UCD-SeRG Lab Manual

Kristen Aiemjoy Ezra Morrison

Last updated: 2026-01-12

Contents

1 Welcome to UCD-SeRG!

1.1 About the lab

Welcome to the Seroepidemiology Research Group (SeRG) at the University of California, Davis, led by Drs. Kristen Aiemjoy and Ezra Morrison. Accurate methods to measure infectious disease burden are essential for guiding public health decisions, yet many infectious diseases remain under-recognized due to limited diagnostics and costly, resource-intensive surveillance systems. Our work addresses this gap by developing seroepidemiologic methods to characterize infection burden in populations. Currently, we focus on enteric fever (Salmonella Typhi and Paratyphi), Scrub Typhus (Orientia tsutsugamushi), Melioidosis (Burkholderia pseudomallei), Shigella (Shigella spp.), and Cholera (Vibrio cholerae). We are supported by the US National Institutes of Health, the Bill and Melinda Gates Foundation, and the Department of Defense, and collaborate with partners around the world. To learn more about the lab, visit ucdserg.ucdavis.edu[1].

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 is a fork of the Benjamin-Chung Lab's manual[2], adapted for UCD-SeRG. We are grateful to Dr. Jade Benjamin-Chung and her team for developing and openly sharing their excellent lab manual. You can view the original manual at jadebc.github.io/lab-manual[3]. Original contributors include Jade Benjamin-Chung, Kunal Mishra, Stephanie Djajadi, Nolan Pokpongkiat, Anna Nguyen, Iris Tong, and Gabby Barratt Heitmann.

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

[1] https://ucdserg.ucdavis.edu
[2] https://github.com/jadebc/lab-manual
[3] https://jadebc.github.io/lab-manual/index.html

2 Culture and conduct

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung[1]

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

UCD-SeRG 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[2] training and share their certificate with the lab leadership. Team members will be added 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 lab leadership first).

[1] https://jadebc.github.io/lab-manual/culture-and-conduct.html
[2] https://research.ucdavis.edu/policiescompliance/irb-admin/education/

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[3] and are happy for team members who meet the definition of authorship to be included as co-authors on scientific manuscripts.

[3] http://www.icmje.org/recommendations/browse/roles-and-responsibilities/defining-the-role-of-authors-and-contributors.html

3 Communication and coordination

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung[1]

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

3.1 Microsoft Teams

- Use Microsoft Teams for scheduling, coding related questions, quick check ins, etc. If your Teams 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 mention you (e.g., `@username`) 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 Teams (e.g., it could say "On vacation") so we know not to expect to see you online.
- Please thread messages in Teams 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 Task Management

We use a combination of tools to track and manage project tasks:

- **GitHub Issues and Projects**: For code-related tasks, feature requests, and bug tracking. Lab leadership will assign issues and organize them in GitHub Projects. Issues are prioritized within projects, and you can track your assigned tasks there.
- **Microsoft To-Do** and other M365 task tracking tools: For general lab tasks and personal task management. Lab leadership may assign tasks through these tools, which integrate with Microsoft Teams.
- Generally, strive to complete assigned tasks by the date listed.

[1] https://jadebc.github.io/lab-manual/communication-and-coordination.html

- Use checklists to break down tasks into smaller chunks. Sometimes leadership will create these for you, but you can also add them yourself.
- Update task status as you make progress so the team can stay coordinated.

3.4 Google Drive

- We mostly use Google Drive to create shared documents with longer descriptions of tasks. These documents may be linked to in GitHub Issues or other task tracking tools. Lab leadership 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 UC Davis Box and SharePoint

- Human subjects data for research studies are generally stored in UC Davis Box or SharePoint. Please check with lab leadership about whether there are special storage and transfer requirements for the datasets you are working with for each study.
- You can access Box via your UC Davis credentials. For more information, visit UC Davis Box Support[2].
- SharePoint is also used for collaborative document storage and team file sharing. Access SharePoint through your UC Davis Microsoft 365 account.

3.6 Meetings

- Our meetings start on the hour.
- If you are going to be late, please send a message in our Teams 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.

3.7 Code Review

When submitting code to or reviewing code from colleagues, use best practices to provide and receive constructive feedback:

- Tidyverse code review guide[3]: Best practices for reviewing R code, including what to look for and how to provide constructive feedback.

[2] https://kb.ucdavis.edu/?id=6485
[3] https://code-review.tidyverse.org/

4 Reproducibility

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung[1]

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[2]
- 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

[1] https://jadebc.github.io/lab-manual/reproducibility.html
[2]

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/)[3]

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].

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[5], share links to Github[6] repositories, and archive code on Zenodo[7].

[3] https://www.equator-network.org/about-us/what-is-a-reporting-guideline/
[4] https://www.medrxiv.org/
[5] osf.io
[6] github.com
[7] zenodo.org

5 Code repositories

Adapted by UCD-SeRG team from original by Kunal Mishra, Jade Benjamin-Chung, and Stephanie Djajadi[1]

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.

UCD-SeRG projects use R package structure for most R-based work. This provides benefits for reproducibility, collaboration, and code quality even for analysis-only projects.

5.1 Package Structure

All R projects in our lab should be structured as R packages, even if they are primarily analysis projects and not intended for distribution on CRAN or Bioconductor. This standardized structure provides numerous benefits:

5.1.1 Why Use R Package Structure?

1. **Organized code**: Clear separation of functions (`R/`), documentation (`man/`), tests (`tests/`), data (`data/`), and vignettes/analyses
2. **Dependency management**: `DESCRIPTION` file explicitly declares all package dependencies and version restrictions, which simplifies installing those dependencies.
3. **Automatic documentation**: `roxygen2` generates help files from inline comments
4. **Built-in testing**: `testthat` framework integrates seamlessly with package structure
5. **Code quality**: Tools like `devtools::check()` and `lintr` enforce best practices
6. **Reproducibility**: Package structure makes it easy to share and reproduce analyses
7. **Reusable functions**: Decompose complex analyses into well-documented, testable functions
8. **Version control**: Track changes to code, documentation, and data together

5.1.2 Basic Package Structure

```
myproject/
  DESCRIPTION          # Package metadata and dependencies
  NAMESPACE            # Auto-generated, don't edit manually
  R/                   # All R functions (reusable code)
     analysis_functions.R
     data_prep.R
     plotting.R
```

[1] https://jadebc.github.io/lab-manual/code-repositories.html

```
man/                 # Auto-generated documentation
tests/
    testthat/       # Unit tests
data/                # Processed data objects (.rda files)
data-raw/            # Raw data and data processing scripts
    0-prep-data.sh  # Shell scripts for data preparation
    process_survey_data.R
    clean_lab_results.R
vignettes/           # Long-form documentation
    intro.qmd       # Main vignettes (shipped with package)
    tutorial.qmd
    articles/       # Website-only articles (not shipped)
        advanced-topics.qmd
        case-studies.qmd
inst/                # Additional files to include in package
    extdata/        # External data files and .RDS results
        analysis_results.rds
        processed_data.rds
    output/         # Figure and table outputs
        figures/
            fig1.pdf
            fig2.png
        tables/
            table1.csv
            table2.xlsx
    analyses/       # Analyses using restricted data (see below)
.Rproj               # RStudio project file
```

5.1.3 Where to Place Analysis Files

5.1.3.1 Vignettes vs Articles

Vignettes (`vignettes/*.qmd`): - **Shipped with the package** when installed - Accessible via `vignette()` and `browseVignettes()` in R - Displayed on CRAN - Built at package build time - Use for core package documentation and tutorials - Created with `usethis::use_vignette("name")`

Articles (`vignettes/articles/*.qmd`): - **Website-only** (not shipped with the package) - Only appear on the pkgdown website - Not accessible via `vignette()` in R - Not displayed on CRAN - Use for supplementary content, blog posts, extended tutorials, or frequently updated material - Created with `usethis::use_article("name")` - Automatically added to `.Rbuildignore`

When to use each: - **Vignette**: Essential tutorials users need offline, core package workflows - **Article**: Supplementary material, case studies, advanced topics, blog-style content

5.1.3.2 Public Analyses (vignettes/)

Use `vignettes/` for analysis workbooks that:

- Use publicly available data
- Should be accessible to all package users
- Are core to understanding the package

Use `vignettes/articles/` for:

- Extended case studies
- Blog-style posts
- Supplementary analyses
- Material that updates frequently

All vignettes and articles will be rendered by `pkgdown::build_site()` on your package website.

5.1.3.3 Analyses with Restricted Data (inst/analyses/)

For analyses that rely on **private, sensitive, or restricted data**, place `.qmd` or `.qmd` files in `inst/analyses/`:

```
myproject/
  inst/
     analyses/
        01-confidential-data-analysis.qmd
        02-unpublished-results.qmd
        README.md  # Document data access requirements
     extdata/
  vignettes/
      01-public-analysis.qmd
      02-demo-with-simulated-data.qmd
```

Benefits of this approach:

- Analyses with restricted data are included in version control alongside your code
- They're clearly separated from public documentation
- `inst/analyses/` is **excluded from `pkgdown` builds** and package documentation
- Collaborators with data access can still run these analyses
- You maintain a complete record of all project work

Note on privacy: Files in `inst/analyses/` are not inherently private—they will be visible if your repository is public. Use this folder for analyses that rely on restricted data that is stored separately, not for storing the restricted data itself. If you need to keep the analysis code private, use a private repository.

Best practices for analyses with restricted data:

1. **Document data requirements**: Include a README.md in `inst/analyses/` explaining:
    - What data is required
    - Where to obtain it (if permissible)
    - Data access restrictions
    - How to set up data paths

2. **Use relative paths carefully**: Structure your code so data paths can be configured:

```
# In inst/analyses/01-analysis.qmd
# Users should set this based on their local setup
data_dir <- Sys.getenv("MYPROJECT_DATA",
                       default = "~/restricted_data/myproject")
raw_data <- readr::read_csv(file.path(data_dir, "sensitive.csv"))
```

3. **Create public alternatives**: When possible, create companion vignettes in `vignettes/` using:
   - Simulated data that mimics the structure
   - Publicly available datasets
   - Aggregated/de-identified summaries
4. **Add to .Rbuildignore**: Ensure `inst/analyses/` doesn't cause package checks to fail:

```
# Use usethis to add to .Rbuildignore
usethis::use_build_ignore("inst/analyses")
```

5.1.4 Keep Analysis Workbooks Tidy

Decompose reusable functions from your analysis notebooks into the `R/` directory. Your vignettes should:

- Be clean, readable narratives of your analysis
- Call well-documented functions from your package
- Focus on the "what" and "why" rather than implementation details
- Be reproducible by others with a single click (or with documented data access for private analyses)

Example of what NOT to do (all code in vignette):

```
# Bad: 100 lines of data manipulation in vignette
raw_data <- read_csv("data.csv")
# ... 100 lines of cleaning, transforming, reshaping ...
cleaned_data <- final_result
```

Example of what TO do (functions in R/, simple calls in vignette):

```
# Good: Clean vignette calling documented functions
raw_data <- read_csv("data.csv")
cleaned_data <- prep_study_data(raw_data)  # Function in R/data_prep.R
```

5.1.5 Shell Scripts and Automation

Shell scripts are useful for automating workflows and ensuring reproducibility. Place shell scripts in `data-raw/` alongside the R scripts they coordinate:

```
data-raw/
   0-prep-data.sh          # Shell script to run all data prep
   01-load-survey.R
   02-clean-survey.R
   03-merge-datasets.R
   04-create-analysis-data.R
```

Using shell scripts:

```
# data-raw/0-prep-data.sh
#!/bin/bash
Rscript data-raw/01-load-survey.R
Rscript data-raw/02-clean-survey.R
Rscript data-raw/03-merge-datasets.R
Rscript data-raw/04-create-analysis-data.R
```

This is especially useful when data upstream changes — you can simply run the shell script to reproduce everything. After running shell scripts, `.Rout` log files will be generated for each script. It is important to check these files to ensure everything has run correctly.

5.1.6 Storing Analysis Outputs

Results files (.RDS): Save analysis results in `inst/extdata/`:

```
# Save results
readr::write_rds(analysis_results, here("inst", "extdata", "analysis_results.rds"))

# Load results later
results <- readr::read_rds(here("inst", "extdata", "analysis_results.rds"))
```

Figures and tables: Save publication outputs in `inst/output/`:

```
# Save figure
ggsave(here("inst", "output", "figures", "fig1_incidence_trends.pdf"),
       width = 8, height = 6)

# Save table
readr::write_csv(summary_table,
                 here("inst", "output", "tables", "table1_demographics.csv"))
```

Organization:

```
inst/
   extdata/
       analysis_results.rds
       model_fits.rds
       processed_data.rds
   output/
       figures/
           fig1_incidence_trends.pdf
           fig2_risk_factors.png
           figS1_sensitivity.pdf
       tables/
            table1_demographics.csv
            table2_main_results.xlsx
            tableS1_detailed_results.csv
```

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 Organizing the `data-raw` folder

The `data-raw` folder serves as a catch-all for scripts that do not (yet) fit into the package structure described above. The `data-raw` folder should still be organized. We recommend the following subdirectory structure for `data-raw`:

```
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
```

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

5.3.1 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.

5.3.2 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.

5.3.3 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 Chapter 11 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.

6 R Coding Practices

Adapted by UCD-SeRG team from original by Kunal Mishra, Jade Benjamin-Chung, Stephanie Djajadi, and Iris Tong[1]

6.1 Lab Protocols for Code and Data

Just as wet labs have strict safety protocols to ensure reproducible results and prevent contamination, our computational lab has protocols for coding and data management. These protocols are not suggestions—they are essential practices that:

- **Ensure reproducibility**: Others (including your future self) can recreate your analysis
- **Prevent errors**: Systematic approaches reduce the risk of mistakes
- **Enable collaboration**: Consistent practices allow team members to work together efficiently
- **Maintain data integrity**: Proper handling prevents data corruption and loss
- **Support publication**: Well-documented, reproducible code is increasingly required for publication

Violating these protocols can have serious consequences, including invalid results, wasted time, inability to publish, and damage to scientific credibility. Treat coding and data management protocols with the same seriousness as you would safety protocols in a wet lab.

