A short tutorial on R

Babak Shahbaba

Department of Statistics
University of California at Irvine, Irvine, CA, USA

1 Starting with R

In the R Console, you 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 *Submit*. 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 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 14, we include the option `base=10`

```
> log(14, base = 10)

[1] 1.146128
```

In general, you can always learn more about a function 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(14, base = 10), digits = 2)

[1] 1.15
```

In the above code, the output of `log(14, base=10)` becomes the first argument of the function `round()`.

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. Simply typing the name of an object displays its contents.

```
> x

[1] 2

> y

[1] 3

> 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 a period `.` in a name to separate words (e.g., `my.object`).

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
```

Here, the `c()` function is used to combine the numbers into a vector. Since $1, 2, \ldots, 5$ 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
```

Here, we stored the results in the same object (`x`) to avoid creating a new one.

To create sequences and store them in vector objects, we can also use the `seq()` function for additional flexibility. The following commands creates 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`).

```
> 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 size and specify its elements later.

We can find the length of a vector (i.e., number of elements) using the `length()` command:

```
> length(x)

[1] 5

> length(y)

[1] 9
```

Functions `sum()`, `min()` and `max()` calculate the sum, find the minimum, and find the maximum of elements in a vector.

```
> x

[1] 1 2 3 4 5

> sum(x)

[1] 15

> min(x)

[1] 1

> max(x)

[1] 5
```

The elements of a vector can be accessed by providing their index using square brackets `[ ]`. The first index is always 1. For example, try retrieving the 2^{nd} element of `x` and the 5^{th} element of `y`.

```
> x[2]

[1] 2

> y[5]

[1] 5
```

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 3^{rd} element of a vector, we use negative indexing.

```
> y[-3]

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

A vector can also hold characters and strings, which must be surronded by single or double quotations marks. For example, suppose we have a sample of 5 patients. We can create a vector storing their gender as

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

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`. Note that these values must be in capital letters and can be abbreviated `T` and `F`, respectively. 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
```

Internally, R assigns 0 to `FALSE` elements and 1 to `TRUE` elements. The `as.integer()` function returns the internal coding.

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

[1] 1 1 0 1 0
```

Functions can also be applied to logical vectors. For instance, the `sum()` function returns the number of healthy subjects.

```
> sum(is.healthy)

[1] 3
```

Logical vectors are usually derived from other vectors using relational 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. The opposite relational operator is `!=`, which returns `TRUE` if the two sides are not equal:

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

[1] FALSE  TRUE  TRUE FALSE  TRUE
```

The other relational operators commonly applied to numbers and vectors are less than `<`

```
> 4 < 5

[1] TRUE
```

less than or equal to

```
> x[2] <= y[2]

[1] FALSE
```

greater than

```
> sum(y) > max(x)

[1] TRUE
```

and greater than or equal to

```
> 4 >= 0

[1] TRUE
```

R also uses *Boolean* operators. 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 either of the corresponding elements is `TRUE`:

```
> is.male | is.healthy

[1]  TRUE  TRUE FALSE  TRUE FALSE
```

The Boolean operators can be used with the relational operators. The following commands create a numerical vector for the age of the five subjects and then find which subjects are male and less than 40 years old.

```
> age = c(60, 43, 72, 35, 47)
> is.young.male <- is.male & (age < 40)
> is.young.male

[1] FALSE FALSE FALSE  TRUE FALSE
```

Instead of using individual vectors, it is easier 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 from the same type (e.g., numerical, character, logical), a *matrix* can be used. For example, assume that for our 5 subjects, we have also measured BMI 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:

```
> 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:

In general, matrices are two dimensional objects comprised of numerical values. The object `data.1` is a 5×3 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 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
```

Transposing the matrix (e.g., interchanging the rows and columns) is accomplished with the `t` function:

```
> data.t = t(data.1)
> data.t

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

```
> data.1[, 2]

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

A matrix can also be created by re-arranging 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, `nrow` and `ncol` are the number of rows and columns, respectively. The length of `data` must be equal to $nrow \times ncol$. 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
```

We can also create a matrix with missing values and specify its elements later.

```
> matrix(data = NA, nrow = 3, ncol = 4, byrow = TRUE)
```

```
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
```

3 Data frames

To store our data in a matrix format, all the measurements must be numerical. Often, however, we have measurements of mixed types (e.g., numerical, character, factor, logical). In this case, we can store them in a table format similar to the format of spreadsheets (e.g., Excel). The resulting object (which includes multiple objects of possibly different types) is called a *data frame* object.

```
> 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
```

Again to access elements of a data frame, we use the square brackets `[ , ]` with the appropriate row and column indices. For example, the blood pressure of the 3^{rd} subject is

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

[1] 21
```

As before, we can access an entire row (all the measurements for one subject) by only specifying the row index, and an entire column (all the measurements for one variable) by only specifying the column index. We can also access an entire column by providing the variable 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 `[ ]` (without the comma). For instance, try obtaining the `BMI` for the 4^{th} subject and the `gender` of the 2^{nd} subject.

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

[1] 27

> data.df$gender[2]

[1] female
Levels: female male
```

