CSSS508, Week 5

Importing, Exporting, and Cleaning Data

Chuck Lanfear

Apr 28, 2021

Updated: Apr 26, 2021

UW CS&SS

Today's Theme:

"Data Custodian Work"

Issues around getting data *in* and *out* of R and making it analytically *ready*:

- Working directories and projects
- Importing and exporting data: `readr` and `haven`
- Cleaning and reshaping data: `tidyr`
- Dates and times: `lubridate`
- Controlling factor variables: `forcats`

Directories

Working Directory

You may recall that the **working directory** is where R will look for and save things by default.

You can find out what it is using the function `getwd()`.

On my computer when I knitted these slides, it happened to be:

```
getwd()
```

```
## [1] "C:/Users/cclan/OneDrive/GitHub/CSSS508/Lectures/Week5"
```

Changing Your Working Directory

You can use `setwd(dir = "C:/path/to/new/working/directory")` to change the working directory.

Working Directory Suggestions:

- `.Rmd` files use their current directory as a working directory: Just put everything you need in there!
- For larger projects, instead of setting a working directory, it is usually better to use RStudio projects to manage working directories.
- Windows users: If you copy a path from Explorer, make sure to change back slashes (`\`) to forward slashes (`/`) for the filepaths
- If you need to set a working, put `setwd()` at the start of your file so that someone using another computer knows they need to modify it

Projects in RStudio

A better way to deal with working directories: RStudio's **project** feature in the top-right dropdown. This has lots of advantages:

- Sets your working directory to be the project directory.
- Remembers objects in your workspace, command history, etc. next time you re-open that project.
- Reduces risk of intermingling different work using the same variable names (e.g. `n`) by using separate RStudio instances for each project.
- Easy to integrate with version control systems (e.g. `git`)
    - I usually make each RStudio project its own GitHub repository.

If you're interested in advanced project management, ask me after class or check out my presentation on reproducible research with rrtools.

Relative Paths

Once you've set the working directory—or you're in an RStudio project—you can refer to folders and files within the working directory using relative paths.

```
library(ggplot2)
a_plot <- ggplot(data = cars, aes(x = speed, y = dist)) +
    geom_point()
ggsave("graphics/cars_plot.png", plot = a_plot)
```

The above would save an image called "cars_plot.png" inside an existing folder called "graphics" within my working directory.

Relative paths are nice, because all locations of loaded and saved files can be changed just by altering the working directory.

Relative paths also allow others to download your files or entire project and use them on their computer without modifying all the paths!

Importing and Exporting Data

Special Data Access Packages

If you are working with a popular data source, try Googling to see if it has a devoted R package on *CRAN* or *Github* (use `remotes::install_github("user/repository")` for these). Examples:

- `WDI`: World Development Indicators (World Bank)
- `WHO`: World Health Organization API
- `tidycensus`: Census and American Community Survey [1]
- `quantmod`: financial data from Yahoo, FRED, Google

[1] We'll use this in our lecture on geographical data!

Delimited Text Files

Besides a package, the easiest way to work with external data is for it to be stored in a *delimited* text file, e.g. comma-separated values (**.csv**) or tab-separated values (**.tsv**). Here is **.csv** data:

```
"Subject","Depression","Sex","Week","HamD","Imipramine"
101,"Non-endogenous","Male",0,26,NA
101,"Non-endogenous","Male",1,22,NA
101,"Non-endogenous","Male",2,18,4.04305
101,"Non-endogenous","Male",3,7,3.93183
101,"Non-endogenous","Male",4,4,4.33073
101,"Non-endogenous","Male",5,3,4.36945
103,"Non-endogenous","Female",0,33,NA
103,"Non-endogenous","Female",1,24,NA
103,"Non-endogenous","Female",2,15,2.77259
```

`readr`

R has a variety of built-in functions for importing data stored in text files, like `read.table()` and `read.csv()`. I recommend using the versions in the `readr` package instead: `read_csv()`, `read_tsv()`, and `read_delim()`:

`readr` function features:

- Faster![1]
- Better defaults (e.g. doesn't automatically convert character data to factors)
- A *little* smarter about dates and times
- Handy function `problems()` you can run if there are errors
- Loading bars for large files

```r
library(readr)
```

[1] `vroom` is even faster!

`readr` Importing Example

Let's import some data about song ranks on the Billboard Hot 100 in 2000:

```r
billboard_2000_raw <- read_csv(file = "https://clanfear.github.io/CSSS508/Lectures/Week5/dat
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   .default = col_double(),
##   artist = col_character(),
##   track = col_character(),
##   time = col_time(format = ""),
##   date.entered = col_date(format = ""),
##   wk66 = col_logical(),
##   wk67 = col_logical(),
##   wk68 = col_logical(),
##   wk69 = col_logical(),
##   wk70 = col_logical(),
##   wk71 = col_logical(),
##   wk72 = col_logical(),
##   wk73 = col_logical(),
##   wk74 = col_logical(),
##   wk75 = col_logical(),
##   wk76 = col_logical()
## )
## i Use `spec()` for the full column specifications.
```

Did It Load?

Look at the data types for the last few columns:

```
str(billboard_2000_raw[, 65:ncol(billboard_2000_raw)])
```

```
## tibble[,17] [317 x 17] (S3: tbl_df/tbl/data.frame)
##  $ wk60: num [1:317] NA NA NA NA NA NA NA NA NA NA ...
##  $ wk61: num [1:317] NA NA NA NA NA NA NA NA NA NA ...
##  $ wk62: num [1:317] NA NA NA NA NA NA NA NA NA NA ...
##  $ wk63: num [1:317] NA NA NA NA NA NA NA NA NA NA ...
##  $ wk64: num [1:317] NA NA NA NA NA NA NA NA NA NA ...
##  $ wk65: num [1:317] NA NA NA NA NA NA NA NA NA NA ...
##  $ wk66: logi [1:317] NA NA NA NA NA NA ...
##  $ wk67: logi [1:317] NA NA NA NA NA NA ...
##  $ wk68: logi [1:317] NA NA NA NA NA NA ...
##  $ wk69: logi [1:317] NA NA NA NA NA NA ...
##  $ wk70: logi [1:317] NA NA NA NA NA NA ...
##  $ wk71: logi [1:317] NA NA NA NA NA NA ...
##  $ wk72: logi [1:317] NA NA NA NA NA NA ...
##  $ wk73: logi [1:317] NA NA NA NA NA NA ...
##  $ wk74: logi [1:317] NA NA NA NA NA NA ...
##  $ wk75: logi [1:317] NA NA NA NA NA NA ...
##  $ wk76: logi [1:317] NA NA NA NA NA NA ...
```

What Went Wrong?

`readr` uses the values in the first 1000 rows to guess the type of the column (integer, logical, numeric, character). There are not many songs in the data that charted for 60+ weeks—and none in the first 1000 that charted for 66+ weeks!