6.2 Essential R Package Development Tools

The following tools are essential for R package development in our lab:

6.2.1 usethis: Package Setup and Management

`usethis` automates common package development tasks:

```
# Install usethis
install.packages("usethis")

# Create a new package
usethis::create_package("~/myproject")

# Add common components
usethis::use_mit_license()          # Add a license
```

[1] https://jadebc.github.io/lab-manual/coding-practices.html

```r
usethis::use_git()                  # Initialize git
usethis::use_github()               # Connect to GitHub
usethis::use_testthat()             # Set up testing infrastructure
usethis::use_vignette("intro")      # Create a vignette (shipped with package)
usethis::use_article("case-study")  # Create an article (website-only)
usethis::use_data_raw("dataset")    # Create data processing script
usethis::use_package("dplyr")       # Add a dependency
usethis::use_pipe()                 # Import magrittr pipe operator (no longer recommended)

# Increment version
usethis::use_version()              # Increment package version
```

6.2.2 devtools: Development Workflow

`devtools` provides the core development workflow:

```r
# Install devtools
install.packages("devtools")

# Load your package for interactive development
devtools::load_all()                # Like library(), but for development

# Documentation
devtools::document()                # Generate documentation from roxygen2

# Testing
devtools::test()                    # Run all tests
devtools::test_active_file()        # Run tests in current file

# Checking
devtools::check()                   # R CMD check (comprehensive validation)
devtools::check_man()               # Check documentation only

# Dependencies
devtools::install_dev_deps()        # Install all development dependencies

# Building
devtools::build()                   # Build package bundle
devtools::install()                 # Install package locally
```

6.2.3 pkgdown: Package Websites

`pkgdown` builds beautiful documentation websites from your package:

```r
# Install pkgdown
install.packages("pkgdown")

# Set up pkgdown
```

```r
usethis::use_pkgdown()

# Build website locally
pkgdown::build_site()

# Preview in browser
pkgdown::build_site(preview = TRUE)

# Build components separately
pkgdown::build_reference()          # Function reference
pkgdown::build_articles()           # Vignettes
pkgdown::build_home()               # Home page from README
```

Configure your pkgdown site with `_pkgdown.yml`:

```yaml
url: https://ucd-serg.github.io/myproject

template:
  bootstrap: 5

reference:
  - title: "Data Preparation"
    desc: "Functions for preparing and cleaning data"
    contents:
    - prep_study_data
    - validate_data

  - title: "Analysis"
    desc: "Core analysis functions"
    contents:
    - run_primary_analysis
    - sensitivity_analysis

articles:
  - title: "Analysis Workflow"
    navbar: Analysis
    contents:
    - 01-data-preparation
    - 02-primary-analysis
    - 03-sensitivity-analysis
```

6.3 Complete Package Development Workflow

Here's the typical workflow for developing an R package in our lab:

6.3.1 1. Initial Setup

```
# Create package structure
usethis::create_package("~/myproject")

# Set up infrastructure
usethis::use_git()
usethis::use_github()
usethis::use_testthat()
usethis::use_pkgdown()
usethis::use_mit_license()
usethis::use_readme_rmd()
```

6.3.2 2. Add Dependencies

```
# Add packages your project depends on
usethis::use_package("dplyr")
usethis::use_package("ggplot2")
usethis::use_package("readr")

# Add packages only needed for development/testing
usethis::use_package("testthat", type = "Suggests")
```

6.3.3 3. Write Functions

Create functions in `R/` directory with roxygen2 documentation:

```
#' Prepare Study Data
#'
#' Clean and prepare raw study data for analysis.
#'
#' @param raw_data A data frame containing raw study data
#' @param validate Logical; whether to run validation checks
#'
#' @returns A cleaned data frame ready for analysis
#'
#' @examples
#' raw_data <- read_csv("data.csv")
#' clean_data <- prep_study_data(raw_data)
#'
#' @export
prep_study_data <- function(raw_data, validate = TRUE) {
  # Function implementation
}
```

6.3.4 4. Document

```
# Generate documentation from roxygen2 comments
devtools::document()
```

6.3.5 5. Test

Create tests in `tests/testthat/`:

```
# tests/testthat/test-data_prep.R
test_that("prep_study_data handles missing values", {
  raw_data <- data.frame(x = c(1, NA, 3))
  result <- prep_study_data(raw_data)
  expect_false(anyNA(result$x))
})
```

Run tests:

```
devtools::test()
```

6.3.6 6. Check

```
# Comprehensive package check
devtools::check()
```

Fix any warnings or errors before proceeding.

6.3.7 7. Build Documentation Site

```
pkgdown::build_site()
```

6.3.8 8. Share and Publish

```
# Push to GitHub
# The pkgdown site can be automatically deployed to GitHub Pages
# using GitHub Actions
```

Organizing scripts

```
Just as your data "flows" through your project, data should flow naturally through a script

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](https:/
Absenteeism/blob/master/Master's%20Thesis%20-%20Spatial%20Epidemiology%20of%20Influenza/2a%
%20Statistical-Inputs.R) for a good example of how to cleanly organize a file in a way that



## Testing Requirements {#sec-r-testing}

**ALWAYS establish tests BEFORE modifying functions.** This ensures changes preserve existi

### When to Use Snapshot Tests

Use snapshot tests (`expect_snapshot()`, `expect_snapshot_value()`) when:

- Testing complex data structures (data frames, lists, model outputs)
- Validating statistical results where exact values may vary slightly
- Output format stability is important

```r
test_that("prep_study_data produces expected structure", {
  result <- prep_study_data(raw_data)
  expect_snapshot_value(result, style = "serialize")
})
```
```

6.3.9 When to Use Explicit Value Tests

Use explicit tests (`expect_equal()`, `expect_identical()`) when:

- Testing simple scalar outputs
- Validating specific numeric thresholds
- Testing Boolean returns or categorical outputs

```r
test_that("calculate_mean returns correct value", {
  expect_equal(calculate_mean(c(1, 2, 3)), 2)
  expect_equal(calculate_ratio(3, 7), 0.4285714, tolerance = 1e-6)
})
```

6.3.10 Testing Best Practices

- **Seed randomness**: Use `withr::local_seed()` for reproducible tests
- **Use small test cases**: Keep tests fast
- **Test edge cases**: Missing values, empty inputs, boundary conditions

- **Test errors**: Verify functions fail appropriately with invalid input

```
test_that("prep_study_data handles edge cases", {
  # Empty input
  expect_error(prep_study_data(data.frame()))

  # Missing required columns
  expect_error(prep_study_data(data.frame(x = 1)))

  # Valid input with missing values
  result <- prep_study_data(data.frame(id = 1:3, value = c(1, NA, 3)))
  expect_true(all(!is.na(result$value)))
})
```

6.4 Iterative Operations

When applying analyses with different variations (outcomes, exposures, subgroups), use functional programming approaches:

6.4.1 `lapply()` and `sapply()`

```
# Apply function to each element
results <- lapply(outcomes, function(y) {
  run_analysis(data, outcome = y)
})

# Simplify to vector if possible
summary_stats <- sapply(data_list, mean)
```

6.4.2 `purrr::map()` Family

The `purrr` package provides type-stable alternatives:

```
library(purrr)

# Always returns a list
results <- map(outcomes, ~ run_analysis(data, outcome = .x))

# Type-specific variants
means <- map_dbl(data_list, mean)        # Returns numeric vector
models <- map(splits, ~ lm(y ~ x, data = .x))  # Returns list of models
```

6.4.3 `purrr::pmap()` for Multiple Arguments

When iterating over multiple parameter lists:

```
params <- tibble(
  outcome = c("outcome1", "outcome2", "outcome3"),
  exposure = c("exp1", "exp2", "exp3"),
  covariate_set = list(c("age", "sex"), c("age"), c("age", "sex", "bmi"))
)

results <- pmap(params, function(outcome, exposure, covariate_set) {
  run_analysis(
    data = study_data,
    outcome = outcome,
    exposure = exposure,
    covariates = covariate_set
  )
})
```

6.4.4 Parallel Processing

For computationally intensive work, use `future` and `furrr`:

```
library(future)
library(furrr)

# Set up parallel processing
plan(multisession, workers = availableCores() - 1)

# Parallel version of map()
results <- future_map(large_list, time_consuming_function, .progress = TRUE)
```

6.5 Reading and Saving Data

6.5.1 RDS Files (Preferred)

Use RDS format for R objects:

```
# Save single object
readr::write_rds(analysis_results, here("results", "analysis.rds"))

# Read back
results <- readr::read_rds(here("results", "analysis.rds"))
```

Avoid `.RData` files because: - You can't control object names when loading - Can't load individual objects - Creates confusion in older code

6.5.2 CSV Files

For tabular data that may be shared with non-R users:

```
# Write
readr::write_csv(data, here("data-raw", "clean_data.csv"))

# Read
data <- readr::read_csv(here("data-raw", "clean_data.csv"))

# For very large files, use data.table
data.table::fwrite(large_data, "big_file.csv")
data <- data.table::fread("big_file.csv")
```

6.6 Version Control and Collaboration

6.6.1 Version Numbers

Follow semantic versioning (MAJOR.MINOR.PATCH):

- Development versions: `0.0.0.9000`, `0.0.0.9001`, etc.
- First release: `0.1.0`
- Bug fixes: increment PATCH (e.g., `0.1.0` → `0.1.1`)
- New features: increment MINOR (e.g., `0.1.1` → `0.2.0`)
- Breaking changes: increment MAJOR (e.g., `0.2.0` → `1.0.0`)

```
# Increment version
usethis::use_version()
```

6.6.2 NEWS File

Document all user-facing changes in `NEWS.md`:

````
# myproject 0.2.0


## Continuous Integration {#sec-r-ci}

Set up automated checks using GitHub Actions:

```r
# Add standard R package CI workflows
usethis::use_github_action("check-standard")    # R CMD check
usethis::use_github_action("test-coverage")     # Code coverage
usethis::use_github_action("pkgdown")           # Deploy website
````

These workflows automatically:

- Run R CMD check on multiple platforms (Linux, macOS, Windows)
````

- Calculate test coverage
- Build and deploy your pkgdown website
- Ensure code quality before merging

6.7 Quality Assurance Checklist

Before submitting a pull request or finalizing analysis, verify:

- ☐ All functions have complete roxygen2 documentation
- ☐ All functions have corresponding tests
- ☐ `devtools::document()` has been run
- ☐ `devtools::test()` passes with no failures
- ☐ `devtools::check()` passes with no errors, warnings, or notes
- ☐ `lintr::lint_package()` shows no issues (or only acceptable ones)
- ☐ `spelling::spell_check_package()` passes
- ☐ Version number has been incremented
- ☐ `NEWS.md` has been updated with changes
- ☐ `README.Rmd` has been updated (if needed) and `README.md` regenerated
- ☐ `pkgdown::build_site()` builds successfully
- ☐ All changes committed and pushed to GitHub

6.8 Automated Code Styling

6.8.1 RStudio Built-in Formatting

Use RStudio's built-in autoformatter (keyboard shortcut: `CMD-Shift-A` or `Ctrl-Shift-A`) to quickly format highlighted code.

6.8.2 styler Package

For automated styling of entire projects:

```
# Install styler
install.packages("styler")

# Style all files in R/ directory
styler::style_dir("R/")

# Style entire package
styler::style_pkg()

# Note: styler modifies files in-place
# Always use with version control so you can review changes
```

6.8.3 lintr Package

For checking code style without modifying files:

```
# Install lintr
install.packages("lintr")

# Lint the entire package
lintr::lint_package()

# Lint a specific file
lintr::lint("R/my_function.R")
```

The linter checks for:

- Unused variables
- Improper whitespace
- Line length issues
- Style guide violations

You can customize linting rules by creating a `.lintr` file in your project root.

See also Section 7.11.

6.9 Documenting your code

6.9.1 Function headers

Every function you write must include documentation to describe its purpose, inputs, and outputs. For any reproducible workflows, this is 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.

We use `{roxygen2}` (Wickham et al. 2024) for function documentation. Roxygen2 allows you to describe your functions in special comments next to their definitions, and automatically generates R documentation files (`.Rd` files) and helps manage your package `NAMESPACE`. The roxygen2 format uses `#'` comments placed immediately before the function definition.

Here is an example of documenting a function using roxygen2:

```
#' Calculate flu season means by site
#'
#' Make a dataframe with rows for flu season and site
#' containing the number of patients with an outcome, the total patients,
#' and the percent of patients with the outcome.
#'
#' @param data A data frame with variables flu_season, site, studyID, and yname
#' @param yname A string for the outcome name
#' @param silent A boolean specifying whether to suppress console output
#'   (default: TRUE)
```

```
#'
#' @returns A dataframe as described above
#'
#' @examples
#' calc_fluseas_mean(my_data, "hospitalized", silent = FALSE)
#'
calc_fluseas_mean <- function(data, yname, silent = TRUE) {
  ### function code here

}
```

The roxygen2 header tells you what the function does, its various inputs, and how you might use it. Also notice that all optional arguments (i.e. ones with pre-specified defaults) follow arguments that require user input.

For more information on roxygen2 syntax and features, see https://roxygen2.r-lib.org/.

> **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).

6.9.2 Script headers

Every file in a project that doesn't have roxygen function documentation should at least 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 [...]
##############################################################################
```

6.9.3 Sections and subsections

Rstudio (v1.4 or more recent[2]) 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 -------
```

6.9.4 Code folding

Consider using RStudio's code folding[3] 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:

6.9.5 Comments in the body of your code

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[4] 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
```

See also Section 7.2 for function documentation style guidelines.

6.10 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[5]

We recommend you use **snake_case**.

[2] https://blog.rstudio.com/2020/12/02/rstudio-v1-4-preview-little-things/
[3] https://support.rstudio.com/hc/en-us/articles/200484568-Code-Folding-and-Sections
[4] https://github.com/kmishra9/Flu-Absenteeism/blob/master/Master's%20Thesis%20-%20Spatial%20Epidemiology%20of%20Influenza/1b%20-%20Map-Management.R
[5] https://spin.atomicobject.com/2017/11/01/good-variable-names/

- 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, it's 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 it's worth getting rid of these bad habits now.

See also Section 7.10.

6.11 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
)
```

6.12 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].

See also Section 7.9 for code style guidelines on using the here package.

6.13 Reading/Saving Data

6.13.1 `.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.

6.13.2 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.

6.14 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] https://github.com/jennybc/here_here

6.14.1 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[7] 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[8].

6.14.2 Dropbox

Follow the instructions at this link[9] 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")
```

6.15 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[10] 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[11], which leans heavily on many Tidyverse packages and may be worth checking out.

[7] https://r-box.github.io/boxr/articles/boxr-app-interactive.html#create
[8] https://github.com/r-box/boxr
[9] https://github.com/karthik/rdrop2
[10] https://rviews.rstudio.com/2017/06/08/what-is-the-tidyverse/
[11] https://r4ds.had.co.nz/

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 `tibble::tribble()`
`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)`
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()`
—	—
`sessionInfo()`	`sessioninfo::session_info()`

For a more extensive set of syntactical translations to Tidyverse, you can check out this document[12].

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:

[12] https://tavareshugo.github.io/data_carpentry_extras/base-r_tidyverse_equivalents/base-r_tidyverse_equivalents.html#reshaping_data

- Tidy Eval in 5 Minutes[13] (video)
- Tidy Evaluation[14] (e-book)
- Data Frame Columns as Arguments to Dplyr Functions[15] (blog)
- Standard Evaluation for *_join[16] (stackoverflow)
- Programming with dplyr[17] (package vignette)

See also Section 7.8

6.16 Coding with R and Python

If you're using both R and Python, you may wish to check out the Feather package[18] for exchanging data between the two languages extremely quickly[19].

6.17 Repeating analyses with different variations

In many cases, we will need to apply our modeling on different combinations of interests (outcomes, exposures, etc.). We can certainly use a `for` loop to repeat the execution of a wrapper function, but generally, `for` loops request high memory usage and produce the results in long computation time.

Fortunately, R has some functions which implement looping in a compact form to help repeating your analyses with different variations (subgroups, outcomes, covariate sets, etc.) with better performances.

6.17.1 `lapply()` and `sapply()`

`lapply()` is a function in the base R package that applies a function to each element of a list and returns a list. It's typically faster than `for`. Here is a simple generic example:

```
result <- lapply(X = mylist, FUN = func)
```

There is another very similar function called `sapply()`. It also takes a list as its input, but if the output of the `func` is of the same length for each element in the input list, then `sapply()` will simplify the output to the simplest data structure possible, which will usually be a vector.

[13] https://www.youtube.com/watch?v=nERXS3ssntw
[14] https://tidyeval.tidyverse.org/index.html
[15] https://www.brodrigues.co/blog/2016-07-18-data-frame-columns-as-arguments-to-dplyr-functions/
[16] https://stackoverflow.com/questions/28125816/r-standard-evaluation-for-join-dplyr
[17] https://dplyr.tidyverse.org/articles/programming.html
[18] https://www.rdocumentation.org/packages/feather/versions/0.3.3
[19] https://blog.rstudio.com/2016/03/29/feather/

6.17.2 `mapply()` and `pmap()`

Sometimes, we'd like to employ a wrapper function that takes arguments from multiple different lists/vectors. Then, we can consider using `mapply()` from the base R package or `pmap()` from the `purrr` package.

Please see the simple specific example below where the two input lists are of the same length and we are doing a pairwise calculation:

```
mylist1 = list(0:3)
mylist2 = list(6:9)
mylists = list(mylist1, mylist2)

square_sum <- function(x, y) {
  x^2 + y^2
}

#Use `mapply()`
result1 <- mapply(FUN = square_sum, mylist1, mylist2)

#Use `pmap()`
library(purrr)
result2 <- pmap(.l = mylists, .f = square_sum)

#unlist(as.list(result1)) = result2 = [36 50 68 90]
```

There are two major differences between `mapply()` and `pmap()`. The first difference is that `mapply()` takes seperate lists as its input arguments, while `pmap()` takes a list of list. Secondly, the output of `mapply()` will be in the form of a matrix or an array, but `pmap()` produces a list directly.

However, when **the input lists are of different lengths AND/OR the wrapper function doesn't take arguments in pairs, `mapply()` and `pmap()` may not give the preferable results.**

Both `mapply()` and `pmap()` will recycle shorter input lists to match the length of the longest input list. Assume that now `mylist2 = list(6:12)`. Then, `pmap(mylists, square_sum)` will generate `[36  50  68  90 100 122 148]` where elements 0, 1, and 2 are recycled to match 10, 11, and 12. And it will return an error message that "longer object length is not a multiple of shorter object length."

Thus, unless the recycling pattern described above is desirable feature for a certain experiment design, **when the input lists are of different lengths, the best practice is probably to use `lapply()` and then combine the results.**

Here is an example where we'd like to find the `square_sum` for every element combination of `mylist1` and `mylist2`.

