Programming Camp Day 3

Augustus Kmetz

2025-09-12

Welcome back!

Plans for today:

- Tidy data
- Review
- Relational Data
- Regressions in R

Tidy Data

What is tidy data?

"All happy families are alike; each unhappy family is unhappy in its own way."

— Leo Tolstoy

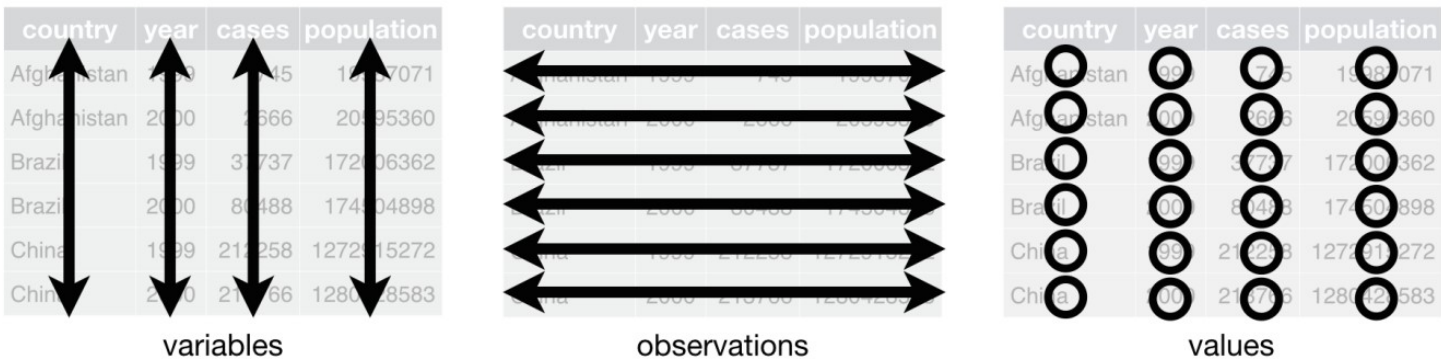
"Tidy dataset are all alike, but every messy dataset is messy in its own way."

— Hadley Wickham

What is tidy data?

There are three interrelated rules which make a data set tidy

- each variable must have its own column
- each observation must have its own row
- each value must have its own cell



Datasets in different forms

Look at the built in data sets `table1`, `table2`, `table3`, `table4a`, `table4b`, and `table5`

Each data set shows the same four variables:

- country
- year
- population
- number of TB cases

But each data set organizes the values in a different way

Q: Which one is tidy?

Datasets in different forms

Look at the built in data sets `table1`, `table2`, `table3`, `table4a`, `table4b`, and `table5`

Each data set shows the same four variables:

- country
- year
- population
- number of TB cases

But each data set organizes the values in a different way

Q: Which one is tidy?

Answer: `table1` is tidy.

Why tidy data?

- Working with tidy data we can use the same tools in similar ways for different data sets
- working with untidy data often means reinventing the wheel
- R is naturally vectorized. Most built-in R functions work with vectors of values

Why does untidy data exist?

The principles of tidy data seem so obvious

But

- Most people aren't familiar with the principles of tidy data, and it's hard to derive them yourself unless you spend a lot of time working with data
- Data is often organised to facilitate some use other than analysis. For example, data is often organised to make entry as easy as possible.

