CSSS508, Lecture 10

Model Results and Reproducibility

Michael Pearce
(based on slides from Chuck Lanfear)

December 8, 2022

Topics

Last time, we learned about,

1. `ggmap` for mapping in `ggplot2`
2. Density plots
3. Labeling points

Today, we will cover,

1. Working with Model Results
    - Tidy model output with `broom`
    - Visualizing models with `geom_smooth`
    - Tables with `pander`
2. Reproducible research
3. Best practices
4. Wrapping up the course!

Working with Model Results

broom

`broom` is a package that "tidies up" the output from models such a `lm()` and `glm()`.

It has a small number of key functions:

- `tidy()` - Creates a dataframe summary of a model.
- `augment()` - Adds columns—such as fitted values—to the data used in the model.
- `glance()` - Provides one row of fit statistics for models.

```
library(broom)
```

Model Output is a List

`lm()` and `summary()` produce lists as output, which **cannot** go directly into tidyverse functions, like `ggplot`.

```
lm_1 <- lm(yn ~ num1 + fac1, data = ex_dat)
summary(lm_1)
```

```
##
## Call:
## lm(formula = yn ~ num1 + fac1, data = ex_dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -12.354  -1.803  -0.211   1.995   6.994
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   0.4024     0.3924   1.025   0.3064
## num1          0.6464     0.1055   6.128 4.80e-09 ***
## fac1B         1.0415     0.5164   2.017   0.0451 *
## fac1C         2.9500     0.5064   5.825 2.31e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.906 on 196 degrees of freedom
## Multiple R-squared:  0.2767,    Adjusted R-squared:  0.2657
## F-statistic:    25 on 3 and 196 DF,  p-value: 9.729e-14
```

Model Output Varies!

Different models --> Different outputs! You can't reuse the same code to handle output from every model.

```
glm_1 <- glm(yb ~ num1 + fac1, data = ex_dat, family=binomial(link="logit"))
summary(glm_1)
```

```
##
## Call:
## glm(formula = yb ~ num1 + fac1, family = binomial(link = "logit"),
##     data = ex_dat)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.9021  -1.0383   0.4636   0.9609   2.0082
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.12865    0.31448  -3.589 0.000332 ***
## num1         0.39370    0.08903   4.422 9.77e-06 ***
## fac1B        0.59368    0.38510   1.542 0.123166
## fac1C        1.47743    0.39745   3.717 0.000201 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 277.18  on 199  degrees of freedom
## Residual deviance: 239.93  on 196  degrees of freedom
## AIC: 247.93
##
## Number of Fisher Scoring iterations: 3
```

broom::tidy()

`tidy()` solve this problem, giving results as a dataframe.

```
lm_1 %>% tidy()
```

```
## # A tibble: 4 × 5
##   term        estimate std.error statistic       p.value
##   <chr>          <dbl>     <dbl>     <dbl>         <dbl>
## 1 (Intercept)    0.402     0.392      1.03 0.306
## 2 num1           0.646     0.105      6.13 0.00000000480
## 3 fac1B          1.04      0.516      2.02 0.0451
## 4 fac1C          2.95      0.506      5.83 0.0000000231
```

Each type of model (e.g. `glm`, `lmer`) has a different *method* with its own additional arguments. See `?tidy.lm` for an example.

broom::tidy()

This output is also completely identical between different models.

This can be very useful and important if running models with different test statistics... or just running a lot of models!

```
glm_1 %>% tidy()
```

```
## # A tibble: 4 × 5
##   term        estimate std.error statistic    p.value
##   <chr>          <dbl>     <dbl>     <dbl>      <dbl>
## 1 (Intercept)   -1.13     0.314      -3.59 0.000332
## 2 num1           0.394    0.0890      4.42 0.00000977
## 3 fac1B          0.594    0.385       1.54 0.123
## 4 fac1C          1.48     0.397       3.72 0.000201
```

broom::glance()

