Benjamin-Chung Lab Manual

Updated: 2021-10-20

Contents

Chapter 1

Welcome to the Benjamin-Chung Lab!

1.1 About the lab

Welcome to the lab of Dr. Jade Benjamin-Chung, Assistant Professor in Epidemiology & Population Health at Stanford University. Our mission is to improve population health by creating high quality evidence about what health interventions work in whom and where, when, and how to implement them. Most of our research is focused on infectious diseases, including malaria, diarrhea, soil-transmitted helminths, and influenza. Our focus is on improving the health of vulnerable populations from low-resource settings, both domestically and internationally. We use a variety of epidemiologic, computational, and statistical methods, including causal inference and machine learning methods, in pursuit of our mission. To learn more about the lab, visit jadebc.net.

1.2 About this lab manual

This lab manual covers our communication strategy, code of conduct, and best practices for reproducibility of computational workflows. It is a living document that is updated regularly.

This manual was created with input from a large number of team members and with inspiration from other scientists' lab manuals. Contributors include Jade Benjamin-Chung, Kunal Mishra, Stephanie Djajadi, Nolan Pokpongkiat, and Anna Nguyen.

Feel free to draw from this manual (and please cite it if you do!).

This work is licensed under a Creative Commons Attribution-NonCommercial

4.0 International License.

Chapter 2

Culture and conduct

by Jade Benjamin-Chung

2.1 Lab culture

We are committed to a lab culture that is collaborative, supportive, inclusive, open, and free from discrimination and harassment.

We encourage students / staff of all experience levels to respectfully share their honest opinions and ideas on any topic. Our group has thrived upon such respectful honest input from team members over the years, and this document is a product of years of student and staff input (and even debate) that has gradually improved our productivity and overall quality of our work.

2.2 Diversity, equity, and inclusion

The Benjamin-Chung lab recognizes the importance of and is committed to cultivating a culture of diversity, equity, and inclusion. This means being a safe, supportive, and anti-racist environment in which students from diverse backgrounds are equally and inclusively supported in their education and training. Diversity takes many forms, and includes, but is not limited to, differences in race, ethnicity, gender, sexuality, socioeconomic status, religion, disability, and political affiliation.

2.3 Protecting human subjects

All lab members must complete CITI Human Subjects Biomedical Group 1 training and share their certificate with Jade. She will add team members to

relevant Institutional Review Board protocols prior to their start date to ensure they have permission to work with identifiable datasets.

One of the most relevant aspects of protecting human subjects in our work is maintaining confidentiality. For students supporting our data science efforts, in practice this means:

- Be sure to understand and comply with project-specific policies about where data can be saved, particularly if the data include personal identifiers.
- Do not share data with anyone without permission, including to other members of the group, who might not be on the same IRB protocol as you (check with Jade first).

Remember, data that looks like it does not contain identifiers to you might still be classified as data that requires special protection by our IRB or under HIPAA, so always proceed with caution and ask for help if you have any concerns about how to maintain study participant confidentiality.

2.4 Authorship

We adhere to the ICMJE Definition of authorship and are happy for team members who meet the definition of authorship to be included as co-authors on scientific manuscripts.

Chapter 3

Communication and coordination

by Jade Benjamin-Chung

One benefit of the academic environment is its schedule flexibility. This means that lab members may choose to work in the early morning, evening, or weekends. That said, we do not expect lab members to respond outside of business hours (unless there are special circumstances).

3.1 Slack

- Use Slack for scheduling, coding related questions, quick check ins, etc. If your Slack message exceeds 200 words, it might be time to use email.
- Use channels instead of direct messages unless you need to discuss something private.
- Please make an effort to respond to messages that message you (e.g., `@jade`) as quickly as possible and always within 24 hours.
- If you are unusually busy (e.g., taking MCAT/GRE, taking many exams) or on vacation please alert the team in advance so we can expect you not to respond at all / as quickly as usual and also set your status in Slack (e.g., it could say "On vacation") so we know not to expect to see you online.
- Please thread messages in Slack as much as possible.

3.2 Email

- Use email for longer messages (>200 words) or messages that merit preservation.

- Generally, strive to respond within 24 hours hours. As noted above, if you are unusually busy or on vacation please alert the team in advance so we can expect you not to respond at all / as quickly as usual.

3.3 Trello

- Jade will add new cards within our shared Trello board that outline your tasks.
- The higher a card is within your list, the higher priority it is.
- Generally, strive to complete the tasks in your card by the date listed.
- Use checklists to break down a task into smaller chunks. Sometimes Jade will write this for you, but you can also add this yourself.
- Jade will move your card to the "Completed" list when it is done.

3.4 Google Drives

- We mostly use Google Drive to create shared documents with longer descriptions of tasks. These documents are linked to in Trello. Jade often shares these with the whole team since tasks are overlapping, and even if a task is assigned to one person, others may have valuable insights.

3.5 Meetings

- Our meetings start on the hour.
- If you are going to be late, please send a message in our Slack channel.
- If you are regularly not able to come on the hour, notify the team and we might choose the modify the agenda order or the start time.

Chapter 4

Reproducibility

by Jade Benjamin-Chung

Our lab adopts the following practices to maximize the reproducibility of our work.

1. Design studies with appropriate methodology and adherence to best practices in epidemiology and biostatistics
2. Register study protocols
3. Write and register pre-analysis plans
4. Create reproducible workflows
5. Process and analyze data with internal replication and masking
6. Use reporting checklists with manuscripts
7. Publish preprints
8. Publish data (when possible) and replication scripts

4.1 What is the reproducibility crisis?

In the past decade, an increasing number of studies have found that published study findings could not be reproduced. Researchers found that it was not possible to reproduce estimates from published studies: 1) with the same data and same or similar code and 2) with newly collected data using the same (or similar) study design. These "failures" of reproducibility were frequent enough and broad enough in scope, occurring across a range of disciplines (epidemiology, psychology, economics, and others) to be deeply troubling. Program and policy decisions based on erroneous research findings could lead to wasted resources, and at worst, could harm intended beneficiaries. This crisis has motivated new practices in reproducibility, transparency, and openness. Our lab is committed to adopting these best practices, and much of the remainder of the lab manual focuses on how to do so.

Recommended readings on the "reproducibility crisis":

- Nuzzo R. How scientists fool themselves – and how they can stop. 2015. https://www.nature.com/articles/526182a
- Stoddart C. Is there a reproducibility crisis in science? 2016. https://www.nature.com/articles/d41586-019-00067-3
- Munafo MR, et al. A manifesto for reproducible science. *Nature Human Behavior* 2017 http://dx.doi.org/10.1038/s41562-016-0021

4.2 Study design

Appropriate study design is beyond the scope of this lab manual and is something trainees develop through their coursework and mentoring.

4.3 Register study protocols

We register all randomized trials on clinicaltrials.gov, and in some cases register observational studies as well.

4.4 Write and register pre-analysis plans

We write pre-analysis plans for most original research projects that are not exploratory in nature, although in some cases, we write pre-analysis plans for exploratory studies as well. The format and content of pre-analysis plans can vary from project to project. Here is an example of one: https://osf.io/tgbxr/. Generally, these include:

1. Brief background on the study (a condensed version of the introduction section of the paper)
2. Hypotheses / objectives
3. Study design
4. Description of data
5. Definition of outcomes
6. Definition of interventions / exposures
7. Definition of covariates
8. Statistical power calculation
9. Statistical analysis:

- Type of model
- Covariate selection / screening
- Standard error estimation method
- Missing data analysis
- Assessment of effect modification / subgroup analyses
- Sensitivity analyses

- Negative control analyses

4.5 Create reproducible workflows

Reproducible workflows allow a user to reproduce study estimates and ideally figures and tables with a "single click". In practice, this typically means running a single bash script that sources all replication scripts in a repository. These replication scripts complete data processing, data analysis, and figure/table generation. The following chapters provide detailed guidance on this topic:

- Chapter 5: Code repositories
- Chapter 6: Coding practices
- Chapter 7: Coding style
- Chapter 8: Code publication
- Chapter 9: Working with big data
- Chapter 10: Github
- Chapter 11: Unix

4.6 Process and analyze data with internal replication and masking

See my video on this topic: https://www.youtube.com/watch?v=WoYkY9MkbRE

4.7 Use reporting checklists with manuscripts

Using reporting checklists helps ensure that peer-reviewed articles contain the information needed for readers to assess the validity of your work and/or attempt to reproduce it. A collection of reporting checklists is available here: https://www.equator-network.org/about-us/what-is-a-reporting-guideline/)

4.8 Publish preprints

A preprint is a scientific manuscript that has not been peer reviewed. Preprint servers create digital object identifiers (DOIs) and can be cited in other articles and in grant applications. Because the peer review process can take many months, publishing preprints prior to or during peer review enables other scientists to immediately learn from and build on your work. Importantly, NIH allows applicants to include preprint citations in their biosketches. In most cases, we publish preprints on medRxiv.

