R403: Probabilistic and Statistical Computations with R

Topic 1: Introduction to R. Common Data Types and Structures

Kaloyan Ganev

2022/2023

Introductory Notes

History of R

- R is a dialect of the S language
- S was developed in the 1970s in the Bell Labs
- Initially written in Fortran, rewritten in C in the late 1980s
- Currently available through a commercial product called S-PLUS
- R was developed in 1991 by Ross Ihaka and Robert Gentleman (University of Auckland, New Zealand)
- First public announcement: 1993
- GNU GPL: 1995
- Version 1.0.0: 2000

How to Get the Software

- Available for free from https://www.r-project.org/
- All major OS's supported
- Lots of contributed packages available from CRAN mirrors
- Non-CRAN packages will most probably not be used throughout the course
- Check out documentation links on the R Project site for what R and additional packages do

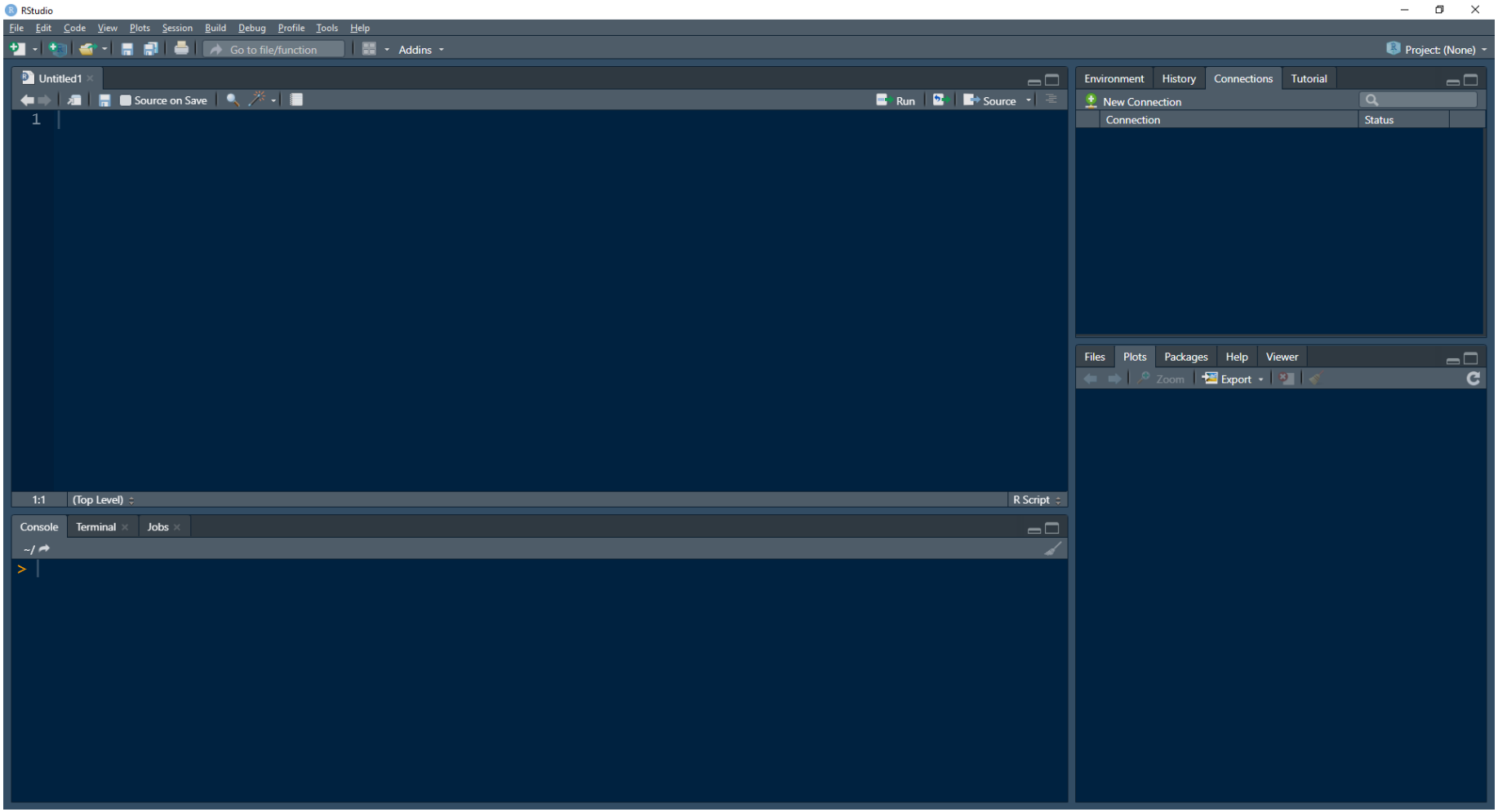
GUIs

- Although R is command-line driven, there are some GUIs that provide more convenience in working with it
- Some examples:
  - R Commander (http://socserv.mcmaster.ca/jfox/Misc/Rcmdr/)
  - Deducer (http://www.deducer.org/pmwiki/index.php?n=Main.DeducerManual?from=Main.HomePage)
  - JGR (http://www.rforge.net/JGR/)
  - RKWard (https://rkward.kde.org/)
  - RStudio (https://www.rstudio.com/)
  - Jamovi (https://www.jamovi.org)
  - BlueSky Statistics (https://www.blueskystatistics.com/)

Advantages of R

- Free and open-source
- Flexible: you can do whatever you want, no pre-programmed black boxes
- Pre-built packages so that you don't have to reinvent the wheel
- Enormous graphing capabilities, publication-quality graphical output
- Active world-wide community (check out for example Stack Overflow, Stack Exchange, etc.)
- Huge amounts of online resources (manuals, videos, hints, etc.), many of them freely available
- etc.

RStudio: a Quick Overview



Basic Capabilities of R

Before We Begin...

- Set your working directory: this is where R reads and writes files (if you don't instruct it otherwise explicitly)
- By default, the working directory is your home directory; type to see:

```
getwd()
```

- To permanently change the working directory, in RStudio go to `Tools -> Global Options...` and there click Browse next to `Default working directory`
- If you would like to change it on a case-by-case basis, use:

```
setwd(<path to directory>)
```

The RStudio Console

- This is where commands are typed and issued to R
- Commands are written and executed one at a time