`glance()` similarly produces dataframes of fit statistics.

```
glance(lm_1)
```

```
## # A tibble: 1 × 12
##   r.squared adj.r.sq…¹ sigma stati…²  p.value    df logLik   AIC   BIC
##       <dbl>      <dbl> <dbl>   <dbl>    <dbl> <dbl>  <dbl> <dbl> <dbl>
## 1     0.277      0.266  2.91    25.0 9.73e-14     3  -495. 1000. 1017.
## # … with 3 more variables: deviance <dbl>, df.residual <int>,
## #   nobs <int>, and abbreviated variable names ¹adj.r.squared,
## #   ²statistic
```

`broom::augment()`

`augment()` takes values generated by a model and adds them back to the original data.

- E.g. fitted values, residuals, and leverage statistics.

```
augment(lm_1) %>% head()
```

```
## # A tibble: 6 × 9
##       yn  num1 fac1  .fitted .resid   .hat .sigma    .cooksd .std.re…¹
##    <dbl> <dbl> <fct>   <dbl>  <dbl>  <dbl>  <dbl>      <dbl>     <dbl>
## 1  1.17   1.37 A       1.29  -0.117 0.0168   2.91 0.00000707   -0.0407
## 2 -2.70  -1.13 A      -0.330 -2.37  0.0232   2.91 0.00406      -0.827
## 3  3.17   1.27 A       1.23   1.95  0.0167   2.91 0.00194       0.676
## 4  6.18   1.67 B       2.52   3.66  0.0154   2.90 0.00628       1.27
## 5  5.78   1.19 C       4.12   1.66  0.0137   2.91 0.00114       0.574
## 6  0.595 -1.48 C       2.40  -1.80  0.0231   2.91 0.00233      -0.627
## # … with abbreviated variable name ¹.std.resid
```

See `?augment.lm` for examples of what `augment()` can do.

Plotting Model Results

`geom_smooth()`

I have used `geom_smooth()` in many past examples.

`geom_smooth()` generates "smoothed conditional means" including loess curves and generalized additive models (GAMs).

Note, however, that most regression models are conditional mean models, such as ordinary least squares and generalized linear models.

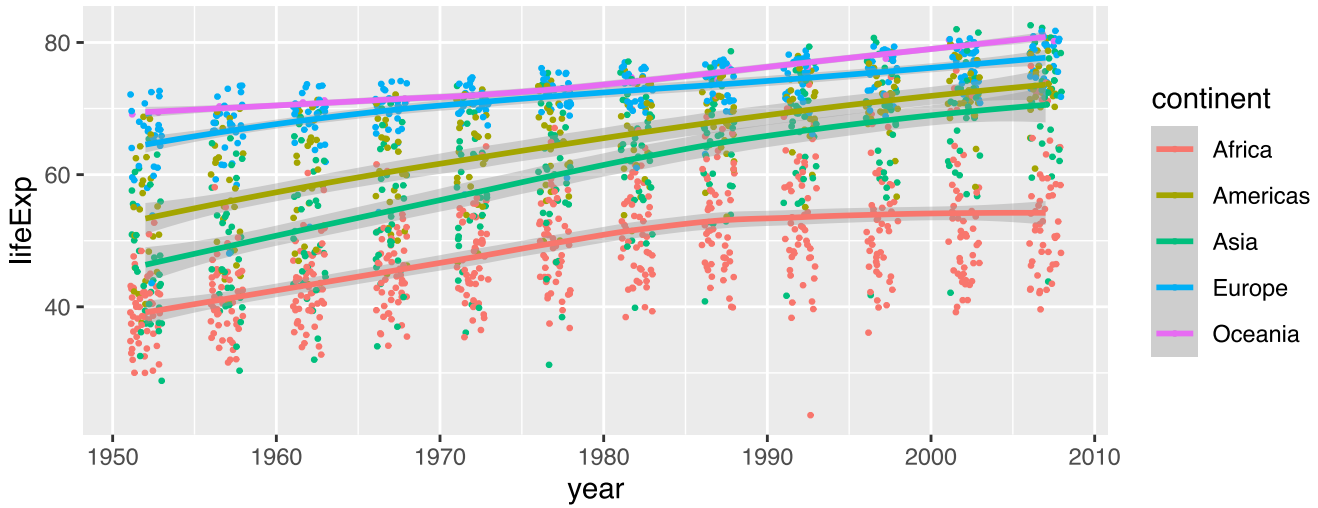
We can use `geom_smooth()` to add a layer depicting common bivariate models.