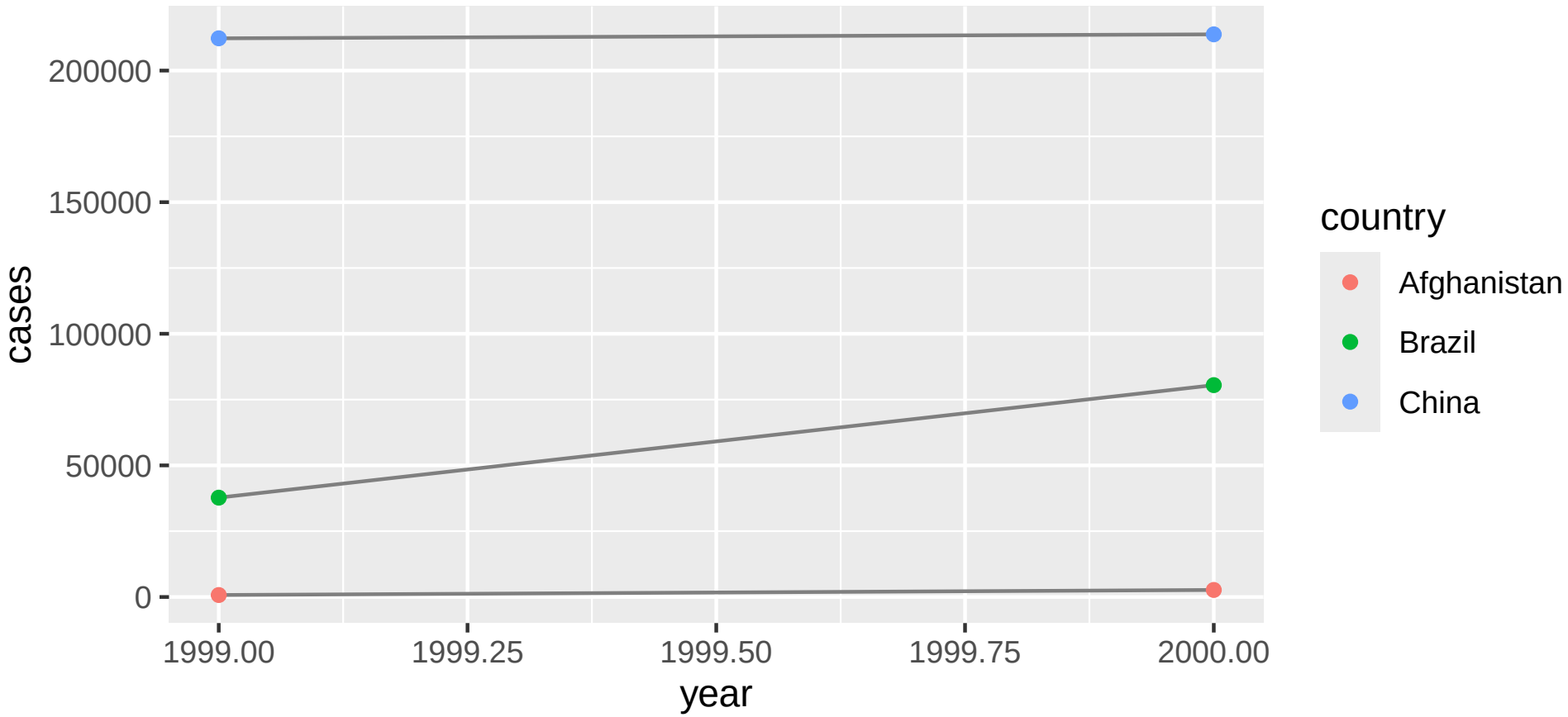
Dealing with untidy data

First Step: determine what are the variables and what are the observations

Second Step: deal with the following common issues

- one variable is spread across multiple columns -> need to `pivot_longer()`
- one observation scattered across multiple rows -> need to `pivot_wider()`
- one column contains values for multiple variables -> need to `separate()`
- multiple columns store information on a single variable -> need to `unite()`

Dealing with untidy data



Recreate the plot showing change in cases over time using `table2` instead of `table1`. What do you need to do first?

pivot_longer()

Common problem: some column names names are not the names, but the values of a variable

`pivot_longer()` makes data sets longer by increasing the number of rows and decreasing the number of columns

pivot_longer()

To tidy up `table4a` we need to `pivot_longer()` those columns into a new pair of variables. We need three pieces of information:

- The set of columns that represent values, not variables (using `select()` style notation)
- The name of the variable to move the column names to: `year`
- The name of the variable to move the column values to: `cases`

```
table4a %>%
  pivot_longer(c(`1999`, `2000`),
  names_to = "year", values_to = "cases")
```

Q: How can we fix up `table4b`?

```
table4b %>%
  pivot_longer(c(`1999`, `2000`),
  names_to = "year", values_to = "population")
```

pivot_longer()

To tidy up `table4a` we need to `pivot_longer()` those columns into a new pair of variables. We need three pieces of information:

- The set of columns that represent values, not variables (using `select()` style notation)
- The name of the variable to move the column names to: `year`
- The name of the variable to move the column values to: `cases`

```
table4a %>%
  pivot_longer(c(`1999`, `2000`),
  names_to = "year", values_to = "cases")
```

Q: How can we fix up `table4b`?

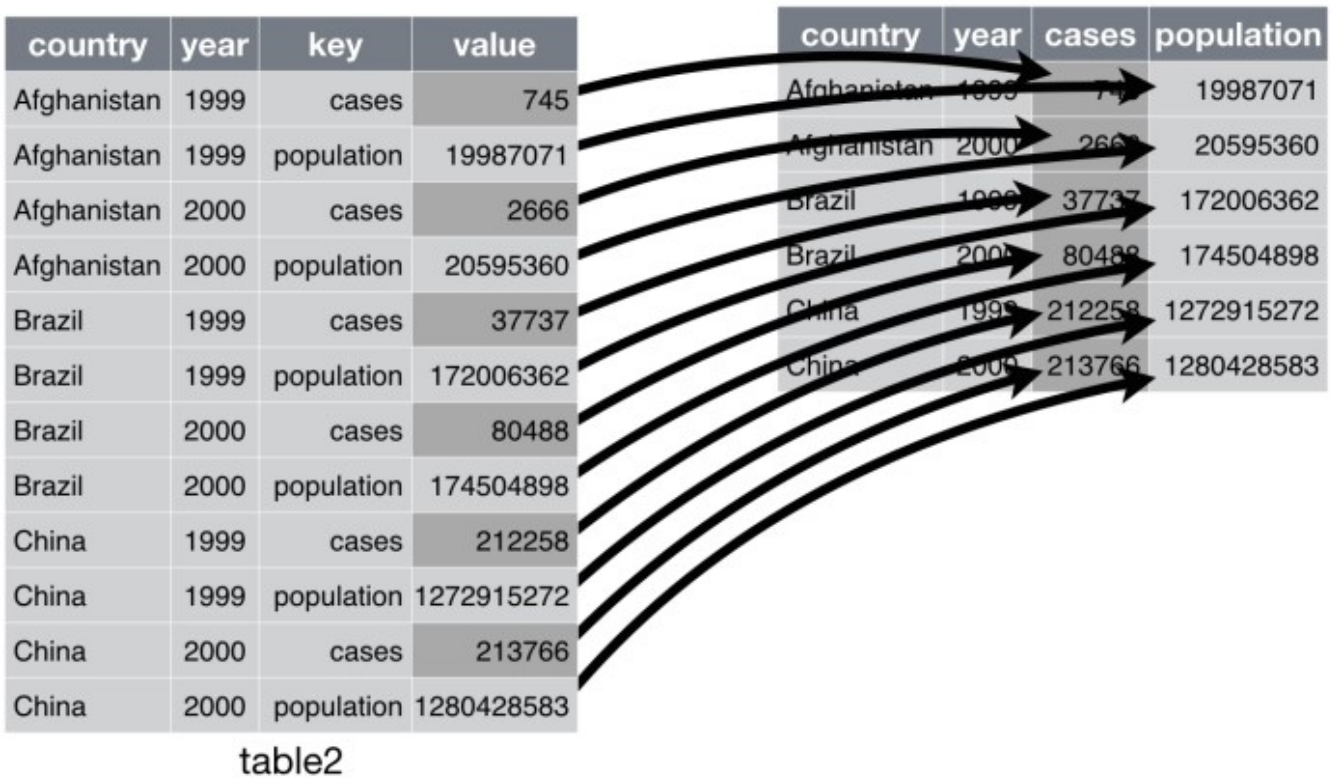
```
table4b %>%
  pivot_longer(c(`1999`, `2000`),
  names_to = "year", values_to = "population")
```

Answer:

pivot_wider()

Common problem: some observations are spread across multiple rows

`pivot_wider()` is the opposite of `pivot_longer()`



country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

table2

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

Spreading

To tidy up `table2` we need two pieces of information:

- The column to take variable names from: `type`
- The column to take values from: `count`

```
table2 %>%
    pivot_wider(names_from = type, values_from = count)
```

Your turn!

