Workshop 1

Getting started with R, Rstudio and Git

Antoine Vernet

Today's plan

- R basics
- GUI and IDE
- Rmarkdown
- Git

R programming

The Hitchhiker's Guide to the Galaxy (trailer 1)

R is cool!

Why?

- R is a complete programming language
  - you can do statistics and more! (e.g. web scraping)
- the community is huge and growing!
  - lots of packages
  - lots of code examples and tutorials
  - lots of places to find help
- many companies are seeking skilled R analysts!

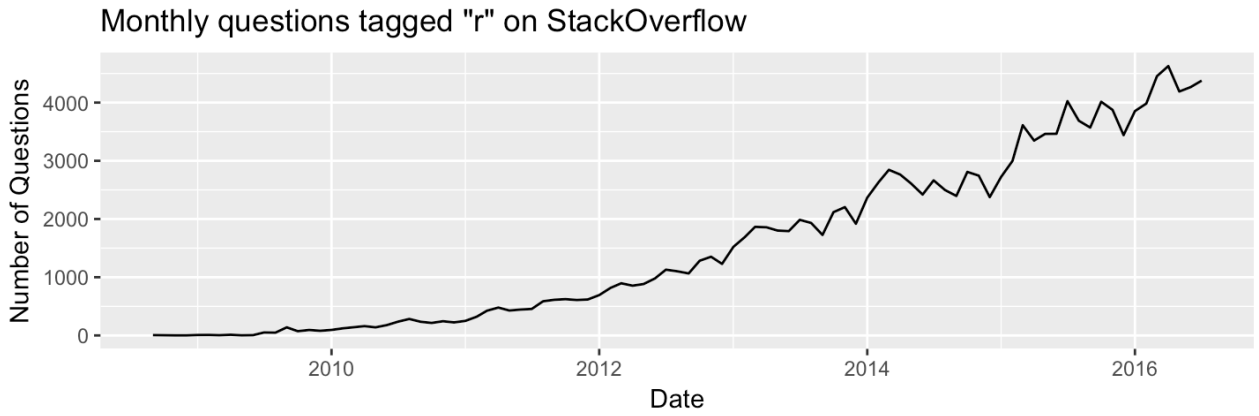
Examples of things made with R

- Scientific papers
- Websites
- Web apps: example 1, example 2
- these slides

Getting help with R

There are many places to get help with R. The main ones are:

- Rseek, for a version of google that understands that R is not a letter.
- Stack Overflow, for programming questions
- Cross Validated, for statistics questions
- R mailing lists, for everything else



Other places to find information and help

- R-bloggers, for the freshest news about the R community
- Coursera, particularly the classes by the team from John Hopkins University
- Datacamp, remember that you have free access to Datacamp with the class.

RStudio

The standard R gui is underwhelming. Rstudio offers lots of shortcuts and helpful integration:

- code completion
- git integration
- debugging
- package development tools
- authoring (Rmarkdown)

You can learn more about RStudio (and R) by reading the documentation on the RStudio website and by watching their webinars

RStudio projects

RStudio allows you to create projects which:

- conserve the state of your R session
- you can customise your interface at the project level
- help with git integration

Practice

- Fire up RStudio
- Create a project
- Create an associated git repository

R basics

Assignement

The assignement in R is not = it is <–

```
an_object <- some_results_of_an_operation
```

Vectors

A vector is a unidimensional data structure

```
v <- c(1:5)
v
```

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

```
is.vector(v)
```

```
## [1] TRUE
```

Scalars

R does not have scalars so uses vector of length 1 instead to represent single numbers

```
v <- 1
is.vector(v)
```

```
## [1] TRUE
```

```
mode(v)
```

```
## [1] "numeric"
```

```
typeof(v)
```

```
## [1] "double"
```

Matrices

Matrices are two dimensional arrays

```
matrix <- matrix(data = c(1:6), nrow = 2, byrow = TRUE)

matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Types

Matrix and vectors can only hold data of one type. If you try to create a vector or matrix with data of different type, it will be of the type that allows to represent all elements of the matrix

```
matrix1 <- matrix(data = c(1:6), nrow = 2, byrow = TRUE)
matrix2 <- matrix(data = c(1, 2, 3, 4, 5, 3.14), nrow = 2,
                  byrow = TRUE)
matrix3 <- matrix(data = c(1:5, "Marmalade"), nrow = 2, byrow = TRUE)
```

Types (con't)