We'll look at this with the `gapminder` data from Week 2.

```
library(gapminder)
```

Default `geom_smooth()`

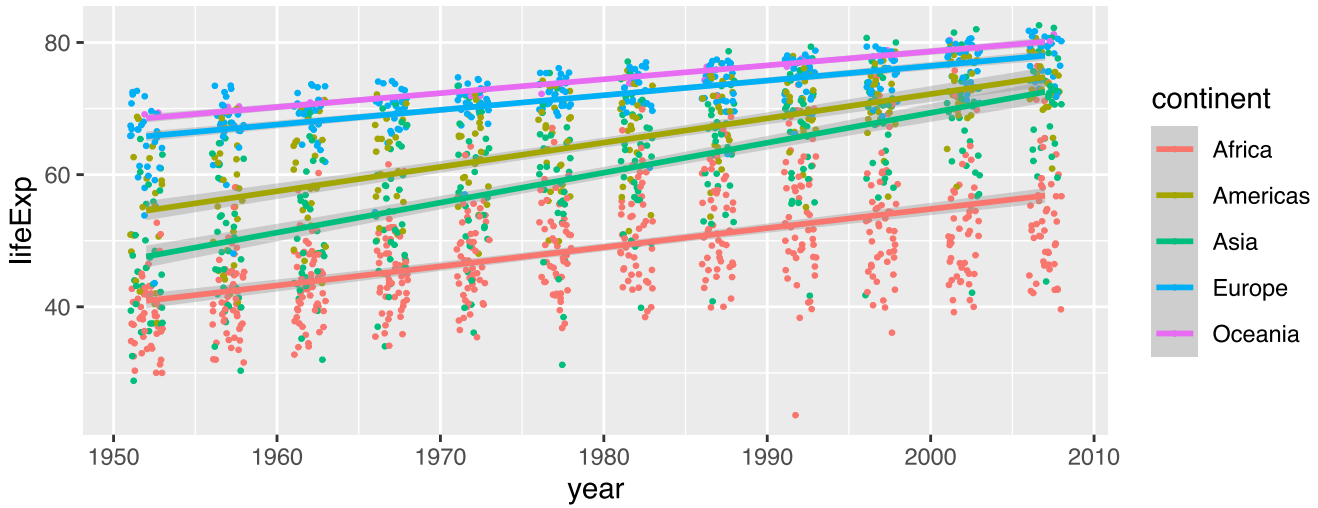
```
ggplot(data = gapminder,
       aes(x = year, y = lifeExp, color = continent)) +
  geom_point(position = position_jitter(1,0), size = 0.5) +
  geom_smooth()
```



By default, `geom_smooth()` chooses either a loess smoother (N < 1000) or a GAM depending on the number of observations.