Since it encountered no values, `readr` assumed the `wk66`-`wk76` columns were *character* to be sure nothing would be lost. Use the `col_types` argument to fix this:

```
# paste is a string concatenation function
# i = integer, c = character, D = date
# rep("i", 76) does the 76 weeks of integer ranks
bb_types <- paste(c("icccD", rep("i", 76)), collapse="")

billboard_2000_raw <-
  read_csv(file = "https://clanfear.github.io/CSSS508/Lectures/Week5/
           col_types = bb_types)
```

Alternate Solutions

You could also deal with this by adjusting the maximum rows used by `readr` to guess column types:

```
read_csv(file, guess_max=5000) # Default is 1000
```

Or you could use `read.csv()` in the `foreign` package. This is a base R alternative that is slower and a bit dumber.

Another alternative would be using `vroom`, a package for high-speed reading of text data like `.csv` files.

```
vroom::vroom(file)
```

One advantage of `vroom`: You can give it a vector of filenames and it will read every file and combine them into one dataframe.

`vroom` has less error checking than `readr`, though, so best to use on files you have examined first.

Spreadsheet Files

For Excel files (`.xls` or `.xlsx`), I recommend using `readxl` and `writexl`.

For Google Docs Spreadsheets, there's the `googlesheets4` package.

You won't keep text formatting, color, comments, or merged cells so if these mean something in your data (*bad!*), you'll need to get creative.

If an Excel sheet gives you grief (say, due to merged cells), the simplest thing is open them up, export to CSV, then import in R—and compare carefully to make sure everything worked!

If you need to programmatically work with non-tabular Excel sheets--particularly if you need to retain meaningful formatting--look to the powerful but complex `unpivotr` and `tidyxl` packages.

Writing Delimited Files

Getting data out of R into a delimited file is very similar to getting it into R:

```
write_csv(billboard_2000_raw, path = "billboard_data.csv")
```

This saved the data we pulled off the web in a file called `billboard_data.csv` in my working directory.

Saving in R Formats

Exporting to a `.csv` drops R metadata, such as whether a variable is a character or factor. You can save objects (data frames, lists, etc.) in R formats to preserve this.

- `.Rds` format:
    - Used for single objects, doesn't save original the object name
    - Save: `write_rds(old_object_name, "path.Rds")`
    - Load: `new_object_name <- read_rds("path.Rds")`
- `.Rdata` or `.Rda` format:
    - Used for saving multiple files where the original object names are preserved
    - Save: `save(object1, object2, ... , file = "path.Rdata")`
    - Load: `load("path.Rdata")` *without assignment operator*

I pretty much always just save as `.Rdata`--but that is personal preference.

`dput()`

For asking for help, it is useful to prepare a snippet of your data with `dput()`:[1]

```
dput(head(cars, 8))
```

```
## structure(list(speed = c(4, 4, 7, 7, 8, 9, 10, 10), dist = c(2,
## 10, 4, 22, 16, 10, 18, 26)), row.names = c(NA, 8L), class = "data.frame")
```

The output of `dput()` can be copied and assigned to an object in R:

```
temp <- structure(list(speed = c(4, 4, 7, 7, 8, 9, 10, 10),
                       dist = c(2, 10, 4, 22, 16, 10, 18, 26)),
                       .Names = c("speed", "dist"),
                       row.names = c(NA, 8L), class = "data.frame")
```

[1] A reprex is even better!

Reading in Data from Other Software

Working with **Stata** or **SPSS** users? You can use a package to bring in their saved data files:

- `haven` for Stata, SPSS, and SAS.
  - Part of the `tidyverse` family
- `foreign` for Stata, SPSS, Minitab
  - Part of base R

For less common formats, Google it. I've yet to encounter a data format without an R package to handle it (or at least a clever hack).

If you encounter a mysterious file extension (e.g. `.dat`), try opening it with a good text editor first (e.g. Atom or Sublime); there's a good chance it is actually raw text with a delimiter or fixed format that R can handle!

Tidying Data

Initial Spot Checks

First things to check after loading new data:

- Did the last rows/columns from the original file make it in?
    - May need to use different package or manually specify range
- Are the column names in good shape?
    - Modify a `col_names=` argument or fix with `rename()`
- Are there "decorative" blank rows or columns to remove?
    - `filter()` or `select()` out those rows/columns
- How are missing values represented: `NA`, `" "` (blank), `.` (period), `999`?
    - Use `mutate()` with `ifelse()` to fix these (perhaps *en masse* with looping)
- Are there character data (e.g. ZIP codes with leading zeroes) being incorrectly represented as numeric or vice versa?
    - Modify `col_types=` argument, or use `mutate()` and `as.numeric()`

Slightly Messy Data

Program	Female	Male
Evans School	10	6
Arts & Sciences	5	6
Public Health	2	3
Other	5	1

- What is an observation?
  - A group of students from a program of a given gender
- What are the variables?
  - Program, gender
- What are the values?
  - Program: Evans School, Arts & Sciences, Public Health, Other
  - Gender: Female, Male -- **in the column headings, not its own column!**
  - Count: **spread over two columns!**

Tidy Version

Program	Gender	Count
Evans School	Female	10
Evans School	Male	6
Arts & Sciences	Female	5
Arts & Sciences	Male	6
Public Health	Female	2
Public Health	Male	3
Other	Female	5
Other	Male	1

Each variable is a column.

Each observation is a row.

Ready to throw into `ggplot()`!

Billboard is Just Ugly-Messy

year	artist	track	time	date.entered	wk1	wk2	wk3	wk4	wk5
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	87	82	72	77	87
2000	2Ge+her	The Hardest Part Of ...	3:15	2000-09-02	91	87	92	NA	NA
2000	3 Doors Down	Kryptonite	3:53	2000-04-08	81	70	68	67	66
2000	3 Doors Down	Loser	4:24	2000-10-21	76	76	72	69	67
2000	504 Boyz	Wobble Wobble	3:35	2000-04-15	57	34	25	17	17
2000	98^0	Give Me Just One Nig...	3:24	2000-08-19	51	39	34	26	26
2000	A*Teens	Dancing Queen	3:44	2000-07-08	97	97	96	95	100
2000	Aaliyah	I Don't Wanna	4:15	2000-01-29	84	62	51	41	38
2000	Aaliyah	Try Again	4:03	2000-03-18	59	53	38	28	21
2000	Adams, Yolanda	Open My Heart	5:30	2000-08-26	76	76	74	69	68
2000	Adkins, Trace	More	3:05	2000-04-29	84	84	75	73	73
2000	Aguilera, Christina	Come On Over Baby (A...	3:38	2000-08-05	57	47	45	29	23

Week columns continue up to `wk76`!

Billboard

- What are the **observations** in the data?
    - Week since entering the Billboard Hot 100 per song
- What are the **variables** in the data?
    - Year, artist, track, song length, date entered Hot 100, week since first entered Hot 100 (**spread over many columns**), rank during week (**spread over many columns**)
- What are the **values** in the data?
    - e.g. 2000; 3 Doors Down; Kryptonite; 3 minutes 53 seconds; April 8, 2000; Week 3 (**stuck in column headings**); rank 68 (**spread over many columns**)

Tidy Data

Tidy data (aka "long data") are such that:

1. The values for a single observation are in their own row.
2. The values for a single variable are in their own column.
3. There is only one value per cell.

Why do we want tidy data?

