

# Appendix A

## Installing R and R-Commander

This appendix gives detailed instructions for installing R and R-Commander.

### A.1 Installing R

1. Go to <http://www.r-project.org/>.
2. Click on the download R link.
3. Then select a location closest to you.
4. Click on your operating system (Linux, MacOS X, Windows) and follow directions.

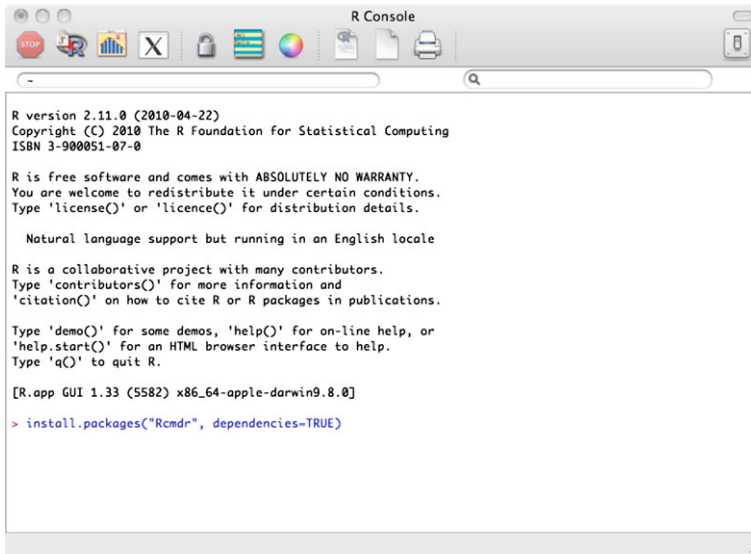
If you are a **Mac** user, download the latest .dmg file and follow the instructions. Once installed, R will appear as an icon in the Applications Folder. After you install R, you should go back to the same webpage (where you obtained the latest .dmg file), click on “tools”, which is located under “Subdirectories”, and install the universal build of **Tcl/Tk** for **X11**. The file name contains “tcltk”, three numbers representing the current version, and “x11.dmg”. (Currently, the file name is “tcltk-8.5.5-x11.dmg”.) This file includes additional tools necessary for building R for Mac OS X.

If you are running Windows, click on `base` and then on the link that downloads R for Windows. (In the link, the current version number appears after “R”.) When the dialog box opens, click `Run`, and a “Setup Wizard” should appear. Keep clicking `Next` until the Wizard is finished. Now, you should see an icon on your desktop, with a large capital R.

### A.2 Installing R-Commander

#### A.2.1 From the Command Line

You can download R-Commander from the command line by following these steps:



**Fig. A.1** Installing R-Commander by entering the command `install.packages("Rcmdr", dependencies=TRUE)` in R Console

1. Once you have installed R, open it by double-clicking on the icon.
2. A window called “R Console” will open.
3. Make sure you have a working internet connection. Then, at the prompt (the `>` symbol), type the following command exactly and then press enter (Fig. A.1):

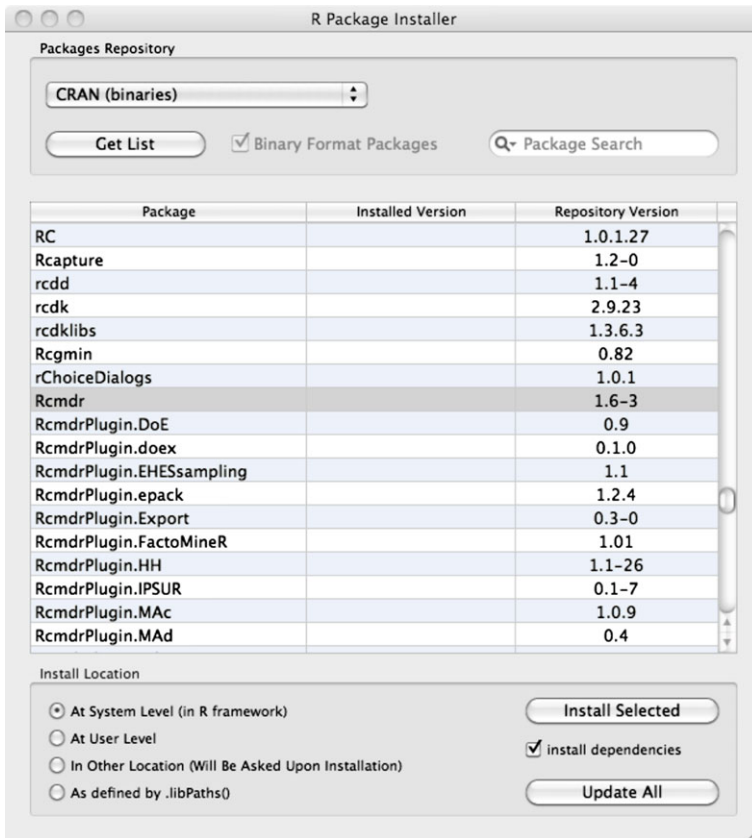
```
> install.packages("Rcmdr", dependencies = TRUE)
```

4. R may respond by asking you to select a mirror site and listing them in a pop-up box. Choose a nearby location.
5. Depending on your connection speed, the installation may take awhile. Be patient and wait until you see the prompt again before you do anything.

### A.2.2 From the Menu Bar

Alternatively, you can download R-Commander from the menu bar by following these steps:

1. Open R by clicking on its icon.
2. Click “Packages & Data” from the menu bar and then click “Package Installer”. This opens a window similar to Fig. A.2.
3. Click “Get List”.
4. Scroll down and select “Rcmdr”.



**Fig. A.2** Installing R-Commander using “Package Installer” from the menu bar

5. Check “Install dependencies” and click “Install Selected”.
6. R may respond by asking you to select a mirror site and listing them in a pop-up box. Choose a nearby location.
7. The installation may take awhile. Wait until you see the prompt again before you do anything.

## A.3 Starting R-Commander

If R is not already open, open it by clicking on its icon. To open R-Commander, at the prompt enter the following command (Fig. A.3):

```
> library(Rcmdr)
```

