

Lab Manual

Jade Benjamin-Chung, Kunal Mishra, Stephanie Djajadi

2019-08-27

Contents

1	Welcome to our lab!	5
2	Communication and coordination	7
2.1	Slack	7
2.2	Email	7
2.3	Trello	8
2.4	Google Drives	8
2.5	Google Calendar / Meetings	8
3	Code of conduct	9
3.1	Lab culture	9
3.2	Protecting human subjects	9
3.3	Authorship	10
3.4	Logging hours	10
4	Code repositories	11
4.1	Project Structure	11
4.2	.Rproj files	12
4.3	Configuration ('config') File	12
4.4	Order Files and Directories	13
4.5	Using Bash scripts to ensure reproducibility	13
5	Coding practices	15
5.1	Organizing scripts	15
5.2	Documenting your code	15
5.3	Object naming	17
5.4	Function calls	18
5.5	The here package	18
5.6	Reading/Saving Data	18
5.7	Tidyverse	19
5.8	Coding with R and Python	21
6	Coding style	23
6.1	Comments	23

6.2	Line breaks	24
6.3	Automated Tools for Style and Project Workflow	26
7	Working with Big Data	29
7.1	The data.table package	29
7.2	Using downsampled data	30
7.3	Optimal RStudio set up	30
8	Github	31
8.1	How often should I commit?	31
8.2	What should be pushed to github?	31
9	Unix commands	33
9.1	Basics	33
9.2	Syntax for both Mac/Windows	33
9.3	Basics	34
9.4	Running scripts	34
9.5	Checking tasks and killing jobs	34
9.6	Running big jobs	34
10	Resources	37
10.1	Resources for R	37
10.2	Resources for Github	37
10.3	Authorship	37

Chapter 1

Welcome to our lab!

We are a team of epidemiologists and biostatisticians engaged in global health research. This lab manual covers our communication strategy and code of conduct and goes into detail about best practices for data science. It is a living document that is updated regularly.

Chapter 2

Communication and coordination

by Jade Benjamin-Chung

These communications guidelines are evolving as we increasingly adopt Slack, but here some general principles:

2.1 Slack

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

2.2 Email

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

- Generally, strive to respond within 24 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.

2.3 Trello

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

2.4 Google Drives

- We mostly use Google Drive to create shared documents with longer descriptions of tasks. These documents are linked to in Trello. Jade often shares these with the whole team since tasks are overlapping, and even if a task is assigned to one person, others may have valuable insights.
- Please invite both of Jade’s email addresses to any documents you create (jadebc@gmail.com, jadebc@berkeley.edu).

2.5 Google Calendar / Meetings

- We use Google Calendar to set meetings. Please make sure your calendar is set up correctly because sometimes you might not receive a specific email or Slack message about it – only a Google Calendar invitation.
- Our meetings start on the hour, not on Berkeley time.
- If you are going to be late, please send a message in our Slack channel.
- If you are regularly not able to come on the hour, notify the team and we might choose to modify the agenda order or the start time.

Chapter 3

Code of conduct

3.1 Lab culture

It goes without saying that we strive to work in an environment that is collaborative, supportive, open, and free from discrimination and harassment, per University policies.

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.

3.2 Protecting human subjects

All lab members must complete CITI Human Subjects Biomedical Group 1 training and share their certificate with Jade. She will add team members to relevant Institutional Review Board protocols prior to their start date to ensure they have permission to work with identifiable datasets.

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

- If you are using a virtual computer (e.g., Bluevelvet, AWS, GHAP), never save the data in that system to your personal computer or any other computer without prior permission.
- Do not share data with anyone without permission, including to other members of the group, who might not be on the same IRB protocol as

you (check with Jade first).

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

3.3 Authorship

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

3.4 Logging hours

Please use Caltime to log your hours. If you have a non-exempt appointment (this is the default), you need to punch in when you start working and punch out when you stop working. The particular hours /days when you work are not important; rather, we monitor your total hours in a 2-week period. If you have trouble remembering to punch in/out, please devise a system that works for you (i.e., set timers / reminders). Please avoid missing punches, and if you do, please send Jade a Slack message with the time and date you intended to punch out.

If you have an exempt appointment (this is the case if you also have a teaching appointment), you do not need to punch in / out. You will be expected to work, on average, for a certain number of hours per week.

You may log hours for data science team meetings. You may not log hours for Colford-Hubbard Research Group meetings or commute time, if applicable.

Chapter 4

Code repositories

By Kunal Mishra and Jade Benjamin-Chung

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.

4.1 Project Structure

We recommend the following directory structure:

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

```

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

```

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

4.2 .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.

