout. We could also have written

```
stripchart(list(lean=expend.lean, obese=expend.obese))
```

but `stripchart(expend.lean, expend.obese)` cannot be used.

4.5 Tables

Categorical data are usually described in the form of tables. This section outlines how you can create tables from your data and calculate relative frequencies.

4.5.1 Generating tables

We deal mainly with two-way tables. In the first example, we enter a table directly, as is required for tables taken from a book or a journal article.

A two-way table can be entered as a matrix object (Section 1.2.7). Altman (1991, p. 242) contains an example on caffeine consumption by marital status among women giving birth. That table may be input as follows:

```
> caff.marital <- matrix(c(652,1537,598,242,36,46,38,21,218
+ ,327,106,67),
+ nrow=3,byrow=T)
> caff.marital
     [,1] [,2] [,3] [,4]
[1,]  652 1537  598  242
[2,]   36   46   38   21
[3,]  218  327  106   67
```

The `matrix` function needs an argument containing the table values as a single vector and also the number of rows in the argument `nrow`. By default, the values are entered columnwise; if rowwise entry is desired, then you need to specify `byrow=T`.

You might also give the number of columns instead of rows using `ncol`. If exactly one of `ncol` and `nrow` is given, R will compute the other one so that it fits the number of values. If both `ncol` and `nrow` are given and it does not fit the number of values, the values will be "recycled", which in some (other!) circumstances can be useful.

To get readable printouts, you can add row and column names to the matrices.

```
> colnames(caff.marital) <- c("0","1-150","151-300",">300")
> rownames(caff.marital) <- c("Married","Prev.married","Single")
> caff.marital
              0 1-150 151-300 >300
Married      652  1537     598  242
Prev.married  36    46      38   21
Single       218   327     106   67
```

Furthermore, you can name the row and column names as follows. This is particularly useful if you are generating many tables with similar classification criteria.

```
> names(dimnames(caff.marital)) <- c("marital","consumption")
> caff.marital
             consumption
marital          0 1-150 151-300 >300
  Married      652  1537     598  242
  Prev.married  36    46      38   21
  Single       218   327     106   67
```

Actually, I glossed over something. Tables are not completely equivalent to matrices. There is a `"table"` class for which special methods exist, and you can convert to that class using `as.table(caff.marital)`. The `table` function below returns an object of class `"table"`.

For most elementary purposes, you can use matrices where two-dimensional tables are expected. One important case where you do need `as.table` is when converting a table to a data frame of counts:

```
> as.data.frame(as.table(caff.marital))
          marital consumption Freq
1         Married           0  652
2    Prev.married           0   36
3          Single           0  218
4         Married       1-150 1537
5    Prev.married       1-150   46
6          Single       1-150  327
7         Married     151-300  598
8    Prev.married     151-300   38
9          Single     151-300  106
10        Married        >300  242
11   Prev.married        >300   21
12         Single        >300   67
```

In practice, the more frequent case is that you have a data frame with variables for each person in a data set. In that case, you should do the tabulation with `table`, `xtabs`, or `ftable`. These functions will generally work for tabulating numeric vectors as well as factor variables, but the latter will have their levels used for row and column names automatically. Hence, it is recommended to convert numerically coded categorical data into factors. The `table` function is the oldest and most basic of the three. The two others offer formula-based interfaces and better printing of multiway tables.

The data set `juul` was introduced on p. 68. Here we look at some other variables in that data set, namely `sex` and `menarche`; the latter indicates whether or not a girl has had her first period. We can generate some simple tables as follows:

```
> table(sex)
sex
  M   F
621 713
> table(sex,menarche)
   menarche
sex  No Yes
  M   0   0
  F 369 335
> table(menarche,tanner)
        tanner
menarche   I II III IV   V
     No  221 43  32 14   2
     Yes   1  1   5 26 202
```

Of course, the table of menarche versus sex is just a check on internal consistency of the data. The table of menarche versus Tanner stage of puberty is more interesting.

There are also tables with more than two sides, but not many simple statistical functions use them. Briefly, to tabulate such data, just write, for example, `table(factor1,factor2,factor3)`. To input a table of cell counts, use the `array` function (an analogue of `matrix`).

The `xtabs` function is quite similar to `table` except that it uses a model formula interface. This most often uses a one-sided formula where you just list the classification variables separated by `+`.

```
> xtabs(~ tanner + sex, data=juul)
      sex
tanner   M   F
   I   291 224
   II   55  48
   III  34  38
   IV   41  40
   V   124 204
```

Notice how the interface allows you to refer to variables in a data frame without attaching it. The empty left-hand side can be replaced by a vector of counts in order to handle pretabulated data.