```
mylist1 <- list(0:3)
mylist2 <- list(6:12)

square_sum <- function(x, y) {
```

```
  x^2 + y^2
}

results <- list()

for (i in seq_along(mylist1[[1]])) {
  result <- lapply(X = mylist2, FUN = function(y) square_sum(mylist1[[1]][i], y))
  results[[i]] <- result
}
```

This example doesn't work in the way that 0 is paired to 6, 1 is paired to 7, and so on. Instead, every element in `mylist1` will be paired with every element in `mylist2`. Thus, the "unlisted" results from the example will have $4 * 7 = 28$ elements.

We can use `flatten()` or `unlist()` functions to decrease the depths of our results. If the results are data frames, then we will need to use `bind_rows()` to combine them.

6.17.3 Parallel processing with `parallel` and `future` packages

One big drawback of `lapply()` is its long computation time, especially when the list length is long. Fortunately, computers nowadays must have multiple cores which makes parallel processing possible to help make computation much faster.

Assume you have a list called `mylist` of length 1000, and `lapply(X = mylist, FUN = func)` applies the function to each of the 1000 elements one by one in T seconds. If we could execute the `func` in n processors simultaneously, then ideally, we would shrink the computation time to T/n seconds.

In practice, using functions under the `parallel` and the `future` packages, we can split `mylist` into smaller chunks and apply the function to each element of the several chunks in parallel in different cores to significantly reduce the run time.

6.17.3.1 `parLapply()`

Below is a generic example of `parLapply()`:

```
library(parallel)

# Set how many processors will be used to process the list and make cluster
n_cores <- 4
cl <- makeCluster(n_cores)

#Use parLapply() to apply func to each element in mylist
result <- parLapply(cl = cl, x = mylist, FUN = func)

#Stop the parallel processing
stopCluster(cl)
```

Let's still assume `mylist` is of length 1000. The `parLapply` above splits `mylist` into 4 sub-lists each of length 250 and applies the function to the elements of each sub-list in parallel. To be more specific, first apply the function to element 1, 251, 501, 751; second apply to element 2, 252, 502, 752; so on and so forth. As such, the computation time will be greatly reduced.

You can use `parallel::detectCores()` to test how many cores your machine has and to help decide what to put for `n_cores`. It would be a good idea to leave at least one core free for the operating system to use.

We will always start `parLapply()` with `makeCluster()`. **`stopCluster()` is not fully necessary but follows the best practices.** If not stopped, the processing will continue in the back end and consuming the computation capacity for other software in your machine. But keep in mind that stopping the cluster is similar quitting R, meaning that you will need to re-load the packages needed when you need to do parallel processing use `parLapply()` again.

6.17.3.2 `future.lapply()`

Below is a generic example of `future.lapply()`:

```
library(future)
library(future.apply)

# First, plan how the future_lapply() will be resolved
future::plan(
  multisession, workers = future::availableCores() - 1
)

# Use future_lapply() to apply func to each element in mylist
future_lapply(x = mylist, FUN = func)
```

Here, `future::availableCores()` checks how many cores your machine has. Similar to `parLapply()` showed above, `future_lapply()` parallelizes the computation of `lapply()` by executing the function `func` simultaneously on different sub-lists of `mylist`.

6.18 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).

This section provides guidance on both constructing effective pull requests and reviewing code submitted by others. Much of the content in this section is adapted from the tidyverse code review guide[20], which provides excellent principles for code review in R package development.

[20] https://code-review.tidyverse.org/

6.19 Constructing Pull Requests

6.19.1 Write Focused PRs

A focused pull request is **one self-contained change** that addresses just one thing. Writing focused PRs has several benefits:

- **Faster reviews**: It's easier for a reviewer to find 5-10 minutes to review a single bug fix than to set aside an hour for one large PR implementing many features.
- **More thorough reviews**: Large PRs with many changes can overwhelm reviewers, leading to important points being missed.
- **Fewer bugs**: Smaller changes make it easier to reason about impacts and identify potential issues.
- **Easier to merge**: Large PRs take longer and are more likely to have merge conflicts.
- **Less wasted work**: If the overall direction is wrong, you've wasted less time on a small PR.

As a guideline, 100 lines is usually a reasonable size for a PR, and 1000 lines is usually too large. However, the number of files affected also matters—a 200-line change in one file might be fine, but the same change spread across 50 files is usually too large.

6.19.2 Writing PR Descriptions

When you submit a pull request, include a detailed PR title and description. A comprehensive description helps your reviewer and provides valuable historical context.

PR Title: The title should be a short summary (ideally under 72 characters) of what is being done. It should be informative enough that future developers can understand what the PR did without reading the full description.

Poor titles that lack context:

- "Fix bug"
- "Add patch"
- "Moving code from A to B"

Better titles that summarize the actual change:

- "Fix missing value handling in data processing function"
- "Add support for custom date formats in import functions"

PR Description Body: The description should provide context that helps the reviewer understand your PR. Consider including:

- A brief description of the problem being solved
- Links to related issues (e.g., "Closes #123" or "Related to #456")
- A before/after example showing changed behavior
- Possible shortcomings of the approach being used
- For complex PRs, a suggested reading order for the reviewer
- The `Files` tab of a Pull Request page on GitHub allows you to annotate your pull request with inline comments. These comments are not part of the source files; they only exist in GitHub's metadata. Use these comments to explain *changes* whose reasoning might not be self-apparent to a reviewer.

6.19.3 Add Tests

Focused PRs should include related test code. A PR that adds or changes logic should be accompanied by new or updated tests for the new behavior. Pure refactoring PRs should also be covered by tests—if tests don't exist for code you're refactoring, add them in a separate PR first to validate that behavior is unchanged.

6.19.4 Separate Out Refactorings

It's usually best to do refactorings in a separate PR from feature changes or bug fixes. For example, moving and renaming a function should be in a different PR from fixing a bug in that function. This makes it much easier for reviewers to understand the changes introduced by each PR.

Small cleanups (like fixing a local variable name) can be included in a feature change or bug fix PR, but large refactorings should be separate.

6.20 Reviewing Pull Requests

6.20.1 Purpose of Code Review

The primary purpose of code review is to ensure that the overall code health of our projects improves over time. Reviewers should balance the need to make forward progress with the importance of maintaining code quality.

Key principle: Reviewers should favor approving a PR once it is in a state where it definitely improves the overall code health of the system, even if the PR isn't perfect. There is no such thing as "perfect" code—there is only better code. Rather than seeking perfection, seek continuous improvement.

6.20.2 Writing Review Comments

When reviewing code, maintain courtesy and respect while being clear and helpful:

- Comment on the **code**, not the author
- Explain **why** you're making suggestions (reference best practices, design patterns, or how the suggestion improves code health)
- Balance pointing out problems with providing guidance (help authors learn while being constructive)
- Highlight positive aspects too—if you see good practices, comment on those to reinforce them

Poor comment: "Why did you use this approach when there's obviously a better way?"

Better comment: "This approach adds complexity without clear benefits. Consider using [alternative approach] instead, which would simplify the logic and improve readability."

6.20.3 Mentoring Through Review

Code review is an excellent opportunity for mentoring. As a reviewer:

- Leave comments that help authors learn something new
- Link to relevant sections of style guides or best practices documentation
- Consider pair programming for complex reviews—live review sessions can be very effective for teaching

6.20.4 Giving Constructive Feedback

In general, it is the author's responsibility to fix a PR, not the reviewer's. Strike a balance between pointing out problems and providing direct guidance. Sometimes pointing out issues and letting the author decide on a solution helps them learn and may result in a better solution since they are closer to the code.

For very small tweaks (typos, comment additions), use GitHub's suggestion feature to allow authors to quickly accept changes directly in the UI.

6.21 Creating a Pull Request Template

GitHub allows you to create a pull request template in a repository to standardize the information in pull requests. When you add a template, everyone will automatically see its contents in the pull request body.

Follow these steps to add a pull request template:

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 default branch.
    - To store the file in a hidden directory, name it `.github/pull_request_template.md`.
4. In the body of the new file, add your pull request template.

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

## Related Issues

Closes #(issue number)
Related to #(issue number)

## Testing

Describe how this change has been tested.
```

```
## Checklist

- [ ] Tests added/updated
- [ ] Documentation updated
- [ ] Code follows project style guidelines

## Who should review the pull request?

@username
```

6.22 Additional Resources

6.22.1 R Package Development

- R Packages book[21] by Hadley Wickham and Jenny Bryan - comprehensive guide to R package development
- Tidyverse design guide[22] - principles for designing R packages and APIs that are intuitive, composable, and consistent with tidyverse philosophy
- usethis documentation[23] - workflow automation for R projects
- devtools documentation[24] - essential development tools
- pkgdown documentation[25] - create package websites
- testthat documentation[26] - unit testing framework

6.22.2 General R Programming

- R for Data Science[27] by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund - learn data science with the tidyverse
- Advanced R[28] by Hadley Wickham - deep dive into R programming and internals

6.22.3 Shiny Development

- Mastering Shiny[29] by Hadley Wickham - comprehensive guide to building web applications with Shiny
- Engineering Production-Grade Shiny Apps[30] by Colin Fay, Sébastien Rochette, Vincent Guyader, and Cervan Girard - best practices for production Shiny applications

[21] https://r-pkgs.org/
[22] https://design.tidyverse.org/
[23] https://usethis.r-lib.org/
[24] https://devtools.r-lib.org/
[25] https://pkgdown.r-lib.org/
[26] https://testthat.r-lib.org/
[27] https://r4ds.hadley.nz/
[28] https://adv-r.hadley.nz/
[29] https://mastering-shiny.org/
[30] https://engineering-shiny.org/

6.22.4 Git and Version Control

- Happy Git and GitHub for the useR[31] by Jenny Bryan - essential guide to using Git and GitHub with R

[31] https://happygitwithr.com/

7 R Code Style

Adapted by UCD-SeRG team from original by Kunal Mishra, Jade Benjamin-Chung, and Stephanie Djajadi[1]

Follow these code style guidelines for all R code:

7.1 General Principles

- **Follow tidyverse style guide**: https://style.tidyverse.org
- **Use native pipe**: `|>` not `%>%` (available in R >= 4.1.0)
- **Naming**: Use `snake_case` for functions and variables; acronyms may be uppercase (e.g., `prep_IDs_data`)
- **Write tidy code**: Keep code clean, readable, and well-organized

7.2 Function Structure and Documentation

Every function should follow this pattern:

```
#' Short Title (One Line)
#'
#' Longer description providing details about what the function does,
#' when to use it, and important considerations.
#'
#' @param param1 Description of first parameter, including type and constraints
#' @param param2 Description of second parameter
#'
#' @returns Description of return value, including type and structure
#'
#' @examples
#' # Example usage
#' result <- my_function(param1 = "value", param2 = 10)
#'
#' @export
my_function <- function(param1, param2) {
  # Implementation
}
```

See also Section 6.9 for general code documentation practices.

[1] https://jadebc.github.io/lab-manual/coding-style.html

7.3 Comments

Use comments to explain *why*, not *what*:

```
# Good: Explains reasoning
# Use log scale because distribution is highly skewed
ggplot(data, aes(x = log10(income))) + geom_histogram()

# Bad: States the obvious
# Create a histogram
ggplot(data, aes(x = income)) + geom_histogram()
```

File headers (for scripts in `data-raw/` or `inst/analyses/`):

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

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[2] 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 accommodate 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.

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[3] 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
```

[2] https://github.com/kmishra9/Flu-Absenteeism/blob/master/Master's%20Thesis%20-%20Spatial%20Epidemiology%20of%20Influenza/2a%20-%20Statistical-Inputs.R

[3] https://github.com/kmishra9/Flu-Absenteeism/blob/master/Master's%20Thesis%20-%20Spatial%20Epidemiology%20of%20Influenza/1b%20-%20Map-Management.R

Multi-Line Comments - Occasionally, multi-line comments are necessary. You should manually insert line breaks to "hard-wrap" code and comments, whenever lines become longer than 80 characters. `lintr` should object otherwise, even for comments. Try to break lines at semantic boundaries: ends of sentences or phrases. Long lines in source code files make it more difficult to see and comment on diffs in pull requests.

In prose text chunks, Quarto ignores single line breaks, so you should also line-break your prose text in .qmd files to keep them under 80 characters.

You can configure RStudio's settings to display the 80-character margin.

7.4 Line Breaks and Formatting

Blank Lines Before Lists

Always include a blank line before starting a bullet list or numbered list in markdown/Quarto documents. This ensures proper rendering and readability.

Correct:

```
Here are the requirements:

- First item
- Second item
```

Incorrect:

```
Here are the requirements:
- First item
- Second item
```

Here's what happens if you don't add the blank line:

Here are the requirements: - First item - Second item

Line Breaks in Code

- 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) +

  aes(x=.data[["year"]], y=.data[["rd"]], group=.data[[group]]) +

  geom_point(mapping = aes(col = .data[[group]], shape = .data[[group]]),
             position=position_dodge(width=0.2),
             size=2.5) +

  geom_errorbar(mapping = aes(ymin=.data[["lb"]], ymax= .data[["ub"]], col= .data[[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.

7.5 Markdown and Quarto Formatting

7.5.1 Writing about code in Quarto documents

When writing about code in prose sections of quarto documents, use backticks to apply a code style: for example, `dplyr::mutate()`. When talking about packages, use backticks and curly-braces: for example, `{dplyr}`.

7.6 Messaging and User Communication

Use `cli` package functions for all user-facing messages in package functions:

```
# Good
cli::cli_inform("Analysis complete")
cli::cli_warn("Missing data detected")
cli::cli_abort("Invalid input: {x}")

# Bad - don't use these in package code
message("Analysis complete")
warning("Missing data detected")
stop("Invalid input")
```

7.7 Package Code Practices

- **No `library()` in package code**: Use `::` notation or declare in DESCRIPTION Imports
- **Document all exports**: Use roxygen2 (`@title`, `@description`, `@param`, `@returns`, `@examples`)
- **Avoid code duplication**: Extract repeated logic into helper functions

7.8 Tidyverse Replacements

Use modern tidyverse/alternatives for base R functions:

```
# Data structures
tibble::tibble()           # instead of data.frame()
tibble::tribble()          # instead of manual data.frame creation

# I/O
readr::read_csv()          # instead of read.csv()
readr::write_csv()         # instead of write.csv()
```

```
readr::read_rds()          # instead of readRDS()
readr::write_rds()         # instead of saveRDS()