```
matrix1
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```
matrix2
```

```
##      [,1] [,2] [,3]
## [1,]    1    2 3.00
## [2,]    4    5 3.14
```

```
matrix3
```

```
##      [,1] [,2] [,3]
## [1,] "1"  "2"  "3"
## [2,] "4"  "5"  "Marmalade"
```

Types (con't)

```
typeof(matrix1)
```

```
## [1] "integer"
```

```
typeof(matrix2)
```

```
## [1] "double"
```

```
typeof(matrix3)
```

```
## [1] "character"
```

Data frame

Data frames are two dimensional arrays (like matrices) but they can hold various types of data

```
dt <- data.frame(name = c("Julia", "Simon", "Christina"),
                 age = c(24, 26, 19),
                 gender = c("Female", "Male", "Female"))
dt
```

```
##        name age gender
## 1     Julia  24 Female
## 2     Simon  26   Male
## 3 Christina  19 Female
```

Data frame (con't)

Within each column, every element has to have the same type.

There are two important function that are useful with data frames (and other types): `summary` and `str`

```
summary(dt)
```

```
##         name        age          gender
##  Christina:1   Min.   :19.0   Female:2
##  Julia    :1   1st Qu.:21.5   Male  :1
##  Simon    :1   Median :24.0
##                Mean   :23.0
##                3rd Qu.:25.0
##                Max.   :26.0
```

Data frame (con't)

```
str(dt)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ name  : Factor w/ 3 levels "Christina","Julia",..: 2 3 1
##  $ age   : num  24 26 19
##  $ gender: Factor w/ 2 levels "Female","Male": 1 2 1
```

Lists

Lists are very useful because of their flexibility, but this also makes them sometimes difficult to work with.

Elements of a lists can be of any type (even list)

```
l <- list(c(1:3), "Today", matrix(1:6, nrow = 2))
l
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "Today"
##
## [[3]]
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Accessing elements of a list

Two ways: `[[]]` and `[]`, which do not produce the same result

```
l[[1]]
```

```
## [1] 1 2 3
```

```
l[1]
```

```
## [[1]]
## [1] 1 2 3
```

Accessing elements of a list (con't)

Two ways: `[[]]` and `[]`, which do not produce the same result

```
typeof(l[[1]])
```

```
## [1] "integer"
```

```
typeof(l[1])
```

```
## [1] "list"
```

Missing values

R codes missing values as `NA`

It also has codes for not-a-number: `NaN`

And `NULL`

Think carefully about these when you are using/writing functions as they might not all evaluate to the same result.

Packages

Much of R strenght rely on the ecosystem of packages that it comes with.

In many languages, library is the term used to describe what an R package is (confusingly, you load a package in memory using the function `library()`).

What is a package?

It is a collection of functions that allow you to perform specific types of operations, usually because they are not available among the functions the software comes prepackaged with.

R has packages for tasks like data loading, data wrangling, visualisation, modelling and many more tasks.

Installing packages

Packages are installed with the function `install.packages()` and updated with `update.packages()`

```
install.packages("name of the package")
install.packages(c("package_1", "package_2"))
```

Once you have installed a package, you need to load it before you can use the functions in it.

```
library(name_of_the_package)

# What most people do (me included)
function_from_package(args)

# What you should do in your scripts
# This is safe if two packages have a function with the same name
name_of_the_package::function_from_package(args)
```

Loading and writing data

Depending on the files, R offers a lot of facilities to load and write data. We will explore some of them in the following few sessions.

CSV files can be easily read in and written using `read.csv` and `write.csv`

```
data <- read.csv("Name_of_file.csv")
write.csv(Object_Name, "Path/to/file.csv")
```

Associated functions are `read.delim`, `read.table`.

Loading and writing RData files

RData is R's native format. It has some advantages:

- Compressed
- Versatile (you can save more than 1 object and their properties are preserved)

And some disadvantages:

- Only R will read the format natively (the format is open though)

```
data <- load("Name_of_file.RData")
save(Object_Name, "file.RData")
```

Base R versus packages

This is how you do it in base R, but there are many packages to read in data. Let's have a quick look at the most common ones.

- `readr` from the `tidyverse`
- `data.table`

`readr` from the `tidyverse`

```
data <- read_csv("Name_of_file.csv")
write_csv(Object, "filename.csv")
```

Very similar to base R but:

- Faster
- Produce `tibbles`, don't convert character to factor (double-edged sword), don't botch the column names
- Does not depend on your operating system (some base R behaviour is OS dependent)

`data.table`

```
data <- fread("Name_of_file.csv")
```

- Detect separator and column classes automatically
- Really fast (faster than `readr`)

Tibble

We introduced R `data.frame` earlier. Tibbles are a special case of dataframes.