4 Importing data from text files

Usually, the data is available in a tabular format as a delimited text file. Before importing a data set into R, enter the command `getwd()` to see the current working directory for R. (This is where all the objects are saved and accessed.) In general, we can import tabular data into R using the function `read.table()`. For instance, let us try importing the `BodyTemperature` data set. (This data set is available online at `http://www.ics.uci.edu/~babaks/BWR`)

```
> 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.) The `header=TRUE` option tells R that the variable names are contained in the first line of the data, and the `sep=" "` option tells R that the columns in the data set are separated by white spaces. If the columns were separated by commas, 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$Age[2:4]

[1] 32 42 33

> BodyTemperature$Age[5]

[1] 26
```

5 Lists

The data frames we created above (either directly or by importing a text file) includes objects of different types, but they all have the same length. To store objects of different types and possibly

with different length, we can use a *list* instead. For example, suppose 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(our.data = BodyTemperature,
+     investigators = c("Smith", "Jackson",
+         "Stern"), students = c("Steve", "Mary"))
> length(our.study)

[1] 3
```

We created a list with 3 *components*. Each component has a name, which can be used to access that component.

```
> our.study$investigator

[1] "Smith"   "Jackson" "Stern"
```

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

```
> our.study[[2]]

[1] "Smith"   "Jackson" "Stern"
```

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

```
> our.study[[2]][3]

[1] "Stern"

> 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
```

6 Loading add-on packages

A package includes a set of functions that are commonly use for specific area of statistical analysis. R users constantly create new packages and make them publicly available on CRAN (Comprehensive R Archive Network). To use a package, you first need to download the packages from CRAN and install it in your local R library. For this, we can use the `install.packages()` function. For example, suppose we want to learn the basic concepts of Bayesian statistical inference. Jim Albert has created a package called `LearnBayes` for this purpose. The following command downloads the `LearnBayes` package.

```
> install.packages("LearnBayes", 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 "Rcmdr" depends. The referrence manual for this package is available at `http://cran.r-project.org/web/packages/LearnBayes`.

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(LearnBayes)
```

Now we can use all the functions available in this package. We can also use all the data sets and the author of the package has included in the package. For example, the `LearnBayes` package includes a data set called `birdextinct` that shows the measurements on breedings pairs of landbird species collected from 16 islands about Britain. To use this data set, we enter the following command

```
> data(birdextinct)
```

The data set is now available to us as a data frame.

7 Conditional statements

Consider the `birthwt` data set from the `MASS` package. The data set includes the birthweight of babies, `bwt`. It also includes an 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 we did not have this variable and we wanted to create it. First, let us load the `birthwt` data set into R.

```
> library(MASS)
> data(birthwt)
> dim(birthwt)

[1] 189  10
```

The data set includes 189 cases and 10 variables.

We now create an empty vector, called `low2`, of size 189 within the `birthwt` data frame.

```
> birthwt$low2 <- rep(NA, 189)
```

Then we want to examine the birthweight of each baby, and *if* it is below 2500, we assign the value of "1" to the `low2` 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 `low2` of the first observation.

```
> if (birthwt$bwt[1] < 2500) {
+     birthwt$low2[1] <- 1
+ }
```

Check the result:

```
> birthwt$bwt[1]

[1] 2523

> birthwt$low2[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$bwt[1] < 2500) {
+     birthwt$low2[1] <- 1
+ } else {
+     birthwt$low2[1] <- 0
+ }
> birthwt$bwt[1]

[1] 2523
```

```
> birthwt$low2[1]

[1] 0
```

Conditional statements can have multiple `else` statements to test multiple conditions.

```
> if (condition1) {
+     expression1
+ } else if (condition2) {
+     expression2
+ } else {
+     expression3
+ }
```

8 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
+ }
```

where i is the loop counter and 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 an empty vector of size 10 and store $i + 1$ as the i^{th} element of the vector.

```
> x <- rep(NA, 10)
> for (i in 1:10) {
+     x[i] <- i + 1
+ }
> x

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

For the above example, we use the following loop:

```
> for (i in 1:189) {
+     if (birthwt$bwt[i] < 2500) {
+         birthwt$low2[i] <- 1
+     }
+     else {
+         birthwt$low2[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 set the value of `low2` for that row to 1, otherwise, it sets it to 0. The variable `low2` variable, which you created using the above loop and conditional statements, will be exactly the same as the existing variable `low`.

9 Saving and loading the workspace

So far, we have been using R interactively. Alternatively, we can write the above commands in a file with ".R" extension, (e.g., `myFile.R`) and run the file by typing

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

If the file is not in the current working directory, we have to give the full directory address. At any time, we can view a list of the objects in the workspace with the `ls()` function.

We can exit R by typing `q()`. Usually, R asks the user whether to save the workspace image. By typing `y` (yes), R saves an image of the workspace in a file called `.RData`, and this file is automatically loaded the next time R opens. (The `save.image()` function can also be used to save the entire workspace.)

To save only a few objects, use the `save()` function. For example, assume we only want to save the `BodyTemperature` and `our.study` objects in a file called "myObjects.Rdata".

```
> save(bodyTemperature, our.study, file = "myObjects.Rdata")
```

Later, you can load these objects with

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

As before, give the full address for the file if it is not located in the current working directory.