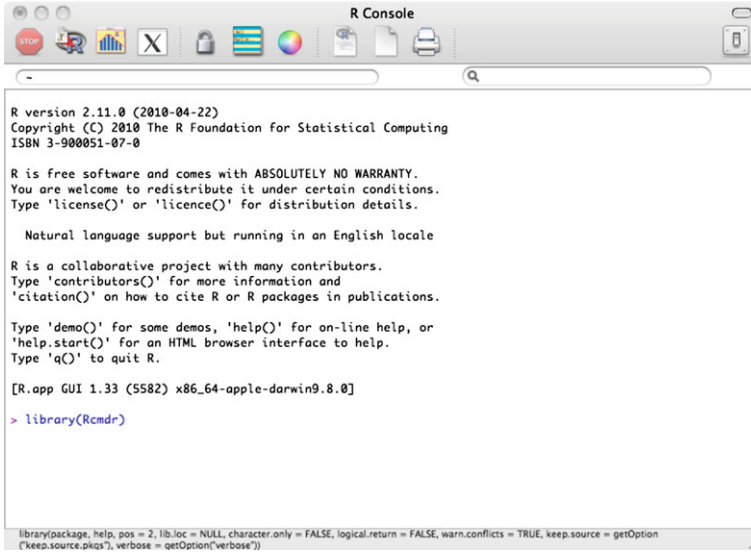


Fig. A.3 Opening R-Commander by entering the command library(Rcmdr) in R Console

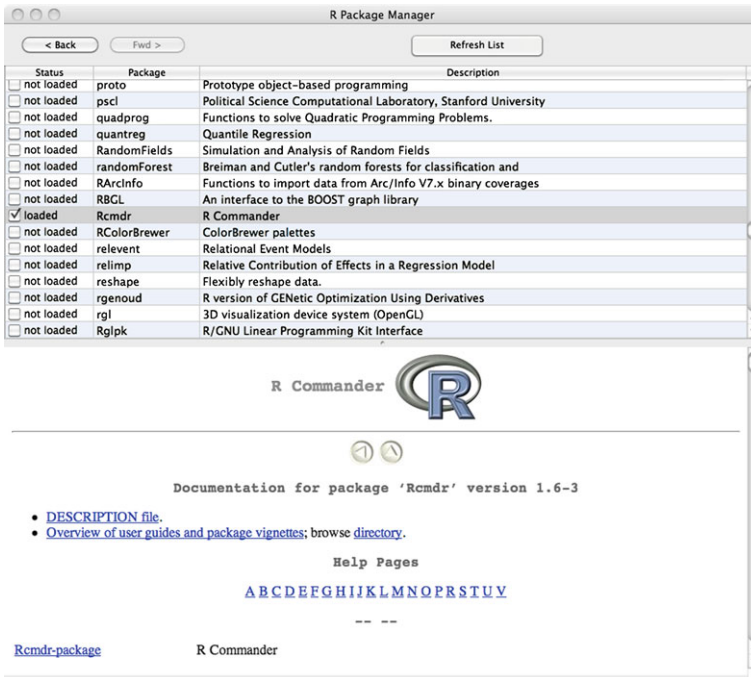


Fig. A.4 The “Package Manager” window

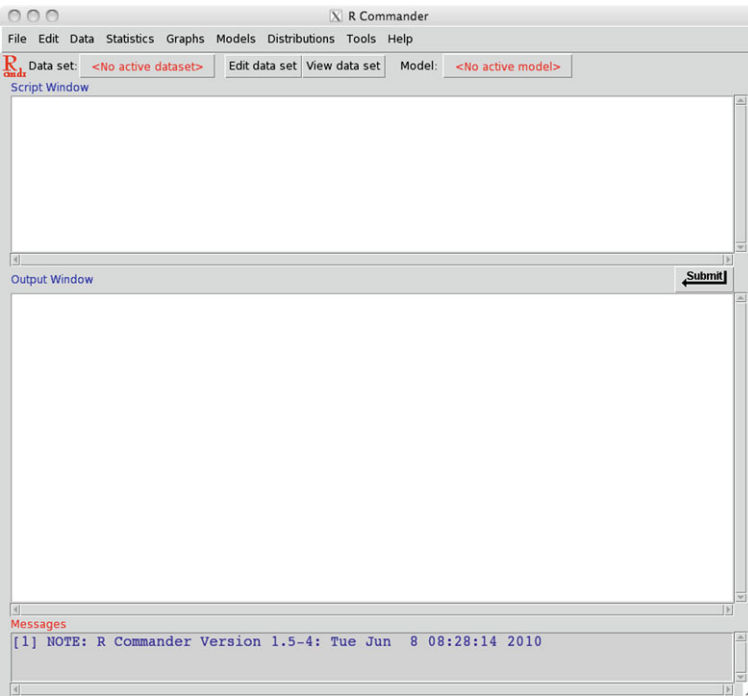


Fig. A.5 The R-Commander window

Alternatively, you can load R-Commander by clicking “Packages & Data” from the menu bar and then clicking “Package Manager”. This will open a window similar to Fig. A.4. Scroll down to find R-Commander (“Rcmdr” package) and check its box to change the status from “not loaded” to “loaded”.

You should see a large new window pop-up, labeled R-Commander (Fig. A.5). You are now ready to analyze your data. If you close this window while R is still open, you can start R-Commander again by entering the command `Commander()` in R Console. (Entering `library(Rcmdr)` in this situation will not work unless you close R and open it again.) Alternatively, you can use the “Package Manager” window to unload and reload R-Commander.

If you are a Mac user and your OS version is 10.4.11 or lower, you might need to open X11 manually before entering the command `library(Rcmdr)` in R Console. To do this, under `Misc` from the top bar menu choose `Run X11 Server`. If you do not have X11, you can install it from your install DVD or find it online.

## Appendix B

### Basic R

This appendix gives a brief introduction to R language for those who want to use R (instead of R-Commander) for data analysis. The introduction given here is meant to help with the initial steps toward this goal. To use R more broadly and more effectively, one needs to refer to more advanced references for R programming as well as reading the advanced sections in this book.

#### B.1 Starting with R

In the R Console, you can type commands directly at the prompt, `>`, and execute them by pressing enter. Commands can also be entered in the *Script* window in R-Commander and executed by pressing the “Submit” button. Both the R Console and R-Commander provide an interactive environment where the results are immediately shown similar to a calculator. In fact, R can be used as a calculator. The basic arithmetic operators are `+` for addition, `-` for subtraction, `*` for multiplication, and `/` for division. The `^` operator is used to raise a number (or a variable) to a power. Try executing the following commands:

```
> 65 + 32
```

```
[1] 97
```

```
> 3 * 1.7 - 2
```

```
[1] 3.1
```

```
> 4 * 3 + 6/0.2
```

```
[1] 42
```

```
> 5^2
```

```
[1] 25
```

There are also built in functions for finding the square root `sqrt()`, the exponential `exp()`, and the natural logarithm `log()`:

```
> sqrt(430)
[1] 20.73644
> exp(-1.3)
[1] 0.2725318
> log(25)
[1] 3.218876
```

Here, the numbers in the parentheses serve as input (parameters or arguments) to the functions. Most functions have multiple parameters and options. For example, to take the base-10 logarithm of 25, we include the option `base=10`:

```
> log(25, base = 10)
[1] 1.39794
```

In general, you can always learn more about a function and its options (arguments) by writing its name after a question mark (e.g., `?log`).

We can combine two or more functions such that the output from one function becomes the input for another function. For example, the following code combines the above `log()` function with the `round()` function, which is used to specify the number of decimal places (here, two digits):

```
> round(log(25, base = 10), digits = 2)
[1] 1.4
```

In the above code, the output of `log(25, base=10)` becomes the first argument of the function `round()`. The second argument, `digits=2`, specifies the number of decimal places.

## B.2 Creating Objects in R

Instead of directly entering commands such as `2+3`, we can create *objects* to hold values and then perform operations on these objects. For example, the following set of commands creates two objects `x` and `y`, adds the values stored in these objects, and assigns the result to the third object `z`:

```
> x <- 2
> y <- 3
> z <- x + y
```

In general, we use left arrow `<-` (i.e., type `<` and then `-`) to assign values to an object. Almost always, we can use the equal sign “=” instead of `<-` for assignment. For example, we could use `y = 3` to assign the value 3 to `y`, and use `z = x+y` to set `z` equal to the sum of `x` and `y`. The two options, “=” and `<-`, have some small differences, which are not discussed here.

Simply typing the name of an object displays its contents. We could also use the function `print()`:

```
> x
[1] 2
> print(y)
[1] 3
> print(z)
[1] 5
```

Object names are case sensitive. For example, `x` and `X` are two different objects. A name cannot start with a digit or an underscore `_` and cannot be among the list of reserved words such as `if`, `function`, `NULL`. We can use the period `.` in a name to separate words (e.g., `my.object`).

The objects are created and stored by their names. We can obtain the list of objects that are currently stored within R by using the function `objects()`:

```
> objects()
[1] "x" "y" "z"
```

The collection of these objects is called the *workspace*. (Note that although we are not specifying the values of arguments, we still need to type parentheses when using functions.) When closing an R session, you have the opportunity to save the objects permanently for future use. This way, the workspace is saved in a file called `.RData`, which will be restored next time you open R. If you want to save only few objects, as opposed to the entire workspace, you can use the function `save()`. For example, suppose that we only want to save the objects `x` and `y`:

```
> save(x, y, file = "myObjects.RData")
```

The above command saves `x` and `y` in a file called `myObjects.RData` in the *current working directory*. You can see where the current working directory is by



entering the command `getwd()`. If you want to save your objects in another directory, either enter the full path when specifying the file name in the `save()` function or use the menu bar to change the directory. (For Mac, this option is located under “Misc”. For Windows, it is located under “File”.)