The formatting of multiway tables from `table` or `xtabs` is not really nice; e.g.,

```
> xtabs(~ dgn + diab + coma, data=stroke)
, , coma = No

     diab
dgn    No Yes
  ICH  53   6
  ID  143  21
  INF 411  64
  SAH  38   0

, , coma = Yes

     diab
dgn    No Yes
  ICH  19   1
  ID   23   3
  INF  23   2
  SAH   9   0
```

As you add dimensions, you get more of these two-sided subtables and it becomes rather easy to lose track. This is where `ftable` comes in. This function creates "flat" tables; e.g., like this:

```
> ftable(coma + diab ~ dgn, data=stroke)
    coma  No     Yes
    diab  No Yes  No Yes
dgn
ICH       53   6  19   1
ID       143  21  23   3
INF      411  64  23   2
SAH       38   0   9   0
```

That is, variables on the left-hand side tabulate across the page and those on the right tabulate downwards. `ftable` works on raw data as shown, but its `data` argument can also be a table as generated by one of the other functions.

Like any matrix, a table can be transposed with the `t` function:

```
> t(caff.marital)
           marital
consumption Married Prev.married Single
    0           652           36    218
    1-150      1537           46    327
    151-300     598           38    106
    >300        242           21     67
```

For multiway tables, exchanging indices (generalized transposition) is done by `aperm`.

4.5.2 *Marginal tables and relative frequency*

It is often desired to compute marginal tables; that is, the sums of the counts along one or the other dimension of a table. Due to missing values, this might not coincide with just tabulating a single factor. This is done fairly easily using the `apply` function (Section 1.2.15), but there is also a simplified version called `margin.table`, described below.

First, we need to generate the table itself:

```
> tanner.sex <- table(tanner,sex)
```

(`tanner.sex` is an arbitrarily chosen name for the crosstable.)

```
> tanner.sex
      sex
tanner   M   F
   I   291 224
   II   55  48
   III  34  38
   IV   41  40
   V   124 204
```

Then we compute the marginal tables:

```
> margin.table(tanner.sex,1)
tanner
  I  II III  IV   V
515 103  72  81 328
> margin.table(tanner.sex,2)
sex
  M   F
545 554
```

The second argument to `margin.table` is the number of the marginal index: 1 and 2 give row and column totals, respectively.

Relative frequencies in a table are generally expressed as proportions of the row or column totals. Tables of relative frequencies can be constructed using `prop.table` as follows:

```
> prop.table(tanner.sex,1)
      sex
tanner         M         F
   I   0.5650485 0.4349515
   II  0.5339806 0.4660194
   III 0.4722222 0.5277778
   IV  0.5061728 0.4938272
   V   0.3780488 0.6219512
```

Note that the *rows* (1st index) sum to 1. If a table of percentages is desired, just multiply the entire table by 100.

`prop.table` cannot be used to express the numbers relative to the grand total of the table, but you can of course always write

```
> tanner.sex/sum(tanner.sex)
      sex
tanner          M          F
   I   0.26478617 0.20382166
   II  0.05004550 0.04367607
   III 0.03093722 0.03457689
   IV  0.03730664 0.03639672
   V   0.11282985 0.18562329
```

The functions `margin.table` and `prop.table` also work on multiway tables — the `margin` argument can be a vector if the relevant margin has two or more dimensions.

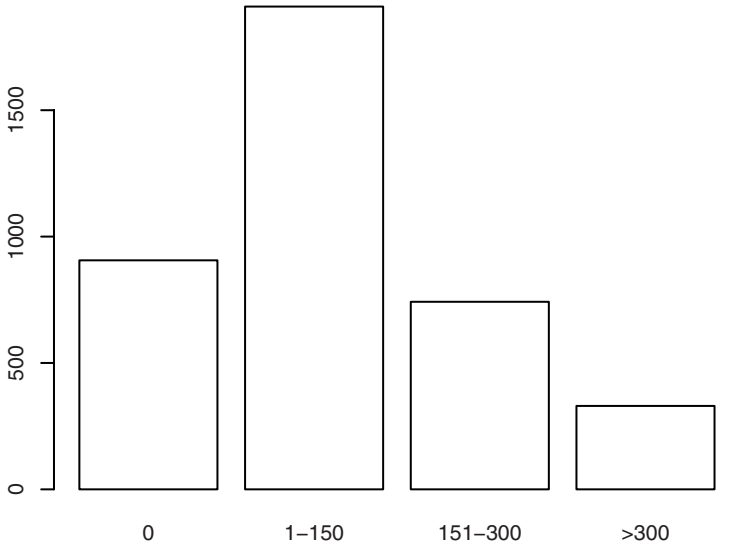

Figure 4.9. Simple `barplot` of total caffeine consumption.

4.6 Graphical display of tables

For presentation purposes, it may be desirable to display a graph rather than a table of counts or percentages. In this section, the main methods for doing this are described.