10 Creating functions

So far, we have created objects, performed specific tasks, and obtained their results. If we need to repeat the same task over an over again, a more efficient approach would be to create a function. The function we create itself is an object and is similar to existing functions in R, such as `sum()`, `log()`, `colnames()`, that we have been using so far. 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 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 so 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 the function (Fig. 10).

```
> 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. Therefore, if the user does not specify the number of decimal points, the function uses the default value.

The function then creates two objects, `min.value` and `max.value`, that store the min and max respectively. Next, the function prints the sum of all elements. Finally, the function create 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 "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. 10, we wrote some comments in the text editor to explain what the function. The comments should be always followed by the "#". R will not execute a line that starts with "#"

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 a ".R" extension.

So far, you have just created a file that contains the command necessary to create your function. The function has not created yet. To create your function, you need to execute the commands, you can use the `source()` function to read the codes from the "CreateMyFun.R" file.

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

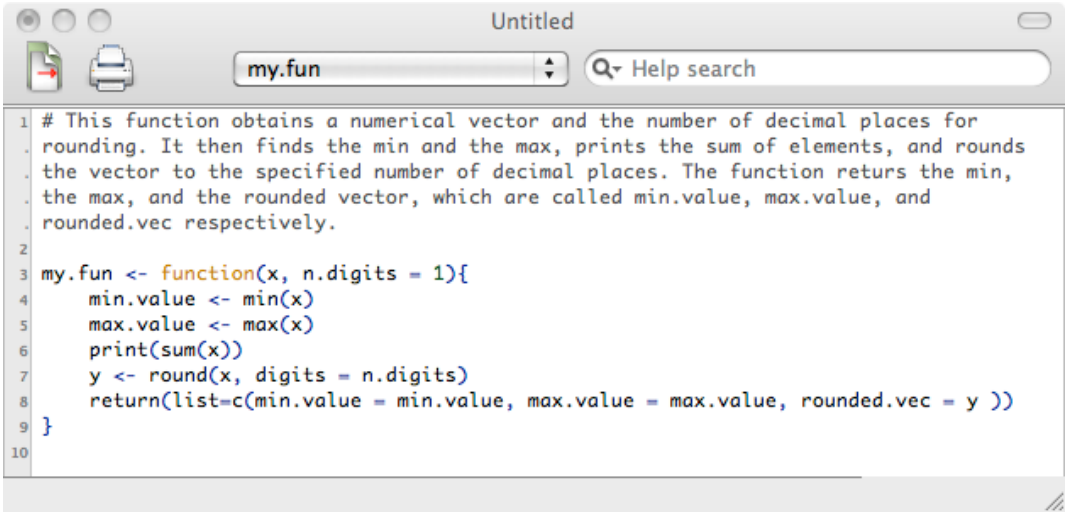

Figure 1: Creating a function called `my.fun()` using a text editor.

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 entering its name.

11 Data exploration with R programming

We believe that writing your own commands gives you more control over the output and a deeper understanding of the material. Here, we review the functions that are commonly used for data exploration. We start by loading the `Pima.tr` data set, which is available from the `MASS` package.

```
> library(MASS)
> data(Pima.tr)
```

The `library()` command loads the `MASS` package, and the `data()` command loads the `Pima.tr` data set. Note that the package should be loaded first before we can access its data sets.

Type `Pima.tr` 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(Pima.tr)
```

```
  npreg glu bp skin  bmi   ped age type
1     5  86 68   28 30.2 0.364  24   No
2     7 195 70   33 25.1 0.163  55  Yes
3     5  77 82   41 35.8 0.156  35   No
4     0 165 76   43 47.9 0.259  26   No
5     0 107 60   25 26.4 0.133  23   No
6     5  97 76   27 35.6 0.378  52  Yes
```

When you obtain a data set from a package, you can use the `help()` function to view the description on the data available in the pachage.

```
> help(Pima.tr)
```

11.1 Bar Graphs and frequencies

A common summary statistic for categorical variables is the frequency n_c. Use the `table()` function to obtain the frequencies for the categorical variable `type` from the `Pima.tr` data set.

```
> type.freq <- table(Pima.tr$type)
> type.freq
```

```
 No Yes
132  68
```

Once again, the $ symbol is being used to access the `type` variable in from the `Pima.tr` data set.

Now, use the `type.freq` table to create the bar graph. Bar graphs show us how observations categorical variables are distributed in the sample.

```
> barplot(type.freq, xlab = "Type",
+     ylab = "Frequency",
+     main = "Frequency Bar Graph of Type")
```

The first parameter to the `barplot()` function is the frequency table. The options `xlab` and `ylab` label the x and y axes, respectively. Likewise, the `main` option puts a title on the plot.