4.9 Publish data (when possible) and replication scripts

Publishing data and replication scripts allows other scientists to reproduce your work and to build upon it. We typically publish data on Open Science Framework, share links to Github repositories, and archive code on Zenodo.

Chapter 5

Code repositories

By Kunal Mishra, Jade Benjamin-Chung, and Stephanie Djajadi

Each study has at least one code repository that typically holds R code, shell scripts with Unix code, and research outputs (results .RDS files, tables, figures). Repositories may also include datasets. This chapter outlines how to organize these files. Adhering to a standard format makes it easier for us to efficiently collaborate across projects.

5.1 Project Structure

We recommend the following directory structure:

```
0-run-project.sh
0-config.R
1 - Data-Management/
    0-prep-data.sh
    1-prep-cdph-fluseas.R
    2a-prep-absentee.R
    2b-prep-absentee-weighted.R
    3a-prep-absentee-adj.R
    3b-prep-absentee-adj-weighted.R
2 - Analysis/
    0-run-analysis.sh
    1 - Absentee-Mean/
        1-absentee-mean-primary.R
        2-absentee-mean-negative-control.R
        3-absentee-mean-CDC.R
        4-absentee-mean-peakwk.R
        5-absentee-mean-cdph2.R
        6-absentee-mean-cdph3.R
```

```
    2 - Absentee-Positivity-Check/
    3 - Absentee-P1/
    4 - Absentee-P2/
3 - Figures/
    0-run-figures.sh
    ...
4 - Tables/
    0-run-tables.sh
    ...
5 - Results/
    1 - Absentee-Mean/
        1-absentee-mean-primary.RDS
        2-absentee-mean-negative-control.RDS
        3-absentee-mean-CDC.RDS
        4-absentee-mean-peakwk.RDS
        5-absentee-mean-cdph2.RDS
        6-absentee-mean-cdph3.RDS
    ...
.gitignore
.Rproj
```

For brevity, not every directory is "expanded", but we can glean some important takeaways from what we *do* see.

5.2 `.Rproj` files

An "R Project" can be created within RStudio by going to `File >> New Project`. Depending on where you are with your research, choose the most appropriate option. This will save preferences, working directories, and even the results of running code/data (though I'd recommend starting from scratch each time you open your project, in general). Then, ensure that whenever you are working on that specific research project, you open your created project to enable the full utility of `.Rproj` files. This also automatically sets the directory to the top level of the project.

5.3 Configuration ('config') File

This is the single most important file for your project. It will be responsible for a variety of common tasks, declare global variables, load functions, declare paths, and more. *Every other file in the project* will begin with `source("0-config")`, and its role is to reduce redundancy and create an abstraction layer that allows you to make changes in one place (`0-config.R`) rather than 5 different files. To this end, paths which will be reference in multiple scripts (i.e. a `merged_data_path`) can be declared in `0-config.R` and simply referred to by its variable name in scripts. If you ever want to change things, rename them,

or even switch from a downsample to the full data, all you would then to need to do is modify the path in one place and the change will automatically update throughout your project. See the example config file for more details. The paths defined in the `0-config.R` file assume that users have opened the `.Rproj` file, which sets the directory to the top level of the project.

5.4 Order Files and Directories

This makes the jumble of alphabetized filenames much more coherent and places similar code and files next to one another. This also helps us understand how data flows from start to finish and allows us to easily map a script to its output (i.e. `2 - Analysis/1 - Absentee-Mean/1-absentee-mean-primary.R` => `5 - Results/1 - Absentee-Mean/1-absentee-mean-primary.RDS`). If you take nothing else away from this guide, this is the single most helpful suggestion to make your workflow more coherent. Often the particular order of files will be in flux until an analysis is close to completion. At that time it is important to review file order and naming and reproduce everything prior to drafting a manuscript.

5.5 Using Bash scripts to ensure reproducibility

Bash scripts are useful components of a reproducible workflow. At many of the directory levels (i.e. in `3 - Analysis`), there is a bash script that runs each of the analysis scripts. This is exceptionally useful when data "upstream" changes – you simply run the bash script. See the Unix Chapter for further details.

After running bash scripts, `.Rout` log files will be generated for each script that has been executed. It is important to check these files. Scripts may appear to have run correctly in the terminal, but checking the log files is the only way to ensure that everything has run completely.

Chapter 6

Coding practices

by Kunal Mishra, Jade Benjamin-Chung, and Stephanie Djajadi

6.1 Organizing scripts

Just as your data "flows" through your project, data should flow naturally through a script. Very generally, you want to:

1. describe the work completed in the script in a comment header
2. source your configuration file (`0-config.R`)
3. load all your data
4. do all your analysis/computation
5. save your data.

Each of these sections should be "chunked together" using comments. See this file for a good example of how to cleanly organize a file in a way that follows this "flow" and functionally separate pieces of code that are doing different things.

6.2 Documenting your code

6.2.1 File headers

Every file in a project should have a header that allows it to be interpreted on its own. It should include the name of the project and a short description for what this file (among the many in your project) does specifically. You may optionally wish to include the inputs and outputs of the script as well, though the next section makes this significantly less necessary.

```
################################################################################
# @Organization - Example Organization
# @Project - Example Project
```

```
# @Description - This file is responsible for [...]
################################################################################
```

6.2.2 Sections and subsections

Rstudio (v1.4 or more recent) supports the use of Sections and Subsections. You can easily navigate through longer scripts using the navigation pane in RStudio, as shown on the right below.

```
# Section -------

## Subsection -------

### Sub-subsection -------
```

6.2.3 Code folding

Consider using RStudio's code folding feature to collapse and expand different sections of your code. Any comment line with at least four trailing dashes (-), equal signs (=), or pound signs (#) automatically creates a code section. For example:

6.2.4 Comments in the body of your script

Commenting your code is an important part of reproducibility and helps document your code for the future. When things change or break, you'll be thankful for comments. There's no need to comment excessively or unnecessarily, but a comment describing what a large or complex chunk of code does is always helpful. See this file for an example of how to comment your code and notice that comments are always in the form of:

```
# This is a comment -- first letter is capitalized and spaced
away from the pound sign
```

6.2.5 Function documentation

Every function you write must include a header to document its purpose, inputs, and outputs. For any reproducible workflows, they are essential, because R is dynamically typed. This means, you can pass a `string` into an argument that is meant to be a `data.table`, or a `list` into an argument meant for a `tibble`. It is the responsibility of a function's author to document what each argument is meant to do and its basic type. This is an example for documenting a function (inspired by JavaDocs and R's Plumber API docs):

```
##############################################
##############################################
# Documentation: calc_fluseas_mean
# Usage: calc_fluseas_mean(data, yname)
```

```
# Description: Make a dataframe with rows for flu season and site
# and the number of patients with an outcome, the total patients,
# and the percent of patients with the outcome

# Args/Options:
# data: a data frame with variables flu_season, site, studyID, and yname
# yname: a string for the outcome name
# silent: a boolean specifying whether the function shouldn't output anything to the console (DEF

# Returns: the dataframe as described above
# Output: prints the data frame described above if silent is not True

calc_fluseas_mean = function(data, yname, silent = TRUE) {
 ### function code here

}
```

The header tells you what the function does, its various inputs, and how you might go about using the function to do what you want. Also notice that all optional arguments (i.e. ones with pre-specified defaults) follow arguments that require user input.

- **Note**: As someone trying to call a function, it is possible to access a function's documentation (and internal code) by `CMD-Left-Click`ing the function's name in RStudio
- **Note**: Depending on how important your function is, the complexity of your function code, and the complexity of different types of data in your project, you can also add "type-checking" to your function with the `assertthat::assert_that()` function. You can, for example, `assert_that(is.data.frame(statistical_input))`, which will ensure that collaborators or reviewers of your project attempting to use your function are using it in the way that it is intended by calling it with (at the minimum) the correct type of arguments. You can extend this to ensure that certain assumptions regarding the inputs are fulfilled as well (i.e. that `time_column`, `location_column`, `value_column`, and `population_column` all exist within the `statistical_input` tibble).

6.3 Object naming

Generally we recommend using nouns for objects and verbs for functions. This is because functions are performing actions, while objects are not.

Try to make your variable names both more expressive and more explicit. Being a bit more verbose is useful and easy in the age of autocompletion! For example, instead of naming a variable `vaxcov_1718`, try naming it `vaccination_coverage_2017_18`. Similarly, `flu_res` could be named

`absentee_flu_residuals`, making your code more readable and explicit.

- For more help, check out Be Expressive: How to Give Your Variables Better Names

We recommend you use **Snake_Case**.