After you save the `x` and `y` in `myObjects.RData`, you can load these objects for future use with the `load()` function:

```
> load("myObjects.RData")
```

Give the full address for the file if it is not located in the current working directory. Alternatively, you can change the working directory from the menu bar.

### B.3 Vectors

Using objects allows for more flexibility. For example, we can store more than one value in an object and apply a function or an operation to its contents. The following commands create a *vector* object `x` that contains numbers 1 through 5 and then apply two different functions to it:

```
> x <- c(1, 2, 3, 4, 5)
> x
```

```
[1] 1 2 3 4 5
```

```
> 2 * x + 1
```

```
[1] 3 5 7 9 11
```

```
> exp(x)
```

```
[1] 2.718282 7.389056 20.085537 54.598150
[5] 148.413159
```

The `c()` function is used to combine its arguments (here, integers from 1 to 5) into a vector. Since `1, 2, ..., 5` is a sequence of consecutive numbers, we could simply use the colon “`:`” operator to create the vector:

```
> x <- 1:5
> x
```

```
[1] 1 2 3 4 5
```

To create sequences and store them in vector objects, we can also use the `seq()` function for additional flexibility. The following commands create a vector object `y` containing a sequence increasing by 2 from `-3` to `14`:

```
> y <- seq(from = -3, to = 14, by = 2)
> y

[1] -3 -1  1  3  5  7  9 11 13
```

If the elements of a vector are all the same, we can use the `rep()` function:

```
> z <- rep(5, times = 10)
> z

[1] 5 5 5 5 5 5 5 5 5 5
```

The following function creates a vector of size 10 where all its elements are unknown. In R, missing values are represented by NA (Not Available):

```
> z <- rep(NA, times = 10)
> z

[1] NA NA NA NA NA NA NA NA NA NA
```

This way, we can create a vector object of a given length and specify its elements later.

To find the length of a vector (i.e., number of elements), we use the `length()` command:

```
> length(x)

[1] 5

> length(y)

[1] 9
```

Functions `sum()`, `mean()`, `min()`, and `max()` return the sum, average, minimum, and maximum for a vector:

```
> x

[1] 1 2 3 4 5

> sum(x)

[1] 15

> mean(x)

[1] 3
```

```
> min(x)
[1] 1
> max(x)
[1] 5
```

The elements of a vector can be accessed by providing their index using square brackets [ ]. For example, try retrieving the first element of `x` and the 4th element of `y`:

```
> x[1]
[1] 1
> y[4]
[1] 3
```

The colon `:` operator can be used to obtain a sequence of elements. For instance, elements 3 through 6 of `y` can be accessed with

```
> y[3:6]
[1] 1 3 5 7
```

To select all but the 4th element of a vector, we use negative indexing:

```
> y[-4]
[1] -3 -1 1 5 7 9 11 13
```

The above objects are all *numerical* vectors. A vector can also hold character strings delimited by single or double quotations marks. For example, suppose that have a sample of 5 patients. We can create a *character* vector storing their gender as

```
> gender <- c("male", "female", "female", "male",
+            "female")
```

(The plus sign means that the second line is the continuation of the command in the first line. You should not type it when trying the above command in R.) Retrieving the elements of the vector is as before:

```
> gender[3]
[1] "female"
```

A vector could also be *logical*, where the elements are either TRUE or FALSE (NA if the element is missing). Note that these values must be in capital letters and can be abbreviated by T and F (not recommended). For example, create a vector storing the health status of the five patients:

```
> is.healthy <- c(TRUE, TRUE, FALSE, TRUE, FALSE)
> is.healthy

[1] TRUE TRUE FALSE TRUE FALSE
```

When used in ordinary arithmetic, logical vectors are coerced to integer vectors, 0 for FALSE elements and 1 for TRUE elements. For example, applying the `sum()` function to the above logical vector turns the TRUE and FALSE values to ones and zeros, and returns their sum, which in this case is equivalent to the number of healthy subjects:

```
> sum(is.healthy)

[1] 3
```

Use the `as.integer()` function to see the equivalent integer vector for `is.healthy`:

```
> as.integer(is.healthy)

[1] 1 1 0 1 0
```

Logical vectors are usually derived from other vectors using logical operators. For example, with the `gender` vector, we can create a logical vector showing which subjects are male:

```
> gender

[1] "male" "female" "female" "male" "female"

> is.male <- (gender == "male")
> is.male

[1] TRUE FALSE FALSE TRUE FALSE
```

Here, `==` (i.e., two equal signs) is a relational operator that returns TRUE if the two sides are equal and returns FALSE otherwise. As the second example, we create a numerical vector for the age of the five subjects and then check to see which person is 60 years old:

```
> age <- c(60, 43, 72, 35, 47)
> is.60 <- (age == 60)
```

```
> is.60
```

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

The `!=` operator, on the other hand, returns a TRUE value when the two sides are not equal:

```
> is.female <- (gender != "male")
> is.female
```

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

```
> not.60 <- (age != 60)
> not.60
```

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

The other relational operators commonly applied to numerical vectors are “less than”, `<`, “less than or equal to”, `<=`, “greater than”, `>`, “greater than or equal to”: `>=`

```
> age < 43
```

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

```
> age <= 43
```

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

```
> age > 43
```

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

```
> age >= 43
```

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

We can also use *Boolean* operators to create new logical vectors based on existing ones. The logical NOT, `!`, negates the elements of a logical vector (i.e., changes TRUE to FALSE and vice versa). For example, create a `is.female` vector from the `is.male` vector:

```
> is.female <- !is.male
> is.female
```

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

The logical AND, `&`, compares the elements of two logical vectors and returns TRUE only when the corresponding elements are both TRUE:

```

> is.male
[1] TRUE FALSE FALSE TRUE FALSE
> is.healthy
[1] TRUE TRUE FALSE TRUE FALSE
> is.male & is.healthy
[1] TRUE FALSE FALSE TRUE FALSE

```

The logical OR, `|`, also compares the elements of two logical vectors and returns TRUE when at least one of the corresponding elements is TRUE:

```

> is.male | is.healthy
[1] TRUE TRUE FALSE TRUE FALSE

```

We can use combinations of two or more logical operators. The following commands check to see which subjects are male and less than 45 years old:

```

> is.young.male <- is.male & (age < 45)
> is.young.male
[1] FALSE FALSE FALSE TRUE FALSE

```

Using the `which()` function, we can obtain the indices of TRUE elements for a given logical function:

```

> ind.male <- which(is.male)
> ind.male
[1] 1 4
> ind.young <- which(age < 45)
> ind.young
[1] 2 4

```

We can then use these indices to obtain their corresponding elements from a vector:

```

> age[ind.male]
[1] 60 35
> is.male[ind.young]
[1] FALSE TRUE

```

We can combine the two steps:

```
> age[is.male]
[1] 60 35
> age[gender == "male"]
[1] 60 35
> is.male[age < 45]
[1] FALSE TRUE
> gender[age < 45]
[1] "female" "male"
```

## B.4 Matrices

In the above examples, we used one vector for each characteristic (e.g., age, health status, gender) of our five subjects. It is easier, of course, to store the subject information in a table format, where each row corresponds to an individual and each column to a characteristic. If all these measurements are of the same type (e.g., numerical, character, logical), a *matrix* can be used. For example, besides age, assume that for our five subjects, we have also measured BMI (body mass index) and blood pressure:

```
> BMI = c(28, 32, 21, 27, 35)
> bp = c(124, 145, 127, 133, 140)
```