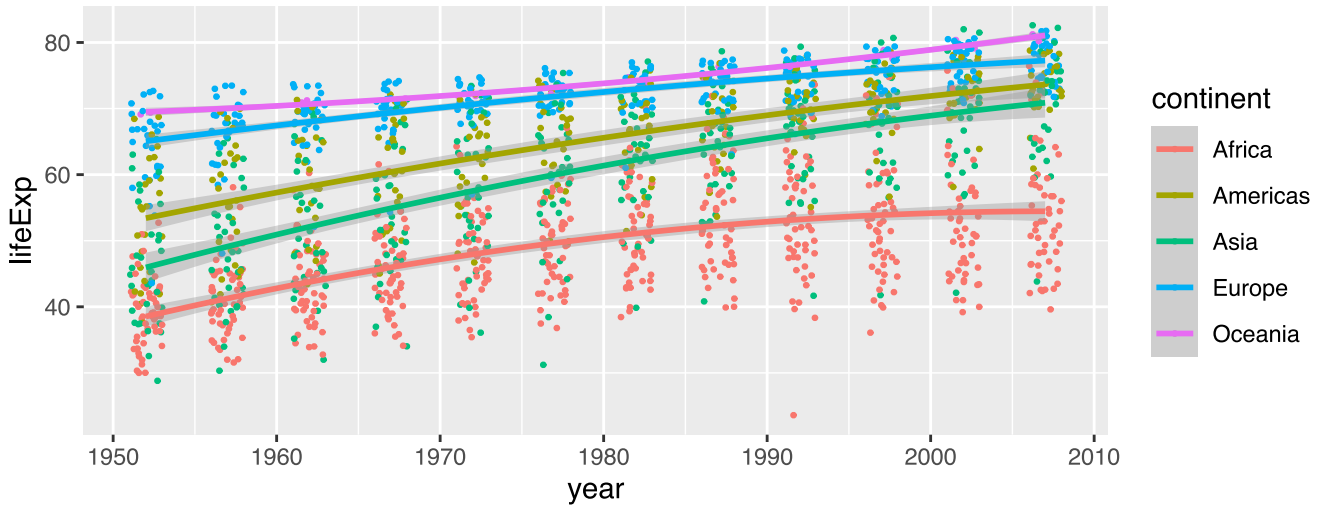
Linear `glm`

```
ggplot(data = gapminder,
       aes(x = year, y = lifeExp, color = continent)) +
  geom_point(position = position_jitter(1,0), size = 0.5) +
  geom_smooth(method = "glm", formula = y ~ x)
```



Polynomial `glm`

```
ggplot(data = gapminder,
       aes(x = year, y = lifeExp, color = continent)) +
  geom_point(position = position_jitter(1,0), size = 0.5) +
  geom_smooth(method = "glm", formula = y ~ poly(x, 2))
```



`poly(x, 2)` produces a quadratic model which contains a linear term (`x`) and a quadratic term (`x^2`).

Making Tables

`pander` Regression Tables

We've used `pander` to create nice tables for dataframes. But `pander` has *methods* to handle all sort of objects that you might want displayed nicely.

This includes model output, such as from `lm()`, `glm()`, and `summary()`.

```
library(pander)
```

`pander()` and `lm()`

You can send an `lm()` object straight to `pander`:

```
pander(lm_1)
```

	Estimate	Std. Error	t value	Pr(>t)
(Intercept)	37.23	1.599	23.28	2.565e-20
wt	-3.878	0.6327	-6.129	1.12e-06
hp	-0.03177	0.00903	-3.519	0.001451

Table: Fitting linear model: mpg ~ wt + hp

`pander()` and `summary()`

You can do this with `summary()` as well, for added information:

```
pander(summary(lm_1))
```

	Estimate	Std. Error	t value	Pr(>t)
(Intercept)	37.23	1.599	23.28	2.565e-20
wt	-3.878	0.6327	-6.129	1.12e-06
hp	-0.03177	0.00903	-3.519	0.001451

