R-Bootcamp (day 3)

Dr. Matteo Tanadini, Claude Renaux & Co.

31 January - 3 February 2022

Outline

Section 1

Fitting models

Fitting Models (1)

We learned how to:

- read data (most often as a dataframe object)
- prepare data (we ... scratched the surface)
- visualise data

⇒ let's start fitting models

Fitting Models (2)

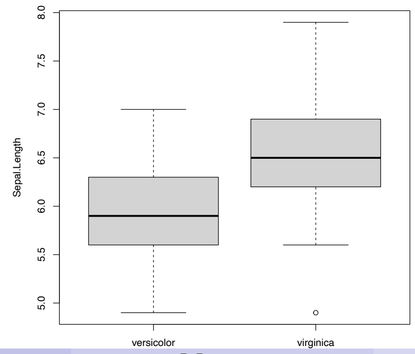
The simplest statistical model:

Fitting Models (2)

The simplest statistical model:
The t-test to compare the mean of two groups of observations

```
d.iris.2.sp <- iris[iris$Species != "setosa", ]
```

```
boxplot(Sepal.Length ~ Species, data = d.iris.2.sp)
```



Fitting Models (3)

The simplest statistical model:

The t-test to compare the mean of two groups of observations

```
t.test(Sepal.Length ~ Species, data = d.iris.2.sp)


Welch Two Sample t-test

data:  Sepal.Length by Species
t = -6, df = 94, p-value = 2e-07
alternative hypothesis: true difference in means between group versicolor and group
95 percent confidence interval:
 -0.88 -0.42
sample estimates:
mean in group versicolor  mean in group virginica
                     5.9                      6.6
```


Fitting Models (4)

A slightly more complex statistical model: The linear model

```
lm.iris <- lm(Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width + Species,
              data = d.iris.2.sp)
```

- usually, when fitting a model, you store it as an object[1].
- note the "formula interface"

[1] The t-test models are one of the very few exceptions, where the model itself is not stored, but rather directly printed.

Fitting Models (5)

Getting the results of a fitting process: the `summary()` function.

```
summary(lm.iris)


Call:
lm(formula = Sepal.Length ~ Sepal.Width + Petal.Length + Petal.Width +
    Species, data = d.iris.2.sp)

Residuals:
    Min      1Q  Median      3Q     Max
-0.7759 -0.2451  0.0102  0.2586  0.7568

Coefficients:
                 Estimate Std. Error t value Pr(>|t|)
(Intercept)        1.5898     0.3319    4.79  6.1e-06 ***
Sepal.Width        0.3314     0.1320    2.51   0.0138 *
Petal.Length       0.8969     0.0767   11.69  < 2e-16 ***
Petal.Width       -0.2962     0.1783   -1.66   0.1000
Speciesvirginica  -0.3671     0.1352   -2.71   0.0079 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.33 on 95 degrees of freedom
Multiple R-squared:  0.767,Adjusted R-squared:  0.758
F-statistic: 78.4 on 4 and 95 DF,  p-value: <2e-16
```

Question: What type of object is a linear model?

Fitting Models (5)

Further "model-related" functions (see democode)

- `anova()` to compare two (or more) models[2]
- `update()` to modify an existing model
- `fitted()` to get the fitted values
- `residuals()` to get the residuals
- `plot()` to show the model diagnostics/fit
- ...

[2] `anova()` is a very "generic" function that does not only perform ANOVA tests!

Fitting Models (6)

Good practices:

- store models in objects
- use the formula notation (e.g. "y $\sim a + b + c$")
- avoid using "y $\sim$." in model formulas[3]
- name models in a comprehensible and helpful way ("mod.iris.interaction" is preferred over "mi2")
- name objects "derived" from models accordingly (e.g. "res.mod.iris.interaction")

[3] This may create issues when the data is modified. Note, however, that "y $\sim$." can be safely used when using graphical functions such as `pairs()`.

