Lecture 2: More on Git and GitHub

Irina Gaynanova

Before we start

- Open **GitIntro** project in Rstudio we have created last time with readme.md file and open corresponding GitHub repository in browser
- If you don't have them
  - Create a new GitHub repository with arbitrary name, click **initialize with readme** and then clone the created repository to your local machine via git clone. Create a new R studio project associated with directory.

```
git clone URL
```

Git workflow - recall

- Git workflow

raw changes -> staged changes (tracking via add) -> committed changes (via commit)

- Each commit is a snapshot of a project version. All differences are from the latest commit.
- One project -> one repository -> one .Rproj -> one .gitignore

Do not try to nest other version controlled repositories within the existing ones

Why commit messages matter?

Figure 1: Commit message history

Good commit practices

- make frequent commits
- do not commit half-done work, i.e. “Started to fix bug X”
- test your code before you commit
- avoid committing large chunks of code (break it into smaller pieces)
- commit logical changes together (2 separate bugs - 2 separate commits)
- **Make good commit messages**

Examples of bad commit messages

- “fixed a bug”
- “changed a few things”
- “more code adjustments”
- “update”

Good commit messages

See **"How to Write a Git Commit Message"**

Rule of thumb: should finish the sentence

If applied, this commit will ***your subject line here***

Examples:

- If applied, this commit will **update getting started documentation**
- If applied, this commit will **fix bug associated with data input**
- If applied, this commit will **add function generateY**

At a minimum, you should be able to guess what happened in that commit.

Practice creating version control folders locally

- from Terminal/Git bash/Shell

```
git init FOLDER_NAME
```

or directly from the folder

```
git init
```

- From Rstudio

New project -> Check mark for git version control.

Github: Collaborative Workflow

- First get all external updates via **pull** or on console

```
git pull origin master
```

- Make your local changes and commit them
- Upload the changes to GitHub with **push** or

```
git push origin master
```

Github: Collaborative Workflow - commiting from Github

- **Push** any changes you have locally to Github
- Find you readme.md file in Github repository, make changes and **commit them via GitHub**
- Go to your local repository, and **pull** the changes.

Conflict resolution

- Go to the **Github** repository you just synced, update readme.md directly online from **GitHub**, and commit the changes online
- Go to your **local repository**, make a different update to readme.md and commit the changes
- Try to **push** your changes to Github -> what happened?

Conflict resolution

- To resolve the conflict, you have to **pull** the changes first, then **merge** them, and then do **push**
- First do the **pull**. What happened?
- Decide on the change you want to keep, commit the change, and do **push** now

Github flows: From Github to Local and back (HW1)

- Clone your HW1 directory locally (if not yet)
- Make sure you **open the correct .Rproj** - the one associated with current folder
- Make changes to last name/first name, stage them (add) and commit
- Update the external directory on Github via **push**.

NEXT:

- Code style and commenting in R
- Code timing via **microbenchmark** package

R Style guide

References: H.Wickham's Advanced R and Google's R style guide

- use meaningful names

```
# example of linear regression parameters

# GOOD
beta0 <- 10
intercept <- 10

# BAD
x <- 10
y <- 10
adjsgf <- 10
```

R Style guide

- avoid using names of existing functions and variables

```
# BAD
# T <- FALSE
# c <- 10
# mean <- function(x) sum(x)
```

- can lead to undersirable behavior

```
T <- FALSE
x <- 5
y <- 10 / 2
xy_equal <- x == y
xy_equal == T
```

```
[1] FALSE
```

R Style guide

- use spaces around infix operators (+, -, <-, =, etc.)

```r
# GOOD
#####################################
# Example 1
x <- 3
sigma <- 2
density_x <- exp(-(x - 5)^2 / (2 * sigma^2)) / (2 * pi * s


# BAD
#####################################
# Example 1
density_x<-exp(-(x-5)^2/(2*sigma^2))/(2*pi*sigma)
```

R Style guide

- use spaces around infix operators (+, -, <-, =, etc.)

```
# GOOD
########################################
# Example 2
vec <- c(1, 4, NA)
sum_vec <- sum(vec, na.rm = TRUE)

# BAD
########################################
# Example 2
vec<-c(1,4,NA)
sum_vec<-sum(vec,na.rm=TRUE)
```

R Style guide

- comma placement for matrix elements

```r
mat <- matrix(rnorm(30), 10, 3)

mat[1, ] # GOOD
```

```
[1] -1.8085395 -0.7113929 -0.2784552
```

```r
mat[1,] # BAD
```

```
[1] -1.8085395 -0.7113929 -0.2784552
```

```r
mat[1 ,] # BAD
```

```
[1] -1.8085395 -0.7113929 -0.2784552
```

R Style guide

- **Good news**: R package formatR can be used for automatic code formatting according to style rules

R Style guide

- use meaningul comments generously throughout the code

```
# BAD - What is this doing?
a = 100
b = 50
c = rnorm(a)
b = rnorm(b)
d = t.test(c,b)
```

R Style guide

- use meaningul comments generously throughout the code

```
# GOOD
# Specify sample sizes for two groups
n1 <- 100
n2 <- 50
# Generate data from each group
sample1 <- rnorm(n1)
sample2 <- rnorm(n2)
# Perform two-sided t-test of difference in means
t.test.out <- t.test(sample1, sample2)
```