Why are `pivot_longer()` and `pivot_wider()` not perfectly symmetrical?

Carefully consider the following example:

```
stocks <- tibble(
  year   = c(2015, 2015, 2016, 2016),
  half   = c(   1,    2,    1,    2),
  return = c(1.88, 0.59, 0.92, 0.17)
)

stocks %>%
  pivot_wider(names_from = year,
              values_from = return) %>%
  pivot_longer(`2015`:`2016`,
               names_to = "year",
               values_to = "return")
```

(Hint: look at the variable types and think about column names.)

Your turn!

Question: Why does this code fail?

```
table4a %>%
  pivot_longer(c(1999, 2000),
               names_to = "year",
               values_to = "cases")
```

Your turn!

Question: Why does this code fail?

```
table4a %>%
  pivot_longer(c(1999, 2000),
               names_to = "year",
               values_to = "cases")
```

Answer: The column names are strings, but `pivot_longer()` expects them to be numbers because of how the list is specified.

Separate

Common problem: some data sets have columns with values corresponding to multiple variables

`separate()` makes tables wider (`unite()` is the opposite of separate)

country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

table3

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

separate()

`separate()` splits one column into multiple columns whenever a separator appears

```
table3 %>%
  separate(col = rate, into = c("cases", "population"))
```

- you can specify the separator character `sep = "/"`
- by default leaves the column type as is, you can ask it to convert `convert = T`

```
table3 %>%
  separate(rate, into = c("cases", "population"),
           sep = "/", convert = T)
```

unite()

`unite()` is very similar to `sperate()` it combines multiple columns into a single column

```
unite(table5, col = full_year, century, year)

unite(table5, col = full_year, century, year, sep = "")
```

Review

nycflights13

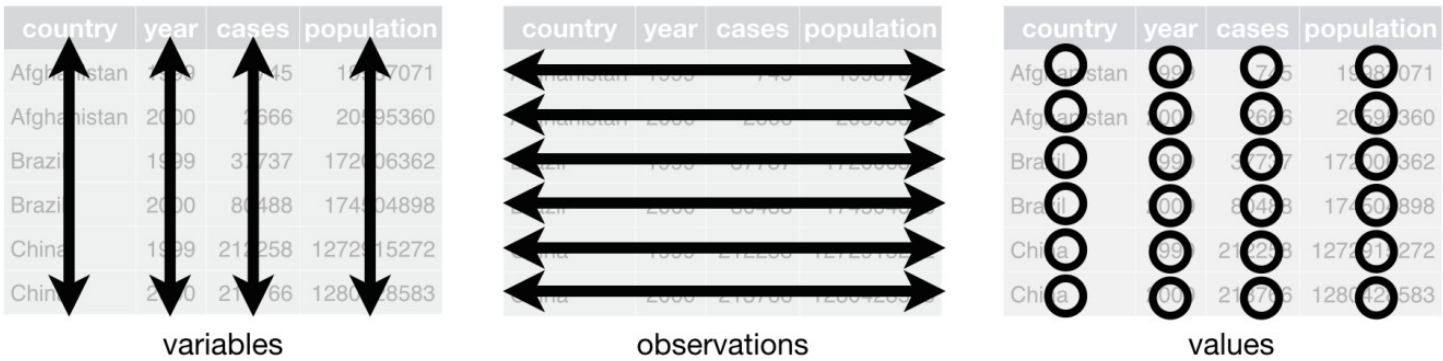
We will look at the data set `flights`

- 336,776 flights that departed from New York City in 2013
- Data from the US Bureau of Transportation statistics

Please load the `nycflights13` package

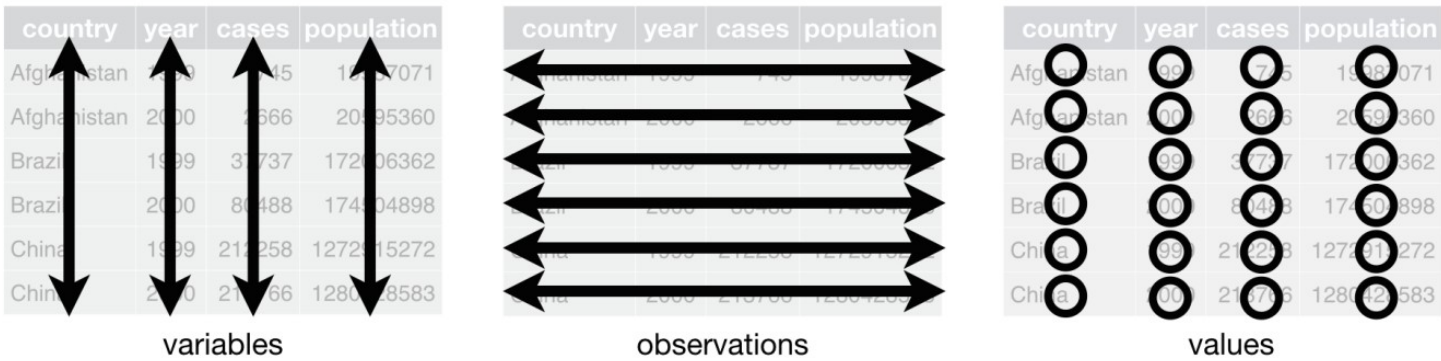
Take a look at the `flights` dataset

Tidy data



Q: Is `flights` a tidy data set?

Tidy data



Q: Is `flights` a tidy data set?

Answer: Yes, because each variable is in a column and each observation is in a row.

`dplyr` verbs (functions)

`dplyr` verbs handle the vast majority of your data needs:

- `filter()` - for picking observations by their values
- `select()` - for picking variables by their names
- `arrange()` - for reordering rows
- `mutate()` - for creating new variables with functions on existing variables
- `summarise()` - for collapsing many values down to a single summary

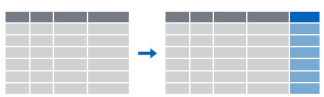
dplyr recap

Extract variables with **select()**

Extract cases with **filter()**

Arrange cases, with **arrange()**.