Section 2

Missing values

Missing values (1)

- in **R** missing values are coded as `NA` (remember: **R** is case-sensitive)
- `NA`s can be set manually
- OR ... most often `NA`s come from empty cells in spreadsheets

Missing values (2)

Finding missing values

```
v.1 <- c(1:4, NA, 67, 3:1)
v.1

[1]  1  2  3  4 NA 67  3  2  1

##
anyNA(v.1)

[1] TRUE

is.na(v.1)

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

which(is.na(v.1))

[1] 5
```


Missing values (3)

What happens when you handle missing values?

```
v.1 <- c(1:4, NA, 67, 3:1)
##
mean(v.1)

[1] NA

##
mean(v.1, na.rm = TRUE)

[1] 10
```

"Simple" functions can drop the missing values.

Missing values (4)

"higher" functions, such as `lm()` have the "na.action" argument (in most cases set to "na.omit")

When fitting e.g. a linear model, all rows of your data that contain at least a missing value will be dropped!
⇒ Sample can decrease dramatically.

There are three possible solutions:

- do nothing
- discard the columns/predictors that contain many missing values
- impute missing values

Missing values (5)

A possible way to deal with missing values is to impute them

- imputation allows us to use all observations when fitting models
- **However** imputation can introduce errors and bias (see **R**-code)
- imputation must be done with care (and only when really needed)
- alternatively, you can create a new class (e.g. "unknown")

Missing values (6)

Good practices:

- make sure missing values are coded as such (e.g. not as -999)
- create "NA-free" datasets to fit models (rather than use "na.action")
- impute missing values only if really needed
- prefer creating the new class "unknown" when possible

Section 3

Packages

Packages (1)

By default, **R** comes with a few hundreds objects such as functions and datasets. These objects are stored in half a dozen packages. These packages, which are written by the **R**-core team, represent **R** itself (its core). For example:

- the function `plot()` is contained in package {*graphics*}
- the function `lm()` is contained in package {*stats*}
- the dataset `iris` is contained in package {*datasets*}

Packages (2)

In addition to the "basic" (or "default") **R**-packages, one can freely download an use tens of thousand add-on packages.

- using add-on packages is the rule, not the exception
- The vast majority of add-on **R**-packages are available on CRAN (https://cran.r-project.org/)
- A "task views" page exists on CRAN
- add-on packages come with a "reference manual"[4]
- some add-on packages come with one (or several) vignette(s), which are very handy introductions to the package

[4] A "reference manual" contains a brief package description and the help page for each object present in the package itself.

Packages (3)

One of the greatest things about **R** is that there are thousands of add-on packages available

One of the worst things about **R** is that there are thousands of add-on packages available

Packages (4)

+ thousand of functions are freely available (e.g. modelling functions or graphical functions)
+ most of the time, any new statistical method is (fairly soon) implemented in **R**
+ the source code is available (open-source concept)
+ the user does not need to implement many functions by him/herself
- the quality of add-on packages is variable (from great to ...)
- it can be difficult to navigate through the huge variety of packages that deal with a certain topic (e.g. ROC analysis)
- packages can change over time (sometimes in a undesired direction)
- packages can get "abandoned" over time

Packages (5)

Quality depends on the authors ...

A package is a **GOOD** package if (any of these):

- the author/maintainer is a member of the **R**-core team (check email via `maintainer("packageName")`)
- the package was published in the "Journal of Statistical Software" (google or see references)[5]
- is a "recommended" package according to CRAN (see "priority" on CRAN)
- has tests

[5] JSS really checks that the package is doing what the authors claim. CRAN does only some "structure" checks.

Packages (6)

Quality depends on the authors...
The package is a **likely to be a GOOD** package if (any of them):

- the author/maintainer is a member of a statistics department (again check email)
- the author works for a large pharma company
- comes with a companion book[6]
- it was downloaded many times and is a popular package

[6] Note, however, that not all famous books are necessarily good. The "R-book" is likely the most infamous example.

CRAN

☞ Go to CRAN (`https://cran.r-project.org/`) ☞

Packages (7)

Installing and loading add-on packages

- via commands
  - `install.packages("boot")`
  - `library("boot")`
- via Rstudio
  - Packages pane → "Install"
  - Packages pane → check the package on the list

Packages (8)

Loading packages can lead to "conflicts" as different packages can contain objects with the same name (e.g. the `gam()` function exists in {*gam*} but also in {*mgcv*} package)

Packages (9)

Good practices:

- Load packages at the beginning of your code
- periodically update packages (same for **R**)
- avoid conflicts