Introduction to R

Week 6: Analyze your data

Louisa Smith

August 17 - August 21

Let's put it
ALL
together

Agenda

Week 1: The basics ✅

Week 2: Figures ✅

Week 3: Selecting, filtering, and mutating ✅

Week 4: Grouping and tables ✅

Week 5: Functions ✅

Week 6: Analyze your data

> Put everything you've learned into action, and more!

Organization

```
my-project/
  ├─ my-project.Rproj
  ├─ README
  ├─ data/
  │    ├── raw/
  │    └── processed/
  ├─ code/
  ├─ results/
  │    ├── tables/
  │    ├── figures/
  │    └── output/
  └─ docs/
```

- An `.Rproj` file is mostly just a placeholder. It remembers various options, and makes it easy to open a new RStudio session that starts up in the correct working directory. You never need to edit it directly.
- A README file can just be a text file that includes notes for yourself or future users.
- I like to have a folder for raw data -- which I never touch -- and a folder(s) for datasets that I create along the way.

Referring to files with the `here` package

```r
source(here::here("code", "functions.R"))

dat <- read_csv(here::here(
             "data", "raw", "data.csv"))

p <- ggplot(dat) + geom_point(aes(x, y))

ggsave(plot = p,
       filename = here::here(
       "results", "figures", "fig.pdf"))
```

- The `here` package lets you refer to files without worrying too much about relative paths.
- Construct file paths with reference to the top directory holding your `.Rproj` file.
- `here::here("data", "raw", "data.csv")` for me, here, becomes `"/Users/louisahsmith/Google Drive/Teaching/R course/materials/data/raw/data.csv"`
- But if I send you this file, it will become whatever file path *you* need it to be.

Referring to the `here` package

```
here::here()
```

is equivalent to

```
library(here)
here()
```

I just prefer to write out the package name whenever I need it, but you can load the package for your entire session if you want.

Note that you can refer to any function without loading the whole package this way:

```
tableone::CreateTableOne()
```

is the same as

```
library(tableone)
CreateTableOne()
```

The `source()` function

Will run code from another file.

```r
# run the code in script.R, assuming it's in my current working directory
source("script.R")

# run the code in my-project/code/functions.R from wherever I am in my-project
source(here::here("code", "functions.R"))
```

All the objects will be created, packages loaded, etc. as if you had run the code directly from the console.

The `source()` function

Can even run code directly from a URL.

```
source("https://raw.githubusercontent.com/louisahsmith/intro-to-R-2020/master/s
```

```
## Error in eval(ei, envir): object 'new_vals' not found
```

Remember the first week when I had you generating errors on purpose?

- Reading code from another file can make it a bit harder to debug.
- But it's nice when you have functions, etc. that you use a lot and want to include them at the start of every script.

Reading in data

You could also begin your scripts by reading in your data via a data-cleaning file with `source()`.

Each of these have different arguments that will allow you to read in specific columns only, skip rows, give the variables names, etc. There are also better options out there if your dataset is really big (look into the `data.table` or the `vroom` package), and if you have other types of data.

```r
library(tidyverse)
dat <- read_csv("data.csv")
dat <- read_table("data.dat")
dat <- read_rds("data.rds")
dat <- readxl::read_excel("data.xlsx")
dat <- haven::read_sas("data.sas7bdat")
dat <- haven::read_stata("data.dta")
dat <- googlesheets4::read_sheet("sheet-id")
```

Saving your data

But once you've cleaned your data and created your dataset, you probably just want to save a copy so you don't need to perform all your data cleaning functions every time you want to use it.

- You can basically do the opposite of most of the `read` functions: `write`.
- The one I usually use, if I'm creating data for myself, is `write_rds()`. It creates an R object you can read in with `read_rds()`, so you can guarantee nothing will change in between writing and reading.
- If I'm sharing data, I usually use `write_csv()`.

Note: these are the `tidyverse` versions of the functions, which have better defaults, are more consistent, and are just more likely to do what you want. The "base R" versions are: `read.csv()`, `write.csv()`, `readRDS()` and `saveRDS()`.