Observations	Residual Std. Error	R^2	Adjusted R^2
32	2.593	0.8268	0.8148

Table: Fitting linear model: mpg ~ wt + hp

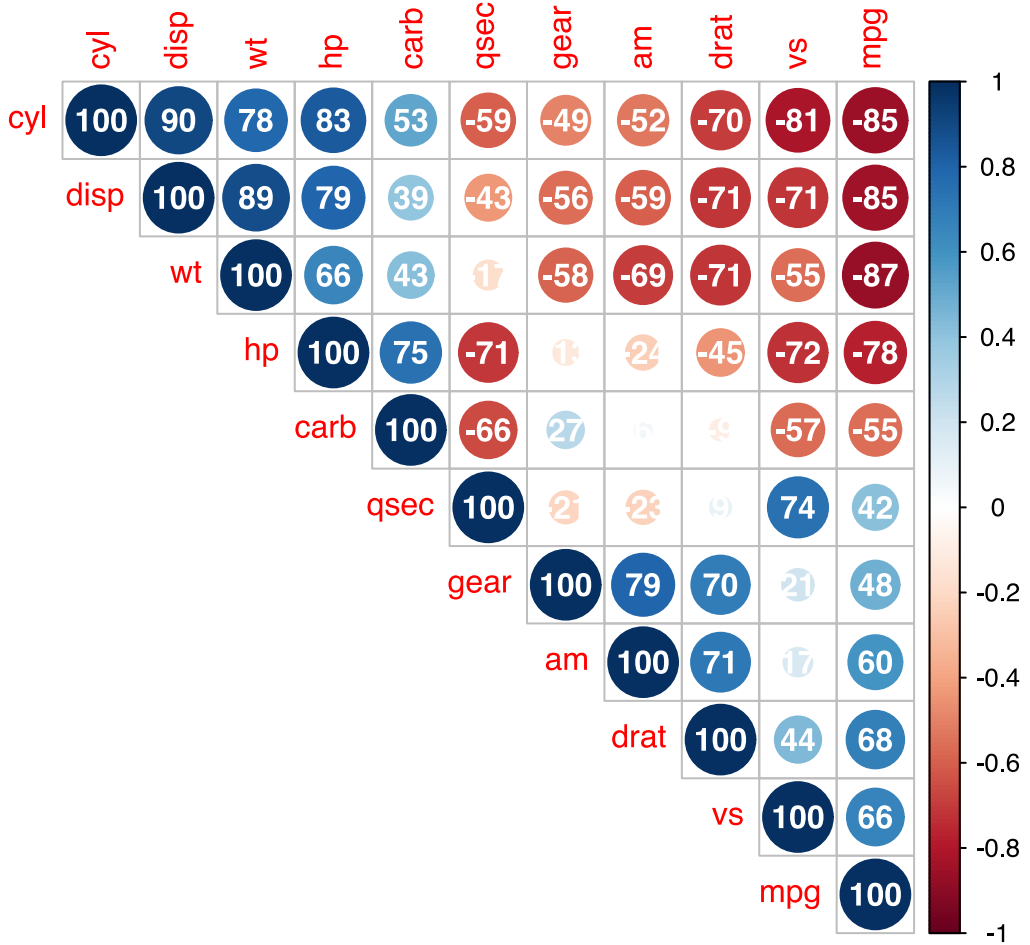
Bonus: `corrplot`

The `corrplot` package creates **correlograms**:

- Interpreting correlations easier than in a table.
- First argument is to call `cor()`, a base R function for calculating correlations.
- See the vignette for customization options.

```
library(corrplot)
corrplot(
  cor(mtcars),
  addCoef.col = "white",
  addCoefasPercent=T,
  type="upper",
  order="AOE")
```

Correlogram



Reproducible Research

Why Reproducibility?

Reproducibility is not *replication*.

- **Replication** is running a new study to show if and how results of a prior study hold.
- **Reproducibility** is about rerunning *the same study* and getting the *same results*.

Reproducible studies can still be *wrong*... and in fact reproducibility makes proving a study wrong *much easier*.

Reproducibility means:

- Transparent research practices.
- Minimal barriers to verifying your results.

Any study that isn't reproducible can be trusted only on faith.

Reproducibility Definitions

Reproducibility comes in three forms (Stodden 2014):

1. **Empirical:** Repeatability in data collection.
2. **Statistical:** Verification with alternate methods of inference.
3. **Computational:** Reproducibility in cleaning, organizing, and presenting data and results.

R is particularly well suited to enabling **computational reproducibility**.[1]

They will not fix flawed research design, nor offer a remedy for improper application of statistical methods.

Those are the difficult, non-automatable things you want skills in.

[1] Python is equally well suited.

Computational Reproducibility

Elements of computational reproducibility:

- **Shared data**
  - Researchers need your original data to verify and replicate your work.
- **Shared code**
  - Your code must be shared to make decisions transparent.
- **Documentation**
  - The operation of code should be either self-documenting or have written descriptions to make its use clear.
- **Version Control**
  - Documents the research process.
  - Prevents losing work and facilitates sharing.