Now create a matrix with the `cbind()` function for column-wise binding of age, BMI, and bp:

```
> data.1 = cbind(age, BMI, bp)
> data.1
      age BMI  bp
[1,]  60  28 124
[2,]  43  32 145
[3,]  72  21 127
[4,]  35  27 133
[5,]  47  35 140
```

If we had wanted a matrix where each row represented a characteristic and each column a subject, we would have used the `rbind()` function for row-wise binding:

```
> data.2 = rbind(age, BMI, bp)
> data.2

      [,1] [,2] [,3] [,4] [,5]
age    60   43   72   35   47
BMI    28   32   21   27   35
bp    124  145  127  133  140
```

We could obtain `data.2` by transposing (e.g., interchanging the rows and columns) `data.1` using the `t()` function:

```
> t(data.1)

      [,1] [,2] [,3] [,4] [,5]
age    60   43   72   35   47
BMI    28   32   21   27   35
bp    124  145  127  133  140
```

In general, matrices are two-dimensional objects comprised of values of the same type. The object `data.1` is a  $5 \times 3$  matrix. The function `dim` returns the size (i.e., the number of rows and columns) of a matrix:

```
> dim(data.1)

[1] 5 3
```

When creating the matrix `data.1`, R automatically uses the vector names as the column names. They can be changed or accessed with the function `colnames()`:

```
> colnames(data.1)

[1] "age" "BMI" "bp"
```

Likewise, we can obtain or provide the row names using the function `rownames()`:

```
> rownames(data.1) <- c("subject1", "subject2",
+ "subject3", "subject4", "subject5")
> data.1

      age BMI bp
subject1 60 28 124
subject2 43 32 145
subject3 72 21 127
subject4 35 27 133
subject5 47 35 140
```



To access the elements of a matrix, we still use square brackets [ ], but this time, we have to provide both the row index and the column index. For instance, the age of the third subject is

```
> data.1[3, 1]

[1] 72
```

If only a row number is provided, R returns all elements of that row (e.g., all the measurements for one subject):

```
> data.1[2, ]

age BMI bp
43 32 145
```

Likewise, if only a column is specified, R returns all elements of that column (e.g., all the measurements for one characteristic):

```
> data.1[, 2]

subject1 subject2 subject3 subject4 subject5
28        32        21        27        35
```

A matrix can also be created by rearranging the elements of a vector with the `matrix` function:

```
> matrix(data = 1:12, nrow = 3, ncol = 4)

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Here, the argument `data` specifies the numbers (vector), whose rearrangement creates the matrix. The arguments `nrow` and `ncol` are the number of rows and columns, respectively. By default, the matrix is filled by columns. To fill the matrix by rows, we must use the argument `byrow=TRUE`:

```
> matrix(data = 1:12, nrow = 3, ncol = 4, byrow = TRUE)

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

The length of `data` is usually equal to  $nrow \times ncol$ . If there are too few elements in `data` to fill the matrix, then the elements are recycled. In the following example, all elements of the matrix are set to zero:

```
> mat <- matrix(data = 0, nrow = 3, ncol = 3)
> mat

      [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
```

We can obtain or set the diagonal elements of a matrix using the `diag()` function:

```
> diag(mat) <- 1
> mat

      [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Similar to vectors, we can create a matrix with missing values (NA) and specify the elements later:

```
> mat <- matrix(data = NA, nrow = 2, ncol = 3)
> mat

      [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> mat[1, 3] <- 5
> mat

      [,1] [,2] [,3]
[1,]   NA   NA    5
[2,]   NA   NA   NA
```

## B.5 Data Frames

In general, we use matrices when all measurements are of the same type (e.g., numerical, character, logical). Otherwise, the type of measurements could change when we mix different types. In the following example, we show this by using the `mode()` function that returns the type for a given object:

```
> mat <- cbind(age, gender, is.healthy)
> mode(mat)

[1] "character"
```

In the above example, the matrix type is character, although `age` is numeric and `is.healthy` is logical. Note that their values will be printed in quotation marks, which is used for character strings, if you enter `mat`.

To avoid this issue, we can store data with information of different types in a table format similar to the format of spreadsheets. The resulting object (which includes multiple objects of possibly different types) is called a *data frame* object. For this, we use the function `data.frame()`:

```
> data.df = data.frame(age, gender, is.healthy,
+   BMI, bp)
> data.df
```

	age	gender	is.healthy	BMI	bp
1	60	male	TRUE	28	124
2	43	female	TRUE	32	145
3	72	female	FALSE	21	127
4	35	male	TRUE	27	133
5	47	female	FALSE	35	140

To create a data frame this way, all vectors must have the same length. You may notice that while `gender` is a character vector, its elements are not printed with quotation marks as before. The reason is that character vectors included in data frames are coerced to a different type called *factors*. A factor is a vector object that is used to provide a compact way to represent categorical data. Each *level* of a factor vector represents a unique category (e.g., female or male). We can directly create factors using the `factor()` function:

```
> factor(gender)
```

```
[1] male   female female male   female
Levels: female male
```

To access elements of a data frame, we use the square brackets `[ , ]` with the appropriate row and column indices. For example, the BMI of the 3rd subject is

```
> data.df[3, 4]
```

```
[1] 21
```

As before, we can access an entire row (e.g., all the measurements for one subject) by only specifying the row index and an entire column (e.g., all the measurements for one variable) by only specifying the column index. We can also access an entire column by providing the column name:

```
> data.df[, "age"]
```

```
[1] 60 43 72 35 47
```

The `$` operator also retrieves an entire column from the data frame:

```
> data.df$age
[1] 60 43 72 35 47
```

This column can then be treated as a vector and its elements accessed with the square brackets as before. For instance, try obtaining the BMI for the 3rd subject and the gender of the 2nd subject:

```
> data.df$BMI[4]
[1] 27

> data.df$gender[2]
[1] female
Levels: female male
```

### ***B.5.1 Creating Data Frames Using a Spreadsheet-Like Environment***

We can create a data frame by invoking a spreadsheet-like environment in R. For this, we start by creating an empty data frame object:

```
> new.df <- data.frame()
```

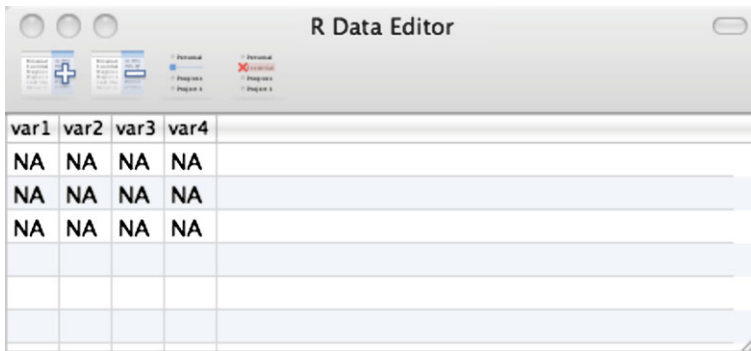
Then, we use the function `fix()` to edit the newly created data frame `new.df`.

