Workflow and organization for statistical projects

Patrick Breheny

Associate Professor
Department of Biostatistics

September 30, 2021

Introduction

- A simple analysis that can be done in a dozen lines of code doesn't require any special organization
- As projects grow in complexity, however, they quickly become unmanageable unless you give careful thought to organizing them
- Today, I'll present my thoughts on this topic, which have evolved gradually over the years
- Some of these recommendations are personal preferences, although many of them represent best practices from software engineering

Overview

- Workflow and organization is an important topic and the department should probably teach it in some course, but it's not clear which course it would fit into
- The attitude seems to be that you pick this stuff up while working as an RA
  - . . . but maybe you pick up bad habits
- The first half of this talk, I'll give a broad overview of important general principles
- In the second half, I'll present an example and go over specific R/Rstudio tools

A helpful exercise

- Let's start by considering our goals: What do we want our project to look like?
- A helpful exercise is to pretend you are seeing this project for the first time. . . would it be obvious to you where everything is, where to find the code that produced a certain analysis, how to re-run that analysis?
- This is a helpful exercise to go through not only because this exact scenario is likely to happen one day, when you start to work with another RA or you transition to a different project or something like that. . .
- . . . but also because that person may very well be you; you got busy with other things and now you're coming back to the project two months later and need to remember where everything is

Organization basics

- Goal #1: we want the project to be clearly organized
- A complex project will have many files – there is no getting around this fact
- However, by organizing files clearly into subdirectories, you won't be overwhelmed by all the files at once; you can just appreciate the top-level view instead
- A basic principle of directory organization is that different types of files should be in different directories
- For example, don't put data in the same directory as code, don't put code in the same directory as figures

An example structure

- There are situations where it may make sense to violate this principle, but in general the directory structure for any statistical project should look something like:
  - `data`
  - `R` (for code that defines functions)
  - `scripts` (code that actually does things)
  - `explore` (code that may or may not run/be useful)
  - `figures`
  - `tables`
  - `README.md` (statement of project goals, definitions of scientific terms related to the project, an overview of the analysis workflow, naming and coding conventions, etc.)

Clutter

- Another big part of organization is removing as much clutter as possible
- Obsolete code is a major obstacle to the passing the "pretend you're seeing this project for the first time" exercise
- The ideal solution to this problem is version control using `git`/GitHub (which allows you to delete the file, delete lines from a file, but if you want to go back to some point in the past and look at it, you can), although this is the subject of a whole separate talk
- If you don't know how to use `git`, you can put that code in an `obsolete` folder; this is definitely a worse solution, but still better than keeping directories half-filled with broken code and `.r` files half-filled with commented-out irrelevant code

Reproducibility

- A second major goal is reproducibility, which is important for two reasons:
  - It's critical to the scientific process
  - You will almost certainly need to re-run your analysis many times, so plan accordingly
- In thinking about reproducibility, it's important to separate files into three types:
  - Data
  - Code
  - Output

Data

- You can't reproduce data (you're a statistician, someone else collected the data)
- The number one rule with data files in terms of reproducibility is: don't mess with them
- These files should be copied into the project as-is and never modified
- It is often a good idea to record where the data came from (its "provenance"); for example, I often title data files something like '2021-09-29.xls' to note that it was e-mailed to me on that date

Code

- With code, the goal is: you should be able to start a fresh instance of R, run the code in that script (and only that code), and have it produce the same result every time
- It is important, then, that your code depends as little as possible on external factors – it does something different if some other code has been first, etc.
- We'll talk more about this later, but let's establish one thing first: under no circumstances should you ever save your R workspace or restore a saved workspace
- In doing so, you're committing yourself to a different goal, "it should be impossible to look at my code and guess what it will do. . . I want utter unpredictability"

Output

- By "Output", I mean anything that your code produces
- The key test of reproducibility is: You should be able to delete all of your output and think, "that's fine, I'll just click the 'reproduce my analysis button' and get it all back"
- If your analysis doesn't take that long to run, you should literally do this
- Unfortunately, the more time-consuming an analysis is, the harder it is to verify reproducibility

Portable

- Another important goal is *portability*, meaning that different people can run your code on different machines, and it still works
- For example, absolute paths should be avoided whenever possible
- With apologies to Jenny Bryan for stealing her joke. . .
- . . . if the first line of your R code is

  ```
  setwd("C:\Users\pbreheny\path\that\only\I\have")
  ```