Often it is more informative to report the relative frequencies. The relative frequency is the percentage or proportion in each category and is calcuated by $p_c = n_c/n$ as in Eq. **??**. Therefore, we need the frequencies n_c (stored in the `type.freq` table) and the total sample size n. Since the sum of the frequencies is the total sample size, $\sum_c n_c = n$, we can use the `sum()` function to add the entries in the frequency table:

```
> n <- sum(type.freq)
> n

[1] 200
```

Now create a table of relative frequencies by dividing the frequency table by the sample size.

```
> type.rel.freq <- type.freq/n
```

Use the `round()` function to limit the output to 2 decimal places.

```
> round(type.rel.freq, 2)

  No  Yes
0.66 0.34
```

We can also multiply the relative frequencies by 100 to get the percentages:

```
> round(type.rel.freq * 100, 0)

 No Yes
 66  34
```

Finally, you can create a relative frequency barplot with

```
> barplot(type.rel.freq,
+     xlab = "Type", ylab = "Relative Frequency",
+     main = "Relative Frequency Bar Graph of Type")
```

If the levels of a categorical variable in your data set is coded as numbers, you need to convert the type of variable to *factor* using the `factor()` function so R recognizes it as categorical. You can use the function `is.factor()` to examine whether a random variable is a factor. For example, the `smoke` variable (smoking status) in `birthwt` is coded as 0 for mothers who did not smoke during their pregnancy, and 1 for mother who smoked during their pregnancy. R automatically considers this variable as numerical. To convert the variable to categorical, use the following code:

```
> birthwt$smoke <- factor(birthwt$smoke)
> is.factor(birthwt$smoke)
```

```
[1] TRUE
```

```
> table(birthwt$smoke)
```

```
  0   1
115  74
```

11.2 Histograms

Histograms show how observations for a numerical random variable are distributed amongst possible values. To create the frequency histogram for `age`, use the `hist()` function with the `freq` option set to "TRUE" (which is the default).

```
> hist(Pima.tr$age, freq = TRUE,
+     xlab = "Age", ylab = "Frequency",
+     col = "grey", main = "Frequency Histogram of Age")
```

Then create a density histogram of `age` by setting the `freq` option to "FALSE".

```
> hist(Pima.tr$age, freq = FALSE,
+     xlab = "Age", ylab = "Density",
+     col = "grey", main = "Density Histogram of Age")
```

11.3 Summary Statistics

We can obtain single point summary statistics for numerical data with the `mean()` and `median()` functions. Find these measures for numerical variables in `Pima.tr`.

```
> mean(Pima.tr$npreg)
```

```
[1] 3.57
```

```
> median(Pima.tr$bmi)
```

```
[1] 32.8
```

The `quantile()` function with the `probs` option returns the specified quantiles.

```
> quantile(Pima.tr$bmi, probs = c(0.1, 0.25, 0.5,
+     0.9))
```

```
   10%    25%    50%    90%
24.200 27.575 32.800 39.400
```

Here, the desired quantiles are specified as a vector using the combine `c()` function in the `probs` function. The five number summary along with the mean can simply be obtained with the `summary()` function:

```
> summary(Pima.tr$bmi)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  18.20   27.58   32.80   32.31   36.50   47.90
```

Let us present the five number summary visually with a boxplot:

```
> boxplot(PIma.tr$bmi, ylab = "BMI")
```

While the default is to create vertical boxplots, we can also create horizontal boxplots by specifying the `horizontal` option to true.

```
> boxplot(Pima.tr$bmi, ylab = "BMI", horizontal = T)
```

Find the interquartile range (IQR) or the length of the box with the `IQR()` function.

```
> IQR(Pima.tr$bmi)
```

```
[1] 8.925
```

The smallest and largest observations can be obtained with the `range()` function. (The functions `min()` and `max()` could also be applied.)

```
> minMax <- range(Pima.tr$bmi)
> minMax
```

```
[1] 18.2 47.9
```

Here, we created vector `minMax` with the minimum as the first element and the maximum as the second element. Obtain the range by subtracting the first element from the second.

```
> minMax[2] - minMax[1]
```

```
[1] 29.7
```

The variance and standard deviation are also easily calculated with `var()` and `sd()`

```
> var(Pima.tr$bmi)
```

```
[1] 37.5795
```

```
> sd(Pima.tr$bmi)
```

```
[1] 6.130212
```

11.4 Creating categories for numerical variables

The `hist()` function automatically divides the range of possible values into several intervals. Instead, as discussed above, we can create more meaningful intervals, which will be treated as categories. To create a categorical variable `weight.status` based on the `bmi` variable in `Pima.tr`, we can go through each observation one by one and assign her to one of the four categories: "Underweight", "Normal", "Overweight", and "Obese". To do this, we can use **loops** and **conditional** statements, which are discussed in Appendix **??**.