# Data manipulation
dplyr::bind_rows()         # instead of rbind()
dplyr::bind_cols()         # instead of cbind()

# String operations
stringr::str_which()       # instead of grep()
stringr::str_replace()     # instead of gsub()

# Session info
sessioninfo::session_info() # instead of sessionInfo()
```

See also Section 6.15.

7.9 The here Package

The `here` package helps manage file paths in projects by automatically finding the project root and building paths relative to it:

```
library(here)

# Automatically finds project root and builds paths
data <- readr::read_csv(here("data-raw", "survey.csv"))
saveRDS(results, here("inst", "analyses", "results.rds"))
```

This solves the problem of different working directory paths across collaborators. For example, one person might have the project at `/home/oski/Some-R-Project` while another has it at `/home/bear/R-Code/Some-R-Project`. The `here` package handles this automatically.

This works regardless of where collaborators clone the repository. For more details, see the here package vignette[4].

See also Section 6.12 for detailed explanation of the here package.

7.10 Object Naming

Use descriptive names that are both expressive and explicit. Being verbose is useful and easy in the age of autocompletion:

```
# Good
vaccination_coverage_2017_18
absentee_flu_residuals

# Less good
```

[4] https://github.com/jennybc/here_here

```
vaxcov_1718
flu_res
```

Prefer nouns for objects and verbs for functions:

```
# Good
clean_data <- prep_study_data(raw_data)  # verb for function, noun for object

# Less clear
data <- process(input)
```

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

Use `snake_case` for all variable and function names. Avoid using `.` in names (as in base R's `read.csv()`), as this goes against best practices in modern R and other languages. Modern packages like `readr::read_csv()` follow this convention.

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.

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, it's 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 it's worth getting rid of these bad habits now.

For more help, check out Be Expressive: How to Give Your Variables Better Names[5]

7.11 Automated Tools for Style and Project Workflow

7.11.1 Styling

7.11.1.1 RStudio shortcuts

1. **Code Autoformatting** - RStudio includes a fantastic built-in utility (keyboard shortcut: `CMD-Shift-A` (Mac) or `Ctrl-Shift-A` (Windows/Linux)) 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[6] 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$schnam, pattern
  "Carl Munck Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Munck
  "Community United Elementary School" = str_subset(string = OUSD_school_shapes$schnam, pat
  "East Oakland PRIDE Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern
  "EnCompass Academy" = str_subset(string = OUSD_school_shapes$schnam, pattern = "EnCompass
  "Global Family School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Global
  "International Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern
  "Madison Park Lower Campus" = "Madison Park Academy TK-5",
  "Manzanita Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "
  "Martin Luther King Jr Elementary" = str_subset(string = OUSD_school_shapes$schnam, patte
  "PLACE @ Prescott" = "Preparatory Literary Academy of Cultural Excellence",
  "RISE Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Rise
)
```

```
# After
OUSD_not_found_aliases = list(
  "Brookfield Village Elementary"      = str_subset(string = OUSD_school_shapes$schnam, pat
  "Carl Munck Elementary"              = str_subset(string = OUSD_school_shapes$schnam, pat
  "Community United Elementary School" = str_subset(string = OUSD_school_shapes$schnam, pat
  "East Oakland PRIDE Elementary"      = str_subset(string = OUSD_school_shapes$schnam, pat
  "EnCompass Academy"                  = str_subset(string = OUSD_school_shapes$schnam, pat
  "Global Family School"               = str_subset(string = OUSD_school_shapes$schnam, pat
```

[5] https://spin.atomicobject.com/2017/11/01/good-variable-names/
[6] https://www.r-bloggers.com/align-assign-rstudio-addin-to-align-assignment-operators/

```
  "International Community School"     = str_subset(string = OUSD_school_shapes$schnam, pat
  "Madison Park Lower Campus"          = "Madison Park Academy TK-5",
  "Manzanita Community School"         = str_subset(string = OUSD_school_shapes$schnam, pat
  "Martin Luther King Jr Elementary"   = str_subset(string = OUSD_school_shapes$schnam, pat
  "PLACE @ Prescott"                   = "Preparatory Literary Academy of Cultural Excellen
  "RISE Community School"              = str_subset(string = OUSD_school_shapes$schnam, pat
)
```

7.11.1.2 `{styler}`

`{styler}` is another cool R package from the Tidyverse[7] 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[8] 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[a] if you're really hardcore.
>
> [a] http://styler.r-lib.org/articles/customizing_styler.html

> **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 initial code, I wouldn't recommend going this route.

> **styler Package**
>
> For automated styling of entire projects:

[7] https://www.tidyverse.org/articles/2017/12/styler-1.0.0/
[8] https://www.rdocumentation.org/packages/styler/versions/1.1.0/topics/style_dir

```r
# Install styler
install.packages("styler")

# Style all files in R/ directory
styler::style_dir("R/")

# Style entire package
styler::style_pkg()

# Note: styler modifies files in-place
# Always use with version control so you can review changes
```

7.11.1.3 {lintr}

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[9]. 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 superfluous whitespace, and improper use of parentheses or brackets. A list of its other purposes can be found in this link[10], and most guidelines are based on Hadley Wickham's R Style Guide[11].

Note

You can customize your settings to set defaults or to exclude files. More details can be found here[a].

[a] https://cran.r-project.org/web/packages/lintr/readme/README.html#project-configuration

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.

lintr package

For checking code style without modifying files:

```r
# Install lintr
install.packages("lintr")

# Lint the entire package
lintr::lint_package()

# Lint a specific file
lintr::lint("R/my_function.R")
```

[9] https://www.rdocumentation.org/packages/lintr/versions/1.0.3
[10] https://cran.r-project.org/web/packages/lintr/readme/README.html#available-linters
[11] http://r-pkgs.had.co.nz/style.html

The linter checks for:

- Unused variables
- Improper whitespace
- Line length issues
- Style guide violations

You can customize linting rules by creating a `.lintr` or `lintr.R` file in your project root.

7.12 Additional Resources

- Tidyverse style guide[12]: Detailed coding style conventions for writing clear, consistent R code. Covers naming, syntax, pipes, functions, and more.

[12] https://style.tidyverse.org/

8 Big data

Adapted by UCD-SeRG team from original by Kunal Mishra and Jade Benjamin-Chung[1]

8.1 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 specificially, `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[2] and all 3 answers are worth a read.

You can also achieve a performance boost by running `dplyr` commands on `data.tables`, 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`.

8.2 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.

8.3 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

[1] https://jadebc.github.io/lab-manual/big-data.html

[2] https://stackoverflow.com/questions/21435339/data-table-vs-dplyr-can-one-do-something-well-the-other-cant-or-does-poorly/27840349#27840349

- **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.

9 Data masking

Adapted by UCD-SeRG team from original by Anna Nguyen, Jade Benjamin-Chung, and Gabby Barratt Heitmann[1]

For information about UC Davis computing resources for data-intensive work, see Chapter 15.

9.1 General Overview

This chapter covers data masking, a unique process in R in which columns are treated as distinct objects within their dataframe's environment. In our lab, data masking most frequently comes up when writing wrapper functions where arguments to indicate column names are supplied as strings. We often do this when we repeat the same code on multiple columns, and want to apply a function to a vector of strings that correspond to column names in a dataframe. For example, we might want to clean multiple columns using the same function or estimate the same model under different feature sets. Here, we try to break down what data masking is, why this error comes up, and common approaches to solve this problem.

9.1.1 What is Data Masking?

Within certain tidyverse operations, columns are called as if they were variables. For example, while running `df |> mutate(X = …)` R recognizes that `X` specifically references a column in `df` without explicitly stating its membership `df |> mutate(df$X = …)` or calling the column name as a string `df |> mutate("X" = …)`.

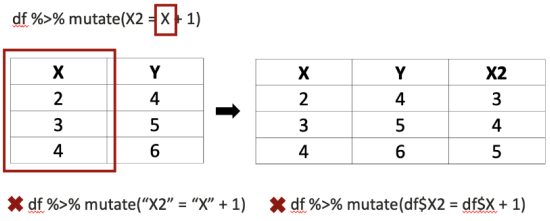

Figure 9.1: Data masking in tidyverse operations

However, this behavior may introduce errors when we attempt to incorporate variables from the global environment within these tidyverse pipelines. In the example shown in Figure 9.1, `column_name = "X"` followed by `df |> mutate(X2 = column_name + 1)` would yield an error, since `column_name` is not a column in `df` and the variable `column_name` is not defined within the environment of `df`

[1] https://jadebc.github.io/lab-manual/data-masking.html

9.1.2 Using tidy evaluation for data masking

In dplyr-based R programming, we make use of tidy evaluation. This allows us to avoid using base R syntax to reference specific columns in a data frame. By leveraging Tidy evaluation-based data masking, we can employ long pipes with several dplyr verbs to manipulate our data using stand-alone variables that store column names as strings.

For example, consider a data frame "df" that contains a column called "heavyrain" that we want to manipulate. Suppose we wanted to convert the values of "heavyrain" into a factor.

Using base R, which does not mask data, heavyrain must have quotes to be treated as a data-variable:

```
df[["outcome"]] = as.factor(df[["heavyrain"]])
```

In a dplyr pipe, heavyrain is being masked using tidy evaluation and will be correctly interpreted as a column because it is recognized as a data-variable: df |> mutate(outcome = as.factor(heavyrain))

With modified data masking, heavyrain is a string that is coerced into being recognized as a data-variable:

```
var_name = "heavyrain"
df |> mutate(outcome = as.factor(!!sym(var_name))
```

While cleaner and often more convenient, the data frame that var_name is in is now "masked" and we refer to the vectors in the dataframe (data-variables) as though it is an object of its own (an environmental-variable). This is why we can just say the variable's name in the context of a pipe – we treat it as though it's an object defined in our environment. Within normal scripts, this is usually fine, because the data frame is "held on to" in the pipe. However, it can cause some programming hurdles when writing functions that take strings of variable/column names as arguments. In the next section, we briefly describe how to troubleshoot common errors in data masking, as relevant to our lab's work.

9.2 Technical Overview

This section covers the R functions and tools that we often use in the context of data masking, focusing on the bang bang operator (`!!`) with symbol coercion (`sym()`) and the Walrus operator (`:=`).

The combined use of `!!` and `sym()` allows us to use strings, rather than data-variables, to reference column names within dplyr. Together, `!!sym("column_name")` forces dplyr to recognize "column_name" as a data-variable prior to evaluating the rest of the expression, enabling the ability to perform calculations on the column while referring to it as a string. `sym()` is a function that turns strings into symbols. In the context of a dplyr pipe, these symbols are interpreted as data-variables. The `!!` (bang bang) operator tells dplyr to evaluate the sym() expression first, e.g. to unquote its expression (e.g. "column_name") and evaluate it as a pre-existing object, first. This is helpful because often we use `sym("column_name")` within a larger expression, and dplyr might evaluate other elements of the expression first without !!, causing errors.

When we want to create a new column (via mutate or summarize), the Walrus operator (`:=`) allows us to specify the new column's name using a string. For example, while `df |> mutate("new_column" = values)` would yield an error, `df |> mutate("new_column" := values)` will correctly create a new column called "new_column". If we want to use a variable representing a string, we can use !! to force the variable to be evaluated before using `:=` to assign the value of the new column.

```
col_name = "new_column"
df |> mutate(!!col_name := values)
```

9.2.1 Example

Suppose we want to write a function "generate_descriptive_table" to summarize how the prevalence of "outcome" varies under different levels of a "risk_factor" in a data frame "df"

We can start by writing the function shell:

```
generate_descriptive_table <- function (df, outcome, rf) {
outcome_dist_by_rf <- ….
return(outcome_dist_by_rf)
}
```

Next, we can filter the data frame for only rows in which "rf" and "outcome" are not missing. We can use !! and sym() within filter to evaluate the strings stored in "rf" and "outcome". Note that defining `!!sym(outcome)` or `!!sym(outcome)` in variables *outside of the dplyr pipeline* will *not* work.

```
generate_descriptive_table <- function (df, outcome, rf,) {
  outcome_dist_by_rf <- df |>
  filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
  ….
  return(outcome_dist_by_rf)
}
```

Similarly, we use !! and sym() in group_by to evaluate column name, stored as a string in the argument "rf"

```
generate_descriptive_table <- function (df, outcome, rf,) {
  outcome_dist_by_rf <- df |>
  filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
  ….
  return(outcome_dist_by_rf)
}
```

Finally, we can use the walrus operator, !! and sym() with "summarize" to create a new column that takes the mean of the column referenced in "rf". We also use "glue" or "paste" to give the new column an informative name that includes the "outcome" it describes.

```r
generate_descriptive_table <- function (df, outcome, rf,) {
  outcome_dist_by_rf <- df |>
  filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
  group_by(!!sym(rf)) |>
  summarize(!!(glue::glue("{outcome}_prev")) := mean(!!sym(outcome)))
  return(outcome_dist_by_rf)
}
```

OR

```r
generate_descriptive_table <- function (df, outcome, rf,) {
  outcome_dist_by_rf <- df |>
  filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
  group_by(!!sym(rf)) |>
  summarize(!!(paste0(outcome, "_prev")) := mean(!!sym(outcome)))
  return(outcome_dist_by_rf)
}
```

OR

```r
generate_descriptive_table <- function (df, outcome, rf,) {
  new_column_name = paste0(outcome, "_prev")
  outcome_dist_by_rf <- df |>
  filter(!is.na(!!sym(outcome)), !is.na(!!sym(rf))) |>
  group_by(!!sym(rf)) |>
  summarize(!!(new_column_name) := mean(!!sym(outcome)))
  return(outcome_dist_by_rf)
}
```

10 Github

Adapted by UCD-SeRG team from original by Stephanie Djajadi and Nolan Pokpongkiat[1]

10.1 Basics

- A detailed tutorial of Git can be found here on the CS61B website[2].
- If you are already familiar with Git, you can reference the summary at the end of Section B[3].
- If you have made a mistake in Git, you can refer to this article[4] 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[5] 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[6]. You can also find instructions on

[1] https://jadebc.github.io/lab-manual/github.html
[2] https://sp19.datastructur.es/materials/guides/using-git#b-local-repositories-narrative-introduction
[3] https://sp19.datastructur.es/materials/guides/using-git#b-local-repositories-narrative-introduction
[4] https://sethrobertson.github.io/GitFixUm/fixup.html
[5] https://desktop.github.com/
[6] https://sp19.datastructur.es/materials/guides/using-git#e-git-branching-advanced-git-optional

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:

Table 10.1: Standard Git workflow for new projects

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[7]	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

[7] https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/creating-a-pull-request#creating-the-pull-request

Table 10.2: 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 Repeated Amend Workflow

When working on a complex task, you may want to make frequent incremental commits to protect your progress, but avoid cluttering your Git history with many tiny "work in

progress" commits. The **Repeated Amend** pattern lets you build up a polished commit gradually.

10.7.1 Basic Workflow

Start with a clean working tree in a functional state. Then:

1. Make a small change and verify your project still works
2. Stage and commit with a temporary message like "WIP" (work in progress)
3. **Do not push yet**
4. Make another small change and verify it works
5. Stage and amend the previous commit: `git commit --amend --no-edit`
6. Repeat steps 4-5 as needed
7. When finished, amend one final time with a proper commit message
8. Push your completed work

In RStudio, you can use the "Amend previous commit" checkbox when committing.

10.7.2 Key Points

- Each amend replaces the previous commit rather than creating a new one
- This keeps your history clean while letting you work incrementally
- Only use this pattern before pushing - never amend commits that others may have pulled
- If you need to undo changes, use `git reset --hard` to return to your last commit state
- Think of commits as climbing protection: use them when in uncertain territory

For more details and troubleshooting scenarios, see the Repeated Amend chapter[8] in Happy Git with R.