- Base R allows `.` in variable names and functions (such as `read.csv()`), but this goes against best practices for variable naming in many other coding languages. For consistency's sake, `snake_case` has been adopted across languages, and modern packages and functions typically use it (i.e. `readr::read_csv()`). As a very general rule of thumb, if a package you're using doesn't use `snake_case`, there may be an updated version or more modern package that *does*, bringing with it the variety of performance improvements and bug fixes inherent in more mature and modern software.
- **Note**: you may also see `camelCase` throughout the R code you come across. This is *okay* but not ideal – try to stay consistent across all your code with `snake_case`.
- **Note**: again, its also worth noting there's nothing inherently wrong with using `.` in variable names, just that it goes against style best practices that are cropping up in data science, so its worth getting rid of these bad habits now.

6.4 Function calls

In a function call, use "named arguments" and put each argument on a separate line to make your code more readable.

Here's an example of what not to do when calling the function a function `calc_fluseas_mean` (defined above):

```
mean_Y = calc_fluseas_mean(flu_data, "maari_yn", FALSE)
```

And here it is again using the best practices we've outlined:

```
mean_Y = calc_fluseas_mean(
  data = flu_data,
  yname = "maari_yn",
  silent = FALSE
)
```

6.5 The here package

The `here` package is one great R package that helps multiple collaborators deal with the mess that is working directories within an R project structure. Let's

say we have an R project at the path `/home/oski/Some-R-Project`. My collaborator might clone the repository and work with it at some other path, such as `/home/bear/R-Code/Some-R-Project`. Dealing with working directories and paths explicitly can be a very large pain, and as you might imagine, setting up a Config with paths requires those paths to flexibly work for all contributors to a project. This is where the `here` package comes in and this a great vignette describing it.

6.6 Reading/Saving Data

6.6.1 `.RDS` vs `.RData` Files

One of the most common ways to load and save data in Base R is with the `load()` and `save()` functions to serialize multiple objects in a single `.RData` file. The biggest problems with this practice include an inability to control the names of things getting loaded in, the inherent confusion this creates in understanding older code, and the inability to load individual elements of a saved file. For this, we recommend using the RDS format to save R objects.

- **Note**: if you have many related R objects you would have otherwise saved all together using the `save` function, the functional equivalent with `RDS` would be to create a (named) list containing each of these objects, and saving it.

6.6.2 CSVs

Once again, the `readr` package as part of the Tidvyerse is great, with a much faster `read_csv()` than Base R's `read.csv()`. For massive CSVs (> 5 GB), you'll find `data.table::fread()` to be the fastest CSV reader in any data science language out there. For writing CSVs, `readr::write_csv()` and `data.table::fwrite()` outclass Base R's `write.csv()` by a significant margin as well.

6.7 Integrating Box and Dropbox

Box and Dropbox are cloud-based file sharing systems that are useful when dealing with large files. When our scripts generate large output files, the files can slow down the workflow if they are pushed to GitHub. This makes collaboration difficult when not everyone has a copy of the file, unless we decide to duplicate files and share them manually. The files might also take up a lot of local storage. Box and Dropbox help us avoid these issues by automatically storing the files, reading data, and writing data back to the cloud.

Box and Dropbox are separate platforms, but we can use either one to store and share files. To use them, we can install the packages that have been created to integrate Box and Dropbox into R. The set-up instructions are detailed below.

Make sure to authenticate before reading and writing from either Box or Dropbox. The authentication commands should go in the configuration file; it only needs to be done once. This will prompt you to give your login credentials for Box and Dropbox and will allow your application to access your shared folders.

6.7.1 Box

Follow the instructions in this section to use the `boxr` package. Note that there are a few setup steps that need to be done on the box website before you can use the `boxr` package, explained here in the section "Creating an Interactive App." This gets the authentication keys that must be put in box. Once that is done, add the authentication keys to your code in the configuration file, with `box_auth(client_id = "<your_client_id>", client_secret = "<your_client_secret_id>")`. It is also important to set the default working directory so that the code can reference the correct folder in box: `box_setwd(<folder_id>)`. The folder ID is the sequence of digits at the end of the URL.

Further details can be found here.

6.7.2 Dropbox

Follow the instructions at this link to use the `rdrop2` package. Similar to the `boxr` package, you must authenticate before reading and writing from Dropbox, which can be done by adding `drop_auth()` to the configuration file.

Saving the authentication token is not required, although it may be useful if you plan on using Dropbox frequently. To do so, save the token with the following commands. Tokens are valid until they are manually revoked.

```
# first time only
# save the output of drop_auth to an RDS file
token <- drop_auth()
# this token only has to be generated once, it is valid until revoked
saveRDS(token, "/path/to/tokenfile.RDS")

# all future usages
# to use a stored token, provide the rdstoken argument
drop_auth(rdstoken = "/path/to/tokenfile.RDS")
```

6.8 Tidyverse

Throughout this document there have been references to the Tidyverse, but this section is to explicitly show you how to transform your Base R tendencies to Tidyverse (or Data.Table, Tidyverse's performance-optimized competitor). For most of our work that does not utilize very large datasets, we recommend that you code in Tidyverse rather than Base R. Tidyverse is quickly becoming the

gold standard in R data analysis and modern data science packages and code should use Tidyverse style and packages unless there's a significant reason not to (i.e. big data pipelines that would benefit from Data.Table's performance optimizations).

The package author has published a great textbook on R for Data Science, which leans heavily on many Tidyverse packages and may be worth checking out.

The following list is not exhaustive, but is a compact overview to begin to translate Base R into something better:

Base R	Better Style, Performance, and Utility
—	—
`read.csv()`	`readr::read_csv()` or `data.table::fread()`
`write.csv()`	`readr::write_csv()` or `data.table::fwrite()`
`readRDS`	`readr::read_rds()`
`saveRDS()`	`readr::write_rds()`
—	—
`data.frame()`	`tibble::tibble()` or `data.table::data.table()`
`rbind()`	`dplyr::bind_rows()`
`cbind()`	`dplyr::bind_cols()`
`df$some_column`	`df %>% dplyr::pull(some_column)`
`df$some_column = ...`	`df %>% dplyr::mutate(some_column = ...)`
`df[get_rows_condition,]`	`df %>% dplyr::filter(get_rows_condition)`
`df[,c(col1, col2)]`	`df %>% dplyr::select(col1, col2)`
`merge(df1, df2, by = ..., all.x = ..., all.y = ...)`	`df1 %>% dplyr::left_join(df2, by = ...)` or `dplyr::full_join` or `dplyr::inner_join` or `dplyr::right_join`
—	—
`str()`	`dplyr::glimpse()`
`grep(pattern, x)`	`stringr::str_which(string, pattern)`
`gsub(pattern, replacement, x)`	`stringr::str_replace(string, pattern, replacement)`
`ifelse(test_expression, yes, no)`	`if_else(condition, true, false)`

Base R	Better Style, Performance, and Utility
Nested: `ifelse(test_expression1, yes1, ifelse(test_expression2, yes2, ifelse(test_expression3, yes3, no)))`	`case_when(test_expression1 ~ yes1, test_expression2 ~ yes2, test_expression3 ~ yes3, TRUE ~ no)`
`proc.time()`	`tictoc::tic()` and `tictoc::toc()`
`stopifnot()`	`assertthat::assert_that()` or `assertthat::see_if()` or `assertthat::validate_that()`

For a more extensive set of syntactical translations to Tidyverse, you can check out this document.

Working with Tidyverse within functions can be somewhat of a pain due to non-standard evaluation (NSE) semantics. If you're an avid function writer, we'd recommend checking out the following resources:

- Tidy Eval in 5 Minutes (video)
- Tidy Evaluation (e-book)
- Data Frame Columns as Arguments to Dplyr Functions (blog)
- Standard Evaluation for *_join (stackoverflow)
- Programming with dplyr (package vignette)

6.9 Coding with R and Python

If you're using both R and Python, you may wish to check out the Feather package for exchanging data between the two languages extremely quickly.

6.10 Reviewing Code

Before publishing new changes, it is important to ensure that the code has been tested and well-documented. GitHub makes it possible to document all of these changes in a pull request. Pull requests can be used to describe changes in a branch that are ready to be merged with the base branch (more information in the GitHub section). Github allows users to create a pull request template in a repository to standardize and customize the information in a pull request. When you add a pull request template to your repository, everyone will automatically see the template's contents in the pull request body.

6.10.1 Creating a Pull Request Template

Follow the instructions below to add a pull request template to a repository. More details can be found at this GitHub link.