Analysis plan

So my process might look like this:

1. **Clean the raw data in** `code/clean_data.R`

```r
# read in the raw dataset my collaborators gave me
dat <- read_csv(here::here("data", "raw", "dataset.csv"))

# do whatever cleaning/subsetting I need to
newdat <- dat %>%
  mutate(new_var = var * 2) %>%
  filter(age >= 40) %>%
  select(age, new_var)

# save as an r object for later analysis
write_rds(newdat, here::here("data", "processed", "over_40.rds"))
```

Analysis plan

2. In `code/main_analysis.R`, read in the clean data

```r
dat <- read_rds(here::here("data", "processed", "over_40.rds"))

# run a linear regression model
model <- lm(new_var ~ age, data = dat)

# save the model for making tables and figures later
write_rds(model, here::here("results", "output", "linear_mod.rds"))
```

Analysis plan

3. In `code/make_tables.R`, make some tables

```r
dat <- read_rds(here::here("data", "processed", "over_40.rds"))
mod <- read_rds(here::here("data", "output", "linear_mod.rds"))

# make a table 1
tab1 <- tableone::CreateTableOne(..., data = dat)

# make a table of analysis results
tab2 <- broom::tidy(model)
```

Depending on my needs, I might read these tables into an RMarkdown document, save them to a `.csv` file, etc.

Now on to making figures, etc....

1

YOUR TURN...

Exercises 6.1: Change the file paths so the document knits.

Missing values

- R uses `NA` for missing values
- Unlike some other statistical software, it will return `NA` to any logical statement
    - This makes it somewhat harder to deal with but also harder to make mistakes

```r
3 < NA
```

```
## [1] NA
```

```r
mean(c(1, 2, NA))
```

```
## [1] NA
```

```r
mean(c(1, 2, NA), na.rm = TRUE)
```

```
## [1] 1.5
```

Special NA functions

Certain functions deal with missing values explicitly

```
vals <- c(1, 2, NA)
is.na(vals)
```

```
## [1] FALSE FALSE  TRUE
```

```
anyNA(vals)
```

```
## [1] TRUE
```

```
na.omit(vals)
```

```
## [1] 1 2
```

Creating NAs with na_if()

You might read in data that has special values to indicate missingness.

In NLSY, -1 = Refused, -2 = Don't know, -3 = Invalid missing, -4 = Valid missing, -5 = Non-interview

```
nlsy[1, c("id", "glasses", "age_bir")]
```

```
## # A tibble: 1 x 3
##      id glasses age_bir
##   <dbl>   <dbl>   <dbl>
## 1     1      -4      -5
```

```
nlsy_na <- nlsy %>% na_if(-1) %>% na_if(-2) %>%
  na_if(-3) %>% na_if(-4) %>% na_if(-5)
nlsy_na[1, c("id", "glasses", "age_bir")]
```

```
## # A tibble: 1 x 3
##      id glasses age_bir
##   <dbl>   <dbl>   <dbl>
## 1     1      NA      NA
```

This is obviously a bit annoying if you have a lot of values that indicate missingness. In that case, you may want to look into the naniar package.

More `na_if()`

The `na_if()` strategy is generally the most useful if you're determining NA's over the course of your analysis, or if you have different NA values for different variables.

```r
# we decide that person 2 is a mistake...
nlsy_bad <- nlsy %>%
  mutate(id = na_if(id, 2))
nlsy_bad[1:3, c("id", "glasses", "age_bir")]
```

```
## # A tibble: 3 x 3
##      id glasses age_bir
##   <dbl>   <dbl>   <dbl>
## 1     1      -4      -5
## 2    NA       0      34
## 3     3       0      19
```

Better: read in NA's directly

Or, if you know a priori which values indicate missingness (e.g., "."), you can specify that when reading in the data.

```
nlsy <- read_csv(here::here("data", "nlsy.csv"),
                 col_names = colnames_nlsy, skip = 1,
                 na = c("-1", "-2", "-3", "-4", "-5"))
```

- You have to write the values as strings, even if they're numbers
- Caveat: This way you use the info about the reason for missingness. If that's important, read in the data first, create a variable for missingness reason (e.g., use `fct_recode()`), then changes the values to `NA`.