4.6.1 Barplots

Barplots are made using `barplot`. This function takes an argument, which can be a vector or a matrix. The simplest variant goes as follows (Figure 4.9):

```
> total.caff <- margin.table(caff.marital,2)
> total.caff
consumption
      0   1-150 151-300    >300
    906    1910     742     330
> barplot(total.caff, col="white")
```

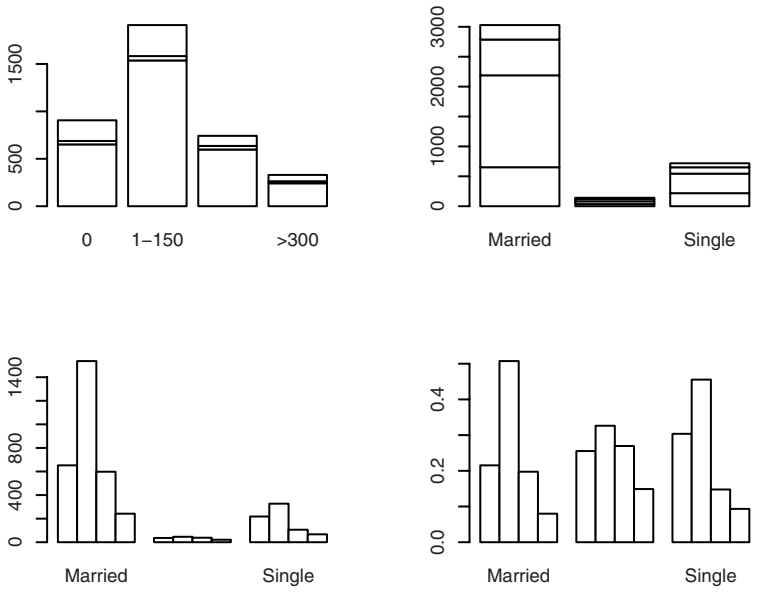

Figure 4.10. Four variants of `barplot` on a two-way table.

Without the `col="white"` argument, the plot comes out in colour, but this is not suitable for a black and white book illustration.

If the argument is a matrix, then `barplot` creates by default a "stacked barplot", where the columns are partitioned according to the contributions from different rows of the table. If you want to place the row contributions beside each other instead, you can use the argument `beside=T`. A series of variants is found in Figure 4.10, which is constructed as follows:

```
> par(mfrow=c(2,2))
> barplot(caff.marital, col="white")
> barplot(t(caff.marital), col="white")
> barplot(t(caff.marital), col="white", beside=T)
> barplot(prop.table(t(caff.marital),2), col="white", beside=T)
> par(mfrow=c(1,1))
```

In the last three plots, we switched rows and columns with the transposition function `t`. In the very last one, the columns are expressed as proportions of the total number in the group. Thus, information is lost on the relative sizes of the marital status groups, but the group of previously married women (recall that the data set deals with women giving birth)

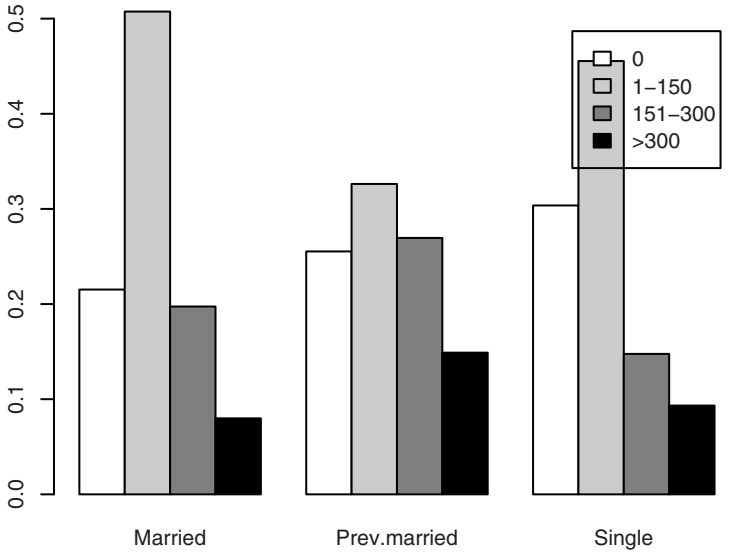

Figure 4.11. Bar plot with specified colours and legend.

is so small that it otherwise becomes almost impossible to compare their caffeine consumption profile with those of the other groups.

As usual, there are a multitude of ways to "prettify" the plots. Here is one possibility (Figure 4.11):

```
> barplot(prop.table(t(caff.marital),2),beside=T,
+ legend.text=colnames(caff.marital),
+ col=c("white","grey80","grey50","black"))
```

Notice that the legend overlaps the top of one of the columns. R is not designed to be able to find a clear area in which to place the legend. However, you can get full control of the legend's position if you insert it explicitly with the `legend` function. For that purpose, it will be helpful to use `locator()`, which allows you to click a mouse button over the plot and have the coordinates returned. See p. 209 for more about this.

4.6.2 Dotcharts

The Cleveland dotcharts, named after William S. Cleveland (1994), can be employed to study a table from both sides at the same time. They contain

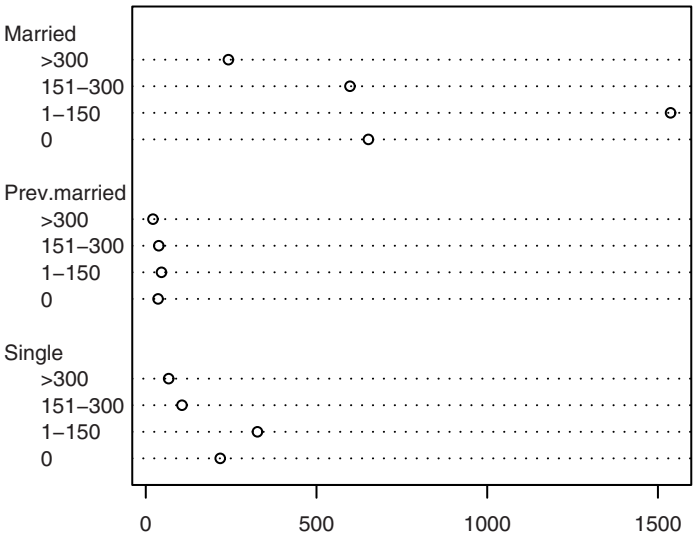

Figure 4.12. Dotchart of caffeine consumption.

