UNIT-IV – Getting Started with R

4.1 Introduction to R

R is a programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team. R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac. This programming language was named **R**, based on the first letter of first name of the two R authors (Robert Gentleman and Ross Ihaka), and partly a play on the name of the Bell Labs Language **S**.

4.2 Features of R

The following are the important features of R −

- R is a well-developed, simple and effective programming language which includes conditionals, loops; user defined recursive functions and input and output facilities.
- R has an effective data handling and storage facility,
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis.
- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.
- One nice feature that R shares with many popular open source projects is frequent releases.
- Finally, one of the joys of using R has nothing to do with the language itself, but rather with the active and vibrant user community.

4.3 R Nuts and Bolts

4.3.1 Entering Input

At the R prompt we type expressions. The <- symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
```

```
> msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
x <- ##
```

The above one is incomplete expression. The # character indicates a comment. Anything to the right of the # (including the # itself) is ignored. This is the only comment character in R. Unlike some other languages, R does not support multi-line comments or comment blocks.

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.

```
> x <- 5          ## nothing printed

> x              ## auto-printing

[1] 5

> print(x)       ## explicit printing

[1] 5
```

The [1] shown in the output indicates that x is a vector and 5 is its first element. Typically with interactive work, we do not explicitly print objects with the print function; it is much easier to just auto-print them by typing the name of the object and hitting return/enter. However, when writing scripts, functions, or longer programs, there is sometimes a need to explicitly print objects because auto-printing does not work in those settings.

When an R vector is printed you will notice that an index for the vector is printed in square brackets [] on the side. For example, see this integer sequence of length 20.

```
> x <- 10:30

> x [1] 10 11 12 13 14 15 16 17 18 19 20 21

    [13] 22 23 24 25 26 27 28 29 30
```

The numbers in the square brackets are not part of the vector itself; they are merely part of the printed output.

4.3.2 R Objects

R has five basic or “atomic” classes of objects:

- character

- numeric (real numbers)
- integer
- complex
- logical (True/False)

The most basic type of R object is a vector. Empty vectors can be created with the vector() function. There is really only one rule about vectors in R, which is that a vector can only contain objects of the same class.

Numbers

- Numbers in R are generally treated as numeric objects (i.e. double precision real numbers).
- If you explicitly want an integer, you need to specify the L suffix. So entering 1 in R gives you a numeric object; entering 1L explicitly gives you an integer object.
- There is also a special number Inf which represents infinity. This allows us to represent entities like 1 / 0. This way, Inf can be used in ordinary calculations; e.g. 1 / Inf is 0.
- The value NaN represents an undefined value ("not a number"); e.g. 0 / 0; NaN can also be thought of as a missing value (more on that later)

Attributes

R objects can have attributes, which are like metadata for the object. These metadata can be very useful in that they help to describe the object. For example, column names on a data frame help to tell us what data are contained in each of the columns. Some examples of R object attributes are

- names, dim names
- dimensions (e.g. matrices, arrays)
- class (e.g. integer, numeric)
- length
- other user-defined attributes/metadata

Attributes of an object (if any) can be accessed using the attributes() function. Not all R objects contain attributes, in which case the attributes() function returns NULL.

4.3.3 Creating Vectors

The c() function can be used to create vectors of objects by concatenating things together.

```
> x <- c(0.5, 0.6) ## numeric
> x <- c(TRUE, FALSE) ## logical
> x <- c(T, F) ## logical
> x <- c("a", "b", "c") ## character
> x <- 9:29 ## integer
> x <- c(1+0i, 2+4i) ## complex
```

Note that in the above example, T and F are short-hand ways to specify TRUE and FALSE. You can also use the vector() function to initialize vectors.

```
> x <- vector("numeric", length = 10)
> x
[1] 0 0 0 0 0 0 0 0 0 0
```

4.3.4 Mixing Objects

There are occasions when different classes of R objects get mixed together. Sometimes this happens by accident but it can also happen on purpose.