Comparing R codes for speed

- R package **microbenchmark** is well-suited for comparing small chunks of code
- Your code can often be significantly improved

```
library(microbenchmark)
x <- 120
microbenchmark(
  sqrt(x),
  x^(0.5)
)
```

```
Unit: nanoseconds
    expr min    lq   mean median    uq   max neval
 sqrt(x) 101 154.5 206.93  165.5 179.0  4036   100
 x^(0.5) 207 306.0 457.16  331.5 366.5 11805   100
```

Comparing R codes for speed

```r
library(microbenchmark)
p <- 1000
x <- runif(p, min = 100, max = 120)
microbenchmark(
  sqrt(x),
  x^(0.5)
)
```

```
Unit: microseconds
     expr    min      lq     mean  median      uq    max nev
  sqrt(x)  2.417  2.5195  2.75877  2.5925  2.7275 11.930    1
 x^(0.5) 23.848 23.9755 24.10862 24.0290 24.1130 28.627    1
```

Comparing R codes for speed

```
p <- 100000
x <- runif(p, min = 100, max = 120)
microbenchmark(
  sqrt(x),
  x^(0.5)
)
```

```
Unit: microseconds
    expr      min       lq      mean    median       uq  9
 sqrt(x)  233.564  593.768  738.9439  632.2225  775.963  96
 x^(0.5) 2358.401 2788.042 3385.5486 3091.9505 3666.620 103
```

Comparing R codes for speed

- Take advantage of **crossprod** and **tcrossprod** functions. Suppose we want to calculate $x^T Ax$

```
p <- 3000
x <- rnorm(p)
A <- matrix(rnorm(p^2), p, p)
microbenchmark(
    t(x) %*% A %*% x,
    crossprod(x, A %*% x)
)
```

```
Unit: milliseconds
                  expr      min       lq     mean   median
      t(x) %*% A %*% x 14.91187 16.32867 17.42696 17.21991
 crossprod(x, A %*% x) 13.22144 14.38213 15.75310 15.44475
      max neval
 21.44628   100
 23.09832   100
```

Comparing R codes for speed

- **Hack** for calculating $\|x\|_2^2$ for large p

```
p <- 100000
x <- rnorm(p)
as.numeric(crossprod(x))
```

```
[1] 100031.9
```

```
sum(x^2)
```

```
[1] 100031.9
```

Comparing R codes for speed

- **Hack** for calculating $\|x\|_2^2$ for large p

```
microbenchmark(
as.numeric(crossprod(x)),
sum(x^2)
)
```

```
Unit: microseconds
                     expr     min       lq     mean   media
 as.numeric(crossprod(x)) 135.899 143.0070 199.5925 167.333
                 sum(x^2) 207.911 571.2125 734.1715 718.585
      max neval
  833.834   100
 3531.887   100
```

Comparing R codes for speed

- Take advantage of **colSums**, **colMeans** and corresponding row functions

```
n <- 100
p <- 3000
A <- matrix(rnorm(n * p), n, p)
microbenchmark(
    colMeans(A),
    apply(A, 2, mean)
)
```

```
Unit: microseconds
              expr       min         lq       mean     media
       colMeans(A)   188.876   240.4355   310.4489    292.48
 apply(A, 2, mean) 15763.904 21763.5185 27026.5516 26272.33
      max neval
  1045.44   100
 49609.58   100
```

Comparing R codes for speed

- Use **solve** wisely

```
n <- 10000
p <- 300
A <- matrix(rnorm(n * p), n, p)
S <- cov(A)
x <- rnorm(p)
microbenchmark(
    solve(S) %*% x,
    solve(S, x)
)
```

```
Unit: milliseconds
           expr       min        lq     mean    median    36.
 solve(S) %*% x 27.776194 30.881571 36.09988 32.941939 36.1
    solve(S, x)  7.646888  8.574461 10.52027  9.524531 11.7
 neval
   100
   100
```

Comparing R codes for speed

- fast SVD solvers, especially when only few vectors are desired

```
library(irlba)
n <- 200
p <- 5000
X <- matrix(rnorm(n*p), n, p)

microbenchmark(
    svd(X, nu = 1, nv = 1),
    irlba(X, nu = 1, nv = 1),
    times = 10
)
```

```
Unit: milliseconds
                     expr       min       lq     mean   mec
   svd(X, nu = 1, nv = 1) 702.71543 722.2205 827.1534 780.2
 irlba(X, nu = 1, nv = 1)  97.37796 103.9433 113.1119 112.4
      max neval
```

Comparing R codes for speed

- fast SVD solvers, especially when only few vectors are desired

```
library(irlba)
library(RSpectra) # function svds, notice extra argument be
n <- 200
p <- 5000
X <- matrix(rnorm(n*p), n, p)

microbenchmark(
    svds(X, 1, nu = 1, nv = 1),
    irlba(X, nu = 1, nv = 1),
    times = 50
)
```

```
Unit: milliseconds
                       expr       min        lq      mean  
 svds(X, 1, nu = 1, nv = 1) 105.95371 115.91902 121.6011 1
   irlba(X, nu = 1, nv = 1)  77.55656  96.00879 106.3822 1
```