the same information as barplots with `beside=T` but give quite a different visual impression. We content ourselves with a single example here (Figure 4.12):

```
> dotchart(t(caff.marital), lcolor="black")
```

(The line colour was changed from the default `"gray"` because it tends to be hard to see in print.)

4.6.3 Piecharts

Piecharts are traditionally frowned upon by statisticians because they are so often used to make trivial data look impressive and are difficult to decode for the human mind. They very rarely contain information that would not have been at least as effectively conveyed in a barplot. Once in a while they are useful, though, and it is no problem to get R to draw them. Here is a way to represent the table of caffeine consumption versus marital status (Figure 4.13; see Section 4.4.3 for an explanation of the "`par` magic" used to reduce the space between the subplots):

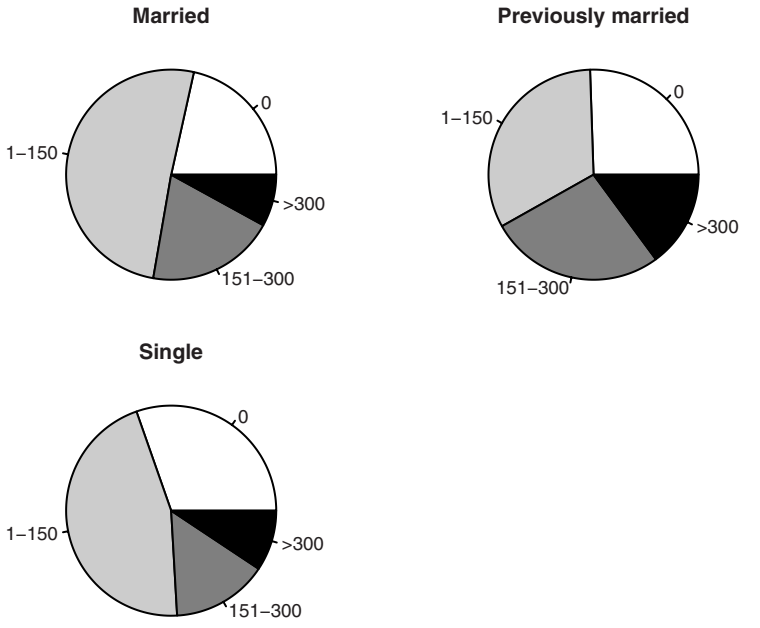

Figure 4.13. Pie charts of caffeine consumption according to marital status.

```
> opar <- par(mfrow=c(2,2),mex=0.8, mar=c(1,1,2,1))
> slices <- c("white","grey80","grey50","black")
> pie(caff.marital["Married",], main="Married", col=slices)
> pie(caff.marital["Prev.married",],
+          main="Previously married", col=slices)
> pie(caff.marital["Single",], main="Single", col=slices)
> par(opar)
```

The `col` argument sets the colour of the pie slices.

There are more possibilities with `piechart`. The help page for `pie` contains an illustrative example concerning the distribution of pie sales (!) by pie type.

4.7 Exercises

4.1 Explore the possibilities for different kinds of line and point plots. Vary the plot symbol, line type, line width, and colour.

4.2 If you make a plot like `plot(rnorm(10),type="o")` with overplotted lines and points, the lines will be visible inside the plotting symbols. How can this be avoided?

4.3 How can you overlay two `qqnorm` plots in the same plotting area? What goes wrong if you try to generate the plot using `type="l"`, and how do you avoid that?

4.4 Plot a histogram for the `react` data set. Since these data are highly discretized, the histogram will be biased. Why? You may want to try `truehist` from the `MASS` package as a replacement.

4.5 Generate a sample vector `z` of five random numbers from the uniform distribution, and plot `quantile(z,x)` as a function of `x` (use `curve`, for instance).

5

One- and two-sample tests

Most of the rest of this book describes applications of R for actual statistical analysis. The focus to some extent shifts from explanation of the syntax to description of the output and specific arguments to the relevant functions.

Some of the most basic statistical tests deal with comparing continuous data either between two groups or against an a priori stipulated value. This is the topic for this chapter.

Two functions are introduced here, namely `t.test` and `wilcox.test` for t tests and Wilcoxon tests, respectively. Both can be applied to one- and two-sample problems as well as paired data. Notice that the "two-sample Wilcoxon test" is the same as the one called the "Mann–Whitney test" in many textbooks.