First, we start by creating an empty vector of size 200 within the `Pima.tr` data frame.

```
> Pima.tr$weight.status <- rep(NA, 200)
```

Next, we set the values of `weight.status` for all observations by using `if-else()` statements within a `for()` loop.

```
> for (i in 1:200) {
+     if (Pima.tr$bmi[i] < 18.5) {
+         Pima.tr$weight.status[i] <- "Underweight"
+     }
+     else if (Pima.tr$bmi[i] >= 18.5 &
+         Pima.tr$bmi[i] < 24.9) {
+         Pima.tr$weight.status[i] <- "Normal"
+     }
+     else if (Pima.tr$bmi[i] >= 24.9 &
+         Pima.tr$bmi[i] < 29.9) {
+         Pima.tr$weight.status[i] <- "Overweight"
+     }
+     else {
+         Pima.tr$weight.status[i] <- "Obese"
+     }
+ }
```

Here, the loop counter goes from 1 to 200. Use the `head()` function to view the result:

```
> head(Pima.tr)
```

```
  npreg glu bp skin  bmi   ped age type
1     5  86 68   28 30.2 0.364  24   No
2     7 195 70   33 25.1 0.163  55  Yes
3     5  77 82   41 35.8 0.156  35   No
4     0 165 76   43 47.9 0.259  26   No
5     0 107 60   25 26.4 0.133  23   No
6     5  97 76   27 35.6 0.378  52  Yes
  weight.status
```

```
1         Obese
2    Overweight
3         Obese
4         Obese
5    Overweight
6         Obese
```

Before we use the newly created variable `weight.status` in statistical analysis, we should convert it to factor so R recognize it as a categorical variable.

```
> Pima.tr$weight.status <- factor(Pima.tr$weight.status)
```

While the above code makes `weight.status` a factor variable, it does not take into account the ordering of levels. The levels are ordered alphabetically, and can be examined using the `levels()` function.

```
> levels(Pima.tr$weight.status)

[1] "Normal"      "Obese"       "Overweight"
[4] "Underweight"
```

We can provide the right ordering when we use the `factor()` function to convert the variable.

```
> Pima.tr$weight.status <- factor(Pima.tr$weight.status,
+     levels = c("Underweight", "Normal",
+         "Overweight", "Obese"))
> levels(Pima.tr$weight.status)

[1] "Underweight" "Normal"      "Overweight"
[4] "Obese"
```

11.5 Handling missing data in R

To find missing values of a random variable, we can use the `is.na()` function, which returns "TRUE" when the value is missing, and "FALSE" otherwise.

```
> is.na(Pima.tr2$bp)
```

To obtain the indices of observations whose values are missing, we can use the `which()` function along with the `is.na()` function. In general, `which()` can be used to find the "TRUE" indices of a logical object (e.g., vector).

```
> which(is.na(Pima.tr2$bp))
```

In contrast, the `complete.cases()` function returns a logical vector indicating which cases (observations) in the data set are complete, i.e., have no missing values.

```
> complete.cases(Pima.tr2)
```

To remove cases with missing values, we can use the `na.omit()` function.

```
> Pima.complete <- na.omit(Pima.tr2)
```

Here, the newly created `Pima.complete` data set includes only the complete cases from `Pima.tr2`.

12 Exploring relationships with R programming

12.1 Numerical variables

Pearson's correlation coefficient, which quantifies the strength and direction of the linear relationship between two numerical variables, is easily obtained with the `cor()` function:

```
> cor(bodyfat$abdomen, bodyfat$siri)
```

```
[1] 0.8134323
```

The resulting correlation coefficient of $r = 0.81$ suggests there is evidence of a strong positive relationship between abdomen circumference and percent body fat. Likewise, the `cor` function can be use to obtain the correlation matrix for multiple variables.

```
> options(width = 60)
> cor.matrix <- cor(bodyfat[, c("siri", "weight",
+     "height", "abdomen")])
> round(cor.matrix, 2)
```

```
         siri weight height abdomen
siri     1.00   0.61  -0.09    0.81
weight   0.61   1.00   0.31    0.89
height  -0.09   0.31   1.00    0.09
abdomen  0.81   0.89   0.09    1.00
```

Here, we are using the combine function `c()` to specify we want the correlation matrix for the columns labeled "siri", "weight", "height" and "abdomen". Then, the `round()` function is used to round the output to 2 decimal places.

An easy way to visualize the relationship between two numerical variables is to use scatterplots. In R, you can use the `plot()` function for this purpose. For example, the following code creates the scatterplot of percent body fat (`siri`) by abdomen circumference (`abdomen`).

```
> plot(bodyfat$abdomen, bodyfat$siri, xlab = "Abdomen Circumference",
+     ylab = "Percent Body Fat")
```

The first parameter to the `plot()` function is the variable to be represented by the x-axis, and the second parameter is variable to be represented by the y-axis.

It would be easier to detect patterns if trend lines are added to this graph. These trend lines can be obtained with the `lm()` function, which will be discussed in detail in Chapter **??**.

```
> trendLine <- lm(bodyfat$siri ~ bodyfat$abdomen)
```

The parameter to the `lm()` function is the formula of the response variable (`siri`), which is the y-axis in the scatterplot, by the explanatory variable (`abdomen`), which is the x-axis in the scatterplot.

We can then use the `abline()` function to add a straight line to an existing plot.

```
> abline(trendLine)
```

By default, `abline()` draws a solid line. We can set the line type to dashed line by using the option `lty=2`.

```
> abline(trendLine, lty = 2)
```

The final graph is similar to Fig. **??**.