```
> fix(new.df)
```

This way, R opens a window for data editing similar to Fig. B.1. You can use the four icons at the top of the editor to add or delete columns or rows. (Hover your mouse pointer over each icon to see its function.) For `new.df`, we have created four columns and three rows. The column names and the content of the data frame can be edited the same way we edit spreadsheets.

### ***B.5.2 Importing Data from Text Files***

Data are usually available in a tabular format as a delimited text file. We can import the contents of such files into R using the function `read.table()`. For instance, let us try importing the `BodyTemperature` data set (this data set is available online from the book website <http://extras.springer.com>):



**Fig. B.1** R window for data editing invoked by applying the function `fix()` to an empty data frame object

```
> BodyTemperature <- read.table(
+   file = "BodyTemperature.txt",
+   header = TRUE, sep = " ")
```

Here, we are using the `read.table()` function with three arguments. The first argument, `file="BodyTemperature.txt"`, specifies the name and location of the data file. If the file is not in the current working directory, you need to give the full path to the file or change the working directory. The `header=TRUE` option tells R that the variable names are contained in the first line of the data. Set this option to `FALSE` when this is not the case. The `sep` option tells R how the columns are separated in the text file. In this example, the columns are separated by white spaces. If the columns were separated by commas, for example, we would have used `sep=" , "`.

The `BodyTemperature` object is a data frame holding the contents of the "BodyTemperature.txt" file. Type `BodyTemperature` to view the entire data set. If the data set is large, it is better to use the `head()` function, which shows only the first part (few rows) of the data set:

```
> head(BodyTemperature)

  Gender Age HeartRate Temperature
1      M  33         69          97.0
2      M  32         72          98.8
3      M  42         68          96.2
4      F  33         75          97.8
5      F  26         68          98.8
6      M  37         79         101.3
```

In the `BodyTemperature` data frame, the rows correspond to subjects and the columns to variables. To view the names of the columns, try

```
> names(BodyTemperature)

[1] "Gender"      "Age"          "HeartRate"
[4] "Temperature"
```

Accessing observations in the `BodyTemperature` data frame is the same as before. We can use square brackets `[ , ]` with the row and column indices or the `$` operator with the variable name:

```
> BodyTemperature[1:3, 2:4]

  Age HeartRate Temperature
1  33         69         97.0
2  32         72         98.8
3  42         68         96.2

> BodyTemperature$Age[1:3]

[1] 33 32 42
```

## B.6 Lists

The data frames we created above (either directly or by importing a text file) include vectors of different types, but all the vectors have the same length. To combine objects of different types and possibly with different length into one object, we use *lists* instead. For example, suppose that we want to store the above body temperature data along with the name of investigators and students who have been involved in the study. We can create a list as follows:

```
> our.study <- list(data = BodyTemperature,
+   investigators = c("Smith", "Jackson",
+   "Clark"), students = c("Steve", "Mary"))
> length(our.study)

[1] 3
```

We have created a list with three *components*: `data`, `investigators`, and `students`. Each component has a name, which can be used to access that component:

```
> our.study$investigator

[1] "Smith"      "Jackson"    "Clark"
```

The components are ordered, so we can access them using double square brackets `"[[ ]]`:

```
> our.study[[2]]
[1] "Smith" "Jackson" "Clark"
```

If the component is a matrix or a vector, we can access its individual elements as before:

```
> our.study[[2]][3]
[1] "Clark"

> our.study[[1]][2:4, ]
  Gender Age HeartRate Temperature
2      M  32         72         98.8
3      M  42         68         96.2
4      F  33         75         97.8
```

## B.7 Loading Add-on Packages

A package includes a set of functions that are commonly used for a specific application of statistical analysis. R users have been creating new packages and making them publicly available on CRAN (Comprehensive R Archive Network). To use a package, you first need to download it from CRAN and install it in your local R library. For this, we can use the `install.packages()` function. For example, suppose that we want to perform Biodemographic analysis. The “Biodem” package, which is created by Boattini et al., provides a number of functions for this purpose. The following command downloads the `Biodem` package:

```
> install.packages("Biodem", dependencies = TRUE)
```

The first argument specifies the name of the package, and by setting the option `dependencies` to “TRUE”, we install all other packages on which “Biodem” depends. The reference manual for this package is available at <http://cran.r-project.org/web/packages/Biodem/Biodem.pdf>.

After we install a package, we need to load it in R in order to use it. For this, we use the `library()` command:

```
> library(Biodem)
```

Now we can use all the functions available in this package. We can also use all the data sets included in the package. For example, the `Biodem` package includes a data set called `valley` where every row corresponds to a different marriage record. To use this data set, we enter the following command:

```
> data(valley)
```

The data set becomes available in the workspace as a data frame.

One of the most widely used packages in R is the MASS package. This package is automatically installed when you install R. However, you still need to load it before you can use it. Enter the command `library(MASS)` to load this package. One of the data sets available in the MASS package is the `birthwt` data set, which includes the birthweight, `bwt`, of newborn babies.

## B.8 Conditional Statements

The `birthwt` data set from the MASS package includes a binary variable, `low`, that indicates whether the baby had low birthweight. Low birthweight is defined as having birthweight lower than 2500 grams (2.5 kilograms). Suppose that we did not have this variable and we wanted to create it. First, let us load the `birthwt` data set into R:

```
> data(birthwt)
> dim(birthwt)

[1] 189  10
```

The data set includes 189 cases and 10 variables.

We now create an empty vector, called `low`, of size 189:

```
> low <- rep(NA, 189)
```

Alternatively, we could use create an empty numerical vector without specifying its length using the `numeric()` function:

```
> low <- numeric()
```

Note that this way we specify the type of the object. We would have used `character()`, or `data.frame()`, or `list()`, if we wanted to create an empty object of the type character, or data frame, or list.

Now we want to examine the birthweight of each baby, and *if* it is below 2500, we assign the value of “1” to the `low` variable; otherwise, we assign the value “0”. The general format for an `if()` statement is

```
> if (condition) {
+   expression
+ }
```

If the `condition` is true, R runs the commands represented by `expression`. Otherwise, R skips the commands within the brackets `{ }`.

Try an `if()` statement to set the `low` of the first observation:



```
> if (birthwt$bwt[1] < 2500) {
+   low[1] <- 1
+ }
```

Check the result:

```
> birthwt$bwt[1]
```

```
[1] 2523
```

```
> low[1]
```

```
[1] NA
```

Since the condition was not true (i.e., bwt is not below 2500), the expression was not executed. To assign the value “0”, we can use an else statement along with the above if statement. The general format for if-else() statements is

```
> if (condition) {
+   expression1
+ } else {
+   expression2
+ }
```

If the condition is true, R runs the commands represented by expression1; otherwise, R runs the commands represented by expression2. For example, we can use the following code to decide whether the first baby in the birthwt data has low birthweight or not:

```
> if (birthwt$bwt[1] < 2500) {
+   low[1] <- 1
+ } else {
+   low[1] <- 0
+ }
> birthwt$bwt[1]
```

```
[1] 2523
```

```
> low[1]
```

```
[1] 0
```

Conditional statements can have multiple else statements to test multiple conditions:

```
> if (condition1) {
+   expression1
+ } else if (condition2) {
```

```
+     expression2
+ } else {
+     expression3
+ }
```

## B.9 Loops

To apply the above conditional statements to all observations, we can use a `for()` loop, which has the general format

```
> for (i in 1:n) {
+     expression
+ }
```

Here,  $i$  is the loop counter that takes values from 1 through  $n$ . The expression within the loop represents the set of commands to be repeated  $n$  times. For example, the following R commands create the vector  $y$  based on the vector  $x$  one element at a time:

```
> x <- c(3, -2, 5, 6)
> y <- numeric()
> for (i in 1:4) {
+     y[i] <- x[i] + 2 * i
+ }
> y

[1] 5 2 11 14
```

For the example discussed in the previous section, we use the following loop:

```
> for (i in 1:189) {
+     if (birthwt$bwt[i] < 2500) {
+         low[i] <- 1
+     }
+     else {
+         low[i] <- 0
+     }
+ }
```

The counter starts from 1 (i.e., the first row), and it ends at 189 (i.e., the last row). At each iteration, evaluate the conditional expression `birthwt$bwt[i] < 2500`. If the expression is true, it sets the value of `low` for that row to 1, otherwise, it sets it to 0. The variable `low` you created using the above loop and conditional statements will be exactly the same as the existing variable `low` in the data frame `birthwt`.

## B.10 Creating Functions

So far, we have been using R to perform specific tasks by creating objects and applying functions to them. If we need to repeat the same task over and over again under different settings, a more efficient approach would be to create a *function* which can be called repeatedly. The function we create is itself an object and is similar to existing functions in R, such as `sum()`, `log()`, and `matrix()`, that we have been using. The general form of creating a function is as follows:

```
> fun.name <- function(arg1, arg2, ...) {
+   expression1
+   expression2
+   ...
+   return(list = c(out1 = output1,
+                   out2 = output2, ...))
+ }
```

For example, suppose that we routinely need to find the min and max for a given numerical vector and print the sum of its elements. Also, we need to round the elements of the vector. However, the number of decimal places could be different for different vectors. Instead of writing the codes to create the function in *R Console*, it is better to write it in a file using a text editor so that we can modify it later. For this, click `File` → `New Document` from the menu bar. This will open a text editor. Now we can type the following commands in the text editor to create our function (Fig. B.2):

```
> my.fun <- function(x, n.digits = 1) {
+   min.value <- min(x)
+   max.value <- max(x)
+   print(sum(x))
+   y <- round(x, digits = n.digits)
+   return(list(min.value = min.value,
+               max.value = max.value, rounded.vec = y))
+ }
```

The above function takes two inputs: a numerical vector `x` and the number of decimal places `n.digits`. For the number of decimal places, we set the default to 1. If the user does not specify the number of decimal points, the function uses the default value. For `x`, there is no default value, so we need to provide its value every time we use this function.

The function then creates two objects, `min.value` and `max.value`, that store the min and max of `x`, respectively. Next, the function prints the sum of all elements. Finally, the function creates a new vector called `y`, which contains the rounded values of the original vector to the number of decimal place specified by `n.digits`.

Using `return()`, we specify the outputs of the function as a list. In this case, the list has three components. The first component is called “min.value”, and it contains the value of the object `min.value`. The second component is called

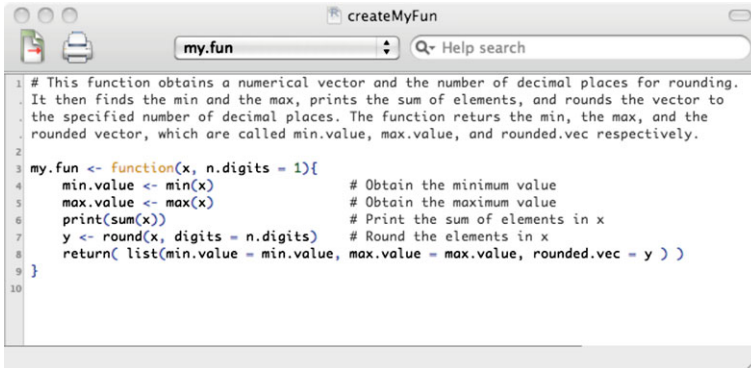


Fig. B.2 Creating a function called `my.fun()` using the text editor in R

“max.value”, and it contains the value of the object `max.value`. The last component is called “rounded.vec”, and it contains the new vector `y`, which was created by rounding the values of the original vector.

Note that in Fig. B.2, we wrote some comments in the text editor to explain what the function does. The comments should be always preceded by the symbol “#”. R regards what we write after “#” as comments and does not execute them.

When we finish typing the commands required to create the function, we save the file by clicking `File` → `Save As`. When prompted, choose a name for your file. For example, we called our file “CreateMyFun.R”. The file will have the “.R” extension.

So far, we have just created a file that contains the command necessary to create the function. The function has not been created yet. To create the function, we need to execute the commands. For this, we can use the `source()` function to read (evaluate) the codes from the “CreateMyFun.R” file:

```
> source("CreateMyFun.R")
```

Again, give the full address for the file if it is not located in the current working directory. We can now use our function the same way we have been using any other function. The following is an example:

```
> out <- my.fun(x = c(1.2, 2.4, 5.7), n.digits = 0)
```

```
[1] 9.3
```

```
> out
```

```
$min.value
```

```
[1] 1.2
```

```
$max.value
```

```
[1] 5.7
```

```
$rounded.vec
```

```
[1] 1 2 6
```

When we run the function, it prints the sum of all elements, which is 9.3, as we requested. The outputs will be assigned to a new object called “out”. Since the output was a list, `out` will be a list, and we can print its contents by using their names.

# References

1. Agresti, A.: *Categorical Data Analysis*. Wiley, New York (2002)
2. Aitchison, J., Dunsmore, I.R.: *Statistical Prediction Analysis*. Cambridge University Press, Cambridge (1975)
3. Allen, D.M.: Mean square error of prediction as a criterion for selecting. *Technometrics* **13**(3), 469–475 (1971)
4. Buijsse, B., Weikert, C., Drogan, D., Bergmann, M., Boeing, H.: Chocolate consumption in relation to blood pressure and risk of cardiovascular disease in German adults. *Eur. Heart J.* **31**(13), 1616–1623 (2010)
5. Cox, D.R., Hinkley, D.V.: *Theoretical Statistics*. Chapman and Hall, London (1974)
6. Christensen, R., Johnson, W., Branscum, A., Hanson, T.E.: *Bayesian Ideas and Data Analysis: An Introduction for Scientists and Statisticians*. Texts in Statistical Science. Taylor and Francis, London (2010)
7. Fox, J.: The R Commander: a basic-statistics graphical user interface to R. *J. Stat. Softw.* **14**, 1–42 (2005)
8. Gelman, A., Carlin, J.B., Stern, H.S., Rubin, D.B.: *Bayesian Data Analysis*. Chapman and Hall, London (2003)
9. Hand, D.J., Daly, F., McConway, K., Lunn, D., Ostrowski, E.: *A Handbook of Small Data Sets*, 1st edn. Chapman & Hall Statistics Texts. Chapman and Hall/CRC, London (1993)
10. Harrell, F.E.: *Regression Modeling Strategies with Applications to Linear Models, Logistic Regression, and Survival Analysis*. Springer, New York (2001)
11. Hastie, T., Tibshirani, R., Friedman, J.: *The Elements of Statistical Learning*, 2nd edn. Springer, Berlin (2009). <http://www-stat.stanford.edu/~tibs/ElemStatLearn/>
12. Houchens, R.L., Schoeps, N.: Comparison of hospital length of stay between two insurers for patients with pediatric asthma. In: Peck, L.H.R., Goodman, A. (eds.) *Statistical Case Studies: A Collaboration Between Academe and Industry*, pp. 45–64. The American Statistical Society, and the Society for Industrial and Applied Mathematics, Philadelphia (1998)
13. Kahn, H.S., Cheng, Y.J., Thompson, T.J., Imperatore, G., Gregg, E.W.: Two risk-scoring systems for predicting incident diabetes mellitus in US adults age 45 to 64 years. *Ann. Intern. Med.* **150**, 741–751 (2009)
14. Kettunen, J.A., Harilainen, A., Sandelin, J., Schlenzka, D., Seitsalo, S., Hietaniemi, K., Malmivaara, A., Kujala, U.M.: Knee arthroscopy and exercise versus exercise only for chronic patellofemoral pain syndrome: a randomized controlled trial. *BMC Med.* **5**, 38 (2007)
15. Layman, P.R., Agyras, E., Glynn, C.J.: Iontophoresis of vincristine versus saline in post-herpetic neuralgia: a controlled trial. *Pain* **25**, 165–170 (1986)
16. Levine, P.H.: An acute effect of cigarette smoking on platelet function: a possible link between smoking and arterial thrombosis. *Circulation* **48**(3), 619–623 (1973)

17. Little, M.A., McSharry, P.E., Hunter, E.J., Spielman, J., Ramig, L.O.: Suitability of dysphonia measurements for telemonitoring of Parkinson's disease. *IEEE Trans. Biomed. Eng.* **56**(4), 1015–1022 (2009)
18. Little, R.J.A., Rubin, D.B.: *Statistical Analysis with Missing Data*, 2nd edn. Wiley-Interscience, New York (2002)
19. Mackowiak, P.A., Wasserman, S.S., Levine, M.M.: A critical appraisal of 98.6°F, the upper limit of the normal body temperature, and other legacies of Carl Reinhold August Wunderlich. *JAMA* **268**, 1578–1580 (1992)
20. McGann, K.P., Marion, G.S., Camp, L., Spangler, J.G.: The influence of gender and race on mean body temperature in a population of healthy older adults. *Arch. Fam. Med.* **2**(12), 1265–1267 (1993)
21. Morewedge, C.K., Huh, Y.E., Vosgerau, J.: Thought for food: imagined consumption reduces actual consumption. *Science* **330**(6010), 1530–1533 (2010)
22. Mufunda, J.: Body mass index and blood pressure: where are we now? *J. Hum. Hypertens.* **21**(1), 5–7 (2007)
23. Nafiu, O.O., Burke, C., Lee, J., Voepel-Lewis, T., Malviya, S., Tremper, K.K.: Neck circumference as a screening measure for identifying children with high body mass index. *Pediatrics* **126**(2), 306–310 (2010)
24. Norton, P.G., Dunn, E.V.: Snoring as a risk factor for disease: an epidemiological survey. *Br. Med. J.* **291**, 630–632 (1985)
25. Penrose, K., Nelson, A., Fisher, A.: Generalized body composition prediction equation for men using simple measurement techniques. *Med. Sci. Sports Exerc.* **17**(2), 189 (1985)
26. Phillips, D.P., Barker, G.E.C.: A July spike in fatal medication errors: a possible effect of new medical residents. *J. Gen. Intern. Med.* **25**(8), 774–779 (2010)
27. Rubin, D.B.: Estimating causal effects from large data sets using propensity scores. *Ann. Intern. Med.* **127**(8 Pt 2), 757–763 (1997)
28. Sacks, F.M., Svetkey, L.P., Vollmer, W.M., Appel, L.J., Bray, G.A., Harsha, D., Obarzanek, E., Conlin, P.R., Miller, E.R., Simons-Morton, D.G., Karanja, N., Lin, P.H.: DASH-sodium collaborative research group: effects on blood pressure of reduced dietary sodium and the dietary approaches to stop hypertension (DASH) diet. DASH-sodium collaborative research group. *N. Engl. J. Med.* **344**(1), 3–10 (2001)
29. Scheffe, H.: *The Analysis of Variance*. Wiley, New York (1959)
30. Seeman, M.V.: The role of estrogen in schizophrenia. *J. Psychiatry Neurosci.* **21**(2), 123–127 (1996)
31. Smith, J.W., Everhart, J.E., Dickson, W.C., Knowler, W.C., Johannes, R.S.: Using the ADAP learning algorithm to forecast the onset of diabetes mellitus. In: Greenes, R.A. (ed.) *Proceedings of the Symposium on Computer Applications in Medical Care*, Washington, 1988, pp. 261–265. IEEE Computer Society Press, Los Alamitos (1988)
32. Sturges, H.A.: The choice of a class interval. *Am. Stat. Assoc.* **21**, 65–66 (1926)
33. Taubert, D., Roesen, R., Lehmann, C., Jung, N., Schömig, E.: Effects of low habitual cocoa intake on blood pressure and bioactive nitric oxide: a randomized controlled trial. *JAMA J. Am. Med. Assoc.* **298**(1), 49–60 (2007)
34. Team, R.D.C.: *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna (2005)
35. Teasdale, N., Bard, C., Larue, J., Fleury, M.: On the cognitive penetrability of posture control. *Exp. Aging Res.* **19**(1), 1–13 (1993)
36. The Steering Committee of the Physicians' Health Study Research Group: Findings from the aspirin component of the ongoing Physicians' Health Study. *N. Engl. J. Med.* **318**, 262–264 (1988)
37. Venables, W.N., Ripley, B.D.: *Main Package of Venables and Ripley's Mass*. R Package Version, 7.3-8 edn. (2010)
38. Weerahandi, S.: *Exact Statistical Methods for Data Analysis*. Springer, Berlin (2003)
39. Witte, A.V., Fobker, M., Gellner, R., Knecht, S., Floel, A.: Caloric restriction improves memory in elderly humans. *Proc. Natl. Acad. Sci. USA* **106**(4), 1255–1260 (2009)

# Index

## A

abline, 144, 287  
Agglomerative, 295  
Allele, 84, 103  
Alternative hypothesis, 173  
Analysis of variance, 221, 225  
ANOVA, 221, 257  
aov, 231, 232  
as.integer, 329  
Association, 4, 62, 243  
Average linkage, 296

## B

Bar plot, 23  
barplot, 52  
Bayes' rule, 98  
Bayes' theorem, 98  
Bayesian, 303  
Bayesian statistics, 101, 303  
Bernoulli, 164, 165  
Bernoulli trials, 115, 139  
Beta, 141, 306  
Between group variation, 2, 223  
Bimodal, 29  
Binomial, 115, 143  
Blocking, 14, 230  
boxplot, 54, 78  
Boxplots, 38, 54  
by, 78, 79

## C

Case-control studies, 13  
Categorical, 235  
Causation, 4, 62, 204  
cbind, 332, 335  
Central limit theorem, 163, 195, 229  
Centroid linkage, 296, 300

Chi-squared test, 248, 250  
chisq.test, 248–250  