10.8 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[9] 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[10], extolling the virtues of a self-contained, portable projects, for your reference.

[8] https://happygitwithr.com/repeated-amend.html
[9] https://github.com/github/gitignore/blob/master/R.gitignore
[10] https://www.tidyverse.org/articles/2017/12/workflow-vs-script/

11 Unix

Adapted by UCD-SeRG team from original by Stephanie Djajadi, Kunal Mishra, Anna Nguyen, and Jade Benjamin-Chung[1]

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.

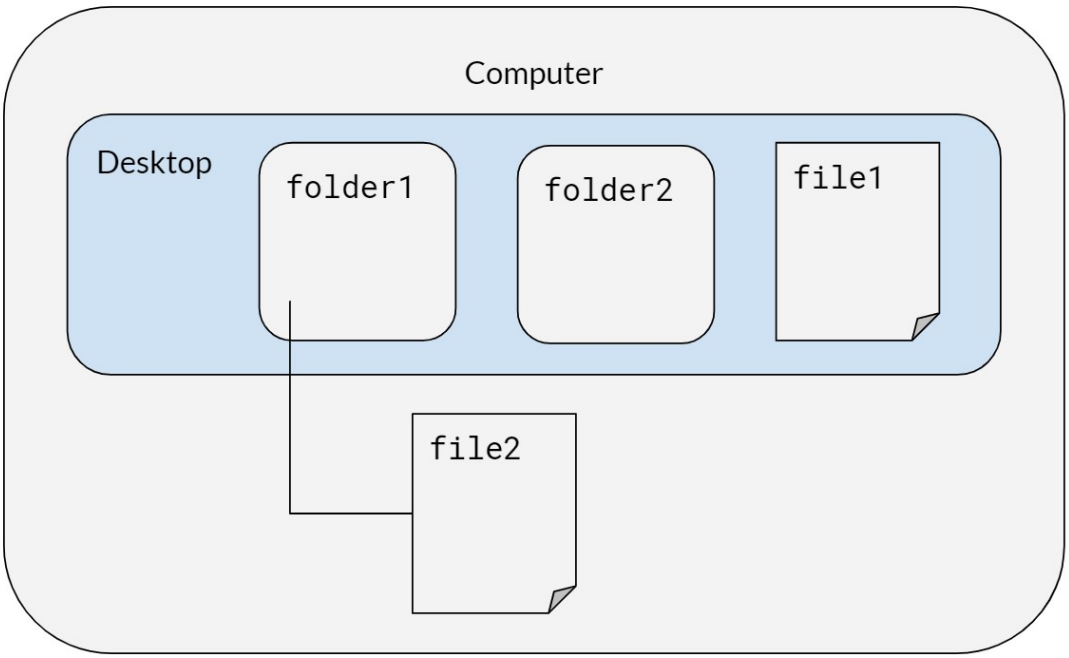

Figure 11.1: Example desktop with folders and files

[1] https://jadebc.github.io/lab-manual/unix.html

11.2 Syntax for both Mac/Windows

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

Table 11.1: Basic Unix commands for Mac and Windows

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`)
`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

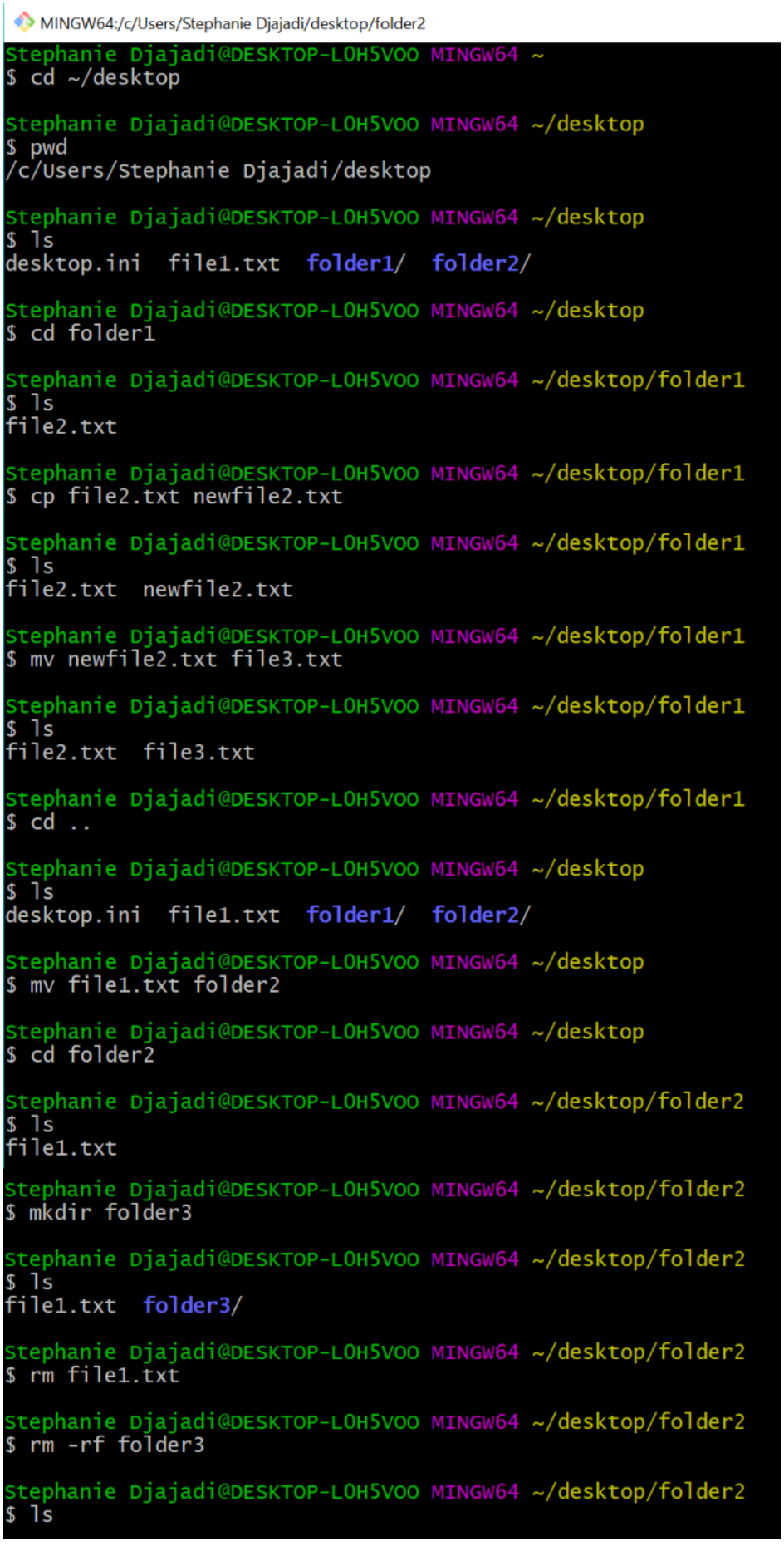

Figure 11.2: Terminal output after executing basic Unix commands

11.3 Running Bash Scripts

Table 11.2: Commands for 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[2]. I strongly recommend 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`

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 `/`.

[2] https://www.howtogeek.com/118594/how-to-edit-your-system-path-for-easy-command-line-access/

11.5 Checking tasks and killing jobs

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[3] has useful tips on using `tmux`.

[3] https://medium.com/@jeongwhanchoi/install-tmux-on-osx-and-basics-commands-for-beginners-be22520fd95e

```
# 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 i

# 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 scri
parser.add_argument("-i", "--identifier", help="Adds an identifier to the directory name wh
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 should be
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
```

12 Reproducible Environments

Adapted by UCD-SeRG team from original by Anna Nguyen[1]

12.1 Package Version Control with renv

12.1.1 Introduction

Replicable code should produce the same results, regardless of when or where it's run. However, our analyses often leverage open-source R packages that are developed by other teams. These packages continue to be developed after research projects are completed, which may include changes to analysis functions that could impact how code runs for both other team members and external replicators.

For example, suppose we had used a function that took in one argument, such that our code contained `example_function(arg_a = "a")`. A few months after we publish our code, the package developers update the function to take in another mandatory argument `arg_b`. If someone runs our code, but has the most recent version of the package, they'll receive an error message that the argument `arg_b` is missing and will not be able to full reproduce our results.

To ensure that the right functions are used in replication efforts, it is important for us to keep track of package versions used in each project.

`renv` can be to promote reproducible environments within R projects. `renv` creates individual package libraries for each project instead of having all projects, which may use different versions of the same package, share the same package library. However, for projects that use many packages, this process can be memory intensive and increase the time needed for a new users to start running code.

In this lab manual chapter, we provide a quick tutorial for integrating `renv` into research workflows. For more detailed instructions, please refer to the `renv` package vignette.

12.1.2 Implementing renv in projects

Ideally, `renv` should be initiated at the start of projects and updated continuously when new packages are introduced in the codebase. However, this process can be initated at any point in a project

To add `renv` to your workflow, follow these steps:

1. Install the `renv` package by running `install.packages("renv")`
2. Create an RProject file and ensure that your working directory is set to the correct folder

[1] https://jadebc.github.io/lab-manual/reproducible-environments.html

3. In the R console, run `renv::init()` to intiialize renv in your R Project
4. This will create the following files: `renv.lock`, `.Rprofile`, `renv/settings.json` and `renv/activate.R`. Commit and push these files to GitHub so that they're accessible to other users.
5. As you write code, update the project's R library by running `renv::snapshot()` in the R console
6. Add `renv::restore()` to the head of your config file, to make sure that all users that run your code are on the same package versions.

12.1.3 Using projects with renv

If you're starting to work on an ongoing project that already has `renv` set up, follow these steps to ensure that you're using the same project versions.

1. Install the `renv` package by running `install.packages("renv")`
2. Pull the most updated version of the project from GitHub
3. Open the project's RProject file
4. Run `renv::restore()`. In our lab's projects, this is often already found at the top of the config file, so you can just run scripts as is.
5. This will pull up a list of the project's packages that need to be updated for you to be consistent with the project. The console will ask if you want to proceed with updating these packages - type "Y" to continue.
6. Wait for the correct versions of each package to install/update. This may take some time, depending on how many packages the project uses.
7. Your R environment should now be using the same package versions as specified in the `renv` lock file. You should now be able to replicate the code.
8. If you make edits to the code and introduce new/updated packages, see the section above for instructions on how to make updates.

13 Code Publication

Adapted by UCD-SeRG team from original by Nolan Pokpongkiat[1]

13.1 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

13.2 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.[2]

13.3 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.

13.4 Document functions

Every function you write must include a header to document its purpose, inputs, and outputs. See template for the function documentation here.[3]

[1] https://jadebc.github.io/lab-manual/code-publication.html
[2] https://ucd-serg.github.io/lab-manual/coding-practices.html#file-headers
[3] https://ucd-serg.github.io/lab-manual/coding-practices.html#function-documentation

13.5 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.

13.6 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.[4]

13.7 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

[4] https://ucd-serg.github.io/lab-manual/unix.html#example-code-for-runfilesavelogs

- 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: UCD-SeRG team (adapted from original 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.

13.8 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.

13.9 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.[5]

[5] https://docs.github.com/en/enterprise/2.13/user/articles/creating-releases

14 Data Publication

Adapted from Fanice Nyatigo and Ben Arnold's chapter in the Proctor-UCSF Lab Manual[1]

14.1 Overview

Warning! ***NEVER*** *push a dataset into the public domain (e.g., GitHub, OSF) without first checking with lab leadership to ensure that it is appropriately de-identified and we have approval from the sponsor and/or human subjects review board to do so. For example, we will need to re-code participant IDs (even if they contain no identifying information) before making data public to completely break the link between IDs and identifiable information stored on our servers.*

If you are releasing data into the public domain, then consider making available *at minimum* a `.csv` file and a codebook of the same name (note: you should have a codebook for internal data as well). We often also make available `.rds` files as well. For example, your `mystudy/data/public` directory could include three files for a single dataset, two with the actual data in `.rds` and `.csv` formats, and a third that describes their contents:

```
analysis_data_public.csv
analysis_data_public.rds
analysis_data_public_codebook.txt
```

In general, datasets are usually too big to save on GitHub, but occasionally they are small. Here is an example of where we actually pushed the data directly to GitHub: https://github.com/ben-arnold/enterics-seroepi/tree/master/data .

If the data are bigger, then maintaining them under version control in your git repository can be unwieldy. Instead, we recommend using another stable repository that has version control, such as the Open Science Framework (osf.io[2]). For example, all of the data from the WASH Benefits trials (led by investigators at Berkeley, icddr,b, IPA-Kenya and others) are all stored through data components nested within in OSF projects: https://osf.io/tprw2/. Another good option is Dryad (datadryad.org[3]) or institutional digital repositories.

[1] https://urlisolation.com/browser?clickId=524DE241-3F8F-4C98-B619-3C278374BF64&traceToken=1728923499%3Bucsfmed_hosted%3Bhttps%3A%2F%2Fproctor-ucsf.github.io%2Fd&url=https%3A%2F%2Fproctor-ucsf.github.io%2Fdcc-handbook%2Fpublicdata.html

[2] https://osf.io

[3] https://datadryad.org/

We recommend cross-linking public files in GitHub (scripts/notebooks only) and OSF/Dryad/institutional digital repositories.

Below are the main steps to making data public, after finalizing the analysis datasets and scripts:
1. Remove Protected Health Information (PHI)
2. Create public IDs or join already created public IDs to the data
3. Create an OSF repository and/or Dryad/institutional digital repository
4. Edit analysis scripts to run using the public datasets and test (optional)
5. Create a public github page for analysis scripts and link to OSF and/or Dryad/Zenodo
6. Go live

14.2 Removing PHI

Once the data is finalized for analysis, the first step is to strip it of Protected Health Information (PHI), or any other data that could be used to link back to specific participants, such as names, birth dates, or GPS coordinates at the village/neighborhood level or below. PHI includes, but is not limited to:

14.2.1 Personal information

These are identifiers that directly point to specific individuals, such as:
- Names, addresses, photographs, date of birth
- A combination of age, sex, and geographic location (below population 20,000) is considered identifiable

14.2.2 Dates

Any specific dates (e.g., study visit dates, birth dates, treatment dates) are usually problematic.
- If a dataset requires high resolution temporal information, coarsen visit or measurement dates to be two variables: year and week of the year (1-52).
- If a dataset requires age, provide that information without a birth date (typically month resolution is sufficient)

Caution! *If making changes to the format of dates or ages, make sure your analysis code runs on these modified versions of the data (step 3)!*

14.2.3 Geographic information

Do not include GPS coordinates (longitude, latitude) except in special circumstances where they have been obfuscated/shifted. Reach out to lab leadership before doing this because it can be complicated.

Do not include place names or codes (e.g., US Zip Codes) if the place contains <20,000 people. For villages or neighborhoods, code them with uninformative IDs. For sub-districts or districts, names are fine.

If an analysis requires GPS locations (e.g., to make a map), then typically we include a disclaimer in the article's data availability statement that explains we cannot make GPS locations public to protect participant confidentiality. As a middle ground, we typically make our *code* public that runs on the geo-located data for transparency, even if independent researchers can't actually run that code (although please be careful to ensure the code itself does not in any way include geographic identifiers).

For more examples of what constitutes PHI, please refer to this link: https://cphs.berkeley.edu/hipaa/hipaa18.html

14.3 Create public IDs

14.3.1 Rationale

The UC Davis IRB requires that public datasets not include the original study IDs to identify participants or other units in the study (such as village IDs). The reason is that those IDs are linked in our private datasets to PHI. By creating a new set of public IDs, the public dataset is one step further removed from the potential to link to PHI.

14.3.2 A single set of public IDs for each study

For each study, it is ideal to create a single set of public IDs whenever possible. We could create a new set of public IDs for every public dataset, but the downside is that independent researchers could no longer link data that might be related. By creating a single set of public IDs associated with each internal study ID, public files retain the link.