Complete cases

Sometimes you may just want to get rid of all the rows with missing values.

```
nrow(nlsy)
```

```
## [1] 12686
```

```
nlsy_cc <- nlsy %>% filter(complete.cases(nlsy))
nrow(nlsy_cc)
```

```
## [1] 1436
```

```
nlsy2 <- nlsy %>% na.omit()
nrow(nlsy2)
```

```
## [1] 1436
```

> Don't do this without good reason!

2

YOUR TURN...

Exercises 6.2: Create and exclude observations based on missing values.

Sharing your results

First: some quick analysis

```r
# load packages
library(tidyverse)
# must install with install.packages if haven't already
library(broom) # for making pretty model output
library(splines) # for adding splines

# read in data
nlsy_clean <- read_rds(here::here("data", "nlsy_clean.rds"))
```

- We're not going into many details because this isn't actually a statistical analysis class, but the broom package is very helpful for regression model results!

Quick regression overview

Model formulas will automatically make indicator variables for factors, with the first level the reference. An intercept will be included unless suppressed with `y ~ -1 + x`.

```r
# linear regression (OLS)
mod_lin1 <- lm(log_inc ~ age_bir + sex + race_eth,
                 data = nlsy_clean)
# another way to do linear regression (GLM)
mod_lin2 <- glm(log_inc ~ age_bir + sex + race_eth,
                family = gaussian(link = "identity"),
                data = nlsy_clean)
# logistic regression
mod_log <- glm(glasses ~ eyesight + sex + race_eth,
               family = binomial(link = "logit"),
               data = nlsy_clean)
# poisson regression
mod_pois <- glm(nsibs ~ sleep_wkdy + sleep_wknd,
                  family = poisson(link = "log"),
                  data = nlsy_clean)
```

Quick regression overview, cont.

- Create interactions with `*` (will automatically include main terms too).
- Create polynomial terms with, e.g., `I(x^2)`.
- Create splines with the splines package and the `ns()` functions (or other packages/functions).

```r
mod_big <- glm(log_inc ~ sex * age_bir +
                         nsibs +
                         I(nsibs^2) +
                         ns(sleep_wkdy, knots = 3),
               family = gaussian(link = "identity"),
               data = nlsy_clean)
```

- Like the `tidyverse` packages, you don't need to quote the variable names or use `data$variable` notation in model formulas

Look at results

```r
summary(mod_log)
```

```
##
## Call:
## glm(formula = glasses ~ eyesight + sex + race_eth, family = binomial(link = "logit"),
##     data = nlsy_clean)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.4275  -1.0825  -0.8343   1.2221   1.6261
##
## Coefficients:
##                                 Estimate Std. Error z value Pr(>|z|)
## (Intercept)                     -0.51260    0.06317  -8.114 4.89e-16 ***
## eyesightVery Good               -0.07920    0.05359  -1.478   0.1394
## eyesightGood                    -0.07146    0.06188  -1.155   0.2481
## eyesightFair                    -0.21488    0.09105  -2.360   0.0183 *
## eyesightPoor                     0.10558    0.18152   0.582   0.5608
## sexFemale                        0.69281    0.04493  15.420  < 2e-16 ***
## race_ethNon-Hispanic Black      -0.28460    0.06493  -4.383 1.17e-05 ***
```

Look at results

Or use the `tidy()` function from the `broom` package, which nicely summarizes all sorts of models.

```r
# from the broom package
tidy(mod_log)
```

```
## # A tibble: 8 x 5
##   term                            estimate std.error statistic  p.value
##   <chr>                              <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)                      -0.513     0.0632    -8.11  4.89e-16
## 2 eyesightVery Good                -0.0792    0.0536    -1.48  1.39e- 1
## 3 eyesightGood                     -0.0715    0.0619    -1.15  2.48e- 1
## 4 eyesightFair                     -0.215     0.0911    -2.36  1.83e- 2
## 5 eyesightPoor                      0.106     0.182      0.582 5.61e- 1
## 6 sexFemale                         0.693     0.0449    15.4   1.21e-53
## 7 race_ethNon-Hispanic Black       -0.285     0.0649    -4.38  1.17e- 5
## 8 race_ethNon-Black, Non-Hispanic   0.285     0.0594     4.80  1.60e- 6
```