Levels of Reproducibility

For academic papers, degrees of reproducibility vary:

1. "Read the article"
2. Shared data with documentation
3. Shared data and all code
4. **Interactive document**
5. **Research compendium**
6. Docker compendium: Self-contained ecosystem

Interactive Documents

Interactive documents—like R Markdown docs—combine code and text together into a self-contained document.

- Load and process data
- Run models
- Generate tables and plots in-line with text
- In-text values automatically filled in

Interactive documents allow a reader to examine your computational methods within the document itself; in effect, they are self-documenting.

By re-running the code, they reproduce your results on demand.

Common Platforms:

- **R:** R Markdown
- **Python:** Jupyter Notebooks

Research Compendia

A **research compendium** is a portable, reproducible distribution of an article or other project.

Research compendia feature:

- An interactive document as the foundation
- Files organized in a recognizable structure (e.g. an R package)
- Clear separation of data, method, and output. *Data are read only*.
- A well-documented or even *preserved* computational environment (e.g. Docker)

`rrtools` by UW's Ben Markwick provides a simplified workflow to accomplish this in R.

Bookdown

`bookdown`—which is integrated into `rrtools`—can generate documents in the proper format for articles, theses, books, or dissertations.

`bookdown` provides an accessible alternative to writing $\LaTeX$ for typesetting and reference management.

You can integrate citations and automate reference page generation using bibtex files (such as produced by Zotero).

`bookdown` supports `.html` output for ease and speed and also renders `.pdf` files through $\LaTeX$ for publication-ready documents.

For University of Washington theses and dissertations, consider Ben Marwick's `huskydown` package which uses Markdown but renders via a UW approved $\LaTeX$ template.

Best Practices

Organization Systems

Organizing research projects is something you either do accidentally—and badly—or purposefully with some upfront labor.

Uniform organization makes switching between or revisiting projects easier.

I suggest something like the following:

```
project/
   readme.md
   data/
     derived/
       processed_data.RData
     raw/
       core_data.csv
   docs/
     paper.Rmd
   syntax/
     functions.R
     models.R
```

1. There is a clear hierarchy
   - Written content is in `docs`
   - Code is in `syntax`
   - Data is in `data`
2. Naming is uniform
   - All lower case
   - Words separated by underscores
3. Names are self-descriptive

Workflow versus Project

To summarize Jenny Bryan, one should separate workflow from projects.

Workflow

- The software you use to write your code (e.g. RStudio)
- The location you store a project
- The specific computer you use
- The code you ran earlier or typed into your console

Project

- The raw data
- The code that operates on your raw data
- The packages you use
- The output files or documents

Projects *should not modify anything outside of the project* nor need to be modified by someone else (or future you) to run.

Projects *should be independent of your workflow*.

Portability

For research to be reproducible, it must also be *portable*. Portable software operates *independently of workflow* such as fixed file locations.

Do Not:

- Use `setwd()` in scripts or .Rmd files.
- Use *absolute paths* except for *fixed, immovable sources* (secure data).
  - `read_csv("C:/my_project/data/my_data.csv")`
- Use `install.packages()` in script or .Rmd files.
- Use `rm(list=ls())` anywhere but your console.

Do:

- Use RStudio projects (or the `here` package) to set directories.
- Use *relative paths* to load and save files:
  - `read_csv("./data/my_data.csv")`
- Load all required packages using `library()`.
- Clear your workspace when closing RStudio.
  - Set *Tools* > *Global Options...* > *Save workspace...* to **Never**

Divide and Conquer

Often you do not want to include all code for a project in one `.Rmd` file:

- The code takes too long to knit.
- The file is so long it is difficult to read.

There are two ways to deal with this:

1. Use separate `.R` scripts or `.Rmd` files which save results from complicated parts of a project, then load these results in the main `.Rmd` file.
    - This is good for loading and cleaning large data.
    - Also for running slow models.
2. Use `source()` to run external `.R` scripts when the `.Rmd` knits.
    - This can be used to run large files that aren't impractically slow.
    - Also good for loading project-specific functions.