- Easier to understand many rows than many columns
- Required for plotting in `ggplot2`
- Required for many types of statistical procedures (e.g. hierarchical or mixed effects models)
- Fewer confusing variable names
- Fewer issues with missing values and "imbalanced" repeated measures data

`tidyr`

The `tidyr` package provides functions to tidy up data, similar to `reshape` in Stata or `varstocases` in SPSS. Key functions:

- **`pivot_longer()`**: takes a set of columns and pivots them down to make two new columns (which you can name yourself):
  - A `name` column that stores the original column names
  - A `value` with the values in those original columns
- **`pivot_wider()`**: inverts `pivot_longer()` by taking two columns and pivoting them up into multiple columns
- **`separate()`**: pulls apart one column into multiple columns (common after `pivot_longer()` where values are embedded in column names)
  - `extract_numeric()` does a simple version of this for the common case when you just want grab the number part
- **`extract()`** for pivoting a column into multiple *sets* of columns.
  - See [Hadley's response to this question]() for an example.

pivot_longer()

Let's use `pivot_longer()` to get the week and rank variables out of their current layout into two columns (big increase in rows, big drop in columns):

```
library(tidyr); library(dplyr)
billboard_2000 <- billboard_2000_raw %>%
  pivot_longer(starts_with("wk"),
               names_to ="week",
               values_to = "rank")
dim(billboard_2000)
```

```
## [1] 24092     7
```

`starts_with()` and other syntax and helper functions from `dplyr::select()` work here too.

We could instead use: `pivot_longer(wk1:wk76, names_to = "week", values_to = "rank")` to pull out these contiguous columns.

`pivot`ed Weeks

```
head(billboard_2000)
```

```
## # A tibble: 6 x 7
##    year artist track                      time  date.entered week   rank
##   <int> <chr>  <chr>                      <chr> <date>       <chr> <int>
## 1  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26   wk1      87
## 2  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26   wk2      82
## 3  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26   wk3      72
## 4  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26   wk4      77
## 5  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26   wk5      87
## 6  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26   wk6      94
```

Now we have a single week column!

Pivoting Better?

```
summary(billboard_2000$rank)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
##    1.00   26.00   51.00   51.05   76.00  100.00   18785 
```

This is an improvement, but we don't want to keep the 18785 rows with missing ranks (i.e. observations for weeks since entering the Hot 100 that the song was no longer on the Hot 100).

Pivoting Better: `values_drop_na`

The argument `values_drop_na = TRUE` to `pivot_longer()` will remove rows with missing ranks.

```
billboard_2000 <- billboard_2000_raw %>%
  pivot_longer(starts_with("wk"),
               names_to ="week",
               values_to = "rank",
               values_drop_na = TRUE)
summary(billboard_2000$rank)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00   26.00   51.00   51.05   76.00  100.00
```

No more `NA` values!

```
dim(billboard_2000)
```

```
## [1] 5307    7
```

And way fewer rows!

`parse_number()`

The week column is character, but should be numeric.

```
summary(billboard_2000$week)
```

```
##    Length     Class      Mode 
##      5307 character character
```

`tidyr` provides a convenience function to grab just the numeric information from a column that mixes text and numbers:

```
billboard_2000 <- billboard_2000 %>%
    mutate(week = parse_number(week))
summary(billboard_2000$week)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##    1.00    5.00   10.00   11.47   16.00   65.00
```

For more sophisticated conversion or pattern checking, you'll need to use string parsing (to be covered in week 8).

Or use `names_prefix`

```
billboard_2000 <- billboard_2000_raw %>%
  pivot_longer(starts_with("wk"),
               names_to ="week",
               values_to = "rank",
               values_drop_na = TRUE,
               names_prefix = "wk",
               names_transform = list(week = as.integer))
head(billboard_2000, 3)
```

```
## # A tibble: 3 x 7
##    year artist track                   time  date.entered  week  rank
##   <int> <chr>  <chr>                   <chr> <date>       <int> <int>
## 1  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26       1    87
## 2  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26       2    82
## 3  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26       3    72
```

We use `names_prefix` to remove `"wk"` from the values, and `names_transform` to convert into an integer number.

`separate()`

The track length column isn't analytically friendly. Let's convert it to a number rather than the character (minutes:seconds) format:

```
billboard_2000 <- billboard_2000 %>%
    separate(time, into = c("minutes", "seconds"),
             sep = ":", convert = TRUE) %>%
    mutate(length = minutes + seconds / 60) %>%
    select(-minutes, -seconds)
summary(billboard_2000$length)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   2.600   3.667   3.933   4.031   4.283   7.833
```

`sep = ":"` tells `separate()` to split the column into two where it finds a colon (`:`).

Then we add `seconds / 60` to `minutes` to produce a numeric `length` in minutes.

`pivot_wider()` Motivation

`pivot_wider()` is the opposite of `pivot_longer()`, which you use if you have data for the same observation taking up multiple rows.

Example of data that we probably want to pivot wider (unless we want to plot each statistic in its own facet):

Group	Statistic	Value
A	Mean	1.28
A	Median	1.0
A	SD	0.72
B	Mean	2.81
B	Median	2
B	SD	1.33

A common cue to use `pivot_wider()` is having measurements of different quantities in the same column.

Before `pivot_wider()`

```
(too_long_data <-
   data.frame(Group     = c(rep("A", 3), rep("B", 3)),
              Statistic = rep(c("Mean", "Median", "SD"), 2),
              Value     = c(1.28, 1.0, 0.72, 2.81, 2, 1.33)))
```

```
##   Group Statistic Value
## 1     A      Mean  1.28
## 2     A    Median  1.00
## 3     A        SD  0.72
## 4     B      Mean  2.81
## 5     B    Median  2.00
## 6     B        SD  1.33
```

After `pivot_wider()`

```
(just_right_data <- too_long_data %>%
    pivot_wider(names_from = Statistic, values_from = Value))
```

```
## # A tibble: 2 x 4
##   Group  Mean Median    SD
##   <chr> <dbl>  <dbl> <dbl>
## 1 A      1.28      1  0.72
## 2 B      2.81      2  1.33
```

Charts of 2000: Data Prep

Let's look at songs that hit #1 at some point and look how they got there versus songs that did not:

```
billboard_2000 <- billboard_2000 %>%
    group_by(artist, track) %>%
    mutate(`Weeks at #1` = sum(rank == 1),
           `Peak Rank`   = ifelse(any(rank == 1),
                                  "Hit #1",
                                  "Didn't #1")) %>%
    ungroup()
```