- A simple command:

```
dir()
```

to list the contents of your current working directory

The RStudio Console (2)

- R as a calculator:

```
5 + 6
3 * 4
12 / 7
2 ^ 5
11 %% 5
1 / Inf
0 / 0
```

- Doing this, however, you immediately lose the result (cannot use it later)

Objects and Object Types

- R objects: containers (memory locations, to be fully correct) to keep information for later usage
- Basic types of objects:
  - Vectors
  - Matrices
  - Lists
  - Arrays
  - Factors
  - Data frames

Assigning Values to Objects

- Assignment: putting something in the container; examples:

```
x1 <- 10 # numeric
x2 <-5L # integer
x3 <- 6 + 2i # complex
y <- FALSE # logical
z <- "Hello World!" # character
```

- Print objects: type name or type

```
print(<object name>)
```

Object Attributes

- Use

```
attributes(<object name>)
```

  to explore; for example:

```
z1 <- rnorm(625)
z2 <- matrix(z1, nrow = 25)
attributes(z2)
```

- Attributes can include names, length, dimensions, dimnames, class, etc.

Basic Data Types

Basic Data Types

- You saw them already on the previous slides:
  - Numeric
  - Integer
  - Complex
  - Logical
  - Character
- Those are also called *atomic data types*
- Such “atoms” are used to construct various objects and data structures in R

Basic Data Types (2)

- To get the type of an object:

```
mode(<name of object>) # low level
class(<name of object>) # high level
```

- To check whether an object is of a specific type, e.g.:

```
is.numeric(x) # returns TRUE or FALSE
```

Type Coercion

- Means forceful transforming one data type into another
- Check out:

```
as.numeric(FALSE) # Try also with TRUE
as.character(102)
as.integer(105.88)
as.complex(3.14)
as.numeric("458")
as.numeric("This is text.")
```

- Same logic followed for all other objects, too

The Workspace

- Once created, an object is placed in the workspace
- It stays there until removed, e.g. with the `rm()` command
- Can be seen in the top-right tab labelled “Environment”
- Can also be listed by using the command line:

```
ls()
```

- You can set various options for working with R, e.g. the number of digits to display after the decimal sign:

```
options(digits = 2)
```

- Check out also:

```
help(options)
```

R Scripts

- In essence, text files to hold R commands
- Saved with the .R extension
- Provide convenience and reproducibility
- In RStudio, they can be executed line by line, in blocks, or entirely
- To include an existing script into a new script:

```
source(<script name>)
```