Make tables of summaries with **summarise()**.



Make new variables, with **mutate()**.

The structure of `dplyr` functions

All verbs work similarly:

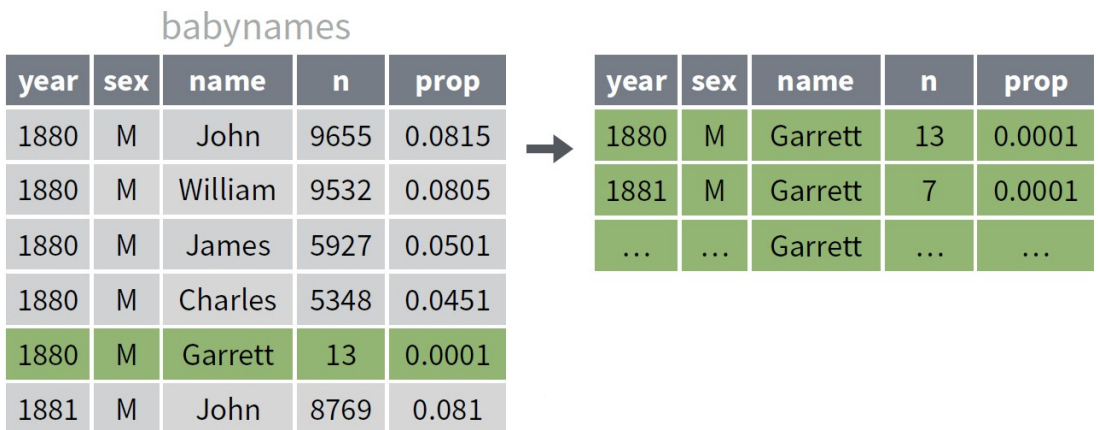
- The first argument is a tibble (or data frame)
- The subsequent ones describe what to do, using the variable names
- The results is a new tibble

filter()

Extract rows that meet logical criteria

```
filter(.data, condition_1, condition_2, ...)
```

```
filter(babynames, name == "Garrett")
```



Logical tests

`x < y`	Less than
`x > y`	Greater than
`x == y`	Equal to
`x <= y`	Less than or equal to
`x >= y`	Greater than or equal to
`x != y`	Not equal to
`x %in% y`	Group membership
`is.na(x)`	Is NA
`!is.na(x)`	Is not NA

Same syntax works for `Inf` and `NaN`: `is.inf()`, `is.nan()`

Boolean operators

`a & b`	and
`a | b`	or
`xor(a,b)`	exactly or
`!a`	not
`( )`	To group tests . & evaluates before |

The pipe operator %>%

Passes results on left into first argument of function on right

In pipe notation you use:

- `x %>% f(y)` rather than `f(x, y)`
- `x %>% f(y) %>% h(z)` rather than `h(f(x,y),z)`

This is similar to Unix pipes

Your turn!

Find all flights that:

1. Had an arrival delay of two or more hours
2. Flew to Houston (IAH or HOU)
3. Arrived more than two hours late, but didn't leave late

Explore a bit:

1. What does a cancelled flight look like?
2. Make a data set called `not_cancelled` that contains all non-cancelled flights
3. Make a data set called `cancelled` that contains all cancelled_flights

arrange()

Order rows from smallest to largest values

```
arrange(.data, order_first, order_second, ...)
```

```
arrange(babynames, n)
```

order rows from largest to smallest values

```
arrange(babynames, desc(n))
```

Your turn!

Use `arrange()` to:

1. Sort flights to find the most delayed flights.
2. Find the flights that left earliest
3. Sort flights to find the fastest (highest speed) flights

summarise()

compute table of summaries

```
summarise(.data,
          summary_stat_1 = function_1(variable_1),
          summary_stat_2 = function_2(variable_2))
```

```
babynames %>%
  summarise(total = sum(n), max = max(n))
```

group_by()

Groups cases by common values

```
group_by(.data, first_variable, second_variable, ...)
```

```
babynames %>%
  group_by(sex) %>%
  summarise(total = sum(n))
```

`ungroup()` - removes grouping criteria from a data frame

Elementary but useful summary functions

Aggregation functions return one value per group

- `min(x)`, `median(x)`, `max(x)`, `quantile(x, p)`
- `n()`, `n_distinct()`, `sum(x)`, `mean(x)`
- `sum(x > 10)`, `mean(x > 0)`
- `sd(x)`, `var(x)`

Window functions return multiple values per group

- `top_n()`
- `lead()`, `lag()`

mutate()