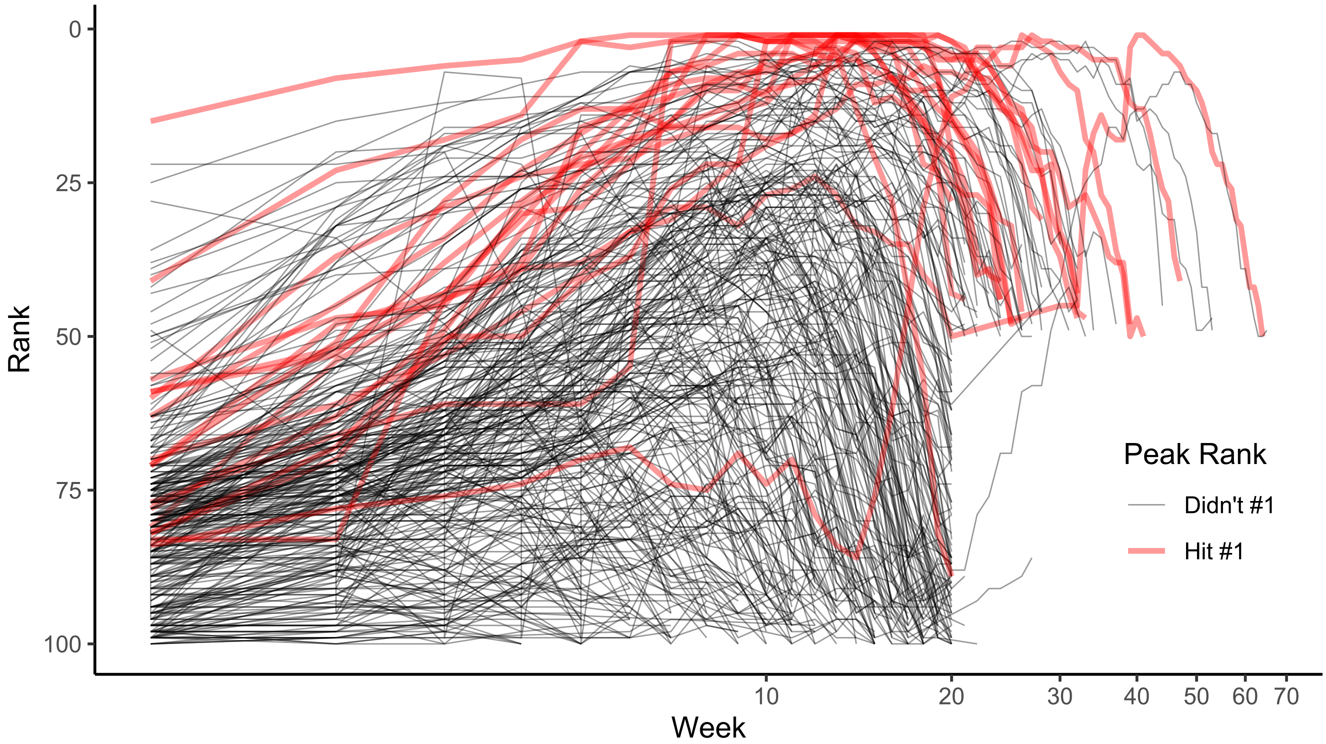
Things to note:

- `any(min_rank==1)` checks to see if *any* value of `rank` is equal to one for the given `artist` and `track`
- `ungroup()` here removes the grouping made by `group_by()`.

Charts of 2000: `ggplot2`

```
library(ggplot2)
billboard_trajectories <-
  ggplot(data = billboard_2000,
         aes(x = week, y = rank, group = track,
             color = `Peak Rank`)
         ) +
  geom_line(aes(size = `Peak Rank`), alpha = 0.4) +
    # rescale time: early weeks more important
  scale_x_log10(breaks = seq(0, 70, 10)) +
  scale_y_reverse() + # want rank 1 on top, not bottom
  theme_classic() +
  xlab("Week") + ylab("Rank") +
  scale_color_manual(values = c("black", "red")) +
  scale_size_manual(values = c(0.25, 1)) +
  theme(legend.position = c(0.90, 0.25),
        legend.background = element_rect(fill="transparent"))
```

Charts of 2000: Beauty!



Observation: There appears to be censoring around week 20 for songs falling out of the top 50 that I'd want to follow up on.

Which Were #1 the Most Weeks?

```
billboard_2000 %>%
    distinct(artist, track, `Weeks at #1`) %>%
    arrange(desc(`Weeks at #1`)) %>%
    head(7)
```

```
## # A tibble: 7 x 3
##   artist              track                   `Weeks at #1`
##   <chr>               <chr>                           <int>
## 1 Destiny's Child     Independent Women Pa...            11
## 2 Santana             Maria, Maria                       10
## 3 Aguilera, Christina Come On Over Baby (A...             4
## 4 Madonna             Music                               4
## 5 Savage Garden       I Knew I Loved You                  4
## 6 Destiny's Child     Say My Name                         3
## 7 Iglesias, Enrique   Be With You                         3
```

Dates and Times

Getting Usable Dates

We have the date the songs first charted, but not the dates for later weeks. We can calculate these now that the data are tidy:

```
billboard_2000 <- billboard_2000 %>%
    mutate(date = date.entered + (week - 1) * 7)
billboard_2000 %>% arrange(artist, track, week) %>%
    select(artist, date.entered, week, date, rank) %>% head(4)
```

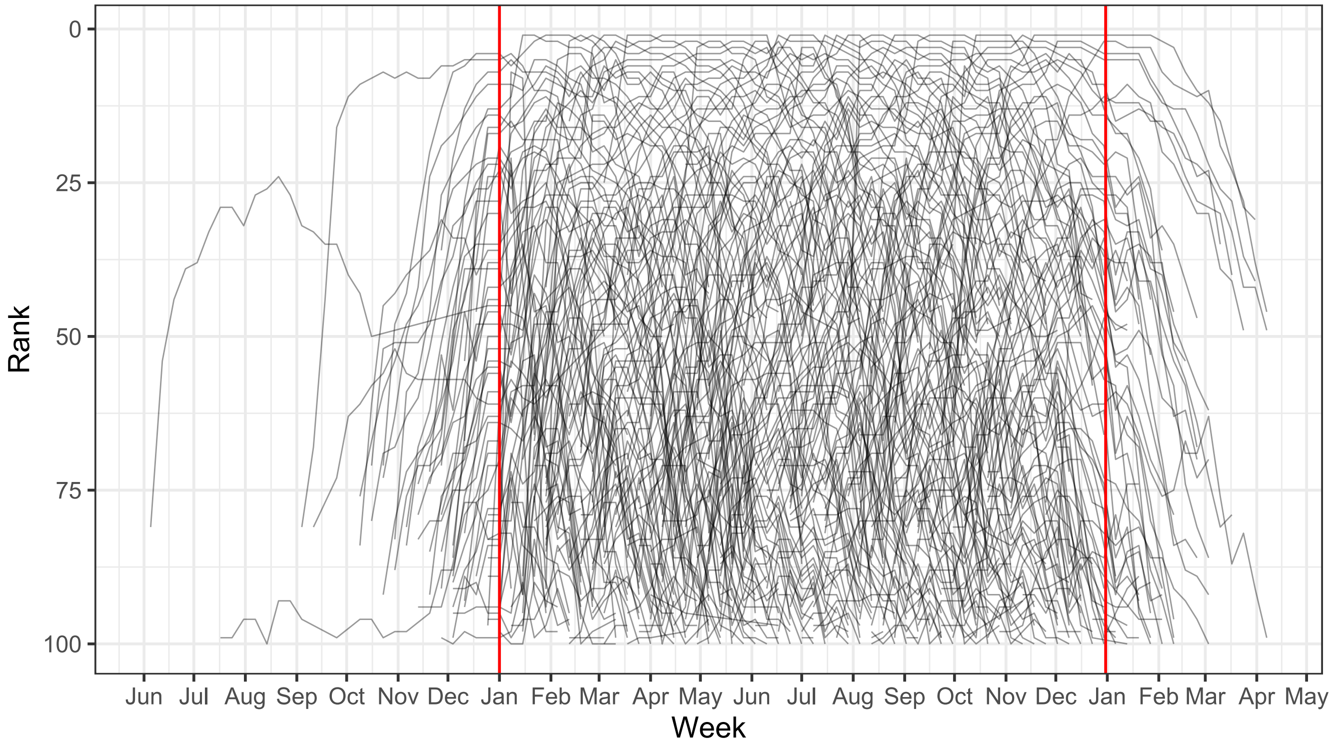
```
## # A tibble: 4 x 5
##   artist date.entered  week date        rank
##   <chr>  <date>       <int> <date>     <int>
## 1 2 Pac  2000-02-26       1 2000-02-26    87
## 2 2 Pac  2000-02-26       2 2000-03-04    82
## 3 2 Pac  2000-02-26       3 2000-03-11    72
## 4 2 Pac  2000-02-26       4 2000-03-18    77
```