```
> y <- c(1.7, "a") ## character
> y <- c(TRUE, 2) ## numeric
> y <- c("a", TRUE) ## character
```

In each case above, we are mixing objects of two different classes in a vector. But remember that the only rule about vectors says this is not allowed. When different objects are mixed in a vector, *coercion (compulsion)* occurs so that every element in the vector is of the same class.

4.3.5 Explicit Coercion

Objects can be explicitly coerced from one class to another using the **as.*** functions, if available.

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
> as.logical(x)
[1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Sometimes, R can't figure out how to coerce an object and this can result in NAs being roduced.

```
> x <- c("a", "b", "c")
> as.numeric(x)
```

```
Warning: NAs introduced by coercion
[1] NA NA NA
> as.logical(x)
[1] NA NA NA
> as.complex(x)
Warning: NAs introduced by coercion
[1] NA NA NA
```

When nonsensical coercion takes place, you will usually get a warning from R.

4.3.6 Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
[,1] [,2] [,3]
[1,] NA NA NA
[2,] NA NA NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Matrices are constructed *column-wise*, so entries can be thought of starting in the "upper left" corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
[,1] [,2] [,3]
[1,] 1 3 5
[2,] 2 4 6
```

Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
[1] 1 2 3 4 5 6 7 8 9 10
> dim(m) <- c(2, 5)
> m
[,1] [,2] [,3] [,4] [,5]
[1,] 1 3 5 7 9
[2,] 2 4 6 8 10
```

Matrices can be created by *column-binding* or *row-binding* with the cbind() and rbind() functions.

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
x y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x, y)
[,1] [,2] [,3]
x 1 2 3
y 10 11 12
```

4.3.7 Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well. Lists can be explicitly created using the list() function, which takes an arbitrary number of arguments.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1
[[2]]
[1] "a"
[[3]]
[1] TRUE
 [[4]]
[1] 1+4i
```

We can also create an empty list of a pre-specified length with the vector() function

```
> x <- vector("list", length = 5)
> x
[[1]]
NULL
[[2]]
NULL
[[3]]
NULL
[[4]]
NULL
[[5]]
NULL
```

4.3.8 Factors

Factors are used to represent categorical data and can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*. Factors are important in statistical modeling and are treated specially by modelling functions like lm() and glm().

Using factors with labels is *better* than using integers because factors are self-describing. Having a variable that has values "Male" and "Female" is better than a variable that has values 1 and 2. Factor objects can be created with the factor() function.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no yes no
Levels: no yes
> table(x)
x
no yes
2 3
> ## See the underlying representation of factor
> unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no" "yes"
```

Often factors will be automatically created for you when you read a dataset in using a function like read.table(). Those functions often default to creating factors when they encounter data that look like characters or strings.

The order of the levels of a factor can be set using the levels argument to factor(). This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x ## Levels are put in alphabetical order
[1] yes yes no yes no
Levels: no yes
> x <- factor(c("yes", "yes", "no", "yes", "no"),
+ levels = c("yes", "no"))
> x
[1] yes yes no yes no
Levels: yes no
```

4.3.9 Missing Values

Missing values are denoted by NA or NaN for q undefined mathematical operations.

- is.na() is used to test objects if they are NA
- is.nan() is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc.

- A NaN value is also NA but the converse is not true

```
> ## Create a vector with NAs in it
> x <- c(1, 2, NA, 10, 3)
> ## Return a logical vector indicating which elements are NA
> is.na(x)
[1] FALSE FALSE TRUE FALSE FALSE
> ## Return a logical vector indicating which elements are NaN
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
> ## Now create a vector with both NA and NaN values
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE TRUE TRUE FALSE
> is.nan(x)
[1] FALSE FALSE TRUE FALSE FALSE
```

4.3.10 Data Frames

Data frames are used to store tabular data in R. They are an important type of object in R and are used in a variety of statistical modeling applications. Hadley Wickham's package dplyr has an optimized set of functions designed to work efficiently with data frames.

Data frames are represented as a special type of list where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.

Unlike matrices, data frames can store different classes of objects in each column. Matrices must have every element be the same class (e.g. all integers or all numeric).

In addition to column names, indicating the names of the variables or predictors, data frames have a special attribute called row.names which indicate information about each row of the data frame.

Data frames are usually created by reading in a dataset using the read.table() or read.csv(). However, data frames can also be created explicitly with the data.frame() function or they can be coerced from other types of objects like lists.

Data frames can be converted to a matrix by calling data.matrix(). While it might seem that the as.matrix() function should be used to coerce a data frame to a matrix, almost always, what you want is the result of data.matrix().

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
```

```
  foo bar
1 1 TRUE
2 2 TRUE
3 3 FALSE
4 4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

4.3.11 Names

R objects can have names, which is very useful for writing readable code and self-describing objects. Here is an example of assigning names to an integer vector.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("New York", "Seattle", "Los Angeles")
> x
New York Seattle Los Angeles
1 2 3
> names(x)
[1] "New York" "Seattle" "Los Angeles"
```