Pull off a coefficient

```r
coef(mod_log)
```

```
##                      (Intercept)               eyesightVery Good
##                      -0.51260479                     -0.07920222
##                     eyesightGood                    eyesightFair
##                      -0.07145996                     -0.21487546
##                     eyesightPoor                       sexFemale
##                       0.10557518                      0.69280825
##       race_ethNon-Hispanic Black race_ethNon-Black, Non-Hispanic
##                      -0.28460369                      0.28527712
```

```r
coef(mod_log)[6]
```

```
## sexFemale
## 0.6928082
```

```r
tidy(mod_log) %>% slice(6) %>% pull(estimate)
```

```
## [1] 0.6928082
```

Reminder: if you have a model $\beta_0 + \beta_1 x_1 +$ etc., coefficient 6 is really β_5!

Pull off a coefficient by name

```
coef(mod_log)["sexFemale"]
```

```
## sexFemale
## 0.6928082
```

```
tidy(mod_log) %>% filter(term == "sexFemale") %>% pull(estimate)
```

```
## [1] 0.6928082
```

Creating new values

Since it's just a tibble/dataframe, you can create new columns!

```
# 95% confidence interval
res_mod_log <- mod_log %>% tidy() %>%
  mutate(lci = estimate - 1.96 * std.error,
         uci = estimate + 1.96 * std.error)
res_mod_log
```

```
## # A tibble: 8 x 7
##   term                            estimate std.error statistic  p.value    lci     uci
##   <chr>                               <dbl>     <dbl>     <dbl>    <dbl>  <dbl>   <dbl>
## 1 (Intercept)                      -0.513     0.0632    -8.11  4.89e-16 -0.636 -0.389
## 2 eyesightVery Good                -0.0792    0.0536    -1.48  1.39e- 1 -0.184  0.0258
## 3 eyesightGood                     -0.0715    0.0619    -1.15  2.48e- 1 -0.193  0.0498
## 4 eyesightFair                     -0.215     0.0911    -2.36  1.83e- 2 -0.393 -0.0364
## 5 eyesightPoor                      0.106     0.182      0.582 5.61e- 1 -0.250  0.461
## 6 sexFemale                         0.693     0.0449    15.4   1.21e-53  0.605  0.781
## 7 race_ethNon-Hispanic Black       -0.285     0.0649    -4.38  1.17e- 5 -0.412 -0.157
## 8 race_ethNon-Black, Non-Hispanic   0.285     0.0594     4.80  1.60e- 6  0.169  0.402
```

Calculating ORs

Since these are results from a logistic regression, we'll probably want to exponentiate the coefficients and their CIs.

```r
res_mod_log <- res_mod_log %>% select(term, estimate, lci, uci) %>%
  filter(term != "(Intercept)") %>%
  # exponentiate all three columns at once!
  mutate(across(c(estimate, lci, uci), exp))
res_mod_log
```

```
## # A tibble: 7 x 4
##   term                            estimate   lci   uci
##   <chr>                              <dbl> <dbl> <dbl>
## 1 eyesightVery Good                  0.924 0.832 1.03 
## 2 eyesightGood                       0.931 0.825 1.05 
## 3 eyesightFair                       0.807 0.675 0.964
## 4 eyesightPoor                       1.11  0.779 1.59 
## 5 sexFemale                          2.00  1.83  2.18 
## 6 race_ethNon-Hispanic Black         0.752 0.662 0.854
## 7 race_ethNon-Black, Non-Hispanic    1.33  1.18  1.49
```