Vectors, matrices and arrays

Vectors in R

- Unlike in mathematics where vectors are strictly comprised of numbers, in R vectors are ordered collections of data elements
- In other words, you can have a numeric vector, an integer vector, a character vector, etc.
- Created with the `c()` command:

```
x1 <- c(1, 2, 3, 4)
x2 <- seq(1, 10)
x3 <- seq(from = 2, to = 3, by = 0.01)
y <- c("One", "Two", "3")
z <- c(TRUE, FALSE, T, F)
```

Vectors in R (2)

- Note that R treats single elements also as vectors!

```
is.vector(x3)
is.vector(3)
is.vector("text")
```

- What is specific is that a vector can contain only one basic type of data
- For example, a vector can hold only complex numbers; every other number is converted (coerced) to complex

```
y <- c(1.02, 3 + 0.5i) # Check out the next line, too}
y <- c(1.02, 3 +0.5i, "text")
```

Vectors in R (3)

- Vectors in real-life applications contain information on something
- In order to not forget what info is contained in each element, elements can be named in R

```
lunch_costs <- c(3.30, 5.91, 2.75)
names(lunch_costs) <- c("Soup", "Main course", "Dessert")
print(lunch_costs)
str(lunch_costs)
```

Creating Empty Vectors

- One way to create is with the `vector()` function
- You need to specify type (numeric, logical, character, integer, double):

```
em1 <- vector("numeric")
em2 <- vector("character")
em3 <- vector("logical")
```

- Note that some default values are imputed if you additionally specify length as an argument

```
nem1 <- vector("numeric", length = 20)
```

- You can get empty vectors also through:

```
em1a <- numeric()
em2a <- character()
em3a <- logical()
```

Mathematical Operations with Vectors

- All operations are carried out element-wise; check out:

```
v1 <- c(1, 2, 3)
v2 <- c(4, 5, 6)
v1 + 3.5
v1 * 2
v1 / 3
v1 - v2
v1 * v2
v1 ^ v2
```

Mathematical Operations with Vectors (2)

- What if vectors are of different sizes? Recycling

```
v3 <- c(7, 8, 9, 10)
v4 <- v1 + v3
v5 <- v1 * v3
```

- Safest strategy: avoid recycling if possible

Element Sums and Vector Comparison

- If you need to find the sum of all elements in a vector:

```
total_lunch <- sum(lunch_costs)
total_lunch
```

- Compare with another vector:

```
lunch_costs_alt <- c(3.10, 6.25, 2.90)
lunch_costs > lunch_costs_alt
```

Factors

- Statistical models work with two types of variables: quantitative and qualitative
- Quantitative variables are expressed in numbers in a straightforward manner
- In general such variables are continuous range and can therefore have infinitely many values[1]
- Qualitative variables, at the same time, usually can take on only a limited number of values
- Those values are usually non-numerical, and are known as *categories*[2]
- Since computers understand only numbers, categorical values need to be converted to numbers (usually non-negative digits)

[1] Sometimes, their value range can be though restricted.

[2] Therefore quality variables are often mentioned as *categorical variables*.

Factors (2)

- In R, categorical variables are implemented through the usage of *factors*
- Each possible value (category) of a factor is known as a *level*
- Levels are stored as integers
- Factors can be *unordered* or *ordered*
- Ordering could be imposed by the nature of the data (e.g. weekdays, months, temperature, etc.)

Factors (3)

- In the following example, a qualitative variable describing the hair colour of a sample of people is created:

```
hair_col <- c("black", "brown", "black", "blond", "red", "brown
    ", "black")
hair_col_f <- factor(hair_col)
hair_col_f
```

- This is an unordered factor: it does not matter which colour is put first, second, etc.

Factors (4)

- To see how an ordered one can be created, see the next example:

```
temp <- c("freezing", "warm", "hot", "cold", "warm", "freezing"
    )
temp_f <- factor(temp, ordered = TRUE, levels = c("freezing", "
    cold", "warm", "hot"))
temp_f
```

- The way categories are displayed can be changed, too, by changing the labels of categories:

```
temp_f[2] < temp_f[4] # Comparison possible
temp_f <- factor(temp, ordered = TRUE, levels = c("freezing", "
    cold", "warm", "hot"),
labels = c("very cold", "cold", "warm", "hot")) # labels
    changed here
temp_f
```