Maintaining a single set of public IDs requires a shared "bridge" dataset, that includes a row for each study ID and has the associated public ID. For studies with multiple levels of ID, we would typically have separate bridge datasets for each type of ID (e.g,. cluster ID, participant ID, etc.)

Create a public ID that can be used to uniquely identify participants and that can internally be linked to the original study IDs. We recommend creating a subdirectory in the study's shared data directory to store the public IDs. The shared location enables multiple projects to use the same IDs. Create the IDs using a script that reads in the study IDs, creates a unique (uninformative) public ID for the study IDs, and then saves the bridge dataset. The script should be saved in the same directory as the public ID files.

Caution! *Note that small differences may arise if the new public IDs do not necessarily order participants in the same way as the internal IDs. The small differences are all in estimates that rely on resampling, such as Bootstrap CIs, permutation P-values, and TMLE, as the resampling process may lead tp slightly different re-samples. The key here, to ensure the results are consistent irrespective of the dataset used, is simply to not assign public IDs randomly. Use* `rank()` *on the internal ID instead of* `row_number()` *to ensure that the order is always the same.*

14.3.3 Example scripts

We have created a self-contained and reproducible example that you can run and replicate when making data public for your projects. It contains the following files and folders:

1. `data/final/`- folder containing the projects final data in both csv and rds formats

2. `code/DEMO_generate_public_IDs.R`- creates randomly generated public IDs that can be matched to the trial's assigned patient IDs.

3. `data/make_public/DEMO_internal_to_publicID.csv`- the output from step #2, a bridge dataset with two variables- the new public ID and the patient's assigned ID.

4. `code/DEMO_create_public_datasets.R`- joins the public IDs to the trial's full dataset, and strips it of the assigned patient ID.
5. `data/public/`- folder containing the output from step #3- de-identified public dataset, in csv and rds formats, with uniquely identifying public IDs that cannot be easily linked back to the patient's ID.

The example workflow is accessible via GitHub: https://github.com/proctor-ucsf/dcc-handbook/tree/master/templates/making-data-public

14.4 Create a data repository

First, ensure that you create a codebook and metadata file for each public dataset See the DCC guide on Documenting datasets[4]. Use the same name as the datasets, but with "-codebook.txt" / "-codebook.html" / "-codebook.csv" at the end (depending on the file format for the codebook). One nice option is the R codebook package, which also generates JSON output that is machine-readable.

14.4.1 Steps for creating an Open Science Framework (OSF) repository:

1. Create a new OSF project per these instructions: https://help.osf.io/article/252-create-a-project

[4] https://proctor-ucsf.github.io/dcc-handbook/datawrangling.html#documenting-datasets

2. Create a data component and upload the datasets in .csv and .rds format along with the codebooks. The primary format for public dissemination is .csv but we make the .rds files available too as auxiliary files for convenience.
3. Create a notebook component and upload the final .html files (which will not be on github… but see optional item below)
4. On the OSF landing Wiki, provide some context. Here is a recent example: https://osf.io/954bt/
5. Create a Digital Object Identifier (DOI) for the repository. A DOI is a unique identifier that provides a persistent link to content, such as a dataset in this case. Learn more about DOIs[5]
6. Optional: Complete the software checklist and system requirement guide for the analysis to guide others. Include it on the GitHub README for the project: https://github.com/proctor-ucsf/mordor-antibody

14.5 Edit and test analysis scripts

Make minor changes to the analysis scripts so that they run on public data. If using version control in GitHub, the most straight-forward way is to create a branch from the main git branch that reads in the public files, and then renames the new public ID variable, e.g., "id_public" to the internally recognized ID variable name, e.g. "recordID", when reading in the public data. Re-run all the analysis scripts to ensure that they still work with the public version of the dataset.

14.6 Create a public GitHub page for public scripts

At minimum, we should include all of the scripts required to run the analyses. **IMPORTANT**: ensure you have taken a snapshot and saved your computing environment using the `renv` package (`renv`).

See examples:
- ACTION - https://github.com/proctor-ucsf/ACTION-public
- NAITRE - https://github.com/proctor-ucsf/NAITRE-primary

> ***Caution!*** *Read through the scripts carefully to ensure there is no PHI in the code itself*

Once a public GitHub page exists, you can create a new component on an OSF project (step 3, above) and link it to the public version of the GitHub repo.

[5] https://researchdata.princeton.edu/research-lifecycle-guide/whats-doi-and-what-should-i-know-about-citing-datasets

14.7 Go live

On GitHub, it is useful to create an official "release" version to freeze the repository, where you can have "associated files" with each version. Include the .html notebook output as additional files — since they aren't tracked in GitHub, it does provide a way of freezing / saving the HTML output for us and others. OSF examples of a studies from UCSF's Proctor Foundation:
- ACTION - https://osf.io/ca3pe/
- NAITRE - https://osf.io/ujeyb/
- MORDOR Niger antibody study - https://osf.io/dgsq3/

Further reading on end-to-end data management: How to Store and Manage Your Data - PLOS[6]

[6] https://plos.org/resource/how-to-store-and-manage-your-data/#data-management-plan

15 High-performance computing (HPC)

Adapted by UCD-SeRG team from original by Anna Nguyen, Jade Benjamin-Chung, and Gabby Barratt Heitmann[1]

When you need to run a script that requires a large amount of RAM, large files, or that uses parallelization, UC Davis provides several high-performance computing (HPC) resources.

15.1 UC Davis Computing Resources

15.1.1 Available Resources

UC Davis HPC Clusters: - **Farm Cluster** (hpc.ucdavis.edu[2]): UC Davis's primary HPC cluster providing shared computing resources for research

PHS Shared Compute Environments: For lab members affiliated with the School of Public Health Sciences (PHS), additional shared computing environments are available. These environments provide secure, HIPAA-compliant computing resources suitable for working with sensitive health data.

- **Shiva** (shiva.ucdavis.edu): SLURM-based cluster for computational work
- **Mercury** (mercury.ucdavis.edu): RStudio GUI computing environment

For detailed information about PHS shared compute environments, including access procedures, security guidelines, and usage policies, please refer to the PHS Shared Compute Environments Guide[3].

Contact lab leadership for assistance with: - Requesting access to computing resources - Choosing the appropriate computing environment for your project - Setting up your computing environment

15.2 Getting started with SLURM clusters

To access a UC Davis HPC cluster, in terminal, log in using SSH. For example, to access shiva:

```
ssh USERNAME@shiva.ucdavis.edu
```

[1] https://jadebc.github.io/lab-manual/slurm.html
[2] https://hpc.ucdavis.edu/
[3] assets/files/PHS_Shared_Compute_Environments.pdf

You will be prompted to enter your UC Davis credentials and may need to complete two-factor authentication.

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
```

15.2.1 One-Time System Set-Up

To keep the install packages consistent across different nodes, you will need to explicitly set the pathway to your R library directory.

Open your `~/.Renviron` file (`vi ~/.Renviron`) and append the following line:

Note: Once you open the file using `vi [file_name]`, you must press `i` (on Mac OS) or `Insert` (on Windows) to make edits. After you finish, hit `Esc` to exit editing mode and type `:wq` to save and close the file.

```
R_LIBS=~/R/x86_64-pc-linux-gnu-library/4.0.2
```

Alternatively, run an R script with the following code on the cluster:

```
r_environ_file_path = file.path(Sys.getenv("HOME"), ".Renviron")
if (!file.exists(r_environ_file_path)) file.create(r_environ_file_path)

cat("\nR_LIBS=~/R/x86_64-pc-linux-gnu-library/4.0.2",
    file = r_environ_file_path, sep = "\n", append = TRUE)
```

To load packages that run off of C++, you'll need to set the correct compiler options in your R environment.

Open the Makevars file (`vi ~/.R/Makevars`) and append the following lines

```
CXX14FLAGS=-O3 -march=native -mtune=native -fPIC
CXX14=g++
```

Alternatively, create an R script with the following code, and run it on the cluster:

```
dotR = file.path(Sys.getenv("HOME"), ".R")
if (!file.exists(dotR)) dir.create(dotR)

M = file.path(dotR, "Makevars")
if (!file.exists(M)) file.create(M)

cat("\nCXX14FLAGS=-O3 -march=native -mtune=native -fPIC",
    "CXX14=g++",
    file = M, sep = "\n", append = TRUE)
```

15.3 Moving files to the cluster

The `$HOME` directory is a good place to store code and small test files. Save large files to the `$SCRATCH` directory or other designated storage areas. Check with the UC Davis HPC documentation[4] for specific quotas and retention policies. It's best to create a bash script that records the file transfer process for a given project. See example code below:

```
# note: the following steps should be done from your local
# (not after ssh-ing into the cluster)

# securely transfer folders from Box to cluster home directory
# note: the -r option is for folders and is not needed for files
scp -r "Box/project-folder/folder-1/" USERNAME@shiva.ucdavis.edu:/home/users/USERNAME/

# securely transfer folders from Box to your cluster scratch directory
scp -r "Box/project-folder/folder-2/" USERNAME@shiva.ucdavis.edu:/scratch/users/USERNAME/

# securely transfer folders from Box to shared scratch directory
scp -r "Box/project-folder/folder-3/" USERNAME@shiva.ucdavis.edu:/scratch/group/GROUPNAME/
```

15.4 Installing packages on the cluster

When you begin working on a cluster, you will most likely encounter problems with installing packages. To install packages, login to the cluster on the command line and open a development node. Do not attempt to do this in RStudio Server, as you will have to re-do it for every new session you open.

```
ssh USERNAME@shiva.ucdavis.edu

sdev
```

You should only have to install packages once. The cluster may require that you specify the repository where the package is downloaded from. You may also need to add an additional argument to install.packages to prevent the packages from locking after installation:

```
install.packages(<PACKAGE NAME>, repos="http://cran.us.ur-project.org",
                 INSTALL_opts = "--no-lock"")
```

In order for some R packages to work on clusters, it is necessary to load specific software modules before running R. These must be loaded each time you want to use the package in R. For example, for spatial and random effects analyses, you may need the modules/packages below. These modules must also be loaded on the command line prior to opening R in order for package installation to work.

[4] https://hpc.ucdavis.edu/

```
module --force purge # remove any previously loaded modules, including math and devel
module load math
module load math gmp/6.1.2
module load devel
module load gcc/10
module load system
module load json-glib/1.4.4
module load curl/7.81.0
module load physics
module load physics udunits geos
module load physics gdal/2.2.1 # for R/4.0.2
module load physics proj/4.9.3 # for R/4.0.2
module load pandoc/2.7.3

module load R/4.0.2

R # Open R in the Shell window to install individual packages or test code
Rscript install-packages.R # Alternatively, run a package installation script in the Shell
```

Figuring out the issues with some packages will require some trial and error. If you are still encountering problems installing a package, you may have to install other dependencies manually by reading through the error messages. If you try to install a dependency from CRAN and it isn't working, it may be a module. You can search for it using the `module spider` command:

```
module spider DEPENDENCY NAME
```

You can also reach out to UC Davis HPC support for help. Visit hpc.ucdavis.edu[5] for support information.

15.5 Testing your code

Both of the following ways to test code on a cluster are recommended for making small changes, such as editing file paths and making sure the packages and source files load. You should write and test the functionality of your script locally, only testing on the cluster once major bugs are out.

15.5.1 The command line

There are two main ways to explore and test code on computing clusters. The first way is best for users who are comfortable working on the command line and editing code in base R. Even if you are not comfortable yet, this is probably the better way because these commands will transfer between different cluster computers using Slurm.

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.

[5] https://hpc.ucdavis.edu/

```
# open development node
sdev

# Load all the modules required by the packages you are using
module load MODULE NAME

# Load R (default version)*
module load R

# initiate R in command line
R
```

*Note: for collaboration purposes, it's best for everyone to work with one version of R. Check what version is being used for the project you are working on. Some packages only work with some versions of R, so it's best to keep it consistent.

15.5.2 RStudio Server

For RStudio GUI computing, UC Davis provides mercury.ucdavis.edu. This is accessed through a web browser and provides an RStudio interface. You will be prompted to authenticate with your UC Davis credentials. This is the best way to work with R for people who are not comfortable accessing & editing in base R in a Shell application.

Note that mercury does not have SLURM, so it's best suited for interactive work and smaller computations. For large-scale computations requiring SLURM job scheduling, use shiva.ucdavis.edu instead.

When using RStudio Server, you can test your code interactively. However, do NOT use the RStudio Server's Terminal to install packages and configure your environment for SLURM-based clusters, as you will likely need to re-do it for every session/project. For SLURM clusters, use the command line approach described earlier.

15.5.3 Filepaths & configuration on the cluster

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

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

  cluster_path = paste0(Sys.getenv("HOME"), "/project-name/")

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

15.6 Storage & group storage access

15.6.1 Individual storage

There are multiple places to store your files on computing clusters. Each user has their own `$HOME` directory as well as a `$SCRATCH` directory. These are directories that can be accessed via the command line once you've logged in to the cluster:

```
cd $HOME
cd /home/users/USERNAME # Alternatively, use the full path

cd $SCRATCH
cd /scratch/users/USERNAME # Full path
```

You can also navigate to these using the File Explorer if available through a web interface.

`$HOME` typically has a volume quota (e.g., 15 GB). `$SCRATCH` typically has a larger volume quota (e.g., 100 TB), but files here may get deleted after a certain period of inactivity. Thus, use `$SCRATCH` for test files, exploratory analyses, and temporary storage. Use `$HOME` for long-term storage of important files and more finalized analyses.

Check with the UC Davis HPC documentation[6] for specific storage options and quotas.

15.6.2 Group storage

The lab may have shared `$GROUP_HOME` and `$GROUP_SCRATCH` directories to store files for collaborative use. These typically have larger quotas and may have different retention policies. You can access these via the command line or navigate to them using the File Explorer:

```
cd $GROUP_HOME
cd /home/groups/GROUPNAME

cd $GROUP_SCRATCH
cd /scratch/groups/GROUPNAME
```

However, saving files to group storage can be tricky. You can try using the scp command in the section "Moving files to the cluster" to see if you have permission to add files to group directories. Read the next section to ensure any directories you create have the right permissions.

15.6.3 Folder permissions

Generally, when we put folders in `$GROUP_HOME` or `$GROUP_SCRATCH`, it is so that we can collaborate on an analysis within the research group, so multiple people need to be able to access the folders. If you create a new folder in `$GROUP_HOME` or `$GROUP_SCRATCH`, please check the folder's permissions to ensure that other group members are able to access its contents. To check the permissions of a folder, navigate to the level above it, and enter `ls -l`. You will see output like this:

[6] https://hpc.ucdavis.edu/

```
drwxrwxrwx 2 jadebc jadebc  2204 Jun 17 13:12 myfolder
```

Please review this website[7] to learn how to interpret the code on the left side of this output. The website also tells you how to change folder permissions. In order to ensure that all users and group members are able to access a folder's contents, you can use the following command:

```
chmod ugo+rwx FOLDER_NAME
```

15.7 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[8]. Here is an example of an sbatch script with the following options:

- `job-name=run_inc`: Job name that will show up in the SLURM 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
- `time=47:59:00`: Set maximum run time to 47 hours and 59 minutes. If you don't include this option, the cluster will automatically exit scripts after 2 hours of run time (default may vary by cluster).

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
#SBATCH --time=47:59:00

cd $HOME/project-code-repo/2-analysis/
```

[7] https://www.chriswrites.com/how-to-change-file-permissions-using-the-terminal/
[8] https://slurm.schedmd.com/sbatch.html

```
module purge

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

# load gcc, a C++ compiler (required for certain packages)
module load gcc/10

# 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
```

16 Working with AI

AI-powered coding assistants[1] can dramatically accelerate and improve your work, but they require careful and responsible use. Lab members who use AI tools must adhere to the following guidelines:

16.1 Responsibility for validation

You are fully responsible for checking and validating all AI-generated code and content. AI tools can make mistakes, generate insecure code, produce incorrect logic, or suggest approaches that are inappropriate for our specific research context. Before using any AI-generated code:

- Carefully review the code to ensure you understand what it does
- Test the code thoroughly to verify it works as expected
- Verify that the logic is appropriate for your specific use case
- Check that the code follows our lab's coding standards and best practices
- Ensure the code does not introduce security vulnerabilities or data privacy issues

Never blindly copy and paste AI-generated code without understanding it. If you don't understand what the AI has suggested, take the time to learn or ask a colleague for help.