Create new columns

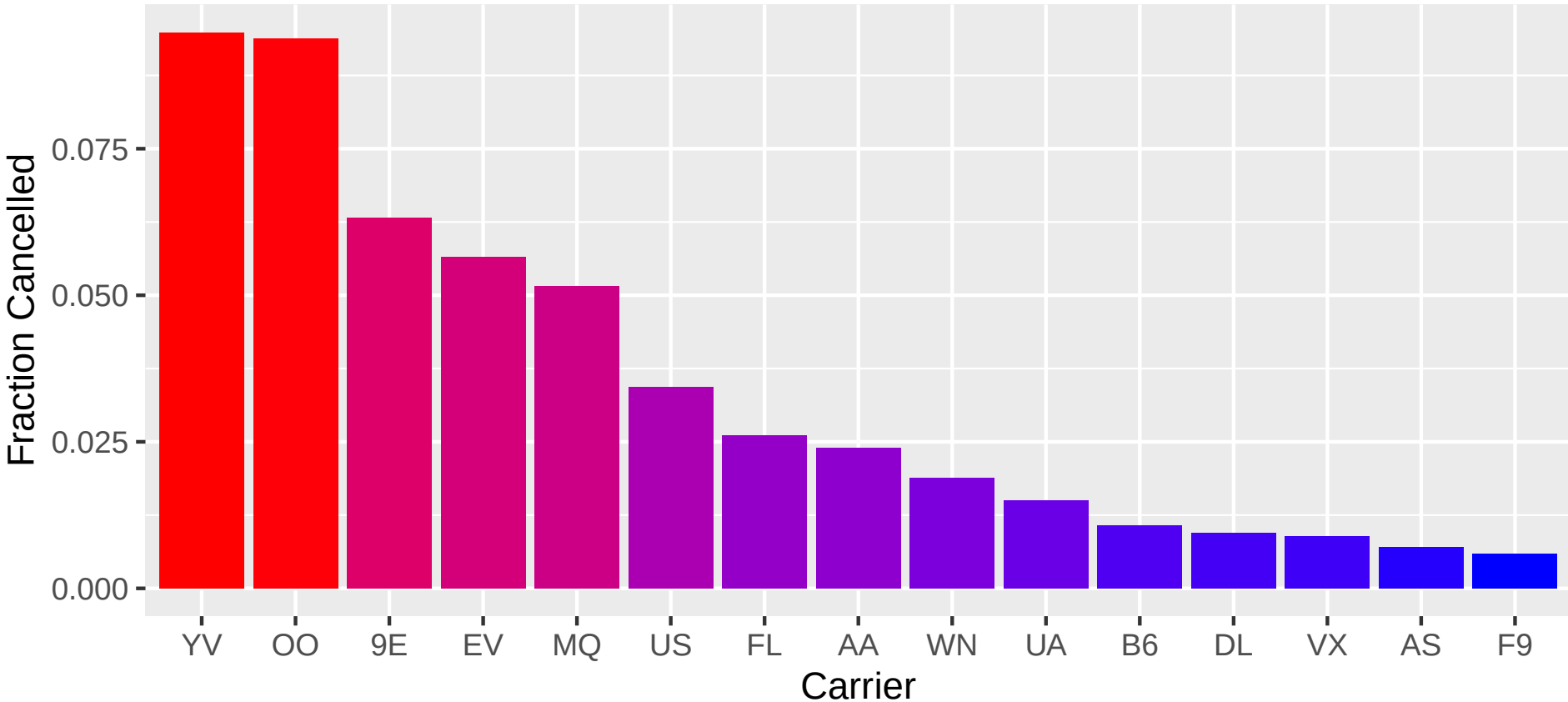
```
mutate(.data,
       new_var_1 = function_1(variable_1),
       new_var_2 = function_2(variable_2))
```

Any vectorized function can be used with `mutate()` including:

- arithmetic operators (+, -, /, *, etc)
- logical operators (<, <=, >=, ==, !=)
- logarithmic and exponential transformations (`log`, `log10`, `exp`)
- offsets (`lead`, `lag`)
- cumulative rolling aggregates (`cumsum`, `cumprod`, `cummin`, `cummax`)
- ranking (`min_rank`, `percent_rank`)

Your turn!

Produce the following chart. What does the underlying data look like?



But these two digit abbreviations are not useful!

Relational Data

Relational Data

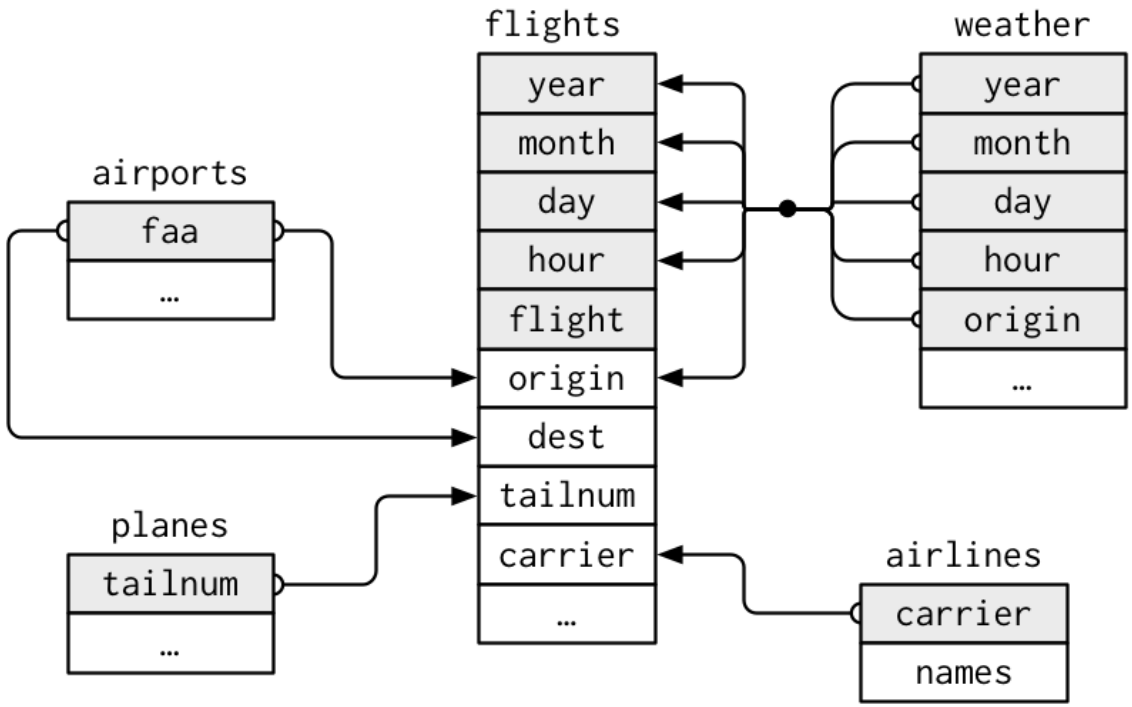
- It's rare that an analysis involves only a single table of data
- Typically you have many tables of data that are related through variables/observations
- To work with relational data you need verbs that work with pairs of tables
- There are two families of verbs designed to work with relational data:
    1. Mutating joins - add new variables to one data frame from matching observations in another
    2. Filtering joins - filter observations from one data frame based on whether or not they match an observation in the other table

nycflights13

There are several tables that are helpful for our analysis all from the `nycflights13` package