Confidence intervals with `str_glue()`

Now we want to combine the lower and upper CI limits.

```r
res_mod_log %>% select(term, estimate, lci, uci) %>%
  filter(term != "(Intercept)") %>%
  mutate(ci = str_glue("({lci}, {uci})"))
```

```
## # A tibble: 7 x 5
##   term                             estimate   lci   uci ci
##   <chr>                               <dbl> <dbl> <dbl> <glue>
## 1 eyesightVery Good                   0.924 0.832 1.03  (0.831734362089001, 1.0261744158234
## 2 eyesightGood                        0.931 0.825 1.05  (0.824698936937584, 1.0510786843918
## 3 eyesightFair                        0.807 0.675 0.964 (0.674797666860532, 0.9642462833711
## 4 eyesightPoor                        1.11  0.779 1.59  (0.778639992096712, 1.5862247743449
## 5 sexFemale                           2.00  1.83  2.18  (1.8307856398006, 2.18337382956281)
## 6 race_ethNon-Hispanic Black          0.752 0.662 0.854 (0.662416384318316, 0.854408001346З
## 7 race_ethNon-Black, Non-Hispanic     1.33  1.18  1.49  (1.18383962963298, 1.4944991893382l
```

We can paste text and R code (within `{}`) together with `str_glue()`. Everything goes in quotation marks.

More `str_glue()`

You can paste any R expression you want evaluated in the curly braces.

You can break up chunks of your string to make it easier to read in your code.

```
str_glue(
  "The intercept from the regression is ",
  "{round(coef(lm(income~sex, data = nlsy_clean))[1])} and a random ",
  "number that I generated is {round(rnorm(1, 0, 1), 3)}."
)
```

```
## The intercept from the regression is 14880 and a random number that I generated is -1.469.
```

More functions are available in the `glue` package. For example, you could write a nice list of the regions in the data like this:

```
glue::glue_collapse(levels(nlsy_clean$region), sep = ", ", last = ", and ")
```

```
## Northeast, North central, South, and West
```

Better: Create a function

We want to take these values and print "OR (95% CI *LCI*, *UCI*)" for each one. Let's make a function to put together everything we've done so far!

```r
ci_func <- function(estimate, lci, uci) {
  OR <- round(exp(estimate), 2)
  lci <- round(exp(lci), 2)
  uci <- round(exp(uci), 2)
  to_print <- str_glue("{OR} (95% CI {lci}, {uci})")
  return(to_print)
}
```

Let's test on some made-up values:

```r
ci_func(.2523421, -.142433, .851234)
```

```
## 1.29 (95% CI 0.87, 2.34)
```

From start to finish

```r
new_mod <- glm(glasses ~ eyesight + sex, family = binomial(link = "logit"),
               data = nlsy_clean)
tidy(new_mod) %>%
  filter(term != "(Intercept)") %>% # we don't care about this term
  mutate(lci = estimate - 1.96 * std.error,
         uci = estimate + 1.96 * std.error,
         OR = ci_func(estimate, lci, uci),
         p.value = scales::pvalue(p.value)) %>% # for formatting p-values
  select(term, OR, p.value)
```

```
## # A tibble: 5 x 3
##   term              OR                       p.value
##   <chr>             <glue>                   <chr>
## 1 eyesightVery Good 0.9 (95% CI 0.81, 1)     0.042
## 2 eyesightGood      0.88 (95% CI 0.78, 0.99) 0.041
## 3 eyesightFair      0.74 (95% CI 0.62, 0.88) <0.001
## 4 eyesightPoor      1.01 (95% CI 0.71, 1.44) 0.959
## 5 sexFemale         1.99 (95% CI 1.82, 2.17) <0.001
```

If you want 2 decimal places no matter what, use something like `format(round(x, digits = 2), nsmall = 2)`

Important R lesson

Whatever you want to write code to do, someone else has already probably done it

The `tidy()` function could have actually done the exponentiating and confidence interval calculating for us. See `help(tidy.glm)`. But much more fun to do it ourselves 😉

```
tidy(new_mod, conf.int = TRUE, exponentiate = TRUE)
```

```
## # A tibble: 6 x 7
##   term              estimate std.error statistic  p.value conf.low conf.high
##   <chr>                <dbl>     <dbl>     <dbl>    <dbl>    <dbl>     <dbl>
## 1 (Intercept)          0.655    0.0430   -9.85   6.59e-23    0.602     0.712
## 2 eyesightVery Good    0.898    0.0531   -2.04   4.18e- 2    0.809     0.996
## 3 eyesightGood         0.882    0.0612   -2.05   4.06e- 2    0.782     0.995
## 4 eyesightFair         0.738    0.0902   -3.37   7.58e- 4    0.618     0.880
## 5 eyesightPoor         1.01     0.181     0.0512 9.59e- 1    0.708     1.44
## 6 sexFemale            1.99     0.0446   15.4    8.85e-54    1.82      2.17
```

If you're wondering why these confidence intervals differ slightly from the last slide, these are likelihood-based and ours were Wald confidence intervals. Don't worry if you don't know what this means!