5.1 One-sample t test

The t tests are based on an assumption that data come from the normal distribution. In the one-sample case we thus have data $x_1, \ldots, x_n$ assumed to be independent realizations of random variables with distribution $N(\mu, \sigma^2)$, which denotes the normal distribution with mean μ and variance σ^2, and we wish to test the *null hypothesis* that $\mu = \mu_0$. We can estimate the parameters μ and σ by the empirical mean $\bar{x}$ and standard

P. Dalgaard, *Introductory Statistics with R*,
DOI: 10.1007/978-0-387-79054-1_5, © Springer Science+Business Media, LLC 2008

deviation s, although we must realize that we could never pinpoint their values exactly.

The key concept is that of the *standard error of the mean*, or SEM. This describes the variation of the average of n random values with mean μ and variance σ^2. This value is

$$\text{SEM} = \sigma/\sqrt{n}$$

and means that if you were to repeat the entire experiment several times and calculate an average for each experiment, then these averages would follow a distribution that is narrower than that of the original distribution. The crucial point is that even based on a single sample, it is possible to calculate an empirical SEM as $s/\sqrt{n}$ using the empirical standard deviation of the sample. This value will tell us how far the observed mean may reasonably have strayed from its true value. For normally distributed data, the rule of thumb is that there is a 95% probability of staying within $\mu \pm 2\sigma$, so we would expect that if μ_0 were the true mean, then $\bar{x}$ should be within 2 SEMs of it. Formally, you calculate

$$t = \frac{\bar{x} - \mu_0}{\text{SEM}}$$

and see whether this falls within an *acceptance region* outside which t should fall with probability equal to a specified *significance level*. This is often chosen as 5%, in which case the acceptance region is almost, but not exactly, the interval from -2 to 2.

In small samples, it is necessary to correct for the fact that an empirical SEM is used and that the distribution of t therefore has somewhat "heavier tails" than the $N(0,1)$: Large deviations happen more frequently than in the normal distribution since they can result from normalizing with an SEM that is too small. The correct values for the acceptance region can be looked up as quantiles in the t distribution with $f = n - 1$ degrees of freedom.

If t falls outside the acceptance region, then we reject the null hypothesis at the chosen significance level. Alternatively (and equivalently), you can calculate the *p-value*, which is the probability of obtaining a value as numerically large as or larger than the observed t and reject the hypothesis if the p-value is less than the significance level.

Sometimes you have prior information on the direction of an effect; for instance, that all plausible mechanisms that would cause μ not to equal μ_0 would tend to make it bigger. In those cases, you may choose to reject the hypothesis only if t falls in the upper tail of the distribution. This is known as *testing against a one-sided alternative*. Since removing the lower tail from the rejection region effectively halves the significance level, a one-sided test at a given level will have a smaller cutoff point. Similarly, p-values

are calculated as the probability of a larger value than observed rather than a numerically larger one, effectively halving the p-value as long as the observed effect is in the stipulated direction. One-sided tests should be used with some care, preferably only when there is a clear statement of the intent to use them in the study protocol. Switching to a one-sided test to make an otherwise nonsignificant result significant could easily be regarded as dishonest.

Here is an example concerning daily energy intake in kJ for 11 women (Altman, 1991, p. 183). First, the values are placed in a data vector:

```
> daily.intake <- c(5260,5470,5640,6180,6390,6515,
+ 6805,7515,7515,8230,8770)
```

Let us first look at some simple summary statistics, even though these are hardly necessary for such a small data set:

```
> mean(daily.intake)
[1] 6753.636
> sd(daily.intake)
[1] 1142.123
> quantile(daily.intake)
  0%  25%  50%  75% 100%
5260 5910 6515 7515 8770
```

You might wish to investigate whether the women's energy intake deviates systematically from a recommended value of 7725 kJ. Assuming that data come from a normal distribution, the object is to test whether this distribution might have mean $\mu = 7725$. This is done with `t.test` as follows:

```
> t.test(daily.intake,mu=7725)

         One Sample t-test

data:  daily.intake
t = -2.8208, df = 10, p-value = 0.01814
alternative hypothesis: true mean is not equal to 7725
95 percent confidence interval:
 5986.348 7520.925
sample estimates:
mean of x
 6753.636
```

This is an example of the exact same type as used in the introductory Section 1.1.4. The description of the output is quite superficial there. Here it is explained more thoroughly.

The layout is common to many of the standard statistical tests, and a "dissection" is given in the following:

```
        One Sample t-test
```

This should be self-explanatory. It is simply a description of the test that we have asked for. Notice that, by looking at the format of the function call, `t.test` has automatically found out that a one-sample test is desired.

```
data:  daily.intake
```

This tells us which data are being tested. Of course, this will be obvious *unless* output has been separated from the command that generated it. This can happen, for example, when using the `source` function to read commands from an external file.

```
t = -2.8208, df = 10, p-value = 0.01814
```

This is where it begins to get interesting. We get the t statistic, the associated degrees of freedom, and the exact p-value. We do not need to use a table of the t distribution to look up which quantiles the t-value can be found between. You can immediately see that $p < 0.05$ and thus that (using the customary 5% level of significance) data deviate significantly from the hypothesis that the mean is 7725.

```
alternative hypothesis: true mean is not equal to 7725
```

This contains two important pieces of information: (a) the value we wanted to test whether the mean could be equal to (7725 kJ) and (b) that the test is two-sided ("not equal to").

```
95 percent confidence interval:
 5986.348 7520.925
```

This is a 95% confidence interval for the true mean; that is, the set of (hypothetical) mean values from which the data do not deviate significantly. It is based on inverting the t test by solving for the values of μ_0 that cause t to lie within its acceptance region. For a 95% confidence interval, the solution is

$$\bar{x} - t_{0.975}(f) \times \text{SEM} < \mu < \bar{x} + t_{0.975}(f) \times \text{SEM}$$

```
sample estimates:
mean of x
 6753.636
```

This final item is the observed mean; that is, the (point) estimate of the true mean.