This works because `date` objects are in units of days—we just add 7 days per week to the start date.

Preparing to Plot Over Calendar Time

```
plot_by_day <-
  ggplot(billboard_2000, aes(x = date, y = rank, group = track)) +
  geom_line(size = 0.25, alpha = 0.4) +
  # just show the month abbreviation label (%b)
  scale_x_date(date_breaks = "1 month", date_labels = "%b") +
  scale_y_reverse() + theme_bw() +
  # add lines for start and end of year:
  # input as dates, then make numeric for plotting
  geom_vline(xintercept = as.numeric(as.Date("2000-01-01", "%Y-%m-%d")),
             col = "red") +
  geom_vline(xintercept = as.numeric(as.Date("2000-12-31", "%Y-%m-%d")),
             col = "red") +
  xlab("Week") + ylab("Rank")
```

Calendar Time Plot!



We see some of the entry dates are before 2000---presumably songs still charting during 2000 that came out earlier.

Dates and Times

To practice working with finer-grained temporal information, let's look at one day of Seattle Police response data obtained from data.seattle.gov:

```r
spd_raw <- read_csv("https://clanfear.github.io/CSSS508/Seattle_Police_Department_911_Incident_Response.csv")
```

```
##
## -- Column specification --------------------------------------------
## cols(
##   `CAD CDW ID` = col_double(),
##   `CAD Event Number` = col_double(),
##   `General Offense Number` = col_double(),
##   `Event Clearance Code` = col_character(),
##   `Event Clearance Description` = col_character(),
##   `Event Clearance SubGroup` = col_character(),
##   `Event Clearance Group` = col_character(),
##   `Event Clearance Date` = col_character(),
##   `Hundred Block Location` = col_character(),
##   `District/Sector` = col_character(),
##   `Zone/Beat` = col_character(),
##   `Census Tract` = col_double(),
##   Longitude = col_double(),
##   Latitude = col_double(),
##   `Incident Location` = col_character(),
##   `Initial Type Description` = col_character(),
##   `Initial Type Subgroup` = col_character(),
##   `Initial Type Group` = col_character(),
##   `At Scene Time` = col_character()
## )
```

SPD Data

```r
glimpse(spd_raw)
```

```
## Rows: 706
## Columns: 19
## $ `CAD CDW ID`                  <dbl> 1701856, 1701857, 1701853, 170~
## $ `CAD Event Number`            <dbl> 16000104006, 16000103970, 1600~
## $ `General Offense Number`      <dbl> 2016104006, 2016103970, 201610~
## $ `Event Clearance Code`        <chr> "063", "064", "161", "245", "2~
## $ `Event Clearance Description` <chr> "THEFT - CAR PROWL", "SHOPLIFT~
## $ `Event Clearance SubGroup`    <chr> "CAR PROWL", "THEFT", "TRESPAS~
## $ `Event Clearance Group`       <chr> "CAR PROWL", "SHOPLIFTING", "T~
## $ `Event Clearance Date`        <chr> "03/25/2016 11:58:30 PM", "03/~
## $ `Hundred Block Location`      <chr> "S KING ST / 8 AV S", "92XX BL~
## $ `District/Sector`             <chr> "K", "S", "D", "M", "M", "B", ~
## $ `Zone/Beat`                   <chr> "K3", "S3", "D2", "M1", "M3", ~
## $ `Census Tract`                <dbl> 9100.102, 11800.602, 7200.106,~
## $ Longitude                     <dbl> -122.3225, -122.2680, -122.342~
## $ Latitude                      <dbl> 47.59835, 47.51985, 47.61422, ~
## $ `Incident Location`           <chr> "(47.598347, -122.32245)", "(4~
## $ `Initial Type Description`    <chr> "THEFT (DOES NOT INCLUDE SHOPL~
## $ `Initial Type Subgroup`       <chr> "OTHER PROPERTY", "SHOPLIFTING~
## $ `Initial Type Group`          <chr> "THEFT", "THEFT", "TRESPASS", ~
## $ `At Scene Time`               <chr> "03/25/2016 10:25:51 PM", "03/~
```

lubridate

```
str(spd_raw$`Event Clearance Date`)
```

```
##  chr [1:706] "03/25/2016 11:58:30 PM" "03/25/2016 11:57:22 PM" ...
```

We want this to be in a date/time format ("POSIXct"), not character. We will work with dates using the `lubridate` package.

```
# install.packages("lubridate")
library(lubridate)
spd <- spd_raw %>%
  mutate(`Event Clearance Date` =
           mdy_hms(`Event Clearance Date`,
                   tz = "America/Los_Angeles"))
str(spd$`Event Clearance Date`)
```

```
##  POSIXct[1:706], format: "2016-03-25 23:58:30" "2016-03-25 23:57:22" ...
```

`mdy_hms()` processes datetimes in month-day-year, hour-minute-second format. It figures out separators for you!

An Aside on Time

Time data are a bit weird.

R uses two primary formats for storing data on times and dates:

- `POSIXct`: Numeric vector of seconds since the beginning of 1970.
- `POSIXlt`: Named list of vectors containing lots of date/time information.

We usually work with `POSIXct`.

`lubridate` gives us many convenience functions for dealing with date/time data.

It is often easiest to just convert time to standard numeric values and work with it that way, however, particularly if it will be used as a variable in a statistical model.