16.2 Disclosure of AI use

You must clearly state whenever you have used AI tools in your work. This is essential for transparency and reproducibility. Specifically:

- In code comments, note when AI tools were used to generate or significantly modify code
- In commit messages, mention if AI tools assisted with the changes
- In manuscripts and reports, acknowledge AI tool usage in the methods or acknowledgments section
- In presentations, disclose AI assistance when relevant

Example code comment:

```
# The following function was generated with assistance from GitHub Copilot
# and has been reviewed and tested to ensure correctness
```

[1] https://en.wikipedia.org/wiki/AI-assisted_software_development

16.3 Attribution of sources

When using AI tools to generate content that borrows from or adapts existing sources, you must ensure proper attribution. AI tools sometimes paraphrase or adapt content from documentation, guides, or other resources without clearly indicating the original source. It is your responsibility to:

- Ask the AI tool to identify and properly cite sources when it borrows or adapts content
- Verify that any content the AI generates includes appropriate citations
- Add citations yourself if the AI fails to do so
- Follow appropriate attribution practices for the type of content (code comments, documentation, academic writing, etc.)

When instructing AI tools to create documentation or written content, explicitly request that they provide proper attribution for any borrowed or adapted material. For example: "Please quote from and paraphrase [source], with proper attribution" rather than simply asking it to summarize information on a topic.

16.4 Coding Agents

We recommend working with **AI coding agents**[2] to help you code[3].

16.4.1 What are AI coding agents?

AI coding agents are AI agents[4] specialized for coding. They differ from other AI coding tools in important ways:

Compared to inline coding assistants (like traditional autocomplete), coding agents work autonomously rather than providing suggestions as you type. They can navigate entire codebases, execute commands, and complete multi-step tasks without constant human guidance.

Compared to AI chatbots (like ChatGPT or Claude), coding agents don't just generate code snippets in conversation—they actively interact with your development environment. While chatbots require you to copy code from a chat window and manually integrate it into your project, coding agents directly read your codebase, make changes to files, run tests and build commands, and create pull requests with their proposed changes. Chatbots are conversational assistants; coding agents are autonomous development tools.

Coding agents are autonomous software programs that can:

- **Understand and execute complex tasks**: Coding agents can interpret natural language instructions and break them down into actionable development tasks
- **Navigate and modify codebases**: They can read, understand, and edit multiple files across a repository to implement features or fix bugs
- **Run tools and commands**: Coding agents can execute build commands, run tests, use linters, and interact with development tools

[2] https://github.com/features/copilot/agents
[3] https://en.wikipedia.org/wiki/AI-assisted_software_development
[4] https://en.wikipedia.org/wiki/AI_agent

- **Make decisions autonomously**: They can plan their approach, make technical decisions, and adjust their strategy based on results
- **Work iteratively**: Coding agents can test their changes, identify issues, and refine their solutions through multiple iterations
- **Create comprehensive solutions**: They can implement complete features that span multiple files, including code, tests, and documentation

Coding agents operate in isolated environments where they can safely experiment and validate changes before proposing them. This allows them to work more independently than inline coding assistants, which require step-by-step human direction. The agent workflow typically involves analyzing requirements, planning an implementation, making changes, testing those changes, and creating a pull request with the results.

While coding agents can handle substantial development tasks, they still require human oversight and review. The human developer remains responsible for:

- Reviewing the agent's work
- Ensuring the solution meets requirements
- Verifying code quality and security
- Making the final decision to merge changes

16.4.2 How to Work with Coding Agents

GitHub Copilot coding agents can be used in several ways to automate development tasks:

16.4.2.1 Assigning Issues to Copilot

You can assign GitHub Issues directly to `@copilot` just like you would assign to a human collaborator:

1. **On GitHub.com**: Navigate to an issue and assign it to Copilot in the assignees section
2. **In VS Code**: In the GitHub Pull Requests or Issues view, right-click an issue and select "Assign to Copilot"
3. **From Copilot Chat**: Delegate tasks to Copilot directly from the chat interface in supported editors

16.4.2.2 The Agent Workflow

Once assigned an issue, the coding agent follows an autonomous workflow:

1. **Analysis**: Reviews the issue description, related discussions, repository instructions, and codebase context
2. **Planning**: Determines what changes are needed and creates a work plan
3. **Development**: Works in an isolated GitHub Actions environment, modifies code, runs tests and linters, and validates changes
4. **Pull Request Creation**: Creates a draft pull request with implemented changes, audit logs, and a summary of modifications

5. **Review and Iteration**: You review the PR and can request changes; the agent will iterate based on your feedback

16.4.2.3 Example: This Document

This very section you're reading was created through the coding agent workflow:

1. **Issue created**: Issue #42[5] requested adding discussion about benefits and hazards of coding agents, including a Matrix film connection and best practices
2. **Agent assigned**: The issue was assigned to `@copilot`
3. **Work completed**: The agent analyzed the requirements, reviewed the repository structure, and implemented the changes across multiple files
4. **Pull request**: PR #50[6] was created with comprehensive content about coding agents, including this "How to Work with Coding Agents" section, benefits and hazards discussion, best practices, and firewall configuration details
5. **Iteration**: The PR received feedback comments requesting additional links, improved wording, and this example section—all of which the agent addressed through follow-up commits

This demonstrates the full lifecycle of working with a coding agent on a real documentation task.

16.4.2.4 Collaborating with Coding Agents

Between iterations of asking coding agents to extend a PR, human collaborators can also push changes directly to the PR branch. This allows for a collaborative workflow where both humans and agents contribute:

- **Human contributions**: You can make quick fixes, add content, or refine the agent's work by pushing commits to the same branch
- **Agent iterations**: After your changes, you can ask the agent to continue working on additional requirements

Important: Try to avoid pushing changes while the coding agent is actively working. Simultaneous edits can produce conflicting diffs that:

- Need to be manually resolved
- May confuse both human and AI collaborators
- Could result in lost work or merge conflicts

Best practice: Wait for the agent to complete its current iteration (indicated by the PR being updated) before pushing your own changes to the branch. Then assign new work to the agent for the next iteration.

[5] https://github.com/UCD-SERG/lab-manual/issues/42
[6] https://github.com/UCD-SERG/lab-manual/pull/50

16.4.2.5 Directly Prompting for Pull Requests

You can also prompt Copilot to create pull requests without first creating an issue:

- Use Copilot Chat in your editor to describe the changes you want
- The agent will analyze your request and create a pull request
- This is useful for quick fixes or well-defined tasks

16.4.2.6 Important Safeguards

- **Human approval required**: Coding agents cannot merge their own changes
- **Branch restrictions**: Agents can only push to their own branches (e.g., `copilot/*`)
- **Full transparency**: All agent actions are logged and visible in the PR

For detailed instructions, see GitHub Copilot coding agent documentation[7].

16.4.3 Benefits and Hazards

Coding agents are powerful programs that can work autonomously. They create pull requests that propose changes to the code in our repositories, potentially including their own configuration files and our automated workflows. They can work powerfully on our behalf, but they require careful oversight and control to ensure they serve our interests and that we understand the consequences of their actions.

Coding agents offer several advantages:

- **Built-in transparency**: Coding agents create a clear record of their role in your work through commit history and code suggestions
- **Context-aware suggestions**: Coding agents understand your codebase and can make contextually relevant suggestions
- **Integration with version control**: Using coding agents within GitHub ensures that AI-assisted changes are tracked alongside all other code changes
- **Interactive workflow**: Coding agents' interactive nature encourages you to review and modify suggestions rather than blindly accepting them
- **Accelerated development**: Coding agents can help you write boilerplate code, refactor existing code, and implement common patterns more quickly
- **Learning opportunities**: Coding agents can suggest approaches or techniques you may not have considered, helping you expand your coding knowledge

However, coding agents also come with significant hazards:

- **Over-reliance**: Depending too heavily on coding agents can atrophy your coding skills and understanding
- **Subtle bugs**: AI-generated code may contain logic errors that are not immediately obvious

[7] https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent

- **Security vulnerabilities**: Coding agents may introduce insecure patterns or fail to follow security best practices
- **Inappropriate solutions**: AI may suggest solutions that work but are not optimal for your specific research context or constraints
- **Hidden biases**: Coding agents may perpetuate coding patterns or approaches that reflect biases in their training data
- **False confidence**: Well-formatted, professional-looking code from AI can mask underlying problems and reduce critical review
- **Workflow manipulation risks**: Coding agents that modify CI/CD workflows (`.github/workflows/*.yml`) or setup configurations can inadvertently or maliciously compromise repository security, expose secrets, or execute harmful commands

16.4.3.1 Further reading/viewing

- *I Robot* (Asimov 1950)
- *Dune* (Herbert 1965)
- "2001: A Space Odyssey" (1968)
- "The Terminator" (1984)
- "The Matrix" (1999)
- "Blade Runner" (1982)
- "WarGames" (1983)
- *Ender's Game* (Card 1985)
- "The Humans are Dead" (Flight of the Conchords 2007)

Figure 16.1: Agents[8]

16.4.4 Best Practices for Safe and Successful Use

To work with coding agents safely and successfully:

1. **Maintain active supervision**: Never assume AI-generated code is correct. Review every line critically.

2. **Understand before accepting**: If you don't understand what the code does, don't use it. Take time to learn or ask a colleague.
3. **Test thoroughly**: AI-generated code must be tested as rigorously as code you write yourself. Don't skip testing because "the AI wrote it."
4. **Start small**: Begin with small, well-defined tasks to build confidence and understanding of the agent's capabilities and limitations.
5. **Verify logic and assumptions**: Check that the AI hasn't made incorrect assumptions about your data, requirements, or scientific context.
6. **Review for security**: Explicitly check for security issues, especially when handling sensitive data or user input.
7. **Iterate and refine**: Use coding agents as a starting point, not an endpoint. Refine and improve the generated code.
8. **Maintain coding practice**: Regularly write code yourself to maintain and develop your skills. Don't let the agent do everything.

> ⚠ Critical: Exercise Extreme Caution with Workflow Files
>
> Be especially careful when allowing coding agents to edit GitHub Actions workflows or CI/CD configurations. These files control automated processes that can:
>
> - Access secrets and credentials
> - Deploy code to production
> - Execute arbitrary commands in your repository
>
> **Never** allow a coding agent to edit workflow files (especially `.github/workflows/*.yml` or `copilot-setup-step.yml`) without thorough manual review. Before approving any workflow run, always check if the workflow files themselves have been modified. Malicious or erroneous changes to workflows can compromise your entire repository and its secrets.

When using coding agents, work interactively with the AI suggestions: review, modify, and test them rather than accepting them wholesale. This interactive approach helps ensure code quality and deepens your understanding of the code.

Remember: AI tools are assistants, not replacements for your expertise and judgment. The quality and correctness of your work remains your responsibility.

16.4.5 Firewall and Network Configuration

Coding agents require specific network access to function properly. If a coding agent is running behind a corporate firewall or on a restricted network, you may need to configure allowlists to enable coding agent functionality.

16.4.5.1 Built-in Agent Firewall

Coding agents run in a GitHub Actions environment with a built-in firewall that limits internet access by default. This firewall helps protect against:

- Data exfiltration
- Accidental leaks of sensitive information
- Execution of malicious instructions

By default, the agent's firewall allows access to:

- Common OS package repositories (Debian, Ubuntu, Red Hat, etc.)
- Popular container registries (Docker Hub, Azure Container Registry, AWS ECR, etc.)
- Language-specific package registries (npm, PyPI, Maven, RubyGems, etc.)
- Common certificate authorities for SSL validation

For the complete list of allowed hosts, see the Copilot allowlist reference[9].

16.4.5.2 Customizing Agent Firewall Settings

In your repository's "Coding agent" settings page, you can:

- Add custom hosts to the allowlist (for internal dependencies or additional registries)
- Opt out of the default recommended allowlist for stricter security
- Disable the firewall entirely (not recommended)

If a coding agent's request is blocked by the firewall, a warning will be added to the pull request or comment, detailing the blocked address and the command that triggered it.

For more information, see Customizing or disabling the firewall for GitHub Copilot coding agent[10].

16.4.6 When to use a coding agent

Coding agent sessions are currently[11] considered "premium requests", which are limited resources; see https://github.com/features/copilot/plans for details. So, use coding agents sparingly. Use them for complex changes that would be difficult or time-consuming for you to complete by hand. Coding agents also take time to get configured for work, every time you make a request. See https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent/customize-the-agent-environment#preinstalling-tools-or-dependencies-in-copilots-environment for ways to reduce that startup time, but it will never be 0. If you can complete the task faster than the coding agent can, you should probably do it yourself.

Also, the less we practice, the weaker our skills get, and the harder it is for us to supervise the agents and make sure they are actually doing what we want it to do, the way we want it to do it. You should exercise your own coding skills regularly, just like you would for any other skill you want to maintain.

[9] https://docs.github.com/en/copilot/reference/copilot-allowlist-reference
[10] https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent/customize-the-agent-firewall
[11] 2026-01-10

16.4.7 Editing with `.docx` files

GitHub Copilot coding agents can read Microsoft Word (`.docx`) files, including tracked changes and comments. This enables a hybrid editing workflow where:

1. Lab members can export Quarto content to Word format for review
2. Reviewers can make edits, add tracked changes, and insert comments in Word
3. Coding agents can read the `.docx` file and translate the edits back to Quarto format

When using this workflow, make sure to explicitly instruct the coding agent to:

- Examine and apply all tracked changes in the `.docx` file
- Read and address all comments in the `.docx` file
- Translate edits from Word formatting to appropriate Quarto/markdown syntax

This approach makes it easier for collaborators who are more comfortable with Word to contribute to the lab manual while maintaining the source files in Quarto format.

16.4.8 Copilot Instructions for this Repository

The `.github/copilot-instructions.md` file in this repository contains specific instructions and guidelines for GitHub Copilot coding agents when working with the lab manual. This file helps ensure that AI-generated contributions follow the lab's formatting standards, coding conventions, and documentation practices.

The copilot instructions file specifies:

- Markdown and Quarto formatting rules (e.g., blank lines before lists, line breaks in prose)
- R code style guidelines (e.g., using native pipe `|>`, following tidyverse style)
- File organization patterns (e.g., using Quarto includes for modular content)
- How to work with DOCX files for hybrid editing workflows
- Repository-specific best practices

By having these instructions in `.github/copilot-instructions.md`, we ensure that coding agents produce consistent, high-quality contributions that align with the lab's established practices. This reduces the review burden and helps maintain consistency across all contributions to the lab manual, whether made by humans or AI assistants.

Unfortunately, we can't render the `copilot-instructions.md` file as part of this lab manual, because it includes dummy

View copilot-instructions.md

17 Copilot Instructions for UCD-SeRG Lab Manual

This file contains guidelines for GitHub Copilot and other AI assistants when working with the lab manual.