- `flights`
- `airline` - let's you look up the full carrier name from its abbreviated code
- `airports` - gives information about each airport identified by the faa airport code
- `planes` - gives information about each plane identified by its `tailnum`
- `weather` - gives the weather at each NYC airport for each hour

nycflights13



nycflights13

In words:

- `flights` connects to `planes` via a single variable `tailnum`
- `flights` connects to `airlines` through the `carrier` variable
- `flights` connects to `airports` in two ways: via the `origin` and `dest` variables
- `flights` connects to `weather` via location (`origin`) and time (`year`, `month`, `day`, `hour`)

Taxonomy of Joins

Four main types of joins (there are more exotic ones)

1. `left_join(x, y)` - keeps all observations in `x`
2. `right_join(x,y)` - keeps all observations in `y`
3. `full_join(x,y)` - keeps all observations in `x` and `y`
4. `inner_join(x,y)` - keeps only observations in `x` and `y`

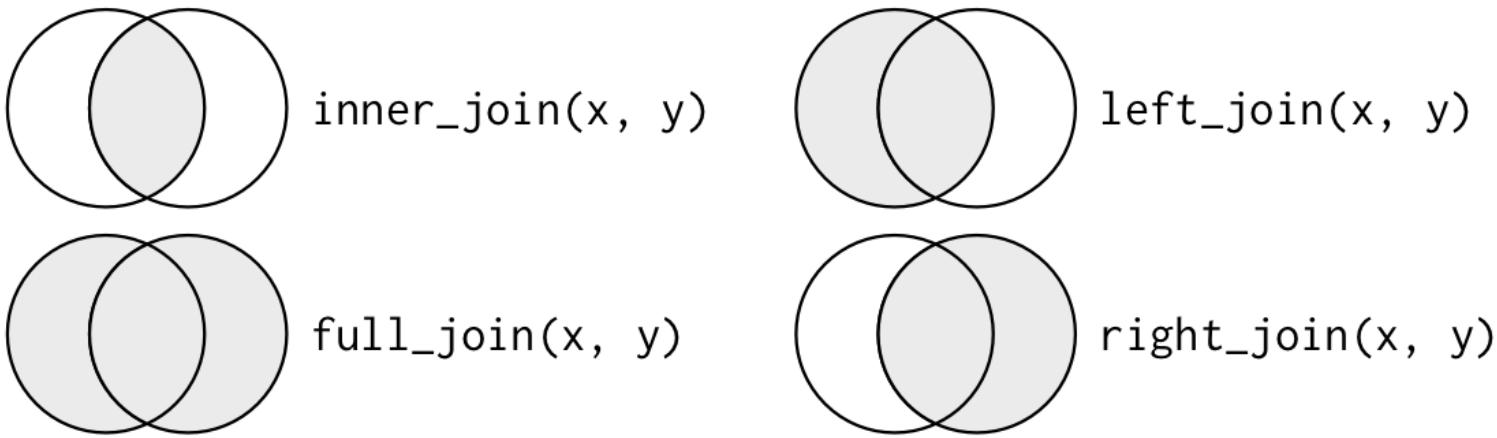
x

1	x1
2	x2
3	x3

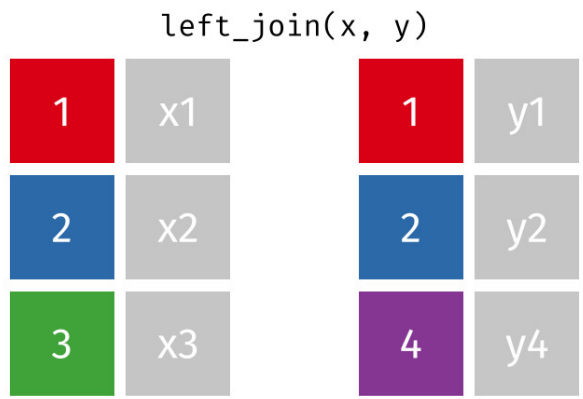
y

1	y1
2	y2
4	y3

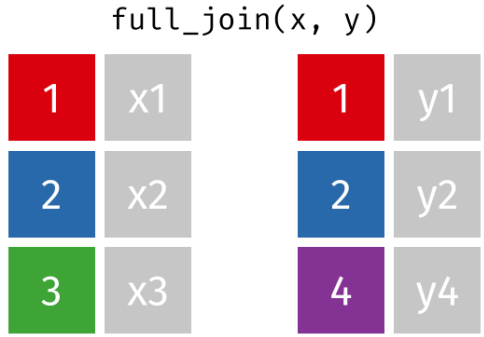
Taxonomy of Joins



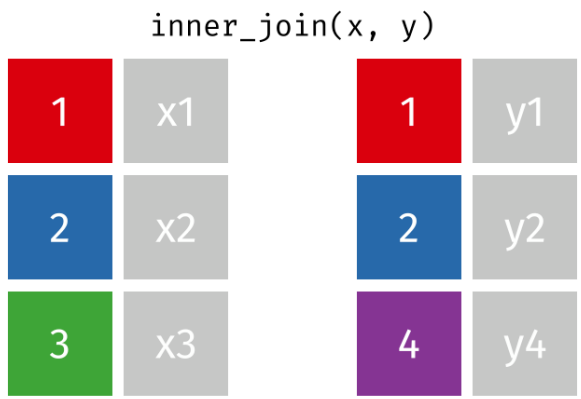
left_join



full_join

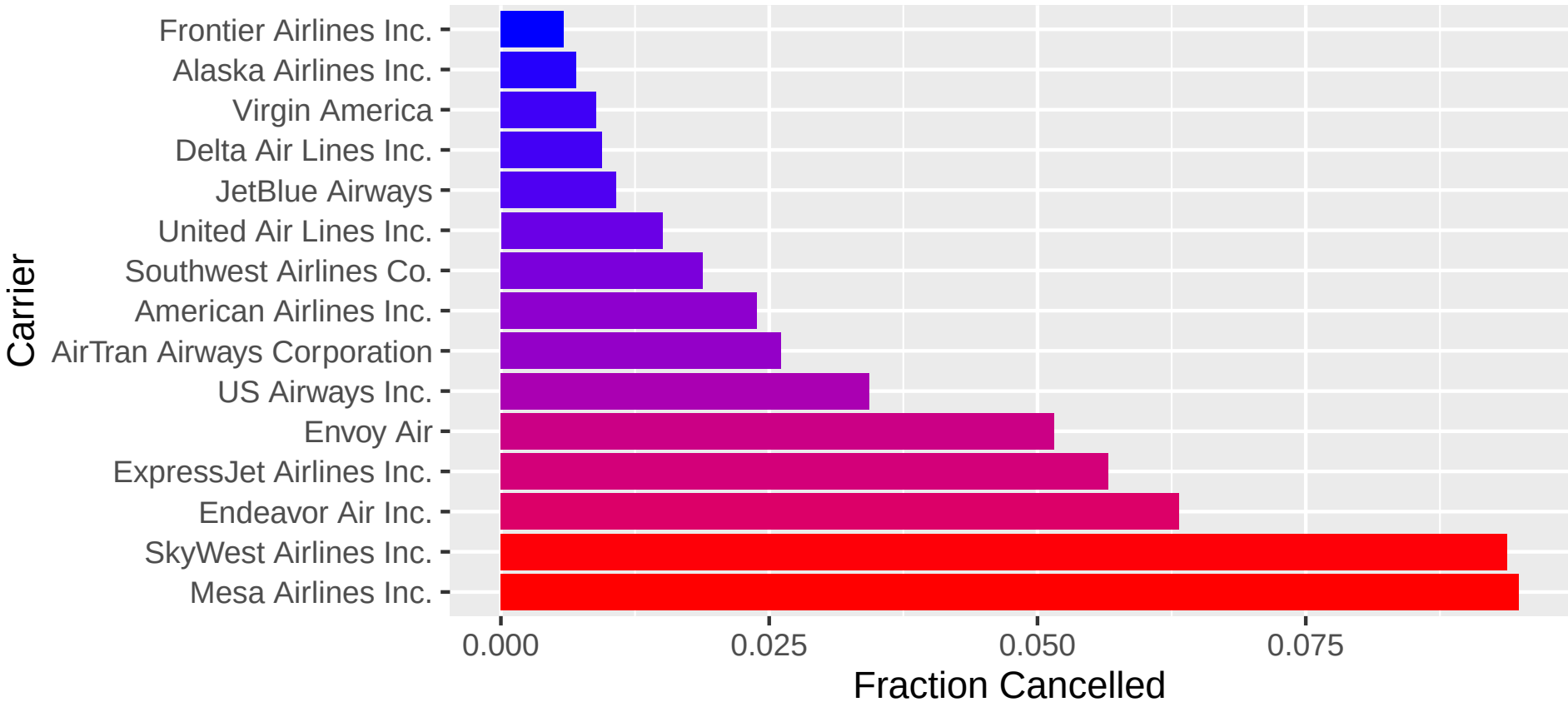


inner_join



Your turn!

Produce the following chart. What does the underlying data look like?



Your turn!

```
flights %>%
  left_join(airlines, by = "carrier") %>%
  group_by(name) %>%
  summarise(frac_can= sum(is.na(arr_delay)) / n()) %>%
  filter(frac_can > 0) %>%
  ggplot() +
  geom_col(aes(y = fct_reorder(name, frac_can, .desc = TRUE),