Tools

Some opinionated advice

On Formats

Avoid "closed" or commercial software and file formats except where absolutely necessary.

Use open source software and file formats.

- It is always better for *science*:
    - People should be able to explore your research without buying commercial software.
    - You do not want your research to be inaccessible when software is updated.
- It is often just *better*.
    - It is usually updated more quickly
    - It tends to be more secure
    - It is rarely abandoned

The ideal: Use software that reads and writes *raw text*.

On Text

Writing and formatting documents are two completely separate jobs.

- Write first
- Format later
- Markdown was made for this

Word processors—like Microsoft Word—try to do both at the same time, usually badly.

They waste time by leading you to format instead of writing.

Find a good modular text editor and learn to use it:

- Atom
- Sublime (Commercial)
- Emacs
- Vim

On Version Control

Version control originates in collaborative software development.

The Idea: All changes ever made to a piece of software are documented, saved automatically, and revertible.

Version control allows all decisions ever made in a research project to be documented automatically.

Version control can:

1. Protect your work from destructive changes
2. Simplify collaboration by merging changes
3. Document design decisions
4. Make your research process transparent

Git and GitHub

`git` is the dominant platform for version control, and GitHub is a free (and now Microsoft owned) platform for hosting **repositories**.

Repositories are folders on your computer where all changes are tracked by Git.

Once satisfied with changes, you "commit" them then "push" them to a remote repository that stores your project.

Others can copy your project ("pull"), and if you permit, make suggestions for changes.

Constantly committing and pulling changes automatically generates a running "history" that documents the evolution of a project.

`git` is integrated into RStudio under the *Tools* menu. It requires some setup.[1]

[1] You can also use the GitHub desktop application.

GitHub as a CV

Beyond archiving projects and allowing sharing, GitHub also serves as a sort of curriculum vitae for the programmer.

By allowing others to view your projects, you can display competence in programming and research.

If you are planning on working in the private sector, an active GitHub profile will give you a leg up on the competition.

If you are aiming for academia, a GitHub account signals technical competence and an interest in research transparency.

Wrapping up the Course

What You've Learned

A lot!

- How to get data into R from a variety of formats
- How to do "data custodian" work to manipulate and clean data
- How to make pretty visualizations
- How to automate with loops and functions
- How to combine text, calculations, plots, and tables into dynamic R Markdown reports
- How to acquire and work with spatial data

You all are now **R**ockstars!!

What Comes Next?

- **Learn more statistics!! (e.g. take more CSSS courses)**
  - Learn foundations to statistical inference, create and evaluate models, consider survey design, make fancy visualizations, etc.
  - All of this is much easier to do if you already know R!
- **Practice, practice, practice!**
  - Replicate analyses you've done for practice (maybe in another language)
  - Think about data using `dplyr` verbs, tidy data principles
  - R Markdown for reproducibility
- **Do more advanced projects**
  - Use version control (git) in RStudio
  - Create interactive Shiny web apps
  - Write your own functions and put them in a package

Course Plugs

If you...

- would like to review math - **CSSS 505: Review of Math for Social Scientists**
- have no stats background yet - **SOC 504: Applied Social Statistics**
- want to learn some stat theory - **CSSS 510: Maximum Likelihood**
- want to master visualization - **CSSS 569: Visualizing Data**
- study events or durations - **CSSS 544: Event History Analysis**
- want to use network data - **CSSS 567: Social Network Analysis**
- want to work with spatial data - **CSSS 554: Spatial Statistics**
- want to work with time series - **CSSS 512: Time Series and Panel Data**

Thank you!

- Please submit your course evals! I *greatly appreciate* any feedback you may have.
- Remember to submit your final assignment (HW 8; due now!) and provide peer review feedback by Tuesday.
- Hand in (optional) HW 9 if you are short of the 20 points necessary to pass.
- Feel free to reach out at any point in the future with questions or comments!