Lists can also have names, which is often very useful.

```
> x <- list("Los Angeles" = 1, Boston = 2, London = 3)
> x
$`Los Angeles`
[1] 1
$Boston
[1] 2
$London
[1] 3
> names(x)
[1] "Los Angeles" "Boston" "London"
```

Matrices can have both column and row names.

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
c d
a 1 3
b 2 4
```

Column names and row names can be set separately using the colnames() and rownames() functions.

```
> colnames(m) <- c("h", "f")
> rownames(m) <- c("x", "z")
> m
h f
x 1 3
z 2 4
```

Note that for data frames, there is a separate function for setting the row names, the row.names() function. Also, data frames do not have column names, they just have names (like lists). So to set the column names of a data frame just use the names() function.

Object Set column names Set row names

data frame names() row.names()
matrix colnames() rownames()

4.4. Getting Data In and Out of R

4.4.1 Reading

There are a few principal functions reading data into R.

- read.table, read.csv, for reading tabular data
- readLines, for reading lines of a text file
- source, for reading in R code files (inverse of dump)
- dget, for reading in R code files (inverse of dput)
- load, for reading in saved workspaces
- unserialize, for reading single R objects in binary form

There are of course, many R packages that have been developed to read in all kinds of other datasets.

4.4.2 Writing

There are analogous functions for writing data to files

- write.table, for writing tabular data to text files (i.e. CSV) or connections
- writeLines, for writing character data line-by-line to a file or connection
- dump, for dumping a textual representation of multiple R objects
- dput, for outputting a textual representation of an R object

- save, for saving an arbitrary number of R objects in binary format (possibly compressed) to a file.
- serialize, for converting an R object into a binary format for outputting to a connection (or file).

4.5 Textual and Binary Formats for Storing Data

There are a variety of ways that data can be stored, including structured text files like CSV or tab-delimited, or more complex binary formats. However, there is an intermediate format that is textual, but not as simple as something like CSV. The format is native to R and is somewhat readable because of its textual nature.

One can create a more descriptive representation of an R object by using the dput() or dump() functions. The dump() and dput() functions are useful because the resulting textual format is editable, and in the case of corruption, potentially recoverable.

There are a few downsides to using these intermediate textual formats. The format is not very space efficient, because all of the metadata is specified. Also, it is really only partially readable. In some instances it might be preferable to have data stored in a CSV file and then have a separate code file that specifies the metadata.

4.5.1 Using dput() and dump()

One way to pass data around is by de-parsing the R object with dput() and reading it back in (parsing it) using dget().

```
> ## Create a data frame
> y <- data.frame(a = 1, b = "a")
> ## Print 'dput' output to console
> dput(y)
    structure(list(a = 1, b = "a"), class = "data.frame", row.names = c(NA, -1L))
```

Notice that the dput() output is in the form of R code and that it preserves metadata like the class of the object, the row names, and the column names.

The output of dput() can also be saved directly to a file.

```
> ## Send 'dput' output to a file
> dput(y, file = "y.R")
> ## Read in 'dput' output from a file
> new.y <- dget("y.R")
> new.y
  a b
1 1 a
```

Multiple objects can be de-parsed at once using the dump function and read back in using source.

```
> x <- "foo"
> y <- data.frame(a = 1L, b = "a")
```

We can dump() R objects to a file by passing a character vector of their names.

```
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
```

rm() **function in R** Language is used to delete objects from the memory.

The inverse of dump() is source(). The **source()** causes **R** to accept its input from the named file or URL or connection or expressions directly.