In general, the `abline()` function can be used to add a straight line to an existing plot. (You first need to create a plot before using `abline`.) For example, `abline(h=2)` draws a horizontal line two units above the origin, `abline(v=-1)` draws a vertical line one unit to the left of origin, and `abline(a=-5, b=2)` draws a line with intercept -5 and slope 2.

To add additional points to an existing plot, you can use the `points()` function. To learn more about this function, enter the command `?points` or `help(points)`.

12.2 Categorical Variables

For two categorical variables, we are interested in creating a contingency table of the counts of each combination. From this contingency table, we can obtain the proportions, relative proportions (risk), odds and the odds ratio. For instance, try creating the contingency table for `smoke` by `low` from the `birthwt` data set with the `table()` function.

```
> table(birthwt$smoke, birthwt$low)
```

```
    0  1
0 86 29
1 44 30
```

The first parameter to the `table()` is the row variable and the second parameter is the column variable.

12.3 Numerical and Categorical Variables

The distribution of the numerical variable can be visualized for each level of categorical variable by using boxplots. For instance, create a boxplot of `bwt` for each level of `smoke`.

```
> boxplot(bwt ~ smoke, ylab = "Birthweight",
+     xlab = "Smoking Status",
+     data = birthwt, main = "Boxplots of Birthweight by Smoking Statust")
```

The first parameter is a formula, using the $\sim$ symbol to plot `bwt` (the response variable) by `smoke` (the explanatory variable).

The summary statistics for `bwt` can be calculated for each level of `smoke`. Using the `which()` function, we can find the indices of smoking mothers (`smoke=1`) in the `birthwt` data set:

```
> smoke.ind <- which(birthwt$smoke == 1)
```

Now, obtain the summary statistics of this group.

```
> summary(birthwt$bwt[smoke.ind])

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    709    2370    2776    2772    3246    4238

> sd(birthwt$bwt[smoke.ind])

[1] 659.6349
```

A more convenient way to obtain summary statistics by group is to use the `by` function.

```
> by(birthwt$bwt, birthwt$smoke, summary)

birthwt$smoke: 0
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1021    2509    3100    3056    3622    4990
------------------------------------
birthwt$smoke: 1
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    709    2370    2776    2772    3246    4238
```

The first parameter of this function specifies the random variable to which we want to apply the `summary` function (or any other function). The second parameter specifies the indicator variable to identify the groups. The last parameter, `summary`, specifies the function we want to apply to different groups. The general form of the `by` function is `by(data, indices, function)`. For example, the following code returns the standard deviation of birthweight for different levels of `ht` (hypertension history).

```
> by(birthwt$bwt, birthwt$ht, sd)

birthwt$ht: 0
[1] 709.4418
------------------------------------
birthwt$ht: 1
[1] 917.3617
```

13 Probability distributions with R programming

As in R-Commander, it is possible to plot theoretical distributions and obtain probabilities directly from the command line.

13.1 Binomial distribution

Assume we want to examing 10 people for a disease that has infection probability 0.2 in the population of interest. The number of people who are infected, Y, therefore has a Binomial(10, 0.2) distribution. Let us first simulate 5 random samples from this distribution (i.e., examine 5 groups each with 10 people):

```
> rbinom(5, size = 10, prob = 0.2)

[1] 3 1 1 4 3
```

where the first argument to the `rbinom()` functions specifies the number of random samples. The `size` option is the number of Bernoulli trials (here $n = 10$), and the `prob` option is the probability for the event of interest: $P(X = 1)$. Each randomly generated number represents the number of people affected by the disease out of 10 people. If we set `size=1`, we will be simulating random samples from the corresponding Bernoulli distribution. For example, we can simulate the disease status for a group of 10 people

```
> rbinom(10, size = 1, prob = 0.2)

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

Now suppose we want to know the probability of observing 3 out of 10 people affected by the disease: $P(Y = 3)$. Then we need probability mass function `dbinom()`, which returns the density of a binomial distribution:

```
> dbinom(3, size = 10, prob = 0.2)

[1] 0.2013266
```

Along with the input $y = 3$, the other parameters to the `dbinom()` function are the number of Bernoulli trials (`size=10`) and the probability (`prob=0.2`) for the event of interest.

We can also create a vector `y` for the range (i.e., all the possible values) of Y and then use this vector as input to `dbinom()` function

```
> y <- 0:10
> Py <- dbinom(y, size = 10, prob = 0.2)
> round(Py, 2)

 [1] 0.11 0.27 0.30 0.20 0.09 0.03 0.01 0.00
 [9] 0.00 0.00 0.00
```

With vectors `y` and `Py` we can plot the probability mass function (pmf), similar to the one shown in Fig. **??**:

```
> plot(y, Py, type = "h", xlab = "Number of Successes",
+     ylab = "Probability Mass",
+     main = "Binomial(10, 0.2)")
> points(y, Py, pch = 16)
> abline(h = 0, col = "gray")
```

In the `plot()` function, the first argument provides the values for the horizontal axis (`y`) and the second argument provides the values for the vertical axis (`fy`). We use the `type="h"` option to create "histogram-like" vertical lines. The points at the top of the lines are added with the `points()` function, whose option `pch=16` gives filled-in circles. Similar to the `plot()` function, the first and second arguments provide the coordinates of points. Lastly, the gray horizontal line at the probability of 0 is added with `abline(h=0, col="gray")`. The functions `points()` and `abline()` only add points and lines to an existing plot; they can not be used alone.

Now suppose we are interested in the probability of observing 3 or fewer affected people in a group of 10. We could sum the values of pmf: $P(Y \leq 3) = P(Y = 0) + P(Y = 1) + P(Y = 2) + P(Y = 3)$