1. On GitHub, navigate to the main page of the repository.

2. Above the file list, click `Create new file`.
3. Name the file `pull_request_template.md`. GitHub will not recognize this as the template if it is named anything else. The file must be on the `master` branch.
    1. To store the file in a hidden directory instead of the main directory, name the file `.github/pull_request_template.md`.
4. In the body of the new file, add your pull request template. This could include:
    - A summary of the changes proposed in the pull request
    - How the change has been tested
    - @mentions of the person or team responsible for reviewing proposed changes

Here is an example pull request template.

```
# Description

## Summary of change

Please include a summary of the change, including any new functions added and example usage.

## Link to Spec

Please include a link to the Trello card or Google document with details of the task.

## Who should review the pull request?
@ ...
```

Chapter 7

Coding style

by Kunal Mishra, Jade Benjamin-Chung, and Stephanie Djajadi

7.1 Comments

1. **File Headers** - Every file in a project should have a header that allows it to be interpreted on its own. It should include the name of the project and a short description for what this file (among the many in your project) does specifically. You may optionally wish to include the inputs and outputs of the script as well, though the next section makes this significantly less necessary.

```
################################################################################
# @Organization - Example Organization
# @Project - Example Project
# @Description - This file is responsible for [...]
################################################################################
```

2. **File Structure** - Just as your data "flows" through your project, data should flow naturally through a script. Very generally, you want to 1) source your config => 2) load all your data => 3) do all your analysis/computation => save your data. Each of these sections should be "chunked together" using comments. See this file for a good example of how to cleanly organize a file in a way that follows this "flow" and functionally separate pieces of code that are doing different things.
    - **Note**: If your computer isn't able to handle this workflow due to RAM or requirements, modifying the ordering of your code to accomodate it won't be ultimately helpful and your code will be fragile, not to mention less readable and messy. You need to look into high-performance computing (HPC) resources in this case.

3. **Single-Line Comments** - Commenting your code is an important part of reproducibility and helps document your code for the future. When things change or break, you'll be thankful for comments. There's no need to comment excessively or unnecessarily, but a comment describing what a large or complex chunk of code does is always helpful. See this file for an example of how to comment your code and notice that comments are always in the form of:

```
# This is a comment -- first letter is capitalized and spaced
away from the pound sign
```

4. **Multi-Line Comments** - Occasionally, multi-line comments are necessary. Don't add line breaks manually to a single-line comment for the purpose of making it "fit" on the screen. Instead, in RStudio > Tools > Global Options > Code > "Soft-wrap R source files" to have lines wrap around. Format your multi-line comments like the file header from above.

7.2 Line breaks

- For `ggplot` calls and `dplyr` pipelines, do not crowd single lines. Here are some nontrivial examples of "beautiful" pipelines, where beauty is defined by coherence:

```
# Example 1
school_names = list(
  OUSD_school_names = absentee_all %>%
    filter(dist.n == 1) %>%
    pull(school) %>%
    unique %>%
    sort,

  WCCSD_school_names = absentee_all %>%
    filter(dist.n == 0) %>%
    pull(school) %>%
    unique %>%
    sort
)

# Example 2
absentee_all = fread(file = raw_data_path) %>%
  mutate(program = case_when(schoolyr %in% pre_program_schoolyrs ~ 0,
                             schoolyr %in% program_schoolyrs ~ 1)) %>%
  mutate(period = case_when(schoolyr %in% pre_program_schoolyrs ~ 0,
                            schoolyr %in% LAIV_schoolyrs ~ 1,
                            schoolyr %in% IIV_schoolyrs ~ 2)) %>%
  filter(schoolyr != "2017-18")
```

And of a complex `ggplot` call:

```
# Example 3
ggplot(data=data,
       mapping=aes_string(x="year", y="rd", group=group)) +

  geom_point(mapping=aes_string(col=group, shape=group),
             position=position_dodge(width=0.2),
             size=2.5) +

  geom_errorbar(mapping=aes_string(ymin="lb", ymax="ub", col=group),
                position=position_dodge(width=0.2),
                width=0.2) +

  geom_point(position=position_dodge(width=0.2),
             size=2.5) +

  geom_errorbar(mapping=aes(ymin=lb, ymax=ub),
                position=position_dodge(width=0.2),
                width=0.1) +

  scale_y_continuous(limits=limits,
                     breaks=breaks,
                     labels=breaks) +

  scale_color_manual(std_legend_title,values=cols,labels=legend_label) +
  scale_shape_manual(std_legend_title,values=shapes, labels=legend_label) +
  geom_hline(yintercept=0, linetype="dashed") +
  xlab("Program year") +
  ylab(yaxis_lab) +
  theme_complete_bw() +
  theme(strip.text.x = element_text(size = 14),
        axis.text.x = element_text(size = 12)) +
  ggtitle(title)
```

Imagine (or perhaps mournfully recall) the mess that can occur when you don't strictly style a complicated `ggplot` call. Trying to fix bugs and ensure your code is working can be a nightmare. Now imagine trying to do it with the same code 6 months after you've written it. Invest the time now and reap the rewards as the code practically explains itself, line by line.

7.3 Automated Tools for Style and Project Workflow

7.3.1 Styling

1. **Code Autoformatting** - RStudio includes a fantastic built-in utility (keyboard shortcut: `CMD-Shift-A`) for autoformatting highlighted chunks of code to fit many of the best practices listed here. It generally makes code more readable and fixes a lot of the small things you may not feel like fixing yourself. Try it out as a "first pass" on some code of yours that *doesn't* follow many of these best practices!

2. **Assignment Aligner** - A cool R package allows you to very powerfully format large chunks of assignment code to be much cleaner and much more readable. Follow the linked instructions and create a keyboard shortcut of your choosing (recommendation: `CMD-Shift-Z`). Here is an example of how assignment aligning can dramatically improve code readability:

```
# Before
OUSD_not_found_aliases = list(
  "Brookfield Village Elementary" = str_subset(string = OUSD_school_shapes$schnam, patt
  "Carl Munck Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern = "
  "Community United Elementary School" = str_subset(string = OUSD_school_shapes$schnam
  "East Oakland PRIDE Elementary" = str_subset(string = OUSD_school_shapes$schnam, patt
  "EnCompass Academy" = str_subset(string = OUSD_school_shapes$schnam, pattern = "EnCo
  "Global Family School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "G
  "International Community School" = str_subset(string = OUSD_school_shapes$schnam, pa
  "Madison Park Lower Campus" = "Madison Park Academy TK-5",
  "Manzanita Community School" = str_subset(string = OUSD_school_shapes$schnam, patter
  "Martin Luther King Jr Elementary" = str_subset(string = OUSD_school_shapes$schnam, p
  "PLACE @ Prescott" = "Preparatory Literary Academy of Cultural Excellence",
  "RISE Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "
)

# After
OUSD_not_found_aliases = list(
  "Brookfield Village Elementary"      = str_subset(string = OUSD_school_shapes$schnam
  "Carl Munck Elementary"              = str_subset(string = OUSD_school_shapes$schnam
  "Community United Elementary School" = str_subset(string = OUSD_school_shapes$schnam
  "East Oakland PRIDE Elementary"      = str_subset(string = OUSD_school_shapes$schnam
  "EnCompass Academy"                  = str_subset(string = OUSD_school_shapes$schnam
  "Global Family School"               = str_subset(string = OUSD_school_shapes$schnam
  "International Community School"     = str_subset(string = OUSD_school_shapes$schnam
  "Madison Park Lower Campus"          = "Madison Park Academy TK-5",
  "Manzanita Community School"         = str_subset(string = OUSD_school_shapes$schnam
  "Martin Luther King Jr Elementary"   = str_subset(string = OUSD_school_shapes$schnam
  "PLACE @ Prescott"                   = "Preparatory Literary Academy of Cultural Exc
  "RISE Community School"              = str_subset(string = OUSD_school_shapes$schnam
)
```

3. **StyleR** - Another cool R package from the Tidyverse that can be powerful

and used as a first pass on entire projects that need refactoring. The most useful function of the package is the `style_dir` function, which will style all files within a given directory. See the function's documentation and the vignette linked above for more details.

- **Note**: The default Tidyverse styler is subtly different from some of the things we've advocated for in this document. Most notably we differ with regards to the assignment operator (`<-` vs `=`) and number of spaces before/after "tokens" (i.e. Assignment Aligner add spaces before `=` signs to align them properly). For this reason, we'd recommend the following: `style_dir(path = ..., scope = "line_breaks", strict = FALSE)`. You can also customize StyleR even more if you're really hardcore.
- **Note**: As is mentioned in the package vignette linked above, StyleR modifies things *in-place*, meaning it overwrites your existing code and replaces it with the updated, properly styled code. This makes it a good fit on projects *with version control*, but if you don't have backups or a good way to revert back to the intial code, I wouldn't recommend going this route.