```
> source("data.R")
> str(y)
  'data.frame': 1 obs. of 2 variables:
  $ a: int 1
  $ b: Factor w/ 1 level "a": 1
> x
 [1] "foo"
```

4.5.2 Binary Formats

The complement to the textual format is the binary format, which is sometimes necessary to use for efficiency purposes, or because there's just no useful way to represent data in a textual manner. Also, with numeric data, one can often lose precision when converting to and from a textual format, so it's better to stick with a binary format.

The key functions for converting R objects into a binary format are save(), save.image(), and serialize(). Individual R objects can be saved to a file using the save() function.

```
> a <- data.frame(x = rnorm(100), y = runif(100))
> b <- c(3, 4.4, 1 / 3)
>
> ## Save 'a' and 'b' to a file
> save(a, b, file = "mydata.rda")
>
> ## Load 'a' and 'b' into your workspace
> load("mydata.rda")
```

If you have a lot of objects that you want to save to a file, you can save all objects in your workspace using the save.image() function.

```
> ## Save everything to a file
> save.image(file = "mydata.RData")
>
> ## load all objects in this file
> load("mydata.RData")
```

Note: used the .rda extension when using save() and the .RData extension when using save.image(). You can use whatever file extension you want. The save() and save.image() functions do not care. However, .rda and .RData are fairly common extensions and you may want to use them because they are recognized by other software.

The serialize() function is used to convert individual R objects into a binary format that can be communicated across an arbitrary connection. This may get sent to a file, but it could get sent over a network or other connection. When you call serialize() on an R object, the output will be a raw vector coded in hexadecimal format.

```
> x <- list(1, 2, 3)
> serialize(x, NULL)
[1] 58 0a 00 00 00 02 00 03 02 01 00 02 03 00 00 00 00 13 00 00 00 03 00
[24] 00 00 0e 00 00 00 01 3f f0 00 00 00 00 00 00 00 00 00 0e 00 00 00 01
[47] 40 00 00 00 00 00 00 00 00 00 00 0e 00 00 00 01 40 08 00 00 00 00 00
[70] 00
```

If you want, this can be sent to a file, but in that case you are better off using something like save(). The benefit of the serialize() function is that it is the only way to perfectly represent an R object in an exportable format, without losing precision or any metadata. If that is what you need, then serialize() is the function for you.

4.6 Managing Data Frames with the dplyr package

4.6.1 Data Frames

The data frame is a key data structure in statistics and in R. The basic structure of a data frame is that there is one obsevation per row and each column represents a variable, a measure, feature, or characteristic of that observation.

4.6.2 dplyr package

The dplyr package is designed to mitigate a lot of problems like filtering, re-ordering, and collapsing with data frames. This package provides a highly optimized set of routines specifically for dealing with data frames.

One important contribution of the dplyr package is that it provides a “grammar” (in particular, verbs) for data manipulation and for operating on data frames. With this grammar, you can sensibly communicate what it is that you are doing to a data frame that other people can understand (assuming they also know the grammar). This is useful because it provides an abstraction for data manipulation that previously did not exist. Another useful contribution is that the dplyr functions are very fast, as many key operations are coded in C++.

4.6.2.1 dplyr Grammar

Some of the key “verbs” provided by the dplyr package are

- select: return a subset of the columns of a data frame, using a flexible notation
- filter: extract a subset of rows from a data frame based on logical conditions
- arrange: reorder rows of a data frame
- rename: rename variables in a data frame
- mutate: add new variables/columns or transform existing variables
- summarise / summarize: generate summary statistics of different variables in the data frame, possibly within strata

- %>%: the “pipe” operator is used to connect multiple verb actions together into a pipeline.

The dplyr package as a number of its own data types that it takes advantage of. For example, there is a handy print method that prevents you from printing a lot of data to the console.

4.6.2.2 Common dplyr Function Properties

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame specified in the first argument, and you can refer to columns in the data frame directly without using the $ operator (just use the column names).
3. The return result of a function is a new data frame
4. Data frames must be properly formatted and annotated for this to all be useful. In particular, the data must be tidy. In short, there should be one observation per row, and each column should represent a feature or characteristic of that observation.