Reproducible Research in R

Dries Debeer & Benjamin Becker

29. and 30. September 2022

FDZ Autumn Academy

Introduction

Introduction

Who are we?

Dries Debeer

Statistical Consultant at Ghent University (FPPW)

scDIFtest, permimp, eatATA, mstDIF

dries.debeer@ugent.be

Benjamin Becker

Researcher at IQB (Verbund Forschungsdaten)

eatGADS, eatDB, eatATA, pisaRT

b.becker@iqb.hu-berlin.de

Introduction

Who are you?

1. Occupation, employer?
2. Previous knowledge and experience
    - with reproducible research
    - with R?
    - with other statistical software?
    - with other programming languages?
3. Specific interest/motivation for this workshop?

Motivation

Why care about reproducible research?

Agenda

- Conceptual things
- Writing Reproducible R Code
- RMarkdown
- Version Control/git

Reproducible Research

RStudio setup

RStudio setup

1. Save the course content to a directory on your machine
2. Open RStudio
3. Choose `File < New Project ...`
4. Choose `Existing Directory`
5. Browse to the directory on your machine where you saved the course content and select the "R-programming" folder as the `Project working directory`
6. Click `Open in new session`
7. Click `Create Project`

RStudio setup - optional

1. Choose `Tools < Global options`
2. Under `General`
   - DON'T `Restore .RData into workspace at startup`
   - NEVER `Save workspace to .Rdata on exit:`
   - Save the code instead!
   - Use `saveRDS()` and `readRDS()` for objects that require a long time to computate
3. Further personalize RStudio

Writing Reproducible R Scripts

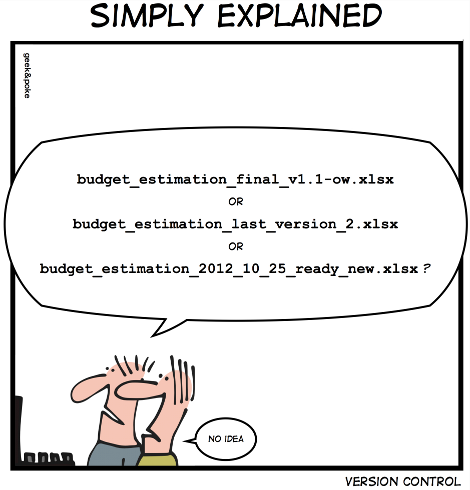
RMarkdown

Version Control via Git and Github

Version Controlling

- Motivation
- Setup
- Work Flows
- Recommendations
- Resources

Motivation


SIMPLY EXPLAINED
geek&poke
budget_estimation_final_v1.1-ow.xlsx
OR
budget_estimation_last_version_2.xlsx
OR
budget_estimation_2012_10_25_ready_new.xlsx ?
NO IDEA
VERSION CONTROL

Motivation 1

Single Author Projects

- Implementation of long term change history
    - What has been changed?
    - When was it changed?
- No ridiculous file names
- No archive sub folder
- Accessibility for others ('Open Science')
- Additional safety net
- ...

Motivation 2

Collaborations

- Who has changed what when exactly?
- Clear, current project state
- No annoying mail attachments or file-sharing platforms
- Parallel work easily possible
- Possibility of hierarchical responsibilities
- ...

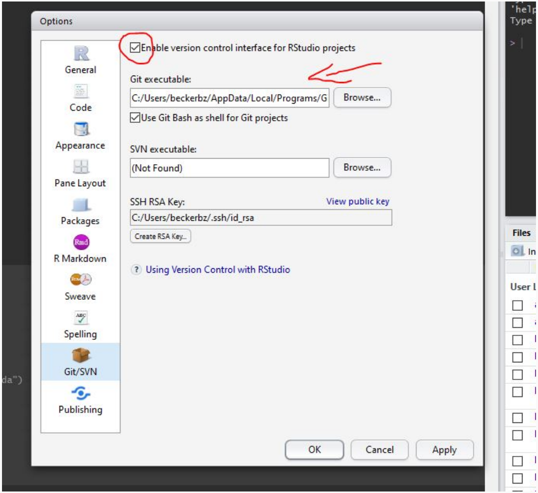
Aber...