Matrices in R

- Matrices are an extension of vectors to two dimensions
- All other properties are maintained, incl. single basic data type and automatic coercion
- One way to create:

```
m1 <- matrix(1:12, nrow = 3) # or,
m1 <- matrix(1:12, ncol = 4)
```

- Note that elements are filled column by column by default; if you want it by rows:

```
m2 <- matrix(1:12, ncol = 4, byrow = TRUE)
```

Matrices in R (2)

- Another way to create:

```
m3 <- cbind(lunch_costs, lunch_costs_alt)
```

- Change column names:

```
colnames(m3) <- c("Costs, option 1", "Costs, option 2")
```

- Add a row of totals:

```
m3a <- rbind(m3, colSums(m3))
rownames(m3a)[4] <- "Total"
```

- `rowSums()` is analogical to `colSums()` and also self-explanatory

Subsets (Slices) of Matrices

- Analogical to vectors, only one more dimension is added:

```
m3a[,1]
m3a[2,]
m3a[c(1, 3), 1]
m3a[4, 2]
m3a["Soup", "Costs, option 2"]
m3a[c(T, F, F, F), c(T,F)]
```

Mathematical Operations with Matrices

- Again, analogical to operations with vectors
- Check out:

```
m1 + 5
m1 * 3
m1 %% 3
m1 ^ 2
```

- Two or more matrices: element-wise!

```
m1 + m2
m1 / m2
m1 %% m2
m1 ^ m2
```

'Traditional' Matrix Algebra in R

- Transposition:

```
t(m1)
```

- Multiplication (possible if matrices are conformable):

```
t(m1) %*% m2 # equivalent to
crossprod(m1,m2)
```

'Traditional' Matrix Algebra in R (2)

- Inverse matrix (if matrix square and non-singular):

```
m4 <- matrix(1:4, nrow = 2)
solve(m4)
```

- Eigenvalues and eigenvectors:

```
eig <- eigen(m4)
eig$val
eig$vec
```

Arrays in R

- Vectors are one-dimensional arrays
- Matrices are two dimensional arrays
- Higher dimensions are also possible
- A 3D array:

```
array1 <- array(1:343, dim = c(7, 7, 7))
```

- You can name dimensions using `dimnames()`

Lists and Data Frames

Lists in R

- Vectors, matrices and arrays can hold information that is only of one type
- When it is necessary to have more than one data type, lists are used
- Created with the `list()` command
- Example:

```
list1 <- list(TRUE, "France", 21)
```

Lists in R (2)

- Can be named

```
names(list1) <- c("Active", "Country", "Age")
```

- Can also hold other complex structures such as vectors, matrices, etc., even other lists

```
list2 <- list(Appearances = c(T,F,T,T,T), Attributes = list1)
```

Subsetting Lists

- Single square brackets return a list

```
list3 <- list2[1]
```

- Double square brackets extract the generic element; in the current example: a vector:

```
vec1 <- list2[[1]]
```

Subsetting Lists (2)

- The same is achieved with the $ sign:

```
vec2 <- list2$Appearances
```

- The resulting vectors (or other types) can be further indexed to get their elements

```
list2[[1]][1]
list2$Appearances[1]
```

Subsetting lists (3)

- Subsetting is also possible by name:

```
list2["Attributes"]
```

- Also, by logical values:

```
list1[c(FALSE, TRUE, FALSE)]
```

Extending Lists

- Easiest done by means of the $ sign

```
teams <- c("Real Madrid", "Barcelona", "Valencia")
list2$clubs <- teams
```

- Using single and double square brackets also possible although somewhat more difficult

Data Frames

- A special type of list
- Contains vectors of the same length (same type not required)
- Correspond to the notion of datasets in other statistical packages
- Variables correspond to columns, while observations correspond to rows

Data Frames (2)

- Created from other objects, e.g. matrices:

```
m4df <- matrix(1:9, nrow = 3)
df1 <- as.data.frame(m4df)
```

- Name variables:

```
names(df1) <- c("One", "Two", "Three")
```

- Subsetting uses both vector and list approaches

Extending Data Frames

- Columns (new variables) and rows (new observations) can be added
- For example:

```
df1$Four <- c(10, 11, 12)
totals <- data.frame(matrix(c(6, 15, 24, 33),
  nrow = 1, ncol = 4))
names(totals) <- c("One", "Two", "Three", "Four")
df2 <- rbind(df1, totals)
```

- Data frames can also be directly read from external sources (text, csv, databases, etc.)
- This is also the most common data structure used in statistical applications