Tibbles:

- Do not change variable names
- Do not convert character to factor

Tibble (2)

If, like me, you use a combination of tidyverse and non-tidyverse packages, you can convert back and forth between the two types:

```
a <- data.frame(b = c(1, 2, 3), c = c(4, 5, 6))

tib <- as.tibble(a)

df <- as.data.frame(tib)
```

This however requires that you keep all names of variables syntactic.

Subsetting and printing tibbles

- Tibble are better formatted than traditional dataframes when printed to the console
- You can subset tibbles using the pipe (but usually, we will use `filter` and `select` from the `dplyr` package)

Other types of data

Many packages allow you to read and write other types of data, below are a few that I find useful and are in wide use:

- `foreign` to read Stata and SPSS files
- `haven` to read Stata, SPSS and SAS files, more recent than foreign and part of the `tidyverse`
- `readxl` to read Excel files (provided they are well formatted), part of the `tidyverse`
- `DBI` to run queries against a database (you will need a database backend, such as `RMySQL`)
- `jsonlite` for json
- `xml2` for xml

A note about formats

Schematically, there are two large families of formats:

- proprietary
- open

Open format are usually longer lived than proprietary format and because their specifications are public one could always write a new parser if needed.

Text formats

Textual formats (.txt, .csv, .md, .Rmd, .html) are open formats. They have two main advantages:

- they are portable
- they can be easily version controlled

They present some disadvantages:

- not always compact
- might require extra code to load/write

```
#for example, for a csv file:
data <- read.csv("./path/to/file.csv", stringsAsFactors = FALSE)

#for RData format
load("./path/to/file.RData")
```

Literate programming

R Markdown

R Markdown is a flavour of Markdown

It allows to write literate programming document that can be rendered in different format (e.g. HTML, PDF, Word)

We will use R Markdown documents heavily in this class (Usually, I will give you a template).

Because it is a text file, it can be version controlled.

R Markdown basic

An R Markdown file usually get the extension .Rmd and is a simple text file. There are two main components to an R Markdown file:

- the YAML header, which allows you to customise the way the document is rendered
- the body, which contains your text and your code.

YAML header

The YAML header is where you put all the options to render your document. For example, the header for these slides looks like this:

```
---
title: "Workshop 1"
author: "Antoine Vernet"
date: ""
output:
  ioslides_presentation:
    css: style.css
  beamer_presentation:
    slide_level: 2
subtitle: Getting started with R, Rstudio and Git
always_allow_html: yes
---
```

YAML header (2)

Headers can be pretty simple like this one, or more elaborate

```
---
title: 'STRENGTHENING THE TIES THAT BIND'
author: Antoine Vernet
date: ''
output:
  pdf_document:
    includes:
      in_header: header.tex
    keep_tex: yes
  html_document: default
  word_document:
    pandoc_args: --smart
    reference_docx: style.docx
bibliography: ../../Bibtexfiles/Dyna_network_paper.bib
csl: ../cslFiles/amj.csl
abstract: Studies of collaboration show...
---
```

R Markdown body

The body of your R Markdown file contains your text and code:

- Text needs to follow the Markdown syntax
- Code goes in code chunks

```
# A title in Markdown

_Italics_ __bold__

[A link](https://www.example.com/)
```

A code chunk:

```{r eval=TRUE}

n <- 10 rnorm(n)

```

Version control

a very short introduction

Git

Git is a version control software started by Linus Torvalds (of Linux's fame).

It allows you to track changes in your file on a specific project.

Useful when writing code:

- when things break, you can rewind to figure it out
- it makes collaboration easier
- it forces you to describe what you do (through commit messages)
- one day it will save you a lot of tears and pain

Basics of Git

Git is integrated with RStudio:

- no need to use the shell
- but knowing how to use the shell is useful when things break
- you need to know a few functions:
    - git init
    - git add
    - git commit
    - git checkout

Learn more: RStudio git integration, an intro to git and RStudio

Bitbucket and Github

- Two platforms to host your code
- Gitbub: great for public code
- Bitbucket: great for non-public code (on an academic licence)
- Both use git and:
  - store your code remotely
  - allow you to collaborate with yourself and others

Git: a short demo

We already created an R project with a git repository. Let's now:

- Create a remote with github or bitbucket
  - You might encounter an error when trying to push back to the repository:
    - click on the "Tools" menu and select "Shell", command: `git push -u origin master`
    - enter your github username and password when prompted
- Once this is done, we can play around with a script and an R Markdown document.

Coding style

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability.

John F Woods