4. **Linter** - Linters are programming tools that check adherence to a given style, syntax errors, and possible semantic issues. The R linter, called `lintr`, can be found in this package. It helps keep files consistent across different authors and even different organizations. For example, it notifies you if you have unused variables, global variables with no visible binding, not enough or superflous whitespace, and improper use of parentheses or brackets. A list of its other purposes can be found in this link, and most guidelines are based on Hadley Wickham's R Style Guide.
    - **Note**: You can customize your settings to set defaults or to exclude files. More details can be found here.
    - **Note**: The `lintr` package goes hand in hand with the `styler` package. The styler can be used to automatically fix the problems that the lintr catches.

Chapter 8

Code Publication

by Nolan Pokpongkiat

8.1 Checklist overview

1. Fill out file headers
2. Clean up comments
3. Document functions
4. Remove deprecated filepaths
5. Ensure project runs via bash
6. Complete the README
7. Clean up feature branches
8. Create Github release

8.2 Fill out file headers

Every file in a project should have a header that allows it to be interpreted on its own. It should include the name of the project and a short description for what this file (among the many in your project) does specifically. See template here.

8.3 Clean up comments

Make sure comments in the code are for code documentation purposes only. Do not leave comments to self in the final script files.

8.4 Document functions

Every function you write must include a header to document its purpose, inputs, and outputs. See template for the function documentation here.

8.5 Remove deprecated filepaths

All file paths should be defined in 0-config.R, and should be set relative to the project working directory. All absolute file paths from your local computer should be removed, and replaced with a relative path. If a third party were to re-run this analysis, if they need to download data from a separate source and change a filepath in the 0-config.R to match, make sure to specify in the README which line of 0-config.R needs to be substituted.

8.6 Ensure project runs via bash

The project should be configured to be entirely reproducible by running a master bash script, run-project.sh, which should live at the top directory. This bash script can call other bash scripts in subfolders, if necessary. Bash scripts should use the runFileSaveLogs utility script, which is a wrapper around the Rscript command, allowing you to specify where .Rout log files are moved after the R scripts are run.

See usage and documentation here.

8.7 Complete the README

A README.md should live at the top directory of the project. This usually includes a Project Overview and a Directory Structure, along with the names of the contributors and the Creative Commons License. See below for a template:

> **Overview**
>
> To date, coronavirus testing in the US has been extremely limited. Confirmed COVID-19 case counts underestimate the total number of infections in the population. We estimated the total COVID-19 infections – both symptomatic and asymptomatic – in the US in March 2020. We used a semi-Bayesian approach to correct for bias due to incomplete testing and imperfect test performance.
>
> **Directory structure**
>
> - 0-config.R: configuration file that sets data directories, sources base functions, and loads required libraries
> - 0-base-functions: folder containing scripts with functions used in the analysis

  - 0-base-functions.R: R script containing general functions used across the analysis
  - 0-bias-corr-functions.R: R script containing functions used in bias correction
  - 0-bias-corr-functions-undertesting.R: R script containing functions used in bias correction to estimate the percentage of underestimation due to incomplete testing vs. imperfect test accuracy
  - 0-prior-functions.R: R script containing functions to generate priors
- 1-data: folder containing data processing scripts NOTE: some scripts are deprecated
- 2-analysis: folder containing analysis scripts. To rerun all scripts in this subdirectory, run the bash script 0-run-analysis.sh.
  - 1-obtain-priors-state.R: obtain priors for each state
  - 2-est-expected-cases-state.R: estimate expected cases in each state
  - 3-est-expected-cases-state-perf-testing.R: estimate expected cases in each state, estimate the percentage of underestimation due to incomplete testing vs. imperfect test accuracy
  - 4-obtain-testing-protocols.R: find testing protocols for each state.
  - 5-summarize-results.R: summarize results; obtain results for in text numerical results.
- 3-figure-table-scripts: folder containing figure scripts. To rerun all scripts in this subdirectory, run the bash script 0-run-figs.sh.
  - 1-fig-testing.R: creates plot of testing patterns by state over time
  - 2-fig-cases-usa-state-bar.R: creates bar plot of confirmed vs. estimated infections by state
  - 3a-fig-map-usa-state.R: creates map of confirmed vs. estimated infections by state
  - 3b-fig-map-usa-state-shiny.R: creates map of confirmed vs. estimated infections by state with search functionality by state
  - 4-fig-priors.R: creates figure with priors for US as a whole

  - 5-fig-density-usa.R: creates figure of distribution of estimated cases in the US
  - 6-table-data-quality.R: creates table of data quality grading from COVID Tracking Project
  - 7-fig-testpos.R: creates figure of the probability of testing positive among those tested by state
  - 8-fig-percent-undertesting-state.R: creates figure of the percentage of under estimation due to incomplete testing
- 4-figures: folder containing figure files.
- 5-results: folder containing analysis results objects.
- 6-sensitivity: folder containing scripts to run the sensitivity analyses

Contributors: Jade Benjamin-Chung, Sean L. Wu, Anna Nguyen, Stephanie Djajadi, Nolan N. Pokpongkiat, Anmol Seth, Andrew Mertens

Wu SL, Mertens A, Crider YS, Nguyen A, Pokpongkiat NN, Djajadi S, et al. Substantial underestimation of SARS-CoV-2 infection in the United States due to incomplete testing and imperfect test accuracy. medRxiv. 2020; 2020.05.12.20091744. doi:10.1101/2020.05.12.20091744

When possible, also include a description of the RDS results that are generated, detailing what data sources were used, where the script lives that creates it, and what information the RDS results hold.

8.8 Clean up feature branches

In the remote repository on Github, all feature branches aside from master should be merged in and deleted. All outstanding PRs should be closed.

8.9 Create Github release

Once all of these items are verified, create a tag to make a Github release, which will tag the repository, creating a marker at this specific point in time.

Detailed instructions here.

Chapter 9

Working with Big Data

by Kunal Mishra and Jade Benjamin-Chung

9.1 The data.table package

It may also be the case that you're working with very large datasets. Generally I would define this as 10+ million rows. As is outlined in this document, the 3 main players in the data analysis space are Base R, `Tidvyerse` (more specifically, `dplyr`), and `data.table`. For a majority of things, Base R is inferior to both `dplyr` and `data.table`, with concise but less clear syntax and less speed. `Dplyr` is architected for medium and smaller data, and while its very fast for everyday usage, it trades off maximum performance for ease of use and syntax compared to `data.table`. An overview of the `dplyr` vs `data.table` debate can be found in this stackoverflow post and all 3 answers are worth a read.

You can also achieve a performance boost by running `dplyr` commands on `data.table`s, which I find to be the best of both worlds, given that a `data.table` is a special type of `data.frame` and fairly easy to convert with the `as.data.table()` function. The speedup is due to `dplyr`'s use of the `data.table` backend and in the future this coupling should become even more natural.

If you want to test whether using a certain coding approach increases speed, consider the `tictoc` package. Run `tic()` before a code chunk and `toc()` after to measure the amount of system time it takes to run the chunk. For example, you might use this to decide if you *really* need to switch a code chunk from `dplyr` to `data.table`.

9.2 Using downsampled data

In our studies with very large datasets, we save "downsampled" data that usually includes a 1% random sample stratified by any important variables, such as year or household id. This allows us to efficiently write and test our code without having to load in large, slow datasets that can cause RStudio to freeze. Be very careful to be sure which dataset you are working with and to label results output accordingly.

9.3 Optimal RStudio set up

Using the following settings will help ensure a smooth experience when working with big data. In RStudio, go to the "Tools" menu, then select "Global Options". Under "General":

Workspace

- **Uncheck** Restore RData into workspace at startup
- Save workspace to RData on exit – choose **never**

History

- **Uncheck** Always save history

Unfortunately RStudio often gets slow and/or freezes after hours working with big datasets. Sometimes it is much more efficient to just use Terminal / gitbash to run code and make updates in git.

Chapter 10

Github

by Stephanie Djajadi and Nolan Pokpongkiat

10.1 Basics

- A detailed tutorial of Git can be found here on the CS61B website.
- If you are already familiar with Git, you can reference the summary at the end of Section B.
- If you have made a mistake in Git, you can refer to this article to undo, fix, or remove commits in git.

10.2 Github Desktop

While knowing how to use Git on the command line will always be useful since the full power of Git and its customizations and flexibilty is designed for use with the command line, Github also provides Github Desktop as an graphical interface to do basic git commands; you can do all of the basic functions of Git using this desktop app. Feel free to use this as an alternative to Git on the command line if you prefer.