Useful Date/Time Functions

```
demo_dts <- spd$`Event Clearance Date`[1:2]
(date_only <- as.Date(demo_dts, tz = "America/Los_Angeles"))
```

```
## [1] "2016-03-25" "2016-03-25"
```

```
(day_of_week_only <- weekdays(demo_dts))
```

```
## [1] "Friday" "Friday"
```

```
(one_hour_later <- demo_dts + dhours(1))
```

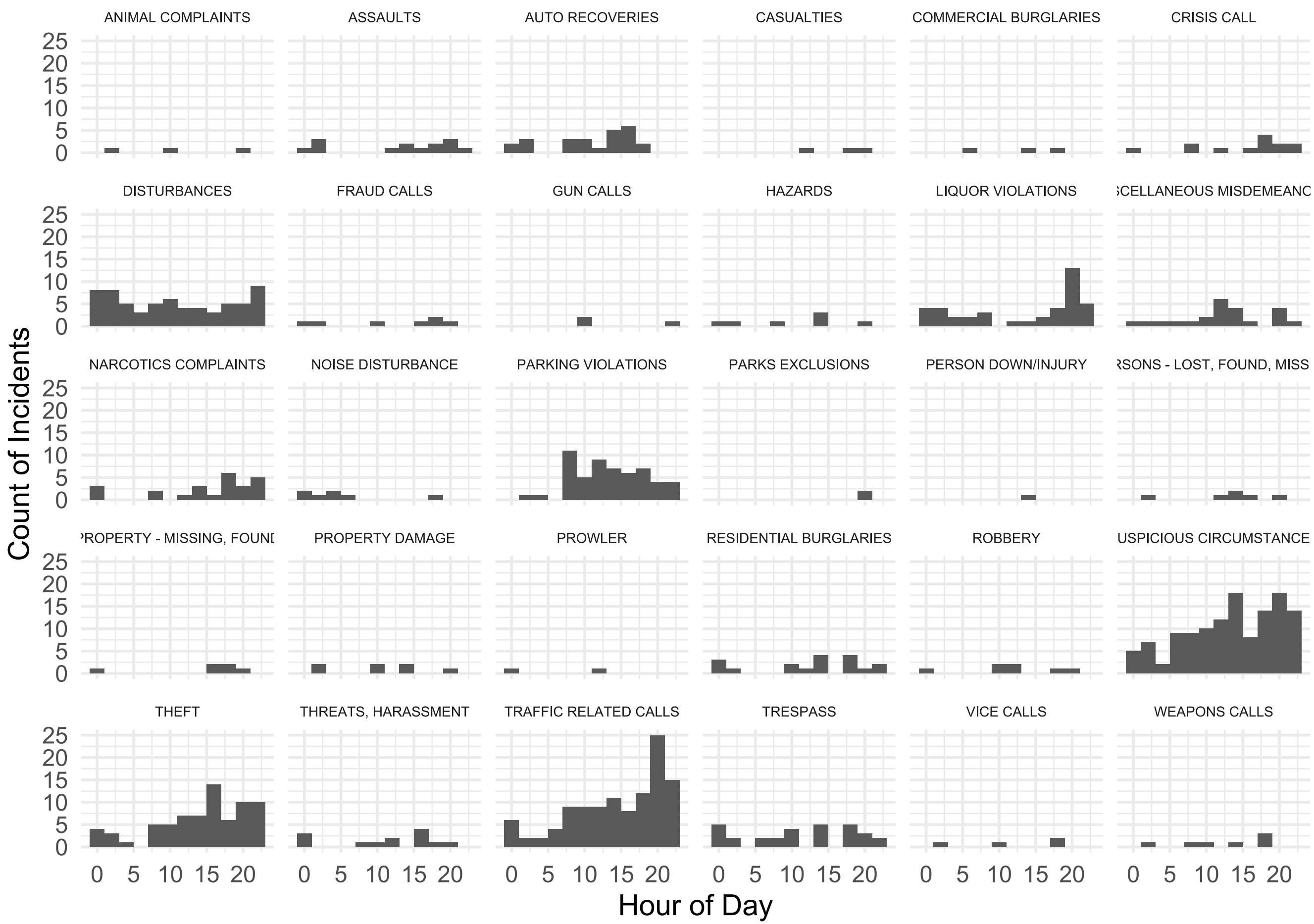
```
## [1] "2016-03-26 00:58:30 PDT" "2016-03-26 00:57:22 PDT"
```

What Time of Day were Incidents Cleared?

```
spd_times <- spd %>%
    select(`Initial Type Group`, `Event Clearance Date`) %>%
    mutate(hour = hour(`Event Clearance Date`))

time_spd_plot <- ggplot(spd_times, aes(x = hour)) +
    geom_histogram(binwidth = 2) +
    facet_wrap( ~ `Initial Type Group`) +
    theme_minimal() +
    theme(strip.text.x = element_text(size = rel(0.6))) +
    ylab("Count of Incidents") + xlab("Hour of Day")
```

SPD Event Clearances, March 25



Managing Factor Variables

Factor Variables

Factors are such a common (and fussy) vector type in R that we need to get to know them a little better when preparing data:

- The order of factor levels controls the order of categories in tables, on axes, in legends, and in facets in `ggplot2`.
    - Often we want to plot in interpretable/aesthetically pleasing order, e.g. from highest to lowest values—not **"Alabama first"**.
- The lowest level of a factor is treated as a reference for regression, and the other levels get their own coefficients.
    - Reference levels are by default alphabetical, which doesn't necessarily coincide with the easiest to understand baseline category.

`forcats`

The `tidyverse` family of packages includes the package `forcats` (an anagram of "factors") that is "for cat(egorical)s".

This package supersedes the functionality of the base factor functions with somewhat more logical and uniform syntax.

To find more, look at the `forcats` manual.

Character to Factor

```
# install.packages("forcats")
library(forcats)
str(spd_times$`Initial Type Group`)
```

```
##  chr [1:706] "THEFT" "THEFT" "TRESPASS" "CRISIS CALL" ...
```

```
spd_times$`Initial Type Group` <-
  factor(spd_times$`Initial Type Group`)
str(spd_times$`Initial Type Group`)
```

```
##  Factor w/ 30 levels "ANIMAL COMPLAINTS",..: 25 25 28 6 24 27 13 12 2 27 .
```

```
head(as.numeric(spd_times$`Initial Type Group`))
```

```
## [1] 25 25 28  6 24 27
```

Releveling by Frequency

`fct_infreq()` reorders the factor levels by the frequency they appear in the data.