  scale_fill_gradient(low = "blue", high = "red") +
  labs(x = "Fraction Cancelled",
       y = "Carrier") +
  guides(fill = "none")
```

Regressions

How does weather impact departure delays?

To make this analysis simpler let's compress our data to daily data

1. Make a data set called `nyc_flights` that contains for each day the mean and max departure delay by carrier/origin airport as well as the fraction of all flights delayed. Filter to carrier/day/airports that have at least 10 flights
2. Make a data set called `nyc_weather` that contains for each day and origin airport, the mean, max, and min temp and wind speed and the total precipitation

Then join these two data sets together into a data set called `nyc_weather_delays`

Data setup

```
nyc_flights <-
  flights %>%
  group_by(carrier, year, month, day, origin) %>%
  summarise(dep_delay = mean(dep_delay, na.rm = T),
            max_dep_delay = max(dep_delay, na.rm = T),
            frac_delayed = sum(dep_delay > 0, na.rm = T) / n(),
            frac_cancelled = sum(is.na(dep_time )) / n(),
            n = n()) %>%
  filter(n >= 10)
```

Data setup

```
nyc_weather <-
  weather %>%
  group_by(year, month, day, origin) %>%
  summarise(temp     = mean(temp, na.rm = TRUE),
            min_temp = min(temp, na.rm = TRUE),
            max_temp = max(temp, na.rm = TRUE),
            wind     = mean(wind_speed, na.rm = TRUE),
            max_wind = max(wind_speed, na.rm = TRUE),
            precip = sum(precip, na.rm = TRUE))

nyc_weather_delays <-
  nyc_flights %>%
  inner_join(nyc_weather, by = c("year", "month", "day", "orig
```