10.3 Git Branching

Branches allow you to keep track of multiple versions of your work simultaneously, and you can easily switch between versions and merge branches together once you've finished working on a section and want it to join the rest of your code. Here are some cases when it may be a good idea to branch:

- You may want to make a dramatic change to your existing code (called refactoring) but it will break other parts of your project. But you want

to be able to simultaneously work on other parts or you are collaborating with others, and you don't want to break the code for them.
- You want to start working on a new part of the project, but you aren't sure yet if your changes will work and make it to the final product.
- You are working with others and don't want to mix up your current work with theirs, even if you want to bring your work together later in the future.

A detailed tutorial on Git Branching can be found here. You can also find instructions on how to handle merge conflicts when joining branches together.

10.4 Example Workflow

A standard workflow when starting on a new project and contributing code looks like this:

Command	Description
SETUP: FIRST TIME ONLY: `git clone <url> <directory_name>`	Clone the repo. This copies of all the project files in its current state on Github to your local computer.
1. `git pull origin master`	update the state of your files to match the most current version on GitHub
2. `git checkout -b <new_branch_name>`	create new branch that you'll be working on and go to it
3. Make some file changes	work on your feature/implementation
4. `git add -p`	add changes to stage for commit, going through changes line by line
5. `git commit -m <commit message>`	commit files with a message
6. `git push -u origin <branch_name>`	push branch to remote and set to track (-u only needed if this is first push)
7. Repeat step 4-5.	work and commit often
8. `git push`	push work to remote branch for others to view
9. Follow the link given from the `git push` command to submit a pull request (PR) on GitHub online	PR merges in work from your branch into master
(10.) Your changes and PR get approved, your reviewer deletes your remote branch upon merging	
11. `git fetch --all --prune`	clean up your local git by untracking deleted remote branches

Other helpful commands are listed below.

10.5 Commonly Used Git Commands

Command	Description
`git clone <url> <directory_name>`	clone a repository, only needs to be done the first time
`git pull origin master`	pull from `master` before making any changes
`git branch`	check what branch you are on
`git branch -a`	check what branch you are on + all remote branches
`git checkout -b <new_branch_name>`	create new branch and go to it (only necessary when you create a new branch)
`git checkout <branch name>`	switch to branch
`git add <file name>`	add file to stage for commit
`git add -p`	adds changes to commit, showing you changes one by one
`git commit -m <commit message>`	commit file with a message
`git push -u origin <branch_name>`	push branch to remote and set to track (-u only works if this is first push)
`git branch --set-upstream-to origin <branch_name>`	set upstream to `origin/<branch_name>` (use if you forgot -u on first push)
`git push origin <branch_name>`	push work to branch
`git checkout <branch_name> git merge master`	switch to branch and merge changes from `master` into `<branch_name>` (two commands)
`git merge <branch_name> master`	switch to branch and merge changes from `master` into `<branch_name>` (one command)
`git checkout --track origin/<branch_name>`	pulls a remote branch and creates a local branch to track it (use when trying to pull someone else's branch onto your local computer)
`git push --delete <remote_name> <branch_name>`	delete remote branch
`git branch -d <branch_name>`	deletes local branch, -D to force
`git fetch --all --prune`	untrack deleted remote branches

10.6 How often should I commit?

It is good practice to commit every 15 minutes, or every time you make a significant change. It is better to commit more rather than less.

10.7 What should be pushed to Github?

Never push .Rout files! If someone else runs an R script and creates an .Rout file at the same time and both of you try to push to github, it is incredibly difficult to reconcile these two logs. If you run logs, keep them on your own system or (preferably) set up a shared directory where all logs are name and date timestamped.

There is a standardized `.gitignore` for `R` which you can download and add to your project. This ensures you're not committing log files or things that would otherwise best be left ignored to GitHub. This is a great discussion of project-oriented workflows, extolling the virtues of a self-contained, portable projects, for your reference.

Chapter 11

Unix commands

by Stephanie Djajadi, Kunal Mishra, Anna Nguyen, and Jade Benjamin-Chung

We typically use Unix commands in Terminal (for Mac users) or Git Bash (for Windows users) to

1. Run a series of scripts in parallel or in a specific order to reproduce our work
2. To check on the progress of a batch of jobs
3. To use git and push to github

11.1 Basics

On the computer, there is a desktop with two folders, `folder1` and `folder2`, and a file called `file1.` Inside `folder1`, we have a file called `file2.` Mac users can run these commands on their terminal; it is recommended that Windows users use Git Bash, not Windows PowerShell.

11.2 Syntax for both Mac/Windows

When typing in directories or file names, quotes are necessary if the name includes spaces.

Command	Description
`cd desktop/folder1`	Change directory to `folder1`
`pwd`	Print working directory
`ls`	List files in the directory
`cp "file2" "newfile2"`	Copy file (remember to include file extensions when typing in file names like `.pdf` or `.R`)

Command	Description
`mv "newfile2" "file3"`	Rename `newfile2` to `file3`
`cd ..`	Go to parent of the working directory (in this case, `desktop`)
`mv "file1" folder2`	Move `file1` to `folder2`
`mkdir folder3`	Make a new folder in `folder2`
`rm <filename>`	Remove files
`rm -rf folder3`	Remove directories (`-r` will attempt to remove the directory recursively, `-rf` will force removal of the directory)
`clear`	Clear terminal screen of all previous commands

11.3 Running Bash Scripts

Windows	Mac / Linux	Description
`chmod +750 <filename.sh>`	`chmod +x <filename.sh>`	Change access permissions for a file (only needs to be done once)
`./<filename.sh>`	`./<filename.sh>`	Run file (`./` to run any executable file)
`bash bash_script_name.sh &`	`bash bash_script_name.sh &`	Run shell script in the background

11.4 Running Rscripts in Windows

Note: This code seems to work only with Windows Command Prompt, not with Git Bash.

When R is installed, it comes with a utility called Rscript. This allows you to run R commands from the command line. If Rscript is in your `PATH,` then typing Rscript into the command line, and pressing enter, will not error. Otherwise, to use Rscript, you will either need to add it to your PATH (as an environment variable), or append the full directory of the location of Rscript on your machine. To find the full directory, search for where R is installed your computer. For instance, it may be something like below (this will vary depending on what version of R you have installed):

```
C:\Program Files\R\R-3.6.0\bin
```

For appending the `PATH` variable, please view this link. I strongly recommend

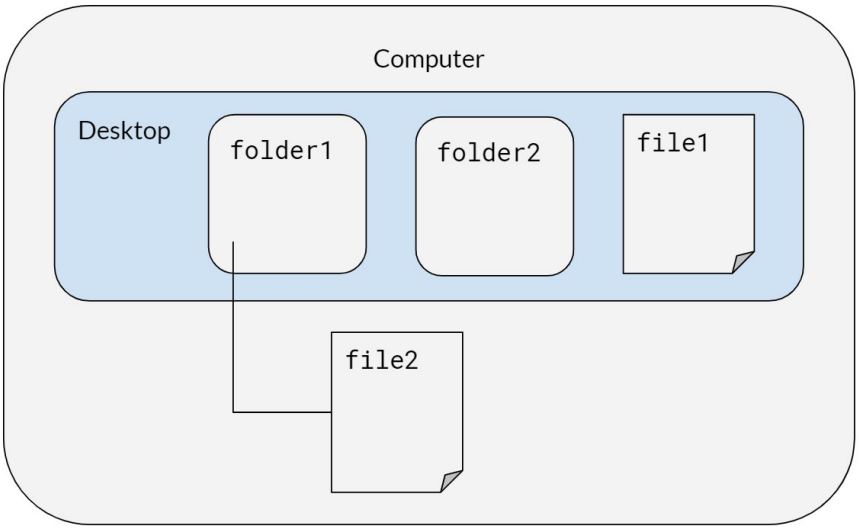

Figure 11.1: Here is our example desktop.

completing this option.

If you add the PATH as an environment variable, then you can run this line of code to test: `Rscript -e "cat('this is a test')"`, where the `-e` flag refers to the expression that will be executed.

If you do not add the PATH as an environment variable, then you can run this line of code to replicate the results from above: `"C:\Program Files\R\R-3.6.0\bin" -e "cat('this is a test')"`

To run an R script from the command line, we can say: `Rscript -e "source('C:/path/to/script/some_code.R')"`

11.4.1 Common Mistakes

- Remember to include all of the quotation marks around file paths that have a spaces.
- If you attempt to run an R script but run into `Error: '\U' used without hex digits in character string starting "'C:\U"`, try replacing all \ with \\ or /.