4.3 Configuration (‘config’) File

This is the single most important file for your project. It will be responsible for a variety of common tasks, declare global variables, load functions, declare paths, and more. *Every other file in the project* will begin with `source("0-config")`, and its role is to reduce redundancy and create an abstraction layer that allows you to make changes in one place (`0-config.R`) rather than 5 different

files. To this end, paths which will be reference in multiple scripts (i.e. a `merged_data_path`) can be declared in `0-config.R` and simply referred to by its variable name in scripts. If you ever want to change things, rename them, or even switch from a downsample to the full data, all you would then 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.

4.4 Order Files and Directories

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

4.5 Using Bash scripts to ensure reproducibility

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

Chapter 5

Coding practices

by Kunal Mishra and Jade Benjamin-Chung

5.1 Organizing scripts

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

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

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

5.2 Documenting your code

5.2.1 File headers

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

#####

```
# @Organization - Example Organization    # @Project - Example
Project    # @Description - This file is responsible for [...]
#####
- **Note**: If your computer isn't able to handle this workflow due to RAM or requirements
```

5.2.2 Comments in the body of your script

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

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

5.2.3 Function documentation

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

```
#####
#####
# Documentation: calc_fluseas_mean
# Usage: calc_fluseas_mean(data, yname)
# Description: Make a dataframe with rows for flu season and site
# and the number of patients with an outcome, the total patients,
# and the percent of patients with the outcome

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

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

calc_fluseas_mean = function(data, yname, silent = TRUE) {
```



```
### function code here

}
```

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

- **Note:** As someone trying to call a function, it is possible to access a function’s documentation (and internal code) by **CMD-Left-Clicking** 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).

5.3 Object naming

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

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

We recommend you use **Snake_Case**. - Base R allows `.` in variable names and functions (such as `read.csv()`), but this goes against best practices for variable naming in many other coding languages. For consistency’s sake, `snake_case` has been adopted across languages, and modern packages and functions typically use it (i.e. `readr::read_csv()`). As a very general rule of thumb, if a package you’re using doesn’t use `snake_case`, there may be an updated version or more modern package that *does*, bringing with it the variety of performance improvements and bug fixes inherent in more mature and modern software.

- **Note:** you may also see `camelCase` throughout the R code you come across. This is *okay* but not ideal – try to stay consistent across all your

code with `snake_case`.

- **Note:** again, its also worth noting there's nothing inherently wrong with using `.` in variable names, just that it goes against style best practices that are cropping up in data science, so its worth getting rid of these bad habits now.