Final challenge

Data analysis from start to finish

1. Prepare and organize your project
2. Load and clean the data
3. Do some exploratory analysis (table 1, figure)
4. Do some regression analysis (results table, figure)

> You can do it!

Prepare your project

Do something totally outside the folders you've used for the course material!

- File -> New Project -> New Directory -> New Project
- Name it something like NLSY and put it in an appropriate folder on your computer
- Within that folder, make new folders as follows:

```
NLSY/
 ├─ NLSY.Rproj
 ├─ data/
 │   ├── raw/
 │   └── processed/
 ├─ code/
 └── results/
     ├── tables/
     └── figures/
```

Prepare the data

- Download the linked dataset and save into `data/raw`.
- Create a new file and save it as `clean_data.R`.
- In that file, read in the NLSY data and load any packages you need. Make sure you replace any missing values with NA. Hint: there are extra missing values in the some variables.
- Add factor labels as necessary. Select the factor variables plus `income`, `id`, and 2 others of your choosing. Then restrict to complete cases and people with incomes < $30,000. Make a variable for the log of income (replace with NA if income <= 0).
- Also in that file, save your new dataset as a `.rds` file to the `data/processed` folder.

Do some exploratory analysis

- Create a file called `create_figure.R`. In this file, read in the cleaned dataset. Load any packages you need. Then make a `ggplot` figure of your choosing to show something about the distribution of the data. Save it to the `results/figures` folder as a `.png` file using the `ggsave()` function.
- Create a file called `table_1.R`. In this file, read in the cleaned dataset and use the `tableone` package to create a table 1 with the variables of your choosing. Modify the following code to save it as a `.csv` file. Open it in Excel/Numbers/Google Sheets/etc. to make sure it worked.

```
tab1 <- CreateTableOne(...) %>% print() %>% as_tibble(rownames = "id")
write_csv(tab1, ...)
```

Do some regression analysis

- In another file called `lin_reg.R`, read in the data and run the following linear regression: `lm(log_inc ~ sex + race_eth +{other variables of your choosing}, data = nlsy)`. Modify the CI function to produce a table of results for a *linear* regression. Add an argument `digits =`, with a default of 2, to allow you to choose the number of digits you'd like. Save it in a separate file called `functions.R`. Use `source()` to read in the function at the beginning of your script.
- Save a table of your results as a `.csv` file. Make the names of the coefficients nice!
- Using the results, use `ggplot` to make a figure. Use `geom_point()` for the point estimates and `geom_errorbar()` for the confidence intervals. It will look something like this:

```
ggplot(data) +
  geom_point(aes(x = , y = )) +
  geom_errorbar(aes(x = , ymin = , ymax = ))
```

- Save that figure as a `.pdf` using `ggsave()`. You may want to play around with the `height =` and `width =` arguments to make it look like you want.

Appendix: some other packages I like but haven't mentioned

- `lubridate`: Work with dates and times really easily. (https://lubridate.tidyverse.org)
- `janitor`: Helps clean variable names, etc. (http://sfirke.github.io/janitor/)
- `furrr`: Speed up your code with parallel processing. (https://davisvaughan.github.io/furrr/)
- `shiny`: Make interactive apps. I made http://selection-bias.louisahsmith.com in shiny. (http://shiny.rstudio.com)
- `drake`: Pipeline for analysis. (https://docs.ropensci.org/drake)
- `rvest`: Scrape data from websites. (https://rvest.tidyverse.org)

3

YOUR TURN...

Exercises 6.3: Work through the challenge!