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Prerequisits

- Git-Installation
- RStudio-Installation
  → Alternatives: Shell, Gitkraken, SmartGit, ...
- Github account
  → Alternatives: Bitbucket, Gitlab, ...
- Connect everything

RStudio



☑ Enable version control interface for RStudio projects

Git executable:

C:/Users/beckerbz/AppData/Local/Programs/G

☑ Use Git Bash as shell for Git projects

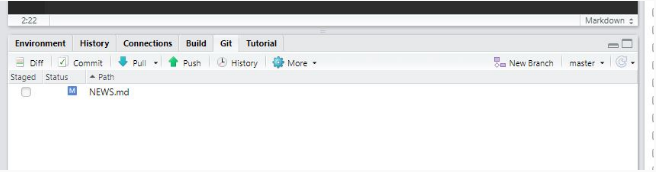
SVN executable:

(Not Found)

SSH RSA Key:

C:/Users/beckerbz/.ssh/id_rsa

RStudio



Staged	Status	Path
☐	M	NEWS.md

Work flow 1

Creating a repository

- Create an **online repository** (e.g. on Github)
  - Use an R specific `.gitignore`
  - Initialize with a short readme (`.md`)
- Clone the repository to your local machine via RStudio as a new project
- An R-Project is added automatically to the existing repository

Excursion: gitignore

- Plain text file
- Which files should not be tracked by git?
  → These then only exist locally in their current version!
- Options
  - Single files
  - Folders
  - Specific data types
  - Combinations of the above
- Use cases
  - Large files (Data, images, ...)
  - Auxiliary files (e.g. created during latex compilation)

Work flow 2

Working with an existing repository

- Before working: Synch your local repo (**Pull** or **clone**)
- Perform changes in your local repository
  → Create/modify/delete files
- **Stage** your changes
- **Commit** your changes (aka new version)
- **Push** your commit(s) (online repository is updated)

Excursion: mergen

Conflicts between different updated versions

- Common when working collaboratively
- Discrepancies between your own different local repos → Git communicates these and indicates conflicts
- Select the desired changes
- Stage selection, commit and push

Excursion: Branches

Multiple parallel versions of a project within one repository

- Common e.g. in areas like software development
- e.g. one stable and one development branch
- Only certain modifications should be made in the stable branch
- **Note**: RStudio GUI has limited support for this

Your impressions?

Recommendations

- Keep it simple!
    - If not necessary, no branches/forks/pull requests
- Have meaningful commits
- Keep it lean (no big files)
- Avoid using the Github homepage working within the repository

Resources

Git + RStudio Resources

- Small Intro (https://r-bio.github.io/intro-git-rstudio/)
- Happy Git with R (https://happygitwithr.com/)
- R Packages and Git (https://r-pkgs.org/git.html)

General Git Resources

- Git Book (http://git-scm.com/book/en/v2)

Good programming practices

“Write code for humans, not for machines!”

Code Style

Invest time in writing readable R-code.

- It will make collaborations easier
- It will make debugging easier
- It will make your analyses more reproducible

There is a complete *tidyverse* style-guide `https://style.tidyverse.org/`.

Go easy on your eyes

- with spaces before and after: `- + / * = <- < == >`
- always use `<-` for assignments
- only use `=` in function calls
- use indentation (largely automatic in RStudio)
- `CamelCaseNames` vs `snake_case_names`
- be consistent!
- wrap long lines at column 70-80 (Rstudio)

White space

```
new_var=(var1*var2/2)-5/(var3+var4)

# versus

new_var <- (var1 * var2 / 2) - 5 / (var3 + var4)
```

Indentation

```r
for(name in names){formula=as.formula(paste0("y~.-",name))
fit<-lm(formula,data=my_data)
coefs[["name"]]=coef(fit)
print(name)
print(summary(fit))}

# versus

for(name in names){
  formula <- as.formula(paste0("y~.-", name))
  fit <- lm(formula, data = my_data)
  coefs[["name"]] <- coef(fit)
  print(name)
  print(summary(fit))
}
```

Wrap long lines

```r
final_results <- data.frame(first_variable =
sqrt(results$mean_squared_error), second_variable =
paste0(results$condition, results$class, sep = ":"),
third_variable = results$bias)

# versus

final_results <- data.frame(
  first_variable = sqrt(results$mean_squared_error),
  second_variable = paste0(results$condition,
                           results$class, sep = ":"),
  third_variable = results$bias)
```

Go easy on your mind

- use meaningful names: “self-explainable”
- always write the formal arguments in function calls (except the first)
- benefit from autocompletion (`<tab>`) => embrace longer names
- use `TRUE` and `FALSE` not `T` and `F`
- comment, comment, comment
    - NOT what (should be clear from the code)
    - but why
    - explain the reasoning, not the code

Use meaningful names

```r
V <- myFun(m1_B)

# versus

RMSE_age_gender <- get_RMSE(lm_age_gender)
```

Programming advice

Use `verbs` for functions and `nouns` for other objects.