5.4 Function calls

In a function call, use “named arguments” and separate arguments by `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
)
```

5.5 The here package

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

5.6 Reading/Saving Data

5.6.1 .RDS vs .RData Files

One of the most common ways to load and save data in Base R is with the `load()` and `save()` functions to serialize multiple objects in a single `.RData` file.

The biggest problems with this practice include an inability to control the names of things getting loaded in, the inherent confusion this creates in understanding older code, and the inability to load individual elements of a saved file. For this, we recommend using the RDS format to save R objects.

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

5.6.2 CSVs

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

5.7 Tidyverse

Throughout this document there have been references to the Tidyverse, but this section is to explicitly show you how to transform your Base R tendencies to Tidyverse (or Data.Table, Tidyverse's performance-optimized competitor). For most of our work that does not utilize very large datasets, we recommend that you code in Tidyverse rather than Base R. Tidyverse is quickly becoming the gold standard in R data analysis and modern data science packages and code should use Tidyverse style and packages unless there's a significant reason not to (i.e. big data pipelines that would benefit from Data.Table's performance optimizations).

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

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

Base R	Better Style, Performance, and Utility
<code>read.csv()</code>	<code>readr::read_csv()</code> or <code>data.table::fread()</code>
<code>write.csv()</code>	<code>readr::write_csv()</code> or <code>data.table::fwrite()</code>

Base R	Better Style, Performance, and Utility
<code>readRDS</code> <code>saveRDS()</code>	<code>readr::read_rds()</code> <code>readr::write_rds()</code>
<code>data.frame()</code>	<code>tibble::tibble()</code> or <code>data.table::data.table()</code>
<code>rbind()</code> <code>cbind()</code> <code>df\$some_column</code>	<code>dplyr::bind_rows()</code> <code>dplyr::bind_cols()</code> <code>df %>%</code>
<code>df\$some_column = ...</code>	<code>dplyr::pull(some_column)</code> <code>df %>%</code> <code>dplyr::mutate(some_column = ...)</code>
<code>df[get_rows_condition,]</code>	<code>df %>%</code> <code>dplyr::filter(get_rows_condition)</code>
<code>df[,c(col1, col2)]</code>	<code>df %>% dplyr::select(col1, col2)</code>
<code>merge(df1, df2, by = ..., all.x = ..., all.y = ...)</code>	<code>df1 %>% dplyr::left_join(df2, by = ...)</code> or <code>dplyr::full_join</code> or <code>dplyr::inner_join</code> or <code>dplyr::right_join</code>
<code>str()</code> <code>grep(pattern, x)</code>	<code>dplyr::glimpse()</code> <code>stringr::str_which(string, pattern)</code>
<code>gsub(pattern, replacement, x)</code>	<code>stringr::str_replace(string, pattern, replacement)</code>
<code>ifelse(test_expression, yes, no)</code>	<code>if_else(condition, true, false)</code>
Nested: <code>ifelse(test_expression1, yes1, ifelse(test_expression2, yes2, ifelse(test_expression3, yes3, no)))</code>	<code>case_when(test_expression1 ~ yes1, test_expression2 ~ yes2, test_expression3 ~ yes3, TRUE ~ no)</code>
<code>proc.time()</code> <code>stopifnot()</code>	<code>tictoc::tic()</code> and <code>tictoc::toc()</code> <code>assertthat::assert_that()</code> or <code>assertthat::see_if()</code> or <code>assertthat::validate_that()</code>

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

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

recommend checking out the following resources:

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

5.8 Coding with R and Python

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

Chapter 6

Coding style

by Kunal Mishra and Jade Benjamin-Chung

6.1 Comments

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

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

2. **File Structure** - Just as your data “flows” through your project, data should flow naturally through a script. Very generally, you want to 1) source your config => 2) load all your data => 3) do all your analysis/computation => save your data. Each of these sections should be “chunked together” using comments. See this file for a good example of how to cleanly organize a file in a way that follows this “flow” and functionally separate pieces of code that are doing different things.

- **Note:** If your computer isn’t able to handle this workflow due to RAM or requirements, modifying the ordering of your code to 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.

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

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

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

6.2 Line breaks

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

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

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

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


And of a complex `ggplot` call:

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

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

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

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

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

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

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

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

6.3 Automated Tools for Style and Project Workflow

6.3.1 Styling

1. **Code Autoformatting** - RStudio includes a fantastic built-in utility (keyboard shortcut: **CMD-Shift-A**) for autoformatting highlighted chunks of code to fit many of the best practices listed here. It generally makes code more readable and fixes a lot of the small things you may not feel like fixing yourself. Try it out as a “first pass” on some code of yours that *doesn't* follow many of these best practices!
2. **Assignment Aligner** - A cool R package allows you to very powerfully format large chunks of assignment code to be much cleaner and much more readable. Follow the linked instructions and create a keyboard shortcut of your choosing (recommendation: **CMD-Shift-Z**). Here is an example of how assignment aligning can dramatically improve code readability:

Before

```

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

```

After

```

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

```

6.3. AUTOMATED TOOLS FOR STYLE AND PROJECT WORKFLOW 27

```
"PLACE @ Prescott"           = "Preparatory Literary Academy of Cultural Excellence",  
"RISE Community School"      = str_subset(string = OUSD_school_shapes$schnam, pattern =  
)
```

3. **StyleR** - Another cool R package from the Tidyverse that can be powerful and used as a first pass on entire projects that need refactoring. The most useful function of the package is the `style_dir` function, which will style all files within a given directory. See the function's documentation and the vignette linked above for more details.

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

Chapter 7

Working with Big Data

by Kunal Mishra and Jade Benjamin-Chung

7.1 The `data.table` package

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

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

7.2 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.

7.3 Optimal RStudio set up

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

Workspace

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

History

- **Uncheck** Always save history

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

Chapter 8

Github

by Stephanie Djajadi and Nolan Pokpongkiat

**** Stephanie please add your table ****

8.1 How often should I commit?

8.2 What should be pushed to github?

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

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

Chapter 9

Unix commands

by Stephanie Djajadi and Kunal Mishra

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

9.1 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.

Stephanie please add your figure then delete this

9.2 Syntax for both Mac/Windows

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

Stephanie please add your table and figure then delete this

9.3 Basics

Stephanie please add your table then delete this

9.4 Running scripts

Stephanie please reformat below `Rscript file.R R CMD BATCH file.R` #
output will be in a new file called `file.Rout` `cat file.Rout` # check output <http://datacornering.com/how-to-run-r-scripts-from-the-windows-command-line-cmd/>
<https://happygitwithr.com/shell.html>

9.5 Checking tasks and killing jobs

Stephanie please add your table then delete this

9.6 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`.
- **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. For help and documentation, you can use the command `./runFileSaveLogs -h`.

Chapter 10

Resources

by Jade Benjamin-Chung and Kunal Mishra

10.1 Resources for R

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

10.2 Resources for Github

10.3 Authorship

- [ICMJE Definition of authorship](#)