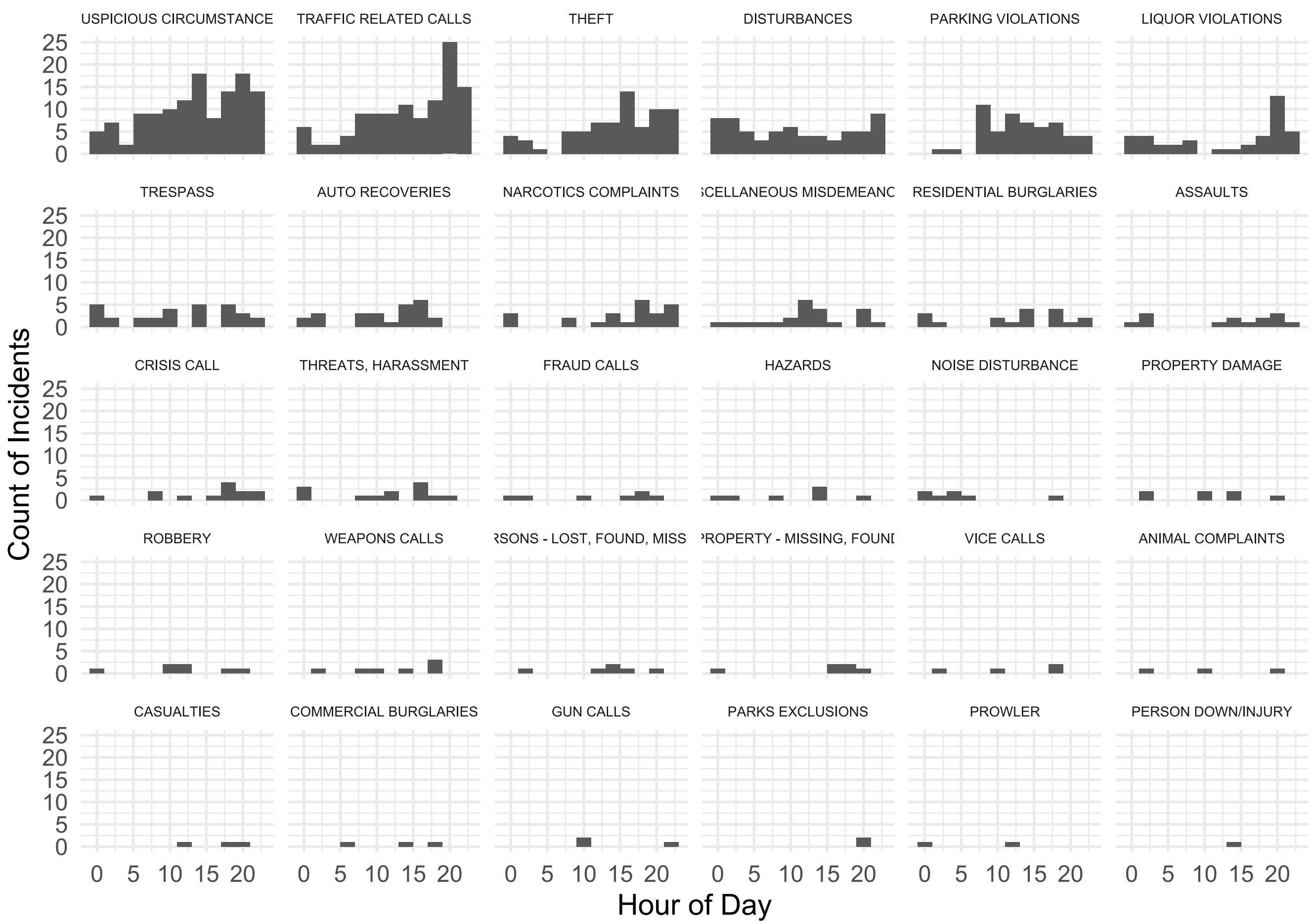
```
spd_times <- spd_times %>%
  mutate(`Initial Type Group` =
         fct_infreq(`Initial Type Group`))
head(levels(spd_times$`Initial Type Group`),4)
```

```
## [1] "SUSPICIOUS CIRCUMSTANCES" "TRAFFIC RELATED CALLS"
## [3] "THEFT"                    "DISTURBANCES"
```

Now the most common levels are first! Time to replot.

```
time_spd_plot_2 <- ggplot(spd_times, aes(x = hour)) +
  geom_histogram(binwidth = 2) +
  facet_wrap( ~ `Initial Type Group`) +
  theme_minimal() +
  theme(strip.text.x = element_text(size = rel(0.6))) +
  ylab("Count of Incidents") + xlab("Hour of Day")
```

Better Ordered Plot



Other Ways to Reorder

A general way to reorder a factor is through the `fct_reorder()` function:

```
fct_reorder(factor_vector,
            quantity_to_order_by,
            function_to_apply_to_quantities_by_factor)
```

This is especially useful for making legends go from highest to lowest value visually using `max()` as your function, or making axis labels go from lowest to highest value using `mean()`.

Use `fct_relevel()` and use the `ref=` argument to change the reference category

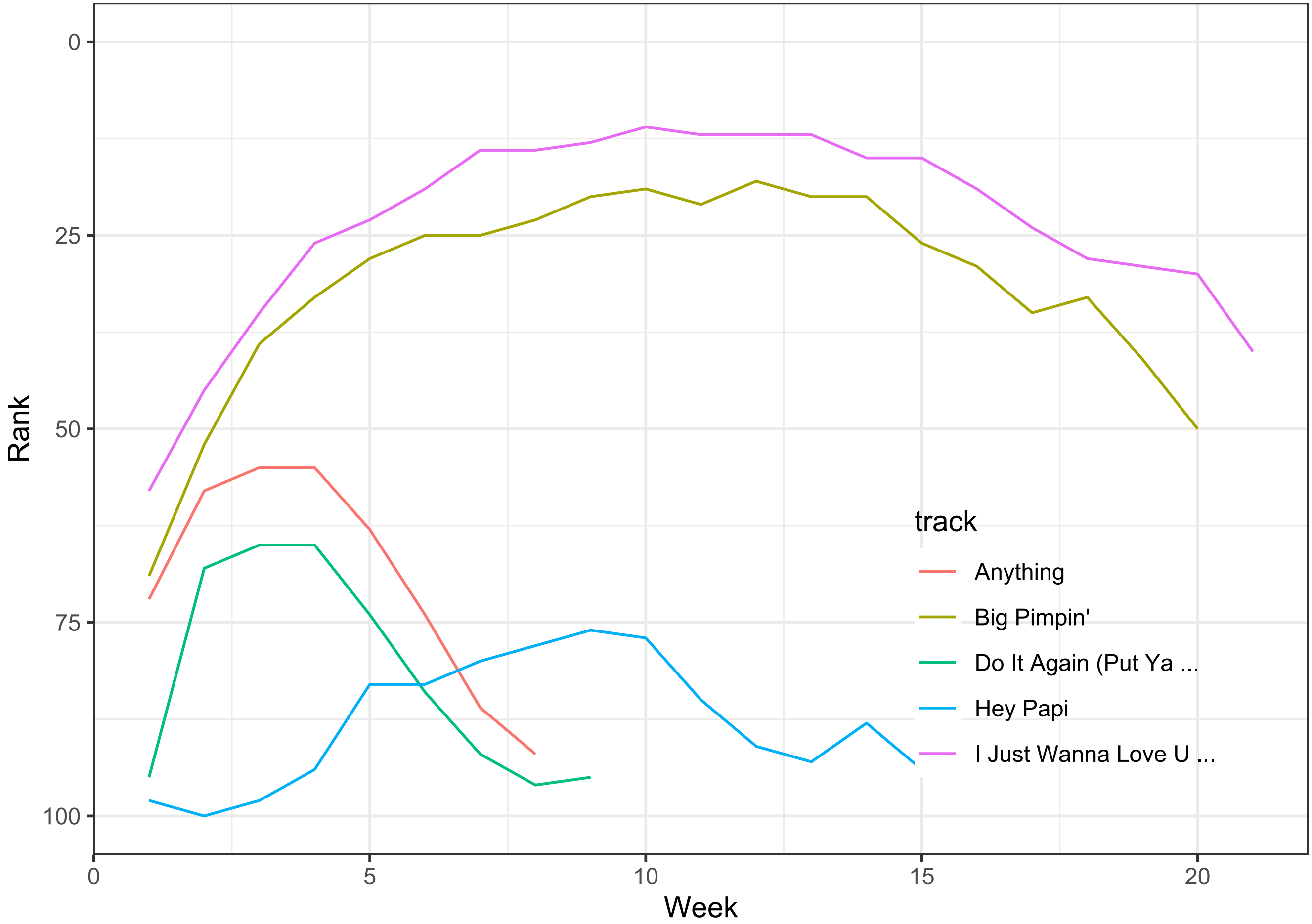
- Good when fitting regressions where you don't care about the overall ordering, just which level is the reference