```
> sum(Py[1:4])

[1] 0.8791261
```

Alternatively, we can use the distribution function for a binomial random variable `pbinom()` and obtain the lower tail probability:

```
> pbinom(3, size = 10, prob = 0.2, lower.tail = TRUE)

[1] 0.8791261
```

As before, the arguments `size=10` and `prob=0.2` specify the parameters of the binomial distribution. The option `lower.tail=TRUE` tells R to find the lower tail probability. By changing the `lower.tail` option to false (`FALSE`), we can find the upper tail probability $P(Y > 3)$:

```
> pbinom(3, size = 10, prob = 0.2, lower.tail = FALSE)

[1] 0.1208739
```

In contrast, to obtain the 0.879 quantile, we use the `qbinom()` function:

```
> qbinom(0.879, size = 10, prob = 0.2, lower.tail = TRUE)

[1] 3
```

13.2 Poisson distribution

Suppose that on average 4 people visit the hospital each hour. Then we can can represent the hourly number of hospital visitation as $X \sim \text{Poisson}(4)$ and simulate 12 samples from this distribution:

```
> rpois(12, 4)

 [1] 3 5 3 7 3 5 9 4 2 5 4 4
```

Each randomly generated number represents the number of people visiting the hospital each hour. Similar to the `rbinom()` function, the first parameter to the `rpois()` function is the number of samples and the remaining argument specifies the distribution parameter.

Suppose we want to know the probability that 6 people visit the hospital in an hour. Then we would use the probability mass function `dpois()`:

```
> dpois(6, 4)

[1] 0.1041956
```

Here, 6 is the specific value of the random variable, and the 4 is the distribution parameter. As before, we can create a plot of the pmf by first creating a vector of possible values and finding their corresponding densities.

To find the probability of 6 or fewer people visiting the hospital (as opposed to the probability that exactly 6 people visit), we need to find the lower tail probability of $x = 6$. For this, we use the `ppois()` function.

```
> ppois(6, 4)

[1] 0.889326
```

The 0.889 quantile of the distribution is

```
> qpois(0.889, 4)

[1] 6
```

13.3 Normal distribution

Suppose BMI in a specific population has a normal distribution with mean of 25 and variance of 16: $X \sim N(25, 16)$. Then we can simulate 5 values from this distribution using the `rnorm()` function.

```
> rnorm(5, mean = 25, sd = 4)

[1] 19.78996 30.38077 23.43738 26.43898 25.71806
```

In the *rnorm*() function, the first parameter the number of samples, the second parameter is the mean and the third parameter is the standard deviation (not the variance).

Now let us plot the pdf of this distribution. A normal random variable can take any value from $-\infty$ to ∞. However, according to the *68-95-99.7 rule* approximately 99.7% of the values fall within the interval [13, 37] (i.e., within 3 standard deviations of the mean). Therefore, the interval $[10, 40]$ is wide enough to plot the distribution:

```
> x <- seq(from = 10, to = 40, length = 100)
```

Here, vector `x` is a sequence of length 100 from 10 to 40. We can then find and plot the density at each point in `x`:

```
> fx <- dnorm(x, mean = 25, sd = 4)
> plot(x, fx, type = "l", xlab = "BMI", ylab = "Density",
+     main = "N(25, 16)")
> abline(h = 0, col = "gray")
```

The `dnorm()` function returns the height of the density curve at a specific point and requires the parameters of the `mean` and the standard deviation `sd`. In the `plot()` function, we are using `type="l"` to plot the points as a continuous line (curve).

Remember that for continuous variables the probability of a specific value is always zero. Instead, for continuous variables, we are interested in the probability of observing a value in a given interval. For instance, the probability of observing a BMI less than or equal to 18.5 is the area under the density curve to the left of 18.5. In R, we find this probability with the cumulative distribution function `pnorm()`:

```
> pnorm(18.5, mean = 25, sd = 4, lower.tail = TRUE)

[1] 0.05208128
```

Once again, we can find the upper tail probability $P(X > 22)$ by setting the option `lower.tail=FALSE`. The `qnorm()` returns the quantile for normal distributions is. For example, the 0.05 quantile for the above distribution is

```
> qnorm(0.05, mean = 25, sd = 4, lower.tail = T)
```

```
[1] 18.42059
```

We can find the probability of a BMI between 25 and 30 by subtracting their lower tail probabilities: $P(25 < X \leq 30) = P(X \leq 30) - P(X \leq 25)$.