11.5 Checking tasks and killing jobs

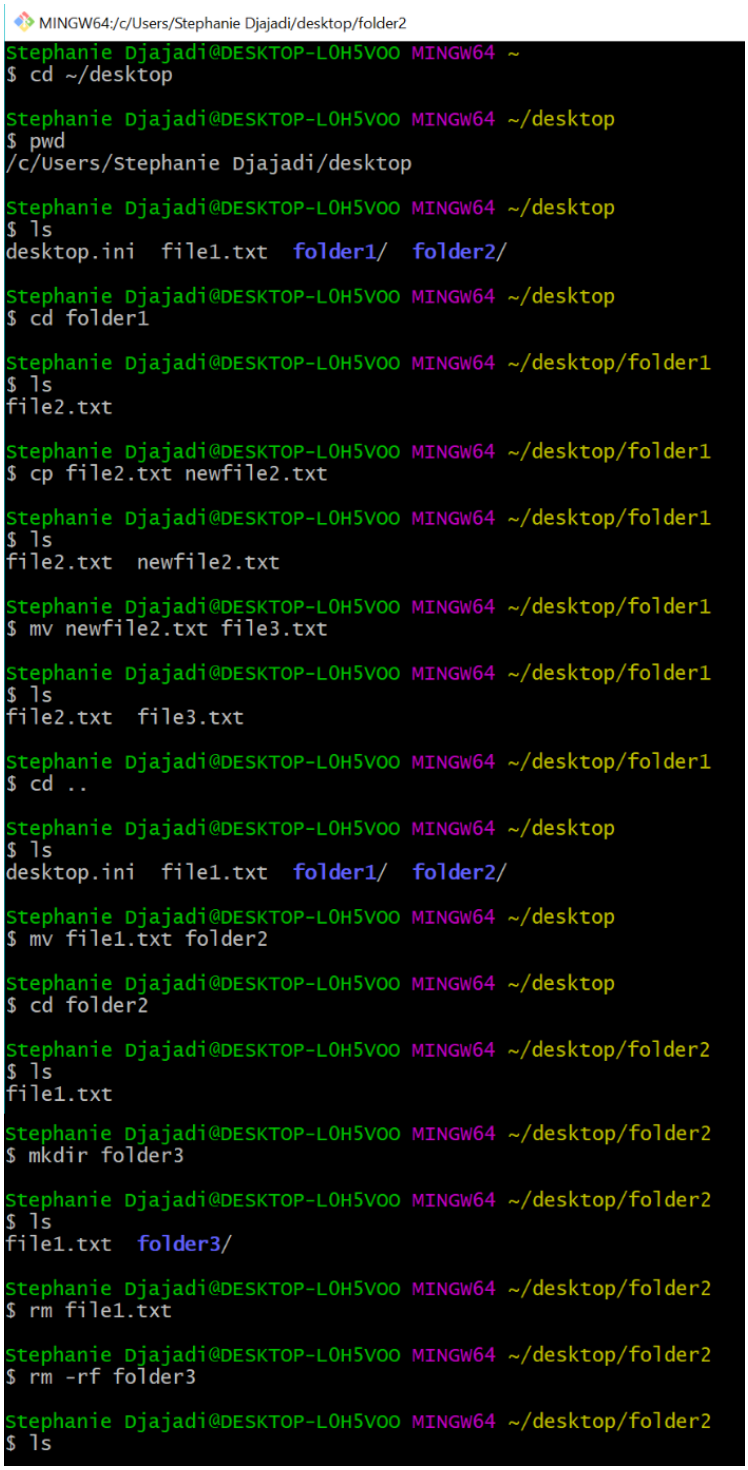

Figure 11.2: Here is an example of what your terminal might look like after executing the commands in the order listed above.

Windows	Mac / Linux	Description
`tasklist`	`ps -v`	List all processes on the command line
	`top -o [cpu/rsize]`	List all running processes, sorted by CPU or memory usage
`taskkill /F /PID pid_number`	`kill <PID_number>`	Kill a process by its process ID
`taskkill /IM "process name" /F`		Kill a process by its name
`start /b program.exe`		Runs jobs in the background (exclude `/b` if you want the program to run in a new console)
	`nohup`	Prevents jobs from stopping
	`disown`	Keeps jobs running in the background even if you close R
`taskkill /?`		Help, lists out other commands

To kill a task in Windows, you can also go to Task Manager > More details > Select your desired app > Click on End Task.

11.6 Running big jobs

For big data workflows, the concept of "backgrounding" a bash script allows you to start a "job" (i.e. run the script) and leave it overnight to run. At the top level, a bash script (`0-run-project.sh`) that simply calls the directory-level bash scripts (i.e. `0-prep-data.sh`, `0-run-analysis.sh`, `0-run-figures.sh`, etc.) is a powerful tool to rerun every script in your project. See the included example bash scripts for more details.

- **Running Bash Scripts in Background**: Running a long bash script is not trivial. Normally you would run a bash script by opening a terminal and typing something like `./run-project.sh`. But what if you leave your computer, log out of your server, or close the terminal? Normally, the bash script will exit and fail to complete. To run it in background, type `./run-project.sh &; disown`. You can see the job running (and CPU utilization) with the command `top` or `ps -v` and check your memory with `free -h`.

Alternatively, to keep code running in the background even when an SSH connection is broken, you can use `tmux`. In terminal or gitbash follow the steps below. This site has useful tips on using `tmux`.

```
# create a new tmux session called session_name
tmux new -ssession_name

# run your job of interest
R CMD BATCH myjob.R &

# check that it is running
ps -v

# to exit the tmux session (Mac)
ctrl + b
d

# to reopen the tmux session to kill the job or
# start another job
tmux attach -tsession_name
```

- **Deleting Previously Computed Results**: One helpful lesson we've learned is that your bash scripts should remove previous results (computed and saved by scripts run at a previous time) so that you never mix results from one run with a previous run. This can happen when an R script errors out before saving its result, and can be difficult to catch because your previously saved result exists (leading you to believe everything ran correctly).
- **Ensuring Things Ran Correctly**: You should check the `.Rout` files generated by the R scripts run by your bash scripts for errors once things are run. A utility file is include in this repository, called `runFileSaveLogs`, and is used by the example bash scripts to… run files and save the generated logs. It is an awesome utility and one I definitely recommend using. Before using `runFileSaveLogs`, it is necessary to put the file in the home working directory. For help and documentation, you can use the command `./runFileSaveLogs -h`. See example code and example usage for `runFileSaveLogs` below.

11.6.1 Example code for `runfileSaveLogs`

```
#!/usr/bin/env python3
# Type "./runFileSaveLogs -h" for help

import os
import sys
```

```
import argparse
import getpass
import datetime
import shutil
import glob
import pathlib

# Setting working directory to this script's current directory
os.chdir(os.path.dirname(os.path.abspath(__file__)))

# Setting up argument parser
parser = argparse.ArgumentParser(description='Runs the argument R script(s) - in parallel if spec

# Function ensuring that the file is valid
def is_valid_file(parser, arg):
    if not os.path.exists(arg):
        parser.error("The file %s does not exist!" % arg)
    else:
        return arg

# Function ensuring that the directory is valid
def is_valid_directory(parser, arg):
    if not os.path.isdir(arg):
        parser.error("The specified path (%s) is not a directory!" % arg)
    else:
        return arg

# Additional arguments that can be added when running runFileSaveLogs
parser.add_argument('-p', '--parallel', action='store_true', help="Runs the argument R scripts in
parser.add_argument("-i", "--identifier", help="Adds an identifier to the directory name where th
parser.add_argument('filenames', nargs='+', type=lambda x: is_valid_file(parser, x))

args = parser.parse_args()
args_dict = vars(args)

print(args_dict)

# Run given R Scripts
for filename in args_dict["filenames"]:
  system_call = "R CMD BATCH" + " " + filename
  if args_dict["parallel"]:
    system_call = "nohup" + " " + system_call + " &"

  os.system(system_call)
```

```
# Create the directory (and any parents) of the log files
currentUser = getpass.getuser()
currentTime = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
logDirPrefix = "/home/kaiserData/logs/" # Change to the directory where the logs shoul
logDir = logDirPrefix + currentTime + "-" + currentUser

# If specified, adds the identifier to the filename of the log
if args.identifier is not None:
  logDir += "-" + args.identifier

logDir += "/"

pathlib.Path(logDir).mkdir(parents=True, exist_ok=True)

# Find and move all logs to this new directory
currentLogPaths = glob.glob('./*.Rout')

for currentLogPath in currentLogPaths:
  filename = currentLogPath.split("/")[-1]
  shutil.move(currentLogPath, logDir + filename)
```

11.6.2 Example usage for `runfileSaveLogs`

This example bash script runs files and generates logs for five scripts in the `kaiserflu/3-figures` folder. Note that the `-i` flag is used as an identifier to add `figures` to the filename of each log.

```
#!/bin/bash

# Copy utility run script into this folder for concision in call
cp ~/kaiserflu/runFileSaveLogs ~/kaiserflu/3-figures/

# Run folder scripts and produce output
cd ~/kaiserflu/3-figures/
./runFileSaveLogs -i "figures" \
fig-mean-season-age.R \
fig-monthly-rate.R \
fig-point-estimates-combined.R \
fig-point-estimates.R \
fig-weekly-rate.R

# Remove copied utility run script
rm runFileSaveLogs
```

Chapter 12

Slurm and cluster computing

by Anna Nguyen, and Jade Benjamin-Chung

When you need to run a script that requires a large amount of RAM, large files, or that uses parallelization, you can use Sherlock, Stanford's computing cluster. Sherlock uses Slurm, an open source, scalable cluster management and job scheduling system for computing clusters. Jade can email Sherlock managers to get you an account. Please refer to the Sherlock user guide to learn about the system and how to use it. Below, we include a few tips specific to how we use Sherlock in our lab.