The function `t.test` has a number of optional arguments, three of which are relevant in one-sample problems. We have already seen the use of `mu`

to specify the mean value μ under the null hypothesis (default is `mu=0`). In addition, you can specify that a one-sided test is desired against alternatives greater than μ by using `alternative="greater"` or alternatives less than μ using `alternative="less"`. The third item that can be specified is the *confidence level* used for the confidence intervals; you would write `conf.level=0.99` to get a 99% interval.

Actually, it is often allowable to abbreviate a longish argument specification; for instance, it is sufficient to write `alt="g"` to get the test against greater alternatives.

5.2 Wilcoxon signed-rank test

The t tests are fairly robust against departures from the normal distribution especially in larger samples, but sometimes you wish to avoid making that assumption. To this end, the *distribution-free methods* are convenient. These are generally obtained by replacing data with corresponding order statistics.

For the one-sample Wilcoxon test, the procedure is to subtract the theoretical μ_0 and rank the differences according to their numerical value, ignoring the sign, and then calculate the sum of the positive or negative ranks. The point is that, assuming only that the distribution is symmetric around μ_0, the test statistic corresponds to selecting each number from 1 to n with probability 1/2 and calculating the sum. The distribution of the test statistic can be calculated exactly, at least in principle. It becomes computationally excessive in large samples, but the distribution is then very well approximated by a normal distribution.

Practical application of the Wilcoxon signed-rank test is done almost exactly like the t test:

```
> wilcox.test(daily.intake, mu=7725)
        Wilcoxon signed rank test with continuity correction

data:  daily.intake
V = 8, p-value = 0.0293
alternative hypothesis: true location is not equal to 7725

Warning message:
In wilcox.test.default(daily.intake, mu = 7725) :
  cannot compute exact p-value with ties
```

There is not quite as much output as from `t.test` due to the fact that there is no such thing as a parameter estimate in a nonparametric test and therefore no confidence limits, etc., either. It is, however, possible under

some assumptions to define a location measure and calculate confidence intervals for it. See the help files for `wilcox.test` for details.

The relative merits of distribution-free (or *nonparametric*) versus parametric methods such as the t test are a contentious issue. If the model assumptions of the parametric test are fulfilled, then it will be somewhat more efficient, on the order of 5% in large samples, although the difference can be larger in small samples. Notice, for instance, that unless the sample size is 6 or above, the signed-rank test simply cannot become significant at the 5% level. This is probably not too important, though; what is more important is that the apparent lack of assumptions for these tests sometimes misleads people into using them for data where the observations are not independent or where a comparison is biased by an important covariate.

The Wilcoxon tests are susceptible to the problem of *ties*, where several observations share the same value. In such cases, you simply use the average of the tied ranks; for example, if there are four identical values corresponding to places 6 to 9, they will all be assigned the value 7.5. This is not a problem for the large-sample normal approximations, but the exact small-sample distributions become much more difficult to calculate and `wilcox.test` cannot do so.

The test statistic V is the sum of the positive ranks. In the example, the p-value is computed from the normal approximation because of the tie at 7515.

The function `wilcox.test` takes arguments `mu` and `alternative`, just like `t.test`. In addition, it has `correct`, which turns a continuity correction on or off (the default is "on", as seen from the output title; `correct=F` turns it off), and `exact`, which specifies whether exact tests should be calculated. Recall that "on/off" options such as these are specified using logical values that can be either `TRUE` or `FALSE`.

5.3 Two-sample t test

The two-sample t test is used to test the hypothesis that two samples may be assumed to come from distributions with the same mean.

The theory for the two-sample t test is not very different in principle from that of the one-sample test. Data are now from two groups, $x_{11}, \ldots, x_{1n_1}$ and $x_{21}, \ldots, x_{2n_2}$, which we assume are sampled from the normal distributions $N(\mu_1, \sigma_1^2)$ and $N(\mu_2, \sigma_2^2)$, and it is desired to test the null hypothesis $\mu_1 = \mu_2$. You then calculate

$$t = \frac{\bar{x}_2 - \bar{x}_1}{\text{SEDM}}$$

where the *standard error of difference of means* is

$$\text{SEDM} = \sqrt{\text{SEM}_1^2 + \text{SEM}_2^2}$$

There are two ways of calculating the SEDM depending on whether or not you assume that the two groups have the same variance. The "classical" approach is to assume that the variances are identical. With this approach, you first calculate a pooled s based on the standard deviations from the two groups and plug that value into the SEM. Under the null hypothesis, the t value will follow a t distribution with $n_1 + n_2 - 2$ degrees of freedom.