  I will come into your office and SET YOUR COMPUTER ON FIRE

Readable

- You also want the code in your project to be *readable*: someone who has never seen the project before should be able to look at your code and see what it's doing without having to run it
- This comes from three things:
  - Making an effort to write better, cleaner code
  - Using functions (more later on this)
  - Documentation (more later on this too)

Reliable

- Finally, you want the code in your project to be *reliable*, in the sense that it actually runs without errors
- This probably seems obvious, but at the same time: how do you actually know that all of your code runs and doesn't have errors?
- The more complicated a project is, the more interconnected they tend to be, and it is very easy to write code that fixes one problem, only to find out later that it broke the code in a separate part of the project

To recap, we want our project to be...

- Organized
- Reproducible
- Portable
- Readable
- Reliable

Introduction

- The preceding remarks are all rather general; let's now get into specifics
- I'm going to work through an example of taking a project and illustrating specific R/RStudio tools for making the project organized, portable, reliable, and reproducible
- Specifically, we'll go through three stages:
  - Turning scripts into functions
  - Documenting those functions
  - Testing those functions

Projects

- To begin with, RStudio provides specific setup files called "projects", and you should use them, especially if you're collaborating with other people
- They don't do anything earth-shattering, but they improve portability by (a) opening Rstudio in the directory containing the project file and (b) by ensuring that all the RStudio options are the same for anyone who opens the project
- From there, just stay in the root directory, never run `setwd()`; this leads to a little bit of extra typing, but makes the code much easier to read and follow, as well as more reliable

"Pseudopackages"

- There are a number of advantages to making your project follow the rough structure of an R package
- The reason for this is that R/RStudio have a number of tools specifically designed for R packages, and you can use these tools even if you have no intention of ever "packaging" the project
- The only thing a project needs in order for R/Rstudio to recognize it as a package is a DESCRIPTION file, the contents of which are essentially boilerplate
- Typically, the only line that needs editing is `Depends:`, where you list packages that your project needs to load (attach)

Functions

- I don't want to belabor this point, but by far the most important thing you can do to keep a project organized is to replace chunks of copy/pasted code with functions
- This is one of the most important principles of software development, known as DRY: don't repeat yourself
- There are many advantages to functions over copy/pasting
  - Less cluttered, easier to read
  - Easier to experiment/explore: run code with different arguments with minimal work
  - Easy to update: only change the function in one place, not dozens of places it's been pasted
  - Less error-prone: Copy/paste errors are probably the most common source of coding errors in biostat graduate students
  - Easier to document and test

Load all

- In a (pseudo)package, functions are put in a folder called `R`
- Then at any point, you can load (source) all the functions with Ctrl+Shift+L
- You can also type `devtools::load_all()`, but keep in mind that you'll have to do this a lot – every time you change the code in a function, you'll need to re-load the function definitions, so you want a convenient shortcut

Document

- After you've written these functions, it's extremely useful to document them
- You can write a lot of documentation if you want or if the function is doing something very complicated, but at a minimum, you should describe the input and output
- You could just write text at the top of the function, but there are many advantages to using the R package `roxygen2` for documentation
  - It's basically just as easy as writing text at the top of the function, but...
  - ...you can also access the documentation through the R help system with `?my_function`
- To rebuild the documentation, Ctrl+Shift+D

Examples

- Functions should also have examples – this serves as both important documentation (showing how to use them) and an important test (do they run without errors?)
- Include these examples at the end of your `roxygen2` documentation like so:

```
#' @examples
#' DT <- month_merge('may', 'june')
#' print(DT)
```

- To run all the examples (and verify that all your functions execute without errors), Ctrl+Shift+X (this might still be in the beta stage)

Testing

- The fact that examples run is a good baseline test for the reliability of your code
- However, it's often a good idea to write explicit tests using a package such as `testthat` or `tinytest`
- This allows you to check not just that a function runs, but that it runs as intended and gives the correct answer
- In particular, you can test with edge cases, verify that errors are produced when they should be, etc.
- It is extremely useful to run all tests periodically (Ctrl+Shift+T), in order to verify that by adding some new feature and modifying a function, that you haven't broken it in such a way that other scripts depending on that function will no longer run