```
> pnorm(30, mean = 25, sd = 4) - pnorm(25, mean = 25,
+     sd = 4)
```

```
[1] 0.3943502
```

We can also create a plot of the cdf by using vector `x` as input to `pnorm()` function:

```
> Fx <- pnorm(x, mean = 25, sd = 4)
> plot(x, Fx, type = "l", xlab = "BMI", ylab = "Cumulative Probability",
+     main = "N(25, 16)")
> abline(h = 0, col = "gray")
```

In general for each distribution, the random generating function starts with r, the probability density function starts with d, the distribution function (i.e., cdf) starts with p and the quantile function starts with q. For the t-distribution, these functions are `rt()`, `dt()`, `pt()` and `qt()`. The corresponding functions for the χ^2-distribution are `rchisq()`, `dchisq()`, `pchisq()` and `qchisq()`.

14 Linear regression with R programming

Fitting a linear regression model in R is straightforward. Here, we model the relationship between percent body fat, `siri`, and abdomen circumference, `abdomen`, using a simple linear regression model. The following commands install the `mfp` package using the `install.packages()` function, load it into R using the `library()` function, and make the data `bodyfat` available for analysis using the `data()` function.

```
> install.packages("mfp", dependencies = TRUE)
> library(mfp)
> data(bodyfat)
```

We set the `dependencies` to `TRUE` to install other packages that are related to `mfp` along with it.

To fit the least-squares regression model, use the `lm()` function:

```
> fit <- lm(siri ~ abdomen, data = bodyfat)
```

The first argument of the function is the formula of the form of "response $\sim$ explanatory". The second argument specifies the data set. By giving the name of the data set this way, we avoid witting the equation as `bodyfat$siri` $\sim$ `bodyfat$abdomen`.

The `fit` object now stores all the output from the linear regression. Type `fit` to get the estimates of the α and β.

```
> fit

Call:
lm(formula = siri ~ abdomen, data = bodyfat)

Coefficients:
(Intercept)      abdomen
   -39.2802       0.6313
```

Of course, the `fit` object contains much more information. Using the `summary()` function, we can obtain the output similar to what R-Commander provides in above examples.

```
> summary(fit)

Call:
lm(formula = siri ~ abdomen, data = bodyfat)

Residuals:
     Min       1Q   Median       3Q      Max
-19.0160  -3.7557   0.0554   3.4215  12.9007

Coefficients:
             Estimate Std. Error t value
(Intercept) -39.28018    2.66034  -14.77
abdomen       0.63130    0.02855   22.11
            Pr(>|t|)
(Intercept)   <2e-16 ***
abdomen       <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.877 on 250 degrees of freedom
Multiple R-squared: 0.6617,        Adjusted R-squared: 0.6603
F-statistic: 488.9 on 1 and 250 DF,  p-value: < 2.2e-16
```

With the `names()` function, we can view all the information contained in the `fit` object.

```
> names(fit)

 [1] "coefficients"  "residuals"
 [3] "effects"       "rank"
 [5] "fitted.values" "assign"
 [7] "qr"            "df.residual"
 [9] "xlevels"       "call"
[11] "terms"         "model"
```

Now we can use the $ operator to access information. For instance, suppose we wanted the estimates of α and β:

```
> fit$coefficients

(Intercept)     abdomen
-39.2801847   0.6313044
```

Likewise, the estimated response values for all people in our sample are stored in the `fitted.values` object within `fit`. Suppose we wanted the estimates for the first 5 people:

```
> fit$fitted.values[1:5]

       1        2        3        4        5
14.50695 13.11808 16.21147 15.26451 23.85025
```

The differences between actual and estimated response values are stored in the `residuals` object within `fit`. Try finding the residuals of the first 5 people:

```
> fit$residuals[1:5]

        1         2         3         4
-2.206949 -7.018079  9.088529 -4.864514
        5
 4.849746
```

Adding the least-squares line to the scatterplot is easy with the `abline()` function:

```
> plot(bodyfat$abdomen, bodyfat$siri,
+     main = "Scatterplot for Percent Body Fat by Abdomen Circumference",
+     xlab = "Andomen Circumference",
+     ylab = "Percent Body Fat")
> abline(fit)
```

If we want to show the least-squares regression line as a dashed line, we use `abline(fit, lty=2)` instead. The `lty` obtain defines the line type.

We can also create a multiple linear regression model to predict percent body fat using abdomen circumference and height. As before, we use the `lm()` function, but now we include both explanatory variables on the right hand side of the formula. (We separate the explanatory variables with plus signs.)

```
> multReg <- lm(siri ~ abdomen + height,
+     data = bodyfat)
> summary(multReg)
```

```
Call:
lm(formula = siri ~ abdomen + height, data = bodyfat)

Residuals:
      Min        1Q    Median        3Q
-18.85134  -3.48247  -0.01562   3.09489
      Max
 11.16331

Coefficients:
             Estimate Std. Error t value
(Intercept) -14.31075    6.04265  -2.368
abdomen       0.64236    0.02759  23.283
height       -0.37053    0.08122  -4.562
            Pr(>|t|)
(Intercept)   0.0186 *
abdomen      < 2e-16 ***
height      7.95e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.695 on 249 degrees of freedom
Multiple R-squared: 0.6878,        Adjusted R-squared: 0.6853
F-statistic: 274.2 on 2 and 249 DF,  p-value: < 2.2e-16
```