An alternative procedure due to Welch is to calculate the SEMs from the separate group standard deviations s_1 and s_2. With this procedure, t is actually not t-distributed, but its distribution may be approximated by a t distribution with a number of degrees of freedom that can be calculated from s_1, s_2, and the group sizes. This is generally not an integer.

The Welch procedure is generally considered the safer one. Usually, the two procedures give very similar results unless both the group sizes and the standard deviations are very different.

We return to the daily energy expenditure data (see Section 1.2.14) and consider the problem of comparing energy expenditures between lean and obese women.

```
> attach(energy)
> energy
   expend stature
1    9.21   obese
2    7.53    lean
3    7.48    lean
...
20   7.58    lean
21   9.19   obese
22   8.11    lean
```

Notice that the necessary information is contained in two parallel columns of a data frame. The factor `stature` contains the group and the numeric variable `expend` the energy expenditure in mega-Joules. R allows data in this format to be analyzed by `t.test` and `wilcox.test` using a model formula specification. An older format (still available) requires you to specify data from each group in a separate variable, but the newer format is much more convenient for data that are kept in data frames and is also more flexible if you later want to group the same response data according to other criteria.

The object is to see whether there is a shift in level between the two groups, so we apply a t test as follows:

```
> t.test(expend~stature)

        Welch Two Sample t-test

data:  expend by stature
t = -3.8555, df = 15.919, p-value = 0.001411
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.459167 -1.004081
sample estimates:
 mean in group lean mean in group obese
           8.066154           10.297778
```

Notice the use of the tilde (~) operator to specify that `expend` is *described by* `stature`.

The output is not much different from that of the one-sample test. The confidence interval is for the *difference* in means and does not contain 0, which is in accordance with the *p*-value indicating a significant difference at the 5% level.

It is Welch's variant of the *t* test that is calculated by default. This is the test where you do not assume that the variance is the same in the two groups, which (among other things) results in the fractional degrees of freedom.

To get the usual (textbook) *t* test, you must specify that you are willing to assume that the variances are the same. This is done via the optional argument `var.equal=T`; that is:

```
> t.test(expend~stature, var.equal=T)

        Two Sample t-test

data:  expend by stature
t = -3.9456, df = 20, p-value = 0.000799
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.411451 -1.051796
sample estimates:
 mean in group lean mean in group obese
           8.066154           10.297778
```

Notice that the degrees of freedom now has become a whole number, namely $13 + 9 - 2 = 20$. The *p*-value has dropped slightly (from 0.14% to 0.08%) and the confidence interval is a little narrower, but overall the changes are slight.

5.4 Comparison of variances

Even though it is possible in R to perform the two-sample t test without the assumption that the variances are the same, you may still be interested in testing that assumption, and R provides the `var.test` function for that purpose, implementing an F test on the ratio of the group variances. It is called the same way as `t.test`:

```
> var.test(expend~stature)

        F test to compare two variances

data:  expend by stature
F = 0.7844, num df = 12, denom df =  8, p-value = 0.6797
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.1867876 2.7547991
sample estimates:
ratio of variances
          0.784446
```

The test is not significant, so there is no evidence against the assumption that the variances are identical. However, the confidence interval is very wide. For small data sets such as this one, the assumption of constant variance is largely a matter of belief. It may also be noted that this test is not robust against departures from a normal distribution. The `stats` package contains several alternative tests for variance homogeneity, each with its own assumptions, benefits, and drawbacks, but we shall not discuss them at length.

Notice that the test is based on the assumption that the groups are independent. You should not apply this test to paired data.

5.5 Two-sample Wilcoxon test

You might prefer a nonparametric test if you doubt the normal distribution assumptions of the t test. The two-sample Wilcoxon test is based on replacing the data by their rank (without regard to grouping) and calculating the sum of the ranks in one group, thus reducing the problem to one of sampling n_1 values without replacement from the numbers 1 to $n_1 + n_2$.

This is done using `wilcox.test`, which behaves similarly to `t.test`:

```
> wilcox.test(expend~stature)

        Wilcoxon rank sum test with continuity correction

data:  expend by stature
W = 12, p-value = 0.002122
alternative hypothesis: true location shift is not equal to 0

Warning message:
In wilcox.test.default(x = c(7.53, 7.48, 8.08, 8.09, 10.15, 8.4,  :
  cannot compute exact p-value with ties
```

The test statistic W is the sum of ranks in the first group minus its theoretical minimum (i.e., it is zero if all the smallest values fall in the first group). Some textbooks use a statistic that is the sum of ranks in the *smallest* group with no minimum correction, which is of course equivalent. Notice that, as in the one-sample example, we are having problems with ties and rely on the approximate normal distribution of W.