Cluster sampling, 12  
Clustering, 291  
colnames, 249, 333  
Complement, 87, 89, 109  
Complete linkage, 295, 301  
complete.cases, 57  
Conditional, 93, 341  
Conditional probability, 93, 94, 175  
Conditional statements, 55, 341, 342  
Confidence coefficient, 160  
Confidence interval, 162, 261  
Confidence level, 160, 161, 163  
Confounding, 4, 5  
Contingency table, 72, 208, 241  
Continuous, 110, 131  
cor, 77  
cor.test, 217, 218  
Correlation, 66, 68  
Correlation test, 214, 217  
Count data, 18, 153  
Cross-sectional, 14  
Cumulative distribution function, 131  
cutree, 300

## D

data, 51, 285  
Data exploration, viii, 17  
Data frame, 336  
Data preprocessing, 40  
Data transformation, 43  
data.frame, 336, 341  
dbeta, 313  
dbinom, 144  
Decision, 5, 106, 311  
Decision theory, 5, 106



Decision tree, 106  
 Degrees of freedom, 130, 198  
 Dendrogram, 27, 28, 129  
 Density, 27, 28  
 Deviation, 34, 48, 134  
`diag`, 335  
 Difference of proportions, 70, 80  
`dim`, 333, 341  
 Discrete, 110, 112  
 Disjoint, 92  
`dist`, 300  
 Distance, 292, 295  
 Distribution, 25, 135, 165  
 Distribution function, 125, 131  
 Divisive, 295  
`dnorm`, 146  
 Dominant, 84  
 Double blind, 14  
`dpois`, 145  
`dt`, 147

**E**

Error term, 260, 272  
 Estimation, 151, 310  
 Euclidean distance, 292  
`exp`, 324, 326  
 Expected frequency, 240  
 Expected loss, 105, 106  
 Expected utility, 105  
 Experiments, 4, 13, 14  
`expression`, 313

**F**

`factor`, 53, 56, 336  
 Factors, 4, 76, 229  
 False negative, 100  
 False positive, 100  
 Fisher's exact test, 246  
 Five-number summary, 37, 54  
`fix`, 337, 338  
`for`, 55, 343  
 Frequency, 20  
 Frequentist statistics, 102  
`function`, 344

**G**

Genotype, 230  
 Goodness of fit, 270, 271, 277

**H**

Hardy-Weinberg, 104, 107  
`hclust`, 300  
`head`, 51, 56, 338  
`help`, 51

Heterozygous, 84, 88  
 Hierarchical clustering, 295, 296  
`hist`, 53, 55  
 Histogram, 26  
 Homozygous, 84  
 Hypothesis, 173  
 Hypothesis testing, 1, 173, 263

**I**

`if`, 341, 342  
`if-else`, 55, 342  
 IID, 152, 174, 226  
 Independence, 109, 240  
`install.packages`, 76, 285, 340  
 Intercept, 259, 267  
 Interquartile range, 38, 54  
 Intersection, 91, 109  
 Interval estimate, 161, 190  
 IQR, 54  
`is.factor`, 53

**J**

Joint probability, 91, 240

**K**

*K*-means, 293  
`kmeans`, 298

**L**

Law of large numbers, 153, 154  
 Law of total probability, 96  
 Least squares, 258, 265, 285  
`legend`, 299  
`length`, 327, 339  
 Library, 7, 57  
`library`, 51, 285, 340  
 Likelihood function, 305  
 Likelihood ratio, 306  
 Linear regression, 254, 275, 285  
 Linear regression models, 253, 263, 266  
 Linear relationship, 63, 211  
`lines`, 62, 65  
 List, 339, 344  
`list`, 341  
`lm`, 285, 288  
`load`, 326  
 Location, 25, 83  
`log`, 324, 344  
 Longitudinal data, 14  
 Loops, 55, 343  
 Loss function, 105, 312  
 Lower tail probability, 117, 175, 311

**M**

Margin of error, 167  
 Marginal probability, 91, 94, 105  
 Matched pairs design, 14  
 Matrix, 66, 213, 298  
*matrix*, 249, 344  
 max, 54, 327, 328  
 Mean, 32, 112, 146  
 mean, 327  
 Median, 32, 53  
 median, 53  
 min, 54, 327  
 Missing values, 40, 57, 327  
 Mode, 22, 29, 335  
 mode, 335  
 Model, 259, 271, 275  
 Model fitting, 253  
 Multiple regression, 275  
 Mutually exclusive, 92

**N**

*names*, 286, 333  
 Nominal, 19, 111  
 Nonlinear, 64, 76  
 Normal, 125, 163  
 Null distribution, 175, 196  
 Null hypothesis, 173, 180, 237  
 Numerical, 18, 74

**O**

*objects*, 325  
 Observation units, 3, 12  
 Observational studies, 4, 12, 69  
 Observed frequency, 240, 243  
 Observed significance level, 175, 183  
 Odds, 71, 305  
 Odds ratio, 71  
 One-sample *t* test, 189  
 One-sided test, 181  
 One-way ANOVA, 221, 229  
 Ordinal, 19, 111  
 Outliers, 5, 33, 64

**P**

*p*-value, 176, 237  
 Package, 8, 19, 318  
 Paired *t* test, 203, 216  
 Parameter, 113, 151  
 Partition, 93, 294  
*pbeta*, 314  
*pbinom*, 145  
 Pearson chi-squared test, 248  
 Pearson's correlation coefficient, 66, 213  
 Percentage, 21, 52

Percentile, 37  
 Phenotype, 84  
 Pie chart, 24  
*plot*, 76, 144, 300  
 Plot of means, 75, 228  
*pnorm*, 147, 188  
 Point estimate, 151, 195  
*points*, 144, 145, 299  
 Poisson, 119, 120  
 Population, 11, 146, 303  
 Population mean, 152, 158  
 Population proportion, 166, 186, 303  
 Population standard deviation, 155, 274  
 Population variance, 154, 162  
 Posterior, 101, 305  
 Posterior mean, 311  
 Posterior odds, 305, 312  
 Posterior probabilities, 101, 305  
 Power, 174, 323  
*ppois*, 146  
 Practical significance, 312  
 Prediction, 2, 253  
*print*, 325  
 Prior, 305  
 Prior odds, 306  
 Probability, 83, 123  
 Probability density function, 121  
 Probability distribution, 110, 308  
 Probability function, 130  
 Probability mass function, 112  
 Proportion, 70, 166  
 Prospective studies, 12  
*pt*, 147, 189

**Q**

*qbeta*, 314  
*qbinom*, 145  
*qnorm*, 147  
*qpois*, 146  
*qt*, 147  
 Quantile, 37  
 Quantile–quantile, 142  
 Quartile, 37

**R**

R Console, 6, 318  
 R-squared, 286  
 Random, 5, 109  
 Random sample, 5, 17, 102  
 Random variable, 73, 112  
 Randomization, 5, 14  
 Randomized block design, 14, 230  
 Randomized experiments, 4, 17  
 Range, 38, 109  
*range*, 54, 299

- Rate, 153
  - `rbind`, 332
  - `rbinom`, 143, 145
  - `read.table`, 337, 338
  - Recessive, 84, 100
  - `rect.hclust`, 300
  - Regression, 253
  - Regression coefficient, 260
  - Regression line, 256
  - Regression parameter, 264
  - Relationship, 4, 61, 193
  - Relative frequency, 21, 28, 52
  - Relative risk, 70, 80
  - `rep`, 327
  - Replication, 13
  - Residual, 257, 266
  - Residual sum of squares, 258
  - Retrospective studies, 12
  - `return`, 344
  - `rnorm`, 146
  - `round`, 52, 77, 324
  - `rownames`, 249
  - `rpois`, 145
  - `rt`, 147
- S**
- Sample, 84, 151, 254
  - Sample mean, 32, 151, 255
  - Sample proportion, 21, 235, 301
  - Sample size, 17, 153
  - Sample space, 84
  - Sample standard deviation, 36, 207, 267
  - Sample variance, 36, 203
  - Sampling distribution, 156, 195
  - Sampling units, 3
  - `save`, 325, 326
  - Scaling, 28, 48, 134
  - `sd`, 55
  - Sensitivity, 100
  - `seq`, 326
  - Shapiro–Wilk, 188
  - Shifting, 48, 134
  - Significance level, 175, 197
  - Simple linear regression, 260, 281, 289
  - Simple random sampling, 11, 124, 272
  - Simulation, 102
  - Single blind, 14
  - Single linkage, 295
  - Skewed, 29
  - `source`, 345
  - Specificity, 100
  - Spread, 25, 75, 124
  - `sqrt`, 324
  - Standard deviation, 34, 48, 157
  - Standard error, 162, 261, 274
  - Standard normal distribution, 130, 161
  - Standardization, 50, 134, 297
  - Statistical inference, viii, 5, 193
  - Statistical significance, 176
  - Stratified sampling, 11
  - Student's *t* distribution, 130
  - Subject, 14, 80
  - `sum`, 52, 327, 329
  - `summary`, 54, 232, 286
  - Summary statistics, 5, 17, 53
  - Symmetric, 29, 126
- T**
- `t`, 333
  - t* distribution, 166, 189, 237
  - t*-critical value, 163, 198
  - t*-score, 185, 263
  - t*-test, 184, 197, 216
  - `t.test`, 189, 216, 217
  - `table`, 51, 77, 338
  - Target population, viii, 1
  - Test, 99, 178, 235
  - Test statistics, 174, 181, 196
  - Time series, 14
  - Tree diagram, 85, 103
  - True negative, 100
  - True positive, 100
  - Two-sample *t* test, 197, 204, 221
  - Two-sided test, 186
  - Type I and type II errors, 174
- U**
- Uncertainty, 83, 105, 185
  - Uniform, 309
  - Unimodal, 29, 125
  - Union, 90
  - Upper tail probability, 117, 123
  - Utility function, 105
- V**
- `var`, 55
  - Variable, 17, 109, 193
  - Variance, 34, 112, 221
  - Vector, 144, 249, 326
  - Visualization, 5, 17, 25
- W**
- `which`, 57, 78, 331
  - Within group variation, 223
  - Working directory, 325
  - Workspace, 325
- Z**
- z*-critical value, 162
  - z*-score, 178, 182, 197
  - z*-test, 177, 188