12.1 Getting started

To access Sherlock, in terminal, log in using the following syntax and replace "USERNAME" with your Stanford alias. You will be prompted to enter your Stanford password (the same one you use for your email and other accounts) and to complete two-factor authentication.

```
ssh USERNAME@login.sherlock.stanford.edu
```

Once you log in, you can view the contents of your home directory in command line by entering `cd $HOME`. You can create subfolders within this directory using the `mkdir` command. For example, you could make a "code" subdirectory and clone a Github repository there using the following code:

```
cd $HOME
mkdir code
git clone https://github.com/jadebc/covid19-infections.git
```

12.2 Moving files to Sherlock

The `$HOME` directory is a good place to store code and small test files (quota: 15 GB per user). Save large files to the `$SCRATCH` directory (quota: 100 TB per user). On the `$SCRATCH` directory, files that are not modified after 90 days are automatically deleted. For this reason, it's best to create a bash script that records the file transfer process for a given project. See example code below:

```
# securely transfer folders from Box to sherlock home directory
# note: the -r option is for folders and is not needed for files
scp -r "Box/malaria-project/folder-1/" USERNAME@login.sherlock.stanford.edu:
scp -r "Box/malaria-project/folder-2/" USERNAME@login.sherlock.stanford.edu:

# log into sherlock
ssh USERNAME@login.sherlock.stanford.edu

# make a malaria-project subfolder in scratch directory
cd $SCRATCH
mkdir malaria-project

# move files from sherlock home directory to scratch directory
cd $HOME
mv folder-1 $SCRATCH/malaria-project/
mv folder-2 $SCRATCH/malaria-project/
```

12.3 Testing your code

Typically, you will want to initially test your scripts by initiating a development node using the command `sdev`. This will allocate a small amount of computing resources for 1 hour. You can access R via command line using the following code, or you can test your code in the Rstudio server via Sherlock.

```
# start development node
sdev

# load R - default version (see Sherlock documentation for which version)
module load R

# initiate R in command line
R
```

In most cases, you will want to test that the file paths work correctly on Sherlock. You will likely need to add code to the configuration file in the project repository that specifies Sherlock-specific file paths. Here is an example:

```
# set sherlock-specific file paths
if(Sys.getenv("LMOD_SYSHOST")=="sherlock"){
```

```
  sherlock_path = paste0(Sys.getenv("HOME"), "/malaria-project/")

  data_path = paste0(sherlock_path, "data/")
  results_path = paste0(sherlock_path, "results/")
}
```

12.4 Running big jobs

Once your test scripts run successfully, you can submit an sbatch script for larger jobs. These are text files with a `.sh` suffix. Use a text editor like Sublime to create such a script. Documentation on sbatch options is available here. Here is an example of an sbatch script with the following options:

- `job-name=run_inc`: Job name that will show up in the Sherlock system
- `begin=now`: Requests to start the job as soon as the requested resources are available
- `dependency=singleton`: Jobs can begin after all previously launched jobs with the same name and user have ended.
- `mail-type=ALL`: Receive all types of email notification (e.g., when job starts, fails, ends)
- `cpus-per-task=16`: Request 16 processors per task. The default is one processor per task.
- `mem=64G`: Request 64 GB memory per node.
- `output=00-run_inc_log.out`: Create a log file called `00-run_inc_log.out` that contains information about the Slurm session

The file `analysis.out` will contain the log file for the R script `analysis.R`.

```
#!/bin/bash

#SBATCH --job-name=run_inc
#SBATCH --begin=now
#SBATCH --dependency=singleton
#SBATCH --mail-type=ALL
#SBATCH --cpus-per-task=16
#SBATCH --mem=64G
#SBATCH --mem=64G
#SBATCH --output=00-run_inc_log.out

cd $HOME/malaria-code-repo/2-analysis/

module purge

# load R version 4.0.2 (required for certain packages)
module load R/4.0.2
```

```
# load software required for spatial analyses in R
module load physics gdal
module load physics proj

R CMD BATCH --no-save analysis.R analysis.out
```

To submit this job, save the code in the chunk above in a script called `myjob.sh` and then enter the following command into terminal:

```
sbatch myjob.sh
```

To check on the status of your job, enter the following code into terminal:

```
squeue -u $USERNAME
```

Chapter 13

Checklists

by Jade Benjamin-Chung

13.1 Pre-analysis plan checklist

- Brief background on the study (a condensed version of the introduction section of the paper)
- Hypotheses / objectives
- Study design
- Description of data
- Definition of outcomes
- Definition of interventions / exposures
- Definition of covariates
- Statistical power calculation
- Statistical model description
- Covariate selection / screening
- Standard error estimation method
- Missing data analysis
- Assessment of effect modification / subgroup analyses
- Sensitivity analyses
- Negative control analyses

13.2 Code checklist

- Does the script run without errors?
- Is code self-contained within repo and/or associated Box folder?
- Is all commented out code / remarks removed?
- Does the header accurately describe the process completed in the script?
- Is the script pushed to its github repository?

- Does the code adhere to the coding style guide?
- Are all warnings ignorable? Should any warnings be intentionally suppressed or addressed?

13.3 Manuscript checklist

This is adapted in part from this article.

- Have you completed the relevant reporting checklist, if applicable? (Collection of checklists)
- Are the study results within the manuscript replicable (i.e., if you rerun the code in the study's repository, the tables and figures will be exactly replicated?)
- Is a target journal selected?
- Is the title declarative, in other words, does it state the object/findings rather than suggest them?
- Is the word count of the manuscript close to the target journal's allowance?
- Does the manuscript adhere to the formatting guide of the target journal?
- Does the manuscript use a consistent voice (passive or active – usually active is preferred … pun intended)?
- Is each figure and table (including supplementary material) referenced in the main text?
- Is there a caption for each figure and table (including supplementary material)?
- Are tables/figures and supplementary material numbered in accordance with their appearance in the main text?
- Does the text use past tense if it is reporting research findings or future tense if it is a study protocol?
- Does the text avoid subjective wording (e.g., "interesting", "dramatic")?
- Does the text use minimal abbreviations, and are all abbreviations defined at first use?
- Does the text avoid directionless words? (e.g., instead of writing, 'Precipitation influences disease risk', write, 'Precipitation was associated with increased disease risk').
- Does the text avoid making causal claims that are not supported by the study design? Be careful about the words "effect", "increase", and "decrease", which are often interpreted as causal.
- Does the text avoid describing results with the word "significant", which can easily be confused with statistical significance? (see references on this topic here)
- Have you drafted author contributions? Do they follow the CRediT Taxonomy for author contributions?

13.4 Figure checklist

- Are the x-axis and y-axis labeled?
- If the figure includes panels, is each panel labeled?
- Are there sufficient numerical / text labels and breaks on the x-axis and y-axis?
- Is the font size appropriate (i.e., large enough to read, not so large that it distracts from the data presented in the figure?)
- Are the colors used colorblind friendly? See a colorblind-friendly palette here, a neat palette generator with colorblind options here, and an article on why this matters here
- Are colors/shapes/line types defined in a legend?
- Are the legends and other labels easy to understand with minimal abbreviations?
- If there is overplotting, is transparency used to show overlapping data?
- Are 95% confidence intervals or other measures of precision shown, if applicable?

Chapter 14

Resources

by Jade Benjamin-Chung and Kunal Mishra

14.1 Resources for R

- dplyr and tidyr cheat sheet
- ggplot cheat sheet
- data table cheat sheet
- RMarkdown cheat sheet
- Hadley Wickham's R Style Guide
- Jade's R-for-epi course
- Tidy Eval in 5 Minutes (video)
- Tidy Evaluation (e-book)
- Data Frame Columns as Arguments to Dplyr Functions (blog)
- Standard Evaluation for *_join (stackoverflow)
- Programming with dplyr (package vignette)

14.2 Resources for Git & Github

- Data Camp introduction to Git
- Introduction to Github

14.3 Scientific figures

- Ten Simple Rules for Better Figures

14.4 Writing

- Tips on how to write a great science paper
- ICMJE Definition of authorship
- Nature article on elements of style for scientific writing