Write formal arguments

Benefit from auto completion using tab

```
m1_B <- lm(outcome ~ age*gender,
           exp1, condition_1, freq)

# versus

lm_age_gender <- lm(outcome ~ age * gender,
                    data = exp1,
                    subset = condition_1,
                    weigths = freq)
```

Comment, comment, comment

```r
## Start every Rscript with a comment that explains
##  what the code in the script does, why it does
##  this, and to which project it belongs.
##  Your future self will be very thankful!
##
## Mention which packages you are using in this Rscript.

## Use sections to separate chunks ----------------------

## Maybe even subsections ===============================

## Recode variables so that missings are coded as "NA"
dat[dat %in% c(99, 999)] <- NA  # missings coded 99 or 999
```

Keep your code slim

Try to limit your *package-dependencies*.

Only load `library()` the packages that you absolutely need. If you are only using `dplyr`, it does not make sense to load the complete `tidyverse`.

Controversial: when possible, use the `::` operator (and consider not loading the package). `<package>::<function>`

- explicit dependencies
- less name conflicts

Never Attach

Forget about `attach()`!

Don't use it, unless you completely understand what happens (see `?attach`).

Use `with(data.frame, expression)` instead.

```r
# using with()
n <- 2e+4
data <- data.frame(x = runif(n),
                   y = runif(n),
                   z = seq_len(n))
result <- with(data, exp(x) / log(z) + 5 * sqrt(y))
```

Testing R code

Writing code is error prone. Incorporate tests and checks in your workflow.

- minimal examples
- write tests and checks
- helpful packages: `testthat`, `RUnit`, `testit`, ...

Speed

Computing speed can become an issue. Avoid common pitfalls:

- don't grow, but replace
- vectorize where possible
- check the computing speed

?system.time, microbenchmark or profiling tools

```
n <- 2e+4
data <- data.frame(x = runif(n),
                   y = runif(n),
                   z = seq_len(n))
```

Speed

Don't grow!

```
system.time({
  new_data <- NULL

  for(row_nr in seq_len(NROW(data))){
    new_data <- cbind(
      data[row_nr,],
      result = exp(data$x[row_nr]) /
        log(data$z[row_nr]) +
        5 * sqrt(data$y[row_nr]))
  }
})


>    user  system elapsed
>    2.09    0.00    2.11
```

Speed

Replace!

```
system.time({
  n_rows <- dim(data)[1]
  data$result <- rep(NA, n_rows)

  for(row_nr in seq_len(n_rows)){
    data$result[row_nr] <- exp(data$x[row_nr]) /
      log(data$z[row_nr]) +
      5 * sqrt(data$y[row_nr])
  }
})


>    user  system elapsed
>    0.31    0.02    0.33
```

Speed

Vectorize!

```
system.time({
  data$result <- exp(data$x) / log(data$z) +
    5 * sqrt(data$y)
})


>    user  system elapsed
>       0       0       0
```

Speed

Compare the speed of different implementations using:

`microbenchmark::microbenchmark`

```r
get_mean1 <- function(x){
  weight <- 1/length(x)
  out <- 0
  for(i in seq_along(x)){
    out <- out + x[i] * weight
  }
  return(out)
}

get_mean2 <- function(x){
  sum(x)/length(x)
}
```

Speed

Compare the speed of different implementations using:

`microbenchmark::microbenchmark`

```
x <- rnorm(500)
microbenchmark::microbenchmark(
  mean(x), get_mean1(x), get_mean2(x))

> Unit: nanoseconds
>          expr   min    lq     mean median    uq     max neval
>       mean(x)  2100  2201  3540.98   2301  2501   90701   100
>  get_mean1(x) 13501 13901 53314.06  14101 15000 3863201   100
>  get_mean2(x)   600   701 13529.91    801   851 1253501   100
```

Speed

Programming advice

Don’t worry about speed before it becomes an issue.

Wrap Up

General Advice

- Investing time in learning R pays off
- It's a steady learning curve
- Learn from masters
- Rewrite important code - the first attempt is usually not the best approach

General R Advice

- Document well
- Use a consistent style
- Write functions
- Split long functions in smaller ones
- Write wrappers
- Use Iteration (don't copy paste)
- Use matrix operations and vectorized functions instead of loops
- Use git

Literature Recommendations

R Resources

- Avanced R Ed. 1 (http://adv-r.had.co.nz/)
- Avanced R Ed. 2 (https://adv-r.hadley.nz/)
- R Inferno (https://www.burns-stat.com/pages/Tutor/R_inferno.pdf)
- R Packages (https://r-pkgs.org/)
- Clean Code (https://mooc.aptikom.or.id/pluginfile.php/1174/mod_resource/content/1/Clean%20Code_%20A%20Handbook%20of%20Agile%20Software%20C%20-%20Robert%20C.%20Martin.pdf)

Thank you for your attention!

Thank you for your attention!

Questions? Remarks?