Reorder Example: Jay-Z

```r
jayz <- billboard_2000 %>%
  filter(artist == "Jay-Z") %>%
  mutate(track = factor(track))

jayz_bad_legend <-
  ggplot(jayz, aes(x = week, y = rank,
                   group = track, color = track)) +
  geom_line() +
  theme_bw() +
  scale_y_reverse(limits = c(100, 0)) +
  theme(legend.position = c(0.80, 0.25),
        legend.background = element_rect(fill="transparent")) +
  xlab("Week") + ylab("Rank")
```

Jay-Z with Bad Legend Order



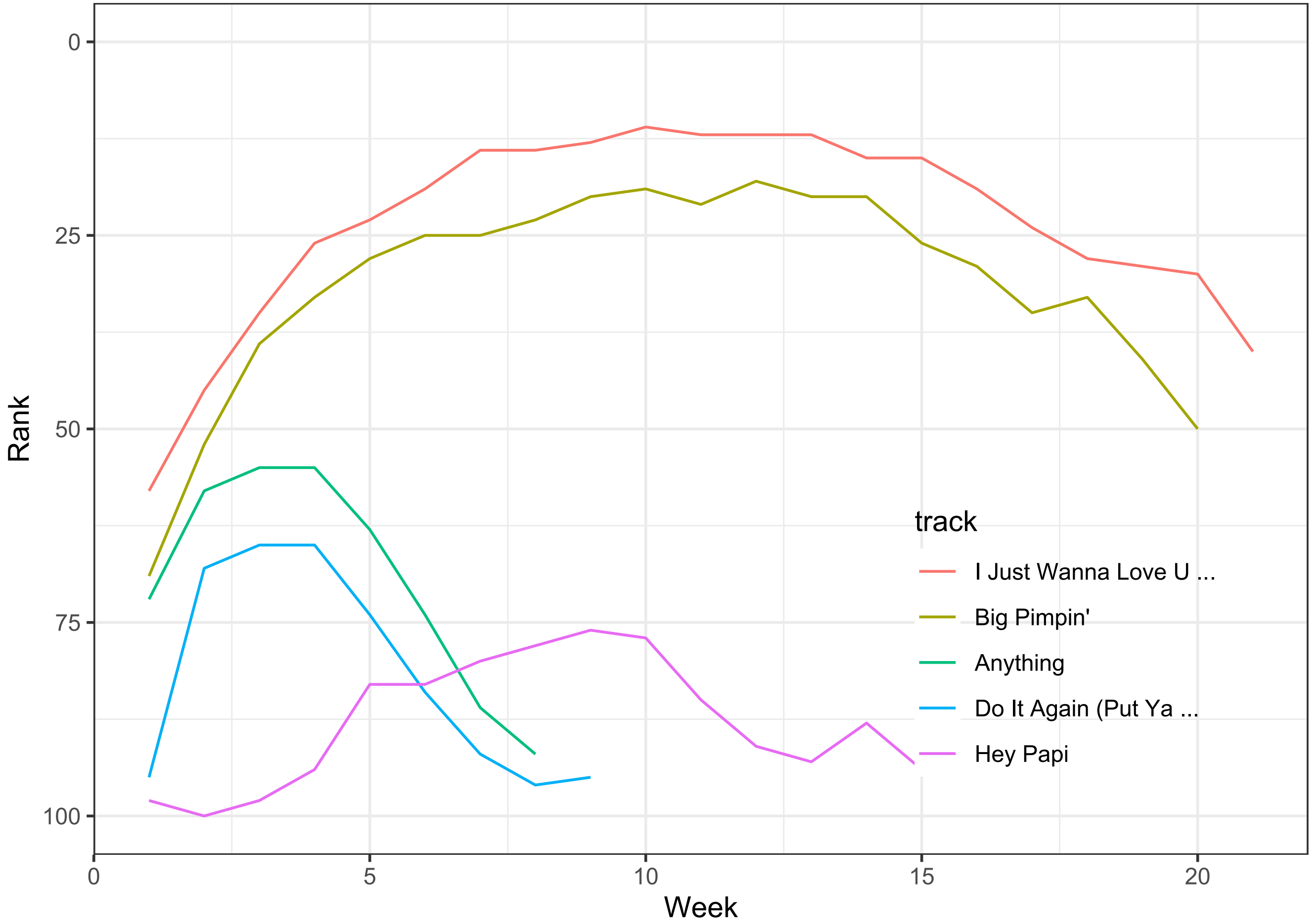
Better Ordering for Jay-Z

```
jayz <- jayz %>% mutate(track = fct_reorder(track, rank, min))

jayz_good_legend <-
  ggplot(jayz, aes(x = week, y = rank,
                   group = track, color = track)) +
  geom_line() +
  theme_bw() +
  scale_y_reverse(limits = c(100, 0)) +
  theme(legend.position = c(0.80, 0.25),
        legend.background = element_rect(fill="transparent")) +
  xlab("Week") + ylab("Rank")
```

This *reorders* `track` based on `rank`'s `min()` value.

Jay-Z with Good Legend Order



Dropping Unused Levels

After subsetting you can end up with fewer *realized* levels than before, but old levels remain linked and can cause problems for regressions. Drop unused levels from variables or your *entire data frame* using `droplevels()`.

```
jayz_biggest <- jayz %>%
  filter(track %in% c("I Just Wanna Love U ...", "Big Pimpin'"))
levels(jayz_biggest$track)
```

```
## [1] "I Just Wanna Love U ..." "Big Pimpin'"
## [3] "Anything"                "Do It Again (Put Ya ..."
## [5] "Hey Papi"
```

```
jayz_biggest <- jayz_biggest %>% droplevels(.)
levels(jayz_biggest$track)
```

```
## [1] "I Just Wanna Love U ..." "Big Pimpin'"
```

Homework

Vote tallies in King County from the 2016 general election are in a 60 MB comma-delimited text file downloaded from King County. They can be found on the course website.

The data have no documentation (aside from what I provide), so show your detective work to answer questions about the data and clean them up in the R Markdown template on the course website. Use *⌘-Click* on Mac or *Right-Click* on Windows to download the .Rmd to the folder you plan to work from, then open it in RStudio.

This homework is two parts to be completed in each of the next two weeks. It can be daunting, so do not wait until Monday to start. I recommend reading instructions closely, working with others, and using the mailing list and Slack.

PART 1 DUE: Next Week

PART 2 DUE: In Two Weeks