OLS estimation with `lm()`

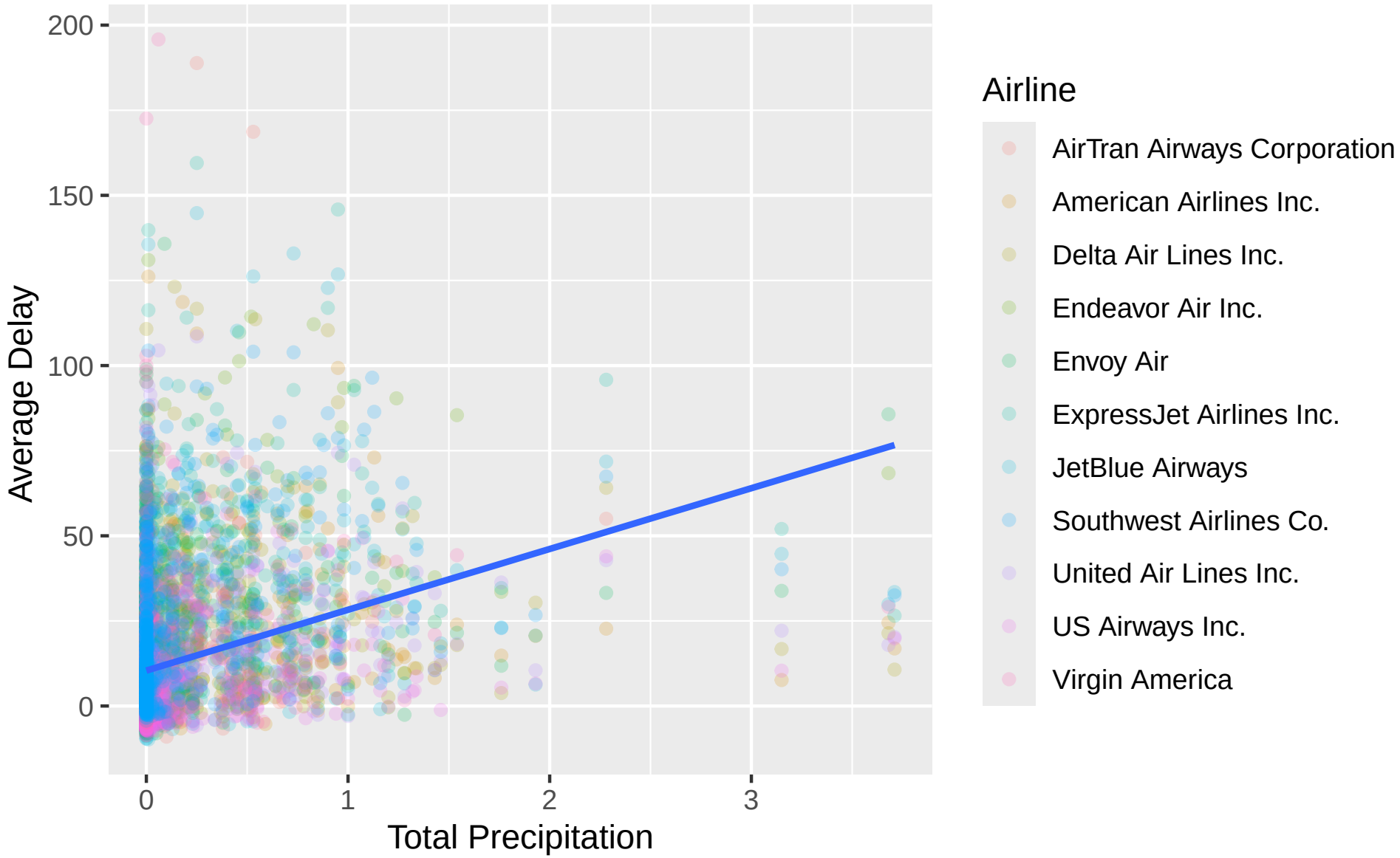
We can run OLS with lm() (linear models):

```
lm(y ~ x1 + x2 + ..., data)
lm(y ~ x1 * x2 + ..., data)
lm(y ~ x1:x2 + ..., data)
```

```
weather_delay_model <-
  lm(dep_delay ~ temp + precip + wind,
     nyc_weather_delays)


summary(weather_delay_model)
```

Adding regression lines to charts



Adding regression lines to charts

```
nyc_weather_delays %>%
  left_join(airlines, by = "carrier") %>%
  ggplot(aes(y = dep_delay, x = precip)) +
  geom_point(aes(color = name), alpha = 0.2) +
  geom_smooth(method = "lm", se = F) +
  labs(x = "Total Precipitation",
       y = "Average Delay ",
       color = "Airline")
```

Adding regression lines for each airline

```
nyc_weather_delays %>%
  left_join(airlines, by = "carrier") %>%
  ggplot(aes(y = dep_delay, x = precip)) +
  geom_point(aes(color = name), alpha = 0.2) +
  geom_smooth(aes(color = name), method = "lm", se = F) +
  labs(x = "Total Precipitation",
       y = "Average Delay ",
       color = "Airline")
```

Robust standard errors

`sandwich` and `coeftest()`

- sandwich package handles error correction behind the scenes
- `vcovHC()` - heteroskedasticity-consistent
- `vcovHAC()` - heteroskedasticity and autocorrelation consistent
- `vcovCL()` - clustered errors
- etc.
- If you want to replicate Stata's `robust` option you can use `vcovHC(type = HC1)`

```
coeftest(weather_delay_model,
         vcov = vcovHC,
         type = "HC1")
```

Lots of other packages: `fixest`, `estimatr`, `modelsummary` etc.

Logit & Probit

```
logit_data <-
  nyc_weather_delays %>%
  mutate(any_cancelled = frac_cancelled > 0)

logit_results <- glm(any_cancelled ~ wind + carrier,
                     data = logit_data,
                     family = binomial)

probit_results <- glm(any_cancelled ~ wind + carrier,
                      data = logit_data,
                      family = binomial(link = "probit" ))

summary(logit_results)
```

Fixed effects

- The package `lfe` implements models with high dimensional fixed effects or/and instrumental variables
- The package `gmm` implements GMM
- The package `rdd` implements regression discontinuity methods
- The package `matchit` implements matching procedures

Fixed effects

```
felm(y ~ x1 + x2 | f1 + f2, data)
felm(y ~ x1 + x2 | f1 + f2, data, robsust = T)
felm(y ~ x1 + x2 |
          fe1 + fe2 |
          (Q|W ~ (x3 + x4)) |
          clu1 + clu2, data)
felm(y ~ x1 + x2 | fe1 + fe2 | 0 | clu1 + clu2, data)
```

Where:

- `y` is the response
- `x1`, `x2` are ordinary covariates
- `f1`, `f2` are factors to be projected out
- `Q` and `W` are covariates which are instrumented by `x3` and `x4`
- `clu1`, `clu2` are factors to be used for computing cluster robust standard errors

Fixed effects

```
weather_delay_model_fe <-
  felm(dep_delay ~ temp + precip + wind + max_wind |
                   origin + carrier, nyc_weather_delays)

summary(weather_delay_model_fe, robust = T)
```

Your turn!

Come up with a “better” model of flight delays/cancellations