17.1 Markdown and Quarto Formatting

17.1.1 Talking about code

When talking about code in prose sections, use backticks to apply code formatting: for example, `dplyr::mutate()` When talking about packages, use backticks and curly-braces: for example, `{dplyr}`

17.1.2 Blank Lines Before Lists

ALWAYS include a blank line before bullet lists and numbered lists in markdown and Quarto (.qmd) files.
Correct:

```
Here are the key points:

- First item
- Second item
- Third item
```

Incorrect:

```
Here are the key points:
- First item
- Second item
- Third item
```

This applies to:

- Bullet lists (starting with `-` or `*`)
- Numbered lists (starting with `1.`, `2.`, etc.)
- Lists in all .qmd files throughout the repository

17.1.3 Line Breaks in Plain Text

ALWAYS line-break at the ends of sentences and long phrases in plain-text paragraphs in .qmd files to avoid long lines.
Correct:

```
When talking about code in prose sections,
use backticks to apply code formatting.
This helps maintain readability in source files
and makes diffs easier to review.
```

Incorrect:

```
When talking about code in prose sections, use backticks to apply code formatting. This h
```

Benefits:

- Improves readability of source .qmd files
- Makes git diffs clearer and easier to review
- Helps identify specific changes in version control
- Prevents horizontal scrolling when editing
- Follows semantic line breaks best practice

Guidelines:

- Break after complete sentences (at periods)
- Break after long phrases or clauses (at commas or conjunctions)
- Break after approximately 60-80 characters when appropriate
- Keep related short phrases together on one line
- Don't break in the middle of inline code, links, or formatting

17.1.4 Why This Matters

- Ensures consistent markdown rendering across different platforms
- Improves readability in both source and rendered forms
- Prevents rendering issues in Quarto books
- Follows markdown best practices

17.1.5 Cross-References for Figures and Tables

ALWAYS use Quarto's cross-reference system for figures, tables, and other captioned content. See Quarto Cross-References documentation[a] for complete details.
Required label prefixes:

- Figures: `#fig-` (e.g., `#fig-data-masking`, `#fig-workflow-diagram`)
- Tables: `#tbl-` (e.g., `#tbl-git-commands`, `#tbl-summary-stats`)
- Equations: `#eq-` (e.g., `#eq-regression-model`)
- Sections: `#sec-` (e.g., `#sec-introduction`) - already in use throughout manual
- Theorems: `#thm-` (e.g., `#thm-central-limit`)
- Lemmas: `#lem-` (e.g., `#lem-auxiliary-result`)
- Corollaries: `#cor-` (e.g., `#cor-special-case`)
- Propositions: `#prp-` (e.g., `#prp-main-result`)
- Examples: `#exm-` (e.g., `#exm-simple-case`)
- Exercises: `#exr-` (e.g., `#exr-practice-problem`)

For figures (images):

```
![Caption text](path/to/image.png){#fig-label}
```

For tables (markdown tables):

```
| Column 1 | Column 2 |
|----------|----------|
| Data     | Data     |

: Caption text {#tbl-label}
```

For code-generated figures:

```
```{r}
#| label: fig-plot-name
#| fig-cap: "Caption text"

# R code to generate plot
```
```

For code-generated tables:

```
```{r}
#| label: tbl-table-name
#| tbl-cap: "Caption text"

# R code to generate table
```
```

Referencing in text:

- Figures: `@fig-label` produces "Figure X"
- Tables: `@tbl-label` produces "Table X"
- Equations: `@eq-label` produces "Equation X"
- Sections: `@sec-label` produces "Section X"

Benefits:

- Automatic numbering of figures, tables, and equations
- Automatic updates when content is reordered
- Clickable cross-references in HTML and PDF output
- Consistent formatting across all output formats
- Better accessibility for screen readers

17.2 R Code Style

- Follow the tidyverse style guide: https://style.tidyverse.org
- Use native pipe `|>` instead of `%>%`
- Use `snake_case` for variable and function names
- Use `.qmd` files exclusively (not `.Rmd`)
- All R projects should use R package structure

17.3 File Organization

17.3.1 Using Quarto Includes for Modular Content

All chapters should use Quarto includes to decompose content into separate files. This modular approach provides significant benefits for version control, collaboration, and content management.

17.3.1.1 Why Use Includes?

1. **Better Git History**: When sections are reordered, only the main chapter file changes (moving include statements), making it immediately clear that content was reorganized rather than edited. When content is edited, only the specific include file changes. This makes reviews focused and precise.
2. **Easier Code Review**: Reviewers can see exactly what changed—either the organization (main file) or the content (include file)—without having to parse through large diffs.
3. **Modular Maintenance**: Each section lives in its own file, making it easier to:
   - Find and edit specific content
   - Reuse sections across chapters if needed
   - Work on different sections simultaneously without merge conflicts
   - Test and preview individual sections
4. **Clear Structure**: The main chapter file becomes a table of contents showing the organization at a glance.

17.3.1.2 Structure Pattern

Main chapter file (e.g., `05-coding-practices.qmd`):

- Contains the chapter title and introduction
- Contains section headings (##, ###, etc.)
- Uses the `include` shortcode to pull in content (see https://quarto.org/docs/authoring/includes.html for details)
- Shows the organization/outline of the chapter

Include files (e.g., `05-coding-practices/lab-protocols-for-code-and-data.qmd`):

- Stored in a subdirectory matching the chapter name
- Contains only the content for that section (no heading)
- The heading stays in the main chapter file
- Named descriptively using kebab-case

17.3.1.3 Required Pattern

Always follow this pattern:

```
## Section Heading

{{ < include folder/section-name.qmd > }}
```

Correct example:

```
## Section heading

{{ < include folder/section-name.qmd > }}
```

Incorrect (don't do this):

```
{{ < include folder/section-name.qmd > }}
```

The heading must be in the main file, followed by a blank line, then the include statement.

17.3.1.4 File Naming Conventions

- Main chapter files: `##-chapter-name.qmd` (e.g., `05-coding-practices.qmd`)
- Subdirectory: `##-chapter-name/` (matches the main file name)
- Include files: `descriptive-section-name.qmd` using kebab-case
- Use descriptive names that clearly indicate the content
- Prefix with underscore `_` for partial/helper files not directly included (e.g., `_lintr-summary.qmd`)

17.3.1.5 Git History Benefits Example

When reordering sections:

```
-## Object naming
+## Function calls

-{{< include demo-folder/section-name.qmd >}}
+{{< include demo-folder/section-2.qmd >}}

-## Function calls
+## Object naming

-{{< include demo-folder/section-2.qmd >}}
+{{< include demo-folder/section-name.qmd >}}
```

This diff clearly shows a reordering (swapping two sections) with no content changes—only the main chapter file changes.
When editing content: Only the specific include file (e.g., `05-coding-practices/function-calls.qmd`) appears in the git diff, making it easy to review the actual content changes without distraction.

17.3.1.6 When to Create a New Include File

Create a new include file when:

- Adding a new section to a chapter

- A section becomes long enough to benefit from being in its own file (>20-30 lines)
- Content might be reused elsewhere
- You want to work on a section independently

17.3.1.7 Migration Strategy

When working with chapters that don't yet use includes:

1. Create a subdirectory matching the chapter name
2. Extract each section into its own include file
3. Update the main chapter file to use includes
4. Keep headings in the main file
5. Ensure blank lines before include statements
6. Test that rendering still works correctly

17.4 Working with DOCX Files

GitHub Copilot can read and process Microsoft Word (.docx) files, which is useful for translating edits made in Word back to Quarto format.
When working with DOCX files:

- **Always examine tracked changes**: Use the `view` tool to read DOCX files and pay special attention to any tracked changes (insertions, deletions, formatting changes)
- **Review comments**: Look for and address any comments in the DOCX file that may provide context or instructions for edits
- **Translate edits to Quarto**: When edits have been made in a DOCX file, apply the equivalent changes to the corresponding `.qmd` files
- **Preserve formatting**: Ensure that formatting, citations, and cross-references are properly converted to Quarto/markdown syntax
- **Verify completeness**: Check that all edits, including those in tracked changes and comments, have been addressed

This workflow enables a hybrid editing process where collaborators can make edits in familiar Word format, and Copilot can translate those edits back to the Quarto source files.

17.5 Additional Guidelines

- Maintain consistency with existing code style
- Preserve all existing content when refactoring
- Add blank lines before all lists
- Follow the lab's R package development workflow (as described throughout this repo)

[a] https://quarto.org/docs/authoring/cross-references.html

18 Checklists

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung[1]

18.1 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

18.2 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[2]?
- Are all warnings ignorable? Should any warnings be intentionally suppressed or addressed?

18.3 Manuscript checklist

This is adapted in part from this article[3].

[1] https://jadebc.github.io/lab-manual/checklists.html
[2] https://ucd-serg.github.io/lab-manual/coding-style.html
[3] https://www.nature.com/articles/d41586-019-01431-z

- Have you completed the relevant reporting checklist, if applicable? (Collection of checklists[4])
- 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[5])
- Have you drafted author contributions? Do they follow the CRediT Taxonomy[6] for author contributions?

18.4 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[7], a neat palette generator with colorblind options here[8], and an article on why this matters here[9]

[4] https://www.equator-network.org/about-us/what-is-a-reporting-guideline/
[5] https://journals.lww.com/epidem/Fulltext/2001/05000/The_Value_of_P.2.aspx
[6] https://journals.plos.org/plosone/s/authorship/?utm_source=plos&utm_medium=blog&utm_campaign=plos-1607-credit#loc-author-contributions
[7] http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/#a-colorblind-friendly-palette
[8] https://medialab.github.io/iwanthue/?utm_source=Nature+Briefing&utm_campaign=2c68711076-briefing-dy-20211006&utm_medium=email&utm_term=0_c9dfd39373-2c68711076-44335685
[9] https://www.nature.com/articles/d41586-021-02696-z

- 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?

19 Resources

Adapted by UCD-SeRG team from original by Jade Benjamin-Chung and Kunal Mishra[1]

19.1 Resources for R

19.1.1 Books and Comprehensive Guides

- R for Data Science[2] by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund - comprehensive introduction to doing data science with R
- R Packages[3] by Hadley Wickham and Jenny Bryan - complete guide to R package development
- Advanced R[4] by Hadley Wickham - deep dive into R programming and internals
- Mastering Shiny[5] by Hadley Wickham - comprehensive guide to building web applications with Shiny
- Engineering Production-Grade Shiny Apps[6] by Colin Fay, Sébastien Rochette, Vincent Guyader, and Cervan Girard - best practices for production Shiny applications
- Happy Git and GitHub for the useR[7] by Jenny Bryan - guide to using Git and GitHub with R
- Jade's R-for-epi course[8]

19.1.2 Cheat Sheets

- dplyr and tidyr cheat sheet[9]
- ggplot cheat sheet[10]
- data table cheat sheet[11]
- RMarkdown cheat sheet[12]

19.1.3 Style and Best Practices

- Hadley Wickham's R Style Guide[13]

[1] https://jadebc.github.io/lab-manual/resources.html
[2] https://r4ds.hadley.nz/
[3] https://r-pkgs.org/
[4] https://adv-r.hadley.nz/
[5] https://mastering-shiny.org/
[6] https://engineering-shiny.org/
[7] https://happygitwithr.com/
[8] https://ucb-epi-r.github.io
[9] https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf
[10] https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf
[11] https://s3.amazonaws.com/assets.datacamp.com/blog_assets/datatable_Cheat_Sheet_R.pdf
[12] https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf
[13] http://adv-r.had.co.nz/Style.html

19.1.4 Tidy Evaluation Resources

- Tidy Eval in 5 Minutes[14] (video)
- Tidy Evaluation[15] (e-book)
- Data Frame Columns as Arguments to Dplyr Functions[16] (blog)
- Standard Evaluation for *_join[17] (stackoverflow)
- Programming with dplyr[18] (package vignette)

19.2 Resources for Git & Github

- Happy Git and GitHub for the useR[19] by Jenny Bryan - comprehensive guide to using Git and GitHub with R
- Data Camp introduction to Git[20]
- Introduction to Github[21]

19.3 Scientific figures

- Ten Simple Rules for Better Figures[22]

19.4 Writing

- Tips on how to write a great science paper[23]
- ICMJE Definition of authorship[24]
- Nature article on elements of style for scientific writing[25]
- The Pathway to Publishing: A Guide to Quantitative Writing in the Health Sciences[26]
- Secret, actionable writing tips[27]

19.5 Presentations

- How to tell a compelling story in scientific presentations[28]
- How to give a killer narratively-driven scientific talk[29]

[14] https://www.youtube.com/watch?v=nERXS3ssntw
[15] https://tidyeval.tidyverse.org/index.html
[16] https://www.brodrigues.co/blog/2016-07-18-data-frame-columns-as-arguments-to-dplyr-functions/
[17] https://stackoverflow.com/questions/28125816/r-standard-evaluation-for-join-dplyr
[18] https://dplyr.tidyverse.org/articles/programming.html
[19] https://happygitwithr.com/
[20] https://www.datacamp.com/courses/introduction-to-git-for-data-science
[21] https://lab.github.com/githubtraining/introduction-to-github
[22] https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003833
[23] https://www.nature.com/articles/d41586-019-02918-5
[24] http://www.icmje.org/recommendations/browse/roles-and-responsibilities/defining-the-role-of-authors-and-contributors.html
[25] https://www.nature.com/articles/nphys724
[26] https://link.springer.com/book/10.1007/978-3-030-98175-4
[27] https://x.com/acagamic/status/1680381737816424450
[28] https://www.nature.com/articles/d41586-021-03603-2
[29] https://www.sciencedirect.com/science/article/pii/S1471491423000928

- How to make a better poster[30]
- How to make an even better poster[31]

19.6 Professional advice

- Professional advice, especially for your first job[32]

19.7 Funding

- Building Your Funding Train[33]
- NIH Grant Writing Resources[34]

19.8 Ethics and global health research

- Global Code of Conduct For Research in Resource-Poor Settings[35]
- Who is a global health expert? Advice for aspiring global health experts[36]
- Transforming Global Health Partnerships[37]

[30] https://www.youtube.com/watch?v=1RwJbhkCA58&t=1171s
[31] https://mitcommlab.mit.edu/be/2023/09/27/toward-an-evenbetterposter-improving-the-betterposter-template/
[32] https://docs.google.com/document/d/1ckgRCcr7FFPyymyMA6Y_-cB_vftOyrrxT28c6gRg0PI/edit
[33] https://grantwriting.stanford.edu/funding-train/#ep
[34] https://www.niaid.nih.gov/grants-contracts/write-grant-application
[35] https://www.globalcodeofconduct.org/wp-content/uploads/2018/05/Global-Code-of-Conduct-Brochure.pdf
[36] https://journals.plos.org/globalpublichealth/article?id=10.1371/journal.pgph.0002269
[37] https://link.springer.com/book/9783031537929

References

"2001: A Space Odyssey." 1968. Film. https://en.wikipedia.org/wiki/2001:_A_Space_Odyssey_(film).

Asimov, Isaac. 1950. "I, Robot." New York: Novel; Gnome Press. https://search.library.ucdavis.edu/permalink/01UCD_INST/9fle3i/alma9900002263504031 26.

"Blade Runner." 1982. Film. https://en.wikipedia.org/wiki/Blade_Runner.

Card, Orson Scott. 1985. "Ender's Game." Novel; Tor Books. https://en.wikipedia.org/wiki/Ender%27s_Game.

Flight of the Conchords. 2007. "The Humans Are Dead." Music Video. https://www.youtube.com/watch?v=B1BdQcJ2ZYY.

Herbert, Frank. 1965. "Dune." Novel. https://en.wikipedia.org/wiki/Dune_(franchise)#Butlerian_Jihad.

"The Matrix." 1999. Film. https://en.wikipedia.org/wiki/The_Matrix.

"The Terminator." 1984. Film. https://en.wikipedia.org/wiki/The_Terminator.

"WarGames." 1983. Film. https://en.wikipedia.org/wiki/WarGames.

Wickham, Hadley, Peter Danenberg, Gábor Csárdi, and Manuel Eugster. 2024. *Roxygen2: In-Line Documentation for r.* https://roxygen2.r-lib.org/.