5.6 The paired t test

Paired tests are used when there are two measurements on the same experimental unit. The theory is essentially based on taking differences and thus reducing the problem to that of a one-sample test. Notice, though, that it is implicitly assumed that such differences have a distribution that is independent of the level. A useful graphical check is to make a scatterplot of the pairs with the line of identity added or to plot the difference against the average of the pair (sometimes called a *Bland–Altman plot*). If there seems to be a tendency for the dispersion to change with the level, then it may be useful to transform the data; frequently the standard deviation is proportional to the level, in which case a logarithmic transformation is useful.

The data on pre- and postmenstrual energy intake in a group of women are considered several times in Chapter 1 (and you may notice that the first column is identical to `daily.intake`, which was used in Section 5.1). There data are entered from the command line, but they are also available as a data set in the `ISwR` package:

```
> attach(intake)
> intake
    pre post
1  5260 3910
2  5470 4220
3  5640 3885
4  6180 5160
```

```
5  6390 5645
6  6515 4680
7  6805 5265
8  7515 5975
9  7515 6790
10 8230 6900
11 8770 7335
```

The point is that the same 11 women are measured twice, so it makes sense to look at individual differences:

```
> post - pre
 [1] -1350 -1250 -1755 -1020  -745 -1835 -1540 -1540  -725 -1330
[11] -1435
```

It is immediately seen that they are all negative. All the women have a lower energy intake postmenstrually than premenstrually. The paired *t* test is obtained as follows:

```
> t.test(pre, post, paired=T)

        Paired t-test

data:  pre and post
t = 11.9414, df = 10, p-value = 3.059e-07
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 1074.072 1566.838
sample estimates:
mean of the differences
               1320.455
```

There is not much new to say about the output; it is virtually identical to that of a one-sample *t* test on the elementwise differences.

Notice that you have to specify `paired=T` explicitly in the call, indicating that you want a paired test. In the old-style interface for the unpaired *t* test, the two groups are specified as separate vectors and you would request that analysis by omitting `paired=T`. If data are actually paired, then it would be seriously inappropriate to analyze them without taking the pairing into account.

Even though it might be considered pedagogically dubious to show what you should *not* do, the following shows the results of an unpaired *t* test on the same data for comparison:

```
> t.test(pre, post) #WRONG!

         Welch Two Sample t-test

data:  pre and post
t = 2.6242, df = 19.92, p-value = 0.01629
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  270.5633 2370.3458
sample estimates:
mean of x mean of y
 6753.636  5433.182
```

The number symbol (or "hash") `#` introduces a comment in R. The rest of the line is skipped.

It is seen that t has become considerably smaller, although still significant at the 5% level. The confidence interval has become almost four times wider than in the correct paired analysis. Both illustrate the loss of efficiency caused by not using the information that the "pre" and "post" measurements are from the same person. Alternatively, you could say that it demonstrates the gain in efficiency obtained by planning the experiment with two measurements on the same person, rather than having two independent groups of pre- and postmenstrual women.

5.7 The matched-pairs Wilcoxon test

The paired Wilcoxon test is the same as a one-sample Wilcoxon signed-rank test on the differences. The call is completely analogous to `t.test`:

```
> wilcox.test(pre, post, paired=T)
         Wilcoxon signed rank test with continuity correction

data:  pre and post
V = 66, p-value = 0.00384
alternative hypothesis: true location shift is not equal to 0

Warning message:
In wilcox.test.default(pre, post, paired = T) :
  cannot compute exact p-value with ties
```

The result does not show any material difference from that of the t test. The p-value is not quite so extreme, which is not too surprising since the Wilcoxon rank sum cannot get any larger than it does when all differences have the same sign, whereas the t statistic can become arbitrarily extreme.

Again, we have trouble with tied data invalidating the exact p calculations. This time it is the two identical differences of -1540.

In the present case it is actually very easy to calculate the exact p-value for the Wilcoxon test. It is the probability of 11 positive differences + the probability of 11 negative ones, $2 \times (1/2)^{11} = 1/1024 = 0.00098$, so the approximate p-value is almost four times too large.

5.8 Exercises

5.1 Do the values of the `react` data set (notice that this is a single vector, not a data frame) look reasonably normally distributed? Does the mean differ significantly from zero according to a t test?

5.2 In the data set `vitcap`, use a t test to compare the vital capacity for the two groups. Calculate a 99% confidence interval for the difference. The result of this comparison may be misleading. Why?

5.3 Perform the analyses of the `react` and `vitcap` data using nonparametric techniques.

5.4 Perform graphical checks of the assumptions for a paired t test in the `intake` data set.

5.5 The function `shapiro.test` computes a test of normality based on the degree of linearity of the Q–Q plot. Apply it to the `react` data. Does it help to remove the outliers?

5.6 The crossover trial in `ashina` can be analyzed for a drug effect in a simple way (how?) if you ignore a potential period effect. However, you can do better. Hint: Consider the intra-individual differences; if there were *only* a period effect present, how should the differences behave in the two groups? Compare the results of the simple method and the improved method.

5.7 Perform 10 one-sample t tests on simulated normally distributed data sets of 25 observations each. Repeat the experiment, but instead simulate samples from a different distribution; try the t distribution with 2 degrees of freedom and the exponential distribution (in the latter case, test for the mean being equal to 1). Can you find a way to automate this so that you can have a larger number of replications?